Using Texture Memory to Generate Anti-aliased Image Textures on PixelFlow

Yulan Wang

Department of Computer Science
University of North Carolina at Chapel Hill
June 1993

Abstract PixelFlow is an experimental parallel graphics computer under design at UNC. This document discusses issues concerned with using PixelFlow's texture memory to generate anti-aliased image textures. These include: perspective interpolation of texture coordinates and texture gradients, mip-map memory organization and address generation, schemes of loading textures into texture memory, and parallel algorithms for generating mip-maps of textures through pre-filtering. Finally, a texture memory manager based on the buddy system is outlined.

CONTENT

1 Introduction

1.1 Background

1.2 About this Document

2 Perspective Interpolation of Texture Coordinates and Gradients

2.1 Perspective Interpolation

2.2 Precision Requirements

2.3 Polygons with Large Depth Variations

3 Mip-Map Memory Organization and Address Generation

3.1 Mip-Map Memory Organization

3.2 Mip-Map Memory Address Generation

3.3 Mip-Map Level Index Computation

4 Texture Loading and Texture Pre-filtering

4.1 Color Encoding

4.2 Texture Loading

4.3 Texture Pre-filtering

5 Texture Memory Management

5.1 A Two Dimensional Buddy System

5.2 Rectangle Textures

5.3 Flat Textures

6 Conclusion

1 Introduction

1.1 Background

Successes of Pixel-Plane5 [Fuchs89] promoted graphics applications such as flight and vehicle simulation, computer-aided design, scientific visualization, and virtual reality. On the other hand, attempts to achieve higher performance and better photo-realism motivated the design of the new graphics computer architecture of PixelFlow [Molnar92].

PixelFlow uses the technique of image composition: it distributes the rendering task over an array of identical renderers, each of which computes a full-screen image of a fraction of the primitives. A high-performance image-composition network composites these images in real time to produce an image of the entire scene.[1]

PixelFlow supports deferred shading with shaders that operate on composited images containing intermediate pixel data. PixelFlow shaders compute complex shading algorithms and procedural textures, as well as image textures. It is the texture memory that makes real time anti-aliased image textures possible.

Figure 1 shows the block diagram of a PixelFlow system. The backplane contains the image composition network. The card cages can be populated with any configuration of renderer boards, shader boards, or framebuffer boards. Each board has a high speed link connecting it to a host computer.

[1] Pixel-Plane5 uses a screen-subdivision approach, in which the screen is divided into disjoint regions. The processors simultaneously rasterize primitives that fall into the different regions. These sub-images are concatenated to form a full-screen image.

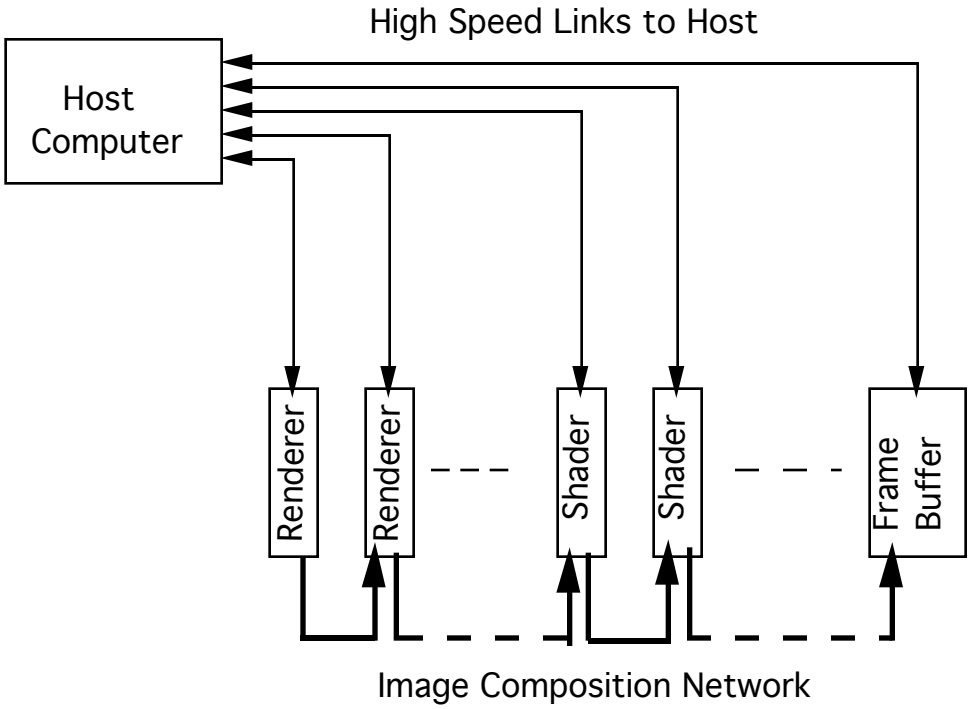

Figure 1: Block Diagram of PixelFlow System

A shader, shown in figure 2, contains a graphics processor (GP), a SIMD pixel processor array, and 2 x 2K x 1K 32 bits of texture memory. (A renderer or a framebuffer is similar to a shader in architecture. They are irrelevant to this document because texture mapping is done on shaders.) The pixel processor array is composed of logic-enhanced memory chips (EMC) plus a linear expression evaluator tree (LEE). Each pixel processor has its own local memory and byte-serial ALU. Thus, pixel processors are capable of general arithmetic and logical operations and operate in SIMD fashion. The LEE tree can evaluate linear expressions of the form Ax+By+C for every pixel processor simultaneously, as fast as the EMC ALU and memory can accept the results.

The texture memory on a shader board provides storage for image textures. To speed up texture lookups, pixel processors are divided into eight modules, and the same amount of texture memory are duplicated for each module. Normally, the texture memory on eight modules have the same content. Therefore, texture lookups can be performed at local texture memory of each module.

The TASIC (Texture Application Specific Integrated Circuit) interface provides communication between texture memory and a special section of EMC memory, called the local port. A Communication link is also established between GP and the local port, but there is no direct link between GP and the texture memory.

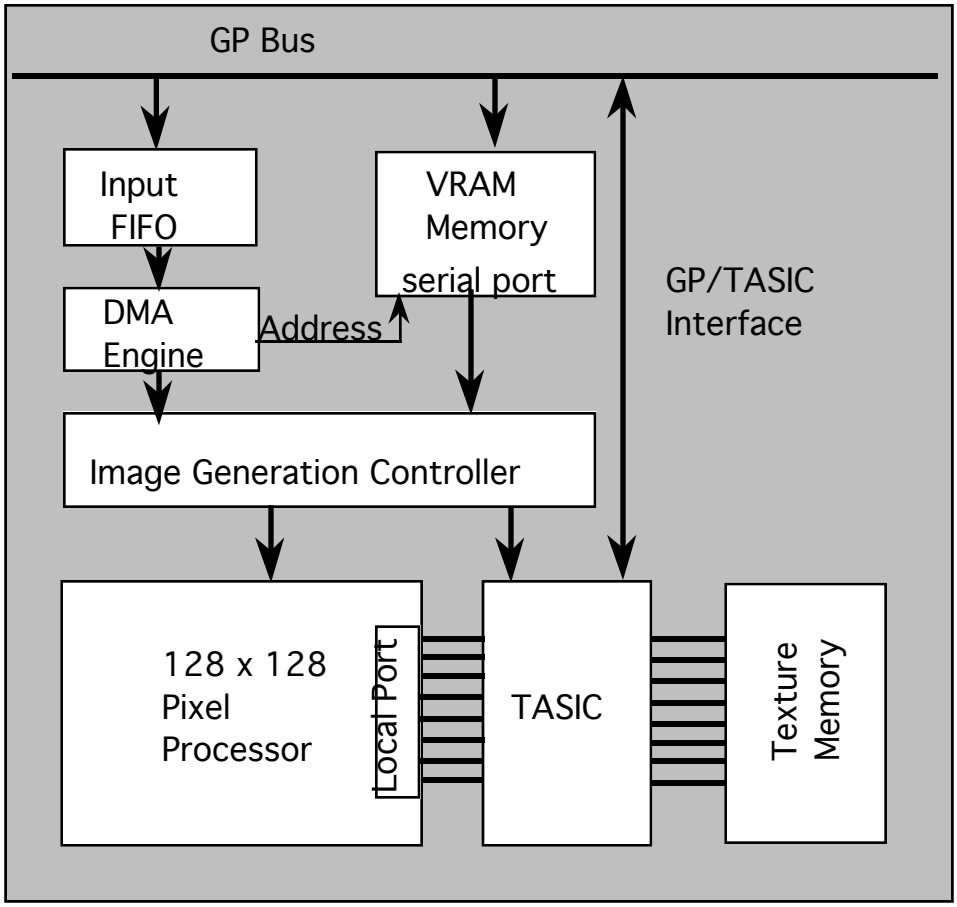

Figure 2: Block Diagram of Shader Board

1.2 About this document

Color picture 1 shows a student's office in Sitterson Hall. The painting on the wall is a texture-mapped polygon. To generate such a picture, we need:

(1) Allocate texture memory for the texture.

(2) Load the texture into texture memory.

(3) Pre-filter the texture to create mip-maps of the texture.

(4) Compute texture coordinates by linear interpolation.

(5) Compute the mip-map level index.

(6) Compute memory addresses for the eight texels.[1]

(7) Look up the eight texels.

(8) Bi-linearly interpolates the four texels from the same level, and then linearly interpolates the two results of the forgoing bi-linear interpolations.

This document discusses issues concerned with using PixelFlow's texture memory to generate anti-aliased image textures. Section 2 discusses perspective interpolation, analyses precision requirements of texture coordinates, and the special care needed for polygons with large depth variations. Section 3 describes mip-map memory organization and the address generation algorithm. Different methods of computing the mip-map level index are compared. Section 4 summarizes several schemes for loading textures into texture memory with emphasis on making full use of memory bandwidth. Parallel pre-filtering algorithms using pixel processors are also presented in this section. Section 5 outlines a texture memory manager based on the buddy system.

[1] An image texture is created by sampling an image. One sample of the texture is called a texel.

Color Picture 1: This model has 2000 polygons. The painting on the wall is a mip-map texture mapped to a polygon. The texture is of size 256x256.

2 Perspective Interpolation of Texture Coordinates and Gradients

Looking up texels of a texture requires two independent indices, referred to as texture coordinates. For polygons, texture coordinates are bound to the vertices of a polygon when it is created. Texture coordinates for arbitrary pixels within a polygon are interpolated values based on texture coordinates at the vertices. Quadric surfaces and cubic parametric surfaces can first be meshed as an array of patches. Meanwhile, the texture coordinates for the corners of patches can be generated. Then, each patch can be treated approximately as a polygon, texture coordinates are interpolated for pixels within patches.

2.1 Perspective Interpolation

One powerful capability of the shader board is to simultaneously evaluate linear expressions of the form Ax+By+C for every pixel processor, where x and y are screen locations of each pixel processor. Many graphics calculations can be cast into this linear expression form. One example is drawing a triangle. By calculating three linear expressions for the three edges (the coefficients are computed from vertices of the triangle), the pixel processors outside the triangle obtain negative values. Then the color for the triangle is boardcast to pixels within the triangle. If the triangle has different colors at the vertices, another linear expression is needed to compute interpolated color for all pixels within the triangle.

However, the linear expression is not precise here because of the fore-shortening effect that perspective projection produces. As an example of fore-shortening, one can think of the road going off into distance. The same distance appears shorter at further, as shown in Figure 3.

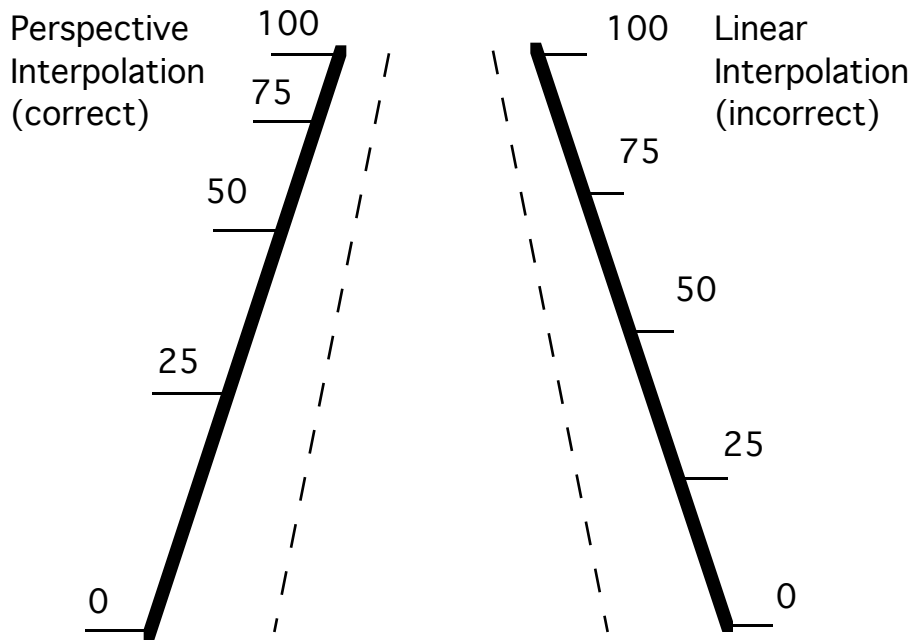

Figure 3: The road goes off into distance

For some computations, the errors introduced by improper linear interpolation is so slight that they are rarely noticed, like color in the above example. But the impact on texture coordinates is noticeable. One explanation is that texture coordinates are essentially geometry attributes rather than appearance attributes. We need perspective interpolation on geometry attributes to display depth cues.

Even though the texture coordinates u, v themselves are not linear in screen space, (u*z) and (v*z) are[1]. Suppose

$$(u*z) = Ax + By + C$$

We can transfer it to:

$$u = \frac{A}{z} * x + \frac{B}{z} * y + C$$

By differentiation, we have

$$\Delta u = \frac{A}{z} * \Delta x + \frac{B}{z} * \Delta y$$

This gives the correct fore-shortening effect -- texture coordinates u and v change steeper for distant objects.

Thus, one way of doing perspective interpolation of texture coordinates is to linearly interpolate (u*z) and (v*z). A parallel z divide($u=\frac{u*z}{z}$) is performed for all pixels. This method can be performed efficiently on PixelFlow because the cost of the division is independent of the number of pixels.

The nice result about texture gradients is that u_x*z^2 can be expressed as a linear form of Ax+By+C. The same can be said for u_y, v_x, and v_y. The following derivation for u_x is given as the proof.

Let

$$z = A_1 x + B_1 y + C_1$$

$$(u*z) = A_2 x + B_2 y + C_2$$

since

$$(u*z)_x = u_x*z + z_x * u$$

we have

$$u_x * z = (u*z)_x - z_x * u$$

Multiply z on both sides, that gives us:

$$u_x * z^2 = (u*z)_x * z - z_x * u * z$$

$$= (u*z)_x * z - z_x * (u*z)$$

$$= A_2 * (A_1 x + B_1 y + C_1) - A_1 * (A_2 * x + B_2 y + C_2)$$

$$= (A_2 * B_1 - A_1 * B_2) y + (A_2 * C_1 - A_1 * C_2)$$

a linear form in term of screen coordinates x, and y.

In summary, texture coordinates are perspectively interpolated by linearly interpolating u*z (u is one of the texture coordinates), followed by

parallel division over z in pixel processors. Texture gradients are perspectively interpolated by linearly interpolating U_X*z^2, followed by parallel division over z^2 in pixel processors. Computationally, these two divisions are accomplished by computing reciprocal and multiplications.

2.2 Precision Requirements of Texture Coordinates

Now consider the computational error involved in the multiplication and the division. Since EMC memory is tight(256 bytes per pixel processor), we hope to use as little memory as possible for representing texture coordinates u and v. For a texture of size 1k x 1k, 10 bits of precision are needed for both texture coordinates u and v. Therefore, two bytes for each of them should be sufficient. Suppose the representation for z needs four bytes, then the easiest solution is to use 6 bytes for the product of u*z. This method appears to double space requirement for each parameter.

However by appropriately scaling on z, an in-place method giving equivalent precision is possible. To be more specific, suppose z is represented by ZLen bits, u is represented by ULen bits. Let

$$u = \frac{u*z}{z}$$

we have

$$\text{error}(u) = \frac{\text{error}(u*z) * z + (u*z) * \text{error}(z)}{z * z}$$

$$= \frac{\text{error}(u*z)}{z} + \frac{u * \text{error}(z)}{z}$$

From this formula, we see one worthwhile effort to reduce error(u) is to ensure

$$z > 2^{ZLen-1},$$

such that

$$\frac{\text{error}(u*z)}{z} < \frac{1}{2^{ZLen-1}} \quad \text{and}$$

$$\frac{\text{error}(z)}{z} < \frac{1}{2^{ZLen-1}}$$

Thus

$$\text{error}(u) < \frac{2}{2^{ZLen-1}} = 2^{-ZLen+2}$$

Whereby 14 bit precision for texture coordinates u and v are acquired with 16 bit representations for u and v only.

One way to ensure $z > (2^{ZLen-1})$ is by proper left shifts. Instead of computing u*z, we compute u*z', where

$$z' = z * 2^{left_shifts},$$

such that z' has no leading zero.

Since z is left shifted to obtain z' before the division, error(z') < 1 is not guaranteed. Actually, if more guard bits than leading zeros in z are available for shifting into the right most bit of z' (z is usually longer than z'), error(z') equals zero. However, in the case that z has more leading zeros than available guard bits in z, error(z') > 1. A safer solution is to compute z' directly. This method guarantees error(z') < 1 at a cost of two bytes of memory for storing z'.

2.3 Polygons with Large Depth Variations

As the above analysis has shown, the z' value is required to have no leading zeros to obtain the required precision for texture coordinates u and v. For polygons with large depth z variation, no single scaling factor of 2^{left_shifts} exists that will scale z into range $[2^{ZLen-1}, 2^{ZLen} - 1]$ exists. Such polygons must be divided into multiple bands, and different scaling factors are applied for the different bands. As a result, the number of polygons actually rendered increases. To reduce the number of polygon bands, the restriction can be loosened a bit to allow four leading zeros in z' value, since only 10 bit precision is needed for the maximum textures of 1k x 1k.

Texture coordinates and gradients are used to compute memory addresses in the next section.

3 Mip-Map Memory Organization and Address Generation

A texture is re-sampled when it is mapped onto a surface. Consequently, when the sampling rate is too low compared to the high frequency inherent in a texture, the texture tends to display aliasing. One method of anti-aliasing is pre-filtering the texture to create a set of maps (tables) containing lower-resolution representations of the texture. These maps are called mip-maps of the texture, with mip's original meaning of multiple in Latin [Williams83].

For any particular polygon occupying a certain size on the screen, the mip-map level index, a parameter indicating the appropriate map resolution is determined from the texture gradients. The map level index is ordinarily a fractional number. Consequently, two levels whose level numbers (integers) enclose the level index are actually referenced. Two groups of four neighboring texels, surrounding the surface point designated by the texture coordinates, are retrieved.

The mip-maps of textures must be well organized in the texture memory to facilitate fast retrieval of texels. Section 3.1 discusses the organization of mip-map textures in the texture memory. Section 3.2 describes the memory address generation algorithm based on the organization. Section 3.3 presents and compares different methods of computing the mip-map level index.

3.1 Mip-Map Memory Organization

When a mip-map texture is mapped onto a surface, eight texels must be retrieved for each pixel. Real time textures would be out of the question if only sequential memory access was supported. On PixelFlow, a mip-map texture is stored in texture memory so that any eight texels that are likely to be requested at the same time are stored in different memory banks. This is done by grouping the eight memory banks into two "panels", with four memory banks in each panel (see figure 4). Banks 0, 1, 2, 3 form panel A. Banks 4, 5, 6, 7 form panel B. Adjacent levels of a mip-map texture are always allocated to different panels, as illustrated in Figure 4. For each mip-map, its size, parity, starting column (U_0) and starting row (V_0) are recorded when memory is allocated for it. Here "parity" indicates the panel in which the lowest level is stored, and (U_0, V_0) indicates the memory addresses for the texel(0, 0) of the lowest level. The memory for the other lower-resolution maps are allocated to the texture according to the rule that (U_i; V_i) = (U_0 >> i; V_0 >> i).

Furthermore, the neighboring texels in the same level are distributed over the four banks of one panel. They are 2 x 2 interleaved. Using a pair of (x, y) coordinates to denote the texel at row x and column y, x and y can be even or odd; by our convention, (even, even) texels are stored in bank 0 or bank 4; (even, odd) texels are stored in bank 1 or bank 5; (odd, even) texels are stored in bank 2 or bank 6; (odd, odd) texels are stored in bank 3 or bank 7, as shown in Figure 4.

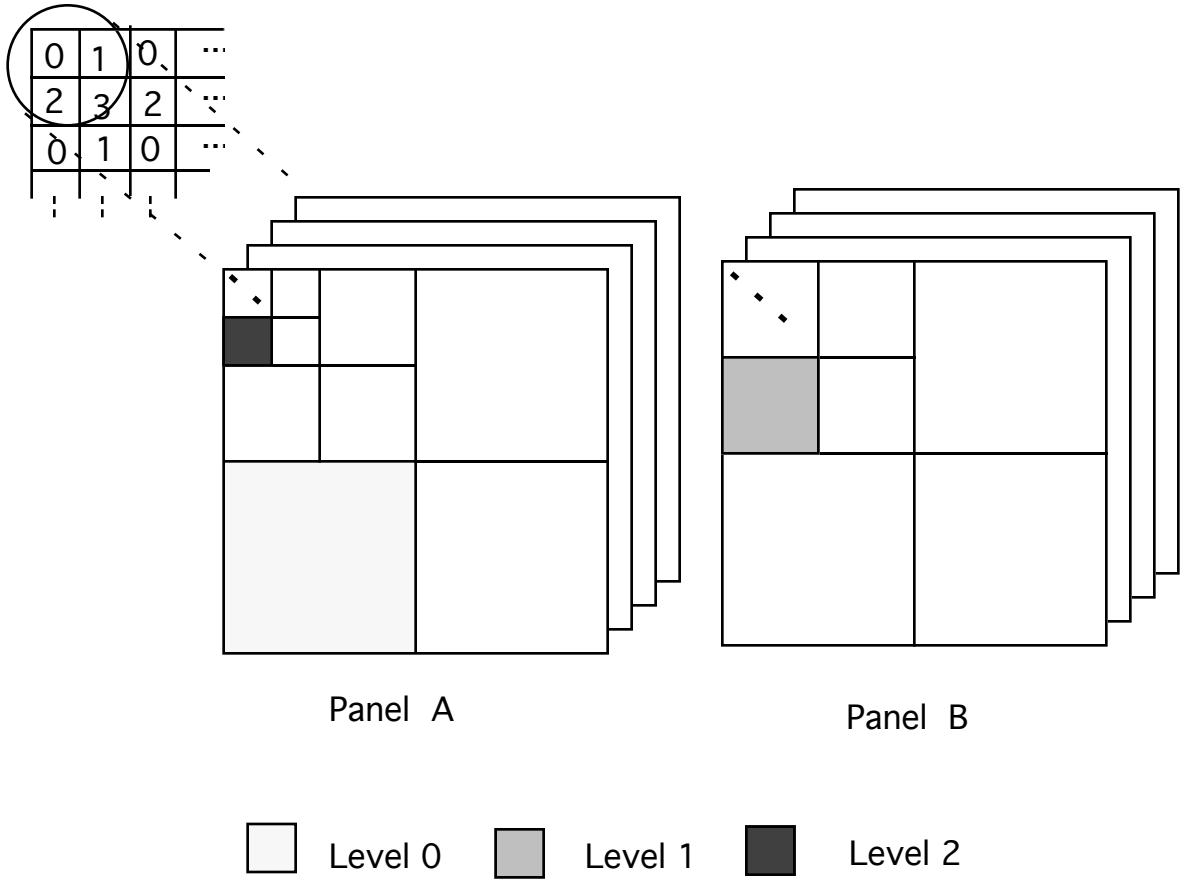

Figure 4: 2 x 2 Interleaved Memory Organization for Mip-Map

3.2 Mip-Map Address Generation

In the last section, I stated that the starting column (Ui) and the starting row (Vi) for map of level i can be derived by formulas of Ui = U0 >> i and Vi = V0 >> i. The remaining parts that need to be added are the offsets of a texel,

designated by the texture coordinates, from texel(0, 0) in the map. This section will give the complete address generation algorithm. For the moment, we assume that the level index D is known; we will discuss its computation in section 3.3.

On PixelFlow, memory addresses for texels in the eight banks are specified with eight pairs, namely (r0, c0), (r1, c1), (r2, c2), (r3, c3), (r4, c4), (r5, c5), (r6, c6), (r7, c7), as shown in figure 5.

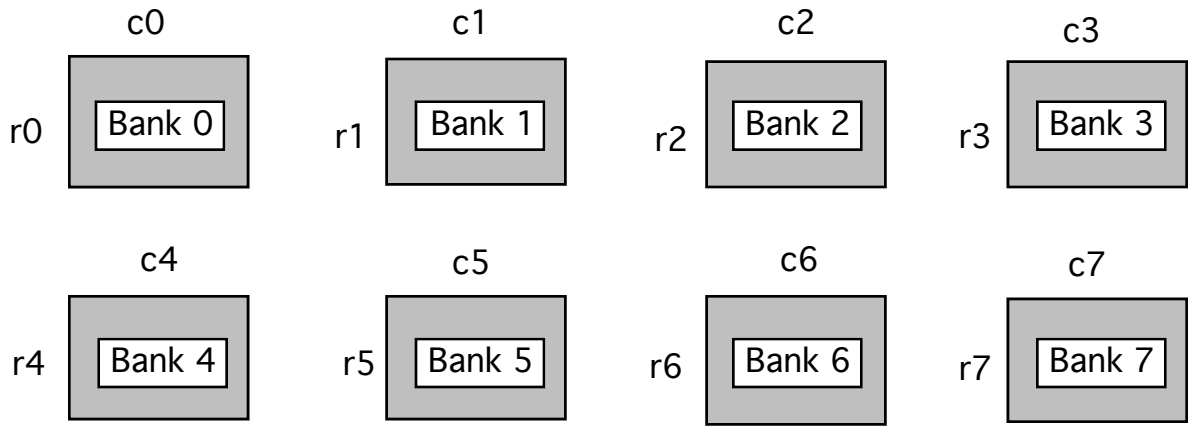

Figure 5: row and column addresses for eight memory banks

Given texture coordinates u, v and level index D, memory addresses for the eight texels are computed by the following algorithm, where Ar0, Ar1, Ac0, Ac1, Br0, Br1, Bc0, Bc1 are introduced as logical addresses; ASize as the size of higher resolution and BSize as the size of lower resolution:

ASize = Size >> Int(D)

BSize = Asize - 1

AU0 = U0 >> ASize

AV0 = V0 >> ASize

Ar0 = u >> ASize

```
Ac0 = v >> ASize
Ar1 = Ar0 + 1
Ac1 = Ac0 + 1
AULerp = u << (UVLen - ASize)
AVLerp = v << (UVLen - ASize)
if (odd(Ar0))
{
    swap Ar0 with Ar1
    swap Ac0 with Ac1
    AULerp = 1 - AULerp
    AVLerp = 1 - AVLerp
}
compute Br0, Br1, Bc0, Bc1, BULerp, BVLerp similarly.
DLerp = Fraction(D)
if (odd(parity + Int(D))
{
    swap Ar0 with Br0
    swap Ar1 with Br1
     swap Ac0 with Bc0
    swap Ac1 with Bc1
    swap AULerp with BULerp
    swap AVLerp with BVLerp
    DLerp = 1 - DLerp
 }
 r0 = Ar0, c0 = Ac0
```

r1 = Ar0, c1 = Ac1

r2 = Ar1, c2 = Ac0

r3 = Ar1, c3 = Ac1

r4 = Br0, c4 = Bc0

r5 = Br0, c5 = Bc1

r6 = Br1, c6 = Bc0

r7 = Br1, c7 = Bc1

AULerp, AVLerp, BULerp, BVLerp, DLerp are the blending factors for the two bilinear interpolations and the one linear interpolation that are applied to the eight texels retrieved. The address generation algorithm is computationally cheap (about 1000 clock cycles), since it only requires shifts, truncations, and bit reverse operations. In return, texture memory for different-resolution maps of a mip-map texture must be tied together.

3.3 Mip Map Level Index Computation

Now we look into the algorithm for computing the level index D. As an intermediate step, We introduce texture scale factor f to represent the maximum magnification factor that can occur when a unit vector in screen space is mapped to texture space. Given f, D is a function of f as :

$$D = \log(\frac{1}{f}) = - \log(f)$$

where $0 < f < 1$.

Let u_x, u_y and v_x, v_y denote texture gradients with respects to x and y. We have:

$$\Delta u = u_x * \Delta x + u_y * \Delta y$$

$$\Delta v = v_x * \Delta x + v_y * \Delta y$$

In matrix form, they can be rewritten as:

$$\begin{vmatrix} \Delta u \\ \Delta v \end{vmatrix} = \begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix} * \begin{vmatrix} \Delta x \\ \Delta y \end{vmatrix}$$

So the problem of computing the maximum texture scale factor f is converted to a problem of computing the maximum eigen value for the matrix:

$$\begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix}$$

namely the root of equation:

$$(u_x - \lambda) * (v_y - \lambda) - v_x * u_y = 0$$

Or

$$\lambda^2 - (u_x + v_y)\lambda + u_x * v_y - v_x * u_y = 0$$

The solution is

$$\lambda = \frac{(u_x+v_y) + \sqrt{(u_x+v_y)^2 - 4*(u_x*v_y - v_x*u_y)}}{2}$$

This formula is seldom used because of its computational and pixel memory expense. However, a more efficient approximation is to assume that log(f*z) is constant across a polygon and then compute an average log(f*z) for the whole polygon. Thus, only one pixel memory for the final result of log(f*z) is used. Finally, D = - log(f) is computed by the formula

$$D = \log(z) - \log(f*z)$$

in parallel on pixel processors, with different z on each pixel processor. Let

$$f*z = \lambda*z = \frac{(u_x+v_y) + \sqrt{(u_x+v_y)^2 - 4*(u_x*v_y - v_x*u_y)}}{2} * z$$

The average

$$\overline{f*z} = \frac{(\overline{u_x*z}+\overline{v_y*z}) + \sqrt{(\overline{u_x*z}+\overline{v_y*z})^2 - 4*(\overline{u_x*z}*\overline{v_y*z} - \overline{v_x*z}*\overline{u_y*z})}}{2}$$

where $\overline{u_x*z} = (u*z)_x - \bar{u} * z_x$

Formulas for u_y, v_x, v_y can be derived similarly.

The above method can be thought of as computing D approximately on the accurate formula of f. The second method we want to compare here is to compute D accurately using an approximate formula. The approximate formula we used for f is:

$$f = \max(\sqrt{u_x^2 + u_y^2}, \sqrt{v_x^2+v_y^2})$$

The first method tends to generate discontinuities of D across the common edge when a quadrilateral is divided into two triangles, since the average log(f*z) for the two triangles may differ. The second method is expected to give better result at a cost of more pixel memory for u_x, u_y, v_x and v_y. The computation cost is one overall multiplication for computing $\frac{1}{z^2}$ and one extra multiplication for each texture gradient.

So far, we have shown how texture coordinates u and v, level index D, are computed for each pixel and how memory addresses for the eight texels are generated from texture coordinates, level index, and the starting column (U_0), starting row (V_0) associated with a texture. However, textures must be loaded into texture memory before they are used to decorate surfaces. In the next section, I will describe several schemes for loading textures into texture memory and pre-filtering algorithms for creating mip-maps of textures.

4 Texture Loading and Texture Pre-filtering

Ordinarily, the lowest-level map of a texture is first loaded into texture memory, and then parallel pre-filtering is performed on the pixel processors

to create other lower-resolution maps, which are loaded into texture memory after the pre-filtering.

4.1 Color Encoding

Various textures have various types of texels. Color image textures usually have color triples (R, G, B) for each texel. z values of depth maps require single real numbers for each texel. Consequently, It is difficult to satisfy different types of textures with a single memory word length. The current design supports 8 bits, 16bits, and 32 bits texel types. When 16 bits texel type is chosen for a color image, textures must be encoded. The simplest encoding method for color images is to divide the 16 bits into three components, with 5 bits, 6bits, and 5bits for R, G, B, respectively. Simple zero-padding and rounding are used for decoding and encoding. This method has been used to generate several acceptable pictures. At the early stage of design, different color models and more complicated encoding schemes were considered, but they were not implemented due to the computation cost.

4.2 Texture Loading

As mentioned before, texture memory are duplicated for each pixel processor module. Texture lookups are done at pixel processor's local texture memory. But, texture loading requires global memory writes to update all eight memory modules. One global write let one pixel processor module update all of the eight texture memory modules. Consequently, if more than one pixel processor module has active pixels, more than one global memory write must be performed.

Before the load command(TAS_MemWr) can be issued, the texels should be stored in the high eight words of the EMC local port. And the memory addresses for these texels need be computed and stored in the low eight words of the EMC local port. Then, the local port is configured such that both the eight words for texels and the eight words for addresses act as outputs [Eyles4].

A texture may be stored in the EMC local port in one of several ways. The goal of designing good loading schemes is to use the memory bandwidth to the

full extent, so that the average cost of loading a texel is as low as possible. Considering number of regions and number of texels in each pixel, three different cases are analyzed below.

(1) Loading 1-region/4-texels textures

Four neighboring texels of one region are stored in each pixel. This is the ideal situation. It can occur if an image texture is broadcast to EMC from GP. The loading procedure is straightforward. A global memory write for each active pixel processor module can be started after generating row addresses and column addresses for each pixel. One shader having eight pixel processor modules can load 4 x 128 x 128 texels by 8 global memory writes. Therefore on average, a texel costs $\frac{8}{4 \text{ x } 128 \text{ x } 128}$ = 0.0001 memory access cycles.

(2) Loading 1-region/1-texel textures

One texel for one region is stored in each pixel. A global memory write can simply load each texel into the correct memory word.

The cost will be $\frac{8}{128 \text{ x } 128}$ = 0.0004 memory access cycles per texel.

(3) Loading 4-regions/1-texel textures

In case that a texture is the result of a rasterization, shadow maps are examples, 4 region and 1 texel can be arranged in EMC and loaded together. It gives lower average cost. Loading 4 region and 1 texel texture follows the following procedure:

* Load four texels belonging to different regions into texture memory by a global memory write. As a result, four different memory banks have different regions' texels instead of neighboring different texels.
* Store the texels back into EMC memory by four consecutive memory reads. Each memory read gives one useful texel for each pixel. The other three are discarded.
* Load the 1 region 4 texels texture into texture memory

Loading 4 x 128 x 128 texels requires two global memory writes and four memory reads are needed. The average cost is

$$\frac{2 * 8 + 4}{4 \times 128 \times 128} = 0.0003 \text{ memory access cycles}$$

4.3 Texture Pre-filtering

Simple box filtering is used for mip-map pre-filtering. Average values are computed from neighboring four texels. Pre-filtering a larger size texture recursively invokes procedures to pre-filter smaller textures. The following c-like code shows the structure of the pre-filtering routines.

```
prefilter_mipmap_1024()
{
    /* perform filtering and obtain texture of size 512 */
     prefilter_mipmap_512();
}

prefilter_mipmap_512()
{
    /* perform filtering and obtain texture of size 256 */
     prefilter_mipmap_256();
}
...
```

The pre-filtering can be carried out on GPs or APs. And the lower resolution textures must be broadcast to EMC pixel memory before being loaded into texture memory. As an alternative, the pre-filtering can done in parallel using pixel processors. The strategy for parallel pre-filtering algorithms is to

activate as few pixel processor modules as possible, because the number of global writes needed afterwards is equal to the number of pixel processor modules that have enabled pixels. When the number of active pixel processor modules is reduced to the minimum, the second effort is to distribute pixel processors equally among EMC chips, because the actual execution time of the global texture write command depends on the chips which has most enabled pixel processors.

(1) Pre-filtering a texture of 512 x 512

The immediate higher level texture would be of 256 x 256. By arranging four neighboring texels in one pixel, a full shader (128 x 128 pixel processors) is needed.

Pre-filtering a texture of 1024 x 1024 is accomplished by dividing it into four 512 x 512 textures and pre-filtering the four 512 x 512 textures.

(2) Pre-filtering a texture of 256 x 256

The immediate higher level texture would be of 128 x 128. By arranging four neighboring texels in one pixel, 64 x 64 pixels will suffice. 64 x 64 pixels require 8 pixel processor modules. But two 64 x 32 pixel arrays can also do the job, which only requires 4 pixel processor modules.

Pre-filtering textures of 128 x 128 is similar. By using two 32 x 16 pixel arrays instead of one 32 x 32 pixel array, only one pixel processor module is used.

(3) pre-filtering a texture of 64 x 64 or smaller textures

For textures smaller than 64 x 64, one pixel processor module suffices. In this case, we want to distribute pixel processors needed to all EMC chips equllay, or almost equally. Fortunately, the layout of EMC chips makes this quite simple, we can naturally choose pixel processors in the same module beginning from the left-up corner.

5 Texture Memory management

Texture memory is a shared resource among textures. It requires management for easy and efficient use like any other resources in a computer

system. Section 5.1 outlines a manager based on the buddy system. Sections 5.2 and 5.3 discuss problems introduced by rectangle and flat textures, respectively.

5.1 A Two Dimensional Buddy System

The texture memory manager is an extended exponential buddy system (Memory is allocated and deposited as blocks that are of size of 2 to some power). Texture memory blocks must be allocated and deposited as a two dimensional array. So a block has four buddies rather than the two buddies in normal buddy systems. Considering square textures only, possible block sizes includes 1024x1024, 512x512, 256x256, 128x128, 64x64, 32x32, 16x16, 8x8, 4x4, 2x2.

The memory manager maintains 10 headers. Header i points to a free list of blocks of size 2^i. When a memory block of size 2^i is requested, the manager first tries to allocate one from the free list pointed to by header i. If there is no such block available(header i has nil value), the nearest headers are searched until a free block is available. This free block is divided into four smaller blocks, three of them are added into free list pointed to by the next header, the other one is the one to be further divided. The division procedure stops when the size of the fours blocks are the same size as requested. Three of the four blocks are added to the free list; the remaining one is returned.

When a block is to be reclaimed, its three buddies can be determined and checked to see if they are free. If they are all free, the four free blocks are coalesced into a single bigger block. The coalescence may be repeated.

5.2 Rectangle Textures

The above discussion only addresses square textures. In reality, many rectangle textures occur. The label across a book cover is an example. Enlarging a rectangle texture to the size of a surrounding square wastes texture memory. On the other hand, if all textures of different size, ranging from 1024 x 1024, 1024 x 512, 1024 x 256, ... 1024 x 2 to 2 x 1024, 2 x 512, 2 x 256, ... 2 x 2, are allowed, the overhead of maintaining the headers would be significant. Alternatively, we may decide to support rectangle textures that are

twice as wide as they are long, or vice versa. Other rectangle textures would be enlarged to the nearest size.

One impact of rectangle textures on the system is that more combinations of texture sizes in two dimensions complicates pre-filtering. Since every different size needs one specific pre-filtering routine, which is specially designed in order to use fewer pixel processor modules and fewer EMC chips.

5.3 Flat Textures

So far, the discussion has assumed that all textures are to be used as mip-map textures. In some cases, a single level texture suffices. Storing a flat texture as a mip-map requires $\frac{3}{4}$ the memory that would otherwise be required. However, a simple flat texture saves pre-filtering time.

6 Conclusion

Although many parameters of graphics primitives can be linearly interpolated in screen space, geometry parameters such as texture coordinates require perspective interpolation. The space overhead is one extra bit for each parameter. And the time overhead is one overall reciprocal and multiplication, and one extra multiplication for each texture coordinate or gradient.

Mip-Map textures must be stored into texture memory in a pre-defined organization to accommodate real time retrieval of texels. The mip-map level index D computation dominates the address generation time. Some approximations can be made to speed up the generation at the cost of image quality. Texture Loading and pre-filtering are considered as initial costs only.

Besides storing textures, texture memory can be used as backing store, or as a communication mechanism between pixel processors. In this case, texture memory serves as a shared memory visible to all pixel processors. Pixel processor's local memory can be swapped out and swapped in from different texture memory segments. The loading schemes presented in this document provide examples of how memory bandwidth can be used efficiently.

Acknowledgments

I want to express special thanks to Steve Molnar and John Rhoades. I also want to thank Anselmo Lastra, John Eyles, Carl Mueller, Marc Olano for suggestions.

REFERENCES:

[1] H. Fuchs, J. Goldfeather, etc., Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes, *Computer Graphics*, 19, 3 (July 1985), 79-88.

[2] H. Fuchs, J. Poulton, etc., Pixel-Plane 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced memories, *Computer Graphics*, 23, 3 (July 1989), 79-88.

[3] S. Molnar, etc., PixelFlow System: High Speed Rendering using Image Composition, *Computer Graphics*, 26, 2 (July 1992), 231-240.

[4] L. Williams, Pyramidal Parametrics, *Computer Graphics*, 17, 3 (July 1983), 1-10.

[5] J. Eyles, S. Molnar, PixelFlow Rasterizer Functional Description, *PixelFlow Documents*, (Dec. 1992).