Hardware for Superior Texture Performance

G. Knittel, A. Schilling, A. Kugler and W. Straßer
WSI/GRIS[†]
University of Tübingen, Germany

Abstract

Mapping textures onto surfaces of computer-generated objects is a technique which greatly improves the realism of their appearance. Unfortunately, this imposes high computational demands and, even worse, tremendous memory bandwidth requirements on the graphics system. Tight cost frames in the industry in conjunction with ever increasing user expectations make the design of a powerful texture mapping unit a difficult task.

To meet these requirements we follow two different approaches. On the technology side, we observe a rapidly emerging technology which offers the combination of enormous transfer rates and computing power: logic-embedded memories.

On the algorithmic side, a common way to reduce data traffic is image compression. Its application to texture mapping, however, is difficult since the decompression must be done at pixel frequency.

In this work we will focus on the latter approach, describing the use of a specific compression scheme for texture mapping. It allows the use of a very simple and fast decompression hardware, bringing high performance texture mapping to low-cost systems.

Keywords: graphics hardware, texture mapping, image compression

1 Introduction

During the rasterization process, mapping images onto objects can be considered as the problem of determining a screen pixel's projection on the image (which we call its *footprint*) and computing an average value which best approximates the correct pixel color. In real-time environments, where several tens of millions of pixels per second are issued by fast rasterizing units, hardware expenses for image mapping become substantial and algorithms must therefore be chosen and adapted very carefully. Thus, the straightforward approach of taking the mean of all image pixels *t* (or *texels*) inside the footprint for the screen pixel's color $C(x,y)$

$$C(x, y) = \frac{1}{M} \cdot \sum_{m=1}^{M} t_m , \qquad (1)$$

or, more generally, defining a filter kernel h, which is convolved over the image $t(\alpha,\beta)$ [1]

$$C(x, y) = \iint (h(x-\alpha, y-\beta) \cdot t(\alpha, \beta))\, d\alpha d\beta \qquad (2)$$

can be excluded from further discussion due to the long computing times. *Summed-area-tables* [3] are an attempt to simplify and speed up the above operations. Instead of the color value, each cell of a summed-area-table holds the sum of all values in a certain region, usually the rectangle defined by the position of the cell and the origin as indicated in Figure 1.

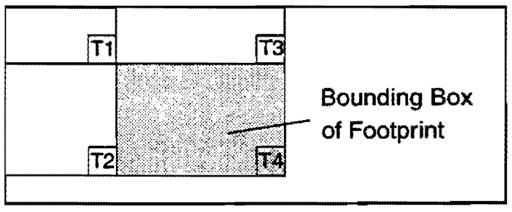

Figure 1: Summed-Area-Table

Given the bounding box of the footprint, $C(x,y)$ is then approximated by accessing the table four times and performing the following operation:

$$C(x, y) = T4 - T3 - T2 + T1 . \qquad (3)$$

However, since the footprint of a pixel is not rectangular, but can be considered as a quadrilateral in the

† Universität Tübingen
Wilhelm-Schickard-Institut für Informatik - Graphisch-Interaktive Systeme (WSI / GRIS)
Auf der Morgenstelle 10, C9
D-72076 Tübingen, Germany
Phone: ..49 7071 29 5461 FAX: ..49 7071 29 5466
email: [knittel,andreas,strasser]@gris.informatik.uni-tuebingen.de
www: http://greco.gris.informatik.uni-tuebingen.de/

general case, a potentially large number of texels within the bounding box contribute without reason to the pixel color. Glassner proposes as a solution to incrementally remove rectangles within the bounding box to best approximate the footprint at the cost of increased computing times [6].

For two reasons summed-area-tables are not well suited for a direct hardware implementation:

- If the color components are 8-bit quantities, a 1024x1024 summed-area-table requires entries as wide as 28 bits for each color component.
- For each pixel four random accesses must be performed which limit the achievable texturing speed.

Parallelization schemes for summed-area-tables show substantial disadvantages. Replicating the entire map four times leads to unacceptable memory capacity requirements. Interleaving the table across four memory banks (similar to the scheme shown in Figure 2 for level 0) would require to round the bounding box dimensions to even numbers. This would further reduce texturing accuracy especially for small footprints.

Another approach is to create a set of prefiltered images, which are selected according to the level of detail (the size of the footprint) and used to interpolate the final pixel color. The most common method is to organize these maps as *mipmap* as proposed by Williams [11]. In a mipmap, we denote the original image as level 0. In level 1, each entry holds an averaged value and represents the area of 2x2 texels. This is continued until we reach the top-level, which has only one entry holding the average color of the entire texture. Thus, in a square mipmap, level n has one fourth the size of level $n-1$.

The shape of the footprint is assumed to be a square of size q^2, where, for example,

$$q = \max\left(\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right), \quad (4)$$

as suggested in [8]. In (4), u and v denote texture coordinates, x and y screen coordinates.

The mipmap is accessed by the texture coordinates u,v of the pixel center and the level λ, which in the general case is a function of $\log_2 q$. For example, if λ has an integer part λ_I and a fractional part λ_F, λ can be written as

$$\lambda_I = \lfloor \log_2 q \rfloor \qquad \text{and} \qquad \lambda_F = \frac{q}{2^{\lambda_I}} - 1 \qquad (5)$$

Nearest-neighbor-sampling, however, is inadequate due to severe aliasing artifacts. Instead, the levels λ and $\lambda+1$ are accessed and bilinearly interpolated at u,v. The final pixel value is linearly interpolated from the results in both levels according to λ_F.

Mipmapping is a reasonable candidate for a hardware implementation due to its regular access function. If the memory is designed to deliver all eight texels for a tri-linear interpolation in a single access, texturing can potentially keep up with fast rasterizer units. This is accomplished by having eight independent memory banks and a conflict-free address distribution as shown in Figure 2. Furthermore, to reduce data traffic between the rasterizer unit and the texture system, all address calculations concerning the eight bank addresses as well as the tri-linear interpolation should be performed locally.

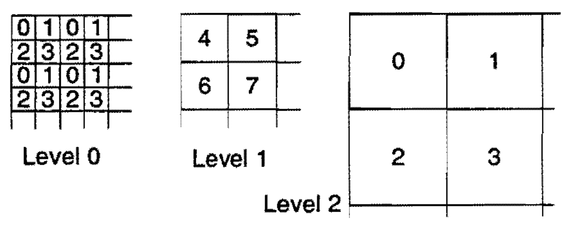

Figure 2: Assignment of Texels to Memory Banks

At this point it becomes obvious that the ideal solution is a highly-integrated memory device which incorporates all needed arithmetic units for fast mipmapping. An architectural description of such a device can be found in [12]. Taking advantage of the potentially high speed of such a memory device, we can alleviate the deficiencies arising from the square footprints. Our novel approach for image enhancement is outlined in section 7. Logic-embedded memories have also been shown to provide a quantum leap in performance in other areas such as the Z-Buffer [4],[9] and thus, we have developed a graphics pipeline based on enhanced memories for high performance in low-cost systems [10].

However, logic-embedded memories, especially logic-embedded *DRAMs*, is a new technology which is very expensive and risky. For this reason, our architectural proposals have not been realized yet.

Mipmapping in a traditional implementation either requires a parallel memory system or sequential accesses to the texture buffer and is therefore either expensive or slow. An obvious solution to both problems would be to compress the textures, thus saving memory costs and reducing memory bandwidth requirements. Commonly used image compression schemes such as JPEG, however, are not suitable for texture mapping since they do not fulfil two essential requirements for texture mapping:

- ❑ The decompression has to be simple and very fast, and
- ❑ random access to texels must be possible.

Here we propose to use a specific compression scheme which meets the above requirements. A hardware architecture is presented which integrates texture mapping units together with a small texture cache on a chip. The filtering can then take advantage of the extremely high bandwidth which is available on-chip. The off-chip bandwidth for updating the on-chip cache is reduced, so that standard off-the-shelf DRAM devices can be used.

The compression algorithms are discussed in section 2 and 3. A hardware architecture is given in section 5. The benefits in terms of costs and performance are detailed in section 6. Section 7 presents a novel way of overcoming the deficiencies of square footprints, and the results can be seen in section 8.

2 Block Truncation Coding / Color Cell Compression

Block Truncation Coding (BTC) was introduced by Delp et. al. [5] in 1978. In 1986, Campbell et. al. [2] presented Color Cell Compression (CCC), a modification and application of BTC to color images.

The main idea of BTC/CCC is to use a local 1-bit quantizer on 4x4 pixel blocks. The compressed data for such a block thus consists of only two colors and 16 bits that indicate, which one of the two colors is assigned to each of the 16 pixels. Figure 3 shows the

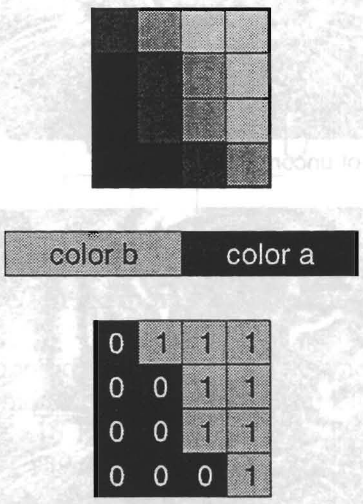

Figure 3: Quantization of 4x4-blocks to two colors

principle of the BTC/CCC. For further data reduction, the 24 bit colors can be globally quantized to 8 bits, using a standard quantizer such as the Heckbert quantizer [7]. The decoding of a CCC-encoded image is very simple. The principle of the decoder circuitry is shown in Figure 4. It consists of a multi-

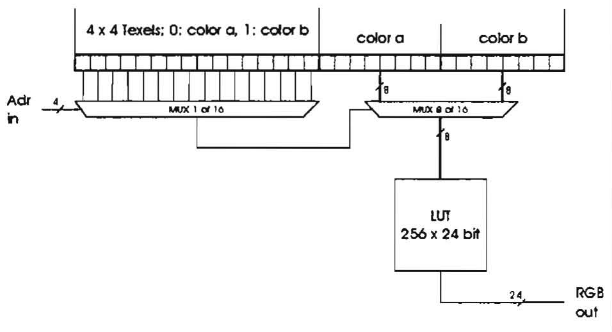

Figure 4: CCC Decoder Circuitry

plexer and a lookup table. Once a 16-texel-block (32 bits) is retrieved from memory, the individual texels are decoded by looking up the two possible colors for that block and selecting the color according to the associated bit from the 16 decision bits. The decoder fulfils the listed requirements for texture mapping in a nearly ideal way.

3 Compressing the Images

The compression of the texture maps can and should be performed in advance. This not only saves space on the storage media for the description of the scene, but also allows more complex algorithms to be used for the quantization due to the off-line compression. [5] shows for grey scale images how the first three sample moments can be preserved by choosing appropriate threshold and output levels. In [2], only the luminance of the input colors is used for clustering. If there are different colors with similar luminance in the same block, this method will fail. The quantization with the minimum mean square error can only be found by exhaustive trial, resulting in long compression times. For a quicker approximate solution we therefore propose to split the input values into two clusters by a plane perpendicular to the axis with the minimum "moment of inertia". For that purpose we calculate the tensor of inertia from the individual colors $\vec{x}_j$ as

$$\Theta_{ik} = \sum_{j=1}^{16} \left\|\vec{x}_j\right\|^2 \delta_{ik} - x_{ji}x_{jk}, \qquad (6)$$

where $i, k \in \{R, G, B\}$ and $\delta_{ik}=1$ for $i=k$, 0 else.

We then calculate the eigenvector with the smallest eigenvalue using standard methods. Multiplication of the individual colors with that eigenvector reduces the clustering problem to one dimension and allows the colors to be sorted according to their distance to a plane parallel to the cutting plane. The quantization threshold is set to the mean distance. In this way the mean color is in the cutting plane.

4 Image Quality

The examples in Figure 5 have been compressed using CCC as described in [2].

5 Texture Mapping Architecture

A VLSI-design for integrated rasterization and texture mapping as shown in Figure 6 is currently being implemented within the framework of the CEC's MONOGRAPH project. A small, four-entry cache of 32 Byte is sufficient to store 64 texels. The colors are looked up immediately after reading their index values from the external DRAM. A bilinear interpolation is performed on 4 neighboring texel colors. If a complete trilinear interpolation is required, the cache should be duplicated for the other level of detail, however, the interface to the DRAM can be used for both levels. A single DRAM chip (256K×16 or 1M×16), connected glueless to the rasterizer, provides 2Mtexel (8Mtexel) of texture memory (1.5Mtexel/6Mtexel mipmapped). The interface requires 31 additional pins (16 data-, 11 address- and 4 controlpins).

A direct mapped cache is used: each memory address is assigned one unique cache cell (A through D) in an interleaved way (see Figure 7). This unambiguous way of addressing allows a simple address calculation logic to be used. A simple form of address look-ahead speeds up the operation of the cache: neighbors of the current block are preloaded if the DRAM interface is idle. We choose the three neighbors adjacent to the quadrant of the point addressed by the current *u*- and *v*-values. This decision can be made using only two bits, one bit of the *u*- and the other of the *v*-address. A more sophisticated lookahead could use the direction of consecutive accesses within a scanline, which could be calculated from the two previous access points. The rasterizer should provide a flag to indicate a scanline change, which could be used to disable the prefetch and lookahead calculation for that cycle.

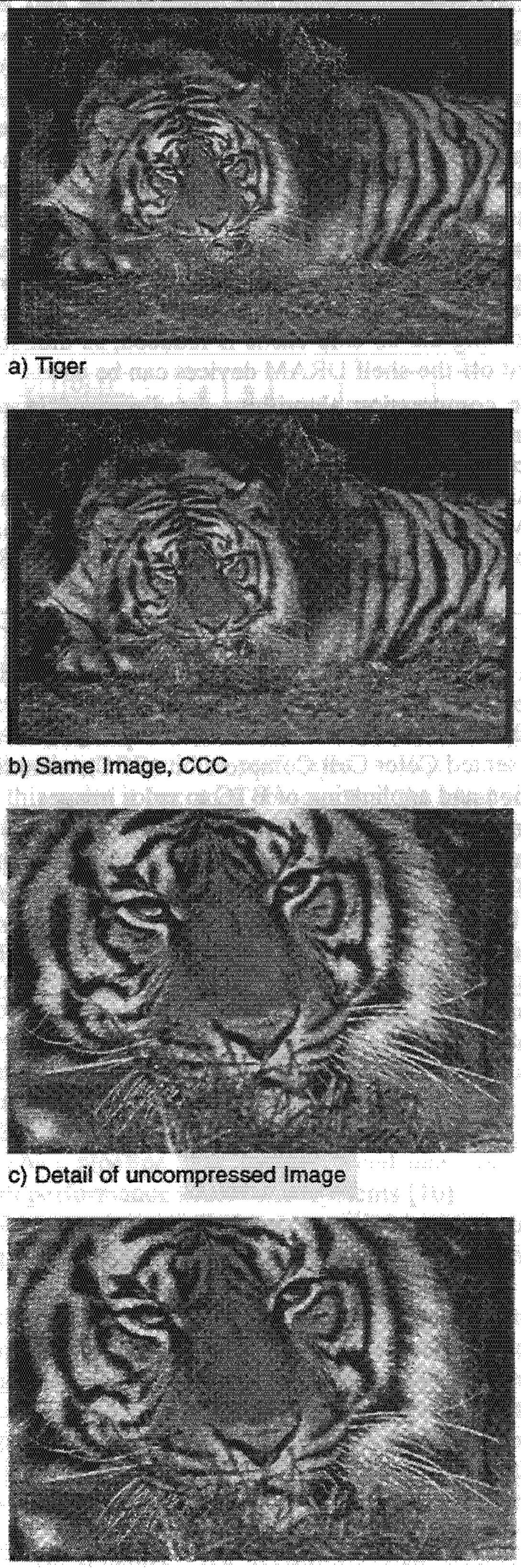
a) Tiger
b) Same Image, CCC
c) Detail of uncompressed Image
d) Detail of compressed Image

Figure 5: Original and compressed Images

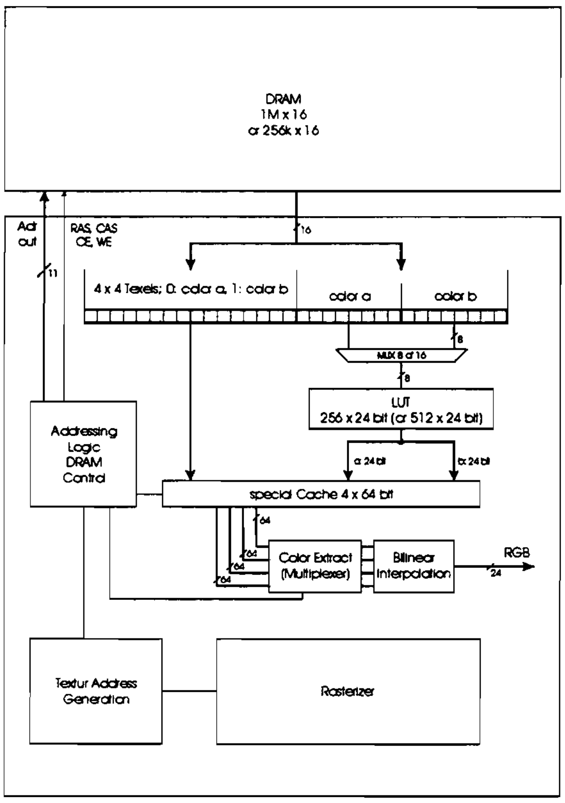

Figure 6: Texture Mapping Architecture

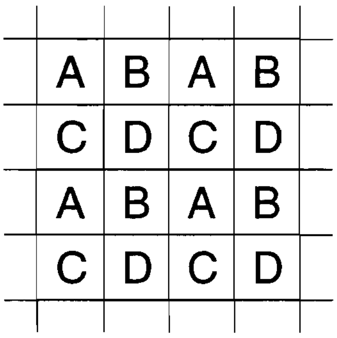

Figure 7: Assignment of Texel Blocks to Cache Cells

6 Price/Performance Comparison

In our proposed architecture, we assume an average of 3 mipmap accesses per 4x4 texel block. If standard 256Kx16 DRAM is used, one page can hold 512 words, which corresponds to 16x16 blocks. We assume therefore that about every 24 blocks a new page has to be accessed. Thus about 2% (1/48) of the DRAM-accesses result in a page fault. This leads to the following performance estimation (assuming bilinear interpolation, a page access time of 50ns and a random access time of 100ns). 24 blocks are accessed in

$$t = 47 \cdot 50ns + 1 \cdot 100ns = 2450ns. \tag{7}$$

72 mipmap accesses can be performed in this time. Assuming 4 accesses for bilinear interpolation, we get a rate R of

$$R = 29.4 Mtexel/s. \tag{8}$$

We compare this to a design using 8-bit index mapping with one 16Mbit DRAM-chip (2Mx8) for the same texture size (the memory has to be 4 times larger). With a page size of 32x32 texels and 1 page fault every 24 mipmap accesses we get the following results. 24 mipmap accesses are performed in

$$t = 71 \cdot 50ns + 1 \cdot 100ns = 3650ns, \tag{9}$$

if one mipmap access includes 4 texel accesses (of which we count only three, as we assume caching for a fair comparison):
For the rate we get:

$$R = 6.6 Mtexel/s. \tag{10}$$

The result of the comparison is summarized in Table 1.

	CCC	Standard System
Memory	4Mbit	16Mbit
Bus	16bit	8bit
Page Size	512 x 16	1024 x 8
Texture size	2Mtexel =1.5Mtexel mipmapped	2Mtexel =1.5Mtexel mipmapped
Speed (mipmap, bilinear interpolation	**29.4Mtexel/s**	**6.6Mtexel/s**
Memory Costs (June 95)	**$15**	**$60**
Prize/Performance Ratio	**2Mtexel/s / $**	**0.1Mtexel/s / $**

Table 1: Prize/Performance Comparison

7 Footprint Assembly

Footprint assembly is a novel way of mapping textures onto surfaces. The main idea is to approximate the projection of the pixel on the texture by a number N of square mipmapped texels. $N = 2^m$ for practical reasons, so the texels can be summed up and shifted right m places to give the final texture color.

Figure 8 shows how the projection of a pixel

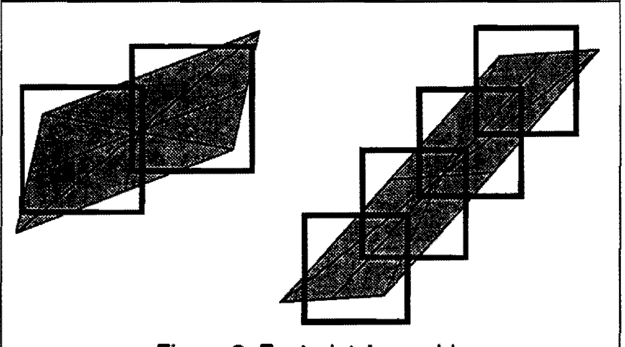

Figure 8: Footprint Assembly

(assumed to be a parallelogram, drawn in dark grey) on the texture map is approximated by a sequence of mipmap accesses. Footprint assembly requires only small additional hardware. The sequence of texture coordinates is generated internally to the texture mapping unit, so that this kind of texturing is still very fast for a reasonable m. To avoid unacceptable computing times the user can set an upper limit for m. Detailed information about footprint assembly can be found in [12].

The enhanced image quality justifies the increased rendering time, as can be seen in Figure 9.

The upper four views were created by standard mipmapping and show vanishing details towards the background. The four images at the bottom were generated by footprint assembly with N limited to 16.

8 Image Quality

The visual quality of a compression scheme can in the end only be judged by human observers. We made simulations of texturing with the described CCC compression scheme in conjunction with footprint assembly. In Figure 9, the results for different combinations can be seen (unfortunately only in black and white). Compared to original textures, still images as well as animations show noticeable but no disturbing artifacts for compressed textures. Mipmapping with bi-linear interpolation generally exhibits severe artifacts, independent of the use of image compression. Footprint assembly retains texture details even for objects in the background. For cost-efficient systems, we propose to combine bilinear interpolation, CCC and footprint assembly, a way to improve texture mapping performance significantly.

9 Conclusion

Compression in the context of texture mapping serves two purposes: it reduces the size and thereby the cost of the required memory and, equally important, it reduces the required memory bandwidth. However, several requirements have to be fulfilled. The most important one is, that the local decompression of the stored texture is possible and can be performed very quickly. Color Cell Compression (CCC) is identified as an extremely useful image compression technique for texture mapping. CCC in conjunction with footprint assembly gives better image quality at higher speed and lower cost than traditional texture mapping.

10 Acknowledgments

This work was done for the MONOGRAPH project, supported by the CEC's ESPRIT Programme. We would like to mention the skill and enthusiasm of Axel Schildan, who made the implementations and helped to make a video sequence.

11 References

[1] **H. C. Andrews and B. R. Hunt,** "*Digital Image Restoration*", Prentice-Hall, 1977

[2] **G. Campbell, T. A. DeFanti, J. Frederiksen, S. A. Joyce, L. A. Leske, J. A. Lindberg and D. J. Sandin,** "*Two Bit/Pixel Full Color Encoding*", SIGGRAPH '86 Conference Proceedings, Computer Graphics, Vol. 20, No. 4, August 1986, pages 215-223

[3] **F. C. Crow,** "*Summed-Area Tables for Texture Mapping*", Proceedings of SIGGRAPH '84, Computer Graphics, Vol. 18, No. 3, July 1984, pages 207-212

[4] **M. F. Deering, S. A. Schlapp and M. G. Lavelle,** "*FBRAM: A new Form of Memory Optimized for 3D Graphics*", Proceedings of SIGGRAPH '94, July 1994, pages 167-174

[5] **E. J. Delp and O. R. Mitchell,** "*Image Compression Using Block Truncation Coding*", IEEE Transactions on Communications, Vol. COM-27, No. 9, Sept. 1979, pages 1335-1342

[6] **A. Glassner,** "*Adaptive Precision in Texture Mapping*", Proceedings of SIGGRAPH '86, Computer Graphics, Vol. 20, No. 4, August 1986, pages 297-306

[7] **P. Heckbert**, *"Color image quantization for frame buffer display"*, Proceedings of SIGGRAPH '82, Computer Graphics, Vol. 16, No. 3, July 1982

[8] **P. Heckbert**, *"Texture Mapping Polygons in Perspective"*, NYIT Computer Graphics Lab Technical Memo #13, April, 1983

[9] **G. Knittel and A. Schilling**, *"Eliminating the Z-Buffer Bottleneck"*, Proceedings of the European Design and Test Conference, Paris, France, March 6-9, 1995, pages 12-16

[10] **G. Knittel, A. Schilling and W. Straßer**, *"GRAMMY: High Performance Graphics Using Graphics Memories"*, in: High Performance Computing for Computer Graphics and Visualisation, Springer-Verlag, London, 1996

[11] **L. Williams**, *"Pyramidal Parametrics"*, Proceedings of SIGGRAPH '83, Computer Graphics, Vol. 17, No. 3, July 1983, pages 1-11

[12] **A. Schilling, G. Knittel and W. Straßer**, *"TEXRAM - A Smart Memory for Texturing"*, Technical Report WSI- 95-12, University of Tübingen, January 1995

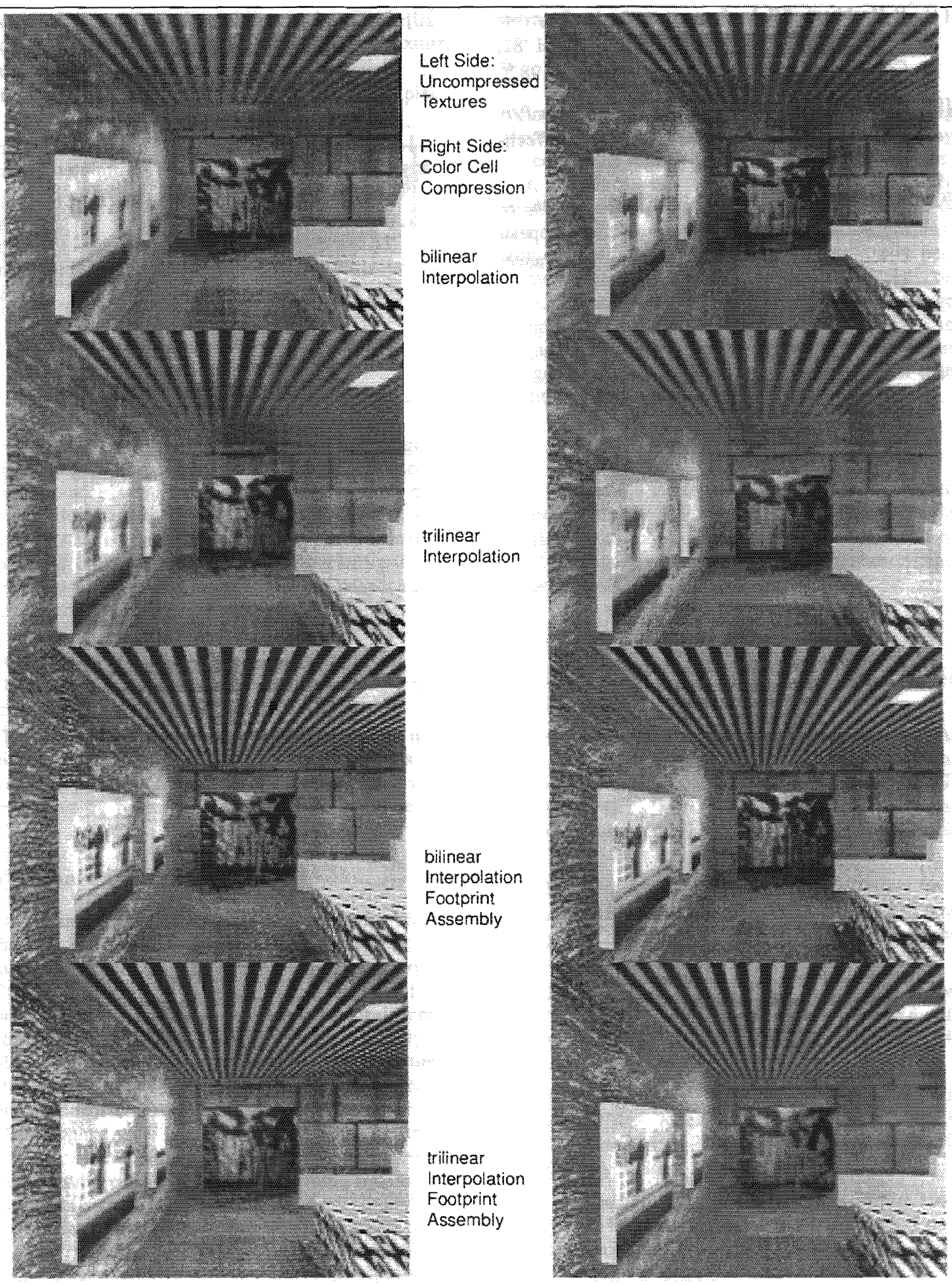
Left Side: Uncompressed Textures
Right Side: Color Cell Compression
bilinear Interpolation
trilinear Interpolation
bilinear Interpolation Footprint Assembly
trilinear Interpolation Footprint Assembly

Figure 9: Texturing Examples