Fast Footprint MIPmapping

Tobias Hüttner, Wolfgang Straßer

WSI/GRIS, University of Tübingen

Abstract

Mapping textures onto surfaces of computer-generated objects is a technique which greatly improves the realism of their appearance. In this paper, we describe a new method for efficient and fast texture filtering to prevent aliasing during texture mapping.

This method, called *Fast Footprint MIPmapping*, is very flexible and can be adapted to the internal bandwidth of a graphics system. It adopts the prefiltered MIPmap data structure of currently available trilinear MIPmapping implementations, but exploits the texels fetched from texture memory in a more optimal manner. Furthermore, like trilinear MIPmapping, fast footprint MIPmapping can easily be realized in hardware.

It is sufficient to fetch only eight texels per textured pixel to achieve a significant improvement over classical trilinear MIPmapping.

CR Categories: I.3 [I.3.3 Picture/Image Generation]: Antialiasing—Bitmap and framebuffer operationsViewing algorithms I.3 [I.3.7 Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture

Keywords: texture mapping

1 INTRODUCTION & PREVIOUS WORK

During the rasterization process, mapping images onto objects can be considered as the problem of determining a screen pixel's projection onto the image (which is usually called *footprint*) and computing an average value which best approximates the correct pixel color. We have adopted the notation of [7] for the following discussion. In real-time environments, where several tens of millions of pixels per second are issued by a fast rasterizing unit, hardware expenses for image mapping become substantial and algorithms must therefore be chosen and adapted 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 $C(x, y)$

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

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

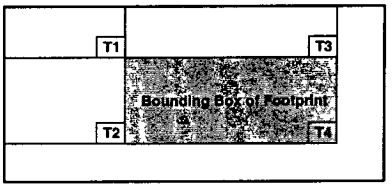

Figure 1: Summed-Area Table.

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

can be excluded from further discussions due to the long computing times. *Summed-area tables* [2] are an attempt to simplify and speed up the above operation by creating an appropriate data structure, the summed-area table, for holding prefiltered data. This data structure is then accessed during rendering. This can be done with different access schemes, for example with a single access per footprint corner, or a 4-access bilinear interpolation per footprint corner. 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.

Given the bounding box of a 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 general case, a potentially large number of texels within the bounding box contributes without reason to the pixel color. Glassner proposes a solution in [3] to incrementally remove rectangles within the bounding box to best approximate the footprint at the cost of increased computing time.

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

1. For each pixel, four accesses must be made that can have very different locations, depending on the bounding box of the footprint. This limits the achievable texturing speed.

2. If the color components are 8-bit quantities, a 1024 × 1024 summed-area table requires entries as wide as 28 bits for each color component.

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 a *MIPmap* as proposed by Williams [9]. 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 2 × 2 texels of level 0. This is continued, until we reach the top-level, which has only one entry holding the average color of the

whole texture. Thus, in a square MIPmap, level n has one fourth of the size of level $n-1$.

The shape of the footprint is assumed to be a square of size q^2, where q is suggested in [4] as

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

In equation (4), u and v denote texture coordinates and x and y are screen coordinates

The MIPmap is accessed by the texture coordinate pair (u, v) of the pixel center and the level λ which in the general case is a function of $\log_2 q$. λ can be expressed with its integer part λ_i and its fractional part λ_f as

$$\begin{aligned} \lambda_i &= \lfloor \log_2 q \rfloor \\ \lambda_f &= \frac{q}{2^{\lambda_i}} - 1 \end{aligned} \tag{5}$$

Nearest-neighbor sampling 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 result in both levels according to λ_f.

Trilinear MIPmapping is a reasonable candidate for a hardware implementation due to its regular access pattern. Due to this, there exist approaches and architectures (for example [7]) to implement this directly into logic-embedded memories. Due to the high costs of chip development and chip productions, these approaches weren't realized for a broad range of systems. Nevertheless, trilinear MIPmapping is the classical filtering approach used over the last decade in graphics systems and it is nowadays available in nearly every PC graphics card. But the approximation of the footprint with a square limits the MIPmap approach severely and people will try to improve it as graphics systems get more powerful on the one hand and the increasing demand for high quality, but cheap visualization on the other hand.

One filtering approach, called *Footprint Assembly*, is described in detail in [7]. Its basic idea is the approximation of the projection of the pixel on the texture by a number N of square mipmapped texels. The pixel's deformation is neglected and it is approximated with a parallelogram given by

$$\mathbf{r_1} = \left[\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x}\right] \text{ and } \mathbf{r_2} = \left[\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y}\right] \tag{6}$$

The pixels center $\mathbf{p}$ in the texture map is the intersection point of the diagonals $\mathbf{d_1}$ and $\mathbf{d_2}$ of the parallelogram. The direction $\mathbf{r}$ in which to step from the pixel center to best approximate the footprint is determined from the larger of the two vectors $\mathbf{r_1}$ and $\mathbf{r_2}$ and

$$\begin{aligned} q &= min(|\mathbf{r_1}|, |\mathbf{r_2}|, \mathbf{d_1}, \mathbf{d_2}) \\ N &= \frac{(max(|\mathbf{r_1}|, |\mathbf{r_2}|)}{q}, \end{aligned} \tag{7}$$

rounded to the nearest power of two as the number of square mipmapped texture elements for the footprint. A difference vector $\Delta\mathbf{r} = (\Delta u, \Delta v)$ is constructed and a sequence of sample points is generated to cover the footprint.

Footprint assembly is able to produce high quality texture filtering, but it has the drawback of being computationally intensive Therefore, [7] proposes a hardware for a logic embedded memory device, which can perform this filtering method during memory access.

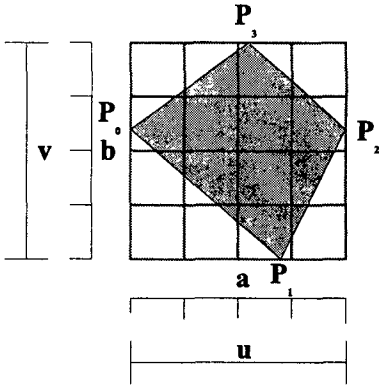

Figure 2. Definition of the footprint

This approach has been adopted in the *TALISMAN* architecture which uses a weighted anisotropic filtering, see [8].

All of these approaches are difficult and costly to be integrated into current rasterizing hardware, since they either require a great amount of computational power or are based on very special system architectures.

It is the goal of this paper to develop a method, which provides a filter quality comparable with footprint assembly but which can be more easily integrated into actual graphics architectures.

2 FAST FOOTPRINT FILTERING

Starting from the classical trilinear MIPmapping, we can easily detect that this filtering method wastes texel information by approximating the footprint by a square, where the footprint is an arbitrary quadrilateral.

Improving filtering means to find a tradeoff between loading more texels to texture a screen pixel and using the loaded texels more efficiently.

The number of texels that can be loaded for real-time filtering is restricted due to strict constants like memory bandwidth or bus width. We will call this limit M. We have to respect this limit, since otherwise system performance will decrease heavily.

Therefore, we will use the MIPmap data structure to be able to precalculate filtering levels. We will access this filtering pyramid with a different approach which we call *Fast Footprint MIPmapping*.

2.1 Calculating a MIPmap level

The first problem we have to solve is depicted in Figure 2 which shows a footprint $[P_0, P_1, P_2, P_3]$. Its bounding box has in texture coordinates the extension (u, v). We want to load a rectangle of $a \times b$ texels from MIPmap level λ to cover the footprint and to respect at the same time the limit M.

For calculating a, b, and λ, we start with the following considerations:

From (u, v), we get the aspect ratio $f = \frac{u}{v}$ of the bounding box. Then we can calculate

$$a \;\; b = M \text{ and } f = \frac{a}{b} \Rightarrow \frac{a^2}{f} = M \Rightarrow a = \sqrt{M \;\; f}. \tag{8}$$

We set

$$a' = a \text{ and } b' = \frac{M}{a} \tag{9}$$

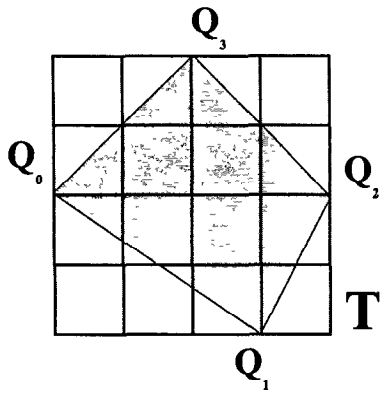

Figure 3: Transformation of corner points to integer positions.

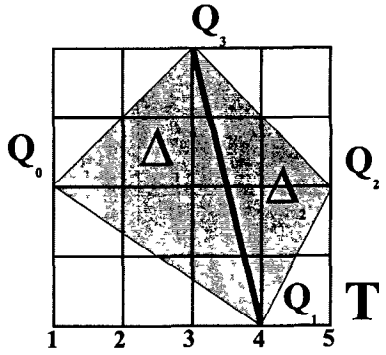

Figure 4: Divide footprint for table lookup.

These values can be used to calculate two MIPmap levels m and n for a' and b'

$$\frac{u}{2^m} = a', \frac{v}{2^n} = b' \Rightarrow m = \frac{\log \frac{u}{a'}}{\log 2}, n = \frac{\log \frac{v}{b'}}{\log 2}. \quad (10)$$

From this, we get λ as

$$\lambda = \lceil \max(m, n) \rceil \quad (11)$$

With this, we know which level we have to access to get the maximum amount of texture information to cover the footprint and to respect M.

2.2 Definition of the weighting table

To do a correct filtering, the contributions of the single texel values to the final pixel value have to be calculated. This is done with a precalculated lookup table, since calculating this on the fly would be too expensive.

We first transform the corner points of the footprint to the integer positions of the texel grid in level λ. This is shown in Figure 3 and generates the quadrilateral $[Q_0, Q_1, Q_2, Q_3]$. Using the orientation of the quadrilateral $[Q_0, Q_1, Q_2, Q_3]$, we can ensure a non empty interior of the transformed points by either adding 0.5 or subtracting 0.5 before snapping to an integer position.

The contribution of a texel to the footprint is now a fixed value. We can calculate for each possible footprint a vector ω consisting of M weights which represent the footprint's coverage for each texel. Computing the weights is a preprocessing step and once it is done, we can store the result in a lookup table. With the help of these weights, a filtering can be performed, since they represent the coverage of the footprint. We store the texels fetched from memory in a linear array T and the weighting vectors in a *weighting table* W.

The filtered pixel value C can then be calculated as

$$C = \sum_{i=1}^{M} (T[i] \cdot (\omega[i])). \quad (12)$$

The weighting vectors allow an easy and efficient computation of the footprint's coverage, but since a footprint is a quadrilateral having four corner points, a huge amount of weighting vectors has to be calculated and stored. For the situation in Figure 3, where we have $M = 16$, precalculating the weighting vectors for all possible footprints would result in $25 \cdot 24 \cdot 23 \cdot 22 = 303,600$ vectors, since we have 4×4 texels, but we have 5×5 possible corner positions.

Each vector has 16 entries and we would have to provide storage for 4,857,600 weights.

By dividing the quadrilateral $[Q_0, Q_1, Q_2, Q_3]$ into two triangles Δ_1 and Δ_2 and filtering each of them separately, we can reduce the number of needed weights significantly, see Figure 4. For the given example, we need only $25 \cdot 24 \cdot 23 = 13,800$ vectors requiring 220,800 weights.

This amount can be reduced even further, if we don't transform the corners to integer positions of the texel grid, but directly to the mid-points of the texels they lie in. This is not as accurate as using the integer positions, since if a corner point of a texel lies on such an integer position, it can be snapped to up to four possible neighboring mid-points. Therefore, this snapping is no longer a unique solution and we have to use a heuristic to ensure a consistent snapping if a corner point is snapped more than once in successive footprints. We choose always the mid-point to the top and the right for corner points on integer positions. With this, aliasing due to inconsistent mid-point snapping can be prevented and the error is in maximum the half of a texel. In the example from Figure 2 are 4×4 mid-point locations possible and we end up with $16 \cdot 15 \cdot 14 \cdot 16 = 53,760$ weights.

The weighting table W is depicted in Figure 5. It is accessed in three stages with a multi-stage lookup structure, one for each corner point. For the example above, the lookup structure consists of $16+16 \cdot 15+16 \cdot 15 \cdot 14 = 3616$ pointers for mid-point snapping. In Figure 5, a single byte value in the range [0..255] represents the weight that will be linearly scaled to [0..1] during the weighting calculation.

Currently, we are numbering the corner positions regularly as depicted in Figure 4. Some combinations of corner points can never occur. There lie always more than two corner points on the borders of T respectively on the mid-points of border texels, since T can be oriented this way when covering the footprint in level λ. By exploiting this fact, the two corner points on the border can be placed at 12 respectively 11 different positions in the example above. We can therefore reduce the amount of necessary weights again to $12 * 11 * 14 * 16 = 29,568$ values. Especially when M is small, we can calculate a whole series of weighting tables in advance for all possible bounding boxes with $a \cdot b = M$. For $M = 16$, the table size needed is 3036 vectors $*16$ weights $= 48,576$ weights for the table with 4×4 texels. The one for 2×8 needs 139,840 weights (2×8 and 8×2 are the same due to symmetry, tables with height or width of only one texel make no sense). With this, we can better approximate elongated and distorted footprints.

Table 1 summarizes the sizes of W and the pointer structures for different values of M. The values are calculated for integer positions (I) and for mid-point snapping (MS). Since in current architectures, M will realistically be restricted to be ≤ 16, we have no space problem with having more than one table, since W and

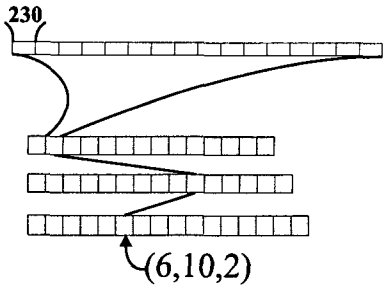

Figure 5: Accessing the weighting table.

M	size of W	(in bytes)	number of pointers
8	(MS 2x4)	8 * 7 * 6 *8=2,688	400
8	(I 3x5)	12*11*13 *8=13,728	2,955
16	(MS 4x4)	12*11*14 *16=29,568	3,136
16	(MS 2x8)	16*15*14 *16=53,760	3,136
16	(I 5x5)	12*11*23 *16=48,576	14,425
16	(I 3x9)	20*19*23 *16=139,840	18,279
32	(MS 2x16)	32*31*30 *32=952,320	30,784
32	(MS 4x8)	20*19*30 *32=364,800	30,784
32	(I 3x17)	36*35*49 *32=1,975,680	127,551
32	(I 5x9)	24*23*49 *32=865,536	87,165
64	(MS 8x8)	28*27*62 *64=2,999,808	254,080
64	(I 9x9)	32*31*79 *64=5,015,552	518,481

Table 1: Sizes of the structures needed for fast footprint MIPmapping.

the pointer structures needed to access W have still feasible sizes. But even if we increase M to 32 or 64, we need approximately 2.5 MB or 7 MB of memory to store W and the pointer structures (one pointer is assumed to be 3 bytes to address 2^{24} possible values). These sizes could be already realized, but eventually they are not economically feasible in current low cost graphic cards. Extrapolating the advances in chip technology that can be seen for example in the rapidly growing size of texture memory on these low cost cards, weighting tables with 32 or 64 can be feasible in the near future. Using the lookup table W, two weighting vectors ω_1 and ω_2 belonging to the triangles Δ_1 and Δ_2 can be generated. The filtered pixel value C can now be calculated as

$$C = \sum_{i=1}^{M} \left(T[i] \quad (\omega_1[i] + \omega_2[i])\right). \tag{13}$$

2.3 HARDWARE REALIZATION

The algorithm shown above can be realized with standard hardware components and is organized in a pipeline having the following successive stages:

- Determination of the weighting vector
  Here we need a multi-stage lookup unit consisting of multiplexers and decoders and a ROM for the vectors. The unit converts the indices of the corner vertices into an access to the ROM table. As already depicted, not all combinations of corner indices can occur. This is coded in the structure and saves memory in the ROM table. We have currently not used the symmetry of triangles covering the weighting mask to further reduce the number of necessary vectors, since this would mean a reordering of the footprint corners that would need additional hardware. We want this design to be stream-line, only consisting of lookups, texel fetches and the final evaluation of the convolution in Equation (13). This ensures speed and can be realized more economically.

- Texel Array T
  The values that are read from texture memory are stored here before they are combined with the weights. The texture memory access itself can be greatly accelerated by using banking and caching techniques, since adjacent footprints have a coherent memory access pattern (see [6]).

- Evaluation of Equation (13)
  This evaluation can be performed with the help of a scalar vector multiplication unit and a second vector unit for calculating the sum of a vector's components.

For our approach, we need no interpolation units, which are necessary for a good quality trilinear MIPmapping. Instead we use lookup tables and a unit which calculates the final pixel value given in equation (13). In our opinion, this hardware effort is comparable to the one needed for trilinear MIPmapping and can also deliver a similar performance, since only basic arithmetic functions are used.

3 RESULTS AND DISCUSSION

We have produced our measurements with a software prototype of the algorithm built into a ray tracing system. Also the other filters, trilinear MIPmapping and footprint assembly, were implemented.

To compare the approaches not only visually, but also statistically. we show in Figure 16, Figure 17 and Figure 18, how our algorithm behaves in selecting MIPmap levels. The pixels are rendered in this ray tracer from the top row to the bottom row. Therefore, we can report, when switches between MIPmap levels occur, since our test scene consists of a textured, flat plane which is sampled with the ray tracer. In these diagrams, the horizontal direction represents the pixel number as the calculation proceeds. In vertical direction, the used MIPmap level is depicted. It turns out, that our method switches earlier to lower levels compared to trilinear MIPmapping, and a bit later than foot print assembly. This is mainly due to the effect explained in Figure 6. Rather distorted footprints extend the bounding box as depicted for the left footprint and we are therefore forced to switch to a higher MIPmap level, but will still sample the footprint correctly with the help of the weighting vectors. It can be clearly seen, that increasing the table size M reduces this behavior and for $M = 16$ and $M = 32$, fast footprint MIPmapping catches up with footprint assembly.

Setting M to 32 is reasonable, since modern graphics chips like the *Riva TNT* chip produced by *NVidia* don't load any longer only the 8 texels necessary for a trilinear MIPmapping. This special chip supports anisotropic filtering and takes up to 8 bilinear samples from up to two adjacent mipmap levels and supports anisotropy of up to 2:1. With this, already 32 texels have to be loaded.

The current implementation has following features:

- Anisotropic filtering is only necessary for a small amount of footprints with heavy distortions. It is therefore possible, to combine trilinear MIPmapping and fast footprint MIPmapping and to use the later one to filter only distorted footprints. This reduces the amount of texture data accessed, since for trilinear MIPmapping, only eight texel values have to be loaded.

- We have currently not a fixed limit M for texel fetching, but we adopt this limit to the footprint characteristics. If footprints with a difficult shape have to be sampled, we raise the size of M up to $2 * M$ which results in a slower sampling due to two steps of fast footprint MIPmapping, but means also improved sampling quality. Footprints, that are more isotropic,

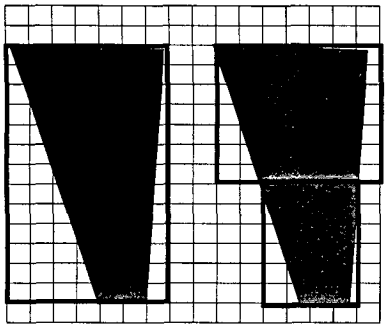

Figure 6: Using two arrays for distorted footprints.

are sampled with smaller tables having less than M texels or they are filtered with trilinear MIPmapping. Furthermore, we use normal bilinear interpolation to access the first level of the MIPmap, if the size of a footprint is smaller than the pixel size at the finest level.

With this we get a better sampling quality without increasing the overhead as much as fixing M on a high level.

We have analyzed this for Figure 9 and the following distribution of texture accesses can be measured:

Pixels to be filtered	275,334
Pixels that can be filtered with trilinear MIPmapping	246,678
Pixels that have to be filtered with Fast Footprint MIPmapping	28,656
Pixels with T between M and $2 * M$	8,836

In Figures 7 - 15, we show the visual behavior of our algorithm compared to the other two. The images are all calculated with a screen resolution of 600 $\times$ 600 pixel. Setting $M = 16$ results in an improvement compared to trilinear MIPmapping, but is still a little bit lower in quality than footprint assembly. $M = 32$ reaches the quality of footprint assembly. This can be clearly seen at the checker board pattern, which has a resolution of 1024 $\times$ 1024 and is therefore a little bit blurry in the foreground due to interpolation, since its resolution is not sufficient in the foreground area.

In Figures 11 - 15, a scene with a map texture having 2048 $\times$ 2048 texels was used. The resolution is sufficient even for the foreground and it turns out, that for such a "real-world" texture which is no artificial test pattern like the checker board, fast footprint MIPmapping with $M = 16$ is sufficient to get a comparable result as with footprint assembly. The difference to $M = 32$ can only be seen in a difference image. Nevertheless, even with fixing M to eight texels we get a significant improvement compared to trilinear MIPmapping in terms of the image being less blurred, see Figure 13. Eight texels is the amount of texture information which has to be fetched for the actual trilinear MIPmapping.

It is important to mention the smooth, not visible transition between the MIPmap levels without interpolating between MIPmap levels as it is done using trilinear MIPmapping. This is necessary to prevent aliasing during animation. We have also confirmed this by calculating animations for the checkerboard scene showing smooth transitions between the frames without aliasing. In figures 19 - 21, the histogram of the difference image of an epsilon change in camera position is shown (the intensity is scaled logarithmically). The camera was viewing the checker board scene in a diagonal direction which generated rather anisotropic footprints. In the histograms of fast footprint MIPmapping, no peaks due to aliasing can be found in the right area of the histogram. Furthermore, the extension of the non-zero values is more or less the same for trilinear MIPmapping and fast footprint MIPmapping. Therefore, we can claim frame-to-frame coherence.

4 CONCLUSIONS AND FUTURE WORK

We have presented a new approach for texture filtering to prevent aliasing during texture mapping. In contrast to classical approaches, our method exploits the texels fetched from texture memory in a more optimal way but still remains feasible for implementation in hardware. Furthermore, it is scalable to respect the internal bandwidth of a graphics system.

The next step concerning this interesting project will be to enhance the filtering quality further. Currently, we investigate, how to access not only one MIPmap level, but to sample the footprint with a number of independent and smaller arrays on different levels of the MIPmap. This seems to be especially useful for pixels which have extended footprints. We can further reduce the loading of texels which are not needed, but contained in the loaded texel rectangle, if we adopt T better to the shape of the footprint, see Figure 6. On the other hand, this will cost additional hardware and introduce latency, since the footprint has to be divided temporarily. Doing this is therefore a tradeoff decision between the cost of the fast footprint structure dictating how much weighting tables and in which size can be realized, the bandwidth of texture memory, and additional costs and latency introduced by footprint subdivision.

Another approach will be to use a compression scheme for storing the weighting table. On one hand, we can further use symmetry arguments to reduce the number of vectors. On the other hand, compressing the vectors themselves is also possible. Furthermore, we consider building a hardware prototype implementation to verify the algorithm not only as a software prototype.

Acknowledgements

We want to thank Ralf Sonntag for supporting us with his *RadioLab* system to produce the images. Furthermore, we would like to thank Andreas Schilling for our interesting and fruitful discussions about this topic.

References

[1] ANDREWS, H C , AND HUNT, B R. Digital image restoration, 1977

[2] CROW, F. C Summed-area tables for texture mapping. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (July 1984), H Christiansen, Ed , vol 18, pp. 207–212.

[3] GLASSNER, A Adaptive precision in texture mapping In *Computer Graphics (SIGGRAPH '86 Proceedings)* (August 1986), D C Evans and R. J. Athay, Eds , vol 20, pp. 297–306

[4] HECKBERT, P S. Texture mapping polygons in perspective. TM 13, NYIT Computer Graphics Lab, April 1983

[5] HECKBERT, P. S. Survey of texture mapping *IEEE Computer Graphics and Applications* (Nov 1986), 56–67

[6] IGEHY, H , ELDRIDGE, M , AND PROUDFOOT, K Prefetching in a texture cache architecture In *Eurographics/SIGGRAPH Hardware Workshop '98 Proceedings)* (1998)

[7] SCHILLING, A , KNITTEL, G , AND STRASSER, W. Texram: A smart memory for texturing *IEEE Computer Graphics & Applications 16*, 3 (May 1996), 32–41.

[8] TORBORG, J , AND KAJIYA, J Talisman. Commodity Real-time 3D graphics for the PC In *SIGGRAPH 96 Conference Proceedings* (Aug 1996), H Rushmeier, Ed , Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 353–364 held in New Orleans, Louisiana, 04-09 August 1996

[9] WILLIAMS, L Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)* (July 1983), pp 1–11

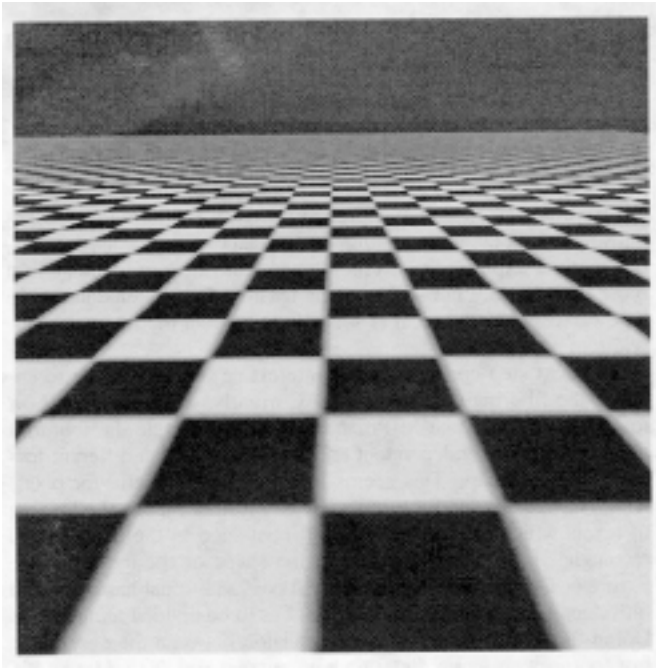

Figure 7: Trilinear MIPmapping.

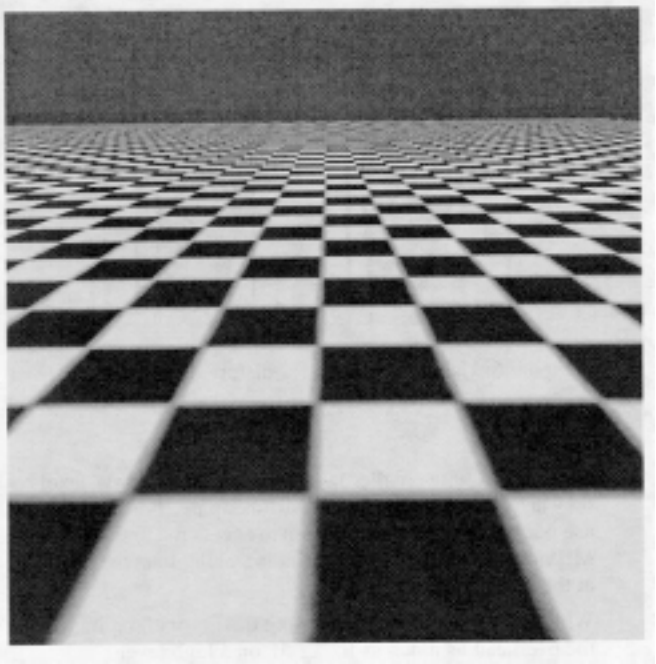

Figure 9: Fast footprint MIPmapping using $M = 16$.

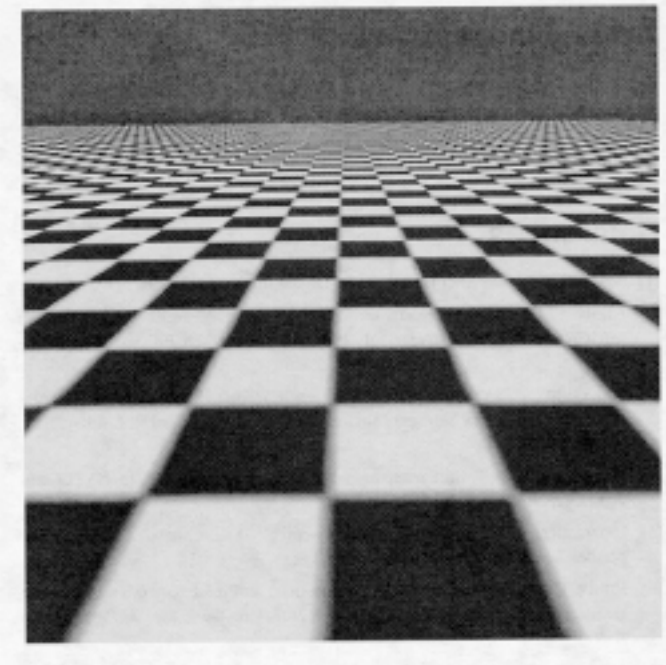

Figure 8: Footprint assembly.

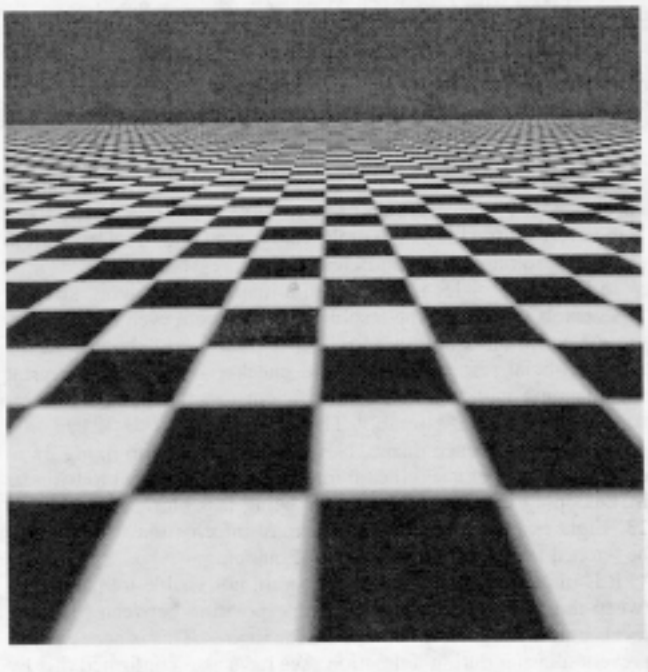

Figure 10: Fast footprint MIPmapping using $M = 32$.

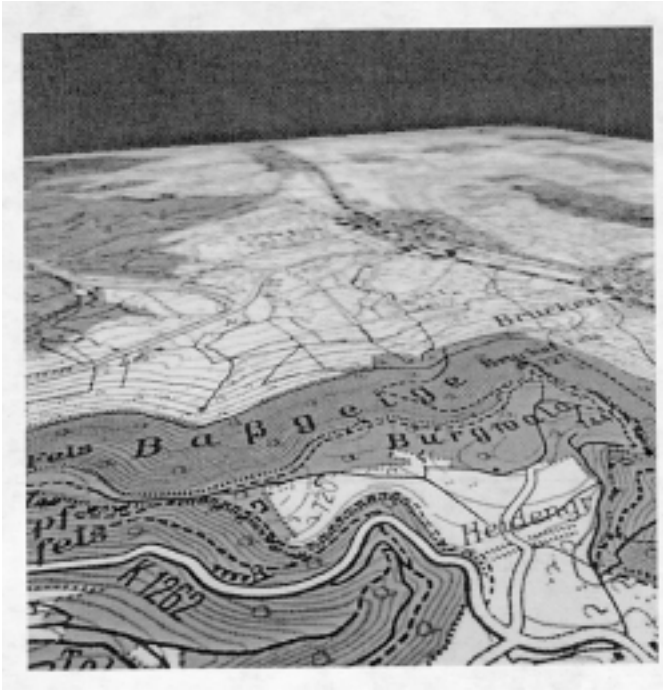

Figure 11: Trilinear MIPmapping.

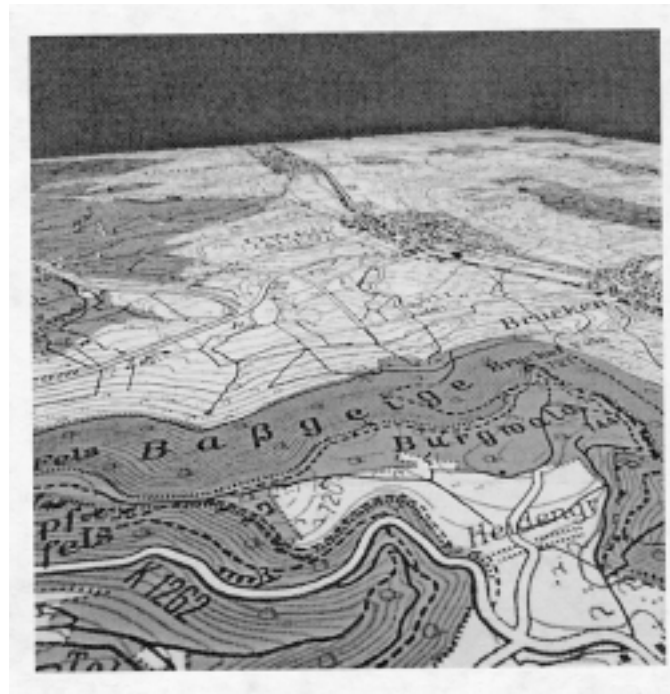

Figure 12: Footprint assembly.

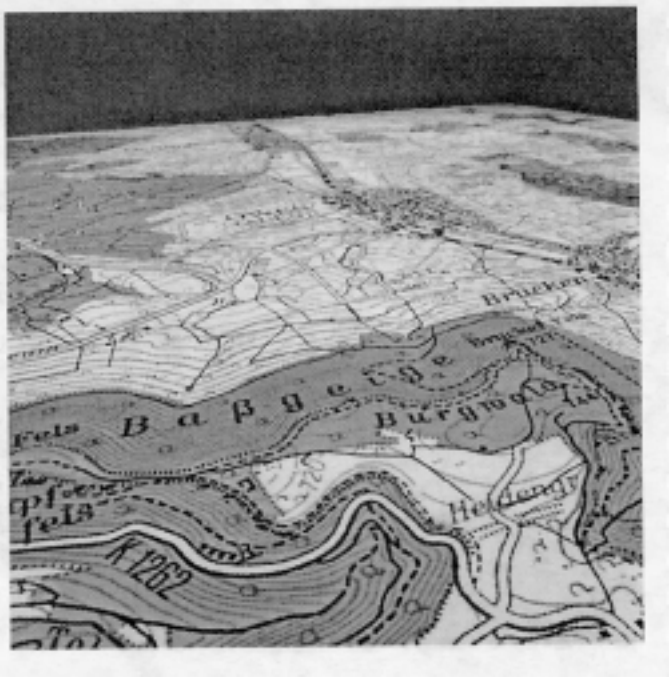

Figure 13: Fast footprint MIPmapping using $M = 8$.

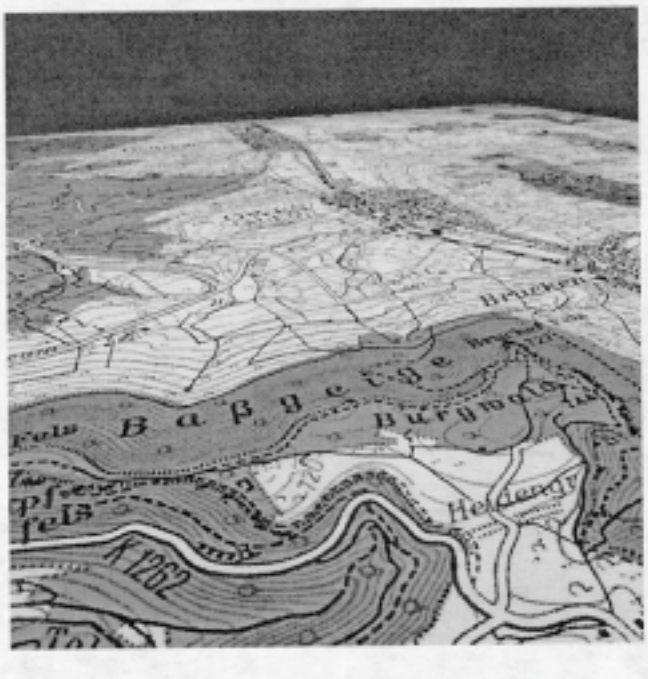

Figure 14: Fast footprint MIPmapping using $M = 16$.

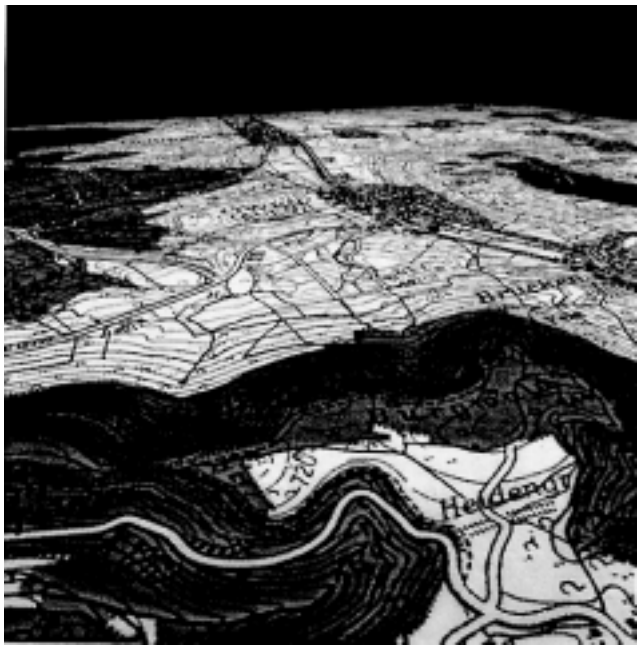

Figure 15: Fast footprint MIPmapping using $M = 32$.

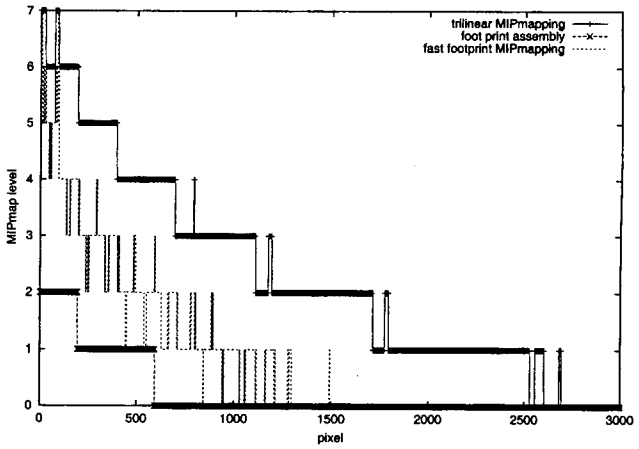

Figure 17: Selected MIPmap levels, $M = 16$.

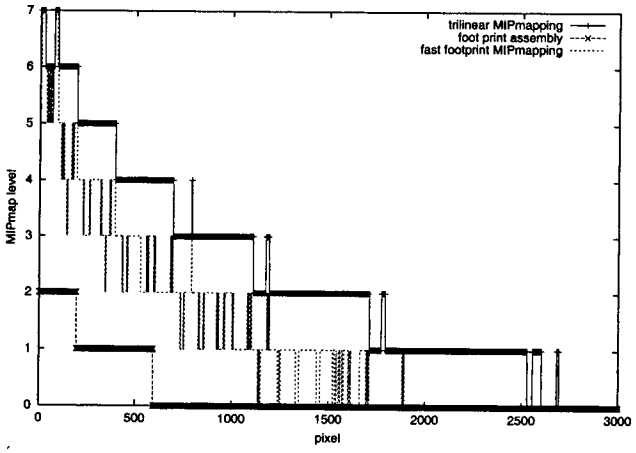

Figure 16: Selected MIPmap levels, $M = 8$.

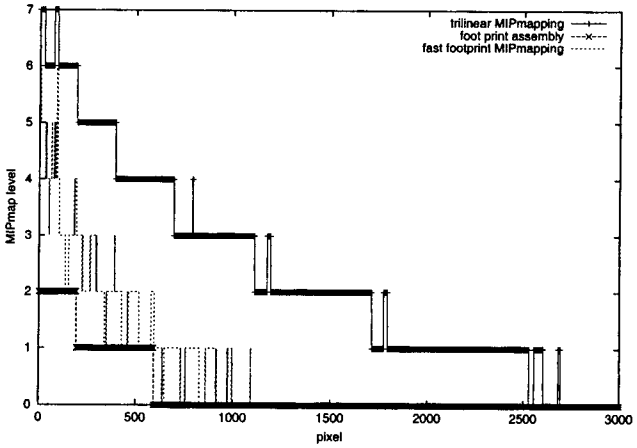

Figure 18: Selected MIPmap levels, $M = 32$.

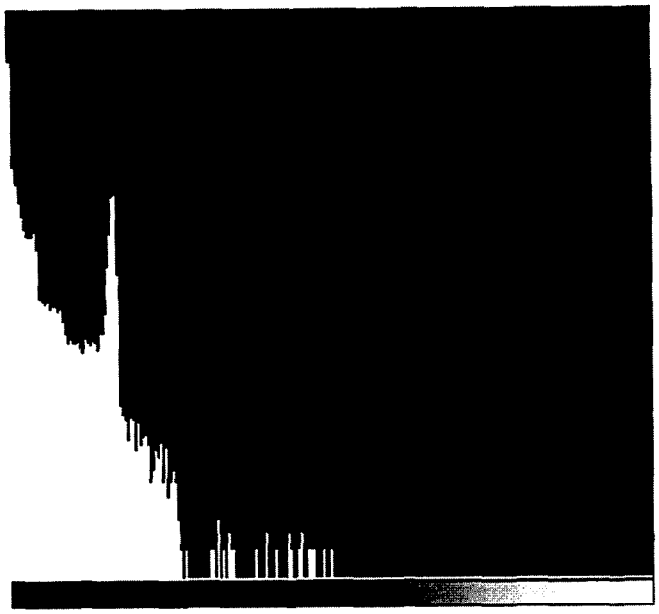

Figure 19: Histogram of trilinear MIPmapping.

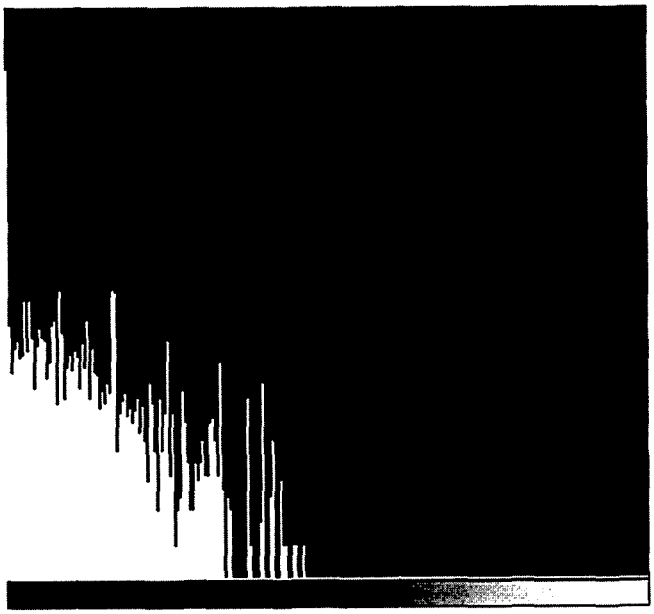

Figure 20: Histogram of fast footprint MIPmapping, $M = 8$.

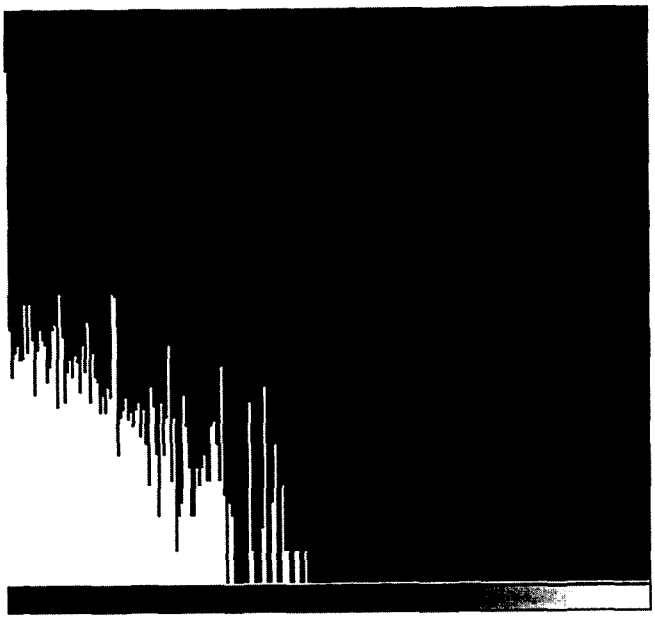

Figure 21: Histogram of fast footprint MIPmapping, $M = 16$.

Figure 11 MIPmap filtering

Figure 13 Fast footprint filtering using $M = 8$

Figure 12 Footprint assembly

Figure 14 Fast footprint filtering using $M = 16$