Graphics Hardware (2002)
Thomas Ertl, Wolfgang Heindrich and Michael Doggett (Editors)

Resample Hardware for 3D Graphics

Koen Meinds and Bart Barenbrug

Philips Research, Eindhoven, The Netherlands

Abstract

Texture mapping is a core technology of current real-time 3D graphics systems. To avoid aliasing artifacts, the texture mapping resample process requires proper filtering. We present a new resample algorithm for two-pass forward texture mapping that is suited to an efficient hardware implementation. This method delivers high quality anti-aliased images using filter techniques based on digital signal processing. We use an input sample driven texture resample and filtering algorithm that "splats" the contribution of each input sample (texel) to output samples (pixels). We show how the algorithm can be efficiently implemented in a hardware resample structure. The algorithm is incorporated and tested in a standard 3D graphics pipeline using the OpenGL interface. Our results exhibit better anti-aliasing of textures than anisotropic filtering found in current advanced graphics chips. We also show that the same texture filtering method can be used to implement edge anti-aliasing. Our edge anti-aliasing results show an absence of aliasing on most edges.

Categories and Subject Descriptors: *I.3.1 Graphics processors, I.3.3 Anti-aliasing, I.3.7 Texture Mapping*

1 Introduction

Almost all current texturing hardware is based on the principle of inverse texture mapping. For real-time systems, bi- or trilinear MIP-map interpolation is the de facto standard. Its main advantage is that it is simple to implement in hardware. Its disadvantages (poor quality, blurry effects) have been partially addressed by the introduction of anisotropic filtering. These filtering techniques require clever caching techniques [11] in order to reduce bandwidth demands. However, with all these algorithmic and architectural efforts, texture mapping in today's real-time systems exhibits annoying aliasing artifacts as is discussed by Kirk[16].

With *forward texture mapping*, texels are mapped to screen space. Forward mapping is sometimes being viewed as being unsuited for real time hardware acceleration. In this paper we introduce a novel texture filter algorithm and hardware structure based on two-pass forward texture mapping [5] that avoids a lot of the drawbacks commonly associated with forward texture mapping. We show how forward texture mapping can be implemented in real time hardware accelerators that generates high quality images which are both texture and polygon edge anti-aliased at less costs compared to inverse texture mapping with super sampling. In order to test functional correctness and to be able to generate images, we have implemented our algorithms in a 3D graphics pipeline using the OpenGL interface.

We use a texel driven resample algorithm that "splats" the color of a texel onto the surrounding pixels of the mapped texel. The image quality resulting from our texture filtering method is higher than the current state-of-the-art anisotropic filtering that uses a maximum of 8 trilinear probes and can be found in commercially available advanced graphics accelerators, whereas computational costs are roughly the same. Comparing our method, using animated sequences, shows absence of texture aliasing artifacts while preserving sharpness, whereas aliasing artifacts are visible with anisotropic filtering.

Besides texture aliasing, that relates to the interior of polygons, there is also aliasing that relates to the edges of polygons. This *edge aliasing* may appear in the form of staircase jaggies. Although texture aliasing and edge aliasing may have different appearances, from a filter theoretical point of view, there is no reason to distinguish between the two: both types of artifacts are due to high frequencies present in the rendered scene that can not be displayed by a discrete screen.

Super sampling is the traditional solution for the edge aliasing problem in consumer 3D graphics accelerators. Although super sampling can be used for both edge anti-aliasing and texture anti-aliasing, in practice, it is mainly used for edge anti-aliasing only. It is expensive to use super sampling for texture anti-aliasing because it requires a high resolution texture MIP-map level that is suited to calculate color values on the (higher resolution) subsample grid. The use of such a higher resolution texture MIP-map level *increases the required texture memory bandwidth*. For edge anti-aliasing the higher resolution texture map is not required. Multi-sampling [13], a variant of super sampling, can only help to reduce edge aliasing. It explicitly focuses on a

koen.meinds@philips.com, bart.barenbrug@philips.com

low texture bandwidth by determining only a single color value for a group of sub-pixels.

Our solution applies a single filtering method to suppress high frequencies induced by both texture mapping and polygon edges. Our edge anti-aliasing quality is better than 4x4 super sampling. Whereas the required off-chip memory bandwidth and computational costs are roughly comparable with 2x2 super sampling.

Our contribution

Our main contributions of the presented two-pass forward texture mapping work are:

- We have developed a novel discretely computable resample process that combines box reconstruction filtering and high order prefiltering and does not suffer from DC-ripple: no post-processing intensity normalization pass is required.
- We have designed an efficient hardware resample structure that implements our discrete resample process.
- We use two-pass forward texture mapping with a couple of lines accumulation buffer memory. No off-chip frame size accumulation buffer that requires high memory bandwidth is used, as is needed for one-pass forward texture mapping with a 2D filter kernel.
- Rasterization takes place in texture space using a variable resolution MIP-map texture grid, to keep required texture bandwidth low.
- We have added a pixel fragment buffer to support high quality edge anti-aliasing.

We do not claim that forward texture mapping can easily be used to support all the features of the most recent 3D graphics API's. Features that have been developed on the inverse texture mapping architecture, such as dependent multi-texturing (e.g. as used for bump mapping), may not have an easy match on a forward texture mapping pipeline. But independent multi-texturing (or pixel shading) may be implemented using multipass texturing. A 3D pipeline implementation without multiple texturing or pixel shading can still be very interesting to various application domains such as "mobile". For shading in screen space perspective correction is required. For example, Gouraud shading using bilinear interpolation in screen space results in minor artifacts [23]. However, procedural *pixel* shading (such as to generate marble patterns) in screen space without perspective correction, may result in *unacceptable* artifacts. It is therefore better to use procedural *texel* shading.

Related work

By means of forward texture mapping, the process of selecting and weighting texels becomes much easier. This has been shown by Ghazanfarpour and Peroche [10] where they describe one-pass forward texture mappping. Catmull and Smith [5] introduced the two-pass forward texture mapping method, that uses two orthogonal 1D resample passes, and has a "scanline" order texture memory access, where they generalize the meaning of scanline to also include vertical "scanline". Fant [9] describes a specific two-pass implementation that uses first order (box-filter) like filtering, from which it is known [2] that it delivers poor quality. The Ampex video spatial effects system uses a two pass image transformation and can perform, amongst others, perspective transformations in real-time on video streams. However, from a patent[1] it can be seen that at each 1D resampling pass they apply inverse texture mapping with a prefilter mapped to input space. In the "Forward Image Mapping" paper [6] a one-pass forward mapping technique is presented that does not require a perspective division per pixel. The "Surface Splatting" paper [24] uses a one-pass forward mapping technique for point primitives, combined with an A-buffer like pixel fragment buffer for transparency and edge anti-aliasing. The papers [6, 10, 24] use an intensity normalization post-processing pass requiring a division per pixel.

Paper organization

In Section 2 we will argue that it is easier to obtain high quality images using forward texture mapping with prefiltering in screen space than with inverse texture mapping with a *mapped* prefilter in texture space. In Section 3, we present our discrete approximation of the texture resample process. In Section 4 we present our resample hardware structure derived from our discrete resample approximation. Section 5 shows how two of our 1D resample structures are merged into a 2D resample unit. Section 6 details how we drive the 2D resample unit with our texture space rasterizer. Section 7 describes how we combine fragments in the fragment buffer to obtain edge anti-aliasing. Images generated by our implementation are discussed and compared with competitive methods in Section 8. Cost factors are compared in Section 9 and the conclusions are given in Section 10.

2 Resample Process

The perspective transformation of a texture map is a two dimensional resampling process: it maps a sampled input image (the texture map) onto a sampled output image (the screen). The ideal resampling process consists of four steps [12, 20]:

1. Use a reconstruction filter to construct a continuous signal from the discrete input;
2. Transform the reconstructed input signal;
3. Prefilter the transformed signal to bandlimit it to the half of the output sample rate;
4. Sample the filtered signal to produce the discrete output.

Most of the texture mapping in a typical 3D scene results in minification (i.e. the number of output samples is smaller than the number of input samples). In such cases, the prefilter that attenuates high frequencies caused by minification, has more influence on the quality of the resulting image than the reconstruction filter. So for minification, a high quality prefilter is desired whereas the reconstruction filter is less important. In the case of magnification the reconstruction filter dominates the prefilter. Minification is the more critical case because it can generate high frequencies that can give rise to annoying aliasing artifacts.

In Section 3 we present our discretization of the resample process. To present this in its proper context we start

with reviewing the inverse texture mapping process and the forward texture mapping process in the next two subsections.

2.1 Inverse texture mapping

Real-time 3D graphics systems traditionally use inverse texture mapping. In this section we discuss some difficulties with inverse texture mapping that hamper proper filtering. With inverse texture mapping a pixel's prefilter (in screen space) is mapped onto input samples (texels). The following schema illustrates the filtering process in the backend of an inverse texture mapping graphics pipeline.

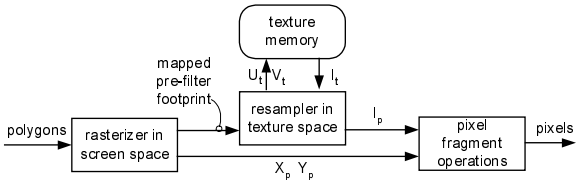

Figure 1: *backend of inverse texture mapping pipeline*

(U_t , V_t) is the texture coordinate of a texel with index t. (X_p , Y_p) is the screen coordinate of pixel with index p . I_t is the color of texel t and I_p is the filtered color of pixel p .

Rasterization of the polygon takes place in screen space. For every pixel traversed, its prefilter (footprint and profile) is mapped into the texture space. The texels within the mapped footprint must be determined and weighted according to the *mapped* profile. The pixel color is computed using the mapped prefilter in *texture space*. The process is *output sample (pixel) driven*.

The next figure illustrates that a square prefilter footprint mapped to texture space results in an arbitrary quadrilateral.

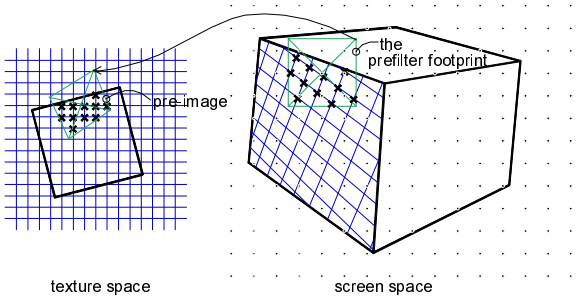

Figure 2: *only texels within the quadrilateral mapped prefilter footprint and within the polygon contribute.*

To determine the texels that fall within the arbitrary quadrilateral is not an easy task. Moreover, only texels confined to the mapped polygon being rasterized should be selected (see the left side of Figure 2). Using the required *mapped* prefilter function to weigh the selected texels complicates matters further. Inverse texture mapping methods, like trilinear filtering and anisotropic filtering such as footprint assembly [19], Feline [17] and Potential MIP mapping [3] can be seen as approximations of this texel selection and weighting process.

2.2 Forward texture mapping

The opposite of inverse texture mapping is forward texture mapping: the texel's reconstruction filter is mapped onto the output pixels. The backend of a forward texture mapping graphics pipeline with filtering is shown in the next figure:

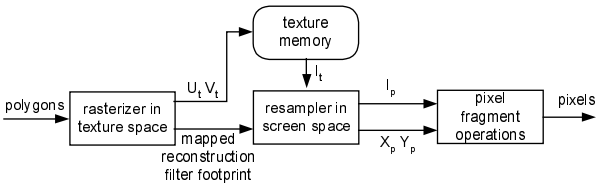

Figure 3: *backend of forward texture mapping pipeline*

Rasterization is done in texture space. Texels confined to the polygon being rasterized are splat on pixels in screen space. The coordinates of these texels are mapped to screen space and all pixels whose prefilter footprint covers such a mapped texel are contributed according to this prefilter.

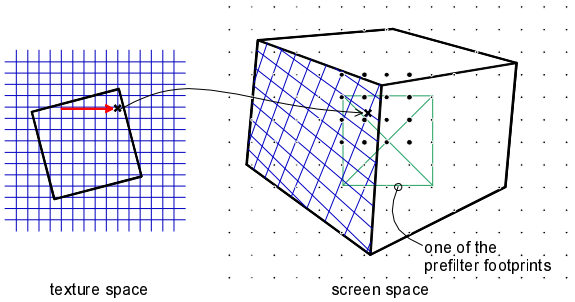

Figure 4: *texels are traversed in texture space and splat to pixels in screen space*

Using this method, the resampling from texel colors to pixel colors, takes place in *screen space* and is *input sample (texel) driven*. We will now argue that this process simplifies proper filtering significantly.

Compared to the inverse texture mapping, firstly, it is easier to determine which texels contribute to a particular pixel: it is simple to determine if a texel coordinate, mapped in screen space, is within the *axis aligned square prefilter footprint* of a pixel. Secondly, contrary to inverse texture mapping, there is no need to transform the filter function from pixel space to texture space. Finally, due to rasterization in texture space, only texels restricted to the polygon are considered for the filtering process.

3 Resample Discretization

Our resampling algorithm is based on the two-pass forward texture mapping technique presented by Catmull and Smith[5]. They have demonstrated how a 2D perspective transformation can be decomposed into two orthogonal 1D resampling passes. In this section we present our novel discretization for such a 1D resampling process, but first we review a common resample discretization and show why it is inappropriate for texture mapping. We now limit ourselves to texture minification (down sampling). In Section 6 we will explain how we handle magnification.

3.1 Discretization with DC-ripple

A common 1D discretization of the resampling process of Section 2, known from digital signal processing [7, 8, 14] and suited for spatial variable resampling such as perspective transformation, uses a zero-order reconstruction filter (dirac function). It is described by the formula:

$$I_p = \sum_t h(X_t - X_p)\ f_t\ I_t \qquad (1)$$

X_t is the mapped texel coordinate U_t. h is the prefilter function. f_t is the *local scaling factor* of texel t and is equal to the length of a to screen space mapped unit texel spacing around texel t. It is used to normalize the intensity of the calculated pixels. f_t varies per texel due to the perspective transformation applied. $f_t \cdot I_t$ is the reconstructed signal by the dirac reconstruction filter.

The aforementioned resampling process suffers from an artifact called *DC-ripple* [14], also known as sample-frequency ripple. DC-ripple is visible as intensity fluctuations on the output signal while the input signal has a constant intensity. DC-ripple is most visible with small filter footprints and scale factors close to one (thus low minification).

Figure 5: *Constant intensity input results in DC-ripple at output. Minification with fixed scale factor 0.9 using a tent filter.*

We realized that DC-ripple is due to the fact that the sum of the weights ($h(X_t - X_p) \cdot f_t$) of the texels is spatial varying (depending upon $X_t - X_p$). This is illustrated in the following figure, where the weight of texel t has been graphically represented by the hatched area:

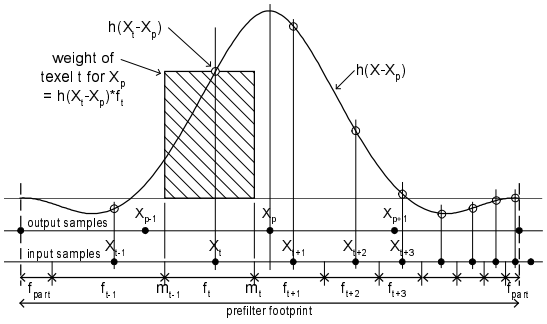

Figure 6: *traditional weight of a texel*

The hatched area can be considered to be an *approximation* of the integral of h between m_{t-1} and m_t. Hence, the sum of the signed area's of all texels within the prefilter footprint is an approximation of the definite integral of the filter function h. This approximation varies for different pixel positions and is the source of the DC-ripple artifact. To avoid DC-ripple we should make sure that the weights of all texels covered by the prefilter footprint sum to the definite integral over the footprint of h, which is a constant.

3.2 Discretization without DC-ripple

To avoid the DC-ripple artifact, we apply a first order (box) reconstruction filter, instead of the zero-order (dirac function times scale) reconstruction filter as used in eq. (1).

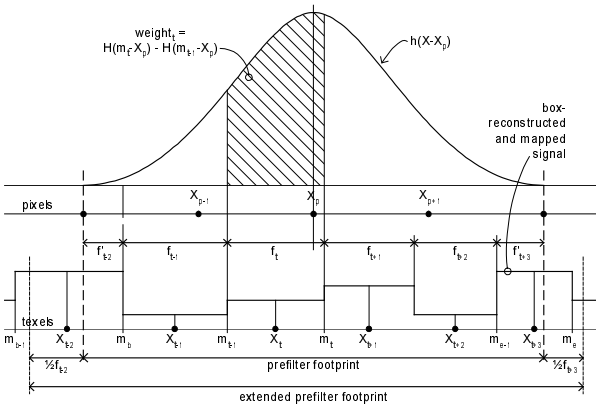

Figure 7: *box reconstruction filter applied to input samples*

The box reconstructed texture, mapped to screen space, is a piece-wise constant signal. We recognized that, for such a signal, exact convolution with the prefilter function h becomes easy. The continuous convolution process with a box filter can be written as:

$$I_p = \int_{-\infty}^{\infty} h(\tau - X_p)\ mbox(\tau) d\tau \qquad (2)$$

With *mbox* being the (piece-wise constant) mapped box-reconstructed input signal: $mbox(\tau) = I_t$ for $m_{t-1} < \tau < m_t$, with m_t being the midpoints of the mapped texels. Because $mbox(\tau)$ has a constant value between two midpoints we can rewrite the above formula as follows:

$$I_p = \sum_{t=-\infty}^{t=\infty} \left(\int_{m_{t-1}}^{m_t} h(\tau - X_p)\ d\tau \right)\ I_t \qquad (3)$$

H being the indefinite integral of h, the last formula may be rewritten to the following discrete computable form:

$$I_p = \sum_{t=b}^{e} (H(m_t - X_p) - H(m_{t-1} - X_p))\ I_t \qquad (4)$$

with b being the left most texel and e the right most texel contributing to I_p. Note that the weight term $H(m_t-X_p) - H(m_{t-1}-X_p)$ calculates the exact area below the prefilter profile between m_{t-1} and m_t in Figure 7, and that the sum of all these weight terms is equal to definite prefilter integral, and thus constant (independent of pixel position X_p). For texture mapping, intensity amplification is undesired: the definite integral over the footprint of h must be *one*. Note that $H(m_{b-1}-X_p) = 0$ and $H(m_e-X_p) = 1$.

Note that a texel within $\frac{1}{2}f_t$ distance to the prefilter footprint boundary (inside and outside the prefilter footprint) will contribute partially to I_p. See Figure 7 where the texels at location X_{t-2} and X_{t+3} are partially contributing.

Traditionally, DC-ripple in input driven resample structures based on (1) is alleviated for a *fixed (or a set of fixed) scaling ratio(s)*, by "fine-tuning" the tabulated filter function values. This does not work for a stepless spatial vari-

able scale factor, as is the case for perspective transformations. For input driven resampling processes with stepless variable scale factors often [6, 10, 24] a per pixel intensity normalization is performed by means of a post-processing pass that *divides each pixel fragment* color, stored in a screen space buffer, by the sum of the filter weights. This will also remove DC-ripple. The normalization post-processing pass *increases (off-chip) memory bandwidth*. It also requires a higher accuracy color representation than required for the final pixel color in order to avoid overflow before the division. Moreover, due to the intensity normalization, the prefilter function is not clearly defined: The amplitude of the prefilter function varies per pixel. Our resample process does not require such a normalization post-processing pass.

3.3 Resampling and Polygon Edges

The resample discretization (4) can be used for accurate prefiltering when the prefilter footprint is intersected by a polygon edge: texels less than $\frac{1}{2}f_t$ distance from the edge should only contribute for the box reconstructed constant signal *within* the polygon. This can easily be achieved by clipping the m_{t-1} and m_t values, before applying formula (4), against the left and right edge coordinates of the polygon. We call these coordinates *xL* and *xR* respectively.

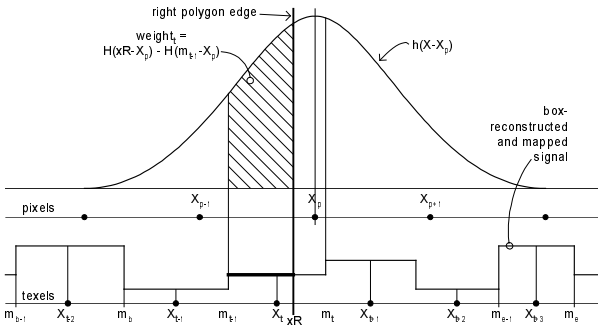

Figure 8: *clipping against right polygon edge xR.*

In the example shown by Figure 8 the weight of texel t is adapted to correspond to *the reconstructed part within* the polygon. Both m_{t-1} and m_t are clipped but only the value of m_t is actually changed.

In our system we use this accurate way of contributing texels, with *partial* weights, for high quality edge anti-aliasing as described in Section 7. We will call pixels whose prefilter foot print is intersected with a polygon edge *partial pixels*. We now introduce the so-called contribution factor C_p, that is needed by our edge anti-aliasing algorithm and is generated by the resample structure of Section 4. The contribution factor C_p, for a pixel, is the sum of the texel weight terms ($H(m_t-X_p) - H(m_{t-1}-X_p)$ of equation (4)) and is 1, if all texels within the prefilter footprint are contributing. If a polygon edge prohibits all texels within the prefilter footprint to contribute, C_p will be between 0 and 1, for a fully positive prefilter profile.

4 Resample Structure

In this section we present a novel 1D resample structure that implements equation (4) as a so-called polyphase transposed direct-form structure [7]. It efficiently integrates the box reconstruction filtering and prefiltering into a single structure.

The resample equation (4) can be implemented in two ways: output sample (pixel) driven and input sample (texel) driven. The method suggested by equation (4) is *output driven*.

Our resample structure is *input driven*: we traverse the texels from left to right, and for each texel t its color is *"splatted"* onto a group of pixels. Pixel coordinate X_p is being defined closest to X_t and to the left: $X_p = \lfloor X_t \rfloor$. Let us consider a prefilter width of 4. For other widths the construction of the resample structure is analogous. We observe that the group of pixels that can be contributed is: *p-2* to *p+2*. There are two cases: in the first case X_p is to the left of m_{t-1} and in the second case X_p lays on interval $[m_{t-1}, m_t]$. The first case is illustrated in Figure 9. In this case the prefilter footprint of *p-2* does not overlap $[m_{t-1}, m_t]$ and only 4 pixels (*p-1* to *p+2*) receive a contribution from texel t. In the second case 5 pixels (*p-2* to *p+2*) receive a contribution. In this case, due to processing texels from left to right, *p-2* will get its last contribution and can be outputted to a next stage in the pipeline. We will call the latter case the "*step*" case and the first case the *"non-step*" case.

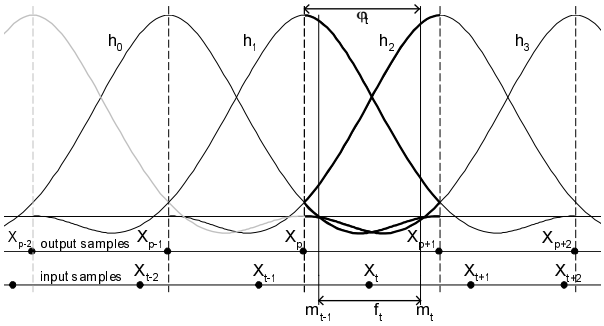

Figure 9: *Contribution of a texel to 4 pixels. Non-step case. h_0 to h_3 corresponds to their integrals H_0 to H_3.*

In both cases, we have to evaluate $H(x)$ for $x = m_t-X_{p-1}$, m_t-X_p, m_t-X_{p+1} and m_t-X_{p+2} to splat the color of t. Note that the *step* case, when 5 pixels get a contribution, does not require more terms to be evaluated because then: $H(m_t-X_{p-2}) = 1$ and $H(m_{t-1}-X_{p+2}) = 0$. Instead of using one H table that has to be indexed with 4 different values, we use 4 tables (H_0 to H_3), all indexed by the same value:

$$H(m_t-X_{p-1}) = H(m_t-X_p+1) = H_0(m_t-X_p)$$
$$H(m_t-X_p) = H(m_t-X_p) = H_1(m_t-X_p)$$
$$H(m_t-X_{p+1}) = H(m_t-X_p-1) = H_2(m_t-X_p)$$
$$H(m_t-X_{p+2}) = H(m_t-X_p-2) = H_3(m_t-X_p)$$

The domain of H is [-2, 2]. The domain of H_0 to H_3 is [0, 1]. H_0 is equal to the last quarter of H and to H_3 is equal to the first quarter of H, see the bold parts of the graphs in Figure 9. We will call $\varphi_t = m_t-X_p$ the *phase*. Because p is the closest pixel left to m_t the phase is equal to the fractional part of m_t.

An algorithm that implements this process is illustrated in Figure 10. The algorithm cycles over the texels, and for each texel its contributions to all the involved pixels is accumulated before the next texel is processed.

```
1    initialization of: m_{t-1} and I_{p-1} to I_{p+2}
2    input per texel: I_t and m_t
3    φ_{t-1}  = frac(m_{t-1})
4    φ_t      = frac(m_t)
5    step = int(m_t) - int(m_{t-1})
6    if step == 1
7      X_{p-2} = int(m_t) - 2
8      I_{p-2} = I_{p-1} + (    1    - H[0][φ_{t-1}]) * I_t
9      I_{p-1} = I_p   + (H[0][φ_t] - H[1][φ_{t-1}]) * I_t
10     I_p     = I_{p+1} + (H[1][φ_t] - H[2][φ_{t-1}]) * I_t
11     I_{p+1} = I_{p+2} + (H[2][φ_t] - H[3][φ_{t-1}]) * I_t
12     I_{p+2} = 0     + (H[3][φ_t] -     0     ) * I_t
13     output per pixel: I_{p-2} and X_{p-2}
14   else /* step == 0 */
15     I_{p-1} = I_{p-1} + (H[0][φ_t] - H[0][φ_{t-1}]) * I_t
16     I_p     = I_p   + (H[1][φ_t] - H[1][φ_{t-1}]) * I_t
17     I_{p+1} = I_{p+1} + (H[2][φ_t] - H[2][φ_{t-1}]) * I_t
18     I_{p+2} = I_{p+2} + (H[3][φ_t] - H[3][φ_{t-1}]) * I_t
19   m_{t-1} = m_t
```

Figure 10: *pseudo code for input driven calculation of equation (4), for a prefilter width of 4.*

For every processed texel t, its color I_t and its right mapped midpoint m_t clipped to the right polygon edge as described in Section 3.3, are input for the algorithm. In the *step* case the pixel color I_{p-2} and coordinate X_{p-2} are outputted. Lines 8 to 12 represent a combination of a texel contribution to 5 pixels, and a shift between the pixel color variables (I_{p-2} to I_{p+2}). In line 5, the step or non-step case is determined. Note that `int(m_{t-1})` is one less than `int(m_t)` only when X_p is to the left of m_{t-1}. It can not differ more than one because we limited ourselves to minification.

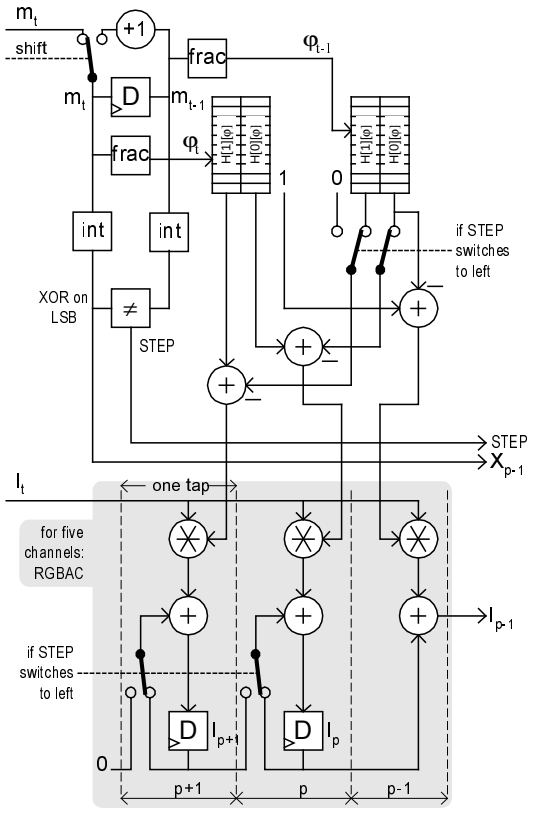

Figure 11: *hardware resample structure implementing the algorithm of Figure 10 for a prefilter width of 2.*

We have designed a hardware resample structure that implements this pseudo code which is shown in Figure 11 for a prefilter width of 2. In dedicated hardware, lines 8 to 12 and lines 15 to 18 from Figure 10, can be implemented in parallel. H_0 to H_3 can be quantized and stored in 4 tables. We combine these in a single table that can output $H_0(\varphi_t)$ to $H_3(\varphi_t)$ in one clock cycle. Preliminary experiments show that a table size of 32 entries and 8 bit values does not introduce visible aliasing.

Both the pixel and texel I variables represent RGBA values extended with the *contribution factor* C introduced in Section 3.3. We can easily generate the contribution factor as an additional color component C. For our system the gray part of Figure 11 is duplicated five times: once for each RGBAC color channel.

The *shift* input is controlled by the rasterizer and is needed in order to generate *partial* pixel values outside the polygon boundary, that are still in the registers after the final texel color has been shifted in. The resample structure as shown here is for prefilters with width equal to an even number of pixels. Slight modification is necessary to accommodate odd filter widths.

5 2D Resample Unit

In this section we describe how we combined two 1D resample structures, described in Section 4, to obtain a 2D resample unit. The main disadvantage of one-pass input driven resampling using a 2D filter is the high random read-write data traffic to a screen space accumulation buffer [12]. Without resorting to tile based rendering, the high data traffic will be to a off-chip accumulation buffer. To avoid the off-chip data traffic we choose an alternative solution using two-pass forward mapping, introduced by Catmull and Smith [5]. The two-pass mapping also allows us to efficiently implement resample discretization (4). Another advantage is that the two-pass resample process requires fewer operations per pixel compared to the one-pass using the same filter footprint.

The following diagram illustrates our implementation.

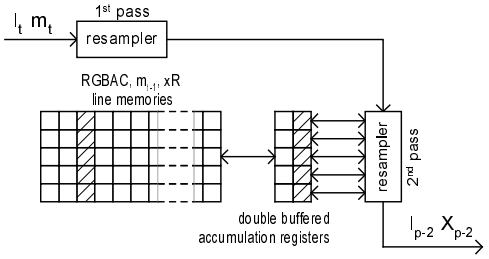

Figure 12: *two pass resampling. The 2nd pass requires some line memories for per pixel color accumulation.*

To avoid the need for memory for the intermediate image between the horizontal and vertical pass, we have *interleaved* both passes: e.g.: for every row produced by the horizontal pass the vertical pass cycles over the columns and consumes one input sample for every column and accumulates the splatted values in the (double buffered) accumulation registers of the structure. These registers should contain the values associated width the filter state of the

current column: For each column the registers values are stored, to be used for the next line, in *on-chip* line memories that can be accessed in a fifo manner. The 1D resample structure of Figure 11 has been designed to minimize the number of registers, and thus to minimize the number of required line memories for the 2D resample unit. The pixels are outputted in an irregular way, to the next stage of the pipeline. But a pixel position is only once outputted per polygon.

Although we have implemented floating-point arithmetic and not fixed point arithmetic in our algorithms yet, we can give a fair estimate of the amount of line memories needed. The values we need to store are m_{t-1}, xR. The xL value is not stored in the line memories: we initialize it in m_{t-1} for each line. Both m_{t-1} and xR can be stored in 15 bits (fixed point 10.5). For a output color I_t with 8 bits per color component about 10 bits for the accumulation registers are sufficient. The extra precision is required due to the repeated accumulation and possible negative parts in the prefilter function. The amount of storage we need for five channels (RGBAC) and a prefilter width of two is:

2 x 15 + 5 x (2 x 10) = 130 bits.

For a scan line length of 1024 pixels this results in a required on-chip memory of 16 Kbyte (29 Kbyte for a four wide prefilter). Note that the line memories only have to be as wide as a scan line for large polygons that indeed fill the whole scan line length. To divide the line memory requirement by, for example, a factor of 2 or 4 we could vertically split the geometry of polygons larger than 512 or 256 pixel spacings.

The bandwidth required to the line memories depends on the average minification factor and the prefilter width. The average minification factor, using MIP-map textures, is roughly 1.5. With N being the prefilter width, we require 1.5 x N read-write accesses per pixel. The bandwidth to the accumulation using a 2D filter is roughly 1.5^2 x N^2. So the required bandwidth to the accumulation buffer of our two-pass algorithm compared to the one-pass is 3 times less for N=2 and 6 times less for N=4. But more important, the bandwidth to the line memories of our two-pass implementation can be kept *on-chip*, whereas in the case of one-pass forward texture mapping the required bandwidth is to a *off-chip* accumulation buffer, unless the accumulation buffer fits on chip or tile based rendering is used.

6 Rasterization Unit

This section discusses the main changes to a traditional inverse texture mapping rasterizer, that are required to implement an efficient two-pass forward texture mapping rasterizer. We restrict ourselves to planar maps.

The vertex coordinates of the polygon to rasterize in texture space are the texture coordinates, u and v, that are given at each of the polygon vertices. Due to our box reconstruction filter, we require texels that lay outside the polygon within a border $\frac{1}{2}f_t$ distance from the polygon boundary. This complicates the design of the rasterizer, especially because the border for the 2nd pass has to be generated by the 1st pass. However, in spite of the complications, the computational costs for the rasterizer setup and the "polygon edge walk", are in the same order of an inverse texture mapping rasterizer.

MIP-maps

Both with inverse texture mapping and forward texture mapping, there is no upper bound to the number of texels that can fall within the (mapped) prefilter footprint of a pixel, and thus also no upper bound to required processing and texture memory bandwidth. For inverse texture mapping, this has been commonly solved by the MIP-map [22] preprocessing technique. In our forward texture mapping pipeline we have also implemented the MIP-map technique. When using MIP-map textures, scale factors close to one are common. So, the common resample discretization (1) is not suited. Our resample structure based on resample discretization (4) is suited.

Because our forward texture mapping is performed in two passes, we do not directly use the standard 3D MIP-maps that are given by OpenGL applications. Our rasterizer traverses the texels of a polygon on a texture grid that can change resolution in a power of two, according to 4D MIP-map [12] level switches. The required 4D MIP-map texels are generated "on-the-fly" from the given 3D MIP-maps using bilinear interpolation or unweighted pixel averaging. The horizontal MIP-map level may change per texel. In our current implementation we keep the vertical MIP-map level constant for the whole polygon to simplify the implementation of the vertical pass. The vertical MIP-map level is based on the highest required vertical resolution within the polygon. This causes unnecessary texels to be fetched for locations within the polygon where a lower resolution would be sufficient. In future work we would like to support vertical MIP-map level switches on a per texel basis.

Note that our forward texture mapping algorithm does not have to average between adjacent MIP-map levels as is the case with commonly used trilinear and anisotropic inverse texture mapping filters. Moreover our resample algorithm does not *require* MIP-map textures for proper filtering. We can use video streams as texture without the need for real-time generation of MIP-map textures at the same speed of real-time MIP-map generation. However, for strong minification we require a higher precision of the accumulation registers in the resample structure of Figure 11. For real-time generation of MIP-maps dedicated hardware circuitry, described in Schilling at al. [19], may be required.

Magnification

In the case of texture magnification our rasterizer inserts "artificial texels", using the bilinear interpolation reconstruction filter that feeds the resample unit, which is also use to generated the 4D MIP-maps. The resample unit still applies the prefilter for magnification. This is required to generate proper "partial pixel values" for the polygon edge anti-aliasing method as described in Section 7.

Other interpolants

Besides texture colors, our texture space rasterizer must also generate other pixel properties. We also generate depth (z) values and diffuse interpolated colors. Specular colors may also be generated, however this is not yet implemented. The color shading can be interpolated in texture

space and blended together with the associated texel colors before the texel is sent to the resample unit.

7 Edge Anti-Aliasing

In this section we describe how we perform edge anti-aliasing, using the pixel fragments delivered by the 2D resample unit (Section 5) for each polygon. The general idea is not to limit the texture prefiltering mechanism to the polygon boundary: the texture space rasterization and the splatting 2D resampling process results in "partial prefiltered" pixels within a border of half a prefilter width outside and inside the polygon boundary. For much of the polygon edge situations, we can use the generated pixel partial colors and contribution factor to correctly obtain a prefiltered pixel, as if the prefilter would be applied on texels from the interior of the same polygon.

7.1 Edge situations

For edges of adjacent polygons, correct prefiltering is obtained by adding partial pixel colors (Section 3.3), delivered by the 2D resample unit. For multiple adjacent polygons that fall within a pixel's prefilter footprint, the pixel color contributions from each polygon are accumulated to reach full contribution.

For silhouette edges, it is more complicated. Let us consider polygon A overlapping polygon B. In the case that the silhouette edge overlaps the interior of B (where the pixels of B have contribution factor $C^B = 1$), we also obtain a correct prefiltered edge. In this case, we use C^A to adjust the pixel color of B before adding:

$$I = I^A + (1 - C^A) \cdot I^B \qquad (5)$$

We can also have a *complex silhouette edge*, where a silhouette edge of A overlaps an edge pixel area of B: the B pixels have partial colors $C^B < 1$. In this case we do not obtain a correct prefiltered edge. We add the partial pixel color of B to A. This might result in no contribution of a third polygon that is adjacent to B or partial overlapped by B. Note that complex silhouette edges are rare.

We obtain correct prefiltering on so-called *arbitrary shaped edges* that are defined by the alpha channel of texture maps. For example they are used to model branches with leaves where the space between the leaves is transparent. A texel color should only contribute to a pixel if its alpha value passes the so-called "alpha-test". We moved the alpha-test operation before the 2D resample unit where it is performed in texture space: this way prefiltering is only applied on texels that pass the alpha-test.

7.2 Fragment buffer

To be able to implement the above process, we require pixel fragments in depth sorted order. Because polygons can be delivered in random depth order, we choose to store pixel fragments, per pixel location in depth sorted order in a pixel fragment buffer, similar to the A-buffer [4]. However, we do not store a measure of geometric coverage per pixel fragment (such as a coverage mask), as is often used [4, 15, 24]. We store the contribution factor C per pixel fragment that represents the partial weight according to the polygon overlap with the 2D prefilter overlap. Whereas the geometric coverage is often obtained using a sub-pixel masks corresponding to a box prefilter, with 1x1 footprint size, our contribution factor is based on a higher order prefilter that can have a larger footprint size, such as 2x2 or 4x4.

The fragment buffer algorithm consists of 2 stages: *insertion* of pixel fragments and *composition* of pixel fragments. We use a fragment buffer with a fixed number of fragments, such as in [15], that can be stored at each pixel location. To prevent overflow during the insertion stage we merge fragments that are closest in their Z values. After all polygons of the scene are rendered the composition stage composes fragments in front-to-back order, according to Section 7.1. The final pixel color is obtained when the sum of the contribution factors of all added fragments is one or more.

Contrary to inverse texture mapping systems, we deal properly with very small triangles or slivers. High quality anti-aliasing prefiltering can be obtained on polygon edges as well as within polygon interiors, except for intersecting and complex silhouette edges. We did not implement support for multipass texturing or multi-texturing blending in the fragment buffer yet.

8 Results

We incorporated our texture mapping and resample algorithms, as described in the previous sections, using a tent prefilter with a footprint size of 2 in a public domain software implementation of OpenGL: Mesa [18]. We used animated image sequences to compare texture anti-aliasing and edge anti-aliasing quality with images generated by a commercially available advanced graphics card. Artifacts are much more noticeable with animated images than with still images.

We compared our generated image sequence with an image sequence generated with anisotropic level 8 filtered images without super sampling and with 4x4 super sampling. We used the popular game Quake III with separately loaded scenery [21], that includes high contrast and high frequency texture maps, to generate the 3D scenery. Lightmaps are switched off as they are not fully supported in the pixel fragment buffer algorithm yet. Using the so-called "timedemo" option of Quake III we used *exactly the same scenery* to generate the image sequence with both the anisotropic texture filtering capable graphics card and our modified OpenGL implementation. We compared the generated image sequences side by side, frame synchronous, on a high quality CRT monitor on a resolution of 640 x 480 pixels. A part of a bitmap (approx. 100x200) of each sequence is shown in Figure 13. The bitmaps are also shown in the color section where a non-super sampled bitmap is included for reference.

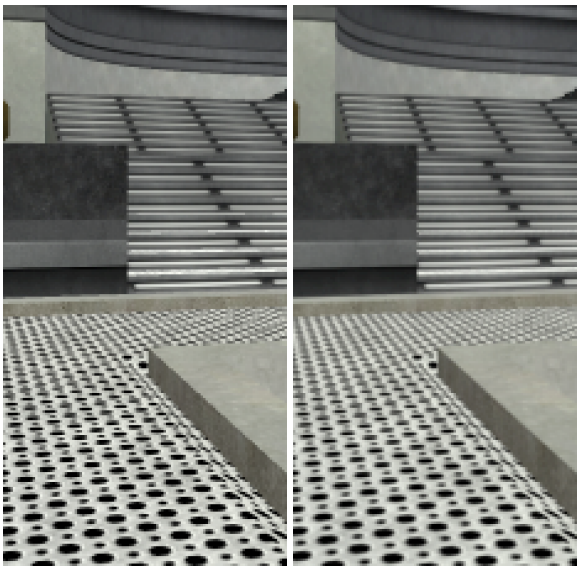

Figure 13: *bitmap left: 4x4 super sampling using anisotropic level 8 texture filtering, right: our implementation using a tent prefilter and a maximum of 4 pixel fragments per pixel.*

Comparing texture filtering quality with the animated sequences shows that our method is better as the anisotropic filtered sequence for both non super sampled and 4x4 super sampled images.

Comparing edge anti-aliasing quality with the animated sequences shows that 4x4 super sampling has some visible aliasing artifacts on all edges. We have less aliasing artifacts on the edges, except for edges due to polygon intersection and so-called complex silhouette edges which are very rare. Moreover, our approach correctly anti-aliases arbitrary shaped edges modeled with the alpha-test, often used to model partial transparent trees.

9 Costs

Off-chip bandwidth

We compare our bandwidth costs to inverse texture mapping. Because inverse texture mapping reads texels in an non regular way a texture cache is required. But this will not always prevent texels being fetched several times. Our method fetches texels only once (except for "repeated textures") and in regular order, decreasing required texture bandwidth, but our method increases required texture bandwidth due to our current implementation that allows no vertical MIP-map level changes.

Our bandwidth requirements to the fragment buffer are about the same as the frame buffer bandwidth requirements of a 2x2 super sampling system. This can be seen using the following reasoning: 2x2 super sampling uses 4 sub-pixels per pixel, that are *at least* read and written once. Our required bandwidth to the fragment buffer depends on the overdraw factor which depends on the scene and slightly on the prefilter footprint size. In our test case scene [21] we measured a average overdraw factor of 3.4 over 400 frames rendered at a screen size of 640x480 and a tent prefilter having a width of 2. The average overdraw factor per frame varies between 2.3 and 4.4. Normally, the average overdraw factor is less than 4, but in the fragment buffer we must insert in sorted order. For small number of fragments per pixel this is not that costly. We have implemented a fragment buffer which uses at most 4 fragments per pixel. So, we require somewhat less off-chip memory bandwidth as 2x2 super sampling, but our image quality is better than 4x4 super sampling.

The measured average overdraw factor, using a prefilter with width of 4, is 4.4 and it varies per frame between 2.9 and 5.8. So, in our test case, we can obtain better filtering using the larger prefilter footprint [23] at only moderate increases in required off-chip memory bandwidth.

Computational costs

Our rasterizer traverses texels instead of pixels. We assume an average horizontal minification factor of 1.5 and a somewhat higher average minification factor of 2.5 for the vertical direction (or vice versa) due to the perspective transformation. So per screen pixel, our rasterizer has to traverse 1.5*2.5 = ~4 texels, comparable to the rasterization (pixel traversal) costs of 2x2 super sampling.

For filtering, the average number of multiplications per screen pixel in our two-pass algorithm, using the tent prefilter which corresponds to bilinear filtering, is roughly the same as for anisotropic texture filtering with at most 8 trilinear probes.

10 Conclusions and Future Work

We have presented a method that implements the texture mapping function within a 3D graphics pipeline, using two-pass forward texture mapping that circumvents the pitfalls of previous work on forward texture mapping. We have implemented our algorithms within a software implementation of OpenGL. We tested our algorithms on functional correctness and image quality using Quake III with 3D scenery from a third-party.

- We have developed a new input driven 1D resample structure that *efficiently* integrates box reconstruction filtering and high order prefiltering. It is suited for texture mapping with a continuously spatial varying scale factor but it does not suffer from DC-ripple. Therefore, no intensity normalization post-processing stage is needed.
- We have shown how two of such 1D resample structures can be combined to obtain a 2D resample unit that does not require off-chip memory bandwidth, as is the case for a *one-pass* input driven 2D resample unit.
- We integrated the 2D resample unit in a 3D pipeline with a texture space rasterizer that uses MIP-map textures to minimize texture memory data traffic.
- We have added a pixel fragment buffer to support high quality edge anti-aliasing.

We do not claim that forward texture mapping can easily support all the features of the most recent 3D graphics API's. Features that have been developed on the inverse texture mapping architecture, such as dependent multi-

texturing may not have an easy match on a forward texture mapping pipeline. A 3D pipeline implementation without these features can still be very interesting to various application domains.

Image quality

Our texture anti-aliasing quality is better than anisotropic texture filtering with 8 trilinear probes, but filtering costs are about the same. The required memory bandwidth costs for edge anti-aliasing, using a prefilter footprint size of 2x2, are about the same as 2x2 super sampling, however, our edge anti-aliasing quality is better than 4x4 super sampling, except on so-called called complex silhouette edges and edges due to polygon intersection. Very small polygons or slivers do not show popup artifacts as is common with inverse texture mapping. If a larger prefilter footprint of 4x4 is used costs are only moderate increased for our test case, but higher quality can be obtained (for example using artificial sharpening filters functions).

Future work

We now use a single vertical MIP-map level for the whole polygon. We aim to change the vertical MIP-map level selection on a per texel basis to avoid unnecessary texel fetches, thereby decreasing required texture bandwidth.

We aim to enhance the fragment buffer insertion and composition algorithm, such to better deal with complex silhouette edges and edges due to intersection. We also aim to add support for multiple texturing and multipass texturing.

Acknowledgements

The authors would like to thank the other project members Marcel Tomáš and Patric Theune. We would also like to thank Frans Peters for his helpful comments and suggestions.

References

1 Bennett, P.P. and Gabriel, S.A. System for spatially transforming images, , *United States patent nr 4472732*, Sep. 18, 1984

2 Blinn J.F., Return of the jaggy, *IEEE Computer Graphics & Applications*, March 1989

3 Cant, R.J. and Shrubsole, P. A. Texture Potential MIP Mapping, A New High-Quality Texture Antialiasing Algorithm, *ACM Transactions on Graphics*, Vol. 19, No. 3, July 2000

4 Carpenter, L. The A-buffer, an Antialiased Hidden Surface Method. *Computer Graphics (SIGGRAPH '84 Proceedings),* vol. 18, pp. 103 - 108, July 1984

5 Catmull, E. and Smith A.R. 3-D Transformations of Images in Scanline Order. *Computer Graphics (SIGGRAPH '80 Proceedings),* vol. 14, no.3, pp. 279 - 285, July 1980

6 Chen, B., Dachille, D. and Kaufman, A. Forward Image Mapping, *IEEE Visualization '99*

7 Crochiere R.E. and Rabiner L.R. *Multirate Digital Signal Processing.* Prentice Hall, 1983

8 Dalfsen A.J., Stessen J.H.J.C. and Janssen J.G.W.M. Sample rate conversion, *United States patent nr 5892695*, Oct. 28, 1997

9 Fant, K.M. A nonaliasing, real-time spatial transform technique, *IEEE Computer Graphics and Applications*, January 1986

10 Ghazanfarpour D. and Peroche B., A high-quality filtering using forward texture mapping, *Comput. & Graphics* vol. 15, no.4, pp. 569 - 577, 1991

11 Hakura, Z. and Gupta. A. The Design and Analysis of a Cache Architecture for Texture Mapping. *Proceedings of 24th International Symposium on Computer Architecture*, 1997

12 Heckbert, P.S. Fundamentals of Texture Mapping and Image Warping. *Masters Thesis*, Dept. of EECS, University of California at Berkeley, 1989.

13 HRAA: High-resolution Antialiasing through Multisampling, *NVidia Technical Brief*, www.nvidia.com/docs/IO/83/ATT/HRAA.pdf

14 Janssen, J.G.W.M.; Stessen, J.H.; de With, P.H.N. *An advanced sampling rate conversion technique for video and graphics signals*, Sixth International Conference on Image Processing and Its Applications, Volume 2, p 771 -775, 15-17 July 1997

15 Jouppi, N.P. and Chang, Chun-Fa Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. *Proceedings 1999 SIGGRAPH/Eurographics Hardware Workshop*

16 Kirk, D.B. Unsolved Problems and Opportunities for High-quality, High-performance 3D Graphics on a PC Platform. Invited paper, *Proceedings 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*. The presentation slides discuss different anisotropic filtering in more detail: www.merl.com/hwws98/presentations/kirk/ ACM 1998.

17 McCormack J., Perry R., Farkas K.I. and Jouppi N.P. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping, *Computer Graphics (SIGGRAPH '99 Proceedings)*, pp. 243-250, August 1999

18 Mesa 3.2, 3D Graphics Library by Brian Paul at http://www.mesa3d.org/

19 Schilling A.G., Knittel G. and Straßer W., Texram: A Smart Memory for Texturing, *IEEE Computer Graphics & Applications*, 1996

20 Smith A. R., Digital Filtering Tutorial for Computer Graphics, part 1, *Lucasfilm technical memo 27*, *www.alvyray.com*, Nov. 20, 1981 also presented as turorial notes at SIGGRAPH '83 and '84

21 Tequila's "Subversive Tendencies" Quake III scene, available at http://bettenberg.home.mindspring.com/teqtrny3.html which uses textures maps from http://www.planetquake.com/hfx/ for some of which we increased the contrast to make the scene more bright since we have not implemented support for lightmaps yet.

22 Williams, L. Pyramidal Parametrics, , *Computer Graphics (SIGGRAPH '83 Proceedings)*, july 1983

23 Wolberg, G. editor. *Digital Image Warping*, IEEE Computer Society Press, Los Alamitos, CA 1990

24 Zwicker, M., Pfister, H., Baar, J. van and Gross, M. Surface Splatting, *Computer Graphics (SIGGRAPH '01 Proceedings)*, pp. 371-378, August 2001

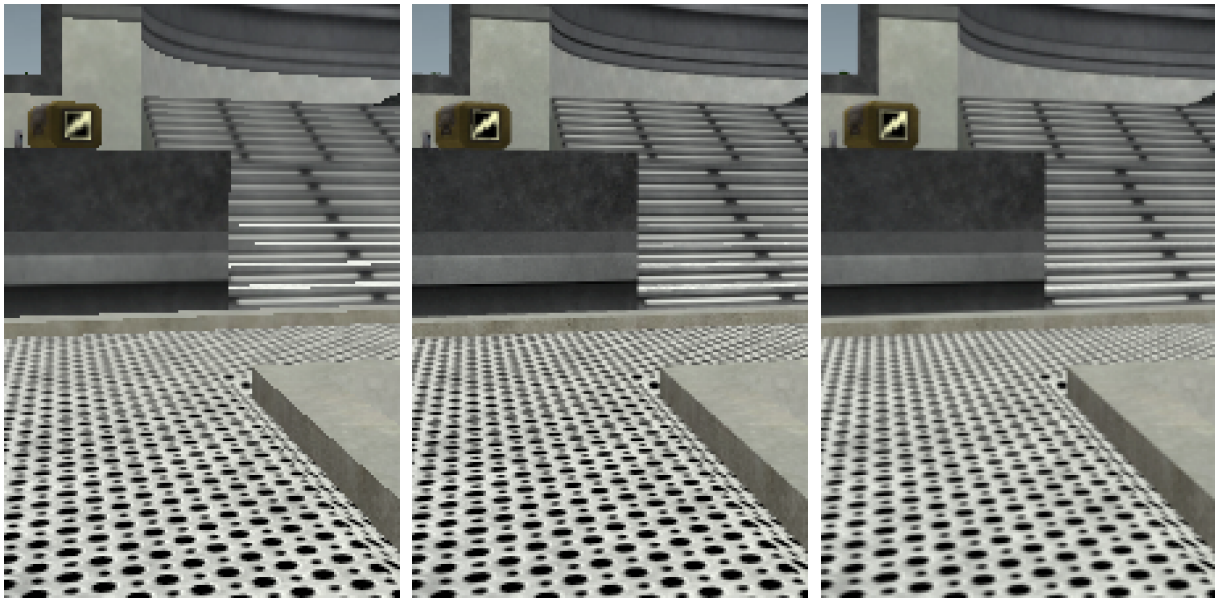

Figure 14: *bitmap left: no super sampling using anisotropic texture filtering with at most 8 trilinear probes, center: 4x4 super sampling using anisotropic texture filtering with at most 8 trilinear probes, right: our implementation using a tent prefilter and a maximum of 4 pixel fragments per pixel*