Analytical Texture Mapping

Koen Meinds and Elmar Eisemann

***Abstract*—Resampling of warped images has been a topic of research for a long time but only seldomly has focused on theoretically exact resampling. We present a resampling method for minification, applied on the texture mapping function of a 3D graphics pipeline, that is derived from sampling theory without making any approximations. Our method supports freely selectable 2D integratable prefilter (anti-aliasing) functions and uses a 2D box reconstruction filter. We have implemented our method both for CPU and GPU (OpenGL) using multiple prefilter functions defined by piece-wise polynomials. The correctness of our exact resampling method has been made plausible by comparing texture mapping results of our method with those of extreme supersampling. We additionally show how the prefilter of our method can also be applied for high quality polygon edge anti-aliasing. Since our proposed method does not use any approximations, up to numerical precision, it can be used as a reference for approximate texture mapping methods.**

***Index Terms*—Texture mapping, space-variant resampling, sampling theory, anti-aliasing.**

I. Introduction

TEXTURE-MAPPING research is a classic topic in computer graphics [1] and built upon work of the mid-1960 s on geometric transformations of digital images in the remote sensing community (e.g., areal photography). It directly relates to sampling theory [2], [3] dating back to 1928; a texture is a sampled representation of an underlying function, reconstructed by a reconstruction filter. When mapped to a surface, it will be sampled again during image generation. Yet, no texture-mapping algorithm currently implements the theoretical resample steps that follow from the theory without introducing approximations. We present an exact analytical solution (up to numerical precision).

We focus on high-quality minification, as magnification can be handled well with standard interpolation schemes, and textures often exhibit higher resolution than how they appear on screen. In this case, high-frequency patterns not representable by the screen sampling grid must be removed to avoid aliasing (such as Moiré and staircase artifacts). An anti-aliasing prefilter removes these before sampling but should avoid attenuating representable frequencies, which would cause blur. While a sinc prefilter is optimal, it is not computationally feasible due to an infinite support. A careful prefilter design balances aliasing, blur and other artifacts.

Received 12 March 2025; revised 7 September 2025; accepted 10 September 2025. Date of publication 17 September 2025; date of current version 6 February 2026. Recommended for acceptance by Y. Dong. *(Corresponding author: Koen Meinds.)*

The authors are with the Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: koen.meinds@proton.me).

This article has supplementary downloadable material available at https://doi.org/10.1109/TVCG.2025.3611315, provided by the authors.

Digital Object Identifier 10.1109/TVCG.2025.3611315

We opt for generality and support arbitrary higher-order piecewise polynomial prefilter functions on rectangular patches, which can be used to approach a windowed-version of the sinc function, e.g., Lanczos3. Hereby, any filter function can be approximated to an arbitrarily high accuracy (e.g., via a Taylor power series polynomial per patch). We use antiderivatives of these prefilter functions to compute the texel contributions to each pixel and guarantee an analytically correct integration, including perspective transformation. While our main target is not performance, we enable a trade-off between computational cost and resulting quality.

Based on a box-filter reconstruction of the texture signal, which we will motivate, our method delivers an exact analytic resampling process. Due to its accuracy, it can serve as an image-quality reference for minification algorithms.

In short, our contributions are:

- An analytical texture-mapping resampling method;
- Integration of a high-degree polynomial prefilter;
- Edge anti-aliasing with prefilters;
- A CPU and GPU implementation.

II. Related Work

Catmull's dissertation [1] could be seen as a starting point for texture mapping. It builds upon earlier works of digital image warping and inspired others; cf. the overview Wolberg [4]. The theoretical texture mapping resampling process is described by Smith [5] and Heckbert [6]. It is derived from the sampling theorem for 1D signals, often attributed to Shannon 1949 [3] who drew on the work of Nyquist [2] and others, first presented by Kotelnikov 1933 [7] as pointed out by Smith [8].

Aliasing is already mentioned in Catmull's dissertation. Proper filtering for texture mapping (Section III-B), needs to address two main problems: 1) the selection of texels contributing to a pixel according to a transformed prefilter footprint in texture space, i.e., anisotropic filtering, and 2) the weighting of these texels according to a reconstruction filter and an anti-aliasing prefilter, the latter an approximate sinc function –for minification– with square support as derived from the sampling theory [9, Sec. 8.4]. Most texture-mapping research has focused on the anisotropic problem; i.e., to efficiently approximate which texels contribute according to the prefilter support mapped in texture space [10], [11], [12], [13]. Less research addresses the second problem. Circular support filters, mostly employ an approximate Gaussian [14], [15], [16], [17], which is not a good approximation of a sinc function with square support. Using a rectangular support, Blinn and Newell [18] used a 2D-tent prefilter, Tumblin and Guenter [19] used a tent and cubic spline prefilter, Khoshelham and Azizi [20] used a Kaiser windowed sinc, and later Manson and Schaefer [21] used an approximate Lanczos2 filter. Another approach are square filters based on

subdivision into two 1D-resampling passes, as introduced by Catmull and Smith [22] and later extended in [23], [24]. The resulting high-quality approximate sinc functions require less computational costs, but in presence of rotations, strongest at a 45°, it is a bad approximation of square-support filters. Our approach handles selecting and weighting the texels analytically exact using a square-support polynomial sinc approximation.

Very approximate but fast prefiltering include the well-known MIP maps, a hierarchy of pre-computed down-scaled texture maps [25]. A more advanced alternative supporting more general prefilter functions are NIL maps [26]. Another approach is "summed-area tables" (SAT) introduced by Crow [27], which can determine the exact integration in an axis-aligned rectangle of a texture, and is later extended to closely approximate quadrilateral regions [28].

Analytical rasterization can be seen as a related problem. Manson and Schaefer [29] focus on curved closed shapes using a radial prefilter that is solved with a horizontal 1D boundary pass. Alternatively, the prefilter-polygon convolution can be computed by subdividing overlap regions into geometrical shapes whose integrals are analytically computable [30]. Both works refer to Duff [31], which describes analytical rasterization by computing the convolution of a polygon with an analytically integrable prefilter. They subdivide the prefilter into "pixel-sized" squares and evaluate the convolution integral of a polygon by clipping it into a patch per square and evaluating each patch using a trapezoid-based method. The recent work of [32] uses the same clipped-polygon patches but evaluates each patch using a Green's-theorem method. Our work is related but focuses on resampling of perspectively transformed discrete images using the sampling theorem. Our convolution integral evaluation can be seen as a SAT generalization for quadrilaterals but applied to a 2D analytical prefilter instead of texture data. We do not require subdivision and clipping against "pixel-sized" squares. Analytical precision, numerical stability, and generalization of the prefilter are also key in our method.

III. Review of Resampling

A. Classical Sampling Theorem

Crochiere and Rabiner [33] Chapter 2.1 (1D), Wolberg [4] Chapter 4 (1D), and Glassner [9] Chapter 8 (1D and 2D) provide a thorough introduction to the sampling theorem. Here, we briefly revisit the topic. The sampling theorem's *first part* states that a –theoretically– exact reconstruction from a sampled signal of an original continuous-space signal requires the sampling frequency f_s to be greater than twice the maximum sinusoidal frequency component $f_{\max}$ present in the original signal. Or alternatively; for a *given sample rate* f_s an exact reconstruction is theoretically possible for a signal whose frequency components are limited to $f_{\max} < \frac{1}{2} f_s$. If this condition is met, the sampling theorem's *second part* states that the exact original values can be reconstructed by convolution with a sinc *reconstruction filter*. The term "$\frac{1}{2} f_s$" is often referred to as the Nyquist frequency.

For $f_{\max} > \frac{1}{2} f_s$, aliasing occurs (Fig. 1), due to undersampling or the continuous-space signal not being bandlimited (enough). A low-pass filter can block frequencies above $\frac{1}{2} f_s$ (the *stopband*) with the sinc filter being the "ideal" low-pass filter [4]. It is a box window in the frequency domain, where the frequencies below $\frac{1}{2} f_s$ (the *pass band*) are fully transmitted and above it (the *stop band*) fully blocked. Having infinite support (footprint), computable approximations are needed, resulting in a non-ideal transition band between the stop and the pass band (Fig. 2), leading to two possible artifacts:

1) Signals with frequencies leaking into the stop band lead –after sampling– to *aliasing*.
2) Signals with frequencies within the pass band, but close to the cut-off $\frac{1}{2} f_s$, are attenuated above/below and lead to sharpening/blur.

Some researchers consider both cases *aliasing* (e.g., [34]), while others restrict the term to undersampling artifacts (e.g., Glassner [9] Section 8.2). For us, *aliasing* are low-frequency aliases of high frequencies passed into the stopband, irrespective whether the cause is undersampling, poor reconstruction, or poor prefiltering. Aliasing is considered more detrimental to perceived image quality than blur/sharpening artifacts (following Wolberg [4] Section 4.5). Hence, practical filters combating aliasing need to limit $f_{\max}$ to $\frac{1}{2} f_s$.

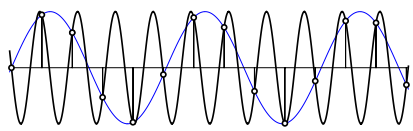


Fig. 1. High frequencies will fold into low frequency aliases (blue) when too high frequensies are present ($f_{\max} \approx 0.8 f_s$).

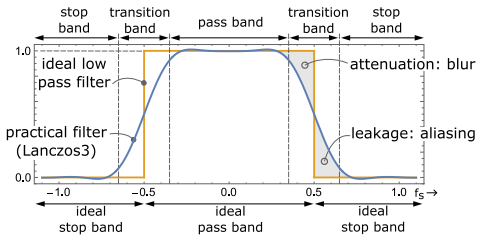


Fig. 2. Ideal and practical low-pass filter frequency response.

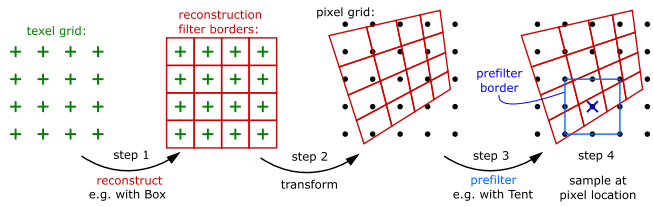


Fig. 3. Resampling steps: Reconstruct continuous-space image from the green texels; using a box reconstruction obtains the red bordered squares of constant value. Transform this continuous-space image from texture space to screen space obtaining quadrilateral areas of constant value. Prefilter the transformed image for each pixel position; the blue pixel is contributed by texels whose quad overlap it's square prefilter footprint (blue) and weighted according to the prefilter function.

B. Resampling Formula

The resampling process, for triangle-based texture mapping is presented in detail by Smith [5] and further developed by others [4], [6], [19], [20], [26]. The sampling theorem's first and second part (Section III-A) result in Step 3 and 1 respectively in the process below. The resampling process steps (Fig. 3) are:

1) Reconstruct a continuous-space image from the integer-grid of input samples.

2) Transform the image to output space.
3) Prefilter (bandlimit) the transformed image.
4) Sample it at integer-grid output positions.

The transformation (Step 2) of the continuous-space image may introduce high frequencies not representable on the output grid. Hence, the need for an anti-alias prefilter (Step 3) before sampling on the output grid (Step 4). These four steps are the starting point of our algorithm. They connect to a mathematical expression, the *resampling formula*, which we derive using our own notation, deviating slightly from the terminology in [6].

A sampled input signal $i_s : \mathbb{R}^2 \to \mathbb{R}$ is represented by the multiplication of the sampling function $s : \mathbb{R}^2 \to \mathbb{R}$ $s(\mathbf{u}) := \sum_{\mathbf{t}\in\mathbb{Z}^2} \delta(\mathbf{u}-\mathbf{t})$ (equally-spaced Dirac pulses) with an original input signal $i_o : \mathbb{R}^2 \to \mathbb{R} : i_s(\mathbf{u}) := i_o(\mathbf{u})s(\mathbf{u}) = \sum_{\mathbf{t}\in\mathbb{Z}^2} i_o(\mathbf{t})\delta(\mathbf{u}-\mathbf{t})$, where values $i_o(\mathbf{t})$ are called *texel* values and typically given via a discrete texture image. Let $m : \mathbb{R}^2 \to \mathbb{R}^2$ denote the mapping from a position $\mathbf{u}$ in *input/texture space* to a position $\mathbf{x}$ in *output/screen space*. We denote $r : \mathbb{R}^2 \to \mathbb{R}$ the reconstruction filter applied in texture space, and $h : \mathbb{R}^2 \to \mathbb{R}$ the prefilter applied in screen space. With $*$ denoting 2D convolution, the resampling steps lead to:

1) $i(\mathbf{u}) := (i_s * r)(\mathbf{u}) = \sum_{\mathbf{t}\in\mathbb{Z}^2} i_s(\mathbf{t}) \cdot r(\mathbf{u}-\mathbf{t})$
2) $j_m(\mathbf{x}) := i(m^{-1}(\mathbf{x}))$
3) $j(\mathbf{x}) := (j_m * h)(\mathbf{x}) = \int_{-\infty}^{\infty}\int j_m(\mathbf{k}) \cdot h(\mathbf{x}-\mathbf{k})\, \mathrm{d}\mathbf{k}$

Fusing the expressions 1–3 leads to the resampling formula, which is evaluated at integer locations (Step 4) to obtain screen *pixel* values:

$$j(\mathbf{x}) = \sum_{\mathbf{t}\in Z^2} i_s(\mathbf{t}) \cdot w_{\mathbf{t}}(\mathbf{x}) \qquad \text{where} \tag{1}$$

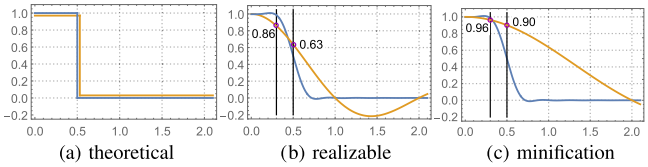
$$w_{\mathbf{t}}(\mathbf{x}) = \int_{-\infty}^{\infty}\int r\big(m^{-1}(\mathbf{k}) - \mathbf{t}\big) \cdot h(\mathbf{x}-\mathbf{k})\, \mathrm{d}\mathbf{k} \tag{2}$$

$w_{\mathbf{t}}(\mathbf{x})$ represents the *weight* of a texel at $\mathbf{t}$ for a pixel at $\mathbf{x}$. In (2) the prefilter center is shifted onto the pixel location, which is the conceptually common view. Alternatively, due to commutativity of convolution (in Step 3) we can write:

$$w_{\mathbf{t}}(\mathbf{x}) = \int_{-\infty}^{\infty}\int r\big(m^{-1}(\mathbf{x}-\mathbf{k}) - \mathbf{t}\big) \cdot h(\mathbf{k})\, \mathrm{d}\mathbf{k} \tag{3}$$

This formulation considers the prefilter at the origin of the screen space; conceptually it takes the mapped reconstructed signal and shifts it according to the pixel position $\mathbf{x}$ on the prefilter at the origin such that the integrator $\mathbf{k}$ is directly used as a parameter for the prefilter function. This interpretation will simplify our derivation.

Please note that the integrals in (2) and (3) are defined in screen space, where a reconstructed texel footprint undergoes a perspective warping (m) into the screen space, whereas the prefilter footprint is not warped. Note, however, that most texture-mapping literature (e.g., [6], [10], [11], [13], [19], [35]) focuses on, using a texture-space integral where a prefilter-pixel footprint is warped (m^{-1}) into texture space. The screen-space integral formulation simplifies exact computations of the *unwarped* prefilter and motivated the direction for our algorithm.



(a) theoretical (b) realizable (c) minification

Fig. 4. Spectra of prefilter (blue) and reconstruction filter (orange) with horizontally the f_s-factor (only positive shown) and vertically the amplification. (a) Ideally both are Box functions. They are identical at scale factor 1 (but orange slightly shifted for illustration purposes). Multiplication results in a spectrum identical to the prefilter spectrum even for minification (scale $<$ 1.0) that widens the reconstruction box spectra. (b) Realizable good-quality (Lanczos3) prefilter spectrum and sinc spectrum of Box reconstruction filter at scale factor 1. Multiplication with the sinc attenuates frequencies in the prefilter's passband (e.g., at $0.3f_s$ with 0.86x) reducing sharpness a bit. *c)* Same as (b) but at scale factor 0.5, stretching the reconstruction spectrum by 2x, obtaining less attenuation in the prefilter's passband and hence increases sharpness.

C. Reconstruction Filter

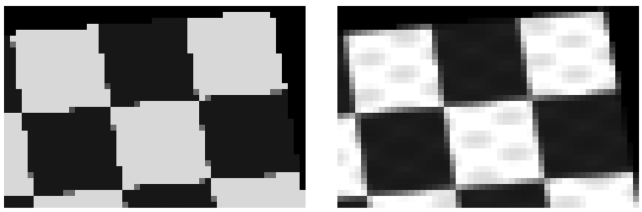
Our work relies on a box-function with square support as reconstruction filter:

$$\mathbf{box}(x,y) = \begin{cases} 1 & \text{if } |x| < 0.5 \text{ and } |y| < 0.5; \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

While it matches the common –wrong– intuition that pixels are little squares [36], it is actually a suitable choice, as it has the *smallest support* that adheres to the *unity of composition* property, does not suffer from *DC-ripple*, and requires *no per-pixel normalization*. It also follows naturally from applying supersampling to the limit using Dirac samples. Our motivation is further elaborated in the following subsections, which also shows why, for high-quality minification, the sinc-approximating quality of the prefilter is more important than that of the reconstruction filter.

1) Reconstruction Filter Influence on Frequency Response: In theory (Section III-A), we could best apply a sinc function for both the prefilter and the reconstruction filter; the combined frequency response (spectrum) is a multiplication of the prefilter spectrum (a box window from $-\frac{1}{2}f_s$ to $\frac{1}{2}f_s$) and the reconstruction filter spectrum (box window whose width is the same at scale factor 1, see Fig. 4(a), but stretches for increasing minification). The net result will be the box window from $-\frac{1}{2}f_s$ to $\frac{1}{2}f_s$ independent of the minification factor, which is the ideal. A practical approximation of these sinc filters leads to a similar story but with approximate box spectra; any leakage of the reconstruction filter spectrum multiplied in the stopband of the prefilter spectrum does no harm when the stopband consists of near-zero values; however attenuation due to the reconstruction-filter spectrum part that overlaps the passband of the prefilter lead to blurriness (Fig. 4(b)). The overlap of this part depends on the scale factor (except for Dirac-reconstruction whose spectrum is a constant of 1); when near to 1 most attenuation occurs; for increasing minification the influence of the reconstruction filter diminishes (compare Fig. 4(b) and (c)), i.e., the prefilter spectrum is "dominant" for the net result (in our minification case). These observations motivate us to use a good-quality approximate-sinc prefilter and choose a 'simple', less compute intensive, reconstruction filter.

2) Dirac Reconstruction: The simplest function is the Dirac. The spectrum of the Dirac function is a constant with value 1. So, when used for reconstruction the combined spectrum



(a) Normalization by division of the sum of the weights obtains jaggy artifacts.

(b) Normalization by multiplication with the local scale factor leads to DC-ripple artifact.

Fig. 5. Dirac reconstruction followed by normalization gives artifacts. Both images are produced by the same slight perspective mapping with a small minification factor (close to one) of a checkers texture with 16x16 blocks of equal gray texels. The images are shown using 16x pixel replication.

will nicely equal the prefilter spectrum. However, the Dirac reconstruction filter has undesirable artifacts, which we explain here. Rewriting (1) with Dirac reconstruction results in $j(\mathbf{x}) = \sum_{m^{-1}(\mathbf{x}) \in Z^2} i(m^{-1}(\mathbf{x})) \cdot h(\mathbf{x})$, which delivers two artifacts: 1) a point sample maps seemingly arbitrarily –fully– to either a pixel's prefilter footprint or to its neighbor's, leading to jaggy edges and intensity jumps. 2) A per-pixel varying scale factor can cause a varying number of texels to accumulate in a pixel, resulting in a changing sum of texel weights and thus a changing intensity with a constant intensity texture. To avoid these intensity changes a normalization strategy could be considered (although this does not follow from sampling theory): Dividing each pixel by the sum of its contributing texel weights avoids intensity variations but cannot solve the jaggy appearance (Fig. 5(a)). The jaggy edges could be solved by normalizing differently; multiplying texel weights with the ratio of their texel areas overlapping the prefilter footprint. However, this is known to lead to intensity fluctuations called DC-ripple [23] (Fig. 5(b)). So using the Dirac reconstruction filter leads to undesirable artifacts even if we introduce a normalization step that does not follow from the sampling theory (Section III-B).

3) Square-Box Reconstruction: After Dirac, the box function (Eq. 4) is the most simple to implement. When the scale factor is close to 1 a box-reconstruction filter spectrum (sinc) leaves many high frequencies above $0.5 f_s$ [37]; as mentioned (Section III-C1), this is not a problem when using a good enough quality prefilter, which will suppress them in the stopband. It also attenuates frequencies in the prefilter's passband (Fig. 4(b)), leading to limited excess blurriness. However, for increasing minification, the reconstruction filter spectrum stretches, and the attenuation in the prefilter passband reduces (Fig. 4(c)).

With a square filter footprint (as opposed to circular) the box function adheres to the unity-of-composition property (no DC-ripple), no normalization is needed, and it avoids jaggy edges. Together with its simple implementation, it motivated our choice of the box reconstruction function.

IV. Algorithm

Our algorithm provides an analytical way of evaluating the resampling formula (1) using a box-function reconstruction filter and a prefilter of an arbitrary piecewise-polynomial function over multiple rectangular regions, which we call *cells* (Fig. 6). Following the resampling steps and Fig. 3, our algorithm can be summarized as follows. The reconstruction (Step 1) with a square-footprint *box* filter transforms the texel samples into a continuous-space signal of square piecewise-constant regions

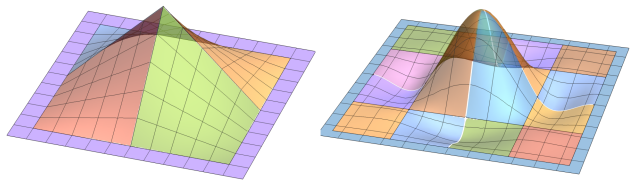


Fig. 6. Two examples of a rectangular piecewise-polynomial function; the 2D tent function consisting of 2x2 cells and a cubic function consisting of 4x4 cells; each cell defines its own polynomial function.

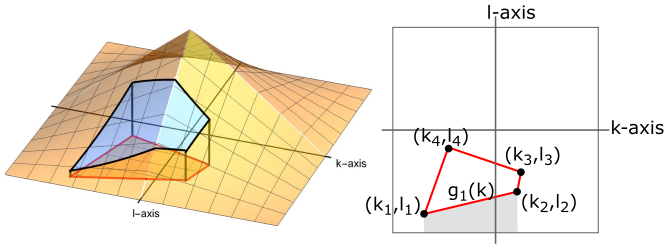


Fig. 7. Left: The red quadrilateral represents a texel quad of a mapped texel square. It acts as a switch to select a signed volume in prefilter h that represents the weight of the texel. Right: Top view of the same texel quad. The prefilter volume bounded by this quad equals the sum of the signed volumes to the south (marked gray for line g1) of the 4 corner-to-corner line segments.

TABLE I
Notations

$s : \mathbb{R}^2 \to \mathbb{R}$	sampling function
i_o / $i_s : \mathbb{R}^2 \to \mathbb{R}$	original / sampled input signal
$m : \mathbb{R}^2 \to \mathbb{R}^2$	mapping (perspective warping) function
$\mathbf{x}$ / $\mathbf{k}$	pixel position / position in *output/screen space*
$\mathbf{t}$ / $\mathbf{u}$	texel position / position in *input/texture space*
$r : \mathbb{R}^2 \to \mathbb{R}$	reconstruction filter (texture space)
$h : \mathbb{R}^2 \to \mathbb{R}$	prefilter (screen space)
$\mathbf{k}_n$	texel-quad corner with coordinates (k_n, l_n)
g_n	texel-quad edge defined by tangent t_n and point $(0, p_n)$
V_{t_n,p_n}	vertical antiderivative of h
H_{t_n,p_n}	slanted antiderivative of V_{t_n,p_n}
$w_\mathbf{t}$	$= \sum_{n=1}^{4} H_{t_n,p_n}(k_{n+1}) - H_{t_n,p_n}(k_n)$ texel weight

centered at texel samples. We call these regions *texel squares*. The perspective mapping of this signal (Step 2) results in adjacent quadrilaterals of constant value in screen space. The application of the prefilter (Step 3) contributes the texel value to a pixel according to a *texel weight* defined by the integration of the prefilter kernel bounded by the quadrilateral of the mapped and shifted texel square (Eq. 3) as illustrated in Fig. 7, which we will call a *texel quad*.

To obtain the texel weight we compute the integral of the prefilter over the texel quad: Loosely speaking, for a general texel-quad edge (with variable tangent and position), we define a 2D analytical integral that represents the signed volume (where the sign depends on the orientation of the segment) of the prefilter over the area to the south of the edge. We call this integral the *quad-edge weight* (signed volume above gray area in Fig. 7 right). The sum of the four quad-edge weights equals the texel weight. The main used notations in the following detailed description are summarized in Table I.

The texel-quad edges are derived from (3): With our reconstruction filter r being a box filter, the integrand equals $h(\mathbf{k})$ if $\mathbf{box}\big(m^{-1}(\mathbf{x} - \mathbf{k}) - \mathbf{t}\big)$ equals one, otherwise it is zero. This expression, a function of $\mathbf{k}$ for a fixed $\mathbf{t}$ and $\mathbf{x}$, thus, acts as

an "on/off-switch" (of the texel-quad interior) applied to the prefilter function h. For a $\mathbf{k}$ inside the texel quad, we have $\mathbf{abs}(m^{-1}(\mathbf{x}-\mathbf{k})-\mathbf{t}) < \begin{pmatrix}0.5\\0.5\end{pmatrix}$. Here both the bold printed "**abs**" and "$<$" are considered per vector element. If we let $\mathbf{t} := (s,t) \in \mathbb{Z}^2$, the four texel-quad corners can be found by solving for $\mathbf{k}$ in all combinations of

$$\mathbf{abs}(m^{-1}(\mathbf{x}-\mathbf{k})-\mathbf{t}) = \begin{pmatrix}0.5\\0.5\end{pmatrix} \Rightarrow \mathbf{k} = \mathbf{x} - m\begin{pmatrix}s\pm 0.5\\t\pm 0.5\end{pmatrix} \tag{5}$$

We number the texel-quad corners $\mathbf{k}_1 \ldots \mathbf{k}_4$ in counter-clockwise order (illustrated in Fig. 7) and define for convenience $\mathbf{k}_5 := \mathbf{k}_1$ and denote the coordinates $(k_n, l_n) := \mathbf{k}_n$. From these corner points, we can derive the line function of the texel-quad edges $g_n : \mathbb{R} \to \mathbb{R}$ described by a tangent t_n and going-through point $(0, p_n)$:

$$g_n(k) := t_n k + p_n \quad \text{with} \quad t_n = \frac{l_{n+1}-l_n}{k_{n+1}-k_n},$$
$$p_n := l_n - k_n t_n \tag{6}$$

For now, we assume that g_n is not vertical. With (6) and each edge considered a north border of the vertical integral of h (where the sign depends on the orientation of the segment) we obtain four signed integrals, whose sum equals the weight of a texel at $\mathbf{t}$ for a pixel at $\mathbf{x}$:

$$w_{\mathbf{t}}(\mathbf{x}) = \sum_{n=1}^{4} \int_{k_n}^{k_{n+1}} \int_{-\infty}^{g_n(k)} h(k,l)\, \mathrm{d}l\, \mathrm{d}k \tag{7}$$

We develop (7) into a discrete computational form using antiderivatives. We adopt the traditional notation for an antiderivative $\int f(x)dx := \int_a^x f(t)dt$ using the indefinite integration symbol (without bounds); a can be chosen freely but to represent integration from a specific starting point a is made equal to that starting point. Defining the vertical antiderivative $\int h(k,l)\, \mathrm{d}l =: V(k,l)$ we can rewrite $\int_{-\infty}^{g_n(k)} h(k,l)\, \mathrm{d}l = \int_b^{g_n(k)} h(k,l)\, \mathrm{d}l = V(k, g_n(k)) - V(k,b)$. With h having finite support and b chosen below the support, $V(k,b) = 0$. Similarly, defining the slanted antiderivative $\int V(k, g_n(k))\, \mathrm{d}k =: H_{t_n,p_n}(k)$, we may write:

$$w_{\mathbf{t}}(\mathbf{x}) = \sum_{n=1}^{4} H_{t_n,p_n}(k_{n+1}) - H_{t_n,p_n}(k_n) \tag{8}$$

By following the recipe of (8), with the texel-quad corner points defined in (5) and deriving an analytical expression for H_{t_n,p_n} for a given prefilter, we obtain an exact computation of the resampling formula (1) (up to numerical precision).

A. Determining the Prefilter Integral

So far, this section explained how the texel weight $w_{\mathbf{t}}$ can be computed from the slanted antiderivatives H_{t_n,p_n}. Here, we will show how to derive these antiderivatives for a piecewise polynomial filter over multiple cells.

For a multi-cell polynomial function, the end points of a quad edge may fall in different cells, hence H_{t_n,p_n} spans multiple cells. To compute the quad-edge weight we could clip the quad edge into per-cell line segments and proceed with accumulating the piecewise contribution of their antiderivatives (each starting at value zero). However, evaluating the per-cell antiderivatives at each of the crossed cell borders can become rather costly, especially for high-order many-cell polynomials, where typically many crossings appear and more expensive high-order power terms are to be evaluated.

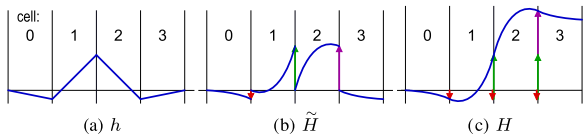


Fig. 8. (a) A piecewise function h. (b) A per-cell antiderivative H of it that starts at zero at the cell's left border. (c) The multi-cell antiderivatives H obtained by consecutively adding the per-cell H integration-result value (accumulation arrows) to each per-cell H antiderivative.

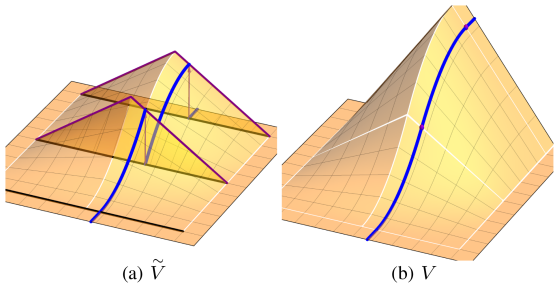


Fig. 9. (a) Per-cell vertical antiderivative V (starting at zero at each cell's south border) of a 2x2 multi-cell tent prefilter function (Fig. 6-left). (b) Multi-cell vertical antiderivative V obtained by consecutively adding, from south to north, the per-cell integration result of V (purple line; a function of k), to the next per-cell antiderivative. For illustration purposes we show this also on the prefilter's north border (not needed by the algorithm).

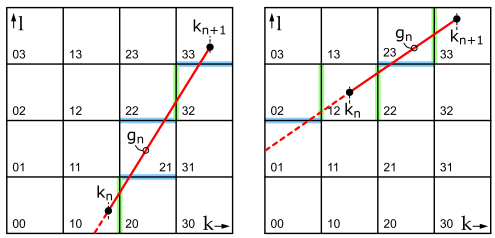


Fig. 10. The set of crossed cell borders can vary strongly depending the quad-edge line function g_n (and thus on t_n and p_n). This causes the determination of the start value of the antiderivative of the cell containing a quad-edge endpoint to become a complicated operation, since it requires the consecutive sum of the cell's integration results along the quad edges up to the endpoint cell. A cell index is shown in each cell.

Instead we are interested in a more efficient alternative that allows for computing of the quad-edge weight by evaluating a multi-cell antiderivative only at the quad-edge end points, similar to the 1D case as illustrated in Fig. 8. In the 1D case, the multi-cell antiderivative for a cell, can be derived by consecutively adding, from left to right, the per-cell integration result to the next cell's antiderivative.

In the 2D vertical-integration case, obtaining the multi-cell antiderivative ($V(k,l) = \int h(k,l)\, \mathrm{d}l$) is similar to the 1D case (Fig. 9(a) and (b)). If the integration is not vertical, the slanted integration case ($\int V(k, g_n(k))\, \mathrm{d}k$) is more difficult. We can still derive it via multi-cell antiderivatives H_{t_n,p_n} for each quad-edge in the prefilter support, but now any quad edge can cross many combinations of cell borders depending on t_n and p_n (Fig. 10) and one would have to evaluate an integration for each cell iteratively to handle the integration constants.

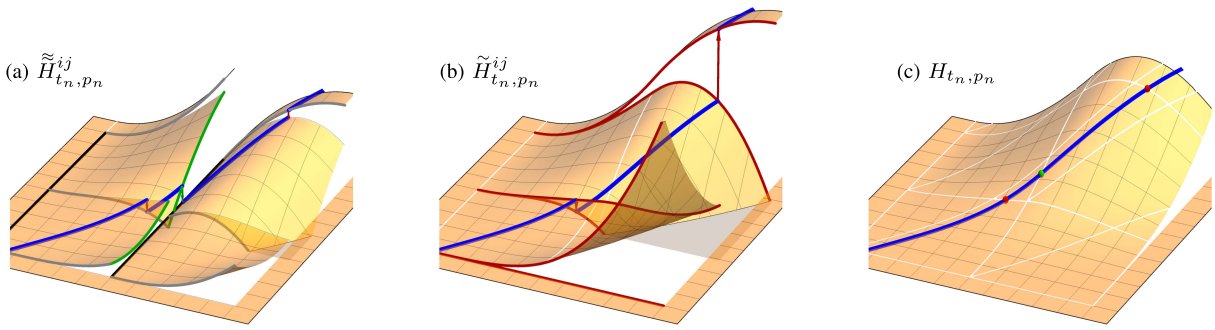


Fig. 11. Continuing Fig. 9: (a) Per-cell slanted antiderivative $H^{ij}_{t_n,p_n}$ of V (blue, shown for $t_n = +1.4$ and variable p_n). It start at zero in each cell's west border (black) and is discontinue at both east (green) and south borders (grey). (b) Horizontal-multi-cell slanted antiderivative $H^{ij}_{t_n,p_n}$, obtained by adding the per-cell integration result value of $H^{ij}_{t_n,p_n}$ (from west to east) to the following eastbound cell. We derive an analytical function of t_n, p_n and l (green) for these values. (c) We obtain our proposed $H^{ij}_{t_n,p_n}$ to evaluate the quad-edge weight by consecutively removing the differences between the antiderivative $H^{ij}_{t_n,p_n}$ of adjacent cells at south-border crossings (two red curves per border). We obtain this difference via another analytical expression of t_n, p_n and k per border. For illustration purposes, we even show these antiderivative in cells north of the 2x2 prefilter; these are not needed by the algorithm.

Alternatively, for each of the combination of crossed borders, we could precompute the expressions. While this approach would then allows evaluating the quad-edge weight by evaluating the suitable multi-cell antiderivative only at quad-edge end points, the high combinatorial complexity, resulting from the various combinations of crossings, can make this solution quickly non-practical, especially when many cells are used to define a piecewise-polynomial prefilter.

We have chosen an 'in-between' method that reduces the possible combinations at the cost of some additional computations. It leads to a more straightforward implementation: First we obtain the multi-cell antiderivative $H^{ij}_{t_n,p_n}$ (with ij a cell identifying index) valid for quad edge combinations that only cross *vertical* cell borders; reducing the combinatorial complexity. Similar as in 1D, each left cell's consecutive sum of per-cell integration results is added to the per-cell antiderivative ($H^{ij}_{t_n,p_n}$, Fig. 11(a)). But instead of adding the consecutive sum as a value, the *function* (of t_n and p_n) of the consecutive sum is derived and analytically added (Fig. 11(b)). If a given quad edge does not cross a horizontal cell border, this multi-cell antiderivative H_{t_n,p_n} can be used directly. If there is a crossing, we need to add a difference value to a cell's $H^{ij}_{t_n,p_n}$ for any crossing with the cell's south border (for a positive tangent t_n) or north border (for a negative tangent t_n), for which we rely on the added analytical expression. We explain the approach for a positive tangent. For each cell's south-border crossing we consecutively add a difference value equal to the subtraction of $H^{ij}_{t_n,p_n}$ of both adjacent cells at the crossing. This difference value is evaluated from a simpler difference function of only t_n and p_n at a fixed vertical position l' (red curves in Fig. 11(b)): $D^{ij}_{t_n,p_n}(l') := H^{i,j-1}_{t_n,p_n}(g_n^{-1}(l')) - H^{ij}_{t_n,p_n}(g_n^{-1}(l'))$, where the crossing-point horizontal position equals $g_n^{-1}(l') = (l' - p_n)/t_n$ (following Eq. 6).

Both $H^{ij}_{t_n,p_n}$ and $D^{ij}_{t_n,p_n}$ are derived beforehand analytically and are algebraically simplified to enable an efficient evaluation. For example, evaluating $H_{t_n,p_n}(k_{n+1})$ at the right quad-edge endpoint of Fig. 10-right equals $H^{33}_{t_n,p_n}(k_{n+1}) + D^{02}_{t_n,p_n} + D^{23}_{t_n,p_n}$. When the slant angle is negative a similar process applies in a south-north mirrored configuration. Now that we can evaluate $H_{t_n,p_n}(k)$, we follow (8) to calculate w_t as the sum of four terms on the interval $[k_n, k_{n+1}]$ for each texel-quad edge.

V. Algorithmic Details

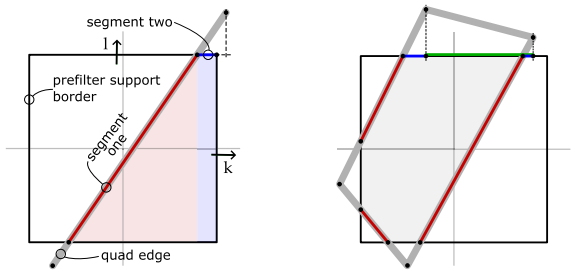
There are two additional elements that need to be taken into account when using our method: First, we will detail how to handle texel quads that extend beyond the prefilter's support (Section V-A). Second, we will discuss the corner case of near-vertical quad edges to ensure numerical stability (Section V-B).

A. Texel-Quad Clamping

In the general case a texel quad extends beyond the prefilter's support; to restrict the computation of the quad-edge weight to the support we "*clamp*" the edge into two potentially zero-length segments, whose weight together equals the quad-edge weight. Segment one is the quad edge clipped against the support borders. If the quad edge intersects with the north border the weight of segment one needs to be complemented; this is done by the weight of segment two, which is a vertical projection of the line segment from the north-border clip point to the adjacent quad-edge end point onto the north border and clipped against the vertical borders (Fig. 12).

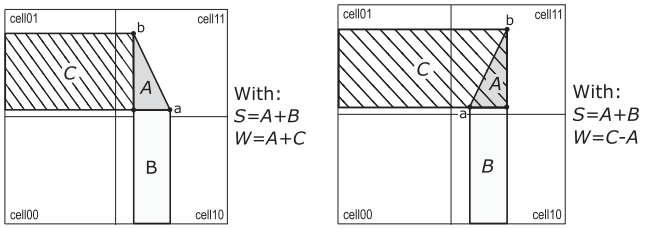
B. Near-Vertical Quad Edges

The floating-point evaluation of (8) is numerically unstable for near-vertical quad edges; this is due to higher-order power-of-t_n terms in H_{t_n,p_n}, which overflow for large tangent values. A simple solution would limit the tangent to a suitable range to avoid overflow, but our focus is on an accurate solution. As depicted in Fig. 13, we will compute the weight of the edge (area to the south S) via a detour. First, we transpose the near-vertical quad edge to a near-horizontal (by swapping k and l coordinates of both endpoints) and evaluate its weight. This evaluates the



(a) Quad edge overlapping support (b) Texel quad overlapping support

Fig. 12. (a) To obtain the proper texel-quad edge weight we must evaluate its integral only over the prefilter support; for this we subdivide the quad edge in two segments: 1) the quad edge clipped against the support borders (red) and 2) the quad edge to the north of the support and vertically projected on the north-border and clipped against the vertical borders (blue). (b) The signed sum of all quad-edge weights equals the texel weight.



(a) Negative edge slope. (b) Positive edge slope.

Fig. 13. To determine the edge weight $S = A{+}B$ of a near-vertical texel-quad edge, we avoid numerical instability by evaluating the weight of the transposed edge $W = A{+}C$ and the weights of the rectangular areas B (non-transposed) and C (transposed). We can then compute $S = W - C{+}B$ or $S = C - W{+}B$, depending on the slope of the edge.

area to the west (W) of the original edge. We observe that both S and W share the integration over an orthogonal triangle A, whose hypotenuse is defined by the quad-edge. They differ by the integration over the rectangular areas B (to the south of the triangle's horizontal leg) and C (to the west of the triangle's vertical leg) We need to distinguish two cases depending on the edge's slope. For a positive slope, we have $S = A + B$ and $W = C - A$, thus, $S = C - W + B$. Otherwise, we have $S = W - C + B$

A final observation is that the edge of area B and transposed edge of area C is horizontal: $t_n = 0$. Instead of evaluating H_{t_n,p_n}, we derive a simplified expression for H_{0,p_n}.

C. Extension to Textured Triangle Meshes

Until now, we focused on texels that are rendered as complete quads. When textures are mapped to a 3D model, it can be that these quads are cut by triangle edges. There exist many supersampling variants of triangle-edge anti-aliasing methods, however these only obtain an approximate result and are exact only in the limit. Our solution naturally extends to exact triangle-edge anti-aliasing. For a given triangle, we remove the texel-quad area that falls outside the triangle before computing the integration of the prefilter over it. Hence using the same prefilter for texture and triangle-edge anti-aliasing.

For general meshes, we need to implement an exact geometrical hidden-surface-removal algorithm. The resulting triangles are then processed from front to back (based on the nearest vertex). For each triangle, we rasterize its texel quads, using a *conservative* rasterization that selects all texel quads whose texel square overlaps with the triangle in texture space. We clip the resulting texel quads against the triangle and against the geometrical union of earlier added clipped texel quads, potentially splitting a texel quad into multiple polygons. We then determine the weight of each polygon and accumulate the result in the pixel's color value.

VI. Implementation

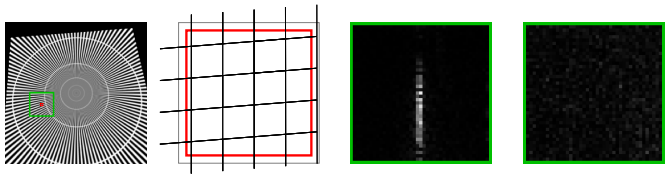
We implemented three versions of our analytical texture mapping (ATM) algorithm in a triangle-based renderer; two C++ CPU versions (forward texture-mapping, *CPU-F*, and inverse texture-mapping, *CPU-I*), as well as a GPU version, *GPU*. The latter demonstrates compatibility with hardware-accelerated 3D pipelines, although our solution is better suited for a forward-texture mapping pipeline. To validate the exactness, we implemented a *reference* texture-mapping algorithm using brute-force supersampling.

Without the edge anti-aliasing of Section V-C both, CPU-F and CPU-I use a custom triangle rasterizer, which traverses sample positions respectively in texture space and screen space, considering positions within a triangle according to tie-breaking rules [38]. When edge anti-aliasing is enabled, CPU-F determines the contributing geometrical overlap of a texel quad with a triangle using Clipper [39] and rasterization must be more conservative; it considers texels whose texel square overlap the triangle. The texel-quad processing of CPU-F and CPU-I are identical, although CPU-I requires the additional work of mapping each pixel's prefilter support to a a quadrilateral (*preimage*) in texture space and determining the texel quads that overlap it. CPU-I uses the software emulation of GLSL shader intrinsics (GLM [40]) and forms the basis of the GPU C++/OpenGL implementation. In short, its fragment shader computes a minimum bounding rectangle of the preimage in normalized texture space using the `fwidth` shader instruction (available as from OpenGL 1.1) and the width of the prefilter, and maps each texel within it to screen space, where prefiltering is applied according to the ATM algorithm.

All three implementations support box, tent and multiple cubic prefilters, among others a strong up-sharpening (Mitchell [34] with B = 0, C = 1), Catmull-Rom $\left(B = 0, C = \frac{1}{2}\right)$, a balanced sharpening/aliasing (B = $\frac{1}{3}$, C = $\frac{1}{3}$) and the non-aliasing B-spline (B = 1, C = 0). In the GPU version, we added a 'generic' cubic prefilter enabling variable B and C. We used Mathematica to derive for each of these filters the antiderivatives for the cell functions V^{ij}, $H^{ij}_{t_n,p_n}$ and the cell difference functions $D^{ij}_{t_n,p_n}$ (Section IV-A). Our implementation should be considered a proof of concept for a reference renderer and leaves room for performance optimizations.

VII. Evaluation

All tests are performed on an a laptop with AMD Ryzen 7 4800H processor and NVIDIA GeForce RTX 3060 GPU. The CPU tests run on a single core at 4.2 GHz.



(a) texture (b) prefilter (c) regular (d) Hammersley

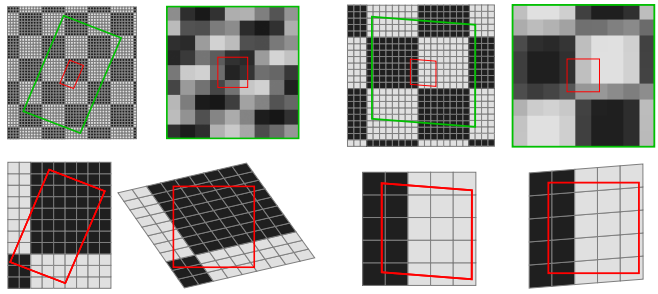
Fig. 14. Difference of regular (c) and Hammersley (d) supersampling with ATM (Tent prefilter) using 2048^2 sub-samples per pixel on an area of 41x41 (green square in (a)) texture mapped pixels of a Siemens star (512 × 512 texels, minified roughly 2x). The center of the area contains texel quads with near-vertical borders; illustrated in the zoomed-in prefilter footprint in (b). Both (c) and (d) show the same intensity range [0, 0.000012] but regular sampling suffers from reduced sampling resolution close to the horizontal direction.

A. *Evaluation Exactness of Texture Filtering*

We produce references via supersampling; each sample in the screen-space prefilter footprint is inversely mapped to texture space to fetch the nearest texel color. Due to their faster convergence than regular sampling (Fig. 14), we evaluate different sampling methods, including regular sampling, Hammersley pattern with base of 2 [41], and a fast Sobol sampling [42].

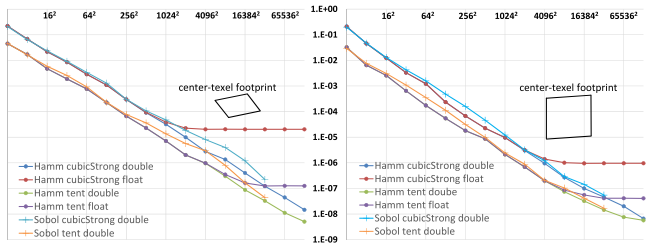
We measure the root-mean-square deviation (RMSD) between ATM (CPU-I) and supersampling for an increasing sample rate using double-precision floating-point values. We show that the supersampling solution approaches our analytical result, when increasing the amount of samples. In this way, we empirically validate the correctness of our analytic derivation.

We evaluate two examples (one with and one without near-vertical quad edges) each for two prefilter functions (a 2 × 2 cells tent function and a 4 × 4 cells cubic (B=0, C=1) function), for a total of four scenarios. To maximize the filtering effects, we use a high-contrast checkers texture with tiles of 8 × 8 texels and intensity either 0.12 or 0.88 on the unit [0..1] range, leaving head room for under/over-shoot due to negative-lobe prefilters. To reduce render time for the higher supersample factors, we render a very small comparison image; only 9 × 9 pixels, illustrated in Fig. 15). As a baseline, we compare to a slow Hammersley sample generator (code from [41]). It enables very high supersample factors $2^{17} \times 2^{17}$ but also took about 41 hours. Please note (Fig. 16) that even with this extreme supersampling, there is still a difference to our solution, which is reduced by using more samples. A recent fast generator, Sobol "maximized minimum distance" [42] only supports a maximum supersample factor of $2^{15} \times 2^{15}$ but is much faster; 39 and 42 minutes for the tent and cubicStrong prefilter respectively. The fastest alternative is a regular grid sampling requiring 22 minutes (cubicStrong) but showing slower convergence. For all four scenarios, we observe a decrease of the RMSD with increasing sample rate until arithmetic precision becomes a limiting factor (Fig. 16). This illustrates the validity of our ATM-I approach, which produces the result in 0.5 to 12 milliseconds, depending on the scenario. As expected, the convergence is roughly linear when samples quadruple. One exception are the highest sample rate counts in the near-vertical quad edges case with a tent



(a) No near-vertical quad edges. (b) Near-vertical quad edges.

Fig. 15. This figure illustrates two test mappings (a) and (b). The top row shows how the texture (left) maps to 9x9 screen pixels (right). The bottom left (texture) zooms in on the preimage (red quadrilateral) and shows the overlapping texel squares and the bottom right (screen) shows how the associated texel quads overlap the prefilter footprint. Mapping (a) equals a rotation and a modest minification. Mapping (b) comprises a vertical shear, almost no rotation and a bit less minification, obtaining near-vertical quad-edges with $|tangent| > 100$ that are transposed filtered (Section V-B).



(a) Without near-vertical quad edges (b) With near-vertical quad edges

Fig. 16. Measured RMSD of ATM (CPU-I) and Hammersley and Sobol sampling using a tent and a strong up-sharpening cubic prefilter in the examples of Fig. 15. The RMSD reduces linearly when the sample count quadruples (horizontal axis). Single-precision arithmetic shows the same RMSD linear reduction but flattens earlier (around 10^{-5} and 10^{-7}) due to the lower numerical precision.

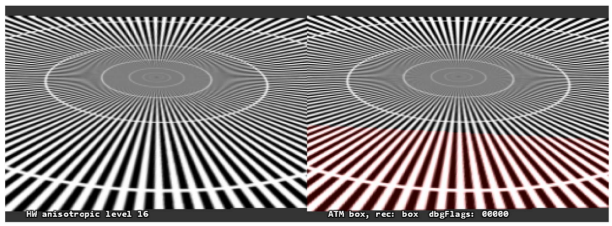


Fig. 17. Screenshot of the GPU demo enabling side-by-side visual comparison of aliasing-critical textures mapped on a slowly moving plane of triangles (red is magnified). This picture is not suited for quality assessments in a pdf viewer.

prefilter; the graph flattens. By comparing to single-precision floating-point arithmetic it shows that this is likely related to numerical imprecision and it illustrates our approach's accuracy being only limited by machine precision.

B. *Evaluation of GPU Version*

To make plausible that our GPU version is exact, up to machine precision, we generated a sequence of 32-bit floating-point per channel images with a textured plane (shown in Fig. 17) for

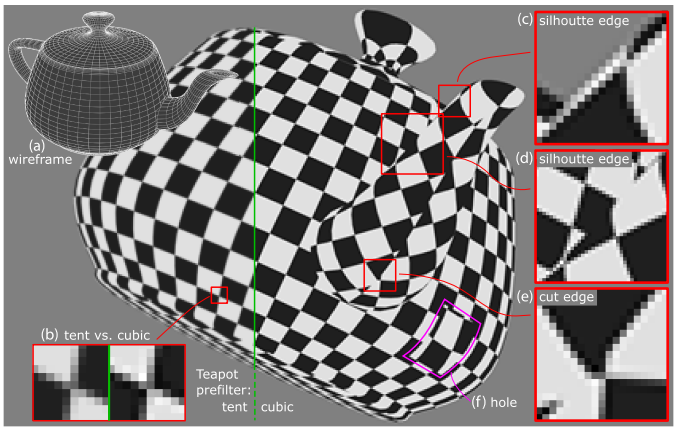


Fig. 18. ATM rendering of Utah Teapot [43] (15704 double-sided textured triangles with same prefilter for texture and triangle-edge anti-aliasing; tent to the left of the green line and an up-sharpening cubic to the right. (a) Wireframe. (b) Magnification showing filter differences on same area. (c) Zoom on silhouette edges (notice under/over-shooting). (d) Silhouette edge a.o. over areas of same intensity (notice correct intensity constancy). (e) Correct result at complex triangle/texture configuration. (f) Correct edge rendering even when omitting a rectangular triangle patch in the front. For inspection: a pdf-viewer will show pixel replicated zoom, if supported (e.g., SumatraPDF).

a changing perspective mapping, with both the CPU-F and the GPU version. The RMSD between them measured in an area inside the plane, stayed below 1.0^{-5} for all frames, which is in line with the RMSD values of Fig. 16.

We visually compared our GPU implementation to anisotropic filtering [11] using real-time rendering. Anisotropic filtering can be considered an approximation of box reconstruction filter and box prefilter. This is because it applies multiple trilinear probes selected from appropriate mipmap levels along the line of anisotropy and averages them; each trilinear probe consist of two bilinear probes, each applying four texel weights to the 4 nearest texels, each weight is equal to the area-overlap of a square footprint centered around the bilinear probe position –the *box prefilter*– with the texel square –the *box-reconstruction filter*– of one of the 4 nearest texels. To 'fairly' compare the quality, we use ATM with a box prefilter and show a slowly animating plane with an aliasing sensitive texture (512×512 Siemens star) rendered (mainly minified) at a 512×512 viewport (Fig. 17). ATM shows a bit less aliasing and more sharpness than $16\times$ dedicated hardware anisotropic filtering. Interestingly, selecting the tent function as a prefilter, which is known to lead to more blur than a box-function, yet, the ATM visually shows equal sharpness compared to anisotropic filtering, but much less aliasing. Rendering speed was not our focus, but, here, it is roughly 0.9, 4 and 40 milliseconds with box, tent, and cubic prefilter, respectively.

C. Evaluation Exactness on Textured Triangle Meshes

To be able to optimally inspect filtering effects on triangle borders (Section V-C) we render (CPU-F) the Utah Teapot without lighting; we applied a 1024×1024 texture and rendered into a 512×512 viewport, obtaining a modest (horizontal and vertical) texture minification at the front of the teapot and the sprout; magnification only appears at the teapot knob (Fig. 18). We used the same checkers texture as in Section VII-A but with tiles of 16×16 texels. To verify exactness, we placed a screen-filling quad behind the teapot and accumulated texel weights for each pixel. In consequence, the accumulations should be close to one. We measured a maximum deviation over all screen pixels less than 10^{-12}, for the box, tent and the cubic prefilter during a rotating-teapot sequence, which had all outside facing triangles visible at least once. This test together with Section VII-A illustrate that ATM filtering can also implement exact triangle-edge anti-aliasing.

Fig. 19. ATM renderings without lighting using up-sharpening cubic prefilter for texture and triangle-edge anti-aliasing. Both models [44] (14894 (left) and 40310 triangles) have applied a 20248×2048 texture atlas and are rendered into 512×512 viewport obtaining minification everywhere. For close inspection: a pdf-viewer, if supported, will show pixel replicated zoom.

Fig. 19 shows color-textured ATM renderings (CPU-F) whose measured per-pixel accumulated weight deviates less than 10^{-12} from one, for the box and the cubic prefilter.

VIII. Conclusion and Future Work

We have presented analytical texture mapping, an analytical exact resampling algorithm derived from sampling theory. The solution can be applied in the 3D graphics pipeline for texture anti-aliasing and for triangle-edge anti-aliasing in textured meshes. It supports general 2D integrable prefilter functions. We have implemented it on CPU and GPU for piece-wise polynomial functions, enabling general prefilter-function approximations with arbitrarily high accuracy. We showed that our algorithm's accuracy is only limited by arithmetic precision and can be used as a reference for image-quality comparisons.

Future work could focus on additional performance optimizations, e.g., utilizing the symmetry of the prefilter or Shader optimizations (such as replacing the per-cell conditional polynomial evaluation by a single polynomial whose coefficients are indexed from a table depending on the cell). Additionally, extensions towards material filtering [45] or even physically-based rendering (e.g., including general camera models) seem promising. For example, Voronoi cell volumes around screen-space samples can improve convergence [46] and could be handled.

Acknowledgment

The authors thank Markus Billeter for useful suggestions and OpenGL support, Guowei (Peter) Lu and Christoph Peters for valuable feedback.

REFERENCES

[1] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," PhD Thesis, Comput. Sci., Univ. Utah, Salt Lake City, UT, USA, Dec. 1974.

[2] H. Nyquist, "Certain topics in telegraph transmission theory," *Trans. Amer. Inst. Electr. Engineers*, vol. 47, no. 2, pp. 617–644, Apr. 1928.

[3] C. E. Shannon, "Communication in the presence of noise," in *Proc. Inst. Radio Engineers*, vol. 37, no. 1, pp. 10–21, Jan. 1949.

[4] G. Wolberg, *Digital Image Warping*. Washington, DC, USA: IEEE Computer Society Press, 1990.

[5] A. R. Smith, "Digital filtering tutorial for computer graphics," Tutorial SIGGRAPH '83, 1983. [Online]. Available: http://www.alvyray.com

[6] P. S. Heckbert, "Fundamentals of texture mapping and image warping," MSc Thesis, Dept. EECS, Univ. California at Berkeley, Berkeley, CA, 1989.

[7] V. A. Kotelnikov, "On the transmission capacity of the 'ether' and of cables in electrical communications," in *Proc. 1st All-Union Conf. Technol. Reconstruction Commun. Sector Low-Current Eng.*, 1933. [Online]. Available: http://ict.open.ac.uk/classics

[8] A. R. Smith, *A Biography of the Pixel*. Cambridge, MA, USA: MIT Press, 2021.

[9] A. S. Glassner, *Principles of Digital Image Synthesis*. San Mateo, CA, USA: Morgan Kaufmann, 1995. [Online]. Available: https://www.glassner.com/portfolio/principles-of-digital-image-synthesis/

[10] A. Glassner, "Adaptive precision in texture mapping," *ACM SIGGRAPH Comput. Graph.*, vol. 20, no. 4, pp. 297–306, 1986.

[11] J. P. Ewins, M. D. Waller, M. White, and P. F. Lister, "Implementing an anisotropic texture filter," *Comput. Graph.*, vol. 24, no. 2, pp. 253–267, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0097849399001594

[12] R. J. Cant and P. A. Shrubsole, "Texture potential MIP mapping, a new high-quality texture antialiasing algorithm," *ACM Trans. Graph.*, vol. 19, no. 3, pp. 164–184, Jul. 2000.

[13] B. Chen, F. Dachille, and A. Kaufman, "Footprint area sampled texturing," *IEEE Trans. Vis. Comput. Graphics*, vol. 10, no. 2, pp. 230–240, Mar./Apr. 2004.

[14] E. A. Feibush, M. Levoy, and R. L. Cook, "Synthetic texturing using digital filters," *ACM SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 294–301, Jul. 1980.

[15] N. Greene and P. S. Heckbert, "Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter," *IEEE Comput. Graph. Appl.*, vol. 6, no. 6, pp. 21–27, Jun. 1986.

[16] J. McCormack, R. Perry, K. I. Farkas, and N. P. Jouppi, "Feline: Fast elliptical lines for anisotropic texture mapping," in *Proc. Annu. Conf. Comput. Graph. Interactive Techn.*, 1999, pp. 243–250.

[17] C. Soler, M. M. Bagher, and D. Nowrouzezahrai, "Efficient and accurate spherical kernel integrals using isotropic decomposition," *ACM Trans. Graph.*, vol. 34, no. 5, Nov. 2015, Art. no. 161.

[18] J. F. Blinn and M. E. Newell, "Texture and reflection in computer generated images," *Commun. ACM*, vol. 19, no. 10, pp. 542–547, Oct. 1976.

[19] J. Tumblin and B. Guenter, "High quality image warp filters on pyramids," Georgia Inst. Technol., Atlanta, GA, GVU Technical ReportGIT-GVU-93–04, 1993. [Online]. Available: http://hdl.handle.net/1853/3611

[20] K. Khoshelham and A. Azizi, "Kaiser filter for antialiasing in digital photogrammetry," *Photogrammetric Rec.*, vol. 19, no. 105, pp. 22–37, Jul. 2003.

[21] J. Manson and S. Schaefer, "Cardinality-constrained texture filtering," *ACM SIGGRAPH Comput. Graph.*, vol. 32, no. 4, pp. 140:1–140:8, 2013.

[22] E. Catmull and A. R. Smith, "3-D transformations of images in scanline order," *ACM SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 279–285, 1980.

[23] K. Meinds and B. Barenbrug, "Resample hardware for 3D graphics," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graph. Hardware*, 2002, pp. 17–26.

[24] M. Kallay and J. Lawrence, "Improving the two-pass resampling algorithm," *J. Graph. Tools*, vol. 8, no. 2, pp. 31–39, 2003.

[25] L. Williams, "Pyramidal parametrics," *ACM SIGGRAPH Comput. Graph.*, vol. 17, no. 3, pp. 1–11, Jul. 1983.

[26] R. C. Lansdale, "Texture mapping and resampling for computer graphics," MSc Thesis, Elect. Eng., Univ. Toronto, Toronto, ON, Canada, 1991.

[27] F. C. Crow, "Summed-area tables for texture mapping," *ACM SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 207–212, Jan. 1984.

[28] G. Lu, J. Guo, P. Kellnhofer, and E. Eisemann, "Sheared polygonal texture filtering," in *Proc. Graphics Interface Conf.*, 2024, Art. no. 19. [Online]. Available: http://graphics.tudelft.nl/Publications-new/2024/LGKE24a

[29] J. Manson and S. Schaefer, "Analytic rasterization of curves with polynomial filters," *Comput. Graph. Forum*, vol. 32, no. 2, pp. 499–507, 2013.

[30] T. Auzinger, M. Guthe, and S. Jeschke, "Analytic anti-aliasing of linear functions on polytopes," *Comput. Graph. Forum*, vol. 31, pp. 335–344, 2012.

[31] T. Duff, "Polygon scan conversion by exact convolution," in *Proc. Raster Imag. Digit. Typogr.*, Cambridge, U.K.: Cambridge Univ. Press, Oct. 1989, pp. 154–168.

[32] J. Olson, "Exact polygonal filtering," Web article, Aug. 2024. [Online]. Available: https://jonathanolson.net/exact-polygonal-filtering

[33] R. E. Crochiere and L. R. Rabiner, *Multirate Digital Signal Processing*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1983.

[34] D. P. Mitchell and A. N. Netravali, "Reconstruction filters in computer graphics," *ACM SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 221–228, 1988.

[35] E. Fiume and L. Williams, "Fast space-variant texture filtering techniques," in *SPIE Proc.*, vol. 1610, pp. 109–120, Feb. 1992.

[36] A. R. Smith, "A pixel is not a little square," Microsoft Technical Memo 6, Jul. 1995. [Online]. Available: http://alvyray.com/Memos/CG/Microsoft/6_pixel.pdf

[37] J. F. Blinn, "Jim blinn's corner–Return of the jaggy (high frequency filtering)," *IEEE Comput. Graph. Appl.*, vol. 9, no. 2, pp. 82–89, Mar. 1989.

[38] M. D. McCool, C. Wales, and K. Moule, "Incremental and hierarchical hilbert order edge equation polygon rasterization," in *Proc. Eurographics/SIGGRAPH Graph. Hardware Workshop*, 2001, pp. 65–72.

[39] A. Johnson, "Clipper, C library for clipping and offsetting lines and polygons, v6.4.2," 2017. [Online]. Available: http://www.angusj.com/delphi/clipper.php

[40] G-Truc Creation, "GLM OpenGL mathematics library," 2020. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[41] T.-T. Wong, W.-S. Luk, and P.-A. Heng, "Hammersley and halton points," *J. Graph. Tools*, vol. 2, no. 2, pp. 9–24, 1997. [Online]. Available: https://appsrv.cse.cuhk.edu.hk/ttwong/papers/udpoint/udpoints.html

[42] A. G. M. Ahmed, "An implementation algorithm of 2D sobol sequence fast, elegant, and compact," in *Proc. Eurographics Symp. Rendering*, 2024. [Online]. Available: https://dblp.org/rec/conf/rt/Ahmed24.html?view=bibtex

[43] Utah teapot 2011, original by Martin Newell 1975, "Online McGuire computer graphics archive," 1975. [Online]. Available: https://casual-effects.com/data/

[44] K. Zhou, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.-Y. Shum, "TextureMontage: Seamless texturing of arbitrary surfaces from multiple images," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1148–1155, 2005.

[45] M. Pharr, B. Wronski, M. Salvi, and M. Fajardo, "Filtering after shading with stochastic texture filtering," in *Proc. ACM Comput. Graph. Interact. Techn.*, vol. 7, no. 1, May 2024, Art. no. 14, doi: 10.1145/3651293.

[46] J. Guo and E. Eisemann, "Geometric sample reweighting for Monte Carlo integration," *Comput. Graph. Forum*, vol. 40, no. 7, pp. 109–119, 2021, doi: 10.1111/cgf.14405.

Koen Meinds received the graduate degree in computer science from Leiden University, in 1994, and worked for more than 30 years in the industry (research, concept creation, development and product integration) in fields related to computer graphics, image/video processing and signal processing. He is currently employed as expert researcher with SeeCubic working on 3D-display image processing topics. His research interests include resampling, perceptual rendering, optical systems, 3D-display techniques and measurement algorithms.

Elmar Eisemann is a full professor heading the Computer Graphics and Visualization Group, Department of Intelligent Systems, Delft University of Technology. His research interests include real-time and perceptual rendering, visualization, alternative representations, shadow algorithms, global illumination, and GPU acceleration techniques.