Collaborative Texture Filtering

T. Akenine-Möller^(D), P. Ebelin^(D), M. Pharr^(D), and B. Wronski^(D)

NVIDIA



Figure 1: During texture magnification, where previously only stochastic texture filtering (STF) was a viable option due to the expense of texel evaluation, our algorithms often produce zero-error results at negligible cost. We compare original One-tap STF [PWSF24] and Wave Communication STF [WPAM25] to our approach, both with and without DLSS [NVI25]. For this scene, one of our methods uses only 0.34 texel evaluations per pixel on average, compared to 4 for classic bilinear interpolation, and yet renders an image with extremely low error. On the right, we show false color visualizations of the scene, where the color maps use black to indicate waves with at least one pixel that uses minification. The top image shows magnification factors ranging from ~ 0.0 to 9.1 (average: 4.3) and the bottom image illustrates how many unique texels are needed per wave (8×4 pixels) to achieve perfect bilinear filtering. The bottom color map uses brown to indicate waves where this is not reached with ≤ 1 texel evaluation per pixel. For all the other waves, our method produces all 2×2 texels needed for perfect bilinear filtering.

Abstract

Recent advances in texture compression provide major improvements in compression ratios, but cannot use the GPU's texture units for decompression and filtering. This has led to the development of stochastic texture filtering (STF) techniques to avoid the high cost of multiple texel evaluations with such formats. Unfortunately, those methods can give undesirable visual appearance changes under magnification and may contain visible noise and filtering with STF by sharing decoded texel values between nearby pixels [WPAM25]. Using GPU wave communication intrinsics, this sharing can be performed inside actively executing shaders without memory traffic overhead. We take this idea further and present novel algorithms that use wave communication between lanes to avoid repeated texel decompression prior to filtering. By distributing unique work across lanes, we can achieve zero-error filtering using ≤ 1 texel evaluations per pixel given a sufficiently large magnification factor. For the remaining cases, we propose novel filtering fallback methods that also achieve higher quality than prior approaches.

CCS Concepts

• Computing methodologies → Texturing; Image processing; Image compression;

Keywords: stochastic texture filtering, wave intrinsics.

1. Introduction

Stochastic texture filtering (STF) has recently reemerged as a useful technique [PWSF24] for filtering texture representations such as neural encodings [VSW*23, FH24, KLM24, DBB*25] that are expensive to evaluate. Not only does it allow performing fewer texel evaluations than traditional deterministic texture filtering requires, but STF also allows efficient implementation of *filtering after shading*, where the filter is applied to the final shaded values rather than the texture input. When shading functions have nonlinearities and textures are minified, filtering the final result can give higher-quality images. However, when textures are magnified, fil-

© 2025 The Author(s).

Proceedings published by Eurographics - The European Association for Computer Graphics. This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

tering after shading can introduce aliasing [PWSF24]; in that case filtering of the texel values is preferable.

With native GPU texture formats, accessing all the texels needed for the filter has limited cost since the cache hierarchy is effective at greatly reducing the memory bandwidth used for redundant texture lookups at nearby pixels. On the other hand, directly filtering custom texture formats in a shader is computationally expensive since each pixel must produce every texel it needs. Caches give less benefit: while they can still reduce the bandwidth, e.g., for reading network weights used by multiple texel evaluations, they do not save the redundant computation.

With the One-tap STF algorithm, each pixel always samples a single texel and filtering after shading is always performed [PWSF24]. Recent work by Wronski et al. introduced a Wave Communication STF algorithm that uses GPU wave-intrinsic instructions to communicate texel values among groups of pixels running in the same GPU wave [WPAM25]. This allows filtering before shading under magnification, giving results significantly closer to full texture filtering compared to One-tap STF without increasing the number of texels evaluated. However, with Wave Communication STF, each pixel independently determines which texel to evaluate based on its own texture filter. Although Wronski et al. introduced methods based on optimized blue noise samples and stochastic sharing masks to increase the probability of pixels finding novel texel samples at their neighbors, their method does not guarantee that nearby pixels will not redundantly evaluate the same texel. Under the GPU SIMT execution model, such redundant computation evaluation does not necessarily harm performance; runtime is generally determined by the maximum computation done in any pixel in a wave, so having multiple lanes do the same computation has no additional runtime cost. However, redundantly evaluating the same texel does not give any new information necessary for more accurate filtering; image quality could be further improved if one could guarantee that different lanes produce unique texels.

In this work, we develop algorithms for collaborative texture filtering (CTF) for when textures are magnified, where collections of pixels communicate to determine their collective texel requirements and then ensure that each pixel evaluates a unique texel. Each pixel then gathers the texels it needs for its filter from its lane neighbors through efficient wave-intrinsic instructions. For moderate $(1.59 \times \text{ or } 2.35 \times, \text{ depending on which of our methods is used})$ or greater magnification with 32-lane waves, our approach guarantees that each pixel has all the texels required for perfect bilinear filtering. For lower magnification factors or with wider filters that require more texels, all texels may not be available. In this case, we have each pixel prioritize texels based on their contribution to its filtered result, similarly to Wave Communication STF. However, for such cases we propose a novel way of combining those texel values that improves image quality compared to One-tap and Wave Communication STF, with negligible additional computational cost.

For high magnification factors, only a few texels may be required by all pixels in a wave. In this case, not only do we ensure that all pixels have all texels needed for filtering, but there is an opportunity for improved performance if the computation for texel evaluation can be split across multiple lanes and performed in parallel.

Our primary contributions are:

- The introduction of three algorithms for collaborative texture filtering across lanes in a wave, used to achieve perfect filtering under magnification: *List Merge*, *Box Sampling*, and *Mask Sampling*, spanning different design points that trade off computation and the effectiveness of finding a minimal set of texels required.
- New *fallback* methods for when our techniques above are unable to provide all texels needed for perfect filtering, which give higher quality output than One-tap STF [PWSF24] or Wave Communication STF [WPAM25].
- Evaluation of our algorithms with bilinear, bicubic B-spline, and Catmull–Rom filters, comparing performance and image quality to One-tap STF and Wave Communication STF.
- Single-tap filtering for filters with both negative and positive weights, such as the Catmull–Rom filter, without requiring more expensive techniques such as positivization [PWSF24].

2. Previous Work

Stochastic texture filtering has seen previous use dating to the 1990s; see Pharr et al. [PWSF24] for an extensive history. When textures are magnified, the filtering after shading normally performed with STF can introduce aliasing; recent work by Wronski et al. [WPAM25] addresses this issue by sharing texel samples across groups of nearby pixels. When all texels required to filter a texture at a pixel are available, it is possible to perform traditional filtering for magnified textures, eliminating this aliasing. In general, even if only some of the additional texels are available, such texel sharing reduces error.

The sharing approach by Wronski et al. is based on *wave intrinsics* that allow efficient sharing of values between shader instances at nearby pixels. Introduced in DirectX HLSL Shader Model 6.0, wave intrinsics expose the notion of a *wave* of individual *lanes* that are executing shader instances as a group, mirroring the underlying GPU execution model [Mic21]. See the paper by Wronski et al. for a more detailed introduction to the concept [WPAM25, Section 2.3]. Because shader instances execute together, wave intrinsics can efficiently use vector registers for communication instead of using local or off-chip memory. (While lanes may be mapped to pixels, vertices, or other elements being processed by shaders, we will sometimes use "pixels" interchangeably with "lanes.") Our work is inspired by that of Wronski et al. and our algorithms make similar use of wave intrinsics.

Communication of data between executing shader instances has been used for a variety of other high-performance graphics algorithms, including in-place screen-space filtering [Pen11, MML12]. Communication between shading passes is also frequently performed using off-chip memory; examples include ReSTIR, which communicates samples between pixels, both spatially and temporally [BWP*20], and post-processing antialiasing techniques like TAA [Kar14, YLS20] and DLSS [NVI25] that temporally accumulate shaded pixel values.

The algorithms we introduce are based on pixels individually identifying texels required for filtering and calculating a shared set of texels that need to be produced for the wave. When this set of texels does not exceed the number of pixels in the group, we distribute the texel decoding cost between different pixels to avoid redundant texel evaluations. Tiled and clustered lighting algorithms [OBA12] similarly communicate between local groups of pixels to determine their spatial and directional bounds. Then, they use local groupshared memory to collaboratively cull the light list to determine a subset of lights that may contribute to the given pixel cluster's shading. Also related are techniques like resolution-matched shadow maps where shadow map lookup requests are written to memory and parallel compaction algorithms determine which shadow quadtree pages to render [LSO07]. (Some virtual texturing algorithms follow similar approaches.) Kenzel et al. propose to avoid redundant work in compute-based rasterization pipelines by merging pixel coverage bit masks [KKSS18]. By using wave intrinsics rather than groupshared or off-chip memory to organize texel requests, our approach achieves high performance and does not require shaders to finish or synchronize execution before launching a separate set of shaders to process requests.

A characteristic of our approach is that each pixel does not necessarily compute a texel value that it needs itself. This mirrors a common GPU programming technique, where threads may not have a fixed association with data elements. Classic examples are highperformance GPU scan and sorting algorithms [SHG011,SHG09].

3. Collaborative Texture Filtering

In this section, we introduce a set of techniques that we call *collaborative texture filtering* (CTF). For simplicity, the majority of our description is related to bilinear filtering, but as we will show later on, it also extends to arbitrary filters, e.g., bicubic B-spline and Catmull–Rom. In addition, we assume that each wave has 32 lanes and is configured as 8×4 pixels. However, our methods generalize to any wave size and can work even better at lower magnification factors with larger waves. The key feature of all our algorithms is that they aim to provide all the texels required for *perfect*, i.e., with zero error, filtering with at most one texel evaluation per lane for all the lanes in a wave.

All our new methods share the same general algorithm flow, which is summarized as:

- 1. The lanes in the wave collaboratively collect information about which texels are needed for perfect filtering.
- 2. Texel evaluation: each lane *produces* up to one texel based on the information from the previous step.
- 3. Each lane *gathers* the texels it needs from across the wave and filters them.

In step 2, we use the term *producing* a texel to include, e.g., a regular texel access using the GPU texture unit, decompressing a texel stored in a custom format, procedural generation, or traversing a data structure to obtain the texel [KLM24]. Note also that we sometimes use "produce a texel" interchangeably with "texel evaluation."

In some cases, e.g., when the number of unique texels from step one is too large, a fallback method is used. In general, one may fall back to any filtering method, such as One-tap STF [PWSF24] or Wave Communication STF [WPAM25], though we present novel fallback methods in Section 3.4 that result in higher image quality.

Next, we describe three implementations of the algorithm flow

above: *List Merge, Box Sampling,* and *Mask Sampling.* Each has different qualities, such as performance and success rate of perfect filtering. A high-level description of them can be found in Figure 2.

3.1. List Merge

List Merge creates a list of the unique texels required by the wave. This list is constructed in shared memory using a hierarchical list merge algorithm. The texture coordinates in each lane determine which $N \times N$ texels are needed for filtering its texture. For now, we will focus on the bilinear filter with 2×2 texture filtering footprint. In this case, the integer coordinates of these four texels are put in a list of length four in each lane.

As illustrated to the left in Figure 2, a hierarchical list merge is then performed, using shared memory to store the lists during the merge. This is similar to the hierarchical parallel sum described by Hoobler [Hoo11], though instead of summing two values, we merge two lists such that the list never contains any duplicate texel coordinate pairs.

After the final list has been formed, we check if the number of elements in the list is larger than the number of available lanes. If so, perfect filtering with one texel evaluation per lane is not possible and we revert to a fallback method (Section 3.4). Otherwise, each lane first produces the texel whose index in the merged list is the same as its lane index. After texels have been produced, each lane loops over lanes in the wave to gather the texels that fall within its filter footprint and then weights them using the original texture filter weights. This method has substantially worse runtime cost than our other methods, and therefore we omit further details. We include it in the discussion because it provides an upper bound on the success rate of when perfect filtering is possible and demonstrates the goal we aim to more efficiently approximate.

3.2. Box Sampling

On the opposite spectrum of complexity to List Merge is a much simpler alternative that we call *Box Sampling*, illustrated in the middle of Figure 2. In this method, each lane calculates an axis-aligned bounding box (AABB) of the texels it needs for texture filtering. The size of this local AABB depends on the texture filter—for example 2×2 texels for bilinear and 4×4 for bicubic filtering. We use those local bounds to compute a global AABB for the wave using the WaveActiveMin() and WaveActiveMax() intrinsics.

If the area *n* of the global bounding box exceeds the number of active lanes in the wave, we use one of the fallback methods (Section 3.4). Otherwise, in cases where at least the first *n* texels are active,[†] we use a simple bijective function to map from the active lane index to coordinates within the bounding box by using modulo and division by the bounding box width. Each active lane with an index < *n* uses this mapping to produce the corresponding texel from the texture. Finally, each pixel uses WaveReadLaneAt() instructions with the source lane index computed through inverse

[†] For cases when this is not true, we refer to the description in our supplemental (Section S2), where an additional mapping function is needed.

4 of 16

T. Akenine-Möller, P. Ebelin, M. Pharr, B. Wronski / Collaborative Texture Filtering



Figure 2: High-level overviews of our three algorithms for the bilinear filtering case, shown with only four lanes (in gray) and where algorithm flow is downward. See our pseudocode in Section S1 for details of early outs to fallback methods. Shared memory is shown in yellow, while cross-wave operations are green. List Merge performs a hierarchical list merge on the 2×2 texel integer coordinates per lane. This gives a list of unique texels needed by the wave to be able to perform bilinear filtering in each lane. Box Sampling is our fastest technique and uses an efficient bijective function to find which texel is produced by which lane. Mask Sampling creates a local bitmask where each bit indicates that the texel is needed by the wave. Mask Sampling is exemplified using an 8×8 bitmask.

mapping from local coordinates inside the AABB to gather the texels needed for its filter, and weights them to generate the pixel's perfectly-filtered value.

While Box Sampling is substantially simpler and faster than List Merge, an AABB may include many unnecessary texels under perspective projection or rotations around 45 degrees, resulting in more frequent need of the fallback method and thus increased error. To address this limitation, we propose a slightly more complex hybrid method that increases the number of perfectly-filtered waves without incurring the excessive cost of List Merge.

3.3. Mask Sampling

Similar to Box Sampling, Mask Sampling starts by creating an AABB over the texel integer coordinates. If it is larger than 16×16 texels, then we use one of the fallback methods (Section 3.4). Otherwise, we use a 16×16 bitmask to encode which texels are needed. This bitmask is stored using an uint 64_t4 variable, i.e., $4 \cdot 64 = 256 = 16 \cdot 16$ bits. Each lane sets the four bits in the mask corresponding to the 2×2 texels it needs for bilinear filtering. We then use WaveActiveBitOr() on these masks to find a *wave mask*, called *B*, that contains a 1 for each texel the wave needs for perfect bilinear filtering. If the number of bits set in the wave mask, *n*, is larger than 32, we also call one of the fallback methods.

Otherwise, we know that there are n unique texels in this wave

and $n \leq 32$, and for now we assume that the first *n* lanes are active in the wave. We refer to the description in our supplemental (Section S2) for cases when this is not true. If i < n, where *i* is the current lane's index, then lane *i* will produce a texel. One of the texels marked by a 1 in the wave mask needs to be selected. Hence, a mapping *h* is needed from lane index *i* to a requested one-dimensional texel coordinate[‡] *t* in the wave mask, *B*. We choose *h* to map from *i* to the *i*th bit that is set in *B*, and call the function t = h(i, B). The number *t* is in [0, 255], and so can be mapped to local two-dimensional coordinates in $[0, 15]^2$ and added to the upper left integer coordinates of the AABB. These coordinates are then used to produce the desired texel.

The final step is to gather texels from the lanes in the wave and filter them. For each two-dimensional texel coordinate needed in a lane, we compute the corresponding integer coordinate, t, and then retrieve the desired texel from lane $i = h^{-1}(t, B)$, where the inverse of finding the *i*th set bit is counting the set bits below bit t. This is illustrated in Figure 3. All the texels needed for the current lane are guaranteed to have been produced by the wave, and these are weighted and the perfectly-filtered value is returned.

A 16×16 bitmask is sufficient for both bilinear and bicubic

[‡] One can convert from two-dimensional coordinates, (t_x, t_y) , to onedimensional as $t = 16 \cdot t_y + t_x$.

Figure 3: Illustration of our bijective function t = h(i, B) used in Mask Sampling. Assume the lane index i = 2, then h(2, B) = 4 since the bit number of the 2nd set bit is 4, as illustrated by the left arrow. Note that we start counting from 0, so the 0th set bit is at position 1 in the example above. To the right, we map from t = 4 to lane index i by counting the number of set bits to the right of t = 4. The result is 2, so $h^{-1}(4, B) = 2$.

filters, though a smaller bitmask may be used to reduce register pressure and computation, potentially at a cost in effectiveness. We have found empirically that an 11×11 bitmask stored in two uint64_t variables is sufficient for the bilinear filter—it gives identical results to the 16×16 bitmask with 25-33% lower runtime in our experiments. We use this variant for measurements in Section 4.

3.4. New Fallback Methods

This section presents novel fallback methods that may be used to handle the case when perfect filtering cannot be achieved (discussed in Section 4). In these methods, we attempt to make each lane in the wave do useful work toward increasing the filtering quality.

In our fallback methods, each lane starts by computing the coordinates of a texel using STF, i.e., randomly selecting a texel with probability based on its corresponding filter weight. In our simplest fallback variant, all lanes produce their corresponding STF texel, gather unique texels within the wave belonging to their filter footprints and then weight those texels to form the final filtered result. Assume that a texel value, \mathbf{p}_k , is produced for lane *k*. Each lane *i* identifies *N* unique texels \mathbf{p}_i , with nonzero filter weights w_i , that contribute to its pixel's filtered value. Then, we estimate each lane's filtered value as

$$\mathbf{c}_{k} = \sum_{i=0}^{N-1} w_{i} \mathbf{p}_{i} + \left(1 - \sum_{i=0}^{N-1} w_{i}\right) \frac{\sum_{i=0}^{N-1} \mathbf{p}_{i}}{N},$$
(1)

where the first term represents the known filtered texel values weighted by their corresponding filter weights. The second term is an estimate of the missing texel values as an unweighted average of the known ones. This estimate would be unbiased if it were not for the lane PDF mismatch [WPAM25]. For the case when we only have one unique texel for lane *i*, we return a single value \mathbf{p}_i , the same as the classic One-tap STF evaluation [PWSF24]. When all texel values necessary for filtering a single lane are known and present in the wave, the second term of the equation is zero and we get perfect filtering with zero error.

An extension to the method above further improves quality. We note that just like in the method described by Wronski et al. [WPAM25], multiple lanes may plan to produce the same texel, resulting in redundant work. To remove some of them, we represent the set of texels to be produced using a bitmask, similar to Mask Sampling. Before the actual STF texel evaluation, each lane sets

© 2025 The Author(s). Proceedings published by Eurographics - The European Association for Computer Graphics. the bit corresponding to the coordinates of its STF texel. We then use WaveActiveBitOr() to find a bitmask containing all texels that the wave is set to produce. Counting the number *n* of set bits in that mask gives us the number of unique texels that need to be produced for this initial texel evaluation. The difference here compared to Mask Sampling is that we only set one bit per lane, rather than $N \times N$. We can then leverage the remaining 32 - n lanes to improve quality. The first *n* lanes each produce their corresponding STF texel. The remaining 32 - n lanes also produce texels, but we let each of these stochastically select a texel from its filter footprint that has not yet been produced by any of the first *n* lanes. These evaluations are spread out evenly over the wave's 32 lanes. When n < 32, this is done with a simple mapping, i.e.,

$$l = \left\lfloor \frac{31(c-n)}{31-n} \right\rceil,\tag{2}$$

where *l* is the resulting lane number, *c* is the current lane number, which in the case of the 32 - n remaining lanes must be in [n, 31], and $\lfloor \cdot \rceil$ rounds to the nearest integer. Our mapping ensures that lanes 0 and 31 always are included when $32 - n \ge 2$. When only one lane is left, the equation above maps it to l = 0. Alternative mappings are also possible. At this point all lanes in the wave have produced a texel, and then weighting is again done as described by Equation 1.

These fallback methods are biased, yet as shown in Section 4, they provide superior quality compared to previous, unbiased alternatives. In the results section, we mark the use of the simpler fallback method described above with C, and the use of the extended version with C+.

4. Results

We start by focusing on bilinear filtering for magnification, with bicubic filtering evaluated in Section 4.2. Additional performance considerations are discussed in Section 4.3. For visual results, we refer to Figure 1, Figure 8, and our supplemental video.

All methods are implemented in the Falcor rendering framework [KCK*22]. To measure quality only on waves with magnification, we use trilinear texture filtering for all waves that have at least one pixel with minification. Performance is measured on an NVIDIA RTX 5090 GPU. Quality results are aggregated over five different textures; see Section \$3.1 for images and discussion of how image metrics were aggregated. Furthermore, as denoising can be expected to be applied as a post-process to most rendering algorithms, image sequences are denoised using the DLSS [NVI25] spatiotemporal denoiser unless stated otherwise. We use its DLAA mode, which does spatiotemporal denoising and antialiasing, but no superresolution. Diagrams generated with non-denoised images corresponding to ones presented in this section can be found in our supplemental material. Our runtime performance measurements focus on the incremental costs of our algorithms and do not include DLSS, which we assume is already in use.

We primarily use the ColorVideoVDP video quality metric [MHA*24] (CVVDP for short) to evaluate error; it attempts to model spatiotemporal aspects of human vision to better correspond to human judgment than simpler metrics such as PSNR. ColorVideoVDP's output is in just-objectionable differences (JODs).



Figure 4: Plot of magnification factors where our algorithms are unable to produce all texels necessary for perfect bilinear filtering and need to rely on fallback methods (Section 3.4). This magnification factor is a function of the rotation of the texel grid compared to the screen pixel grid. The scene used was a rotated, textured quad, with the camera set to view the quad's center head on. Cases where a fallback was necessary are indicated by colored areas. List Merge and Mask Sampling are the most effective, achieving perfect filtering around a magnification factor of 1.59 and above for all quad orientations. (Note that both cover identical areas.) The success rate of Box Sampling varies more with orientation, with a magnification factor of at least 2.35 needed for perfect filtering at the challenging case of a 45 degree rotation.

The maximum JOD value is 10, which corresponds to the image or video being perceptually indistinguishable from the ground truth. Table 1 also includes results with other image metrics as well as error measurements of non-denoised sequences.

4.1. Bilinear Filtering

We start with results for bilinear texture filtering during magnification, which are used to support the recommendations we make in Section 5. While our algorithms are often able to produce perfect filtering, the degree of magnification is sometimes too low to be able to produce all the required texels with one texture evaluation per lane. In such cases, our fallback methods (Section 3.4) are needed. For the bilinear filter, those cases are illustrated by the colored regions in Figure 4. See Figure S8 in our supplemental material for the bicubic counterpart.

Figure 5 plots the quality and performance of our new methods and the state-of-the-art alternatives—One-tap STF [PWSF24] and Wave Communication STF [WPAM25] both as standalone methods but also as fallback options for Box and Mask Sampling. Figures 6 and 7, discussed later in the section, give more detailed presentations of the algorithms' quality and performance. List Merge was not considered because Mask Sampling reaches the same success rate of achieving perfect filtering (Figure 4) while incurring a smaller runtime cost. The diagrams also contain combinations of our proposed fallback methods with Box and Mask Sampling as main methods. In all of our results, the fallback method used is indicated in parenthesis, e.g., "Mask Sampling (C+)" is Mask Sampling with our C+ technique as fallback. This combination is what we call "Ours" in Figure 1. The scenes used for our evaluation are described in our supplemental material. The one used for Figure 5 and Table 1 contains a range of magnification factors to make it similar to a realistic use case.

Considering Figure 5 and Table 1, we see that our methods provide higher-quality results than One-tap STF and Wave Communication STF, though One-tap STF is the fastest method. Following the Pareto frontier in Figure 5, we see that combining Box Sampling with One-tap STF as a fallback gives a large increase in quality compared to One-tap STF by itself without significantly increasing runtime cost. Our fallback methods may then increase quality further, though with a small performance cost. Mask Sampling paired with One-tap STF and Wave Communication STF fallbacks are not quality/performance-efficient options. Those combinations are therefore excluded in the remainder of the paper.

In Table 1, we complement the quality results presented in Figure 5 with the PSNR and FLIP [ANA*20] image quality metrics, as well as measurements for non-denoised sequences. Our methods give significantly higher quality than the state-of-the-art algorithms. Denoising reduces the quality gap between our and prior techniques, which are designed to be used with denoising.

In order to analyze quality and performance as magnification varies, we next consider a scene where the camera is set up to view a rotated quad's center head on. Given the results in Figure 4, we rotated the quad 45 degrees, as that is the most challenging case. Figure 6 shows how quality varies as magnification increases. Mask Sampling achieves perfect filtering at lower levels of magnification than Box Sampling while previous algorithms do not achieve this for any level of magnification. For additional quality results, including convergence rates and maximum errors, see Section S3 in our supplemental material.

Figure 7 shows how the runtime cost of the methods changes during the same animation using a single lookup with a standard RGBA texture. We can see that our extended fallback method (C+) contributes a large portion of the total runtime cost for the variants that use it. This is likely due to its implementation requiring more registers than simpler alternatives. As magnification increases, the less our algorithms need to resort to fallback methods, which reduces their runtime cost.

4.2. Bicubic Filtering

Our comparisons so far have focused on the bilinear filter due to its ubiquity in real-time rendering. However, our methods also work with arbitrary discrete filters. Figure 8 shows visual results and error measurements using a bicubic B-spline filter as well as the Catmull–Rom filter. The results parallel those in Figure 1, with our algorithm giving significantly higher quality than previous approaches.

The Catmull–Rom filter includes negative weights. To sample such filters, One-tap STF uses a Monte Carlo technique called positivization [PWSF24]. Positivization requires sampling the positive *and* negative lobes of the filter separately, thus requiring that two texels per lane are produced. Our method not only provides much higher quality (perfect filtering with zero error for sufficiently large



Figure 5: Pareto frontier (dashed line) indicating the most quality/performance-efficient algorithm alternatives. The corresponding PSNR range for this plot was approximately 35–67 dB.



Figure 6: Quality at different magnification factors. As indicated by Figure 4, perfect bilinear filtering is achieved for magnification factors above 2.35 for all the methods we propose. Mask Sampling achieves perfect filtering at lower magnification factors than Box Sampling. STF quality declines as magnification increases due to increased aliasing and error from filtering after shading [PWSF24].



Figure 7: Performance for an NVIDIA RTX 5090 under different levels of magnification. The numbers in parentheses show the algorithms' average runtime costs. For reference, running at 60 FPS at 2560×1440 corresponds to 4741μ s/Mpixel, indicating that all methods consume a tiny fraction of total frame time.

magnification factors), but also reduces the number of texels produced to ≤ 1 per lane. We make no distinction between positive and negative weights and the types of filters other than computing absolute values of filter weights used for sampling probabilities in our fallback method.

4.3. Performance with Expensive Texture Decompression

In this section, we first present performance results using an existing neural texture decompressor and then consider DCT decompression, where our techniques can be applied to further reduce the cost of texel evaluation by distributing work for a single texel across multiple lanes.

4.3.1. Neural Texture Decompression

To evaluate how our algorithms perform with neural texture compression [VSW*23] (NTC), we added Box Sampling with Onetap STF as fallback into the ntc-renderer in the NTC SDK.§ We rendered the scene *FlightHelmet* and zoomed in on a part of the helmet such that the magnification factor was > 2.35 for every pixel. All NTC evaluations used inference on sample, i.e., texels were decompressed when needed by the renderer. The renderer ran at 7.7 μ s/Mpixel without any texel evaluation, and with one NTC evaluation per pixel using One-tap STF, it used 89 µs/Mpixel. Our Box Sampling with One-tap STF as fallback ran at 93 µs/Mpixel. (Recall also that rendering at, for example, 2560×1440 pixels at 60 FPS is equivalent to 4741 μ s/Mpixel.) Due to the level of magnification, our method produced an image with zero error, while One-tap STF generated noisy images. We also implemented full bilinear filtering where 2×2 NTC texels were decompressed for each pixel. Performance was 516 μ s/Mpixel with the same image quality as our method but more than $4 \times$ higher runtime.

4.3.2. DCT Decompression

When there are more active lanes in a wave than needed texels, there is an opportunity to distribute the work required for producing a texel value across multiple lanes rather than having a single lane produce each texel value; this can give improved performance. As a proof-of-concept, we implemented a simple block-based discrete cosine transform (DCT) compressed texture representation where decompressing a block requires 8×8 matrix multiplications.

For a baseline, we decompress each texel in a single lane, first decoding the red channel, then green and blue. We then added an optimized path for when ten or fewer texels are needed in a 32-lane wave—in that case, lanes decompress just a single color channel and color channels are then assembled into full RGB texel values via wave intrinsics. With a test scene with a plane viewed at an angle where this optimization was applicable at roughly half of the pixels, we saw a $1.64 \times$ speedup. When all pixels had sufficiently large magnification factors, we measured a $2.5 \times$ speedup. Similar performance improvements may be available for neural compression techniques [VSW*23,FH24,KLM24,DBB*25], where neural network evaluation could be distributed across multiple lanes when

[§] https://github.com/NVIDIA-RTX/RTXNTC

Proceedings published by Eurographics - The European Association for Computer Graphics.

Table 1: Error metrics averaged across sequences. \downarrow indicates lower is better, \uparrow indicates higher is better. For each entry, the first number (black) shows results without denoising while the second number (non-black) shows results with denoising.

	One-tap STF [PWSF24]	Wave Comm. STF [WPAM25]	Box Sampling (One-tap STF)	Box Sampling (Wave Comm.)	Box Sampling (C)	Mask Sampling (C)	Box Sampling (C+)	Mask Sampling (C+)
PSNR (†)	28.57 34.94	34.93 42.60	42.92 52.15	48.04 56.63	51.99 60.06	58.02 65.50	58.40 65.56	59.48 66.58
ColorVideoVDP (†)	8.792 8.848	9.618 9.806	9.887 9.931	9.941 9.985	9.973 9.996	9.987 9.998	9.989 9.999	9.990 9.999
$\operatorname{FLIP}\left(\downarrow\right)$	0.0501 0.0328	0.0375 0.0224	0.0034 0.0040	0.0028 0.0036	0.0018 0.0027	0.0003 0.0016	0.0006 0.0018	0.0002 0.0015



Figure 8: Visual comparison and error measurments with the bicubic B-spline and Catmull–Rom filters. All three error metrics indicate that our method achieves very high quality, both with and without denoising. For Catmull–Rom filters with negative weights, our method requires 0.88 texel evaluations per pixel, while One-tap STF requires two texel evaluations to sample the positive and negative lobes of the filter separately. Wave Communication STF does not support filters with negative weights.

idle ones are available. However, a full performance evaluation is out of the scope of this paper and left for future work.

Finally, we note that our method can similarly reduce the computational cost even with traditional texture representations. Pharr et al. [PWSF24] discussed how STF can be used to accelerate common expensive material evaluation techniques requiring multiple material samples, such as triplanar mapping. Our method can be used to similarly distribute sampling of different axis-aligned planes between lanes in a wave, or, in general, even different material samples.

5. Discussion and Limitations

Our results in Section 4.1 suggest that Mask Sampling gives the highest quality independent of whether denoising is used, though Box Sampling does almost as well with a lower runtime cost. This makes Box Sampling our general recommendation. However, because Mask Sampling gives a more accurate estimate of which texels are required for perfect filtering, it is an attractive alternative when combined with larger filters (Section 4.2) where the fallback method may be necessary less often or with decompression methods that split decompression work across lanes (Section 4.3).

We have shown that our algorithms guarantee perfect filtering with small discrete texture filters above modest magnification factors and also do so with larger texture filters at higher magnification factors (Section S3.6). This is a natural outcome, since more texels are needed for larger texture filters. For perfect filtering at lower magnification rates, all of our methods can easily be extended to generate more than one texel per lane. We have verified experimentally that for a screen-aligned quad with magnification factor 1.0, the bilinear filter requires no more than 54 unique texels (occuring

at, e.g., 30° rotation of the quad) for 8×4 waves. Hence, ≤ 2 texel evaluations are sufficient to *always* succeed at returning a perfectly bilinearly-filtered value when all pixels in a wave undergo magnification. Similar analysis for larger filters is left for future work.

While we focused our work on the application of magnification filtering, we found that our fallback methods apply to modest minification filters and trilinear filtering as well, effectively performing filtering before shading. We have decided not to include those results following the recommendations of Pharr et al. [PWSF24], as well as common real-time rendering practice of negative MIP biasing and relying on post-shading spatiotemporal reconstruction filters [Kar14]. Filtering before shading changes the appearance of rendered objects under minification, reduces the perceived texture sharpness, and can lead to aliasing.

One limitation of our algorithms is a consequence of their reliance on obtaining complete enumerable sets of texels to filter. Pharr et al. [PWSF24] proposed a second family of stochastic texture filtering for continuous and potentially infinite filters, based on filter importance sampling. Determining the complete set of texels required for perfect filtering is inherently impossible for any filter with an infinite spatial support, which makes our methods inapplicable. To use our algorithms with such filters, truncation and windowing would be necessary, which would lead to approximation errors. Furthermore, computing multiple filter weights is significantly more costly than sampling the filter PDF in the case of commonly used filters, such as the Gaussian kernel.

6. Conclusions and Future Work

Our family of techniques effectively addresses the shortcomings of stochastic texture filtering during magnification for commonlyused texture filters. Above small magnification factors, they achieve zero error while having a small computational cost, generating no more than one texel per lane.

While we briefly investigated techniques to increase parallelism for decompression when few unique texels are required (Section 4.3.2), further work in this direction may give even greater performance improvements in such cases. Extensions to other decompressors may give similar performance benefits.

In our approach, we rely on inter-wave communication to minimize performance overhead. However, this limits the number of GPU threads collaboratively producing texels to the wave size and limits the applicability of our algorithm to small texture filters. In future work, it might be worth extending our approach to a hybrid between wave communication and using shared memory with larger compute shader launch groups.

Acknowledgments

Many thanks to Tomáš Davidovič for help with details of Falcor internals and to Johannes Deligiannis for help with integrating our techniques with the NTC SDK. Magnus Andersson, Rasmus Barringer, Anders Lindqvist, James Player, and Robert Toth all helped with code optimizations. Thanks also to Aaron Lefohn and NVIDIA Research for supporting this work.

Thanks to *PolyHaven* for the Aerial Rocks, Dirt, and Painted Concrete texture sets and to *ambientCG* for the Bricks and Rails texture sets.

References

- [ANA*20] ANDERSSON P., NILSSON J., AKENINE-MÖLLER T., OS-KARSSON M., ÅSTRÖM K., FAIRCHILD M. D.: FILIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 3*, 2 (2020), 15:1–23. 6, 12
- [BWP*20] BITTERLI B., WYMAN C., PHARR M., SHIRLEY P., LEFOHN A., JAROSZ W.: Spatiotemporal Reservoir Resampling for Real-Time Ray Tracing with Dynamic Direct Lighting. ACM Transactions on Graphics 39, 4 (July 2020). doi:10/gg8xc7.2
- [DBB*25] DUPUY J., BENYOUB A., BELCOUR L., MERECKI M., CHAMBON T.: Intel Co-Presents Cooperative Vectors with Microsoft. Game Developers Conference, 2025. [Online; accessed 2025-03-25]. 1, 7
- [FH24] FUJIEDA S., HARADA T.: Neural Texture Block Compression. In Workshop on Material Appearance Modeling (2024), Hardeberg J. Y., Rushmeier H., (Eds.), The Eurographics Association. 1, 7
- [Hoo11] HOOBLER N.: High Performance Post-Processing. In Game Developers Conference (2011). 3
- [Kar14] KARIS B.: High-Quality Temporal Supersampling. Advances in Real-Time Rendering in Games, SIGGRAPH Courses 1, 10.1145 (2014), 2614028–2615455. 2, 8
- [KCK*22] KALLWEIT S., CLARBERG P., KOLB C., DAVIDOVIČ T., YAO K.-H., FOLEY T., HE Y., WU L., CHEN L., AKENINE-MÖLLER T., WYMAN C., CRASSIN C., BENTY N.: The falcor rendering framework. BSD-Licensed Github Repository, August 2022. 5
- [KKSS18] KENZEL M., KERBL B., SCHMALSTIEG D., STEINBERGER M.: A High-Performance Software Graphics Pipeline Architecture for the GPU. ACM Transactions on Graphics 37, 4 (2018), 1–15. 3
- [KLM24] KIM D., LEE M., MUSETH K.: NeuralVDB: High-Resolution Sparse Volume Representation Using Hierarchical Neural Networks. ACM Transactions on Graphics 43, 2 (2024), 20:1–21. 1, 3, 7

© 2025 The Author(s).

- [LSO07] LEFOHN A. E., SENGUPTA S., OWENS J. D.: Resolution-Matched Shadow Maps. ACM Transactions on Graphics 26, 4 (Oct. 2007), 20–37. 3
- [MHA*24] MANTIUK R. K., HANJI P., ASHRAF M., ASANO Y., CHAPIRO A.: ColorVideoVDP: A Visual Difference Predictor for Image, Video and Display Distortions. ACM Transactions on Graphics 43, 4 (2024), 129:1–20. 5, 12
- [Mic21] MICROSOFT: HLSL Shader Model 6.0. https://learn. microsoft.com/en-us/windows/win32/direct3dhlsl/ hlsl-shader-model-6-0-features-for-direct3d-12, 2021. [Online; accessed 2024-09-11]. 2
- [MML12] MCGUIRE M., MARA M., LUEBKE D.: Scalable Ambient Obscurance. In *High Performance Graphics* (2012), pp. 97–103. 2
- [NVI25] NVIDIA: DLSS 4: Transforming Real-Time Graphics with AI. https://research.nvidia.com/labs/adlr/DLSS4/, 2025. Technical Report. 1, 2, 5, 15
- [OBA12] OLSSON O., BILLETER M., ASSARSSON U.: Clustered Deferred and Forward Shading. In *High Performance Graphics* (2012), pp. 87–96. 3
- [Pen11] PENNER E.: Shader Amortization Using Pixel Quad Message Passing. In GPU Pro 2. CRC Press, 2011, pp. 349–366. 2
- [PWSF24] PHARR M., WRONSKI B., SALVI M., FAJARDO M.: Filtering After Shading with Stochastic Texture Filtering. Proceedings of the ACM on Computer Graphics and Interactive Techniques 7, 1 (2024), 14:1–20. 1, 2, 3, 5, 6, 7, 8, 13, 14, 15
- [SHG09] SATISH N., HARRIS M., GARLAND M.: Designing Efficient Sorting Algorithms for Manycore GPUs. In *IEEE International Sympo*sium on Parallel & Distributed Processing (2009), pp. 1–10. 3
- [SHG011] SENGUPTA S., HARRIS M. J., GARLAND M., OWENS J. D.: Efficient Parallel Scan Algorithms for Many-Core GPUs. In *Scientific Computing with Multicore and Accelerators*, Jakub Kurzak D. A. B., Dongarra J., (Eds.). 2011, pp. 413–442. 3
- [VSW*23] VAIDYANATHAN K., SALVI M., WRONSKI B., AKENINE-MÖLLER T., EBELIN P., LEFOHN A.: Random-Access Neural Compression of Material Textures. ACM Transactions on Graphics 42, 4 (2023), 88:1–25. 1, 7
- [WPAM25] WRONSKI B., PHARR M., AKENINE-MÖLLER T.: Improved Stochastic Texture Filtering Through Sample Reuse. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 8, 1 (2025). arXiv:2504.05562. 1, 2, 3, 5, 6, 8, 13, 14, 15
- [YLS20] YANG L., LIU S., SALVI M.: A Survey of Temporal Antialiasing Techniques. Computer Graphics Forum 39, 2 (2020), 607–621. 2

Proceedings published by Eurographics - The European Association for Computer Graphics.

10 of 16

T. Akenine-Möller, P. Ebelin, M. Pharr, B. Wronski / Collaborative Texture Filtering

Supplemental Material: Collaborative Texture Filtering

This supplemental material includes:

- Detailed code showing example implementations of the Box Sampling and Mask Sampling algorithms.
- Details on the edge remapping technique used at silhouette edges where not all of the wave's lanes are active.
- More information about the scenes, textures, and metrics used for evaluation.
- · Additional evaluation and results.

S1. Code

For simplicity, we omit clamping of out-of-bounds texture coordinates in the code below and we do not include an implementation of List Merge, since it performs poorly and is not a realistic alternative for use in real applications. (Recall that it was mostly included because it computes the number of texels needed for perfect filtering exactly, which was useful for evaluating the other approaches.)

S1.1. Box Sampling

The following helper functions are used in the implementation of Box Sampling for bilinear filtering, which follows.

```
int2 LaneIdxToCoord(uint laneIdx,
             int2 waveUpperLeftIntCoords,
             uint bbWidth)
 4
   {
       uint laneY = laneIdx / bbWidth;
 6
       uint laneX = laneIdx % bbWidth;
       return waveUpperLeftIntCoords + int2(laneX, laneY);
 8
   }
   uint CoordToLaneIdx(int2 coord,
11
            int2 waveUpperLeftIntCoords,
             uint bbWidth)
13
   {
14
       coord -= waveUpperLeftIntCoords;
15
       return coord.x + coord.y * bbWidth;
16
   }
18
   bool LanesLowerThanCountActive(uint count)
19
20
       uint activeLanesMask = WaveActiveBallot(true).x;
21
       uint desiredActiveMask = (count == 32) ? 0xFFFFFFFF : (1 <<</pre>
        count) - 1;
       return (activeLanesBitMask & desiredActiveMask) ==
        desiredActiveMask;
24
```

Inside the shader that accesses the texture, the following code implements Box Sampling:

```
uint2 txDim;
   texture.GetDimensions(txDim.x, txDim.y);
   float2 floatCoords = uv * txDim - float2(0.5f);
   int2 upperLeftIntCoords = int2(floor(floatCoords));
   int2 lowerRightIntCoords = upperLeftIntCoords + int2(1, 1);
   float2 stCoords = floatCoords - upperLeftIntCoords;
   // Compute bounding box of texel integer coordinates
   int2 waveUpperLeftCoords = WaveActiveMin(upperLeftIntCoords);
   int2 waveLowerRightCoords = WaveActiveMax(lowerRightIntCoords);
   int2 bbSize = waveLowerRightCoords - waveUpperLeftCoords + 1;
11
13
   int activeTexelsNeeded = bbSize.x * bbSize.v;
```

```
14 bool requiredLanesActive = LanesLowerThanCountActive(
         activeTexelsNeeded);
16
   if (activeTexelsNeeded > 32 || !requiredLanesActive)
17
     return fallBackMethod();
18
19
   uint curLaneIdx = WaveGetLaneIndex();
20
   float4 texelValue = float4(0.0f);
22
   if (curLaneIdx <= activeTexelsNeeded) {</pre>
23
     uint2 texCoords = LaneIdxToCoord(curLaneIdx,
24
                                       waveUpperLeftCoords, bbSize.x);
25
     texelValue = texture[texCoords];
26
   }
27
28
   float4 bilinWeights = computeBilinearWeights(stCoords);
29
30
   float4 filteredColor = float4(0.0f);
31
   [unroll]
   for (int i = 0; i < 4; i++) {
     int2 texelCoords = int2(upperLeftIntCoords.x + (i % 2),
                             upperLeftIntCoords.y + (i / 2));
34
35
     uint laneIdx = CoordToLaneIdx(texelCoords, waveUpperLeftCoords,
        bbSize.x);
36
     filteredColor += WaveReadLaneAt(texelValue, laneIdx) *
        bilinWeights[i];
   }
   return filteredColor:
```

The first six lines of code compute various coordinates from the texture dimensions and (u, v)-coordinates. Among these are the upper left coordinates of the 2×2 filter footprint needed for bilinear filtering and (s,t)-coordinates which are local coordinates inside the 2×2 region and are in [0,1]. Next, we use WaveActiveMin() and WaveActiveMax() to compute an axis-aligned bounding box over the texel coordinates that are needed for the entire wave.

Next, we check whether the number of needed texels is less than or equal to the number of active lanes in the wave and if all of the lanes are active. For each texel-producing lane, we then use a simple linearly-ordered mapping LaneIdxToCoord() to compute its texel coordinates. Producing a texel in this example is done here via a regular texel fetch using texture []. Finally, we iterate over the texture filtering footprint, gathering and accumulating the necessary texels using the CoordToLaneIdx () and WaveRead-LaneAt () functions. We note that the same algorithm works for any texture filter: the only changes required are the calculations of upperLeftIntCoords, lowerRightIntCoords, computeBilinearWeights, and the final unrolled loop extent.

S1.2. Mask Sampling

10

11

13

15

Code for Mask Sampling with a 16×16 mask is shown below.

```
uint2 txDim;
   texture.GetDimensions(txDim.x, txDim.y);
   floatCoords = uv * txDim - float2(0.5f);
   int2 upperLeftIntCoords = int2(floor(floatCoords));
   int2 lowerRightIntCoords = upperLeftIntCoords + int2(1, 1);
   float2 stCoords = floatCoords - upperLeftIntCoords;
   // Compute bounding box of texel integer coordinates
   int2 waveUpperLeftCoords = WaveActiveMin(upperLeftIntCoords);
   int2 waveLowerRightCoords = WaveActiveMax(lowerRightIntCoords);
   int2 bbSize = waveLowerRightCoords - waveUpperLeftCoords + 1;
   uint2 deltaCoords = upperLeftIntCoords - waveUpperLeftIntCoords;
14 if (bbSize.x > 16 || bbSize.y > 16)
    return fallBackMethod();
```

// Set the 2x2 bits corresponding to the 2x2 texels needed. uint64_t4 mask = uint64_t4(0); 18 19 set2x2Bits(upperLeftIntCoords - waveUpperLeftIntCoords, mask); 20 21 // Compute the ORed mask across the entire wave. uint64_t4 waveMask; 23 waveMask.x = WaveActiveBitOr(mask.x); 24 waveMask.y = WaveActiveBitOr(mask.y); 25 waveMask.z = WaveActiveBitOr(mask.z); waveMask.w = WaveActiveBitOr(mask.w); 26 27 uint curLaneIdx = WaveGetLaneIndex(); 28 uint activeTexelsNeeded = countbits(waveMask); 29 30 uint activeLanesMask = WaveActiveBallot(true).x; // # active lanes 31 32 if (activeTexelsNeeded > countbits(activeLanesBitMask)) 33 return fallBackMethod(); 34 35 float4 curPixelTexelValue; 36 int2 sampledTexelIntCoords; float4 color = float4(0.0f); 37 38 float4 bilinWeights = computeBilinearWeights(stCoords); 39 40 // Does this lane needs to produce a texel? 41 if (curLaneIdx <= activeTexelsNeeded) { 42 uint2 deltaTexCoords = bijectiveFunctionH(curLaneIdx, waveMask); 43 sampledTexelIntCoords = waveUpperLeftIntCoords + deltaTexCoords; 44 curPixelTexelValue = texture[sampledTexelIntCoords]; 45 46 47 // Compute 1D local texture coordinate: uint t = (deltaCoords.y << 4) + deltaCoords.x;</pre> 48 // Read the 2x2 texels needed for this pixel and weight together. 49 50 uint i0 = inverseBijectiveFunctionH(t, waveMask); 51 color += bilinWeights.x * WaveReadLaneAt(curPixelTexelValue, i0); 52 uint i1 = inverseBijectiveFunctionH(t+1, waveMask); color += bilinWeights.y * WaveReadLaneAt(curPixelTexelValue, i1); 54 uint i2 = inverseBijectiveFunctionH(t+16, waveMask); 55 color += bilinWeights.z * WaveReadLaneAt(curPixelTexelValue, i2); 56 uint i3 = inverseBijectiveFunctionH(t+16+1, waveMask); 57 color += bilinWeights.w * WaveReadLaneAt(curPixelTexelValue, i3); 58 return color:

The first six lines of code are the same as for Box Sampling. The mask is stored in an uint64_t4, which has $16 \cdot 16 = 256$ bits, and so lines 14–15 call the fallback method if either of the bounding box dimensions is larger than 16. Lines 18–26 first set the 2×2 bits of the texels needed by the current lane in mask using a function set2x2Bits(); then it is OR of all of these over the entire wave using WaveActiveBitOr() that gives the full wave mask, waveMask.

Lines 28–33 compute how many texels are needed by counting the set bits in waveMask and useWaveActiveBallot(true) to find a bitmask of the active lanes in the wave. As with Box Sampling, the fallback method is called if the number of texels needed is more than the number of active lanes in the wave. Past this point, we know there are a sufficient number of active lanes and that Mask Sampling will succeed at getting all 2×2 texels needed by each active lane in the wave to perform perfect bilinear filtering.

In lines 41–45, the lanes with lane number less than the number of needed texels produce a texel, which in this example simply does a texture lookup using the GPU's texture unit. The bijective function h(i,B) (Figure 3) is used to compute local texture coordinates, which are added to the wave's upper left coordinates and the lookup is performed at the end. Line 48 computes the local one-dimensional texture coordinate of the upper left texel for the current lane. Since we know that the current lane wants the 2×2 texels starting from upperLeftIntCoords, we



bilinear ground truth Mask Sampling w/o Mask Sampling with edge remapping edge remapping

Figure S1: Left: bilinear ground truth filtering near a triangle edge. Middle: if we require the first n lanes to be active if n texels are needed, then the fallback is often used and results may be noisy. Right: with the remapping technique described in Section S2, we only need n active lanes across the wave and this problem is largely eliminated.

can transform that to 2×2 one-dimensional texture coordinates as $\{t, t+1, t+16, t+16+1\}$, where t+1 identifies the texel to the right of t, and t+16 the texel below t, since the mask is 16×16 bits. These coordinates are then fed to the inverse of the bijective function, i.e., $h^{-1}(t,B)$, and then the texel is produced from that lane and weighted, with perfect bilinear filtering as a result.

Similar to Box Sampling, we note that the algorithm above works for any texture filter, only requiring changes to the calculations of upperLeftIntCoords, lowerRightIntCoords, computeBilinearWeights, set2x2Bits, and the final unrolled loop.

S2. Edge Remapping

Close to a triangle edge against the background, a wave will typically not have all of its lanes active. So far, we have assumed that at least the first n lanes are active when n texels are needed and so the mappings from lanes to texels in Box and Mask Sampling may end up with inactive lanes being assigned to generate texels but then not actually doing so. In such cases, we may use the fallback method introduced in Section 3.4 of the main paper, though that can give artifacts as shown in the middle in Figure S1.

Alternatively, we can map lanes to texels more carefully, accounting for inactive lanes. Assume for now that the size of a wave is 8: if all lanes are active, then we have the active lane mask 11111111. In that case Mask Sampling, for example, can simply use its bijective function, t = h(i,B), to map a lane index into a texel coordinate. This fails if the active lane mask is, for example, 11101010. Still, if only four lanes are needed, then we can see that since the number of bits in the mask is ≥ 4 , it should be possible to run the algorithm anyway.

If we enumerate the needed texels from 0 to n - 1 with a number *i*, then we need a way to map that number into a lane number in the active lane mask where there is a 1. This can be done with the same bijective function as is used in Mask Sampling. We simply use h()

T. Akenine-Möller, P. Ebelin, M. Pharr, B. Wronski / Collaborative Texture Filtering



Figure S2: The textures used in our image quality measurements.

once again, i.e., we remap so the texel number is remapped to lane index as h(i,A), where A is the active lane mask. This technique can also be applied to Box Sampling.

On the right of the Figure S1, we present the result of our edge remapping, which substantially improves the image quality. However, it is still sometimes necessary to use the fallback method. This can happen, for example, when two texels are needed (e.g., due to clamping), but there is only a single active lane in the wave.

S3. Additional Evaluation Details and Results

In this section, we provide specifics about the scenes and textures used in the paper as well as the image quality metrics we used. We also present additional quality results.

S3.1. Scenes and Textures

We used two primary scenes and six textures for evaluation.

The first test scene consisted of a single quad, and a camera positioned directly above the quad's center, viewing it head on. The degrees of freedom used for this scene was the camera's distance to the quad, which set the effective magnification factor, the quad's rotation around the axis corresponding to the camera's viewing direction, as well as the texture used for the quad. The quad was shaded with a simple shading algorithm that only considered diffuse and normal textures, with lighting defined by a single light vector. This scene was used for Figures 4, 6, S4, S5, S7, S6, and S8, as well as Table S1.

To analyze the behavior of our algorithms with partially active waves and texture discontinuities, our second scene included 4×4 tessellated spheres in addition to the quad. The camera started some distance away from the quad so that a small amount of minification occurred, and then moved toward the spheres and back. The spheres and the quad used the same texture, though the version used on the spheres was a low-resolution version of the texture, so that we would have magnification also on the spheres. Each mesh in this scene was shaded using the same shading algorithm as the quad in the first scene. In total, the camera animation was 6 seconds long and rendered at 60 frames per second. It is shown in our supplemental video. This scene was used for Figures 5 and S3, as well as Table 1. A version similar to this scene, but with different camera animations and number of spheres, was used for Figures 1 and 8.

The resolution of the images we rendered was 1600×1600 , and the field of view of the camera was 45 degrees.

The five textures used throughout the results section of this work were the diffuse and normal textures for the AERIALROCKS, BRICKS, DIRT, PAINTEDCONCRETE, and RAILS sets. This set of textures shows variety in both diffuse and normal content, including both high- and low-frequency information. Each texture image has a resolution of 4096×4096 and we present them in Figure S2.

S3.2. Metric Specifics

Our results contain PSNR values as well as output from the ColorVideoVDP [MHA*24] and FLIP metrics [ANA*20].

For single images, PSNR was computed as usual by

$$PSNR(\mathbf{G}, \mathbf{T}) = -10\log(MSE(\mathbf{G}, \mathbf{T})), \quad (S1)$$

where **G** is the ground-truth image, **T** is the test image, and MSE is mean-squared error. Both ground-truth and test images were in linear sRGB space. In cases where we have *N* image sequences, each containing *M* frames, we take the average of the individual frame pairs' MSEs before computing PSNR. That is, if $\hat{\mathbf{G}}^i = {\{\mathbf{G}_j^i\}_{j=0}^{M-1}}$ and $\hat{\mathbf{T}}^i = {\{\mathbf{T}_j^i\}_{j=0}^{M-1}}$ are the sets of *M* ground-truth and test frames for sequence *i*, $i \in {0, 1, ..., N-1}$, we compute PSNR as

$$PSNR\left(\{\hat{\mathbf{G}}^{i}\}_{i=0}^{N-1}, \{\hat{\mathbf{T}}^{i}\}_{i=0}^{N-1}\right) = -10\log\left(\frac{1}{NM}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}MSE(\mathbf{G}_{j}^{i}, \mathbf{T}_{j}^{i})\right).$$
(S2)

Notice that the computation in Equation S2 gives the same result as that in Equation S1 when only a single frame pair is considered.

The ColorVideoVDP model requires information about the assumed observer's viewing conditions and display. We use the default settings, and set the assumed number of frames displayed per second to 60 for all our image sequences. The reproducibility information provided when running the metric was "ColorVideoVDP v0.4.2, 75.4 [pix/deg], Lpeak=200, Lblack=0.2, Lrefl=0.3979 [cd/m^2], (standard_4k)." ColorVideoVDP's output is in Just-Objectionable Differences (JOD) units, scaled to have a maximum value of 10. Only if the reference and test images/sequences are visually indistinguishable under the assumed viewing conditions is the JOD value of the test equal to 10. In cases where we compute one JOD over a set of image sequences, that JOD is the average of the JODs for each individual sequence. Using the notation introduced above, we have

$$ColorVideoVDP\left(\{\hat{\mathbf{G}}^{i}\}_{i=0}^{N-1}, \{\hat{\mathbf{T}}^{i}\}_{i=0}^{N-1}\right)$$

$$= \frac{1}{N} \sum_{i=0}^{N-1} ColorVideoVDP\left(\hat{\mathbf{G}}^{i}, \hat{\mathbf{T}}^{i}\right).$$
(S3)

Note that ColorVideoVDP is also able to compare single images. We use this capability for the results in Figure 6.

Like ColorVideoVDP, FLIP requires information about the assumed observer's viewing conditions. In particular, it takes the observer's distance to the display as well as the display's width (in meters and in pixels). To match the assumptions for ColorVideoVDP and FLIP, we use the information provided by ColorVideoVDP for its standard_4k display, namely a distance to display of 0.7472 meters and a display width of 0.664 meters and 3840 pixels. As FLIP acts on single images, and not image sequences, the FLIP error we present for multiple image sequences is the average of each frame pairs' FLIP error, i.e.,

$$\begin{aligned} & \text{FLIP}\left(\{\hat{\mathbf{G}}^{i}\}_{i=0}^{N-1}, \{\hat{\mathbf{T}}^{i}\}_{i=0}^{N-1}\right) \\ &= \frac{1}{NM} \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \text{FLIP}(\mathbf{G}_{j}^{i}, \mathbf{T}_{j}^{i}). \end{aligned} \tag{S4}$$

S3.3. Quality Comparison Without Denoising

In this section, we present diagrams showing quality/performance efficiency and the quality as a function of magnification corresponding to the main paper's Figures 5 and 6, but without denoising the evaluated sequences. The results are shown in Figures S3 and S4. Furthermore, in Figure S5 and Table S1, we examine how the maximum image error differs between the considered algorithms.

We start by noting that the same methods lie on the Pareto frontier, independent of whether the sequences are denoised (Figures 5 and S3). Furthermore, in Figure S4, as also shown in Figure 4, we again see that our algorithms provide perfect bilinear filtering above a magnification threshold (1.59 for Mask Sampling and 2.35 for Box Sampling), while the One-tap STF [PWSF24] and Wave Communication STF [WPAM25] algorithms are unable to achieve that for any magnification factor due to their inherently stochastic nature. At lower magnification factors, where our algorithms are unable to reach perfect filtering and must use fallback alternatives, they still give higher quality than the prior techniques. (A comparison between our fallback methods and the state-of-the-art algorithms is included in Section S3.5.) Furthermore, both Figures S3



Figure S3: Pareto frontier (dashed line) indicating the most quality/performance-efficient algorithms when rendering without denoising. Runtime cost was measured on an NVIDIA RTX 5090. Our new fallback algorithms are marked with C and C+. The PSNR range for this plot was 29–59 dB.



Figure S4: Quality as a function of magnification, without denoising. As indicated by Figure 4, perfect bilinear filtering is achieved above magnification factor 2.35 for our methods. Colored discs mark that an algorithm has achieved perfect bilinear filtering at the corresponding magnification factor, resulting in infinite PSNR. At that and higher magnification factors, perfect bilinear filtering occurs for that algorithm. The small black circle marks a case when perfect bilinear filtering is achieved at a magnification lower than the limits indicated by the disks. This situation can arise for certain magnification and rotation combinations.

and S4 show that Mask Sampling yields higher-quality results than Box Sampling. We also confirm our earlier findings that the Mask Sampling quality increase comes at a performance trade-off, as Box Sampling is the faster of the two.

Finally, we consider the plots in Figure S5 and the results in Table S1, where the plots show the *maximum* errors produced by the algorithms we consider, and the table shows the averages of the curves in those plots. We generated these results using the same simple magnification setup as explained in Section S3.1, with the

T. Akenine-Möller, P. Ebelin, M. Pharr, B. Wronski / Collaborative Texture Filtering



Figure S5: Maximum errors (scaled to [0,255]) produced by the compared algorithms for rotations by $r \in [0,90]$ degrees, given three different levels of magnification. The legend is included above the plots. Note that the y-axis for the rightmost plot differs from that in the other two. The errors from Box and Mask Sampling in Figure S5a are the same, but the curves for Box Sampling are offset by 0.5 for visibility. As implied by Figure 4, at magnification factors larger than 1.59, the maximum error of Mask Sampling is zero for all rotations, while Box Sampling requires more magnification (2.35 or higher) to produce perfect bilinear filtering.

Table S1: Average maximum errors (scaled to [0,255]) across the rotations used in Figure S5, for each of the different magnification factors used in that figure. One-tap STF is the algorithm by Pharr et al. [PWSF24], Wave Comm. is short for Wave Communication STF and is the algorithm by Wronski et al. [WPAM25], Box is short for Box Sampling, and Mask is short for Mask Sampling.

Zoom	One-tap STF	Wave Comm.	Box (One-tap STF)	Box (Wave Comm.)	Box (C)	Box (C+)	Mask (C)	Mask (C+)
1.15	174	163	174	163	161	147	161	147
1.35	172	157	172	156	152	137	150	135
1.55	174	153	154	135	132	117	7	6

quad rotating between [0,90] degrees for each measurement. For low magnification, Box and Mask Sampling perform the same (the Box Sampling curves in Figure S5a are offset slightly for visibility). As magnification increases, Mask Sampling starts producing lower errors than Box Sampling, when both use the same fallback method. At a magnification factor of 1.55, Mask Sampling is able to produce perfect bilinear filtering for most rotations, while Box Sampling requires higher magnification to do so (see Figure 4). The more complex fallback method (C+) often give lower maximum errors than the simpler, faster one. These results agree with Figure S4.

S3.4. Convergence Analysis

We also analyzed whether or not the output of our algorithms converges to the ground-truth image when we increase the number of samples drawn per pixel (SPP), or how large the bias is if they do not; see Figure S6, which shows PSNR as a function of SPP. For these results, we once again use the same simple, rotated quad and set of textures described in Section S3.1. For a given SPP-value, PSNR was computed over a set of magnification factors in the [1.0, 2.5] range, as that is approximately the range where one or more of our algorithms are unable to produce perfect bilinear filtering (see Figure 4). Figure S6 shows that our algorithms achieve better results, though they also show some bias, similar to One-tap STF [PWSF24] and Wave Communication STF [WPAM25]. This



Figure S6: Plot of the error convergence of the considered algorithms when compared to perfect bilinear filtering. The scene used was the first scene described in Section S3.1. Ten magnification factors in the range [1.0, 2.5] were used for each PSNR value.

is a consequence of the difference between filtering after shading and filtering before shading, as discussed by Pharr et al. [PWSF24].

S3.5. Fallback Evaluation

In extreme cases, every wave could contain a silhouette edge, which means that the fallback (Section 3.4) would be called for every pixel. We therefore present image quality results for our fallback methods as a function of magnification factor in Figure S7, both without and with denoising.

Even our simplest proposed fallback (C) provides better image quality than both One-tap STF [PWSF24] and Wave Communication STF [WPAM25]. We also note that without DLSS (top diagram), our C+ method continues to increase the image quality for higher magnification factors, while that happens to a lesser extent for the other methods. This is a consequence of C+ leveraging unused lanes to produce more unique texels and as magnification increases there are more and more unused lanes that can be used to increase image quality. For the comparisons with denoised sequences, since we apply denoising both to the ground-truth and the test sequence and the latter almost is identical to the former (see the high PSNR values for high magnification in Figure S7a), the small errors become slightly larger, spatially, due to the slight blur caused



Figure S7: Evaluation of image quality as a function of magnification factor for all the different fallback methods. For both without and with denoising (DLSS [NVI25]), our algorithms provide superior quality compared to the other methods.

by the denoiser. Presumably, this results in lower PSNR values for the C+ method after denoising compared to before. Although the dB difference caused by these differences is large, the per-pixel errors in the denoised sequences are very small: at high magnification, the largest errors for the C+ method are 1–2 when scaled to the [0,255] range. However, we note that the denoised results for C+ show significantly worse quality than the non-denoised ones around magnification factor 7–8. While inspecting the data, we found that the quality decrease originated from the RAILS results, where errors became slightly larger for those magnification factors despite the image content not changing drastically. This was due to the denoised C+ images being marginally more blurry than the ground truth at those magnification factors. For the other textures, this effect was not observed.

S3.6. Additional Bicubic Results

Finally, we measured the amount of magnification that is needed to achieve perfect bicubic filtering with each of List Merge, Box Sampling, and Mask Sampling, given 32 lanes per wave. Figure S8a is the bicubic counterpart to Figure 4. That is, it shows how much magnification is needed at different rotations of a quad to achieve perfect bicubic filtering with 32 or fewer texture lookups per wave. Figure S8b is similar, but assumes that we are able to do two texture lookups per lane, so that we instead need 64 or fewer lookups per wave. The figures show that perfect bicubic filtering with our methods requires significantly more magnification if we only allow one lookup per pixel. If two lookups are allowed, we see that the required magnification factor is closer to that for perfect bilinear filtering, despite the four times larger filter area.

16 of 16

T. Akenine-Möller, P. Ebelin, M. Pharr, B. Wronski / Collaborative Texture Filtering



Figure S8: Illustration of the minimum degree of magnification necessary to achieve perfect filtering under different rotations of the quad for perfect bicubic filtering. Below those magnification factors, our algorithms need to rely on fallback methods (Section 3.4). Cases where a fallback was necessary are indicated by colored areas. The scene used was the first one described in Section S3.1. The results in the left diagram are based on producing no more than one texel per lane, while those in the right diagram allow up to two texels per lane. Notice that List Merge and Mask Sampling both cover identical areas in both figures. Like in the bilinear case, the success rate of Box Sampling is lower than that of the other two techniques.