Improved Stochastic Texture Filtering Through Sample Reuse

BARTLOMIEJ WRONSKI, NVIDIA, USA
MATT PHARR, NVIDIA, USA
TOMAS AKENINE-MÖLLER, NVIDIA, Sweden

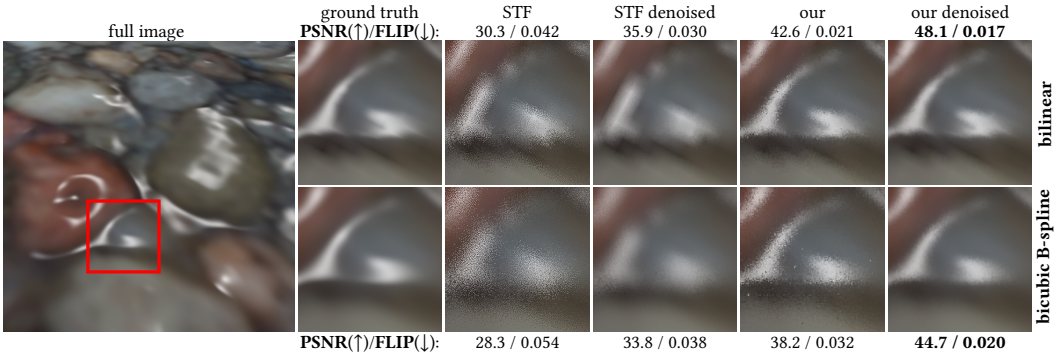

Fig. 1. We present an improved stochastic texture filtering (STF) method with no additional texture lookup cost compared to standard STF. Above, the images from our method have between 9.9 and 12.3 dB higher PSNR and lower FLIP errors for both bilinear and bicubic B-spline filtering compared to standard STF. DLSS was used for denoising.

Stochastic texture filtering (STF) has re-emerged as a technique that can bring down the cost of texture filtering of advanced texture compression methods, e.g., neural texture compression. However, during texture magnification, the swapped order of filtering and shading with STF can result in aliasing. The inability to smoothly interpolate material properties stored in textures, such as surface normals, leads to potentially undesirable appearance changes. We present a novel method to improve the quality of stochastically-filtered magnified textures and reduce the image difference compared to traditional texture filtering. When textures are magnified, nearby pixels filter similar sets of texels and we introduce techniques for sharing texel values among pixels with only a small increase in cost (0.04–0.14 ms per frame). We propose an improvement to weighted importance sampling that guarantees that our method never increases error beyond single-sample stochastic texture filtering. Under high magnification, our method has >10 dB higher PSNR than single-sample STF. Our results show greatly improved image quality both with and without spatiotemporal denoising.

CCS Concepts: • **Computing methodologies** → **Texturing**; Image processing; Image compression.

Additional Key Words and Phrases: stochastic texture filtering, importance sampling, wave intrinsics.

Authors' Contact Information: Bartlomiej Wronski, NVIDIA, Brooklyn, USA, bwronski@nvidia.com; Matt Pharr, NVIDIA, San Francisco, USA, matt@pharr.org; Tomas Akenine-Möller, NVIDIA, Lund, Sweden.

ACM 2577-6193/2025/5-ART14
https://doi.org/10.1145/3728292

ACM Reference Format:
Bartlomiej Wronski, Matt Pharr, and Tomas Akenine-Möller. 2025. Improved Stochastic Texture Filtering Through Sample Reuse. *Proc. ACM Comput. Graph. Interact. Tech.* 8, 1, Article 14 (May 2025), 25 pages. https://doi.org/10.1145/3728292

1 Introduction

The gap between computation and memory bandwidth in microprocessors has been increasing for nearly 50 years [Dongarra 2022]. GPUs are not immune to this trend: over the past 20 years, the computation available on GPUs (measured by FP32 GFLOPS) has increased by a factor of $\sim 2,750\times$, while the peak off-chip memory bandwidth has increased by a factor of just $\sim 60\times$.[1] If specialized tensor operations in hardware is included, the growth in available computation is up to an order of magnitude greater, depending on the precision. This trend has motivated the development of new compression techniques that trade off increased computation to save bandwidth; examples include neural texture compression (NTC) [Vaidyanathan et al. 2023] and NeuralVDB [Kim et al. 2024]. However, these techniques are not compatible with current GPU texture filtering hardware and may be too expensive even for a simple bilinear filter decoding 2×2 texels. Building on earlier work by Hofmann et al. [Hofmann et al. 2021], Vaidyanathan et al. [Vaidyanathan et al. 2023], and others, Pharr et al. [Pharr et al. 2024] thus introduced a family of Monte Carlo stochastic texture filtering (STF) techniques to approximate the effects of traditional texture filtering without increasing the texture sampling cost beyond a single sample.

STF has the additional advantage that it allows efficient implementation of *filtering after shading* (FAS), where the antialiasing filter is applied to the final shaded value rather than the texture inputs to a shader [Pharr et al. 2024]. When the shading function is linear or an affine combination of the input texture values, both filtering before shading (FBS) and FAS converge to the same result. When it is non-affine, results differ: FAS is generally more accurate than FBS when textures are minimized, but it has a number of disadvantages when textures are magnified [Pharr et al. 2024, Section 3.2]:

- FAS can introduce aliasing under magnification.
- FAS is unable to reproduce smooth interpolation of some properties (e.g., surface curvature).
- FAS may produce results different than those intended by the author of the art assets.

Further, spatiotemporal reconstruction techniques like TAA [Karis 2014; Yang et al. 2020] and DLSS [Liu 2022] may fail to converge and produce residual noise when used to integrate STF samples [Pharr et al. 2024, Section 6].

In this work, we address these limitations and improve the quality of stochastic texture magnification without increasing the cost beyond a single texture lookup and without affecting the favorable properties of FAS for minification. We base our approach on a simple insight: under magnification, adjacent screen pixels tend to filter the same set of texels. Thus, if each pixel continues to take a single STF texel sample and shares this sample with nearby pixels, then each pixel may be able to compute a more accurate estimate of the true filtered value. This sharing can be performed efficiently by taking advantage of the SIMT/SIMD nature of GPU execution to communicate between shader threads without any VRAM memory traffic. By averaging multiple texel values, our method performs some filtering before shading. Intuitively, it can be seen as a hybrid between filtering after shading and filtering before shading.

Our contributions include:

[1] We compare the NVIDIA GeForce FX 5800 Ultra, 30 FP32 GFLOPS, 16 GB/s bandwidth, released March 2003, to the NVIDIA 4090 RTX, 82.6 TFLOPS, 1.01 TB/s bandwidth, released October 2022; GPUs from other manufacturers share similar relative ratios.

- An algorithm for efficient STF sample reuse and sharing using wave intrinsics (Section 4) including practical implementation details (Section 4.5).
- An in-depth analysis of the error of the traditional STF estimator (Section 3) including the impact of filtering after shading of non-affine rendering functions. We further explain in the supplementary material (Section S2) the relationship between the estimator variance and the bias introduced by reordering filtering and shading.
- A new Monte Carlo estimator based on weighted importance sampling with a lower error than previous estimators when used for stochastic texture filtering (Section 4.2).
- An algorithm that generates optimized *sharing footprints* introducing variation, in which pixels share texel values with each other and distribute error as blue noise in the rendered images (Section 4.3).
- An algorithm to generate custom blue noise patterns optimized to account for this sharing (Section 4.4).

Under magnification, these techniques provide much better image quality than traditional STF, both visually and using error metrics; see Figure 1. We validate our claims and analyze results with a number of Monte Carlo estimators, different pseudo-random sequences and using different spatiotemporal denoisers in Section 5.

2 Background and Previous Work

Before introducing our algorithm, we first provide relevant background on texture filtering, Monte Carlo sampling, and the GPU quad and wave intrinsics we use for efficient communication between nearby pixels.

2.1 Texture Filtering and Representations

A texture $t(u, v)$ is defined by a set of texels $\mathbf{T}_{u_i,v_i}$ defined at integer coordinates (u_i, v_i) on a grid, scaled by translated Dirac delta functions:[2]

$$t(u, v) = \sum_i^n \delta(u - u_i)\delta(v - v_i)\, \mathbf{T}_{u_i,v_i}. \tag{1}$$

With this notation, n is the total number of texels in the texture. To construct a continuous texture function, it is necessary to specify a reconstruction filter f_r and convolve it with the texture function:

$$t_\mathrm{r}(u, v) = t \otimes f_\mathrm{r} = \iint t(u', v')\, f_\mathrm{r}(u - u', v - v')\, \mathrm{d}u'\, \mathrm{d}v', \tag{2}$$

where $t \otimes f_\mathrm{r}$ is a convolution. In turn, the filtered value at a point $(u, v) \in \mathbb{R}^2$ can be written as a sum of weighted texel values. Given a (u, v) lookup point, we will generally write Equation 2 as

$$t_\mathrm{r}(u, v) = \sum_i f_\mathrm{r}(u - u_i, v - v_i)\, \mathbf{T}_{u_i,v_i}. \tag{3}$$

The bilinear (tent) and bicubic filters are commonly used for texture reconstruction. Given such a filter with limited spatial extent, these sums only need to be over a few weighted texels.

[2]Without loss of generality, we focus on filtering 2D textures and use this assumption throughout the text, but the described techniques apply to any texture dimensionality.

2.2 Monte Carlo Estimators

The standard importance sampling Monte Carlo (MC) estimator samples a random variable X from a probability distribution function (PDF) p, $X \sim p$ and gives the estimate

$$F = \frac{f(X)}{p(X)} \approx \int f(x)\,\mathrm{d}x. \tag{4}$$

This estimator is unbiased if $p(x) > 0$ everywhere when $f(x) > 0$ [Pharr et al. 2023].

A continuous Monte Carlo integral estimator like Equation 4 can be converted to apply to a discrete sum by introducing Dirac delta functions centered at the points where the sum is being evaluated. PDFs are then comprised of corresponding Dirac deltas multiplied by discrete *probability mass functions* (PMFs) that sum to 1. Such an approach was effectively taken in previous applications of STF, where one sample was taken from the sum in Equation 3 based on a PMF proportional to the filter weight. In turn, the importance sampling Monte Carlo estimator approximates the filtered value $t(u, v)$ with a single unweighted texel value; we will call this approach *one-tap STF* in the following. See Section S1 for a derivation showing how the continuous estimator of Equation 4 was applied for one-tap STF.

The Monte Carlo literature offers extensive analysis of convergence of not only estimators, but also different sources of randomness, such as different pseudo- and quasi-random sequences. Most of this analysis focuses on convergence properties with sample counts growing toward infinity. In contrast, real-time rendering uses spatiotemporal filtering [Karis 2014; Yang et al. 2020] and can realistically average only a small number of past samples, sometimes combined with spatial filtering [Liu 2022]. Therefore, real-time rendering focuses on minimizing the perceptual error for a minimal number of samples, assuming small radius spatial filtering and an exponentially moving average temporal filter. The most common randomness sources in real-time rendering are blue noise dithering masks [Georgiev and Fajardo 2016], such as the spatiotemporal blue noise masks [Wolfe et al. 2022] used by Pharr et al. [Pharr et al. 2023]. Wolfe et al. [Wolfe et al. 2022] proposed two different algorithms for generating those masks—a scalar version based on the void and cluster algorithm, and a vector version based on simulated annealing. In Section 4.4, we discuss the impact of those masks on the quality of our algorithm as well as compare them with the most recent spatiotemporal blue noise mask generation advancements [Donnelly et al. 2024].

2.3 Wave Intrinsics

We make extensive use of *wave intrinsics* to share samples between pixels; they were introduced in DirectX HLSL Shader Model 6.0 [Microsoft 2021]. Normally in shader programming, only a single thread of execution is exposed, meaning that each thread does not know what its neighbors are doing. Wave intrinsics allow threads to communicate. HLSL 6.0 uses the term *lane* for a single thread of execution, and *wave*, sometimes called a warp, for a set of lanes that are executed together in parallel. Wave sizes are hardware dependent and not guaranteed by the DirectX specification, but common sizes are 8, 16, 32, and 64 lanes. All our evaluation (Section 5) is done on NVIDIA hardware, where recent GPUs have a wave size of 32; our work generalizes to arbitrary wave sizes. A lane can access a value from any lane in its wave using `WaveReadLaneAt(value, laneId)`; see Figure 2. Wave intrinsics are an attractive method for inter-lane communication since

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Fig. 2. An example of a 32 lane wave: eight 2×2 quads configured as 8×4 pixels. The numbers are lane ids, configured for linear row ordering. Each pixel corresponds to a lane in the wave, and it can access a value of the other wave lanes using `WaveReadLaneAt()`.

they are typically implemented as swap or shuffle instructions within a wave [Microsoft 2021], and thus cost only one instruction with no memory bandwidth impact. Previous work exploited pixel shader quad derivatives to perform in-place screen-space filtering (such as bilateral or convolution filters) [McGuire et al. 2012; Penner 2011]. We build upon those ideas using modern wave intrinsics and generalize beyond screen-space effects.

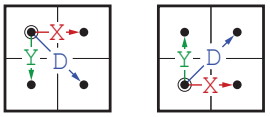

Fig. 3. Quads with the current pixel marked with a circle. The values accessed by `QuadReadAcrossX/Y/Diagonal()` are marked `X`, `Y`, and `D`.

Under magnification, adjacent screen pixels sample adjacent source texels and we aim to share samples between adjacent pixels to improve filtering quality. The zero memory cost of wave intrinsics is highly beneficial for this purpose. Before we describe and motivate our approach in detail, we discuss how wave lanes are mapped to screen pixels. We will show in Section 4.3 that this mapping affects the possible sample sharing configurations and filtering quality.

For compute shaders, it is the programmer's responsibility to map the dispatch indices (and, as a consequence, the wave lane indices) to processed elements, such as pixels, vertices, or rays. Figure 2 organizes 32 lanes as 8×4 lanes but other configurations such as 16×2 lanes are possible.

With pixel shaders, there is only a guarantee of 2×2 quad granularity with the upper left pixel of each quad located at even (x, y) coordinates on the screen. Two quads are shown in Figure 3. HLSL provides special helper functions for quad communication. For a single pixel in a 2×2 quad, one can access computed values in the other pixels in the quad using the `QuadReadAcrossX()`, `QuadReadAcrossY()`, and `QuadReadAcrossDiagonal()` functions (Figure 3). It is still possible to access other wave lanes using the `WaveReadLaneAt()` function in pixel shaders, but they can map to arbitrary screen pixels, inactive lanes, and depend on the rasterizer behavior.

3 Error Analysis of Stochastic Texture Filtering

We observe that the error and noise characteristics of STF do not resemble traditional Monte Carlo rendering noise. A single sample of the rendering equation MC integral is typically extremely noisy, even given a smooth signal. For this reason, most of the Monte Carlo literature analyzes the average error and the error convergence. By comparison, many pixels produced by STF are not noisy, even without temporal integration. See Figure 1 as an example. Aiming to improve STF, we analyze the error of a single STF sample and show that it is not only bounded, but is zero in many common cases.

We begin by observing that all valid texture filters have weights that sum to one. Without loss of generality, we assume a filter with nonnegative weights.[3] The one-tap STF importance sampling estimator effectively selects one of the samples of the texture where the texture filter is nonzero (one element of the sum in Equation 3). To analyze the maximum error of a single sample, we can ignore the sampling probabilities and look at the individual selected samples. The STF estimator has a geometric interpretation—the filtered value lies inside the convex hull of texel values, and STF selects one of the texels. The one-tap STF estimator is thus guaranteed to never leave the range of the contributing texels' values. This property is crucial for texture filtering, as textures often encode physical properties with a defined valid range. The highest error case occurs when the selected texel differs the most from the filtered value, which can be arbitrarily close to another texel.

[3] A similar analysis applies to filters with negative lobes sampled with positivization [Pharr et al. 2024], rescaling the convex hull bounds.

In other words, the maximum error of any interpolating texture filter with nonnegative weights is

$$\max_i \mathbf{T}_{u_i,v_i} - \min_i \mathbf{T}_{u_i,v_i}, \tag{5}$$

with i ranging over the contributing texels.

From Equation 5 and the geometrical interpolation interpretation, it is clear that when all filtered texel values are the same, the one-tap STF estimator error is zero. Natural and computer-generated images have a low-frequency spectral bias [Reinhard et al. 2001] and adjacent texel values often change slowly. While this is not the case for all scenes, many rendering texture assets contain large flat and low-frequency regions. In those regions, the STF estimator error is close or equal to zero.

Given the low error and favorable properties of the existing STF estimator, we set the following goals for the improved estimator:

(1) It should not produce any error where the one-tap STF estimator yields zero error.
(2) It should not increase the error in common low-frequency regions.
(3) It should reduce the error and variance in high-frequency regions, e.g., edges, where the maximum and minimum local values differ significantly.

To achieve properties 1 and 2, we argue that a texture filtering estimator must produce results within the convex hull of the texel values. In Section 5.1, we show how violating those requirements can lead to poor image quality and introduction of new errors as compared to one-tap STF.

4 A Stochastic Texture Filtering Algorithm with Sample Reuse

In this section, we present our entire algorithm for STF with texel reuse across neighboring pixels.

4.1 Evaluation of Existing Estimators

With our approach, each lane starts by sampling a single texel according to its texture filter, just like one-tap STF. After sample sharing, each lane has n sampled texels x_i, each drawn from a PMF p_i corresponding to a distinct lane's texture filter. Our task is to compute an estimate of the texture filtering Equation 2 using these samples. We start by considering the use of standard MC estimators.

The standard importance sampling estimator computes the filtered value as the weighted texel values divided by the probabilities p_i of sampling each x_i:

$$\frac{1}{n}\sum_i^n \frac{f(x_i)}{p_i(x_i)}. \tag{6}$$

Although importance sampling requires that $p_i > 0$ whenever $f > 0$; this may not be the case with sample sharing since other lanes' filters (and thus PMFs), generally differ from the current lane's. If the importance sampling estimator is still used, the error may be unbounded. Further, even if a PMF p_i is valid, the estimator may still produce arbitrarily high error—in the context of STF, consider a pixel where the filter weight for a texel $\mathbf{T}_{u_i,v_i}$ is very small; if it samples that texel and shares it with a neighboring pixel where the filter weight is much larger, the ratio f/p may be arbitrarily large, leading to high variance. This is illustrated in Figure 4.

One way to ensure that the PMFs are valid is by using *defensive importance sampling*, where for example a constant PMF over the entire domain is mixed with the regular sampling PMF [Hesterberg 1995]. However, we have found that this may lead to pixels sometimes having *zero* texels that are inside their own filter, making this approach unsuitable. *Multiple importance sampling* (MIS) [Veach and Guibas 1995] provides an estimator that reduces the variance when some PMFs are a better match to the function than others and further requires that only one of the PMFs be nonzero when $f > 0$. The MIS balance heuristic (and Veach's other heuristics) have the disadvantage of n^2 PMF

evaluations. This cost can be noticeable when sharing samples across multiple pixels. It can be avoided with Bitterli's *pairwise MIS*, which only requires $2n$ PMF evaluations [Bitterli 2022].

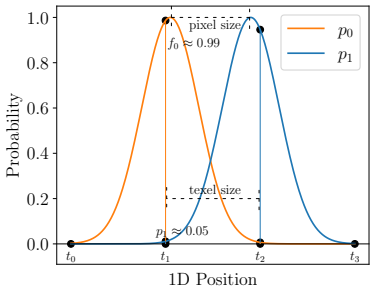

Fig. 4. Reconstruction filters for two adjacent pixels in 1D. Texel locations are on the x-axis. When pixel 1 samples texel t_1 from the distribution p_1 according to its texture reconstruction filter and then shares it with pixel 0, the standard MC importance sampling estimator weights this texel $f_0/p_1 \approx 20$ causing a variance spike, even for constant filtered signals.

We have also considered a *regression estimator* that is based on computing weights for a control variate based on the sample values [Owen 2013, Equation 9.11]. It is:

$$\frac{1}{n}\sum_i^n \frac{f(x_i)}{p_i(x_i)} - \beta\left(\frac{w_i(x_i)}{p_i(x_i)} - 1\right), \tag{7}$$

where $w_i(x_i)$ is the filter weight for texel x_i and

$$\beta = \frac{\sum_i^n (w_i(x_i) - \bar{w})\, f(x_i) w_i(x_i)}{\sum_i^n (w_i(x_i) - \bar{w})^2}, \tag{8}$$

with $\bar{w} = (1/n)\sum_i^n w_i(x_i)$.

One problem with all the approaches in this section is that they do not fulfill the first two goals identified in Section 3 and may yield results outside the convex hull of texel values. One way to work around this issue is to clamp the filtered texture value to lie withing the bounds of texel values used. This introduces bias but significantly reduces the error as we show in our evaluation (Section 5.1).

4.2 Our Estimator

Weighted importance sampling (WIS) estimators (also known as *self-normalized importance sampling*) provide an alternative that provide results bounded by the texel values. WIS began with the "weighted uniform" estimator, suggested by Handscomb [Handscomb 1964] and later analyzed by Powell and Swann [Powell and Swann 1966]. Spanier [Spanier 1979] suggested and analyzed two successive generalizations, leading to sampling from one PDF $x_i \sim p_1$ and then reweighting using a second:[4]

$$\frac{\sum_i^n f(x_i)/p_1(x_i)}{\sum_i^n p_2(x_i)/p_1(x_i)}. \tag{9}$$

These WIS estimators both have a small bias but are *consistent*. In graphics, WIS is commonly used when filtering pixel samples in path tracers, for example [Pharr et al. 2023, Section 5.4.3]. Other prior applications of WIS in rendering include Monte Carlo radiosity [Bekaert et al. 2000] and a photon mapping technique [Szirmay-Kalos and Szecsi 2003].

We propose a generalization of Equation 9 that allows taking samples from n different PDFs $x_i \sim p_i$, as is the case with texel sharing. Assuming for now we have a continuous PDF p_c that is nonzero where $f(x) > 0$, and additional PDFs $p_i(x)$ that are nonzero when p_c is, then our estimator is

$$\frac{\sum_i^n f(x_i)/p_i(x_i)}{\sum_i^n p_c(x_i)/p_i(x_i)}, \tag{10}$$

which we believe is novel.

[4]The reweighting in WIS bears some similarity to importance resampling [Bitterli et al. 2020; Talbot et al. 2005], though those methods are used when it is inexpensive to generate candidate samples and where selecting a subset of those samples for evaluation is desired; this is not the case for texture filtering. See Bitterli et al. [Bitterli et al. 2020, Appendix B] for further discussion of connections between WIS and resampled importance sampling as well as to Monte Carlo ratio estimators.

This estimator is *consistent*, i.e., it converges to the value of the integral $\mu = \int f(x)\mathrm{d}x$ with probability 1. We can see this by first writing it in the equivalent form:

$$\frac{\frac{1}{n}\sum_i^n f(x_i)/p_i(x_i)}{\frac{1}{n}\sum_i^n p_c(x_i)/p_i(x_i)}. \tag{11}$$

Following Owen [Owen 2013, Theorem 9.2], since $p_i > 0$ where $f > 0$, the numerator of Equation 10 is the average of independent random variables that each are unbiased estimates of μ. In turn, the numerator converges to μ. Next, because p_c is a PDF, it integrates to 1. We can use the same argument to show that the denominator converges to 1 as $n \to \infty$. Thus, their ratio converges to μ as well. Like other WIS estimators, our estimator is biased, but the bias is small in practice.

To apply this continuous estimator to the texture filtering sum, we take the reconstructed texture function t_r of Equation 2 for f and define corresponding PDFs with the product of Dirac delta functions at (u_i, v_i) and the reconstruction filters. (See Section S1 in the supplemental for a derivation showing how Pharr et al. effectively did this with one-tap STF [Pharr et al. 2023].) We would like to estimate the filtered texture value at a point (u, v), where each lane has sampled a texel location (u_i, v_i) from its texture filter f_i's PMF p_i. If we write the current lane's texture reconstruction filter as $f_c(u', v') = f_r(u - u', v - v')$ with an associated PMF p_c, the estimator is:

$$\frac{\sum_i^n f_c(u_i, v_i)\,\mathbf{T}_{u_i,v_i}/p_i(u_i, v_i)}{\sum_i^n p_c(u_i, v_i)/p_i(u_i, v_i)}. \tag{12}$$

If the texture reconstruction filter is positive and normalized then $p_c(u_i, v_i) = f_c(u_i, v_i)$ and similarly $p_i = f_i$, giving

$$\frac{\sum_i^n f_c(u_i, v_i)\,\mathbf{T}_{u_i,v_i}/f_i(u_i, v_i)}{\sum_i^n f_c(u_i, v_i)/f_i(u_i, v_i)}. \tag{13}$$

If we define weights $w_i = \frac{f_c(u_i,v_i)}{f_i(u_i,v_i)}$ and simplify, we have:

$$\sum_i^n \frac{w_i}{\sum_j^n w_j}\,\mathbf{T}_{u_i,v_i}. \tag{14}$$

Thus, the weights are normalized and sum to one, guaranteeing that our estimator always produces a convex combination of the filtered texels. As a result, constant texture regions have zero error.

However, unlike the general case, we do not necessarily have $p_i > 0$ where $f > 0$ and $p_c > 0$ and thus convergence is not guaranteed. We find in practice that the error due to this is small. Crucially, unlike the standard importance sampling estimator, ours does not have the risk of returning very large or small values due to a mismatch between f and the p_i. On the other hand, a significant f and p_i mismatch can still cause a visual error in high-contrast regions. The resulting high weight is normalized with other contributing weights but causes our algorithm to effectively select one of the texels from the other lanes. This can result in pixels with a firefly-like appearance in highly specular regions (see Figure 1).

4.2.1 Exact Filtering. Some simple filters, such as bilinear, require only four distinct texels for exact reconstruction. We observe that using high-quality, anti-correlated random number generators, such as blue noise, there is a high probability that all four distinct texels will be sampled in the pixel neighborhood. In such cases, we can return the filtered value directly without using our estimator. We call this extension *exact filtering*, and show an example of its efficiency in Figure 5 (two middle columns), and evaluate it in Section 5. It introduces an additional small bias but reduces the error. Exact filtering is practical with small filter footprints, such as bilinear. In contrast, a bicubic filter requires 16 texels for perfect reconstruction and the probability of sampling all those values even across an entire 32-element wave is low.

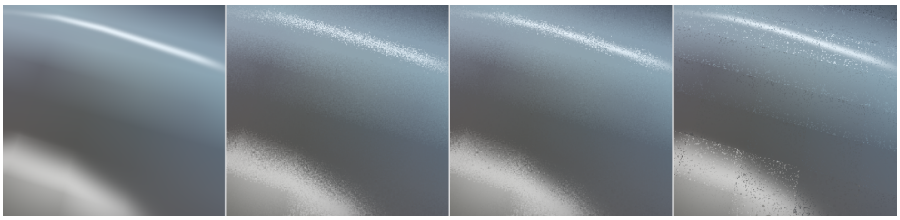

Fig. 5. From left to right: reference bilinear filtering, our WIS estimator, our WIS estimator with *exact filtering*, and exact filtering with equal sampling probabilities of texels inside of the filter footprint. The rightmost image shows a much higher percentage of pixels using exact filtering at the cost of severe visual artifacts, mostly visible at bilinear filter footprint boundaries.

A limitation of exact filtering is that because each pixel samples a texel according to its filter PMF, texels that only make small contributions to the pixels in a wave may be sampled only rarely. We briefly explored strategies that change the sampling probabilities to be more uniform under strong magnification, similarly to defensive importance sampling [Hesterberg 1995]. We found that while those strategies increase the number of pixels where exact filtering is possible, they can lead to severe grid-like visual artifacts (Figure 5 rightmost column). The main reason is that a lane cannot access elements outside of the wave. For instance, the top left pixel of a wave cannot access any elements to the left or above it, which might be needed for its exact bilinear filter depending on how pixels' UV values align with the screen pixel grid.

4.3 Sharing Footprints Within Waves

With our approach, each lane in a wave has an associated *texel sharing footprint* that specifies which other lanes it draws texel values from. With quad intrinsics, these footprints are restricted to fixed grid positions and access is allowed to only three neighbors (Section 2.3). With wave intrinsics, one is free to use any lanes inside the wave and any number of them, giving a wider design space. In the following, we assume a 32-wide wave, though the concepts apply to other wave sizes. A wave with 32 lanes may be configured as 16×2 or 8×4 pixels. We found that 8×4 is preferable, since it allows for larger, square-like sharing footprints, which tend to give lower error.

Our design criteria for sharing footprint size and placement were:

(1) Prefer lanes close to the current lane to improve the chances of sharing a useful texel value.
(2) Use the same sharing filter footprint size for all lanes inside a wave to avoid unnecessary thread divergence.
(3) Maximize the variety of shared texel values between lanes.

4.3.1 Square Footprints. Square sharing footprints are a natural choice that ensures that the closest possible lanes are used for sharing. Figure 6 considers 2×2 footprints. The top half shows the only possible footprint placement for quad intrinsics. With quad intrinsics, the four lanes inside each quad end up with the same set of texel values after sharing. Under high magnification, all will compute nearly the same filtered color introducing high spatial correlation.

In the bottom part of the figure, we show one possible layout for 2×2 footprints when using wave intrinsics, which allows the footprints to have more variety. The false-colored pattern in the lower right shows that this method provides smaller regions with the same texel values used at adjacent lanes compared to quad intrinsics. The error is not significantly reduced, but it is less spatially correlated, which is preferable for spatiotemporal denoising.

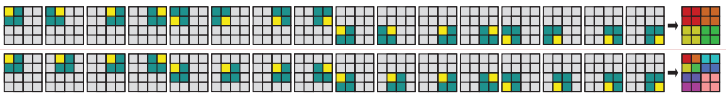

Fig. 6. Examples of 2×2 footprints for waves of size 16, configured as 4×4 lanes. A texel sharing footprint consists of a yellow lane, which is the lane whose filtered value we want to compute, together with a set of texel samples from dark green lanes. **Top**: using quad intrinsics the footprints are restricted by the API to have the upper left coordinates of a quad be even in x and y. All lanes in each quad filter the same texel values. Under high magnification, 2×2 regions produce almost identical color after stochastic filtering, illustrated by the colored pattern at the right. **Bottom**: using wave intrinsics, the 2×2 footprints have more degrees of freedom in that they can be placed anywhere inside the wave. The sharing pattern visualization at the right has fewer regions of similar color, resulting in less spatial correlation.

An illustration of 3×3 and 4×4 footprints can be found in our supplemental material, and 3×3 is also visualized at the top of Figure 8.

4.3.2 Pseudorandom Sparse Footprints. Square footprints are straightforward to implement and exhibit excellent reuse locality, increasing the probability of successful reuse of texels between lanes. However, with larger footprints, they can lead to visible structured square patterns, such as those visible in the left part of Figure 7. The lanes at the edges and corners of the wave filter the same texel values as their neighbors, leading to correlation artifacts that are difficult to remove with spatiotemporal denoisers. When a denoiser averages correlated pixels, the variance is not reduced.

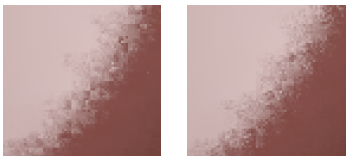

Fig. 7. Single frame crops of *deterministic* (left) and *sparse* (right) sharing footprints with 9 samples. The deterministic pattern yields visible square and edge discontinuities, while using the pseudorandom sparse pattern reduces the blocky appearance and resembles dithering.

To alleviate this issue, we propose an alternative based on pseudorandom, pre-generated sparse sharing footprints. Figure 8 compares a regular 3×3 square footprint to pseudorandom, pre-generated sparse sharing footprints. We balance lane locality (and in turn, locality of the UV coordinates at each lane), where greater locality gives a higher chance of sharing samples, with dithering-like structures that give a result that is easier to resolve with a spatiotemporal filter. Furthermore, we reduce correlations between adjacent lanes by enforcing that none of the lanes are overrepresented in the aggregate of the sharing footprints. The right part of Figure 8 shows the histogram of each lane's usage for both a 3×3 footprint and a sparse footprint. The 3×3 footprint underrepresents the texels sampled by the corner lanes and overrepresents the center ones, while the sparse footprint provides higher uniformity. This uniformity of texel use is beneficial as it avoids repeated similar and correlated texels.

Figure 7 compares the resulting images of the two approaches. Pseudorandom sparse footprints are more noisy, but the noise resembles dithering and is easier to temporally resolve and denoise as we show in Section 5.5. We describe a simple optimization-based method to generate those patterns in Section S5. In our implementation, we generate multiple different patterns that we cycle through over frames to further break up any visual structure.

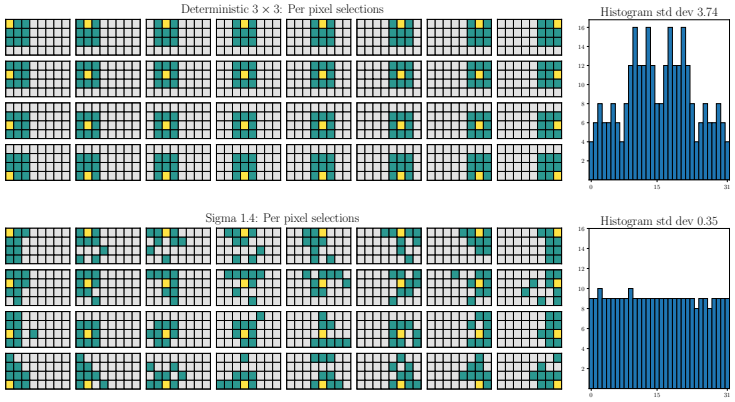

Fig. 8. **Top:** comparison of deterministic 9 sample sharing footprints (3×3 squares) and pseudorandom sparse footprints. The grid diagram (**left**) shows which wave elements samples (green) are reused by the given lane (yellow). The histogram (**right**) shows how many times a texel sample from the given lane is used by all the lanes in the wave. **Bottom:** sparse footprints balance sample reuse locality with uniform texel reuse contributions to break up visible structures.

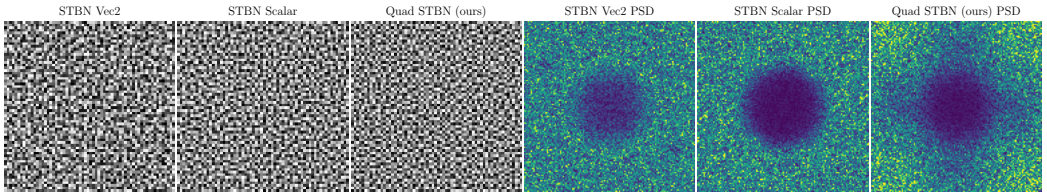

Fig. 9. Comparison of the visual appearance and power spectra of various STBN patterns–Vec2 and scalar versions of the original STBN algorithm [Wolfe et al. 2022] as well as our proposed modification. The Vec2 version has the worst blue noise properties and a significant amount of energy in the low frequencies. We observe in the rightmost power spectral density plot that our modification prioritizes diagonal frequencies and removes horizontal and vertical leftover patterns.

4.4 Blue Noise Mask Improvements

The original STF technique recommended the use of spatiotemporal blue noise masks [Wolfe et al. 2022]. Since our technique relies even more on neighboring sample diversity and because of advancements in blue noise masks [Donnelly et al. 2024], we reevaluate their choice.

We found that the Vec2 version of the STBN mask [Wolfe et al. 2022] yields suboptimal results. This can be explained by the algorithm used to generate those masks—simulated annealing [Wolfe et al. 2022]—rather than void and cluster, yielding inferior blue noise properties. We confirm this in the power spectra in Figure 9. Our first modification is to simply concatenate independent scalar realizations from the void and cluster algorithm.

Further, we adapt the STBN pattern to the unique needs of our algorithm. While STBN masks optimize for blue noise value distribution and anti-correlation of a pixel with all of its neighbors, we are reusing only some of the neighbors for wave sample sharing. For instance, we can never sample across the wave boundaries. In the specific case of 2×2 quad sharing, we always reuse samples between neighbors on a fixed 2×2 grid.

```
// uv is in [0,1]^2 and txDim is the texture resolution.
float texelFloatCoords = uv * txDim - float2(0.5f);
// (...) One-tap STF sampling with access of one texel.
// texelFloatCoords are the original floating-point coordinates for filtering.
// sampled_uv are the integer coordinates of the STF-fetched texel.
// T is the fetched/decompressed single texel.
float p_sampled_uv = GetFilterPMF(texelFloatCoords, sampled_uv);
float4 T = texture[sampled_uv];

float4 sum_w_i_T_i = 0.0f;
float sum_w_i = 0.0f;
for (uint i = 0; i < FOOTPRINT_SIZE; ++i) {
  uint laneIdx = waveLaneSet[currentLaneIdx][i];
  int2 uv_i = WaveReadLaneAt(sampled_uv, laneIdx);
  float p_i = WaveReadLaneAt(p_sampled_uv, laneIdx);
  float4 T_i = WaveReadLaneAt(T, laneIdx);
  float p_c = GetFilterPMF(texelFloatCoords, uv_i);
  float w_i = p_c / p_i;
  sum_w_i_T_i += w_i * T_i;
  sum_w_i += w_i;
}
return sum_w_i_T_i / sum_w_i;
```

Fig. 10. Implementation of the core texel sharing algorithm with filtering using Equation 10.

Our proposed modification changes the void and cluster energy splatting function to splat twice as much energy to the neighbors in each fixed 2×2 quad. We visualize the resulting pattern and its power spectral density in Figure 9. This modification removes some of the horizontal and vertical patterns and spectral components and increases the noise energy in the spectrum diagonals. We evaluate the impact of those modifications in Section 5.3.

4.5 Implementation Details

Here we combine all introduced components of our algorithm in a GPU-friendly manner. We focus on the more general and flexible wave intrinsics; see Figure 10

Our method starts with each lane sampling a single texel according to its filter, following the standard STF approach. Each lane then iterates over the lanes of its associated footprint. In each loop iteration, we use `WaveReadLaneAt()` to fetch the information about texel samples from other lanes in the wave. In particular, we obtain texel values and their associated integer coordinates and the PMF value for sampling them. The integer texel coordinates are needed to compute the current lane's filter sampling PMF. The estimator introduced in Section 4.2 is then used to compute the weight of each sample. We accumulate the weighted texels as well as the weights, and divide the accumulated texels by the accumulated weights. This can be expressed as follows. In the code above, the texel sharing footprint is encoded as a lookup table in the `waveLaneSet` array. If deterministic sharing footprints are used, the corresponding offsets may be computed at runtime. The number of loop iterations (`FOOTPRINT_SIZE`) is based on the size of the sharing footprints—4, 9, or 16 in our experiments. An example implementation of the `GetFilterPMF` function is provided in the supplement (Section S6).

Our approach guarantees that the weight for the current lane is always equal to one due to the equality of the PMFs. Conversely, if a texture sample of a neighbor falls outside of the original lane's filtering footprint, the weight is zero, preventing incorrect contributions. As a consequence, our method is robust when no wave sharing is possible: if the magnification factor is insufficient or if wave neighbors sample texels that make no contribution to the current lane, our algorithm returns exactly the same result as one-tap STF, since each lane is always included in its own footprint.

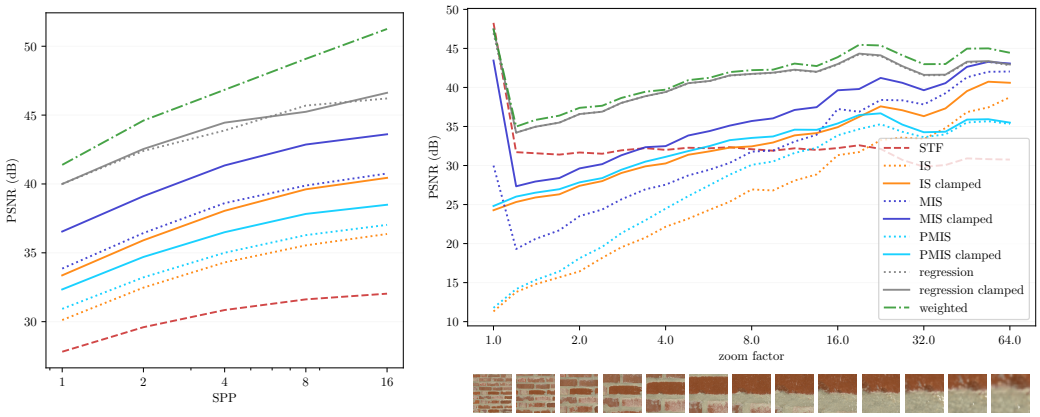

Fig. 11. **Left**: PSNR as a function of the number of samples per pixel (SPP) for the scene in Figure 12. The setup is high magnification, bilinear filtering, diffuse and specular normal mapping, and 3×3 sharing footprints. The estimators we evalute include classic one-tap STF, clamped and non-clamped versions of standard importance sampling (IS), multiple importance sampling (MIS), pairwise MIS (PMIS), the regression estimator, and our new weighted estimator. Our weighted estimator provides the best results. **Right**: PNSR as a function of zoom factor with rendered images below. Note that a zoom factor of 1.0 gives a 1–1 mapping between texels and pixels, and a factor of 64 means a texel maps to 64×64 pixels, for example.

In our implementation, we repeat this loop for each sampled material texture for simplicity. We note however, that in practice when multiple textures share the same UV coordinates and resolution, the loop could be executed only once and the PMF and weight computations could be shared to minimize the amount of shader code and to reduce the arithmetic instruction overhead.

5 Results

We have implemented our technique in Falcor [Kallweit et al. 2022]. Rendering starts with Falcor's `GBufferRaster`-pass, which outputs only depth and UVs, similar to modern visibility buffer approaches [Haar and Aaltonen 2015]. The resulting buffer is fed to a custom compute shader pass, which performs texture lookups and texel sharing before filtering texels and computing shading. All results were rendered using an NVIDIA RTX 4090 GPU and all used Falcor other than the performance measurements with neural texture compression in Section 5.6.

5.1 Comparison of Estimators

We have evaluated the error in rendered images for a number of candidate estimators (Section 4.1 and 4.2); the results are summarized in Figure 11. The baseline is classic one-tap STF. We have included clamped and non-clamped versions of standard importance sampling (IS), multiple importance sampling (MIS), pairwise MIS (PMIS), and the regression estimator. Those are compared against our novel weighted estimator (Equation 10). The left part of Figure 11 shows PSNR (computed in linear RGB space) as a function of samples per pixel (SPP). Figure 12 shows visual results from these estimators for the left part of Figure 11. Our weighted estimator consistently has both lowest numeric error and the best visual appearance with respect to the reference.

We have also evaluated how the degree of texture magnification affects error for the various estimators; results are shown on the right side of Figure 11. At zoom factors of one and slightly greater, there is little texture-space overlap of each pixel's filter. This leads to the problematic case for importance sampling discussed in Section 2.2 where one pixel may sample a texel with low

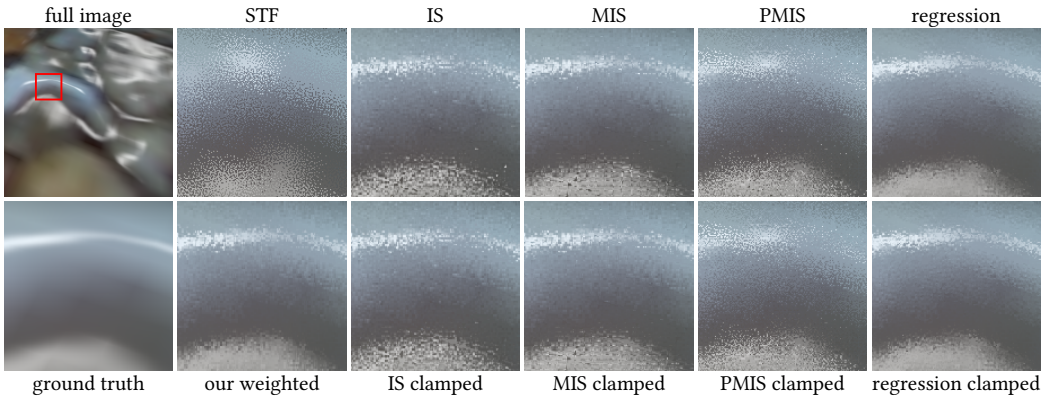

Fig. 12. Visual results from the estimators with bilinear filtering. See the caption in Figure 11 for an explanation of the abbreviations. STF produces a poor highlight, IS and MIS are substantially more noisy than our weighted estimator, and PMIS misses part of the highlight. The regression estimator works well but has higher runtime cost than ours and only guarantees results within the range of the texel values if clamping is used.

Table 1. Quality measures, PSNR (in dB, where higher is better) and FLIP (lower is better), for different filter footprint configurations at 1 SPP. '*' means *exact filtering* (Section 4.2.1) is being used; 'q' means quad intrinsics are used; and 'w' means wave intrinsics are used. All variants of our method have lower error than one-tap STF, with error decreasing more with larger sharing footprints and when exact filtering is used.

	STF	2 × 2q	2 × 2q*	2 × 2w	2 × 2w*	3 × 3	3 × 3*	PS 9×	PS 9×*	4 × 4	4 × 4*
PSNR (↑)	27.82	36.76	36.89	34.94	35.09	40.14	41.60	38.07	39.47	42.29	44.87
FLIP (↓)	0.0480	0.0311	0.0309	0.0342	0.0338	0.0258	0.0214	0.0268	0.0233	0.0243	0.0157

probability but then share it with a pixel where that texel has a high filter weight; as a result, that estimator has high variance. In contrast, methods like our weighted estimator essentially revert to one-tap STF's performance in that case, which is all that can be expected when sharing is rarely successful. As the zoom factor increases, the mismatch of PMFs between nearby pixels decreases and all methods give lower error as sharing is successful more frequently. This provides an empirical demonstration of our main insight—with increasing magnification factors, adjacent pixels tend to share the same texel values and draw them from the same distributions, dramatically improving the reuse probability. Throughout this range, our estimator has the lowest error and the clamped regression estimator is a close second.

In practice our estimator has lower computational cost than the regression estimator as it requires fewer arithmetic operations and only a single loop over the samples rather than three. Therefore, we will focus on our weighted estimator alone for the remainder of this section.

5.2 Evaluation of Sharing Footprints

Table 1 shows the effect of varying the texel sharing footprint with a bilinear filter. It includes square texel sharing footprints, 2×2 quad and wave intrinsics, 3×3, 4×4, as well as a pseudorandom sparse footprint with 9 texels (PS 9×) of sharing. We also present results for *exact filtering* (EF), described in Section 4.2.1. Our optimized quad STBN masks were used for all results except 3 × 3, which uses FAST EMA product (see Section 5.3).

As expected, larger footprints tend to give lower error. For 2 × 2, there are rarely enough samples for EF to be applicable. However, for larger footprints (≥ 9), EF reduces the error. The pseudorandom

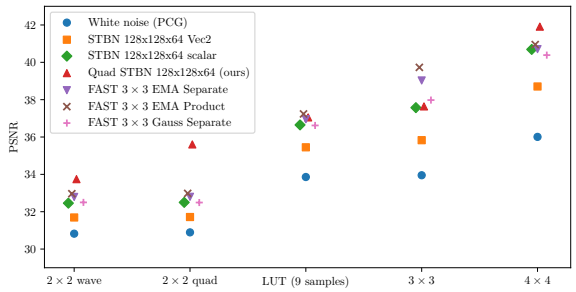

Fig. 13. Impact of the pseudorandom number source on the PSNR score of our algorithm (non-denoised) for different footprints when used for bilinear filtering.

sparse footprint with size 9 (PS 9×), is not quite as good as a 3×3 footprint, which is expected since it spreads out the filter footprint compared to the square 3×3. However, this pattern works better when spatiotemporal denoising is used (Section 5.5).

5.3 Blue Noise Mask Evaluation

We have measured the effect of our modifications to the STBN pattern proposed in Section 4.4 to error in images and compared to FAST noise [Donnelly et al. 2024]. We present the quantitative results in Figure 13. The improvement from our modified STBN pattern is dramatic for the quad sharing variants and it also provides a marginally better PSNR than the alternatives for the 4×4 footprint. FAST noise significantly improves the results for the 3×3 footprint, as it was optimized for a 3×3 blurring kernel. Therefore, we recommend the 3×3 FAST noise for the 3×3 footprint, while for all the other footprints we suggest using our modified STBN pattern.

5.4 Bicubic B-Spline Filtering

Our analysis so far has focused on the bilinear filter due to its ubiquity in real-time rendering and to facilitate the adoption of STF as a drop-in replacement for the traditional hardware magnification filter. However, one of the benefits of STF is that higher-quality filters like bicubic or Gaussian filters may be used at no additional texture sampling cost.

We evaluate our method with a bicubic B-spline filter. This filter is commonly used in offline rendering applications due to its pleasant appearance and removing some diagonal aliasing of the bilinear filter. Figure 14 shows results with different sharing pattern configurations, without and with DLSS spatiotemporal denoising. In this case, one-tap STF fails to reproduce the shape of the specular highlight. Our method, even with the 2×2 square sharing footprint, significantly reduces the error and better preserves the highlight's appearance. Increasing the footprint size further reduces the error, but turns noise into visible structured and square or streak artifacts. We analyze this effect in the following section.

5.5 Spatiotemporal Denoising

Our technique can have non-obvious interactions with spatiotemporal denoisers. On the one hand, it significantly reduces the variance of individual pixels, making denoising easier. Conversely, it introduces correlation between adjacent pixels due to texture sample reuse. Denoisers typically assume uncorrelated (or inversely correlated, in the case of blue noise sampling) signals and struggle with distinguishing between image structure and correlated noise. Numerous temporal filtering techniques are used in production game engines [Yang et al. 2020], so for our evaluation we focus

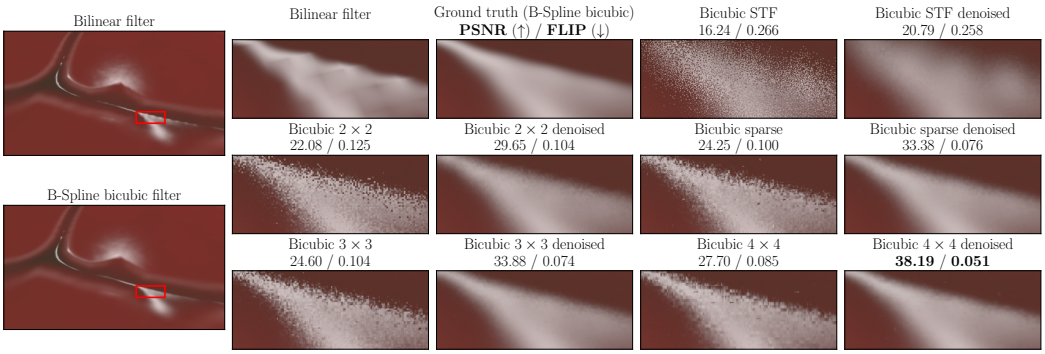

Fig. 14. Use of our method with the B-spline bicubic filter, both for single frames and denoised with DLSS. We provide the bilinear-filtered image for comparison. The B-spline bicubic filter removes unpleasant diagonal specular aliasing. The original STF method fails to reproduce sharp specular peaks. With increasing sharing footprint size the error decreases, especially on specular peaks. However, with large sharing footprints, starting with the 3×3 square, visible square or streak structures start to appear.

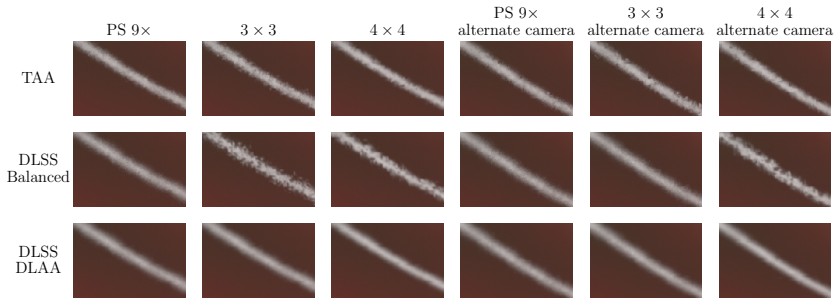

Fig. 15. Results of **DLSS DLAA**, **DLSS Balanced**, and **TAA** with different footprints. TAA is consistently the most noisy but shows acceptable results with sparse footprints (PS 9×). DLSS Balanced struggles with the 3×3 footprint at some camera angles and 4×4 at all tested camera angles. DLAA consistently denoises all footprint configurations well and closely matches reference images. Across the compared denoisers, the sparse footprint shows consistently good results at all tested camera angles.

on two ends of the complexity spectrum—a basic implementation of TAA in Falcor inspired by the Unreal Engine 4 TAA [Karis 2014], as well as a modern machine-learning based approach of DLSS [Liu 2022]. DLSS comes with different quality profiles, from pure spatiotemporal denoising and antialiasing (called *DLAA*), to temporal frame upsampling. Those additional profiles are commonly used to improve the frame rate at the cost of some visual quality. To make our evaluation complete, we analyze behavior of our method combined with both DLAA and DLSS (with the Balanced profile, rendering at 58% resolution). While we describe our observations and include visual examples, we encourage the reader to review our video in the supplementary material to see these effects in motion. Our video shows results using both TAA and the DLAA version of DLSS.

Our main finding is that with any sample reuse footprint, DLSS in the DLAA profile is able to remove the residual noise effectively and with a good temporal stability. On the other hand, the Balanced profile can struggle with larger regular footprints, such as 3×3 or 4×4. While the 4×4 footprint is consistently challenging, the 3×3 footprint shows different behaviors depending on

the camera position—both in static and the dynamic scenes. Addressing those artifacts was our main motivation for pseudorandom sparse footprints (Section 4.3.2); we show their efficiency in Figure 15. While sparse footprints are marginally noisier than the 3×3 footprint, they do not show view-dependent inconsistency.

By comparison, TAA tends to consistently retain small amounts of residual noise of the bilinear filter with all footprints and camera trajectories. Sparse footprints tend to help its noise reduction, but on average, it tends to be much noisier than DLSS. With the bicubic filter, TAA produces severe banding-like visual artifacts caused by the local color bounding box clamping [Karis 2014].

5.6 Performance

To test the computational performance of our method in one of the targeted use cases, we measured frame time in a stand-alone renderer supporting neural texture decompression [Vaidyanathan et al. 2023]. The rendering resolution was 3840×2160 and we zoomed in on a piece of geometry so that every pixel had magnified textures. The material consisted of 9 texture channels split between three textures (base color, normal map, material parameters). The baseline is one-tap STF. Our method with a 2×2 footprint took just 0.04 ms longer than the baseline, while 3×3 and 4×4 took 0.11 ms and 0.14 ms longer, respectively. As a reference, we also attempted to decode 2×2 texels needed for a classic full bilinear filter. A full bilinear filter took more than 5× the cost of a single sample, instead of expected 4× linear scaling, most likely due to high register pressure of NTC decompression.

6 Conclusions and Future Work

We have presented a significant improvement to STF under magnification based on sharing texels among nearby pixels, allowing a more accurate estimate of the filtered value. The additional cost of our method is small and limited to arithmetic and wave register swizzling instructions, i.e., there is no additional texture sampling cost or memory traffic. Our method not only reduces stochastic noise, making post-rendering denoisers more effective, but it also reduces the visual difference between filtering after shading and filtering before shading. These properties make STF more attractive for existing game engines, and thus ease the adoption of novel compression algorithms like NTC [Vaidyanathan et al. 2023] where the number of texels accessed directly affects performance.

Under magnification, our approach produces results closer to filtering before shading than filtering after shading. While this is beneficial for many applications, Pharr et al. [Pharr et al. 2024] discuss cases where filtering after shading is nevertheless preferred. Our technique can be used selectively and not applied to those specific textures, though it would be worthwhile in future work to investigate if we can get other benefits of our approach, such as noise and variance reduction, by applying some of our insights to already-shaded pixel values under magnification.

Developing and evaluating our algorithm, we found that a custom blue noise pattern can improve the quality dramatically, up to 6dB PSNR difference. This result and insight encourages joint research of novel custom blue noise masks together with other low-sample rendering techniques.

With our approach, wave lanes independently sample texels; within a sharing footprint, this may lead to some texels being sampled repeatedly while other useful ones are not sampled at all. It would be useful to find an efficient way to coordinate pixel texture samples to avoid such cases, though we note it might be challenging since each lane has a unique set of other lanes that it draws texels from. However, the benefits of a solution to this problem could go beyond improved image quality—especially under high magnification, it may be possible to sample less than one texel per lane on average, thus improving performance of multi-texture sampling.

Acknowledgments

Many thanks to Pontus Ebelin for help with image & video metrics and reviewing of the paper. Thanks also to Benedikt Bitterli for reading multiple drafts of the paper and offering many helpful suggestions, especially related to how we presented the connection between discrete and continuous MC estimators. We would also like to thank Marco Salvi for thoughtful comments and feedback as well as for suggesting the randomization of sharing footprints and Chris Wyman for proof-reading and suggestions on improving the structure of the manuscript. We had many fruitful discussions during the development of this work Peter Morley, Johannes Deligiannis, Alexey Panteleev, Mike Songy, Nathan Hoobler, and Homam Bahnassi, including a suggestion to use larger wave sharing footprints and their evaluation of the covered techniques in existing production renderers and on real-world game scenes. We also thank Aaron Lefohn for continuous support of this work. The Bricks 090 texture set was retrieved from https://ambientcg.com/ and the gravel texture set from https://kaimoisch.com/free-textures/.

References

Brian D. O. Anderson and John B. Moore. 1979. *Optimal Filtering*. Prentice-Hall.

Philippe Bekaert, Mateu Sbert, and Yves D Willems. 2000. Weighted Importance Sampling Techniques for Monte Carlo radiosity. In *Eurographics Workshop on Rendering*. Springer, 35–46.

Benedikt Bitterli. 2022. Correlations and Reuse for Fast and Accurate Physically Based Light Transport.

Benedikt Bitterli, Chris Wyman, Matt Pharr, Peter Shirley, Aaron Lefohn, and Wojciech Jarosz. 2020. Spatiotemporal Reservoir Resampling for Real-time Ray Tracing with Dynamic Direct Lighting. *ACM Transactions on Graphics (SIGGRAPH)* 39, 4 (July 2020), 148:1–17. https://doi.org/10.1145/3386569.3392481

Jack Dongarra. 2022. A Not So Simple Matter of Software. Turing Award Keynote. https://www.hpcwire.com/2022/11/16/jack-dongarra-a-not-so-simple-matter-of-software/.

William Donnelly, Alan Wolfe, Judith Bütepage, and Jon Valdés. 2024. Filter-Adapted Spatiotemporal Sampling for Real-Time Rendering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 1 (2024), 13:1–16.

Iliyan Georgiev and Marcos Fajardo. 2016. Blue-Noise Dithered Sampling. In *ACM SIGGRAPH 2016 Talks*. 1–1.

Ulrich Haar and Sebastian Aaltonen. 2015. GPU-driven Rendering Pipelines. *SIGGRAPH Advances in Real-Time Rendering in Games course* (2015).

David Christopher Handscomb. 1964. Remarks on a Monte Carlo Integration Method. *Numer. Math.* 6, 1 (1964), 261–268. https://doi.org/10.1007/BF01386074

Tim Hesterberg. 1995. Weighted Average Importance Sampling and Defensive Mixture Distributions. *Technometrics* 37 (1995), 185–194. https://api.semanticscholar.org/CorpusID:122839484

Nikolai Hofmann, Jon Hasselgren, Petrik Clarberg, and Jacob Munkberg. 2021. Interactive Path Tracing and Reconstruction of Sparse Volumes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 4, 1 (April 2021), 5:1–19. https://doi.org/10.1145/3451256

Simon Kallweit, Petrik Clarberg, Craig Kolb, Tomáš Davidovič, Kai-Hwa Yao, Theresa Foley, Yong He, Lifan Wu, Lucy Chen, Tomas Akenine-Möller, Chris Wyman, Cyril Crassin, and Nir Benty. 2022. The Falcor Rendering Framework. BSD-Licensed Github Repository.

Brian Karis. 2014. High-quality Temporal Supersampling. *Advances in Real-Time Rendering in Games, SIGGRAPH Courses* 1, 10.1145 (2014), 2614028–2615455.

Doyub Kim, Minjae Lee, and Ken Museth. 2024. NeuralVDB: High-resolution Sparse Volume Representation using Hierarchical Neural Networks. *ACM Transactions on Graphics* 43, 2 (2024), 20:1–21. https://doi.org/10.1145/3641817

Edward Liu. 2022. DLSS 2.0 - Image Reconstruction for Real-Time Rendering with Deep learning. In *Game Developers Conference*.

Morgan McGuire, Michael Mara, and David Luebke. 2012. Scalable Ambient Obscurance. In *High Performance Graphics*. 97–103.

Microsoft. 2021. HLSL Shader Model 6.0. https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/hlsl-shader-model-6-0-features-for-direct3d-12. [Online; accessed 2024-09-11].

Zackary Misso, Benedikt Bitterli, Iliyan Georgiev, and Wojciech Jarosz. 2022. Unbiased and Consistent Rendering using Biased Estimators. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 41, 4 (July 2022). https://doi.org/10/gqjn66

Art B. Owen. 2013. *Monte Carlo Theory, Methods and Examples*. https://artowen.su.domains/mc/.

Eric Penner. 2011. Shader Amortization Using Pixel Quad Message Passing. In *GPU Pro 2*. CRC Press, 349–366.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically Based Rendering: From Theory to Implementation* (4th ed.). The MIT Press.

Matt Pharr, Bartlomiej Wronski, Marco Salvi, and Marcos Fajardo. 2024. Filtering After Shading with Stochastic Texture Filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 7, 1 (2024), 14:1–20.

Michael JD. Powell and J. Swann. 1966. Weighted Uniform Sampling—a Monte Carlo Technique for Reducing Variance. *IMA Journal of Applied Mathematics* 2, 3 (1966), 228–236.

Erik Reinhard, Peter Shirley, and Tom Troscianko. 2001. Natural Image Statistics for Computer Graphics. *Univ. Utah Tech Report UUCS-01-002* (March 2001).

Gerald L Smith, Stanley F Schmidt, and Leonard A McGee. 1962. *Application of Statistical Filter Theory to the Optimal Estimation of Position and Velocity on Board a Circumlunar Vehicle*. Vol. 135. National Aeronautics and Space Administration.

Jerome Spanier. 1979. A New Family of Estimators for Random Walk Problems. *IMA Journal of Applied Mathematics* 23, 1 (1979), 1–31.

Laszlo Szirmay-Kalos and Laszl Szecsi. 2003. Improved Indirect Photon Mapping with Weighted Importance Sampling. In *Eurographics 2003 - Short Presentations*. https://doi.org//10.2312/egs.20031068

Justin Talbot, David Cline, and Parris Egbert. 2005. Importance Resampling for Global Illumination. In *Eurographics Symposium on Rendering*, Kavita Bala and Philip Dutre (Eds.). https://doi.org//10.2312/EGWR/EGSR05/139-146

Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn. 2023. Random-Access Neural Compression of Material Textures. *ACM Transactions on Graphics* 42, 4 (2023), 88:1–25. https://doi.org/10/gsk4fz

Eric Veach and Leonidas J. Guibas. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of SIGGRAPH*. 419–428. https://doi.org/10.1145/218380.218498

Alan Wolfe, Nathan Morrical, Tomas Akenine-Möller, and Ravi Ramamoorthi. 2022. Spatiotemporal Blue Noise Masks. In *Eurographics Symposium on Rendering*. 117–126.

Lei Yang, Shiqiu Liu, and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum* 39, 2 (2020), 607–621.

Supplemental Material for Improved Stochastic Texture Filtering Through Sample Reuse

This supplemental material includes further discussion of Monte Carlo estimates with stochastic texture filtering (STF), a variance analysis of filtering after shading, further examples of texel sharing footprints as well as a description of our optimization algorithm for generating sparse footprints, an example implementation of the `GetFilterPMF` function used in the example implementation in Section 4.5, and some results and discussion about the use of our sample sharing techniques with volumetric ray marching.

S1 STF and Monte Carlo Integral Estimates

Pharr et al.'s paper on stochastic texture filtering did not make a direct connection between integral Monte Carlo estimators and the stochastic texture filtering algorithms introduced there but rather derived STF algorithms by framing them as stochastic evaluation of sums of weighted texel values [Pharr et al. 2024]. Because our weighted STF estimator, Equation 10, is an integral estimator, here we make the straightforward connection between integral estimators and stochastic texture filtering for completeness.

With traditional STF, we are applying Monte Carlo integration to the integral texture filtering equation, 2:

$$t_{\mathrm{r}}(u,v) = \iint \left(\sum_i^n \delta(u'-u_i)\delta(v'-v_i)\,\mathbf{T}_{u_i,v_i} \right) f_{\mathrm{r}}(u-u', v-v')\,\mathrm{d}u'\,\mathrm{d}v' \tag{S1}$$

$$= \iint \sum_i^n \delta(u'-u_i)\delta(v'-v_i)\,\mathbf{T}_{u_i,v_i}\, f_{\mathrm{r}}(u_i-u', v_i-v')\,\mathrm{d}u'\,\mathrm{d}v'. \tag{S2}$$

We will define the PDF

$$p(u,v) = \sum_i^n \delta(u-u_i)\delta(v-v_i)\, f_{\mathrm{r}}(u_i-u, v_i-v). \tag{S3}$$

Under the assumption that the texture reconstruction filter f_{r} is normalized, it is easy to see that this is a valid PDF.

Samples from the PDF can be taken by selecting a term i of the sum with probability proportional to $f_{\mathrm{r}}(u_i-u, v_i-v)$. In turn, we have a sample point (u_i, v_i).

If we apply the importance sampling Monte Carlo estimator, we have

$$t_{\mathrm{r}}(u,v) \approx \frac{\cancel{\delta(u-u_i)}\,\cancel{\delta(v-v_i)}\,\mathbf{T}_{u_i,v_i}\,\cancel{f_{\mathrm{r}}(u_i-u, v_i-v)}}{\cancel{\delta(u-u_i)}\,\cancel{\delta(v-v_i)}\,\cancel{f_{\mathrm{r}}(u_i-u, v_i-v)}} = \mathbf{T}_{u_i,v_i}, \tag{S4}$$

giving the one-tap STF estimator.

S2 Variance and Bias with Filtering After Shading

For cases where filtering before shading is the desired result, it is useful to be able to characterize the error from STF and filtering after shading in order to evaluate and design STF estimators. This is challenging in general, as a wide variety of nonlinearities are present in shading functions. We therefore propose a simple approach based on statistical analysis of nonlinear transformations of random variables. For a more complete treatment of the statistical analysis of error resulting from nonlinearities applied to random variables, we refer the reader to the statistical and control theory literature and extended Kalman filters [Anderson and Moore 1979; Smith et al. 1962] as well as recent advances in unbiasing rendering algorithms using telescoping Taylor series [Misso et al. 2022].

Consider a shading function f where the true filtered texture value is μ: with filtering before shading, we filter the texture using Equation 3 to compute μ and then return $f(\mu)$. With filtering after shading and one-tap STF [Pharr et al. 2024], the texture is represented by a random variable X corresponding to a single texel that is sampled according to the texture filter, $X \sim f_r$. The estimator is $f(X)$. We would like to understand how the expected value of filtering after shading, $\mathbb{E}[f(X)]$, relates to $f(\mu)$.

To approximate this error, we can use the Taylor expansion of f around μ. For example, consider the second-order expansion:[5]

$$\mathbb{E}[f(X)] = \mathbb{E}[f(\mu + (X - \mu))] \tag{S5}$$

$$\approx \mathbb{E}\left[f(\mu) + f'(\mu)(X - \mu) + \frac{1}{2}f''(\mu)(X - \mu)^2\right] \tag{S6}$$

$$= f(\mu) + f'(\mu)\,\mathbb{E}[X - \mu] + \frac{1}{2}f''(\mu)\,\mathbb{E}\left[(X - \mu)^2\right]. \tag{S7}$$

Because one-tap STF gives an unbiased estimate of μ, $\mathbb{E}[X - \mu] = 0$ and $\mathbb{E}\left[(X - \mu)^2\right]$ is X's variance, which we will denote by σ_X. Dropping $f(\mu)$, the result of filtering before shading, we are left with

$$\frac{f''(\mu)}{2}\sigma_X^2 \tag{S8}$$

as the error due to filtering after shading. In other words, the error depends on the second derivative of the shading function and the squared variance of X (and the higher-order terms we have neglected); we see how the variance of X directly contributes to the final error and the resulting bias.

This analysis fits with our earlier error analysis in Section 3: when the variation in the filtered texel values is small, STF yields a small error and filtering after shading gives results that are close to filtering before shading. With constant signals with zero variance, the error is zero and the two approaches are equivalent.

Unlike one-tap STF, our method uses an estimator that has a small bias (Section 4.2). Thus, $\mathbb{E}[X - \mu] \neq 0$ and $\mathbb{E}\left[(X - \mu)^2\right]$ includes both bias and variance. The error is approximated as:

$$f'(\mu)\,\mathbb{E}[X - \mu] + \frac{1}{2}f''(\mu)\,\mathbb{E}\left[(X - \mu)^2\right]. \tag{S9}$$

The error includes terms that depend on both the first and second derivatives of the shading function f, though in practice the overall error is lower than with one-tap STF since X is closer to μ with our approach.

From these results, we can see why the requirement of not introducing any variation in regions of constant texture values is so important—even a small amount of error may introduce a large error in the shaded result. Furthermore, estimators like standard importance sampling that may have unbounded weights (recall Section 4.1) result in not only much higher variance, but also high further statistical error moments.

While Taylor expansion formally does not apply to every function used in rendering (for example, a step comparison operator used in shadow mapping is not differentiable), this analysis still provides an insight and intuition for evaluating different estimators: that the better the variance reduction of an estimator (or, generalizing to higher-order moments, smaller amounts of noise and tighter distributions), the closer the result is to filtering before shading.

[5]In practice, many functions used in rendering, such as specular shading, are nonlinear and have slowly decaying higher-order derivatives, so a second order expansion is insufficient. However, we can apply this expansion to multi-variate functions and include higher-order derivatives and thus, higher-order statistical moments.

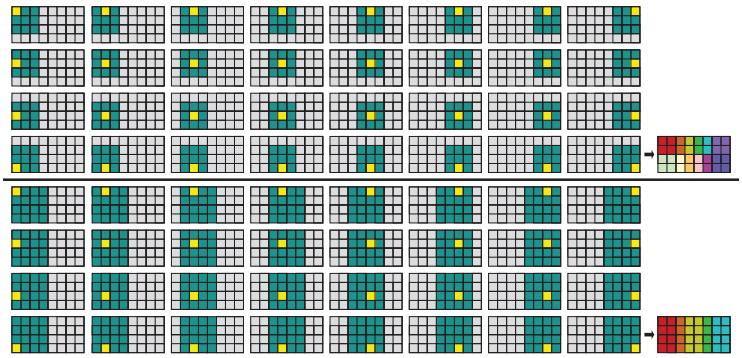

Fig. S1. A set of possible 3×3 footprints (top) and 4×4 footprints (bottom). As in Figure 6, for each footprint, yellow signifies the lane in the wave that the footprint is associated with and green signifies the other lanes that it draws texel values from. The colored illustrations on the right shows which groups of lanes all end up with the same texel values to filter after sharing.

S3 Illustration of Larger Footprints

Figure S1 illustrates the footprint for deterministic 3×3 and 4×4 square footprints in a 32-lane wave. Note that as the footprints become larger, more lanes in the wave all filter the same set of texel values; these texel sharing sets are illustrated in the lower right. However, since the filters are larger, we find that the error is also lower in general, as discussed in Section 5.2.

S4 Wave Intrinsics in Other Shader Stages

While the specific wave mapping for other shading stages, such as ray tracing shaders, is undefined, we have successfully tested our method with general wave intrinsics in those stages. We have verified that our method works with shadow rays that sample alpha masks from textures and is able to achieve some visual quality improvements over the original STF technique. However, we note that lanes can map to arbitrary and distant rays and thus the resulting filtering quality is not guaranteed. We also advise additional caution and checking whether other lanes are active, as behavior of `WaveReadLaneAt(value, laneId)` is undefined for inactive lanes.

S5 Pseudorandom Sparse Footprint Generation Algorithm

To generate the sparse wave sharing footprints introduced in Section 4.3.2, we use a simple three-stage global optimization algorithm.

In the first stage, for each lane in a wave, we generate a set of 16–32 candidate configurations. Candidate elements to share are generated by sampling from a normal distribution with a fixed standard deviation σ. Close to the wave borders and corners, we relax the σ to allow considering farther-away candidates. We continue taking samples, discarding repeated lanes, until the number of candidates equals the desired wave sharing element count. Increasing values of σ reduce the footprints' locality but make it easier to have each lane be used for sharing the same number of times and have irregular shapes that do not lead to visible structure in images (Figure 7). Figure S2 shows results for three different σ values; the bottom part of Figure 8 was generated with a σ of 1.4.

Having generated candidates for each pixel, we proceed to the second stage of the algorithm. We randomly select a candidate for each pixel, count how many times each wave lane has been selected for sharing, and then find the standard deviation of these counts. We use the standard deviation as a score, where lower standard deviations are better, corresponding to greater uniformity in how

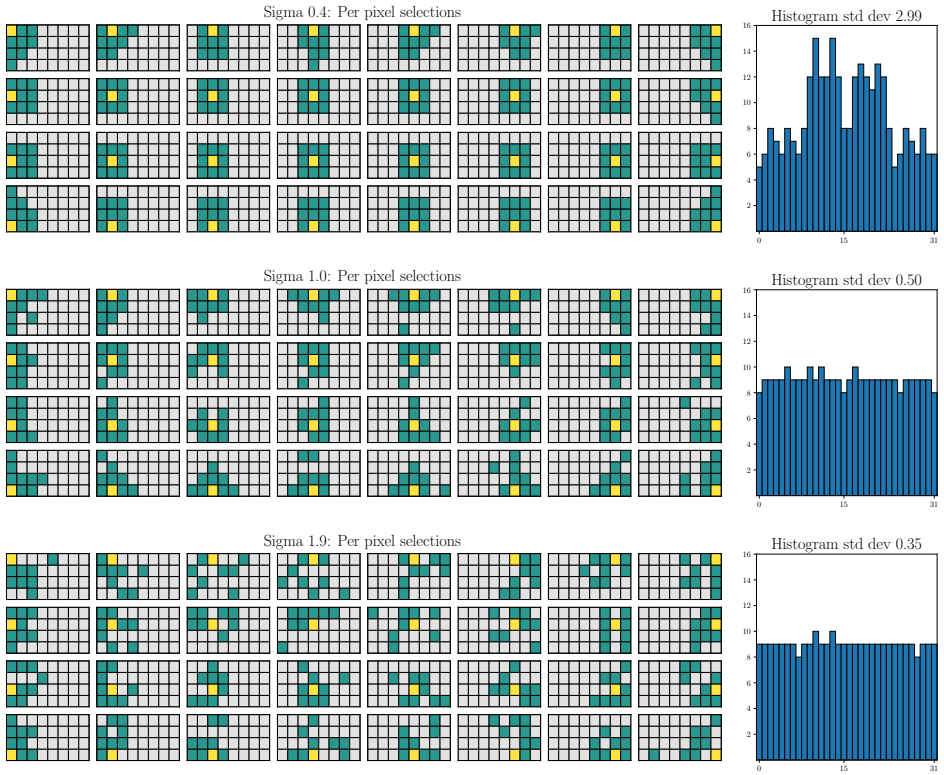

Fig. S2. Three sets of sample sharing footprints, with increasing values of the σ parameter. **Top**: with a small σ, most of the patterns are the same as the deterministic 3×3 footprint; as shown by the histogram on the right, some lanes (the ones in the center of the wave) are used for sharing much more than others (the ones along the edges and the corners.) **Middle**: increasing the σ gives more irregular patterns and a more uniform histogram. **Bottom**: a further increase of σ gives a histogram with slightly lower standard deviation but with irregular patterns that may sample far-away lanes. In general, the farther away the lanes used for sharing are, the less effective reuse will be, since shared texels may be outside of the current lane's texture filter footprint.

often each lane is used. We repeat this process 10,000 times and select the configuration with the best score.

In the third stage, we proceed with a variation of the *coordinate descent* algorithm, where we attempt to improve the configuration selected after the second stage. We exhaustively iterate through all the lanes and check if using any of the other candidates from the first stage for the lane would improve the configuration's overall score. We continue this process until no better candidate is found for any of the lanes.

Finally, because this approach does not guarantee convergence to a global minimum, we repeat it from scratch 30 times and retain the pattern with the best score.

Although our optimization algorithm is brute-force, the search space of waves of 32–64 elements is relatively small and our algorithm still runs quickly. Our naive Python implementation running on a single CPU core can generate a complete set of sharing footprints in less than 40 seconds of CPU time. This has allowed for quick iterations and experiments as well as generating different patterns for different frames to break up temporal artifacts. For the results in the paper, we run ten

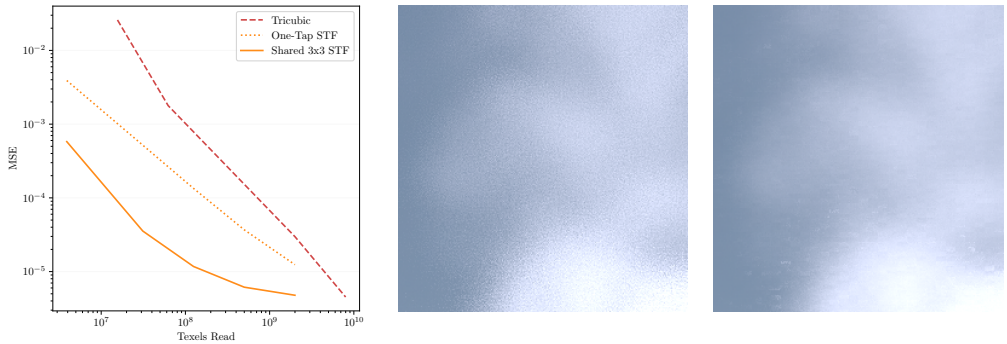

Fig. S3. Sample reuse for volumetric ray marching of a cloud data set. **Left**: Plot comparing mean squared error for full tricubic filtering, one-tap STF, and STF with 3×3 sparse sharing footprints. Sharing significantly reduces MSE, roughly in proportion to the number of successfully shared texel lookups. **Middle**: with 64 ray marching steps and one-tap STF, error manifests as high-frequency noise. **Right**: with 64 steps with texel sharing, numeric error is much lower and noise is reduced, but block artifacts appear.

times as many iterations of the first two stages, with a corresponding increase in pattern generation time by a factor of ~10. We find this time to be acceptable for a preprocess; if higher performance was necessary, the iterations of the first two stages could run in parallel and a higher-performance language like C++ could be used for the implementation. We include our implementation in the supplementary material.

S6 Filter PMF Implementation

Each time through the loop in the code listing in Section 4.5, we consider a texel sampled by one of the lanes in the sharing footprint for the current lane. To evaluate its contribution to the weighted importance sampling estimator, Equation 10, we need to compute the probability that the current lane would have sampled that texel; this is handled by the call to `GetFilterPMF`.

Below is an example implementation of this function for a bilinear filter. Because the filter is normalized, the PMF for a given texel is simply its bilinear filter weight, which is easily computed in a few lines of code. We note that depending on the rendering technique used, this function might need to aditionally verify if other lanes sample from the same texture set and return a zero PMF on any mismatch.

```
float GetFilterPMF(float2 texelFloatCoords, int2 texelIntCoords)
{
    float2 texelDistance = texelFloatCoords - texelIntCoords;
    float2 filterPdf = clamp(1 - abs(texelDistance), 0, 1);
    return filterPdf.x * filterPdf.y;
}
```

S7 Tricubic Reconstruction for Volumetric Rendering

We have also investigated the effect of sample sharing for STF with ray marched volumetric rendering; our results are summarized in Figure S3. As shown in the plot, texel sharing significantly reduces the numeric error compared to one-tap STF for a given number of texel lookups. We also note that one-tap STF has significantly lower error than full tricubic filtering given an equal number of samples; we attribute this to the benefit of importance sampling—full tricubic filtering accesses all the 64 texels under the filter, many of which make a relatively small contribution to the final result.

However, as shown by the images, texel sharing leads to block-structured artifacts in the image with sharing and 64 ray marching steps. We believe that the high-frequency noise from one-tap STF is likely to be preferable in this case as it would be easier to remove with post-rendering filtering like TAA or DLSS. (These artifacts do disappear at higher sampling rates, i.e., more steps along each ray.) We attribute these artifacts to the extent of the tricubic filter: with 3×3 sharing in a wave, even if all neighboring pixels provide unique useful texels, less than 15% of the texels under the filter will be available. Further, nearby pixels will generally filter similar incomplete sets of texels, leading to correlation between pixels in each wave.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009