

# Stochastic Texture Filtering

Marcos Fajardo<sup>1</sup>, Bartłomiej Wronski<sup>2</sup>, Marco Salvi<sup>2</sup>, and Matt Pharr<sup>2</sup>

<sup>1</sup>Shiokara-Engawa Research

<sup>2</sup>NVIDIA

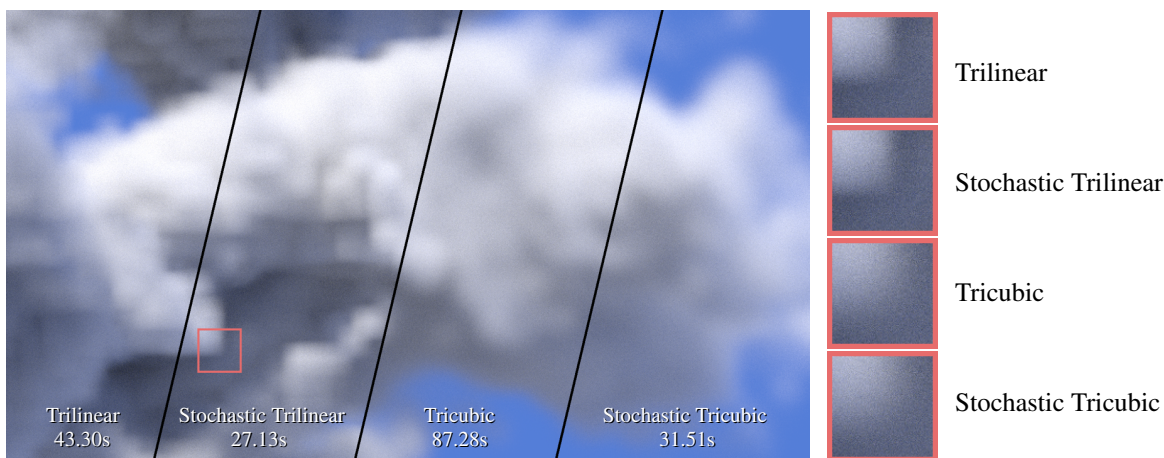


Figure 1: A section of the *Disney Cloud* scene rendered with path tracing. With this close-in viewpoint, trilinear filtering leads to blocky artifacts in the image. Tricubic filtering gives a much better result, though requires 64 voxel lookups into the *NanoVDB* representation. Stochastic filtering performs a single voxel lookup yet provides indistinguishable results, with overall rendering time speedups of  $1.60\times$  and  $2.77\times$  for the trilinear and tricubic filters. Times reported are for *pbrt-v4* running on an NVIDIA 4090 RTX GPU, rendering at 1080p with 256 samples per pixel.

## Abstract

2D texture maps and 3D voxel arrays are widely used to add rich detail to the surfaces and volumes of rendered scenes, and filtered texture lookups are integral to producing high-quality imagery. We show that filtering textures after evaluating lighting, rather than before BSDF evaluation as is current practice, gives a more accurate solution to the rendering equation. These benefits are not merely theoretical, but are apparent in common cases. We further show that stochastically sampling texture filters is crucial for enabling this approach, which has not been possible previously except in limited cases.

Stochastic texture filtering offers additional benefits, including efficient implementation of high-quality texture filters and efficient filtering of textures stored in compressed and sparse data structures, including neural representations. We demonstrate applications in both real-time and offline rendering and show that the additional stochastic error is minimal. Furthermore, this error is handled well by either spatiotemporal denoising or moderate pixel sampling rates.

## 1. Introduction

Image texture maps are essential to rich surface detail in most rendered images, thanks to the advanced texture painting tools available today, and the precise artistic control they allow. Three-dimensional voxel grids play a similar role for volumetric effects like clouds, smoke, and fire, allowing detailed offline physical sim-

ulations to be used. The number and resolution of both has continued to increase over the years.

Texture maps consist of uniform or sparsely distributed discrete points, which require continuous reconstruction through filtering. For computational efficiency, texture filtering is traditionally done prior to shading. For instance, GPUs are equipped with dedicated filtering units capable of bilinear or trilinear filtering, often at no

additional cost. However, this approach often results in low-quality reconstruction. We argue that it is generally better to filter *after* shading and address this gap in our work.

Texture mapping can be a dominant cost of offline rendering pipelines [GIF\*18, FHL\*18, BAC\*18] and the introduction of hardware-accelerated ray tracing to real-time renderers has caused the fraction of rendering time spent in texturing to correspondingly increase [Bur20]. Billions of lookups from textures and voxel grids may be necessary to render a single image, especially with multi-bounce path tracing where shaders are evaluated at every ray intersection. Higher-quality filters, such as anisotropic filters, generally require more texel lookups than simple filters, increasing the amount of memory bandwidth consumed. To save memory usage and bandwidth, recent works propose to store textures in more compressed formats and representations; examples include UDIM’s adaptive tiling, multi-level sparse grids [Mus13], and, recently, neural representations [KLM22, VSW\*23]. Those can reduce memory usage significantly, but are incompatible with hardware-accelerated filtering, and texture access is more computationally costly.

In this work, we introduce *stochastic texture filtering*, applying stochastic sampling to texture filtering and material network evaluation. Our contributions are as follows:

- We describe two ways of stochastically filtering textures, discuss their theoretical and practical differences, and connect them to prior work.
- We show that using stochastic texture filtering after lighting, rather than filtering the texture data, produces more accurate and *appearance-preserving* results.
- We demonstrate that the additional noise introduced by stochastic filtering in offline rendering is negligible and that moderate pixel sampling rates handle it well. In real-time rendering, this noise is effectively suppressed by using spatiotemporal reconstruction algorithms and blue-noise sampling patterns.
- We analyze how by decoding only a single source texel at each look-up, our algorithms make computationally-expensive compressed texture representations (traditional, sparse, or neural compressed) more viable.
- Finally, we show that our stochastic filtering algorithms further improve image quality by the use of high-quality and higher-order interpolating and approximating texture filters at a lower cost than trilinear filtering.

## 2. Background and Previous Work

The use of image textures in rendering dates to Blinn and Newell [Cat74, BN76]. Subsequent milestones in texture mapping include the introduction of spatially-varying filters [FLC80] and the use of image pyramids for efficient filtering [DSS78, Wil83]. See Heckbert’s survey article for comprehensive coverage of early work in this area [Hec86] and see Section 2.1 for further discussion of texture filtering.

A wide range of texture encodings have been developed, trading off memory and bandwidth consumption, computation, and compression error. Block-based compression [DM79] saves memory and bandwidth in exchange for some error; it is ubiquitous in GPUs today [SAM05, SP07, NLP\*12]. Higher compression rates

and lower error can be achieved with adaptive and neural representations [BAC96, Mus13, Mus21, KLM22, VSW\*23], though at a cost of multiple memory accesses and additional computation for each texel lookup; such formats are not supported by current GPUs and require manual filtering in shaders.

Monte Carlo estimation via stochastic sampling [CPC84, Coo86, Kaj86] has become the foundation of most approaches to rendering today. Production rendering has embraced path tracing for over a decade [KFC\*10], and there is now early adoption of path tracing for real-time rendering [CKK\*22]. Although lighting integrals are evaluated stochastically, their integrands are usually evaluated analytically. Integrands that are themselves stochastic have been used for complex BSDF models that cannot be evaluated analytically [HHdD16, GHZ18]. Related to our approach, stochastic evaluation of analytic quantities has been used to improve efficiency for multi-lobe BSDF evaluation [SSK03] and for many light sampling [SWZ96, EK18].

Real-time rendering has also embraced stochastic approaches. UV jittering as an alternative to bilinear filter dates as far back as the 90s and video games Star Trek: 25th Anniversary [Int92] and the original Unreal Engine [SN00]. More contemporary examples include stochastic alpha testing techniques that replace alpha blending with depth-tested random sampling [ESSL10, WM17], stochastic filtering of reflections [Sta15], and raytraced ambient occlusion [BBHW\*19]. Key enabling technologies are temporal anti-aliasing (TAA) [YLS20, Kar14] and temporal super-resolution (TSS) [Liu22]. Both are based on recursive filters and exponential-moving-averaging with adaptive history modification and rejection. TAA and TSS publications commonly describe the practice of *negative MIP biasing* used with screen-space jittering for a sharper image and approximate anisotropic filtering to improve appearance. We take this ad-hoc approach, formalize it, analyze how it deviates from anisotropic filtering, and show why it produces a more accurate filtered shading result.

The motivation for our work includes the filtering algorithms introduced by Hofmann et al. [HHCM21] and Vaidyanathan et al. [VSW\*23], who used stochastic trilinear filtering to improve performance. By avoiding evaluating an expensive decompression algorithm multiple times per voxel or pixel they see significant speedups. The OpenImageIO library [Gri22] also supports stochastic sampling of both MIP levels and anisotropic probes, and Lee et al. replaced filtered texture lookups with nearest-neighbor point samples, relying on the high sampling rates common in film production to resolve texture aliasing [LGXT17]. We expand on their results and provide a theoretical framework for a wider category of texture filters.

The pioneering work of Reeves et al. on shadow map filtering was the first to distinguish between filtering before lighting versus filtering afterward; their percentage closer filtering algorithm is based on filtering binary visibility rather than depth [RSC87]. They further showed the application of stochastic sampling to the filtering computation.

## 2.1. Texture Filtering

Textures are given as discrete, uniformly-spaced samples. Filtering texture lookups is challenging since each access generally requires a spatially-varying anisotropic filter that accesses multiple source texture samples. We can distinguish two main types of texture filtering: interpolation for translation and magnification, and lowpass filtering for minification. For both, we use the notation of a filter function  $f$  that is defined over the texture-space coordinates domain  $\mathbb{R}^n \rightarrow \mathbb{R}$  and  $(u, v, \dots)$  as its inputs. Without loss of generality, we use simplified, two-dimensional notation due to the separability of the sampling process. The filtered texture value is an integral of the product of  $f$  and the texture look-up function  $t$ :

$$F = \int f(u, v)t(u, v) du dv. \quad (1)$$

The texture function  $t$  is defined everywhere, but is non-zero only at discrete locations (typically uniform grid) due to impulse train sampling. We describe two practical realizations of this integral—a discrete one in Section 2.2, and a continuous one in Section 4.4.

**Interpolation:** Interpolation of an  $n$ -dimensional texture is typically done by sequentially interpolating all of the dimensions. If a texture represents a sampled continuous bandlimited signal, the original function value between samples can be perfectly reconstructed using sinc basis functions, though this is impractical due to the sinc’s infinite spatial support and often produces overshoots and ringing artifacts. Many alternative lowpass filters have been proposed, with various trade-offs in computation and number of texels accessed [MN88, TBU00, Get11]. Interpolating polynomial kernels are also often useful—nearest-neighbor (box kernel), linear (tent), and cubic [Key81] interpolation are all widely used. We present some of those kernels in the supplementary material.

**Non-interpolating and convolutional kernels:** Non-interpolating (approximating) kernels—where the original texel values are not preserved—are often used in practice, especially when a mild blurriness is preferred to aliasing or overshoots. The cubic B-spline is useful for texture magnification, as is an approximating quadratic kernel [Dod97] and the truncated Gaussian. The Gaussian filter is the only separable radially symmetric kernel in  $\mathbb{R}^n$  and can yield more pleasant reconstruction of diagonal edges than other kernels.

**Minification:** Minification during rendering transforms a high-resolution source texture to a lower resolution by mapping multiple source texels to a single pixel. Failure to filter those texels may result in significant aliasing. This process is more difficult than magnification; due to perspective transformation, the mapping is non-orthogonal—a single on-screen pixel can map to a trapezoid in texture space (Figure 2).

Because the input texels no longer form a regular grid, simple linear filters are not used in practice. The most common approach is trilinear MIP mapping [Wil83]. MIP mapping computes a bounding box of the filtering extent and selects the MIP map based on the largest of the two axes, leading to over blurring when the texture is not mapped orthogonally to the viewing plane. The practical solution to this blurring is anisotropic filtering of multiple samples from a higher-resolution MIP map. Examples include the elliptically weighted average (EWA) filter [GH86, Hec89] and a variety

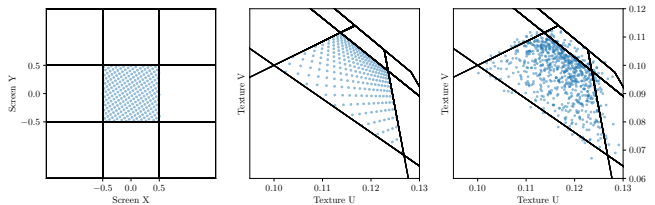


Figure 2: Uniform jittering in screen-space within pixel bounds (**left**) produces trapezoid, non-uniform coverage in the UV texture space (**middle**). Filter importance sampling then *additionally* jitters the resulting UVs in texture space for a desired reconstruction filter (Section 4.4), for example with Gaussian distribution (**right**).

of techniques that approximate high quality filters using multiple bilinear lookups [Bar97, MPFJ99, CS00].

## 2.2. Sampling Techniques

For reference, we summarize the well-known sampling techniques that we apply. See the books by Pharr et al. [PJH23] or Ross [Ros19] for further background. In the following, we will use  $\xi$  to denote uniform random variables in  $[0, 1)$  and angled brackets to denote expectation.

**Separable functions:** An  $n$ -dimensional function that is a product of 1D functions can be sampled by independently sampling each dimension. Many filters used for textures, including Gaussian and polynomials (linear, cubic, etc.) are separable.

**Weighted sums:** The simplest texture filtering function  $f$  can be represented as a set of discrete weights  $w$  defined for multiple discrete texture samples  $t$ . Given normalized weights  $w_i$  and texture values  $t_i$ , the filtered texture value is given by

$$F = \sum_{i=1}^n w_i t_i. \quad (2)$$

If a term  $j$  of the sum is sampled with probability equal to  $w_i$ , then an unbiased estimate of  $F$  is given by the corresponding texture value, unweighted:

$$\langle F \rangle = t_j. \quad (3)$$

Under the assumption that  $w_i$  are normalized, this is a special case of sampling a term according to probabilities  $p_i \propto w_i$  and applying the standard Monte Carlo estimator  $f_j/p_j$ . The weights  $w_i$  are often not normalized, and so must be normalized to find weights  $\tilde{w}_i = w_i/\sum_j w_j$  before filtering. However, in this case, we can simply skip normalization, sample  $j$  with probability proportional to  $w_i$ , and still apply Equation 3 to get the correct result.

**Uniform sample reuse:** Whenever a 1D random variable  $\xi$  is used to make a discrete sampling decision based on a probability  $p$ , then a new independent random variable  $\xi' \in [0, 1)$  can be derived from  $\xi$  [SWZ96]:

$$\xi' = \begin{cases} \xi/p & \text{if } \xi < p \\ (\xi - p)/(1 - p) & \text{otherwise.} \end{cases} \quad (4)$$

This technique can be useful when  $\xi$  is well-distributed (e.g., with

a blue noise spectrum [GF16] or with low discrepancy), allowing additional dimensions to benefit from  $\xi$ 's distribution as well as saving the cost of generating additional random samples.

**Sampling arrays:** An array of weights  $w_i$  (as from Equation 2) can be sampled by summing the weights and selecting the first item  $j$  where  $\xi < \sum_j^n w_j / \sum_i^n w_i$ .

**Weighted reservoir sampling:** Storing or recomputing all of the weights  $w_i$  may be undesirable, especially on GPUs. Weighted reservoir sampling [Cha82] with sample reuse [Oga21] can be applied with weights generated sequentially.

**Positivization:** Although negative weights can be sampled with probability based on their absolute value, doing so does not reduce variance as well as importance sampling does with positive functions [ESG06]. All interpolating filters of a higher order than the linear filter have negative lobes and being able to estimate them with low variance is essential for stochastic texture filtering. We apply positivization [OZ00], partitioning the filter weights  $w_i$  into positive ( $w_i^+$ ) and negative ( $w_i^-$ ) sets and sampling once from each set. Given respective sample indices  $j^+$  and  $j^-$ , the estimator of the filtered texture value of Equation 2 is

$$\langle F \rangle = \sum_i w_i^+ t_{j^+} - \sum_i w_i^- t_{j^-}. \quad (5)$$

If the original filter was normalized, the resulting positive and negative parts won't be and if Equation 2 is used, both sums need to be weighted. We include a practical example of positivization used for sampling the Mitchell bicubic filter in the supplementary material.

### 3. Effect of Texture Filtering on Rendering

Current practice in rendering is to filter textures before performing the lighting calculation, rather than applying the texture filter to the result of light lighting equation. We will start by formalizing the differences between those two approaches. In the following, we will define  $\hat{f}$  as the BSDF times the Lambertian cosine factor and parameterize it with the texture maps  $t_i$  that it depends on. We assume a single texture filter  $f$  and  $(u, v)$  parameterization.<sup>†</sup>

With this notation, the traditional lighting integral that gives outgoing radiance  $L_o$  at a point  $p$  in direction  $\omega_o$  is written:

$$L_o(p) = \int_{\mathbb{S}^2} \hat{f} \left( \omega_o, \omega', \int f(u, v) t_1(u, v) du dv, \dots \right) L_i(p, \omega') d\omega', \quad (6)$$

where the BSDF's parameters beyond the two directions are filtered textures.

Alternatively, we may write the integral with the order of integration exchanged, first integrating over the texture filter's extent and then integrating to compute outgoing radiance at points within the filter:

$$L_o(p) = \int f(u, v) \int_{\mathbb{S}^2} \hat{f}(\omega_o, \omega', t_1(u, v), \dots) L_i(p, \omega') d\omega' du dv. \quad (7)$$

<sup>†</sup> The generalization to different filters and texture coordinate parameterizations for different textures is straightforward, but clutters notation with change of variables factors.

The difference is that rather than using filtered values in the lighting integral, Equation 7 is *applying the filter to the result of the lighting integral itself*.

If a texture makes an affine contribution to the lighting integral (i.e., is a factor or a linear term of it), then both Equations 6 and 7 give the same result, since integration is a linear operator. Thus, they are the same with a texture value used as a diffuse coefficient but differ with a textured surface roughness used in an exponent. (The systematic error in Equation 6 in such cases can be analyzed using the Taylor series expansion of the integrated function. If the function is well-approximated by the first, linear term around the expansion point, the difference will be negligible but for highly non-linear functions with large higher-order Taylor series terms, the error is significant.)

Although filtering textures before integrating lighting is common practice in rendering, we argue that filtering outside of the lighting integral is preferable. There is precedent for this view: for example, this distinction is fundamental to percentage-closer shadow filtering (PCF), which is based on the insight that filtering depth values with shadow map lookups gives incorrect results, and filtering binary visibility is superior [RSC87]. Another motivating example comes from textures that are stored in non-linear formats like sRGB. When using such textures, inverting the non-linearity before filtering is essential for interpolation and minification correctness [Mic15]. Section 5.1 shows results with a number of other examples that illuminate cases where filtering lighting instead of textures gives superior results.

It is straightforward to filter textures first, but other than in special cases like PCF, it has not been obvious how to filter the lighting calculation. However, it is straightforward to apply stochastic sampling to the filter function in Equation 7: we can sample the filter  $f$  to find discrete texture coordinates  $(u', v')$  and use the corresponding texel values when evaluating the lighting integral. If  $f$  is normalized, then the Monte Carlo estimator  $f(u', v') / p(u', v') = 1$ , the filter factor disappears, and we are left to complete the lighting calculation using texels at  $(u', v')$ .

## 4. Filtering Algorithms and Rendering

In order to derive practical stochastic texture filtering algorithms, we can now apply the sampling techniques from Section 2.2 to the filters introduced in Section 2.1.

### 4.1. Linear Filters

Direct application of the array sampling algorithm from Section 2.2 and then Equation 3 gives the following estimator for linear interpolation over  $[0, 1]$ ,  $lerp(v_0, v_1, t) = (1 - t)v_0 + tv_1$ :

$$\langle lerp \rangle = \begin{cases} v_0, & \text{if } \xi > t \\ v_1 & \text{otherwise.} \end{cases} \quad (8)$$

Bilinear interpolation of values at the four corners of the unit square,  $bilerp(v_{00}, v_{10}, v_{01}, v_{11}, s, t)$ , can be implemented with nested linear interpolations. Applying the same approach and



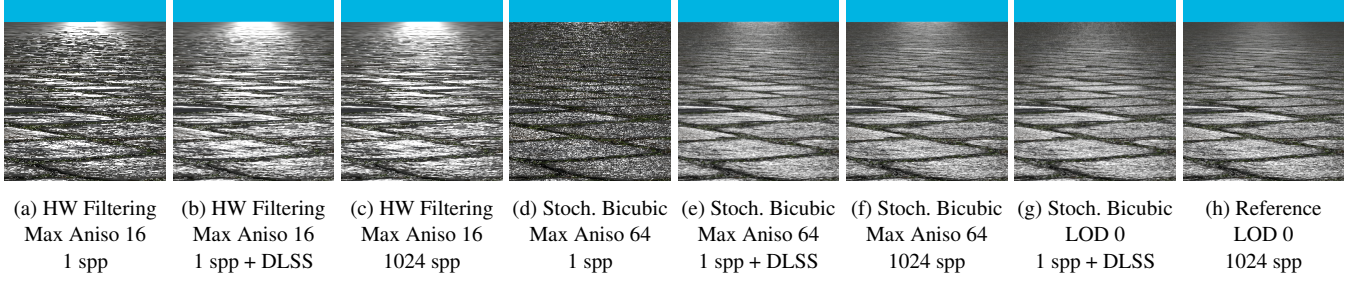


Figure 3: Appearance of a normal mapped material under minification. Stochastic texture filtering more accurately reconstructs the material’s appearance by filtering the material itself, while traditional texture filtering filters the surface normal before shading.

reusing the sample, we have:

$$\langle \text{bilerp} \rangle(s, t) = \begin{cases} v_{00}, & \text{if } \xi > s \text{ and } (\xi - s)/(1 - s) > t \\ v_{01}, & \text{if } \xi > s \text{ and } (\xi - s)/(1 - s) \leq t \\ v_{10}, & \text{if } \xi \leq s \text{ and } \xi/s > t \\ v_{11}, & \text{otherwise.} \end{cases} \quad (9)$$

It is straightforward to extend this estimator to trilinear interpolation, as used with MIP mapping and 3D voxel grids. More generally, the technique can be applied to  $n$ -dimensional interpolation, reducing from  $2^n$  texture lookups to a single one.

#### 4.2. B-Spline and Anisotropic Filters

Multidimensional B-spline filters are defined as a product of 1D cubic B-splines. For example, in 2D, given a lookup point  $(s, t)$  in  $[0, 1]^2$  with associated texture raster coordinates  $(\bar{s}, \bar{t})$ , the filtered texture value is given by  $4 \times 4$  weighted texel values:

$$\sum_{i=-1}^2 \sum_{j=-1}^2 K_{bs}(\lfloor \bar{s} \rfloor + i) K_{bs}(\lfloor \bar{t} \rfloor + j) t(\lfloor \bar{s} \rfloor + i, \lfloor \bar{t} \rfloor + j). \quad (10)$$

The B-spline filter is separable, and we apply weighted reservoir sampling to each dimension; in  $s$ , for example, we sample  $i' \in [-1, 0, 1, 2]$  according to the weights  $K_{bs}(\lfloor \bar{s} - 1 \rfloor)$ ,  $K_{bs}(\lfloor \bar{s} \rfloor)$ ,  $K_{bs}(\lfloor \bar{s} + 1 \rfloor)$ , and  $K_{bs}(\lfloor \bar{s} + 2 \rfloor)$ . The single texel value  $t(\lfloor \bar{s} \rfloor + i', \lfloor \bar{t} \rfloor + j')$  is then the unbiased estimator of Equation 10. Sampling higher-dimensional B-spline filters follows the same approach. For an  $n$  dimensional filter,  $4^n$  texture lookups are replaced with a single lookup. Separable sampling reduces the sample selection cost from  $4^n$  to  $4n$ .

Our implementation of stochastic sampling of the elliptically weighted average filter is also based on reservoir sampling: after stochastically selecting a MIP level based on the ellipse’s extent, we then simply compute all of the EWA filter weights and sample one based on their distribution.

#### 4.3. Material Graphs

Complex patterns are often generated using graphs composed of simple nodes such as scales, mixtures, and color corrections, with textures at the leaves. In offline rendering, it is not uncommon for these graphs to have hundreds of nodes and use many source textures, each of which is filtered at each shading point. Linear combinations of textures can be evaluated stochastically using Equa-

tion 8 and more complex blends such as triplanar mapping, which is based on a blend of three textures weighted by the orientation of the normal vector, can also be sampled stochastically.

#### 4.4. Filter Importance Sampling (FIS)

We have thus far introduced a toolbox of stochastic techniques for estimating discrete image filters. We can use a different approach to stochastically sample continuous filters without discretizing them. For a filtering operation given by the product of a normalized continuous convolutional filter  $f(u, v)$  with a texture  $t(u, v)$  expressed in the form of Equation 1, an unbiased estimate of  $F$  can be found using filter importance sampling [RSC87, Shi90, ESG06] (FIS):  $(u', v')$  is sampled from  $f(u, v)$ ’s distribution and the standard Monte Carlo estimator is applied, giving  $\langle F \rangle = t(u', v')$ . This approach is appealing for stochastic texture filtering since it allows for filters with infinite spatial support and doesn’t have a cost that necessarily scales with the filter’s width. The FIS framework can be used with positivization (Section 2.2) for low variance evaluation of filters with negative lobes.

Filter importance sampling a screen-space reconstruction filter is a common practice in production renderers. It can effectively approximate a minification filter, such as an anisotropic filter (Figure 2 left and middle). However, it is not enough to perform UV jittering for magnification, as it would produce nearest-neighbor interpolated texture and visual artifacts. This motivated prior work to use software bilinear filtering instead [LGXT17]. We propose to use FIS for texture reconstruction and sampling in addition to screen-space reconstruction filtering.

However, FIS assumes the integration of a product of two continuous functions. When using it to filter discrete samples, a practical realization draws a sample  $x'$  from  $f$  and then selects the closest texel  $\lfloor x' + 1/2 \rfloor$ . For  $n$ -dimensional filtering, this corresponds to applying a box reconstruction filter over  $[-1/2, 1/2]^n$  to the texture to make a continuous function  $t(x)$ . Equivalently, it corresponds to convolving the original filter function  $f$  with a box filter, changing its shape. Thus, the filter function that is sampled should be the deconvolution of the desired filter with the box function. This perspective allows us to better understand Hofmann et al.’s stochastic trilinear sampling algorithm, which is based on independent, uniform jittering in each dimension and then applying nearest neighbor sampling [HHCM21]. Their jittering corresponds to applying FIS

to sample the box filter which is then convolved with another box function, giving their stochastic trilinear interpolant.

We can thus filter with a B-spline filter of degree  $n$  by sampling a spline of degree  $n - 1$  and performing a nearest lookup, since approximating B-splines are constructed by repeated convolution of a box filter via the Cox–de Boor recursion formula [DB77]. (For example, a quadratic B-spline filter can be achieved by sampling a triangular PDF over  $[-1.5, 1.5]^2$ .) Sampling can either be performed via CDF inversion or by adding  $n$  uniformly-distributed random variables (also following the Cox–de Boor recursion).

This additional box function can be useful for rapidly changing filters such as a small-sigma Gaussian: evaluating it at discrete points results in subsampling error [Wro21] and the correction requires evaluating the *erf* error function. Filter importance sampling a regular, analytical normal distribution produces the same effect due to the convolution of a nearest-neighbor box function with the Gaussian. Furthermore, a Gaussian convolutional filter is an example of an infinite filter that is truncated in practice. With FIS, it is possible to evaluate an infinite filter by sampling the filter without truncation. This can simplify implementation (it is not necessary to carefully window the filter), as well as save the computational cost of multiple discrete weight evaluations and sample selection.

## 5. Results

We have evaluated stochastic texture filtering in the context of both real-time rasterization and path tracing using *Falcor* [KCK\*22], as well as offline rendering using *pbrt-v4* [PJH22]. All performance measurements were taken using an NVIDIA RTX 4090 GPU.

### 5.1. Filtering Order

It is well known that linear filtering of normal maps is incorrect and leads to changes in appearance [OB10]. An example is shown in Figures 3(a)-(c), where hardware texture filtering is used on a minified normal mapped surface. At points toward the horizon, the filter kernel is wide and filtering of the normals gives values close to the average normal in of the texture. Comparing to the reference image in Figure 3(h), which was rendered with no filtering and many pixel samples, we see that filtering before computing lighting introduces a significant error.

Stochastically filtering the textures allows the use of the estimator in Equation 7, which can be understood to be filtering the material itself over its distribution of normals. Results are shown in Figures 3(d)-(f), which are much closer to the reference; stochastic filtering effectively translates minified bumps and imperfections into increased roughness appearance. Our stochastic filters have some error due to their use of MIP maps, which are linearly filtered, though this error is small, as can be seen by comparing Figures 3(e) and (g), which were rendered using the same settings, save for Figure 3(g) using only the most detailed MIP level.

Because stochastic filtering only uses uninterpolated single texel values, only normals that are present in the normal map are used for lighting calculations. Thus, it can be understood as filtering discrete piecewise-linear microgeometry specified by the normal

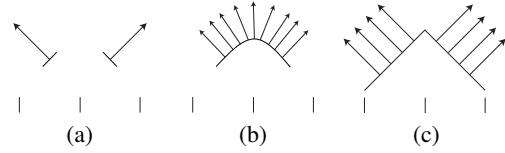


Figure 4: (a) Two texels with normals nearly 90 degrees apart. (b) With bilinear filtering, a smooth distribution of normals is reconstructed. (c) Stochastic filtering always uses single texel values from the image, so reconstructs an edge in this case.

map, rather than using the normals to reconstruct a smooth underlying surface. Depending on the artist’s intent, this behavior may be desirable—consider the example shown in Figure 4 where adjacent texels have significantly-different normals. With bilinear filtering, the filtered normals vary smoothly, corresponding to a smooth underlying surface, while stochastic filtering returns discrete normals.

Filtering BRDF properties prior to shading can lead to values violating the physical constraints of a BRDF model. Consider for example a texture with a scalar “metalness” parameter for a physically-based material model, where texels only have the values 0 and 1: with our approach, the material is only evaluated with metalness values of 0 and 1. At areas where the texture filter spans both values, we filter the material itself with only those two values. With traditional texture filtering, metalness values between 0 and 1 result, which may be nonsensical, depending on the material model. Our proposed filtering order allows for a more artist-friendly, non-linear, and compressed representation of full BRDF material models.

An example is shown in Figures 5(a) and (b), where a grid of temperature values is used to describe the full emission spectrum using Planck’s law, which is non-linear. With the traditional approach, filtered temperature values are used to compute the emission spectrum at points in the volume. In contrast, stochastic filtering effectively computes emission spectra at the grid points and then filters those spectra; it thus preserves appearance under minification, while filtering the temperatures does not. Figure 5(c) shows the error introduced if volumetric MIP maps are used under minification, due to linear filtering of the non-linear temperature. In contrast, using a stochastic minification filter (here, a Gaussian in the plane tangent to the ray), preserves appearance under minification, as shown in Figure 5(d).

### 5.2. Real-time Rendering

We evaluate stochastic texture filtering in a real-time renderer. Unlike software (CPU) renderers, real-time rendering with GPUs can use the hardware texturing unit with excellent bilinear filtering performance on standard texture formats. We do not expect stochastic texture filtering to provide performance benefits with those formats. We show, however, that it allows for efficient and high performance use of novel texture representation and compression formats not supported by existing hardware, as well as optimization of material graphs. Furthermore, we demonstrate how stochastic texture filtering enables magnification filters of significantly higher quality than

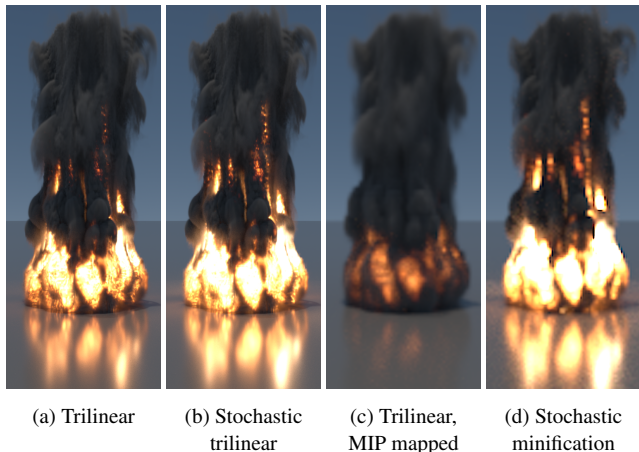


Figure 5: Effect of applying a non-linear mapping after filtering versus before. Traditional trilinear filtering (a) filters first, then uses Planck’s law to compute the volumetric emission spectrum. In contrast, stochastic trilinear filtering (b) takes a sample according to the texture filter and applies Planck’s law. Because Planck’s law is highly non-linear, the results differ. Under minification, (c) MIP mapping introduces error by applying linear filtering to non-linear quantities. Appearance is more accurately preserved with (d) a stochastic minification filter and no MIP maps.

the bilinear filter at the same cost, and more correct appearance preservation and minification.

In our experiments, we used DLSS [Liu22] as a robust temporal integrator. Screen-space jittering for DLSS employs a 32-sample Halton sequence, while Spatio-Temporal Blue Noise (STBN) masks [WMAM\*22] are used as the source of random numbers for stochastic filtering. Our implementation performs stochastic filtering in the shading pass, which uses the Disney BRDF [Bur12] and a single directional light. All images and performance measurements in this section were taken at 4K ( $3840 \times 2160$ ) resolution.

**Magnification, discrete filters:** For magnification, we analyze the visual benefits of high-quality bicubic Mitchell and truncated Gaussian filters with stochastic texture filtering by comparing with a simple bilinear filter, which is known for producing diamond-like artifacts and over-blurring. While the implementation of the stochastic Gaussian filter is straightforward, the Mitchell filter has negative weights and so we apply positivization (Section 2.2). In Figure 6 we observe better image quality from the higher-quality filters: either sharper response without bilinear filtering artifacts, or more pleasant diagonal edges and image smoothness. The use of STBN and DLSS results in no objectionable noise or flicker and the same performance cost as the bilinear filter.

**Magnification, filter importance sampling:** Filter importance sampling makes it possible to use infinite-extent filters without truncation. We compare FIS to sampling discrete filter weights using three Gaussian filters in Figure 7. For discrete sampling, we choose a single sample in the closest  $4 \times 4$  window of texels and for FIS, we use the Box–Muller transform to sample the Gaussian, followed by a nearest-neighbor lookup.



Figure 6: Bilinear filtering (a) compared to stochastic, single sample estimation of the bicubic Mitchell (b) and Gaussian (c) filters, resolved with DLSS’s temporal accumulation. The bicubic Mitchell filter is much sharper than the bilinear filter and does not produce diamond-like artifacts. The Gaussian filter is isotropic and although it tends to blur textures, it produces the most pleasing and natural reconstruction of diagonal lines.

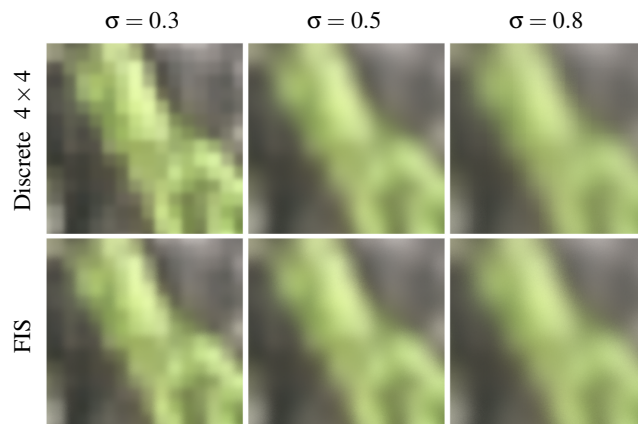


Figure 7: Gaussian texture filtering with varying  $\sigma$ , comparing discrete sample stochastic filtering and filter importance sampling. For  $\sigma = 0.5$ , both produce visually indistinguishable results. FIS gives better results for both relatively small and a large  $\sigma$ .

Results are visually indistinguishable for  $\sigma = 0.5$  but differ for the two other sigmas. With a very small  $\sigma$ , we observe undersampling with discrete sample weights. For the large  $\sigma$ , the limited radius of discrete sampling truncates the Gaussian kernel and produces visual artifacts. This can be improved by enlarging the filtering window, though with a corresponding increase in cost in sampling. FIS does not suffer from either of those issues, though it requires two random variables and cannot filter with exact kernels when convolution with a box filter is not desirable.

**Anisotropic filtering and minification:** Anisotropic filtering techniques commonly model the filter footprint as an ellipse, with axes derived from the partial derivatives of texture coordinates relative to screen coordinates. We build on that theory, but to save computational cost, we do not sample the ellipse in the shader but rely on screen-space jittering within the pixel to approximately sample the same extent. As shown in Figure 2, uniform jittering within the pixel gives a trapezoidal shape and projection in UV space. Although this does not preserve area or the original sample point distribution, it has no additional computational cost and in our experiments, approximates anisotropic filtering well.



The degree of anisotropy is determined by the ratio between the major and minor axes of the ellipse. We choose a MIP level based on the length of the minor axis and sample a single MIP level stochastically. Unlike current GPU hardware filtering, which has a maximum anisotropy ratio of 16, our method allows any anisotropy. We limit the ratio to 64 to avoid GPU texture cache thrashing, rescaling the minor axis if necessary.

Figure 8 shows a plane textured with a checkerboard pattern. Magnification is handled using filter importance sampling. The image reconstructed by DLSS is temporally stable, with occasional flickering in regions containing very high-frequency details. In motion, we observe sporadic ghosting and other temporal artifacts introduced by DLSS, but the overall image quality remains comparable to hardware anisotropic filtering. Although DLSS doesn’t completely remove noise caused by stochastic texture sampling, STBN reduces it, making it barely perceptible and only in magnified high-contrast areas. Figure 3 also demonstrates that temporal reconstruction is effective in recovering a high-quality anisotropically filtered image while only using 1 spp. We note that our approach of combining screen-space jittering with a higher-resolution MIP selection is similar to the ad-hoc practice of *negative MIP biasing* [YLS20, Kar14].

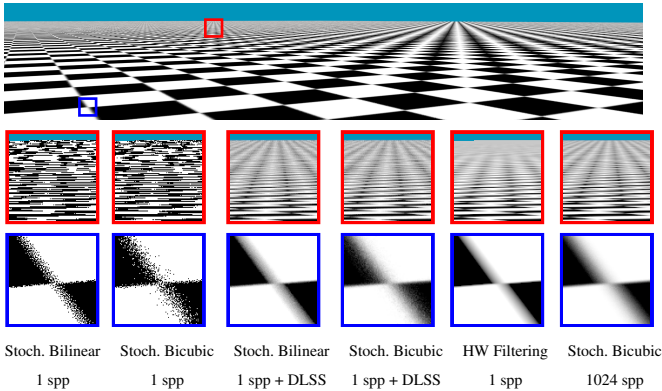


Figure 8: A checkerboard rendered using stochastic anisotropic and bicubic filtering (**top**). Red and blue insets (**bottom rows**) show magnified and magnified areas, respectively, comparing stochastic bilinear and bicubic filtering with hardware anisotropic filtering and a 1024 spp reference solution. Stochastic filtering uses FIS with STBN, except for the reference image that used a uniform distribution for the filtering.

**Triplanar mapping:** Triplanar mapping samples all textures three times with UV coordinates aligned to the  $XY$ ,  $XZ$ , and  $YZ$  planes and blends the filtered results based on the surface normal direction to avoid excessive texture stretching. Since it is a weighted average of three values, we can evaluate it stochastically using Equation 3. Results are shown in Figure 9. We find that DLSS resolves the stochastic sampling error effectively and observe no temporal visual artifacts such as flicker or ghosting. For this scene, the visual differences from filtering before shading versus filtering after shading are minor.

**Texture compression:** Stochastic texture filtering enables the use

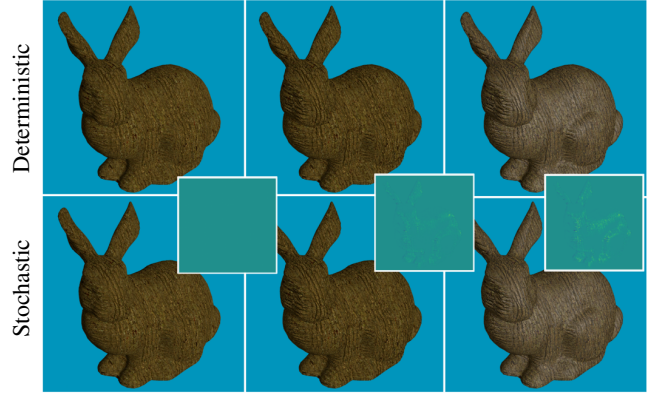


Figure 9: Full triplanar mapping (**top**) compared to its stochastic, single sample estimation (**bottom**). From left to right we present pure diffuse shading without normal mapping, diffuse shading with normal mapping, and full specular and diffuse lighting. Insets show error magnified  $10\times$ .

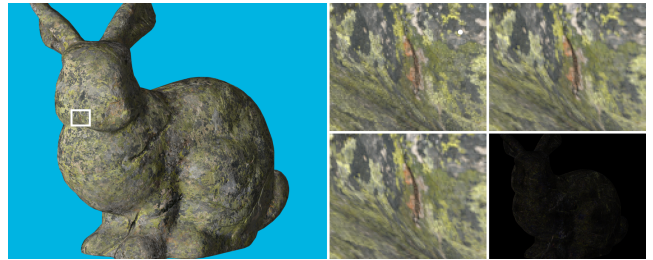


Figure 10: A 4K-resolution rendering of a DCT-compressed 9-channel  $4096 \times 4096$  texture set using stochastic filtering (**left**). Despite some loss of higher frequency details in the original uncompressed texture (**upper left inset**), the stochastic trilinear (**upper right inset**) and deterministic trilinear (**lower left inset**) filtering results appear virtually identical, as shown by the  $10\times$  magnified error image (**bottom right**). Stochastic filtering reduces rendering time from 1.66 ms to 0.57 ms.

of more advanced texture compression and decompression algorithms by requiring only a single texel to be decoded at each lookup [HHCM21, VSW\*23]. To connect those observations to our work, we implemented a much simpler real-time decompression algorithm—the 2D discrete cosine transform (DCT), where  $8 \times 8$  texel blocks store only 4 bytes per channel. We store the six lowest-frequency DCT coefficients for each channel, allocating 7 bits for the DC component and 5 bits for the remaining coefficients, achieving  $16\times$  compression for 8-bit data. This representation is not supported by GPU texture hardware, so texels must be decoded in the material evaluation shader and filtering must be performed manually.

As shown in Figure 10, stochastic trilinear filtering gives nearly identical visual results to deterministic trilinear filtering. We measure a  $2.9\times$  performance improvement; when combined with stochastic triplanar mapping, and performance is  $7.9\times$  better compared to fully-deterministic filtering.



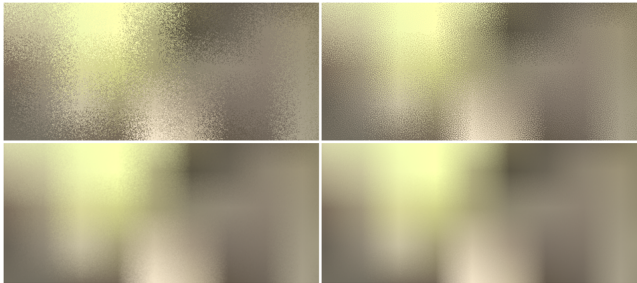


Figure 11: Ablation study on the effectiveness of DLSS and STBN on noise removal in a real-time setting. White noise (**left column**) creates visually distracting patterns of noise, while STBN (**right column**) dramatically reduces its appearance. When using DLSS as a temporal integrator (**bottom row**) the noise is dramatically reduced as compared to a single frame result (**top row**). DLSS and STBN work very well when combined, making the noise almost imperceptible.

**Visual noise ablation study:** To validate the effectiveness of DLSS [Liu22] as the temporal integrator and Spatio-Temporal Blue Noise (STBN) [WMAM\*22] as the source of the randomness, we performed an ablation study presented in Figure 11 and using extreme zoom-in on a high contrast area. We verify that as compared to white noise, STBN dramatically reduces the appearance of noise and improves its perceptual characteristics. Similarly, DLSS removes most of the noise—both in the case of white noise and STBN. When DLSS is used in combination with white noise, some visual grain remains, but it disappears completely when combined with STBN.

### 5.3. Offline Rendering

Offline rendering enjoys more generous pixel sampling rates than real-time rendering, which we have found to be sufficient to resolve the variance introduced by stochastic texture filtering. For example, Figure 12 shows a close view of a region of the *pbrt-v4 Watercolor* scene that exhibits texture magnification, rendered at just 8 spp. Both regular and stochastic texture filtering give very similar results, though stochastic filtering accesses as much as 7.85× fewer texels. Augmenting stochastic EWA with a stochastic bicubic magnification filter gives the best results, with 3.3× fewer texel accesses than regular trilinear filtering.

In order to evaluate the error introduced by stochastic filtering when used with volumetric path tracing, we rendered a view of the *Disney Cloud* data set [Wal17]. *pbrt*’s volumetric path tracer is based on delta tracking with null scattering [KHLN17, MGJ19] and uses ratio tracking [NSJ14] for transmittance. Because the cloud’s density is used to scale the absorption and scattering coefficients and since those make affine contributions to the estimated radiance values, both filtering approaches converge to the same result.

We converted the OpenVDB data set to NanoVDB for use on the GPU and used the 8× downsampled version of the cloud in order to make the differences between filtering algorithms more apparent. The image in Figure 1 was rendered at 1080p resolution with

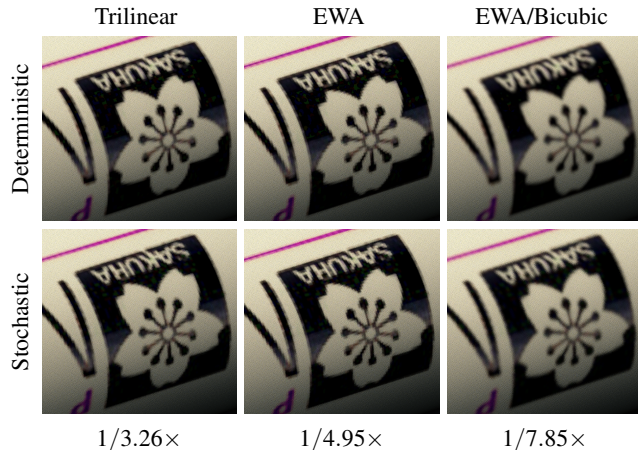


Figure 12: Stochastic texture filtering of a magnified texture, rendered with 8 spp in a path tracer. EWA gives better results along edges than trilinear filtering, though still has artifacts, which a bicubic magnification filter improves. The noise from stochastic texture filtering is minimal, while the reduction in number of texels accessed (ratios at the bottom) ranges from 3.26 – 7.85×.

	Rendering		Filtering	
	Time	Speedup	Time	Speedup
Trilinear	43.30 s		14.12 s	
Stoch. Trilinear	27.13 s	1.60×	3.27 s	4.32×
Tricubic	87.28 s		62.25 s	
Stoch. Tricubic	31.51 s	2.77×	5.10s	12.2×

Table 1: Performance when rendering Figure 1 at 1080p resolution with 256 spp. Stochastic filtering gives a significant performance benefit, both in overall rendering time and time spent filtering.

256 samples per pixel (spp). Trilinear filtering causes block- and diamond-shaped artifacts that are not present with tricubic filtering. Stochastic filtering gives images that are visually indistinguishable from traditional filtering; the error it introduces is far less than the error from Monte Carlo path tracing. For this scene, we saw less than a 5% increase in mean squared error (MSE) due to the stochastic filters. Table 1 reports performance: compared to trilinear filtering, tricubic filtering doubles rendering time since it requires 8× more texel lookups in the NanoVDB multilevel grid. With stochastic filtering, we are able to render using a high-quality tricubic filter in less time than trilinear filtering, with 1/8 as many texel lookups.

## 6. Discussion and Future Work

We have shown that stochastic texture filtering makes it possible to perform filtering outside of the lighting integral, rather than first filtering the texture parameters used by it. By doing so, systematic error is eliminated from rendered images in the common case where a textured parameter has a non-affine contribution to the final result. Examples include shadow mapping, normal mapping, roughness values used for microfacet distributions, and temperatures mapped to emission spectra. Filtering lighting in this way provides the benefit of preserving appearance at different scales.

Stochastic filtering offers additional benefits, including making more complex compressed texture representations viable by reducing complex filters to a single texel lookup. It further allows the use of higher-quality texture filters, as we have shown with bicubic and Gaussian filters; stochastic filtering makes it possible to use high-quality filters in high-performance code, providing further improvements in image quality. We hope that our work will contribute to the adoption of higher-order texture magnification filters in real-time rendering. This shift would reduce the reliance on low-quality bilinear filters, given our demonstration that the minor noise introduced by stochastic texture filtering can be effectively managed using temporal filtering algorithms like DLSS, or by employing moderate pixel sampling rates.

We note that the change to filtering the lighting calculation presents a challenge. Different renderers may produce varying results depending on which filtering method they use, even if their lighting and material systems are the same. It also means that our method could change the appearance of existing 3D assets, requiring art review before being used as a drop-in replacement.

For real-time rendering, we used DLSS to perform temporal filtering. DLSS is a learning-based solution and was not trained on such data. While the overall reconstruction quality is satisfactory, minor flickering and ghosting artifacts remain, especially in high-contrast areas and patterns like a checkerboard. Including stochastically-filtered texture in the training datasets would likely improve the reconstruction quality.

Our approach makes it feasible to use more complex reconstruction filters than are commonly used today. For example, non-linear content-dependent filters (such as steering kernels or the bilateral kernel) can be effective at reconstructing features like edges in images [TFM07] and volumes [YT13] and are essential for super-resolution. If such non-linear, local filter parameters or weights can be obtained cheaply (for example, from preprocessing or computed at a lower resolution [WGE\*19]), our stochastic filtering framework could be applied to them, giving further improvements to image quality.

## Acknowledgments

We would like to thank Aaron Lefohn and NVIDIA for supporting this work, John Burgess for suggesting the connection to percentage closer filtering, and Karthik Vaidyanathan for many discussions and suggestions. We are grateful to Walt Disney Animation Studios for making the detailed cloud model available and to Lennart Demes, author of the *ambientCG* website, for providing a public-domain PBR material database that we used to produce the real-time rendering figures.

## References

- [BAC96] BEERS A. C., AGRAWALA M., CHADDA N.: Rendering from compressed textures. In *Proceedings of SIGGRAPH 1996* (1996), pp. 373–378. doi:10.1145/237170.237276. 2
- [BAC\*18] BURLEY B., ADLER D., CHIANG M. J.-Y., DRISKILL H., HABEL R., KELLY P., KUTZ P., LI Y. K., TEECE D.: The design and evolution of Disney’s Hyperion renderer. *ACM Transactions on Graphics* 37, 3 (July 2018), 33:1–33:22. doi:10/gfjm8w. 2
- [Bar97] BARKANS A. C.: High quality rendering using the Talisman architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (1997), pp. 79–88. doi:10.1145/258694.258722. 3
- [BBHW\*19] BARRÉ-BRISEBOIS C., HALÉN H., WIHLIDAL G., LAURITZEN A., BEKKERS J., STACHOWIAK T., ANDERSSON J.: Hybrid rendering for real-time ray tracing. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (2019), 437–473. 2
- [BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Communications of the ACM* 19, 10 (Oct. 1976), 542–547. doi:10/dct2v6. 2
- [Bur12] BURLEY B.: Physically-based shading at Disney. In *Practical Physically-Based Shading in Film and Game Production* (2012), pp. 10:1–10:7. 7
- [Bur20] BURGESS J.: RTX on—the NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44. doi:10.1109/MM.2020.2971677. 2
- [Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, Dec. 1974. 2
- [Cha82] CHAO M.-T.: A general purpose unequal probability sampling plan. *Biometrika* 69, 3 (Dec. 1982), 653–656. doi:10/fd87zs. 4
- [CKK\*22] CLARBERG P., KALLWEIT S., KOLB C., KOZLOWSKI P., HE Y., WU L., LIU E., BITTERLI B., PHARR M.: Real-Time Path Tracing and Beyond. HPG 2022 Keynote, July 2022. 2
- [Coo86] COOK R. L.: Stochastic sampling in computer graphics. *ACM Transactions on Graphics* 5, 1 (Jan. 1986), 51–72. doi:10/cqwhcc. 2
- [CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *Computer Graphics (Proceedings of SIGGRAPH)* 18, 3 (July 1984), 137–145. doi:10/c9thc3. 2
- [CS00] CANT R. J., SHRUBSOLE P. A.: Texture potential MIP mapping, a new high-quality texture antialiasing algorithm. *ACM Trans. Graph.* 19, 3 (July 2000), 164–184. doi:10.1145/353981.353991. 3
- [DB77] DE BOOR C.: Package for calculating with b-splines. *SIAM Journal on Numerical Analysis* 14, 3 (1977), 441–472. 6
- [DM79] DELP E. J., MITCHELL O. R.: Image compression using block truncation coding. *IEEE Transactions on Communications* 27, 9 (September 1979), 1335–1341. 2
- [Dod97] DODGSON N. A.: Quadratic interpolation for image resampling. *IEEE Transactions on Image Processing* 6, 9 (1997), 1322–1326. 3
- [DSS78] DUNGAN W., STENGER A., SUTTY G.: Texture tile considerations for raster graphics. In *Proceedings SIGGRAPH 1978* (1978), pp. 130–134. doi:10.1145/800248.807383. 2
- [Duc79] DUCHON C. E.: Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology and Climatology* 18, 8 (1979), 1016–1022. 12
- [EK18] ESTEVEZ A. C., KULLA C.: Importance sampling of many lights with adaptive tree splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (Aug. 2018), 25:1–25:17. doi:10/ggh89v. 2
- [ESG06] ERNST M., STAMMINGER M., GREINER G.: Filter importance sampling. In *Proceedings of IEEE Symposium on Interactive Ray Tracing* (Sept. 2006), pp. 125–132. doi:10/c2q6gj. 4, 5
- [ESS10] ENDERTON E., SINTORN E., SHIRLEY P., LUEBKE D.: Stochastic transparency. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), pp. 157–164. 2
- [FHL\*18] FASCIONE L., HANIKA J., LEONE M., DROSKE M., SCHWARZHaupt J., DAVIDOVIĆ T., WEIDLICH A., MENG J.: Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics* 37, 3 (Aug. 2018), 31:1–31:18. doi:10/gfznb. 2
- [FLC80] FEIBUSH E. A., LEVOY M., COOK R. L.: Synthetic texturing using digital filters. *SIGGRAPH Comput. Graph.* 14, 3 (jul 1980), 294–301. doi:10.1145/965105.807507. 2

- [Get11] GETREUER P.: Linear methods for image interpolation. *Image Processing On Line 1* (2011), 238–259. 3
- [GF16] GEORGIEV I., FAJARDO M.: Blue-noise dithered sampling. In *ACM SIGGRAPH Talks* (2016), ACM Press, pp. 35:1–35:1. doi:10/gfznbx.4
- [GH86] GREENE N., HECKBERT P. S.: Creating raster Omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications* 6, 6 (1986), 21–27. doi:10.1109/MCG.1986.276738.3,14
- [GHZ18] GUO Y., HAŠAN M., ZHAO S.: Position-free Monte Carlo simulation for arbitrary layered BSDFs. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 37, 6 (Dec. 2018), 279:1–279:14. doi:10/db3c.2
- [GIF\*18] GEORGIEV I., IZE T., FARNSWORTH M., MONTOYA-VOZMEDIANO R., KING A., LOMMEL B. V., JIMENEZ A., ANSON O., OGAKI S., JOHNSTON E., HERUBEL A., RUSSELL D., SERVANT F., FAJARDO M.: Arnold: A brute-force production path tracer. *ACM Transactions on Graphics* 37, 3 (Aug. 2018), 32:1–32:12. doi:10/gfznb2.2
- [Gri22] GRITZ L.: OpenImageIO 2.4, 2022. URL: <https://github.com/OpenImageIO/oio/releases/tag/v2.4.4.1>. 2
- [Hec86] HECKBERT P. S.: Survey of texture mapping. *IEEE Computer Graphics and Applications* 6, 11 (1986), 56–67. doi:10.1109/MCG.1986.276672.2
- [Hec89] HECKBERT P.: *Fundamentals of Texture Mapping and Image Warping*. PhD thesis, UC Berkeley, June 1989. 3,14
- [HHCM21] HOFMANN N., HASSELGREN J., CLARBERG P., MUNKBERG J.: Interactive path tracing and reconstruction of sparse volumes. *Proc. ACM Comput. Graph. Interact. Tech.* 4, 1 (Apr. 2021). doi:10.1145/3451256.2,5,8
- [HHd16] HEITZ E., HANIKA J., D'EON E., DACHSBACHER C.: Multiple-scattering microfacet BSDFs with the Smith model. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 35, 4 (July 2016), 58:1–58:14. 2
- [Int92] INTERPLAY P.: Texture scaling in star trek: 25th anniversary, 1992. URL: <https://st25sprites.neocities.org/scaling.2>
- [Kaj86] KAJIYA J. T.: The rendering equation. *Computer Graphics (Proceedings of SIGGRAPH)* 20, 4 (Aug. 1986), 143–150. doi:10/cvf53j.2
- [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, 8 2022. URL: <https://github.com/NVIDIAGameWorks/Falcor.6>
- [Key81] KEYS R.: Cubic convolution interpolation for digital image processing. *IEEE transactions on acoustics, speech, and signal processing* 29, 6 (1981), 1153–1160. 3
- [KFC\*10] KŘIVÁNEK J., FAJARDO M., CHRISTENSEN P. H., TABELLION E., BUNNELL M., LARSSON D., KAPLAYAN A.: Global illumination across industries. In *ACM SIGGRAPH Courses* (2010), ACM Press. 2
- [KHLN17] KUTZ P., HABEL R., LI Y. K., NOVÁK J.: Spectral and decomposition tracking for rendering heterogeneous volumes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 36, 4 (July 2017), 111:1–111:16. doi:10/gbxjxg.9
- [KLM22] KIM D., LEE M., MUSETH K.: NeuralVDB: High-resolution sparse volume representation using hierarchical neural networks, 2022. arXiv:2208.04448. 2
- [LGXT17] LEE M., GREEN B., XIE F., TABELLION E.: Vectorized production path tracing. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, Association for Computing Machinery. doi:10.1145/3105762.3105768.2,5
- [Liu22] LIU E.: DLSS 2.0 - Image Reconstruction for Real-Time Rendering with Deep learning. In *Game Developers Conference* (2022). 2,7,9
- [MGJ19] MILLER B., GEORGIEV I., JAROSZ W.: A null-scattering path integral formulation of light transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* 38, 4 (July 2019), 44:1–44:13. doi:10/gf6rzb.9
- [Mic15] MICROSOFT: Direct3d 11.3 functional specification, 2015. URL: [https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11\\_3\\_FunctionalSpec.htm.4](https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm.4)
- [MN88] MITCHELL D. P., NETRAVALI A. N.: Reconstruction filters in computer graphics. *SIGGRAPH Comput. Graph.* 22, 4 (June 1988), 221–228. doi:10/fggxbg.3,12
- [MPFJ99] MCCORMACK J., PERRY R., FARKAS K., JOUPPIN N.: Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH* (1999), pp. 243–250. doi:10.1145/311535.311562.3
- [Mus13] MUSETH K.: VDB: High-resolution sparse volumes with dynamic topology. *ACM Transactions on Graphics* 32, 3 (July 2013), 27:1–27:22. doi:10/gfzq7s.2
- [Mus21] MUSETH K.: NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. *ACM SIGGRAPH 2021 Talks*, 2021. 2
- [NLP\*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive Scalable Texture Compression. In *Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics* (2012), Dachsbacher C., Munkberg J., Pantaleoni J., (Eds.), The Eurographics Association. doi:10.2312/EGGH/HPG12/105-114.2
- [NSJ14] NOVÁK J., SELLE A., JAROSZ W.: Residual ratio tracking for estimating attenuation in participating media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)* 33, 6 (Nov. 2014), 179:1–179:11. doi:10/f6r2nq.9
- [OB10] OLANO M., BAKER D.: LEAN mapping. In *Proceedings of the Symposium on Interactive 3D Graphics and Games* (2010), pp. 181–188. doi:10/fkbvnp.6
- [Oga21] OGAKI S.: Vectorized reservoir sampling. In *SIGGRAPH Asia 2021 Technical Communications* (2021), Association for Computing Machinery. doi:10.1145/3478512.3488602.4,14
- [OZ00] OWEN A., ZHOU Y.: Safe and effective importance sampling. *Journal of the American Statistical Association* 95, 449 (2000), 135–143. 4
- [PJH22] PHARR M., JAKOB W., HUMPHREYS G.: pbrt-v4, 2022. URL: <https://github.com/mmp/pbrt-v4.6>
- [PJH23] PHARR M., JAKOB W., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*, 4th ed. MIT Press, Cambridge, MA, 2023. 3
- [Ros19] ROSS S. M.: *A First Course in Probability*, 10th ed. Pearson, 2019. 3
- [RSC87] REEVES W. T., SALESIN D. H., COOK R. L.: Rendering antialiased shadows with depth maps. *Computer Graphics (Proceedings of SIGGRAPH)* (1987), 283–291. doi:10/bc4hw2.2,4,5
- [SAM05] STRÖM J., AKENINE-MÖLLER T.: iPACKMAN: High-quality, low-complexity texture compression for mobile phones. In *Graphics Hardware* (2005), pp. 63–70. 2
- [Shi90] SHIRLEY P.: *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois, Urbana-Champaign, Nov. 1990. 5



- [SN00] SWEENEY T., NETTLE P.: Texturing as in unreal, 2000. URL: [https://www.flipcode.com/archives/Texturing\\_As\\_In\\_Unreal.shtml](https://www.flipcode.com/archives/Texturing_As_In_Unreal.shtml). 2
- [SP07] STRÖM J., PETTERSSON M.: ETC2: Texture Compression using Invalid Combinations. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2007). doi:10.2312/EGGH/EGGH07/049-054. 2
- [SSK03] SZÉCSI L., SZIRMAY-KALOS L., KELEMEN C.: Variance reduction for Russian-roulette. *Journal of the World Society for Computer Graphics (WSCG) 11*, 1–3 (2003). 2
- [Sta15] STACHOWIAK T.: Stochastic screen-space reflections. In *Advances in Real-Time Rendering in Games, Part I* (2015), ACM SIGGRAPH Courses. doi:10/gf3s6n. 2
- [SWZ96] SHIRLEY P., WANG C., ZIMMERMAN K.: Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics 15*, 1 (Jan. 1996), 1–36. doi:10/ddgbgg. 2, 3
- [TBU00] THÉVENAZ P., BLU T., UNSER M.: Interpolation revisited. *IEEE Transactions on Medical Imaging 19*, 7 (2000), 739–758. 3
- [TFM07] TAKEDA H., FARSIU S., MILANFAR P.: Kernel regression for image processing and reconstruction. *IEEE Transactions on Image Processing 16*, 2 (Feb. 2007), 349–366. doi:10/b4gfw. 10
- [VSW\*23] VAIDYANATHAN K., SALVI M., WRONSKI B., AKENINE-MÖLLER T., EBELIN P., LEFOHN A.: Random-Access Neural Compression of Material Textures. In *Proceedings of SIGGRAPH* (2023). 2, 8
- [Wal17] WALT DISNEY ANIMATION STUDIOS: Cloud data set, 2017. URL: <https://disneyanimation.com/data-sets/>. 9
- [WGE\*19] WRONSKI B., GARCIA-DORADO I., ERNST M., KELLY D., KRAININ M., LIANG C.-K., LEVOY M., MILANFAR P.: Handheld multi-frame super-resolution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 38*, 4 (July 2019), 28:1–28:18. doi:10/gf6d4v. 10
- [Wil83] WILLIAMS L.: Pyramidal parametrics. *Computer Graphics (Proceedings of SIGGRAPH) 17*, 3 (July 1983), 1–11. doi:10/cq4xrd. 2, 3
- [WM17] WYMAN C., MCGUIRE M.: Hashed alpha testing. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2017), pp. 7:1–7:9. 2
- [WMAM\*22] WOLFE A., MORRICAL N., AKENINE-MÖLLER T., RAMAMOORTHY R., GHOSH A., WEI L.: Spatiotemporal blue noise masks. In *Eurographics Symposium on Rendering* (2022), pp. 117–126. 7, 9
- [Wro21] WRONSKI B.: Practical Gaussian filtering: Binomial filter and small sigma Gaussians, 2021. URL: <https://bartwronski.com/2021/10/31/practical-gaussian-filter-binomial-filter-and-small-sigma-gaussians/>. 6
- [YLS20] YANG L., LIU S., SALVI M.: A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum 39*, 2 (2020), 607–621. 2, 8
- [YT13] YU J., TURK G.: Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Transactions on Graphics (TOG) 32*, 1 (2013), 1–12. 10

## Supplemental Material

### Appendix A: Filter Kernels

For reference, we summarize a number of commonly-used filter kernels, starting with interpolating polynomials. Their one-dimensional definitions are listed here;  $n$ -dimensional filtering is performed by filtering each dimension independently—the filters are separable. See Figure 13 for graphs of the kernels and how they filter an example set of samples.

The 0th-order kernel is a unit box function, which corresponds to nearest-neighbor sampling.

$$K_0(t) = \begin{cases} 1, & \text{if } |t| < \frac{1}{2} \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

The first order kernel is the unit tent, which gives linear sampling.

$$K_1(t) = \begin{cases} (1 - |t|), & |t| < 1 \\ 0 & \text{otherwise.} \end{cases} \quad (12)$$

The cubic polynomial kernel is defined as

$$K_3(t) = \begin{cases} (a+2)|t|^3 - (a+3)|t|^2 + 1, & |t| < 1 \\ a|t|^3 - 5a|t|^2 + 8a|t| - 4a, & 1 < |t| < 2 \\ 0 & \text{otherwise,} \end{cases} \quad (13)$$

where  $a$  is an extra degree of freedom in cubic interpolation. Mitchell and Netravali [MN88] recommend a value of  $-0.5$  and it is the closest to Lanczos, a windowed sinc kernel [Duc79] while keeping low evaluation cost.

The Lanczos kernel is:

$$K_{Ln}(t) = \begin{cases} 1 & t = 0, \\ \frac{\sin(x\pi)}{x} \frac{\sin(\pi x/n)}{x/n}, & 0 < |t| < n \\ 0 & \text{otherwise.} \end{cases} \quad (14)$$

### Convolutional

The non-interpolating cubic B-spline is also useful; it is used for both examples in Section 5.3, for example.

$$K_{bs}(t) = \frac{1}{6} \begin{cases} 4 - 3t^2(2 - |t|) & |t| \leq 1 \\ (2 - |t|)^3 & 1 < |t| \leq 2 \\ 0 & \text{otherwise.} \end{cases} \quad (15)$$

### Appendix B: Filter Implementations

For reference, implementations of most of the stochastic filters in the paper are in the following. (We skip cases like the stochastic trilinear filter, since it is a straightforward modification to the stochastic bilinear filter, for example.) See the file `stochtex.h` in the



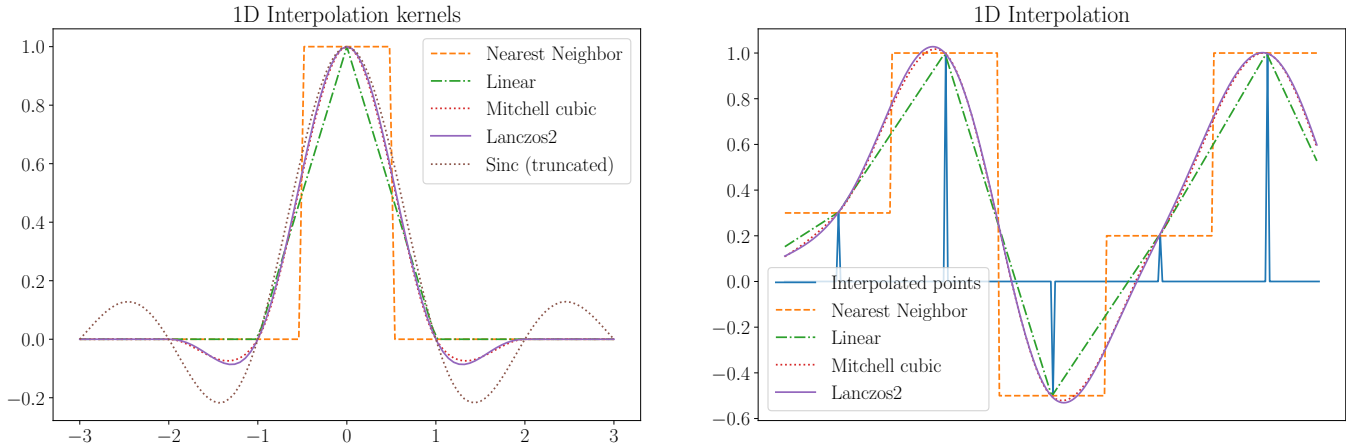


Figure 13: Commonly used 1D interpolation kernels and resulting interpolation of a random 1D points – nearest-neighbor (box filter), linear, Mitchell cubic, and Lanczos2. Lanczos2 is almost identical to the Mitchell kernel.

supplemental material for implementations of all of our stochastic filter functions.

All of the following parameters take a parameter  $u$ , which should be a uniform random sample in  $[0, 1]$  and a lookup point that is assumed to be with respect to texture raster coordinates (i.e., it ranges between 0 and the texture’s resolution in each dimension). They return (by reference) a remapped uniform random sample that may be reused and stochastically-sampled integer texel coordinates.

StochasticBilinear stochastically samples the bilinear function.

Listing 1: Sampling a 2D bilinear kernel

```
Point2i StochasticBilinear(Point2f st, float &u) {
    int s = std::floor(st[0]), t = std::floor(st[1]);
    float ds = st[0] - std::floor(st[0]);
    float dt = st[1] - std::floor(st[1]);
    if (u < ds) {
        ++s;
        u /= ds;
    } else
        u = (u - ds) / (1 - ds);

    if (u < dt) {
        ++t;
        u /= dt;
    } else
        u = (u - dt) / (1 - dt);

    return Point2i(s, t);
}
```

The bicubic kernel based on Equation 15 is stochastically sampled by StochasticBicubic. The computed weights correspond to the weights for the two texels to the left of the lookup point (weights[0] and weights[1]) and the two to the right (weights[2] and weights[3]). This implementation stores each dimension’s filter weights in an array and then samples a single filter tap.

Listing 2: Sampling a B-spline cubic kernel in 2D

```
Point2i StochasticBicubic(Point2f st, float &u) {
    // Compute filter weights
    auto weights = [] (float t, float w[4]) {
        float t2 = t*t;
        w[0] = (1.f/6.f) * (-t*t2 + 3*t2 - 3*t + 1);
        w[1] = (1.f/6.f) * (3*t*t2 - 6*t2 + 4);
        w[2] = (1.f/6.f) * (-3*t*t2 + 3*t2 + 3*t + 1);
        w[3] = (1.f/6.f) * t*t2;
    };
    float ws[4], wt[4];
    weights(st[0] - std::floor(st[0]), ws);
    weights(st[1] - std::floor(st[1]), wt);

    // Sample index based on weights in each dimension.
    int s0 = std::floor(st[0]-1), t0 = std::floor(st[1]-1);
    int s = SampleDiscrete(ws, u, nullptr, &u);
    int t = SampleDiscrete(wt, u, nullptr, nullptr);
    return {s0+s, t0+t};
}
```

In the following implementation, StochasticTricubic uses weighted reservoir sampling to sample the filter in each dimension. In this way, the filter weights can be computed on the fly and do not all need to be stored at once.

Listing 3: Sampling a cubic B-spline kernel in 3D

```

Point3i StochasticTricubic(Point3f pIndex, float &u) {
    int ix = std::floor(pIndex.x);
    int iy = std::floor(pIndex.y);
    int iz = std::floor(pIndex.z);
    float deltas[3] = {pIndex.x - ix, pIndex.y - iy, pIndex.z - iz};

    int idx[3];
    for (int i = 0; i < 3; ++i) {
        float sumWt = 0;
        float t = deltas[i];
        float t2 = t*t;

        // Weighted reservoir sampling, first tap always accepted
        float w0 = (1.f/6.f) * (-t*t2 + 3*t2 - 3*t + 1);
        sumWt = w0;
        idx[i] = 0;

        // Weighted reservoir sampling helper
        auto wrs = [&](int j, float w) {
            sumWt += w;
            float p = w/sumWt;
            if (u < p) {
                idx[i] = j;
                u /= p;
            } else
                u = (u - p) / (1 - p);
        };
        // Sample the other 3 filter taps
        wrs(1, (1.f/6.f) * (3*t*t2 - 6*t2 + 4));
        wrs(2, (1.f/6.f) * (-3*t*t2 + 3*t2 + 3*t + 1));
        wrs(3, (1.f/6.f) * t*t2);
    };

    // idx stores the index of the sampled filter tap in
    // each dimension.
    return Point3i(ix-1+idx[0], iy-1+idx[1], iz-1+idx[2]);
}

```

Listing 4: Stochastic sampling of the EWA kernel

```

Point2i StochasticEWA(Point2f st, Vector2f dst0, Vector2f dst1,
    float &u) {
    // Find ellipse coefficients that bound EWA filter region
    float A = Sqr(dst0[1]) + Sqr(dst1[1]) + 1;
    float B = -2 * (dst0[0] * dst0[1] + dst1[0] * dst1[1]);
    float C = Sqr(dst0[0]) + Sqr(dst1[0]) + 1;
    float invF = 1 / (A * C - Sqr(B) * 0.25f);
    A *= invF;
    B *= invF;
    C *= invF;

    // Compute the ellipse's $(s,t)$ bounding box in texture space
    float det = -Sqr(B) + 4 * A * C;
    float invDet = 1 / det;
    float uSqrt = SafeSqrt(det * C), vSqrt = SafeSqrt(A * det);
    int s0 = std::ceil(st[0] - 2 * invDet * uSqrt);
    int s1 = std::floor(st[0] + 2 * invDet * uSqrt);
    int t0 = std::ceil(st[1] - 2 * invDet * vSqrt);
    int t1 = std::floor(st[1] + 2 * invDet * vSqrt);

    // Scan over ellipse bound and evaluate quadratic equation
    float sumWts = 0;
    Point2i coords;
    for (int it = t0; it <= t1; ++it) {
        float tt = it - st[1];
        for (int is = s0; is <= s1; ++is) {
            float ss = is - st[0];
            float r2 = A * Sqr(ss) + B * ss * tt + C * Sqr(tt);
            if (r2 >= 1)
                continue;

            int index = std::min<int>(r2 * MIPFilterLUTSize,
                MIPFilterLUTSize - 1);
            float weight = MIPFilterLUT[index];
            if (weight <= 0)
                continue;

            sumWts += weight;
            float p = weight / sumWts;
            if (u < p) {
                coords = Point2i(is, it);
                u /= p;
            } else
                u = (u - p) / (1 - p);
        }
    }
    return coords;
}

```

Our implementation for sampling the EWA kernel [GH86, Hec89] is in `StochasticEWA`. It largely follows the implementation of EWA filtering in `pbrt`, except that as filter weights are computed in the innermost loop, vectorized weighted reservoir sampling [Oga21] is used to select a single filter tap.

## The interpolating (negative lobe) bicubic filter

In Section 5.2 we presented results of real-time rendering with a stochastically estimated variant of the Mitchell bicubic filter. This filter has negative lobes and for most of the fractional subpixel offsets, has a mix of negative and positive weights. As described in Section 2.2, the solution that minimizes the variance of such a filter splits the integral into two parts.

From the set of all filter weights, we consider the sets of positive and negative weights separately and select a sample from each of the sets independently. Then, the filter takes two samples, weighted by the sum of the absolute values of each of the sampling sets. One way of implementing it uses weighted reservoir sampling with warping (also described in Section 2.2) and two separate reservoirs for the negative and positive samples. An example implementation in C++-like pseudocode is presented in Listing 5.

Listing 5: Sampling an interpolating (negative lobe) bicubic kernel.

```

TexelValue SampleBicubic(const Texture& texture,
                        const Vector2& pixel_coord,
                        float& u) {
    const Vector2 top_left = floor(pixel_coord);
    const Vector2 fract_offset = pixel_coord - top_left;

    float pos_weights_sum = 0.0f;
    float neg_weights_sum = 0.0f;
    Vector2 selected_neg_offset;
    Vector2 selected_pos_offset;
    for (int dy = -1; dy <= 2; ++dy) {
        float weight_dy = MitchellCubic(fract_offset.y - dy);
        for (int dx = -1; dx <= 2; ++dx) {
            float weight_dx = MitchellCubic(fract_offset.x - dx);
            float w = weight_dy * weight_dx;
            float& selected_reservoir_sum = w < 0.0f ?
                neg_weights_sum :
                pos_weights_sum;
            Vector2& selected_reservoir = w < 0.0f ?
                selected_neg_offset :
                selected_pos_offset;

            selected_reservoir_sum += abs(w);
            float p = abs(w) / selected_reservoir_sum;
            if (u <= p) {
                selected_reservoir = Vector2(dx, dy);
                u = u / p;
            } else {
                u = (u - p) / (1 - p);
            }
        }
    }
    Vector2 pos_coord = top_left + selected_pos_offset;
    TexelValue sampled_val = pos_weights_sum *
        SampleTexture(texture, pos_coord);
    // It's possible to not have any negative sample, for example,
    // when the fractional offset is exactly 0 or very small.
    if (neg_weights_sum != 0.0f) {
        Vector2 neg_coord = top_left + selected_neg_offset;
        sampled_val += -neg_weights_sum *
            SampleTexture(texture, neg_coord);
    }
    return sampled_val;
}

```

### Real-time discrete and filter importance sampling

In Section 5.2 we compared discrete sampling to FIS and their pros and cons for filtering with an infinite Gaussian, discrete approximation, and the impact on the image quality. In Listing 6 and Listing 7 we present an HLSL implementation of both filters. Both implementations produce perturbed UVs for use with a texture sampler set to the Nearest Neighbor texture filtering mode, or to be used with integer Load instructions. Filter Importance Sampling is significantly simpler and uses less arithmetic, but this implementation requires the use of two random variables.

Listing 6: Gaussian Filter Importance Sampling.

```

float2 boxMullerTransform(float2 u)
{
    float2 r;
    float mag = sqrt(-2.0 * log(u.x));
    return mag * float2(cos(2.0 * PI * u.y), sin(2.0 * PI * u.y));
}

float2 FISGaussianUV(float2 uv, float2 dims,
                    float sigma, float2 u)
{
    float2 offset = sigma * boxMullerTransform(fract_part, u);
    return uv + offset / dims;
}

```

Listing 7: Gaussian Filter Discrete Sampling.

```

float2 discreteStochasticGaussianUV(float2 uv, float2 dims,
                                    float sigma, float u)
{
    float2 uv_full = uv * dims - 0.5;
    float2 left_top = floor(uv_full);
    float2 fract_part = uv_full - left_top;

    float inv_sigma_sq = 1.0f / (sigma*sigma);

    float weights_sum = 0.0f;
    float2 offset = float2(0.0f, 0.0f);

    #define FILTER_EXTENT 4
    #define FILTER_NEG_RANGE ((EXTENT-1)/2)
    #define FILTER_POS_RANGE (EXTENT-NEG_RANGE)
    for (int dy = -NEG_RANGE; dy < POS_RANGE; ++dy) {
        for (int dx = -NEG_RANGE; dx < POS_RANGE; ++dx) {
            float offset_sq = dot(float2(dx, dy) - fract_part,
                                float2(dx, dy) - fract_part);
            float w = exp(-0.5 * dist_sq * inv_sigma_sq);
            weights_sum += w;
            float p = w / weights_sum;
            if (u <= p) {
                offset = float2(dx, dy);
                u = u / p;
            } else {
                u = (u - p) / (1 - p);
            }
        }
    }
    return (left_top + offset + 0.5) / dims;
}

```

### Real-time anisotropic filtering

In Section 5.2 we described the stochastic anisotropic LOD for use with screen-space jittering. In Listing 8 we include the HLSL code for this computation.

Listing 8: Texture MIP computation used in real-time implementation.

```

float computeLodAniso(uint2 dim, float4 textureGrads,
                    float minLod, float maxLod,
                    float u, out float2 maxAxis)
{
    float dudx = dim.x * textureGrads.x;
    float dvdx = dim.x * textureGrads.y;
    float dudy = dim.y * textureGrads.z;
    float dvdy = dim.y * textureGrads.w;

    // Find min and max ellipse axis
    maxAxis = float2(dudy, dvdy);
    float2 minAxis = float2(dudx, dvdx);
    if (dot(minAxis, minAxis) > dot(maxAxis, maxAxis))
    {
        minAxis = float2(dudy, dvdy);
        maxAxis = float2(dudx, dvdx);
    }

    float minAxisLength = length(minAxis);
    float maxAxisLength = length(maxAxis);

    float maxAnisotropy = 64;

    if (minAxisLength > 0 &&
        (minAxisLength * maxAnisotropy) < maxAxisLength)
    {
        float scale = maxAxisLength / (minAxisLength * maxAnisotropy);
        minAxisLength *= scale;
    }
    return clamp(log2(minAxisLength) + (u - 0.5), minLod, maxLod);
}

```