Example-Based Microstructure Rendering with Constant Storage

ANONYMOUS AUTHOR(S)

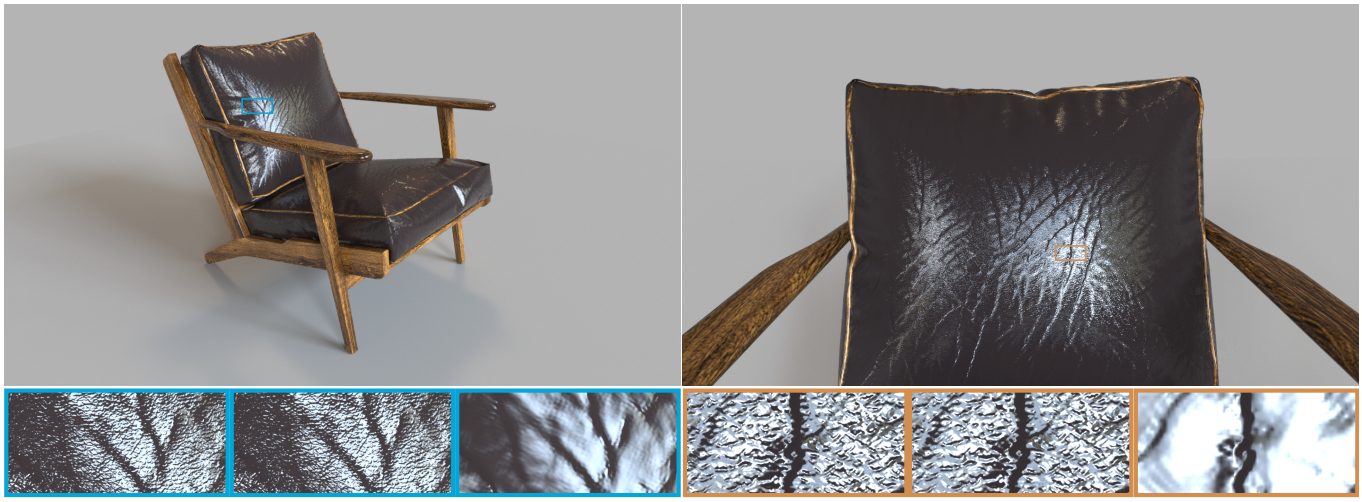


Fig. 1. Comparison between renderings produced with our method and the work of Yan et al. [2016]. The leather material is represented using two normal maps: a standard macro-level map, and a microstructure map, synthesized on-the-fly using our method from a small 512×512 example patch. Bottom: zoomed-in images rendered at higher resolution. For Yan et al. we use a $5K \times 5K$ normal map as input. The storage of our method is 35.0 MB, while the previous method costs 7896.0 MB to handle a similar level of detail without repetition. For comparison, we also show the rendering with a standard microfacet model, lacking the microstructure details.

Rendering glinty details from specular microstructure enhances the level of realism, but previous methods require heavy storage for the high-resolution height field or normal map and associated acceleration structures. In this paper, we aim at dynamically generating theoretically infinite microstructure, preventing obvious tiling artifacts, while achieving constant storage cost. Unlike traditional texture synthesis, our method supports arbitrary point and range queries, and is essentially generating the microstructure implicitly. Our method fits the widely used microfacet rendering framework with multiple importance sampling (MIS), replacing the commonly used microfacet normal distribution functions (NDFs) like GGX by a detailed local solution, with a small amount of runtime performance overhead.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Reflectance modeling**.

Additional Key Words and Phrases: Rendering, surface microstructure, glints, constant storage, procedural by-example noise

0730-0301/2019/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Anonymous Author(s). 2019. Example-Based Microstructure Rendering with Constant Storage. *ACM Trans. Graph.* 1, 1 (September 2019), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Microstructure rendering of glinty details [Yan et al. 2014] has brought a new level of realism to rendering specular highlights, a core effect in computer graphics. This method and subsequent work uses high-resolution normal maps to explicitly define every microfacet normal. However, very large normal maps are required to cover enough surface area without obvious repetition. For example, Yan et al. [2018] used a resolution of one micron per texel, which requires a $10K \times 10K$ normal map to cover just one square centimeter. Worse, hierarchical acceleration structures over these normal maps are needed for efficient pruning of non-contributing normals, making the storage problem even more severe. Designing normal maps of that size, which moreover need to allow for seamless tiling, requires additional tedious effort; normal map data of such size and quality is not easily available. These are key issues reducing the practicality of these methods.

Although texture synthesis methods are ubiquitous, few of them are suitable for the microstructure rendering task. Earlier image quilting techniques [Efros and Freeman 2001; Efros and Leung 1999;

Wei and Levoy 2000] and recent neural network based texture synthesis methods [Jetchev et al. 2016; Zhou et al. 2018] are synthesizing an image starting from a core example image. The problem is that, if we would like to query the synthesized image say at index $(100K, 100K)$, the method has to actually synthesize the image all the way up to that point, which is a clear violation of our constant storage need. We see that a dynamic *point query* is needed instead.

On the other hand, procedural noise methods, such as Perlin noise [Perlin 1985], Gabor noise [Lagae et al. 2009] and texton noise [Galerne et al. 2017], use a few parameters to control the appearance of a non-repeating noise function over an infinitely large space. By-example noise methods [Galerne et al. 2012; Gilet et al. 2012; Heitz and Neyret 2018] offer more artist controllability by providing an example texture and blending patches from it at different querying positions. These methods require no additional storage, and support on-the-fly point queries. However, for microstructure rendering, we need not only a normal map, but also the corresponding acceleration method for pruning non-contributing regions. Unfortunately, none of these methods are able to support min-max queries in an arbitrary range. Such a *range query* capability is a necessary component of a solution to our problem.

We present a method that implicitly generates the normal map along with a range query capability, so that it can directly fit into the microstructure rendering framework. Our method builds upon by-example noise methods to maximize artist controllability, and generates normal maps by blending patches from input examples. We support dynamic point queries and range queries on the implicit normal map generated using any by-example method, as long as the blending operation is *monotonically increasing* (as will be defined in Sec. 4.2). With our method, we are able to render microstructure with non-repetitive patterns, with constant storage cost and a small performance overhead over previous methods.

The rest of the paper is arranged as follows. In Sec. 2, we introduce previous work related to microstructure rendering and general procedural appearance. In Sec. 3, we briefly review the key ideas of microstructure rendering and by-example noise, after which we propose our insight and method framework. Next, in Sec. 4, we describe our point query (Sec. 4.2) and range query (Sec. 4.3) approaches, respectively. We illustrate our implementation details in Sec. 5 and compare our method with previous work in Sec. 6 in terms of overall quality, storage and performance.

2 RELATED WORK

In this section, we organize the related work into two basic categories. We first briefly review previous work on microstructure rendering and capture, then introduce related work on texture synthesis and general procedural appearance.

Microstructure rendering. Surface reflectance in computer graphics is typically described using statistical tools. More specifically, microfacet theory [Torrance and Sparrow 1967] uses smooth analytic functions such as Beckmann [Beckmann and Spizzichino 1987] and GGX [Walter et al. 2007] to model the distribution of surface normals. More recently, Yan et al. [2014] introduced the idea of using patch-local normal distribution functions ($\mathcal{P}$-NDFs) to accurately compute the spatially and directionally varying appearance from explicit specular microstructure such as bumps, brushes, scratches and metallic flakes. The microgeometry is defined using extremely high resolution normal maps. Yan et al. [2016] proposed a position-normal distribution method to accelerate computation, which was later extended to handle wave optics effects [Yan et al. 2018]. All these methods share a common problem with storage cost: the microstructures have to be defined at resolutions of $1 - 10$ microns per texel, which either requires very large textures (and associated acceleration structures) or leads to tiling artifacts.

Since explicit microstructure is costly to store, a series of methods were designed to model specific effects. Jakob et al. [Jakob et al. 2014] introduce a procedural BRDF that produces glitter effects from implicit mirror flake distributions without explicitly storing the underlying microstructure, but is not extensible to other kinds of microgeometry. Raymond et al. [2016] model surfaces as the mixture of a base surface and a collection of 1D scratches, later extended by Werner et al. [2017] for wave optics effects; these methods work well for scratches but do not support other appearances. Zirr et al. [2016] dynamically adds micro-level details to a predefined macro-scale BRDF, but is focused on real-time performance, not on accurate simulation of the appearance of a given microgeometry.

Detailed appearance measurement. Several approaches measure real-world samples and use the measured data to render, either directly or indirectly. Dong et al. [2015] used an interferometry device to acquire the microstructure of brushed metal, but they still use statistical reflectance models to fit the measured data for rendering. Other methods [Graham et al. 2013; Nagano et al. 2015; Nam et al. 2016] aim at measuring accurate heightfields; these could be used with glint rendering methods, but seamless extension of the data across larger surface areas remains a problem.

Texture synthesis. We aim at generating non-repeating appearance, which is also the goal of texture synthesis. Texture synthesis methods can be categorized into three different kinds. The first kind is *by expansion*: starting from a small texture, they dynamically "grow" a new larger texture. Representative work of this kind ranges from the classic image quilting methods [Efros and Freeman 2001; Efros and Leung 1999; Wei and Levoy 2000] to modern solutions using Generative Adversarial Networks (GANs) [Jetchev et al. 2016; Zhou et al. 2018]. These methods, however, are not applicable to our problem – to query the value at a specific location on the generated texture, the texture has to be actually generated from its original position to the query. This violates our goals of zero dynamic memory consumption and minimum performance overhead.

The second kind of related texture synthesis work is *tiling methods*, such as Wang tiles [Cohen et al. 2003; Wang 1961]. These methods first create small tiles from the input texture. These tiles are designed to allow seamless stitching to others, and are thus used as building blocks to generate larger textures. The tiling methods can support point queries; however, since the number of tiles is usually limited due to the difficulty of satisfying the seamless tiling property, repeated tiles are often visible as artifacts.

The third kind is *blending methods*, also known as by-example noise methods. They assume that any point on the resulting texture is blended from several patches from the input texture (example). Different blending methods of the example patches are possible, from simple linear blending (prone to "ghosting" artifacts), to more

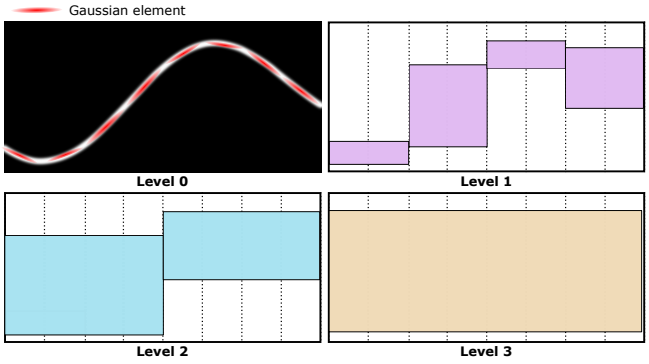


Fig. 2. Top left: Flatland visualization of a position-normal distribution, fitted using Gaussian elements. Others: Different levels of min-max hierarchy over the normal map bound the sets of normals within spatial ranges.

advanced variance preserving [Yu et al. 2011] and histogram preserving [Heitz and Neyret 2018] methods. These methods are procedural and we demonstrate that they can be adapted to our needs, by designing a suitable point and range query for normals and their Jacobians. Our method does not depend on a specific blending method, and we will show different appearances in Sec. 6 for different choices.

Other procedural appearance. Many efforts have focused on designing procedural noise functions, such as Perlin noise [Perlin 1985] and Gabor noise [Lagae et al. 2009], which give non-repeating values over the entire 2D or 3D space. The noise can be later thresholded and post processed in other ways to produce appearance variations that mimic terrain, rust, marble, etc. The noise functions often provide the functionality of point query (the value at any position) and approximate range average query (approximate average value in a given range) for anti-aliasing, but do not support range min-max query (exact minimum and maximum values in a given range), which is a crucial property needed by our method (Sec. 4.3).

3 BACKGROUND

3.1 Rendering details from microstructure

Our method builds upon the framework of microstructure rendering by Yan et al. [2016], where the microstructure is defined using a high resolution normal map. The normal map is bicubically interpolated, specifying a continuous function that returns a 2D normal $\mathbf{n}(\mathbf{u}) = (n_x, n_y)$ (dropping the implicit z-coordinate) for any given 2D texture coordinate $\mathbf{u} = (u, v)$.

During rendering, a spatial footprint $\mathcal{P}$ (i.e. coverage on the texture) can be approximated by the renderer as a Gaussian $G_{\mathcal{P}}$; this footprint can be as large as the pixel projection onto the surface, but is typically smaller (leaving some work to pixel multi-sampling). To evaluate the surface BRDF for the footprint $\mathcal{P}$, we need to query the distribution of the surface normals within the footprint, a.k.a. the patch normal distribution function ($\mathcal{P}$-NDF). To do that, for every position within $\mathcal{P}$, we check whether its normal is close enough to a query direction $\mathbf{s}$, where the closeness is defined using another Gaussian G_r specifying an "intrinsic roughness" of the microstructure. The query can be written formally as

$$D_{\mathcal{P}}(\mathbf{s}) = \int G_{\mathcal{P}}(\mathbf{u}) \mathcal{N}(\mathbf{u}, \mathbf{s}) \, \mathrm{d}\mathbf{u}, \tag{1}$$

where $\mathcal{N}(\mathbf{u}, \mathbf{s}) = G_r(\mathbf{n}(\mathbf{u}) - \mathbf{s})$ is a 4D function of $\mathbf{u}$ and $\mathbf{s}$ called position-normal distribution. Fig. 2 illustrates the position-normal distribution in a simplified flatland case (1D position, 1D normal). Using this definition, the resulting $D_{\mathcal{P}}$ becomes a replacement of the smooth NDF in the classic microfacet model.

Since the 4D position-normal distribution $\mathcal{N}(\mathbf{u}, \mathbf{s})$ is complicated, Yan et al.[2016] approximate it with a mixture of k Gaussian elements in 4D, such that $\mathcal{N}(\mathbf{u}, \mathbf{s}) \approx \sum_{i=1}^{k} G_i(\mathbf{u}, \mathbf{s})$. Each Gaussian element is defined as

$$G_i(\mathbf{u}, \mathbf{s}) = c_i \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}_i)^T \Sigma_i^{-1} (\mathbf{x} - \mathbf{x}_i)\right), \tag{2}$$

where c_i is a constant for normalization, $\mathbf{x} = (\mathbf{u}, \mathbf{s})^T$ is a 4D column vector, and Σ is the covariance matrix computed from the Jacobian of the normal $\mathbf{n}$ at the position $\mathbf{u}$; see Equation 3 in Yan et al.[2016].

The query of the $\mathcal{P}$-NDF at $\mathbf{s}$ thus becomes

$$D_{\mathcal{P}}(\mathbf{s}) \approx \sum_{i=1}^{k} \int G_{\mathcal{P}}(\mathbf{u}) G_i(\mathbf{u}, \mathbf{s}) \, \mathrm{d}\mathbf{u}, \tag{3}$$

where each term of the sum has been simplified to calculating the product integral of two 2D Gaussians (since the two dimensions of $\mathbf{s}$ are given as a query and are constant with respect to integration), which results in an analytical solution.

It has been demonstrated in Yan et al. [2016] that converting each texel of the normal map to a single Gaussian element gives good results in practice. Therefore, the number of Gaussian elements k is usually in the millions. To avoid calculating every Gaussian element's contribution to every query, Yan et al. [2014] build a min-max hierarchy over the normal map. The hierarchy is a tree structure, where each node stores the range of normals in its child nodes. With the hierarchy, a group of Gaussian elements can be pruned together if the bounding box of the normals if far from the query $\mathbf{s}$, meaning its contribution is negligible. The idea was later extended [Yan et al. 2016] to a 4D acceleration structure over both positions and normals, which is essentially multiple hierarchies for the Gaussian elements contributing to certain ranges of normals.

3.2 Procedural by-example noise

The key idea of by-example noise generation is to create a new image patch by blending multiple patches, picked up from different places on a given example. To make this process procedural, at any place on the synthesized noise, we need to know which patches are selected to blend. This is usually done by partitioning the infinite planar domain into regular regions (triangles, quads, etc.), where each region is associated with a unique random seed that is used to pick random patches from the example. Within each region, the blending weights vary linearly. We will describe these weights along with our choice of regions in more detail in Sec. 4.2.

During the noise generation, differences emerge in different choices of patch blending methods. Here, we introduce three representative

methods: linear blending, variance preserving blending [Yu et al. 2011], and histogram preserving blending [Heitz and Neyret 2018].

Linear blending is the most straightforward. It is a simple weighted average of all K inputs:

$$I_l = \sum_{i=1}^{K} w_i I_i, \tag{4}$$

where w_i is the weight of the i-th input I_i at a specific position.

Variance preserving blending builds on top of the linear blending:

$$I_v = (I_l - \bar{I})/W + \bar{I}, \tag{5}$$

where $W = \sqrt{\sum_{i=1}^{K} w_i^2}$ is the L2-norm of all the weights, and $\bar{I}$ is the (uniform-weighted) average of all the inputs.

Histogram preserving blending considers an additional operation and its inverse to the variance preserving blending. That is, it computes a mapping $\mathcal{G}$ that maps the histogram of the example into a 1D Gaussian distribution. There are three steps: apply the mapping $\mathcal{G}$ to "Gaussianize" the example, then perform variance preserving blending, and finally apply the inverse mapping of $\mathcal{G}$ to obtain the blended result. This can be written as

$$I_h = \mathcal{G}^{-1}[\mathcal{G}(I)_v]. \tag{6}$$

Determining which of these blending methods gives the best visual effects in which scenarios is beyond the scope of our paper. Our method works with all blending methods, as long as they satisfy a monotonicity property, as will be analyzed in Sec. 4.3.

4 PROCEDURAL MICROSTRUCTURE RENDERING

In this section, we describe our method to dynamically generate an infinitely large normal map from a given example input, and show how this method is applicable in microstructure rendering with minimum storage and performance overhead.

4.1 Problem analysis and motivation

Our goal is to generate non-repeating microstructure on an infinite domain. As discussed, we need a dynamic method, so that the value at any specific point can be looked up in constant time. The by-example blending methods solve the storage issue, but their naive application introduces heavy computational overhead: for a given pixel footprint $\mathcal{P}$, every Gaussian element inside it has to be computed, regardless of whether it contributes to the querying direction $\mathbf{s}$. Clearly, we need some pruning scheme.

In previous methods, an acceleration structure is built as a preprocess with the knowledge of the entire normal map. This is a significant obstacle – even if a normal map can be generated and queried on the fly, there are no existing algorithms to dynamically build an acceleration structure along with the normal map.

Moreover, to define a Gaussian element, we need the normal at its center as well as its covariance matrix, which is computed from its Jacobian. This immediately implies that during the lookups, we need to query not only a value, but also its derivative.

We describe our solutions to these two main problems. Before we proceed, we first define the terminology we are going to use:

- **example** – the given input normal map (not necessarily tileable),

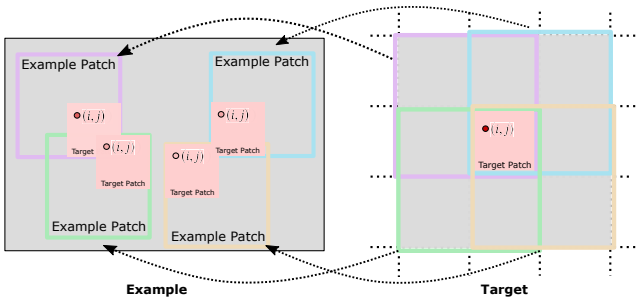


Fig. 3. The target patch (pink square) is the blended result of four different example patches (squares with different color). Each example patch has a deterministic random location in the example, which is specified by the target patch index. Each point in the target patch has one blending weight (represented as opacity of the small red dot, where higher opacity represents more weight) for each blending example patch. These blending weights vary across the target patch.

- **example patch** – a square patch from the example,
- **target** – the infinitely large planar domain on which we synthesize the microstructure, and
- **target patch** – a square patch on the target that is formed by blending several example patches.

We specify our goals formally as:

(1) **point query** – given a texture coordinate $\mathbf{u} \in [-\infty, \infty]^2$, query the normal map value $\mathbf{n}(\mathbf{u})$ and its Jacobian $\mathbf{J}(\mathbf{u})$ of the implicitly synthesized microstructure at a time complexity of $O(1)$, and

(2) **range query** – given a target patch $[\mathbf{u}_1, \mathbf{u}_2]$ (top-left and bottom right corners), query the interval that tightly bounds the values within the patch, $[\mathbf{n}_x^{\min}, \mathbf{n}_x^{\max}, \mathbf{n}_y^{\min}, \mathbf{n}_y^{\max}]$, also at a time complexity of $O(1)$.

In the next subsections, we will introduce the point query and the range query, then describe how these two operations are used together for fast $\mathcal{P}$-NDF queries during the rendering process.

4.2 Point query

The point query operation consists of two different parts. First, based on the point query's location, find the target patch it stays in and the corresponding example patches. Second, perform blending from different points on different example patches.

The first part is similar to the by-example texture synthesis methods introduced in Sec. 3. As Fig. 3 shows, we assume that the target is covered by overlapping example patches. The example patches are squares, and they overlap each other by half the edge length. Thus, any target patch is the blended result of four different example patches. The blending weights are bilinearly interpolated within the patch.

We partition the target into a square grid of target patches. Each target grid vertex is assigned a random number (seed) computed by hashing its index (i, j). This random number is used to locate a specific example patch. Based on the relative position of the point

query inside the target patch, we immediately know the corresponding position in each of the four example patches corresponding to the vertices of the patch.

Once we have found the normals at the four points on the example patches, the second step is to get the value $\mathbf{n}$ and its Jacobian $\mathbf{J}$ at $\mathbf{u}$ on the implicitly synthesized microstructure. It is straightforward to calculate the blended normal value $\mathbf{n}$. Since we already know the four positions on the example patches, we can immediately get $\mathbf{n}$ by applying the blending methods using Eqns. 4, 5 and 6 or any other methods.

We also need to compute the blended Jacobian $\mathbf{J}$. One immediate way is to perform the same point query of normals at four adjacent locations, then compute the Jacobian using central finite differences. However, this method is slower due to multiple queries, and depends on the fixed step size of the numerical differentiation. Instead, we use a fast and accurate solution. We start from the individual Jacobians on the four positions located on example patches. Since blending the normals essentially means that the normals go through a series of functions, it is straightforward to keep track of their individual Jacobians for each function using the chain rule, i.e.

$$\begin{aligned}\mathbf{J} &= \mathbf{J}[f_n f_{n-1} \dots f_2 f_1(\mathbf{n})] \\ &\leftarrow \mathbf{J}_{f_n} \cdot \mathbf{J}_{f_{n-1}} \cdot \dots \mathbf{J}_{f_2} \cdot \mathbf{J}_{f_1} \cdot \mathbf{J}(\mathbf{n}),\end{aligned} \tag{7}$$

where $\mathbf{J}_{f_i}$ is the Jacobian of the i-th function.

We note that previous by-example noise generation methods are all combinations of linear operations $\mathcal{F}_l = A\mathbf{n} + B$ (such as the weighting operation in linear blending) and non-linear operations $\mathcal{F}_{nl}$ (such as the Gaussianization function $\mathcal{G}$). When the normals go through linear operations, their Jacobians can be updated easily:

$$\mathbf{J}_{\mathcal{F}_l} = \mathbf{J}[\mathcal{F}_l(\mathbf{n})] = \mathbf{J}[A\mathbf{n} + B] = A\mathbf{J}. \tag{8}$$

For non-linear operations $\mathcal{F}_{nl}$, computing the Jacobian is also straightforward if $\mathcal{F}_{nl}$ is analytical. If not, these functions or mappings must have been precomputed and tabled, thus the same can be done with their derivatives. In the Appendix, we elaborate the calculation of Jacobians for commonly used blending methods.

4.3 Range query

As analyzed in Sec. 4.1, apart from the functionality to perform point queries, we also need to design a pruning scheme. Specifically, suppose a pixel footprint $\mathcal{P}$ is given on the microstructure. We would like to perform subdivision of the pixel footprint to prune areas with non-contributing normals as if an acceleration hierarchy has been provided (like in previous methods). Essentially, the pruning scheme needs to answer the question: is the queried normal value contained within the interval of normals of a given patch, i.e. between its minimum and maximum values?

Our insight is that we do not need to explicitly build any hierarchy, as long as we are able to answer the query for minimum and maximum normal values, given any positional range on the implicit microstructure. Since any target range is blended from four patches on the example texture, our range query problem becomes two sub-problems. First, querying the minimum and maximum values on the example texture. Second, computing the combined min-max interval as we blend the four query results from the example.

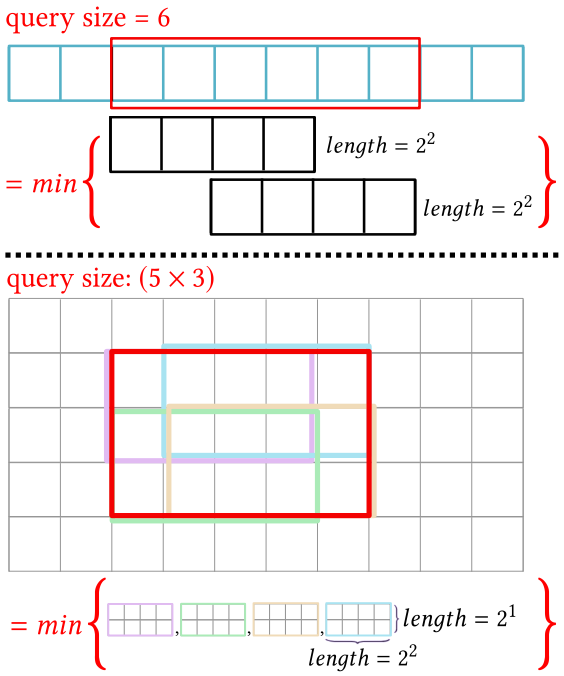


Fig. 4. Top: 1D version of RMQ. For an arbitrary 1D query (here, [2, 7], marked in red), we first find two precomputed range queries ([2, 5] and [4, 7]) with length 2^2; then the minimum of the query [2, 7] is the minimum of the two precomputed minima. Bottom: 2D version of RMQ. For an arbitrary 2D query (here, [2, 0] to [6, 3], marked in red rectangle)), we find four precomputed range queries: [2, 1] to [5, 2] (light purple), [3, 1] to [6, 2] (light blue), [2, 2] to [5, 3] (light green) and [3, 2] to [6, 3] (light yellow), with size $2^2 \times 2^1$. The minimum of the query is the minimum of the four precomputed minima.

4.4 Range minimum query

The first task is a classic algorithmic problem known as the Range Minimum Query (RMQ). In 1D, the RMQ problem has been proven to be solvable within $O(1)$ runtime and $O(n \log n)$ precomputation time and storage, using the sparse-table algorithm [Bender and Farach-Colton 2000]. As Fig. 4 shows, the key idea is to precompute the answers to all possible range queries of length 2^K, where K is a positive integer. For a general query from the i-th element to the j-th element, it takes constant time to find two precomputed range queries, such that (1) one starts at i and the other ends at j, (2) they are of the same length and (3) their union covers the entire range $[i, j]$. Then the minimum of the general query $[i, j]$ is the minimum of the two precomputed minima. Note this algorithm gives the exact minimum (not a conservative approximation); the same approach can be used for a range maximum query.

Extending this 1D algorithm to arbitrary 2D queries is straightforward. As illustrated in Fig. 4, we precompute the answers to all possible 2D range queries of sizes $2^{K_1} \times 2^{K_2}$, where K_1 and K_2 are both positive integers. For an arbitrary query, we can immediately locate four precomputed range queries of the same size and covers the entire query range, each staying in one of the fours corners of

Algorithm 1 Precomputation of RMQ table.

Input:
$\mathcal{N}(\mathbf{u}, \mathbf{s})$: Gausian mixture of example
n: example width / height

Output: $\mathcal{T}$ = RMQ precomputed table

function QUERYEXAM($\mathbf{u}_0, \mathbf{u}_1$)
  if $\mathbf{u}_0$.x == $\mathbf{u}_1$.x **then**
    $\mathbf{n}^{\min}, \mathbf{n}^{\max} \leftarrow \mathcal{N}(\mathbf{u}_0, \mathbf{s})$
  else
    $\mathbf{u}_m \leftarrow \lceil(\mathbf{u}_0 + \mathbf{u}_1) \times 0.5\rceil$
    $\mathbf{u}_{m-1} \leftarrow \mathbf{u}_m - 1$
    $\mathbf{u}_{m,0} \leftarrow$ vec2($\mathbf{u}_0$.x, $\mathbf{u}_m$.y)
    $\mathbf{u}_{m,1} \leftarrow$ vec2($\mathbf{u}_{m-1}$.x, $\mathbf{u}_1$.y)
    $\mathbf{u}_{m,2} \leftarrow$ vec2($\mathbf{u}_m$.x, $\mathbf{u}_0$.y)
    $\mathbf{u}_{m,3} \leftarrow$ vec2($\mathbf{u}_1$.x, $\mathbf{u}_{m-1}$.y)
    $\mathbf{n}_0^{\min}, \mathbf{n}_0^{\max} \leftarrow$ QUERYEXAM($\mathbf{u}_0, \mathbf{u}_{m-1}$)
    $\mathbf{n}_1^{\min}, \mathbf{n}_1^{\max} \leftarrow$ QUERYEXAM($\mathbf{u}_{m,0}, \mathbf{u}_{m,1}$)
    $\mathbf{n}_2^{\min}, \mathbf{n}_2^{\max} \leftarrow$ QUERYEXAM($\mathbf{u}_{m,2}, \mathbf{u}_{m,3}$)
    $\mathbf{n}_3^{\min}, \mathbf{n}_3^{\max} \leftarrow$ QUERYEXAM($\mathbf{u}_m, \mathbf{u}_1$)
    $\mathbf{n}^{\min} \leftarrow \min(\mathbf{n}_0^{\min}, \mathbf{n}_1^{\min}, \mathbf{n}_2^{\min}, \mathbf{n}_3^{\min})$
    $\mathbf{n}^{\max} \leftarrow \max(\mathbf{n}_0^{\max}, \mathbf{n}_1^{\max}, \mathbf{n}_2^{\max}, \mathbf{n}_3^{\max})$
    $k \leftarrow \log_2(\mathbf{u}_1.\text{x} - \mathbf{u}_0.\text{x} + 1)$
    $\mathcal{T}[k][\mathbf{u}_0.\text{x}][\mathbf{u}_0.\text{y}] \leftarrow$ packingTo64Bit($\mathbf{n}^{\min}, \mathbf{n}^{\max}$)
  end if
  return $\mathbf{n}^{\min}, \mathbf{n}^{\max}$
end function

function RMQPRECOMPUTATION
  $l = n \div 4$
  // **size of target patch**
  $k \leftarrow \log_2(l)$
  //**logarithm of target patch size**
  for all $u < n$ **do**
    for all $v < n$ **do**
      $\mathbf{u}_0 \leftarrow$ vec2(u, v)
      $\mathbf{u}_1 \leftarrow \mathbf{u}_0 +$ vec2($l - 1$)
      QUERYEXAM($\mathbf{u}_0, \mathbf{u}_1$)
      //**compute the min. and max. normal recursively**
    end for
  end for
  return $\mathcal{T}$
end function

the query range. The minimum of the general query is the minimum of the four precomputed minima. Then the 2D RMQ problem can be solved within $O(1)$ runtime and $O(mn \log m \log n)$ precomputation time and storage, where $m \times n$ is the resolution of the 2D array.

Despite its fast performance, the additional $O(\log m \log n)$ storage is still costly. We further decrease the storage cost using the insight that, if the shapes of the range queries are restricted to be squares rather than arbitrary rectangles, we are able to omit one dimension from the precomputed data. That is, the precomputed data now

Algorithm 2 Query the RMQ table.

Input:
$\mathcal{T}$ = RMQ precomputed table
$\mathbf{u}_{\min}$ = min uv value of the query
$\mathbf{u}_{\max}$ = max uv value of the query
Output:
$\mathbf{n}^{\min}$ = min normal of the query
$\mathbf{n}^{\max}$ = max normal of the query

$h \leftarrow \mathbf{u}_{\max}.\text{x} - \mathbf{u}_{\min}.\text{x}$
$k \leftarrow \log_2(h + 1)$
$N_{packed} \leftarrow \mathcal{T}[k][\mathbf{u}_{\min}.\text{x}][\mathbf{u}_{\min}.\text{y}]$
//**read the RMQ precomputation table**
$\mathbf{n}^{\min}, \mathbf{n}^{\max} \leftarrow$ unpackingFrom64Bit(N_{packed})
return $\mathbf{n}^{\min}, \mathbf{n}^{\max}$

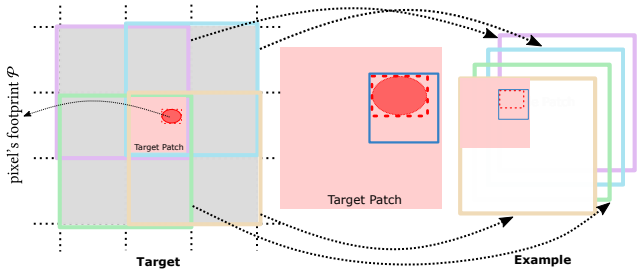


Fig. 5. We perform the range query in the target patch instead of the footprint. We find the tightest square bounding the pixel's footprint $\mathcal{P}$, and start the traversal from the square (blue square).

costs only $O(n^2 \log n)$, where $n \times n$ is the resolution of the example texture and the $\log n$ term accounts for different side lengths of the precomputed queries.

To guarantee square range queries, we consider each target patch that intersects the pixel footprint (see Fig. 5). Since each target patch is perfectly square, any query from a regularly subdivided target patch will also be square. We will describe this in more detail along with our traversal scheme in the next subsection. We further know that the target patch itself has a size of 2^K, so that we are able to find one precomputed square that covers it exactly (see details in Sec. 5).

Note that the precomputed data in our 2D sparse-table algorithm is different from a mip-map style tree structure. Taking the 1D case as an example, a precomputed query in a tree structure must start at multiples of its length. However, the sparse-table algorithm precomputes for all possible starting points. The difference indicates why a tree structure only supports range queries within $O(\log n)$ time.

With these tools, we are able to solve the first sub-problem to perform a range query within each example patch. We list the pseudocode for algorithms that build our optimized 2D sparse-table in Algorithm 1 and the pseudocode to use it for our range queries in Algorithm 2. The next step is to combine the four queried min-max

intervals into one for the target patch, as the normals within these intervals on the example patches are blended.

4.5 Blending range queries

The problem of accurately combining the range queries from the patches being blended still remains. The union of the four queries may not be a conservative bound, since more advanced blending methods may not satisfy the convex-hull property; that is, the blended minimum value could be smaller than any of the input minima.

Our goal is to correctly bound the blended min-max intervals, and make them as tight as possible. To achieve this, we find that all the operations in the by-example noise methods we use, whether linear or non-linear, are *monotonically increasing* with respect to the values being blended. A 1D function f is monotonically increasing if it satisfies the property

$$\text{if } x_1 \le x_2, \text{ then } f(x_1) \le f(x_2). \tag{9}$$

A multi-dimensional function is monotonically increasing if it satisfies the property with respect to every input variable.

One can verify that this monotonically increasing property holds for all our blending methods. Elementary operations used in the methods, such as additions/subtractions and multiplications/divisions by positive values, as well as linear operations with positive weights, clearly satisfy the property. The Gaussianize operation in histogram-preserving blending is essentially equivalent to 1D optimal transport [Monge 1781] (recall that we blend the x and y components of normals separately); therefore, it is also guaranteed to be monotonically increasing [Bonneel et al. 2011].

The monotonically increasing property allows us to apply the same blending method to the endpoints of the min-max intervals from the four source example locations being blended, producing a guaranteed conservative bound. For example, if values $x_1, \cdots, x_4$ are bounded from above by $u_1, \cdots, u_4$ respectively, then the blend of the former will be upper-bounded by the same blend of the latter.

However, one additional issue is that the blending weights themselves can vary over the queried range. This means we also need to bound the range of blending weights over a query region. This is straightforward to do within the traversal scheme that uses our range query, which will be introduced in the next subsection. The resulting min-max interval of the blended normal is no longer guaranteed to be tight, but is always correct (conservatively bounding) and works efficiently in practice.

4.6 Implicit hierarchy traversal

As mentioned in subsection 4.3, we subdivide from the target patch instead of the pixel's footprint. Given the pixel's footprint $\mathcal{P}$, we find its overlap with target patch, and then get the tightest square bounding with length as a power of two, and start the traversal from the square (see Alg. 3). The square is subdivided in smaller squares until their content overlaps entirely with the bounding box (red dash rectangle) of the footprint.

For all these squares, if there are more than one texels, we perform a range query to get their min-max normal interval. If the half vector locates in the normal range, we subdivide the square into four small squares and continue the traversal. If the half vector is not included in the range, then all the texels in the square are discarded. The traversal continues until there is only one texel in the square. If the half vector is located in the min-max interval of the texel, we blend the normal and the Jacobian to get a Gaussian element. If there is an intersection with the pixel's footprint and the Gaussian element, we gather the contribution from the Gaussian element.

The min-max interval for the top levels might not be tight in the beginning. However, as the traversal proceeds until lower levels, min-max interval becomes more and more accurate, and converges to the tightest boundary on the finest level.

Algorithm 3 Build a hierarchy for a given foot print.

Input:
$\mathcal{T}$ = RMQ precomputed table
$\mathcal{N}(\mathbf{u}, \mathbf{s})$ = GMM of example
$\mathbf{s}$ = half vector
$\mathbf{u}_{\min}$ = min uv value of a pixel's footprint $\mathcal{P}$
$\mathbf{u}_{\max}$ = max uv value of a pixel's footprint $\mathcal{P}$
Output:
D = contribution to the pixel's footprint

```
function TRAVSQUARE(u_0, u_1)
    N ← u_1.x − u_0.x
    inside ← true
    if N > 0 then
        inside ←include(T(u_0, u_1), s)
    else
        inside ←include(N(u_0, s), s)
    end if
    if !inside then
        return 0;
    end if
    if N == 0 then
        n, J ← blendNormalJacobian(u_0)
        D+ = gaussianContribution(n, J)
    else
        u_0^c[4], u_1^c[4] ← subdivide(u_0, u_1)
        for all i < 4 do
            D += TRAVSQUARE(u_0^c[i], u_1^c[i])
        end for
    end if
return D
end function
```

```
function TRAVERSAL(u_min, u_max)
    x ← u_max.x − u_min.x
    y ← u_max.y − u_min.y
    l^2_max ← 2^int(log_2(max(x, y)))+1
    u'_min ← u_min
    u'_max ← u_min + l^2_max
    //get the starting square for traversal
    return TRAVSQUARE(u'_min, u'_max)
end function
```

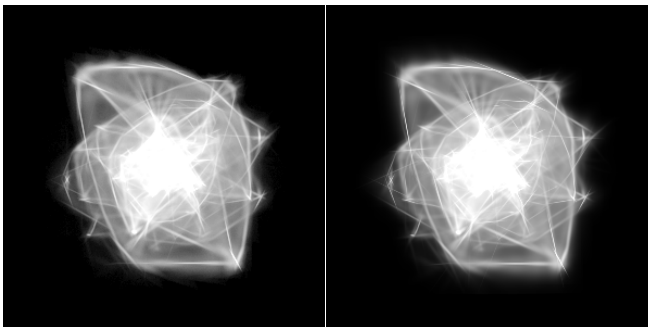


Fig. 6. Normal distribution function visualizations with binning the importance sampled directions and with per-pixel evaluation provide closely matching results.

5 IMPLEMENTATION DETAILS

5.1 Example range precomputation and packing

Given an example range, the minimum and maximum normals need to be quickly computed. In our implementation, we use a 3D precomputed table to represent the minimum and maximum normal: two dimensions represent corner location in the example, and the third dimension represents the logarithm of query size. The table is computed recursively, starting from the finest level.

To save memory, we further pack four values (two for minimum normal and two for maximum normal) into 64 bits. The four values are normalized into $[0, 1]$, converted into 16-bit integers and then combined into a 64 bit value. In the end, our precomputed range query table is compact, about 14 MB for a 512×512 example texture.

5.2 Footprint coverage and multiple target patches

Each footprint might cover multiple target patches. We bound the footprint with each covering target patch and generate several small sub-footprints. We call these sub-footprints footprints, for simplicity.

5.3 Range query

In the range query, we query and blend the min-max normal intervals from four different example patches. To locate these example patches, we first need their size. We always use square examples of edge length 2^K, and assume that example patches are also squares with an edge length exactly half of the example. Note that neither of these two choices are required. Our RMQ algorithms will work as long as the example patches are guaranteed to be squares of edge length 2^K.

Then we determine the individual starting positions of the example patches on the example. This is the same for both point query and range query. We start from the random seed associated with each target patch, and uniformly select four random positions on the target. To guarantee that no example patch will extend outside of the example, valid coordinates of the starting positions are constrained to be smaller than the difference between the edge length of the example and the edge length of an example patch.

With the example patches selected, given a range inside a target patch, we immediately know its four corresponding ranges in the example. We compute their min-max normal intervals for each of the four ranges using the RMQ precomputed table (see Section 4.3).

The four min-max normal from the four example patched are then blended (see Section 4.3).

5.4 Importance sampling

For a given shading point, we get its corresponding uv-coordinate, and then we find the normal and Jacobian of the four texels around it to get four GMM elements. By picking an element proportional to its contribution to the footprint, then picking a normal from that element, we get the sampled direction. We validate the correctness of our importance sampling in Figure 6.

6 RESULTS AND COMPARISON

We have implemented our algorithm inside the Mitsuba renderer [Jakob 2010]. We compared our algorithm against Yan et al. [2016] for quality validation. All timings in this section are measured on a 2.20GHz Intel i7 (40 cores) with 32 GB of main memory. Unless otherwise specified, all timings correspond to pictures with 1280×720 pixels, except the BentQuad scene with 512×512. In all of our results, we use histogram preserving blending [Heitz and Neyret 2018] as the blending method, except in Figure 10.

In Table 1, we report all the scene settings, computation time and memory costs for our test scenes. Figure 6 illustrates the NDF images with sampling (binning) and with evaluation. The images are the equivalent, which confirms the correctness of our method.

Chair scene. This scene shows a chair with two leather pillows (75cm wide), rendered using environment lighting. The leather pillows have a macro-level normal map and detailed microstructure bumps. The macro map covers 75cm × 75cm. The micro example normal map with resolution 512×512 covers 75mm × 75mm. In Yan et al. [2016], we synthesize an equivalent large normal map ($5K \times 5K$) offline and use it for rendering. Compared to Yan et al. [2016], our method produces exactly the same results, with only a fraction (0.44%) of memory cost. Regarding the time cost, our method has a small overhead (13%).

Macbook. This scene shows a laptop with a roughened aluminum matte finish. It is rendered using a point light and environment lighting. The laptop is about 30cm wide. The input example 512×512 covers 3mm × 3mm. In Yan et al. [2016], we use the same input texture and tile it. In Figure 9, We can observe the obvious repeating patterns in the results of Yan et al. [2016]. In Table 1, we report the memory cost of both methods. Our method costs 35 MB, while Yan et al. [2016] costs 62 MB for the same (small) normal map, as our range query tables are slightly more space-efficient than their hierarchy.

In Figure 10, we compare the results of our method with different blending methods: linear, variance preserving and histogram preserving blending. Our method does not rely on any specific blending method. We observe both variance preserving and histogram preserving blending provide acceptable quality.

Kettle scene. Figure 8 illustrates a Kettle with brushed metal on the body under two small area lights and environment lighting. The kettle is about 30cm high. The input brushed metal normal map with 512×512 resolution covers about 9mm × 9mm. For Yan et

Scene	#Triangle	Intrinsic Rough.	Normal map (ours)		Normal map (Yan[2016])		Time (min.)		Memory (MB)	
	K		Res.	Tile	Res.	Tile	Ours	Yan[2016]	Ours	Yan[2016]
Chair	303.0	0.01	512^2	10	$5K^2$	1	3.93	3.48	35.0	7896.0
Macbook	18.4	0.005	512^2	100	512^2	100	6.71	4.38	35.0	62.0
Kettle	175.3	0.005	512^2	32	$2K^2$	8	3.90	3.50	35.0	1119.9
BentQuad	19.6	0.005	$1K^2$	2	$1K^2$	2	1.26	–	148.0	–
Shoe	13.3	0.01	512^2	20	512^2	20	3.43	3.33	35.0	62.0

Table 1. Scene settings, computation time and memory costs for our test scenes. #Triangle is the count of triangles in the scene. Intrinsic Rough. presents the intrinsic roughness of the material. Normal map (ours) and Normal map (Yan[2016]) represent the input normal map setting for our method and Yan et al. [2016].

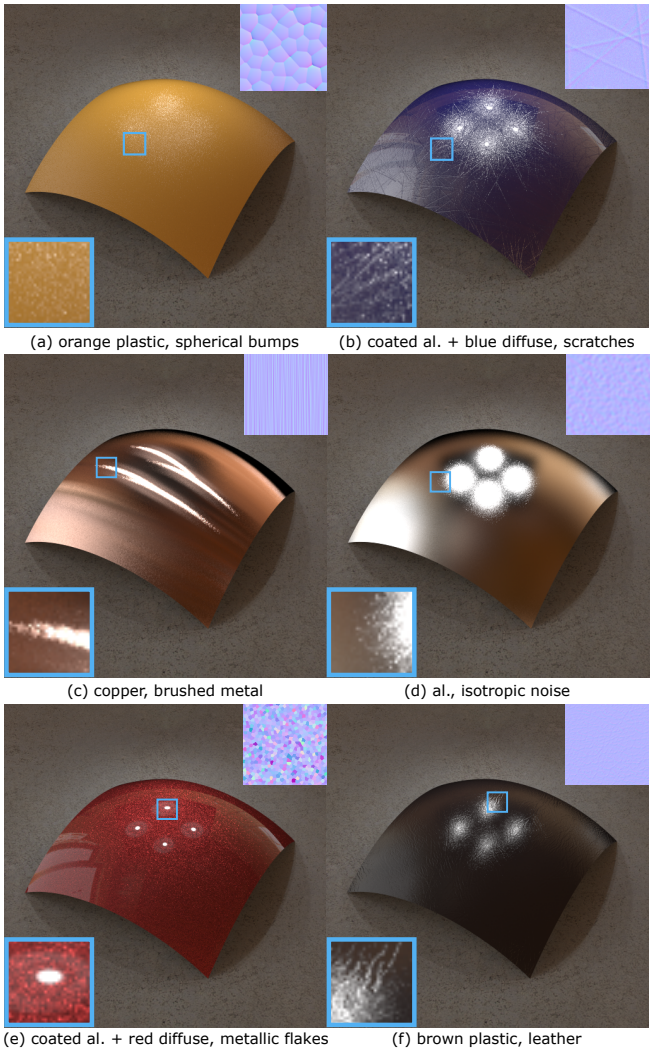


Fig. 7. Rendered results of different normal maps.

al. [2016], we used a $2K \times 2K$ tillable input texture and tiled it. There are no visible differences between our results and those of Yan et

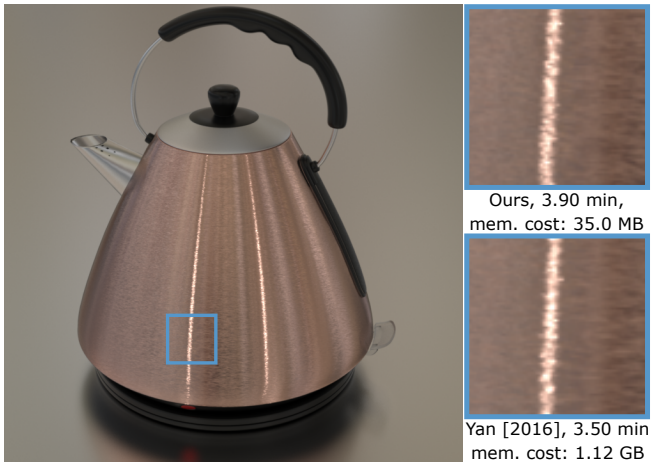


Fig. 8. Comparison between our method and Yan et al. [2016] with a tiled texture on the Kettle Scene. Normal map: brushed metal. The results are similar, but the memory cost of our method in only a small fraction (about 3%) of theirs.

al. [2016] (see Figure 8). The memory cost for our method is only a small fraction (about 3 %) of theirs.

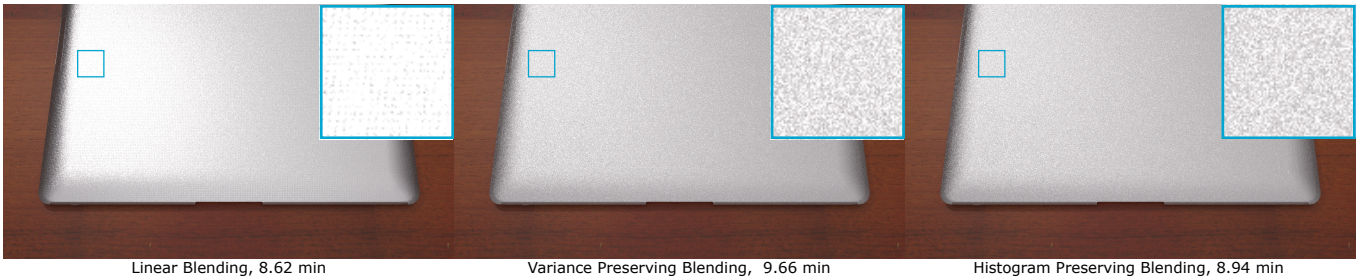
BentQuad scene. Figures 7 and 11 show a simple scene with a 5cm × 5cm bent quad with a scratched normal map illuminated by a textured light. The resolution of input isotropic noise normal map is $1K \times 1K$, and covers 2.5cm × 2.5cm. In Figure 11 we show the results with BRDF sampling only, evaluation only and their combination under the multiple importance sampling framework. We also show the result with environment lighting in the right image.

In Figure 7, we show the results of BentQuad with different BRDF types with different normal maps used as examples.

Shoe scene. This scene shows a shoe with coated metallic flakes under environment lighting. We found that no existing blending method works well with flakes. However, we can easily fix this by using our method without blending (choosing each point from a single patch); the rest of the framework is unchanged. The blended normal maps will have visible seams. However, since every flake has a constant normal, its Jacobian is always zero and does not have to be re-computed from a normal map. Thus, discontinuities in the blended normal maps will not introduce any discontinuous artifacts during rendering. As shown in Figure 12, even though without

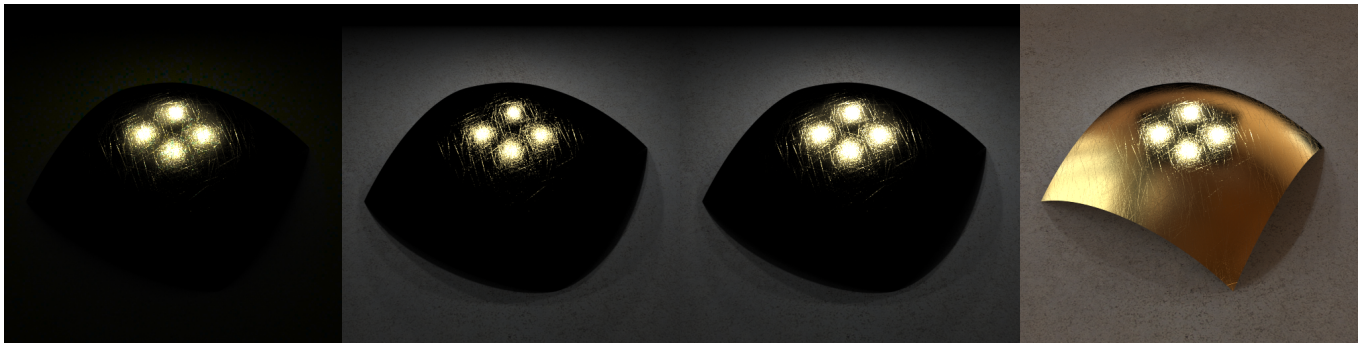
Ours, 6.71 min Yan et al. [2016], 4.38 min

Fig. 9. Comparison between our method and Yan et al. [2016] with a tiled texture on the Macbook Scene. The repeated pattern is visible in Yan et al. [2016]. Normal map: isotropic noise.



Linear Blending, 8.62 min Variance Preserving Blending, 9.66 min Histogram Preserving Blending, 8.94 min

Fig. 10. Comparison between different blending method (linear, variance preserving and histogram preserving on the Macbook Scene. Normal map: isotropic noise.Linear blending has artifacts issues. Both variance preserving and histogram preserving blending provide acceptable quality.



(a) sampling (b) evaluation (c) combined (d) with envmap, 1,26 min

Fig. 11. Our material model can be used inside a standard BRDF sampling/evaluation framework with multiple importance sampling. BRDF sampling alone (a) captures only a small fraction of scratches. Light sampling (b) captures illumination from the high-intensity parts of the HDR light texture onto the scratches. The combined result (c) has the benefits of both estimators. (d) shows the result with extra environment lighting.

blending the method produces boundary artifacts in the synthesized normal maps, this problem does not exist in the rendering results.

7 CONCLUSION AND FUTURE WORK

We have presented a method that allows rendering of specular glints from an arbitrarily large, non-repeating synthesized microstructure. Our method has constant storage cost and a small performance overhead. By designing point query and range query schemes for general by-example texture synthesis methods, we are able to dynamically and implicitly generate an infinite normal map, together with the required Jacobians and range queries. We demonstrate

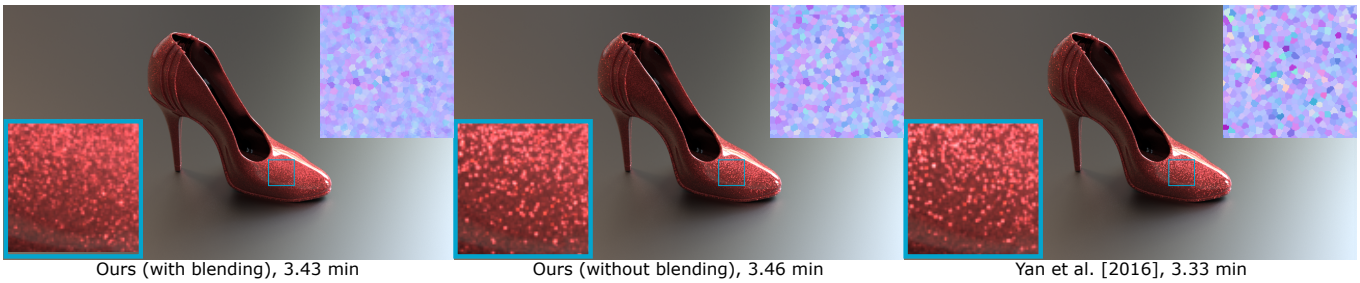


Fig. 12. Comparison between our method (with blending), our method (without blending) and Yan et al. [2016] with a tiled texture on the Shoe scene. The normal map depicts metallic flakes as small constant regions. Our method (with blending) has an over-smoothing issue on the metallic flake normal map. However, using synthesis with no blending fixes this issue: while it produces normal maps with boundary artifacts, it will not affect the rendering results, both theoretically and practically.

that our method produces plausible and controllable details, supports inputs from any source, and fits into a Monte Carlo rendering framework with multiple importance sampling. Our method can be treated as a standard BRDF, much like the common microfacet BRDF but replacing its smooth NDF with our solution.

In the future, it would be interesting to optimize our method for real-time implementation on GPUs. Extending our method for rendering with wave optics could also be a worthwhile direction.

REFERENCES

Petr Beckmann and Andre Spizzichino. 1987. The scattering of electromagnetic waves from rough surfaces. *Norwood, MA, Artech House, Inc., 1987, 511 p.* (1987).

Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN '00)*. 88–94.

Nicolas Bonneel, Michiel van de Panne, Sylvain Paris, and Wolfgang Heidrich. 2011. Displacement Interpolation Using Lagrangian Mass Transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2011)* 30, 6 (Dec. 2011), 158:1–158:12.

Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. Wang Tiles for Image and Texture Generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013)* 22, 3 (2003), 287–294.

Zhao Dong, Bruce Walter, Steve Marschner, and Donald P. Greenberg. 2015. Predicting Appearance from Measured Microgeometry of Metal Surfaces. *ACM Transactions on Graphics* 35, 1 (2015), 9:1–9:13.

Alexei A. Efros and William T. Freeman. 2001. Image Quilting for Texture Synthesis and Transfer. *Proceedings of SIGGRAPH 2001* (August 2001), 341–346.

Alexei A. Efros and Thomas K. Leung. 1999. Texture Synthesis by Non-parametric Sampling. In *IEEE International Conference on Computer Vision*. Corfu, Greece, 1033–1038.

Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. 2012. Gabor Noise by Example. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)* 31, 4 (2012), 73:1–73:9.

Bruno Galerne, Arthur Leclaire, and Lionel Moisan. 2017. Texton Noise. *Computer Graphics Forum* 36, 8 (2017), 205–218.

Guillaume Gilet, Jean-Michel Dischler, and Djamchid Ghazanfarpour. 2012. Multiple Kernels Noise for Improved Procedural Texturing. *Vis. Comput.* 28, 6-8 (2012), 679–689.

Paul Graham, Borom Tunwattanapong, Jay Busch, Xueming Yu, Andrew Jones, Paul Debevec, and Abhijeet Ghosh. 2013. Measurement-based synthesis of facial microgeometry. *Computer Graphics Forum* 32, 2pt3 (2013), 335–344.

Eric Heitz and Fabrice Neyret. 2018. High-Performance By-Example Noise Using a Histogram-Preserving Blending Operator. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2 (2018), 31:1–31:25.

Wenzel Jakob. 2010. Mitsuba Renderer. http://www.mitsuba-renderer.org/.

Wenzel Jakob, Miloš Hašan, Ling-Qi Yan, Jason Lawrence, Ravi Ramamoorthi, and Steve Marschner. 2014. Discrete Stochastic Microfacet Models. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33, 4 (2014).

Nikolay Jetchev, Urs Bergmann, and Roland Vollgraf. 2016. Texture Synthesis with Spatial Generative Adversarial Networks. *CoRR* abs/1611.08207 (2016). arXiv:1611.08207 http://arxiv.org/abs/1611.08207

Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. 2009. Procedural Noise using Sparse Gabor Convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)* 28, 3 (2009), 54–64.

Gaspard Monge. 1781. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences de Paris* (1781).

Koki Nagano, Graham Fyffe, Oleg Alexander, Jernej Barbiç, Hao Li, Abhijeet Ghosh, and Paul E Debevec. 2015. Skin microstructure deformation with displacement map convolution. *ACM Transactions on Graphics* 34, 4 (2015), 109–1.

Giljoo Nam, Joo Ho Lee, Hongzhi Wu, Diego Gutierrez, and Min H Kim. 2016. Simultaneous acquisition of microscale reflectance and normals. *ACM Transactions on Graphics* 35, 6 (2016), 185–1.

Ken Perlin. 1985. An Image Synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '85)*. 287–296.

Boris Raymond, Gaël Guennebaud, and Pascal Barla. 2016. Multi-scale Rendering of Scratched Materials Using a Structured SV-BRDF Model. *ACM Transactions on Graphics* 35, 4 (2016), 57:1–57:11.

Kenneth E Torrance and Ephraim M Sparrow. 1967. Theory for off-specular reflection from roughened surfaces. *Josa* 57, 9 (1967), 1105–1114.

Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. 2007. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*. 195–206.

Hao Wang. 1961. Proving theorems by pattern recognition - II. *The Bell System Technical Journal* 40, 1 (1961), 1–41.

Li-Yi Wei and Marc Levoy. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 479–488.

Sebastian Werner, Zdravko Velinov, Wenzel Jakob, and Matthias B. Hullin. 2017. Scratch iridescence: Wave-optical rendering of diffractive surface structure. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2017)* 36, 6 (2017), 207:1–207:14.

Ling-Qi Yan, Miloš Hašan, Wenzel Jakob, Jason Lawrence, Steve Marschner, and Ravi Ramamoorthi. 2014. Rendering Glints on High-Resolution Normal-Mapped Specular Surfaces. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2014)* 33, 4 (2014).

Ling-Qi Yan, Miloš Hašan, Steve Marschner, and Ravi Ramamoorthi. 2016. Position-Normal Distributions for Efficient Rendering of Specular Microstructure. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2016)* 35, 4 (2016).

Ling-Qi Yan, Miloš Hašan, Bruce Walter, Steve Marschner, and Ravi Ramamoorthi. 2018. Rendering Specular Microgeometry with Wave Optics. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2018)* 37, 4 (2018).

Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. 2011. Lagrangian Texture Advection: Preserving Both Spectrum and Velocity Field. *IEEE Transactions on Visualization and Computer Graphics* 17, 11 (2011), 1612–1623.

Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. 2018. Non-stationary texture synthesis by adversarial expansion. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 49.

Tobias Zirr and Anton S Kaplanyan. 2016. Real-time rendering of procedural multiscale materials. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 139–148.

A APPENDIX

We use chain rule to compute the blended Jacobian $\mathbf{J}$. Starting from the individual Jacobians on the four positions located on example patches, we keep track of their individual Jacobians for each function

using chain rule (Equation 7). We consider three different blending methods: linear, variance preserving and histogram preserving blending.

Linear Blending. Since the normals in linear blending go through a set of linear operations (see Equation 4), the formula of the blended Jacobian with chain rule is:

$$\mathbf{J}_l = \sum w_i \mathbf{J}_n, \tag{10}$$

where w_i represents the weight of each example during blending, and $\mathbf{J}_i$ is the Jacobian associated with normal $\mathbf{n}$ of the i_{th} input.

Variance Preserving Blending. In the variance preserving blending, extra linear operations are performed (see Equation 5) after linear blending, resulting in:

$$\mathbf{J}_v = \mathbf{J}_l / W, \tag{11}$$

where $W = \sqrt{\sum_{i=1}^{K} w_i^2}$ is the L2-norm of all the weights.

Histogram Preserving Blending. Additional "Gaussianize" ($\mathcal{G}$) and "inverse Gaussianize" ($\mathcal{G}^{-1'}$) operations are performed in histogram preserving blending, which are non-linear operations. Since these two operations are precomputed in a table, we compute their derivatives along with the precomputed values. During blending, the Jacobian is then computed as follows:

$$\mathbf{J}_h = \mathcal{G}^{-1'}\left[\mathcal{G}(\mathbf{n})_v\right] \cdot \left[\mathcal{G}'(\mathbf{n}) \otimes \mathbf{J}\right]_v, \tag{12}$$

where $\mathcal{G}^{-1'}$ and $\mathcal{G}'$ represent the inverse Gaussianize and Gaussianize derivatives, and they should be diagonal matrices, since we blend the two components of normal, $\mathbf{n}_x$ and $\mathbf{n}_y$, separately. $\mathcal{G}(\mathbf{n})_v$ is the variance blended normal (see Equation 11). $\left[\mathcal{G}'(\mathbf{n}) \otimes \mathbf{J}\right]_v$ represents performing the "Gaussianize" operation on the input Jacobian, which is an element-wise matrix multiplication, and then followed by a variance preserving operation.