Authors Draft
Final version to appear in Computer Graphics Forum
(Proc. High Performance Graphics 2025)

COMPUTER GRAPHICS *forum*
Volume 44 (2025), Number 8

Triangle Rejection Sampling for Density-Equipped Meshes on GPU

Jérémie Schertzer [1,2] Theo Thonat [1] Tamy Boubekeur [1]

[1]Adobe , [2]Mongolia International University

Figure 1: *Spatially-varying fiber distribution using our method over a parametrized 3D surface mesh (top left), driven by a density map applied over the surface (bottom left). 70M grass strand locations are computed in less than 10ms by our new sampling method, enabling interactive design of their distribution.*

Abstract

Non-uniform random point sampling on 3D surfaces offers a powerful framework to capture complex distributions such as fur seeds, scattered geometric instances or light emitter flux. As most other on-surface signals, practitioners design such distribution by the means of 2D maps, parameterized over the surface, and indicating locally the desired point density that should be synthesized, along with any primitive-specific attribute such as fiber thickness or instance size. Numerous application scenarios imply large such point sets which would ideally be generated in real-time to be consumed immediately by downstream applications. We propose a method to distribute such white noise point sets under non-uniform densities, designed to cope with parallel GPU execution and able to produce, in real time, hundreds of millions of density-constrained samples over arbitrary 3D triangle meshes. At the core of our method, we introduce a stratified rejection sampling scheme where triangles act as strata, greatly improving the locality of the sampling process, and significantly diminishing the probability of rejecting a sample. Our method relies on a series of simple GPU kernels, introducing a new fast and exact texel-triangle overlap computation method as well as the notion of unordered cumulative sum. As a result, our approach provides real-time systems with the ability to tailor, on-the-fly, highly dynamic density distributions in the form of procedural or raster maps. We illustrate its application with interactive fur design, geometry instancing, and Monte Carlo light sampling.

1. Introduction

Surface sampling serves various purposes in 3D computer graphics, including adaptive surface remeshing, particle systems, point-based modeling, texturing and importance sampling. One extreme scenario concerns the design of fur: millions of random fibers need to be distributed over 3D surfaces to obtain a realistic look, often with a non-uniform random distribution tailored by an artist. In fact, their design is often achieved indirectly, where the user works interactively with a small subset of samples, letting an offline process generate the actual full count set prior to final rendering. This in-

Table 1: *Notations*

$*, *_{axis}$	Convolution operator (2D, 1D along *axis*)
Surface	
$\mathcal{M}$	Triangle Mesh
$\mathcal{T}, \mathcal{T}_{uv}$	Triangle (resp. in world and *uv* space)
σ	Permutation of the triangles order
Density	
$\mathcal{D}$	Density map (in $samples/m^2$)
$\mathcal{D}^{\max}_{\mathcal{M}}$	Maximum density over $\mathcal{M}$
$\mathcal{D}^{\max}_{\mathcal{T}}, \overline{\mathcal{D}}_{\mathcal{T}}$	Per-triangle (maximum, average) density
Samples	
N_{target}	Optional user-requested sample count
$\overline{N}_{\mathcal{T}}, \overline{N}$	Expected sample count (per-triangle, total)
$N_{\mathcal{T}}, N$	Effective sample count (per-triangle, total)

direction between what is designed and what is obtained induces a slow try-and-test loop, where the user lacks immediate feedback on the actual dense fiber distribution. Similarly, emissivity sampling in Monte Carlo rendering requires non uniform random surface sampling when the emission is modeled through a map: instant feedback on the lighting design necessitates an efficient mechanism to sample the surface emission as its map evolves dynamically. These two use cases exemplify a broader problem: the real-time generation of a large number of random surface samples from a 3D triangle mesh, where the sampling distribution is guided by a 2D density map defined over the surface.

Contributions. We propose a new high-performance algorithm to sample very dense non-uniform random point sets on a 3D surface mesh. Our algorithm permits to control locally the density of the sample set by the mean of a density map which can dynamically evolve at every frame. Basically, our contribution takes the form of a stratified rejection sampling scheme designed for fine grain parallel-scalable GPU execution, where **triangles act as strata**. In particular, we introduce **(i)** a rasterization-based counting method, **(ii)** the notion of *unordered cumulative sums* for fast per-triangle sample count determination and **(iii)** a new high-speed method to compute the area of texel-triangle overlaps. We exemplify our approach with downstream interactive use cases which need to be fed with hundreds of millions of dynamic 3D samples per second capturing non uniform on-surface densities, including fiber seeding, geometric instancing and emissive surface sampling.

Notations. We consider as input a 3D surface mesh $\mathcal{M}$ made of triangles enriched with a target point sample distribution modeled as a 2D density scalar map $\mathcal{D}$. This map is expressed in samples per scene unit square and is mapped onto the surface using standard UV mapping. The texture coordinates *uv* are defined on $\mathcal{M}$ using linear interpolation of the triangles vertices *uv* attributes. Under those notations, we seek to generate N point samples, where each sample is defined by its supporting triangle index and its barycentric coordinates on the triangle. The number N can be either user-provided, or emerging from the mesh and its density map, in which case it has to be computed. We refer to Table 1 for all notations.

2. Previous Work

Our work takes place in the context of point sampling 3D triangle meshes – the de facto standard shape representation in 3D computer graphics. Our target sampling process is driven by two constraints: **(i)** non-uniform distribution, where the local density of samples is tailored by a 2D density map parameterized on the surface mesh and **(ii)** real-time computational cost for tens of millions of samples, with a highly efficient GPU execution that enables e.g., fur design.

Rejection sampling is a general sampling technique that can be applied to our scenario. As described by Pharr et al. [PJH23], rejection sampling draws samples from an arbitrary distribution, as long as its probability density function (PDF) f can be bounded by cp, where $c > 0$ is a constant, and p is another PDF that we already know how to sample from. In the context of density-based mesh sampling, p is typically chosen as a uniform sampling of the surface, and c as the maximum density over the mesh. Uniformly sampling the mesh requires to sample the discrete probability distribution given by the triangle areas. Many techniques exist to sample such distribution, for example by building its cumulative distribution function (CDF), itself sampled using binary search or an inverse CDF [CRW09], or using a data structure like an alias table [Vos91].

Rejection sampling applied to a density-based mesh sampling leads to a simple procedure, shown in Algorithm 1, where after a lightweight CDF preprocess, samples are generated independently, and for each sample, candidates samples are iteratively tested until one gets accepted. At first glance, Algorithm 1 seems well suited for a highly-parallel GPU implementation, since all samples can be trivially generated in parallel. It however suffers from shortcomings regarding its runtime efficiency for three main reasons.

First, rejection sampling relies on an unbounded loop whose effective duration can vary greatly between samples. Given the Single Instruction Multiple Thread (SIMT) programming model of GPUs, where bundles of threads called *subgroups* execute simultaneously the same instructions, the loop duration for a given thread is always equal to the worst loop duration over the entire *subgroup* the thread belongs to. Moreover, the loop stopping criterion is based on the maximum density over the mesh, which can be a very inefficient density bound when the density is sparse.

Secondly, rejection sampling relies on very incoherent memory

Algorithm 1: Plain Rejection Sampling

Input: Mesh $\mathcal{M}$, density map $\mathcal{D}$, and sample count N.
Result: Output sample set $\mathcal{S}$ with N samples.

```
distribution ← TriangleAreaDistribution(M)
parallel for sample in S do
    while true do
        T ← Sample(distribution)
        sample ← SampleUniform(T)
        if D(sample) ≥ D^max_M · Rand0to1() then
            end thread
return S
```

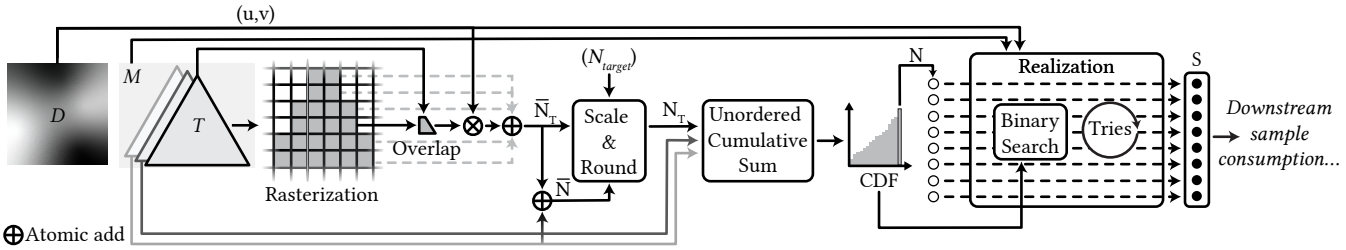


Figure 2: *Overview of our Triangle Rejection Sampling algorithm.*

accesses which, given the low computational complexity of Algorithm 1, is the limiting factor for its GPU execution. At each new sample candidate, triangle and density data have to be loaded from memory. But since there is no spatial coherency when uniformly sampling a surface, each new sample candidate is likely to produce a cache miss for both triangle-related and density-related data. Since every thread can sample any location on the surface, there is also no spatial coherency for threads belonging to the same *subgroup*.

Third, rejection sampling also suffers from memory inefficiency when the sample count N emerges from the distribution rather than being provided upfront, because memory has to be allocated considering the worst case, where the maximum density is applied everywhere on the mesh. Related to this problem, approaches using space-filling curves for high-quality sampling of surfaces and meshes [QLLM13] were proposed, as well as methods exploiting a spatial structure guiding the sampling quality [NS04]. Those contributions propose low-discrepancy sampling of meshes, but are not tailored for real-time scenarios involving millions of samples to synthesize in a couple of milliseconds as they remain weakly compatible with fine-grained GPU parallel execution.

Poisson disk samplers [YXSH13, YWW14, Yuk15, WS18, BJFH19] focus on the sample set spectral properties but lead to more than 3 orders of magnitude lower throughput than our method, even on GPU [BWWM10]. Still, an initial random sampling is often expected, a key step for which our method establishes a new state of the art. We refer to the survey of Yan et al. [YGW*15] for a comprehensive overview of blue noise sampling methods. Recently, Abdalla et al. [ASHW23] explored digital dyadic sequences for high performance sampling but their use for triangle meshes equipped with density maps and their performance in such a context remains unclear.

To the best of our knowledge, only *Fast Random Sampling of Triangular Meshes* (FRS), proposed by Šik and Křivánek [ŠK13] fits our constraints. In their paper, the authors propose an adaptive subdivision of the triangle mesh to match the density map resolution. The density is therefore approximately constant over all subdivided micro-triangles, allowing a straightforward point sampling on any micro-triangle. However, in the case of a high-resolution density map, the subdivision depth and thus processing cost, as well as the memory footprint, grow substantially. Indeed, the massive amount of generated micro-triangles has to be stored in memory, and a CDF still has to be built over the micro-triangles areas. The authors remain unclear about the scalability of their technique, providing results only on a 4 cores CPU implementation.

Finally, Portsmouth [Por17] proposed an efficient inversion sampling for the case of a mesh with a per-vertex density field, which does not cope for the more general case of parameterized density maps exhibiting arbitrary resolution and sparsity.

Our method takes inspiration from rejection sampling, and addresses its shortcomings by proposing a stratified version of rejection sampling where triangles act as strata. This greatly improves the locality of the sampling process, critical for fast memory accesses, and improves the probability of sample acceptance, making the overall sampling process extremely efficient on modern highly-parallel SIMT programming model. Complementing the use of plain rejection sampling, our analysis also incorporates a "naïve" version of our proposed stratification approach as a comparative baseline (Section 6), to highlight the critical role of the specific components we introduce in achieving high performance triangle stratification.

3. Our Method

3.1. Overview

Our on-surface sampling method is a stratified rejection sampling method over the mesh triangles (Figure 2), and relies on a series of simple GPU kernels that we organize in three main passes. First, we decompose the problem of sampling N points on the surface into multiple and independent per-triangle sampling problems (Figure 3). The *counting pass* (Section 3.2) estimates how many samples $\overline{N}_\mathcal{T}$ should be drawn for each triangle, and also retrieves the maximum density per triangle. Second, we propose an *unordered cumulative sum pass* (Section 3.3) to compute a CDF of these per-triangle sample counts. Third, the *realization pass* (Section 3.4) sets a position for each sample, that can be consumed by the downstream application e.g., as locations to instantiate geometric primitives or evaluate light emitted radiance.

3.2. Counting pass

The counting pass, summarized in Algorithm 2, computes the effective number of samples $N_\mathcal{T}$ we want to assign to each triangle. We first have to compute the per-triangle sample counts $\overline{N}_\mathcal{T}$ that emerge from the density map applied to the mesh. To do so, we integrate the density map over each triangle. Since the uv mapping

Algorithm 2: Counting pass

Input: Mesh $\mathcal{M}$, density map $\mathcal{D}$, and optional user-provided target sample count N_{target}
Result: Per-triangle effective sample counts $N_\mathcal{T}$ and max densities $\mathcal{D}^{max}_\mathcal{T}$.

$\overline{N}_\mathcal{T}, \mathcal{D}^{max}_\mathcal{T} \leftarrow 0$ /* For all triangles */
parallel for $fragment \in UVRasterization(\mathcal{M})$ **do**
  $uv, \mathcal{T}, \mathcal{T}_{uv} \leftarrow fragment$
  $d \leftarrow \mathcal{D}(uv)$
  $w \leftarrow \text{TexelOverlap}(uv, \mathcal{T}_{uv})$ /* Section 4 */
  $\overline{N}_{fragment} \leftarrow d \cdot w \cdot \text{Area}(\mathcal{T})/\text{Area}(\mathcal{T}_{uv})$
  $\overline{N}_\mathcal{T} \leftarrow \overline{N}_\mathcal{T} + \overline{N}_{fragment}$ /* AtomicAdd */
  $\mathcal{D}^{max}_\mathcal{T} \leftarrow \max(\mathcal{D}^{max}_\mathcal{T}, d)$ /* AtomicMax */
$\overline{N} \leftarrow 0$
parallel for $\mathcal{T} \in \mathcal{M}$ **do**
  $\overline{N} \leftarrow \overline{N} + \overline{N}_\mathcal{T}$ /* AtomicAdd */
$scaling \leftarrow$ **if** N_{target} **then** $N_{target}/\overline{N}$ **else** 1
parallel for $\mathcal{T} \in \mathcal{M}$ **do**
  $N_\mathcal{T} \leftarrow \text{StochasticRounding}(scaling \cdot \overline{N}_\mathcal{T})$
return $N_\mathcal{T}, \mathcal{D}^{max}_\mathcal{T}$

distortion is constant per triangle, we have:

$$\overline{N}_\mathcal{T} = \int_\mathcal{T} \mathcal{D}(s)\, dS = \frac{\text{Area}(\mathcal{T})}{\text{Area}(\mathcal{T}_{uv})} \int_{\mathcal{T}_{uv}} \mathcal{D}(uv)\, \mathrm{d}uv \tag{1}$$

Assuming a nearest-neighbor filtering of the density map, the integral above in uv space is accurately computed as a weighted sum over all texels p of the density texture $\mathcal{D}$ that overlap the triangle:

$$\int_{\mathcal{T}_{uv}} \mathcal{D}(uv)\, \mathrm{d}uv = \sum_{p \cap \mathcal{T}_{uv} \neq \emptyset} \mathcal{D}(p) \text{Area}(p \cap \mathcal{T}_{uv}) \tag{2}$$

Evaluating Equation (2) for each triangle in parallel can be done by assigning one thread to each triangle, each thread iterating over all the texels that overlap the triangle in uv space and accumulating the terms using an off-the-shelf texel-triangle intersection routine [LF15]. This approach, that we will call *naive triangle stratification* for the rest of the paper would be quite inefficient for two reasons. First, the number of texels can vary greatly between triangles. So all threads within a *subgroup* would have to wait for the thread corresponding to the triangle with the largest amount of texels covered. We instead rely on the hardware rasterization pipeline and rasterize the mesh in uv space to compute the sum at a finer granularity. Each thread, or rather each fragment shader invocation, is responsible for computing a single term of the sum in Equation (2), rather than computing the full sum. Therefore, every fragment only has to retrieve a single density value and compute a single triangle-texel overlap. The max density and expected sample count (for the triangle that generated this fragment) are then updated using atomic operations. Second, existing texel-triangle intersection methods are not well suited for GPU execution, so we propose instead a novel method that is described in more details in Section 4.

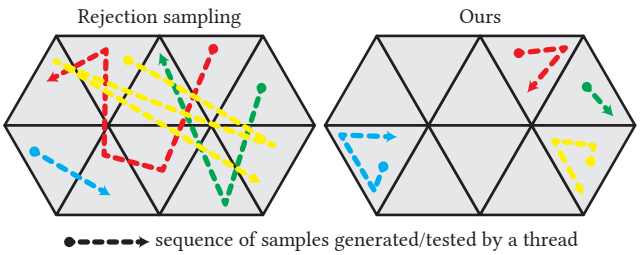


Figure 3: ***Triangles as native strata.*** *Left: rejection sampling on a surface mesh. Right: our triangle stratification.*

After the uv rasterization process, we obtain the per-triangle expected sample counts that emerge from the mesh and the density. However, for performance reasons, memory constraints, or user-defined control, an application may require to set the number of samples generated to a target count N_{target}. Our technique easily copes with this constraint by scaling the expected sample counts by the ratio between the target and expected total sample count. Finally, since the expected sample counts $\overline{N}_\mathcal{T}$ are decimal numbers, they are rounded to obtain the effective integer amount of samples $N_\mathcal{T}$ assigned to each triangle. The rounding is performed up or down stochastically based on the fractional part of the sample counts, to ensure asymptotic convergence of the effective total sample count towards N_{target}.

Floating-point addition induces that the exact values for $\overline{N}_\mathcal{T}$ and $\overline{N}$ depend on the parallel execution scheduling order. In practice, we observed that the value fluctuations only affect digits around the last floating-point significant digit, so had little impact on the final integer sample counts.

3.3. Unordered cumulative sum pass

At this point we have computed the number of samples we want to draw from each triangle. These sample counts, stored in an array, define a distribution that we wish to sample efficiently using a CDF to assign a supporting triangle to each sample. This means computing a prefix sum of the array which, in a highly-parallel setup, either involves several simple GPU kernel calls [HSJ86, HSO07], or a single but complex GPU kernel call [MG16]. However, noticing that

Algorithm 3: Unordered Cumulative Sum

Input: Array A containing 32-bit unsigned integers
Result: Cumulative sum of A as if A has been shuffled by a permutation σ beforehand, and the inverse permutation σ^{-1}.

$counter \leftarrow 0$ /* 64-bit counter */
parallel for $input\ index$ ***in*** $range(0, length(A))$ **do**
  $element = A[input\ index]$
  $increment \leftarrow \text{PackUint2x32}(element, 1)$
  $result \leftarrow \text{AtomicAdd}(counter, increment)$
  $sum,\ processing\ index \leftarrow \text{UnpackUint2x32}(result)$
  $CDF[processing\ index] \leftarrow sum + element$
  $\sigma^{-1}[processing\ index] \leftarrow input\ index$
return CDF, σ^{-1}

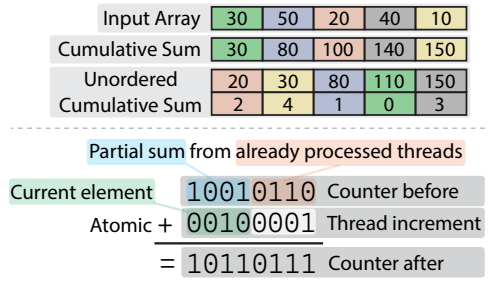


Figure 4: *Top: given an input array, we compare the standard cumulative sum, with one possible outcome for our unordered cumulative sum. Bottom: bit representation of the atomic counter increment that simultaneously updates the number of already processed threads and the partial sum of their corresponding elements (shown here using* 2×4 *bits instead of* 2×32*).*

sampling a discrete distribution is invariant under any permutation of its elements, we propose a novel parallel cumulative sum computation algorithm, which requires a single and simple GPU kernel call, shown in Algorithm 3, and naturally deals with the unspecified scheduling order of GPUs.

We now describe our *unordered* cumulative sum computation, whose result is highlighted in Figure 4 (top). Launching threads in parallel, each thread is responsible for reading one element from the input array, and writing one entry in the output cumulative sum. This entry is equal to the sum of all the elements already processed by other threads, incremented by the currently processed element, and its writing location in the output cumulative sum is equal to the number of threads already processed. These two quantities, namely the number of processed threads and the sum of their elements, should be tracked, but the subtlety here is that these two quantities must be incremented synchronously with respect to other threads. While such synchronous update is possible on GPU for a single variable using atomic operations, the simultaneous update of two variables in not possible on current shader execution models, nor efficient in more general GPGPU contexts. We address this limitation with a standard practice, consisting in packing two 32-bit integers into a single 64-bit integer, effectively combining two atomic counters into a single one and guaranteeing their synchronous updates, as shown in Figure 4 (bottom).

Since the threads processing order is unspecified in a parallel execution context, the presented algorithm computes a cumulative sum, as if the input has been shuffled by some permutation. Consequently, when the distribution is sampled later on using e.g., a binary search on the *unordered* cumulative sum, the processing order index is returned instead of the triangle index we seek. This is why we also output the inverse permutation, which in our case maps processing indices back to triangle indices.

The implementation shown in Algorithm 3 limits the number of triangles and the number of samples to be less than 2^{32}. However, if the triangle count is represented using n bits, then our *unordered* cumulative sum can actually handle up to $64-n$ bits samples with manual packing. Since 128-bit atomics are currently not supported by any GPU, applications requiring more than $64-n$ bits would have to use a standard prefix sum instead of Algorithm 3.

3.4. Realization pass

In this last pass, summarized in Algorithm 4, the samples locations are finally computed. Similarly to rejection sampling (Algorithm 1), we assign one thread per output sample and, for each, we iteratively generate sample candidates until one gets accepted. We used the low distortion map proposed by Heitz [Hei19] to uniformly sample locations on triangles, ensuring efficient and arithmetic-only computations. Our method however differs from plain rejection sampling in two crucial ways.

First, each sample is assigned to a fixed support triangle, whose index is obtained using a binary search on the previously computed *unordered* cumulative sum. Therefore, all candidate samples for a given thread are located on the same triangle. This greatly improves the efficiency of the memory reads, as not only the triangle data has to be fetched only once per thread, but all density texture fetches will be located on the same triangle in *uv* space. So for reasonably sized triangles in *uv* space, all density fetches after the first one are likely to be read directly from the texture cache. This reasoning also extends to the scale of multiple threads. Since the supporting triangle index only depends on the sample index, samples that are processed in the same group of threads are likely to share the same support triangle, and are therefore likely to benefit from memory caching. Second, we use our precomputed per-triangle maximum density $\mathcal{D}_{\mathcal{T}}^{\max}$ instead of the less tight mesh global maximum density, which increases the samples acceptance rate and thus reduces on average the duration of the sampling loop.

Upon acceptance, a sample can be consumed directly or stored for future use. We show both use cases in our example applications (Figure 15). For fur generation, we process samples directly in order to compute the fibers attributes, while for instancing and light sampling, samples are stored into a buffer that is consumed by a subsequent rendering pass.

Algorithm 4: Realization pass

Input: Mesh $\mathcal{M}$, density map $\mathcal{D}$, per-triangle max densities $\mathcal{D}_{\mathcal{T}}^{max}$, and *unordered CDF* of the effective sample count per triangle and its corresponding inverse permutation σ^{-1}.
Result: An array $\mathcal{S}$ containing N samples.

$N \leftarrow$ last element(CDF) `/* Sample count */`
parallel for *sample index* ***in*** *range*$(0,N)$ **do**
  processing index $\leftarrow$ BinarySearch(CDF, *sample index*)
  triangle index $\leftarrow \sigma^{-1}$[*processing index*]
  $\mathcal{T} \leftarrow$ *triangle index*
  while ***true*** **do**
    sample $\leftarrow$ SampleUniform($\mathcal{T}$)
    if $\mathcal{D}$(*sample*) $\geq \mathcal{D}_{\mathcal{T}}^{max} \cdot$ Rand0to1() **then**
      $\mathcal{S}$[*sample index*] $\leftarrow$ *sample*
      end thread
return $\mathcal{S}$

4. Triangle-texel overlap computation

In this section, we describe our contribution for computing the triangle-texel overlap. If we consider the general problem of intersecting a 2D triangle with a 2D square, the resulting polygon can have various shapes, from a triangle to a convex heptagon, or can be empty. Considering Equation (2), we actually only need the area of that triangle-texel intersection polygon, not its explicit shape. To the best of our knowledge, existing techniques for computing the overlap first extract the intersection polygon [YSZ*06, OCON82], and then retrieve the area using closed-form formulas such as the shoelace formula. Although efficient, those techniques present undesirable characteristics in our GPU context: memory footprint, branching, and computation cost. We instead consider the triangle-texel overlap under a signal processing perspective. The overlap area then emerges as a 2D convolution, from which we derive a satisfying analytic closed-form that is well suited for GPU, with no branching, minimal memory footprint, and lightweight computational cost.

In the texel space, we define over $\mathbb{R}^2$ the indicator function $\mathbb{1}_{\mathcal{T}}$ of a triangle that returns 1 inside $\mathcal{T}_{uv}$, and 0 otherwise. Similarly, we define the indicator function for a unit square centered at the origin $\mathbb{1}_{\square}$. Using those functions, the triangle-texel overlap $\text{Area}(p \cap \mathcal{T}_{uv})$ with a texel centered in (x,y) can be naturally expressed as the result of a convolution (Figure 5f):

$$\text{Area}(p(x,y) \cap \mathcal{T}_{uv}) = (\mathbb{1}_{\mathcal{T}} * \mathbb{1}_{\square})(x,y) \tag{3}$$

The unit square indicator kernel is separable since $\mathbb{1}_{\square}(x,y) = \mathbb{1}_{\sqcap}(x) \cdot \mathbb{1}_{\sqcap}(y)$, where $\mathbb{1}_{\sqcap} = \mathbb{1}_{[-0.5,0.5]}$ is the centered 1D unit gate function. Our goal is therefore to find an expression for the triangle indicator function $\mathbb{1}_{\mathcal{T}}$ such that its convolution by a unit gate is tractable. For that purpose, we use for each oriented edge i of the triangle $\mathcal{T}_{uv}$ the function $E_i(x,y) = \mathbb{1}_{[y_1^i, y_2^i]}(y) \left(\mathbb{1}_{\mathcal{H}_i}(x,y) - \frac{1}{2} \right)$, where $[y_1^i, y_2^i]$ is the y-span of the edge, and $\mathcal{H}_i$ its oriented half-space (Figure 5a)

The edge functions E_i have two important properties. First, we have $\sum E_i = \mathbb{1}_{\mathcal{T}}$ (Figure 5b), so by linearity of convolution, only $E_i * \mathbb{1}_{\square} = E_i *_x \mathbb{1}_{\sqcap} *_y \mathbb{1}_{\sqcap}$ is needed to compute the overlap area. Second, E_i is an oriented step function in 2D, so its convolution along the x-axis with $\mathbb{1}_{\sqcap}$ results in a piecewise linear (clamp) 2D function (Figure 5c), that can be expressed, up to a constant, as the difference of two functions $f_i^{\pm}$ (Figure 5e):

$$(E_i *_x \mathbb{1}_{\sqcap})(x,y) = \mathbb{1}_{[y_1^i, y_2^i]}(y) \left(f_i^+(x,y) - f_i^-(x,y) \right) \tag{4}$$

where $f_i^{\pm}(x,y) = \frac{1}{2n_i^x} \left| \text{dot}(\mathbf{n_i}, (x,y) - \mathbf{v_i}) \pm \frac{n_i^x}{2} \right|$, $\mathbf{n_i}$ is the edge normal (with n_i^x its x component), and $\mathbf{v_i}$ is any point on the edge. Putting everything together, we have:

$$\begin{aligned} \mathbb{1}_{\mathcal{T}} * \mathbb{1}_{\square} &= \sum_{1 \le i \le 3} \left(\mathbb{1}_{[y_1^i, y_2^i]}(y) \left(f_i^+(x,y) - f_i^-(x,y) \right) \right) *_y \mathbb{1}_{\sqcap} \\ &= \sum_{1 \le i \le 3} \int_{\max(y-0.5,\, y_1^i)}^{\min(y+0.5,\, y_2^i)} f_i^+(x,t) - f_i^-(x,t)\, \mathrm{d}t \end{aligned} \tag{5}$$

Since $f_i^{\pm}$ is the absolute value of a linear function in x and y, each of the six integrals in Equation (5) has a simple closed-form solution. The triangle-texel overlap computation therefore only involves a low amount of GPU-friendly instructions, regardless of the intersection shape.

Although this contribution was motivated by the triangle-texel overlap context, Equation (5) can be trivially extended to compute the area of the intersection between any polygon without self-intersection, and a rectangle. The computational cost is linear with the number of edges of the polygon, independently from its concavity, and the formula does not suffer from branching.

5. Implementation details

In this section, we describe practical details and considerations that are relevant for correctness and performance.

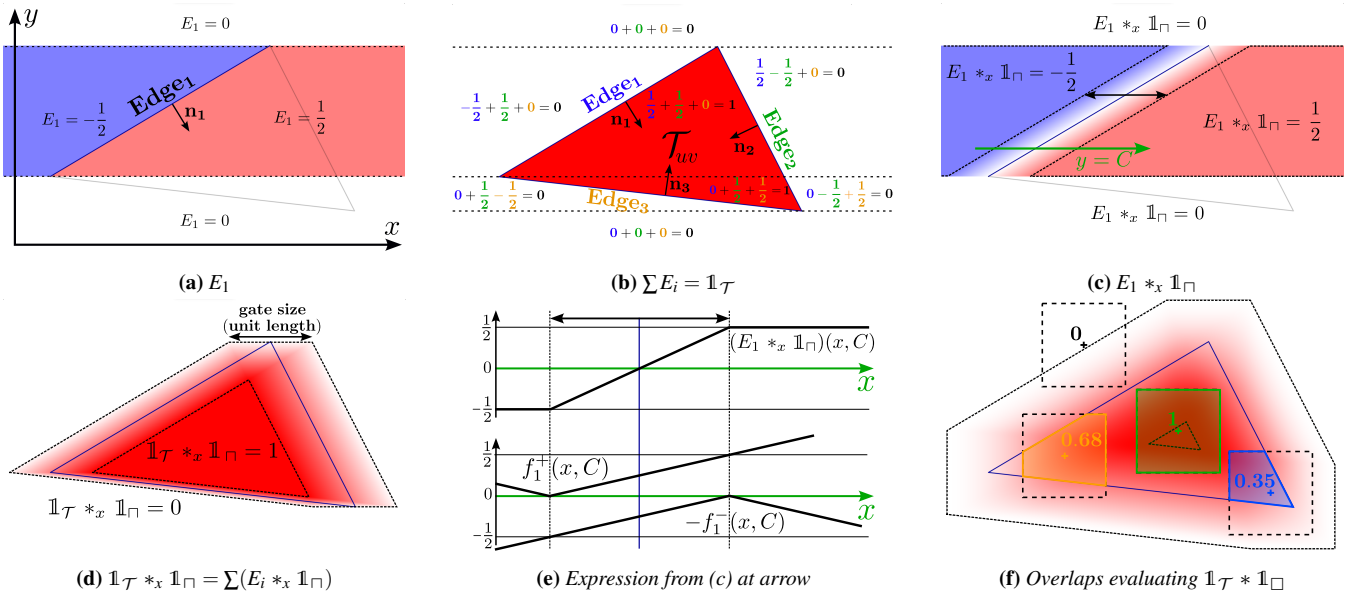
Conservative rasterization To enumerate, for each triangle, all density texels that overlap it, the *counting pass* described in Section 3.2 rasterizes the input mesh in *uv* space, i.e. using texture coordinates as output positions in the vertex shader. However, the native rasterization pipeline only generates fragments whose center lies inside the triangle. Fortunately, modern graphics APIs propose a *conservative rasterization* mode which, for every non-degenerate triangle, guarantees that all fragments overlapping the triangle will be generated and will contribute to their respective density integral.

UV extent Another consequence of the *uv* rasterization is that the viewport size should be equal to the size of the input density map. In case the input *uv* mapping spans multiple *uv* tiles, the viewport size should be extended accordingly. Note that since our rasterization pass does not use a depth or color attachment, the viewport size has no impact on memory consumption.

Subgroup-level reduction Most of the kernels from Algorithm 2 and Algorithm 3 perform some kind of parallel reduction using atomic operations. While atomic operations are quite efficient on current hardware [ELF11], they remain our performance bottleneck. Still, highly efficient data sharing is possible between threads that belong to the same *subgroup*, through the usage of dedicated intrinsic functions [MYK23]. Thanks to those functions, each atomic operation per thread can be replaced with one atomic operation for a group of threads. We refer to the Vulkan Subgroup Tutorial [Hen23] for implementation examples.

Note that the rasterization kernel requires a bit more care because reductions outputs are triangle-dependent. So for a given *subgroup*, only the subset of fragments that were generated by the same triangle can use intrinsics to accelerate reductions. In practice, we simply use subgroup-level reduction for a *subgroup* when all of its fragments were generated by the same triangle, and fall back to per-fragment atomic operations otherwise.

Triangle-texel overlap Within the counting pass, we compute the overlap of a triangle with many different texels (see Algorithm 2). Therefore some terms from Equation (5) can be precomputed per triangle to speed-up computations. This property integrates smoothly with the rasterization pipeline where per-triangle features



(a) E_1 **(b)** $\sum E_i = \mathbb{1}_\mathcal{T}$ **(c)** $E_1 *_x \mathbb{1}_\sqcap$

(d) $\mathbb{1}_\mathcal{T} *_x \mathbb{1}_\sqcap = \sum (E_i *_x \mathbb{1}_\sqcap)$ **(e)** *Expression from (c) at arrow* **(f)** *Overlaps evaluating* $\mathbb{1}_\mathcal{T} * \mathbb{1}_\square$

Figure 5: *Breakdown of our triangle-texel overlap computation*

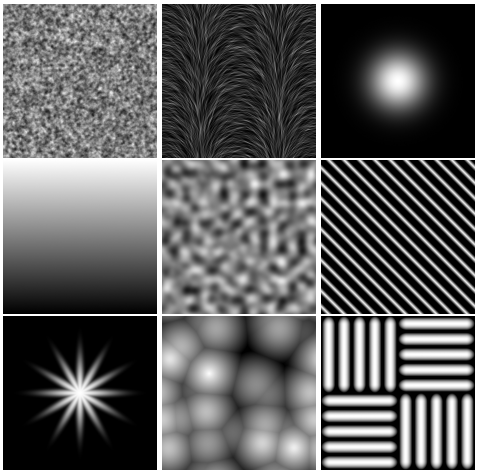


Figure 6: *Our density benchmark dataset covers distributions which are diverse in structures and frequencies.*

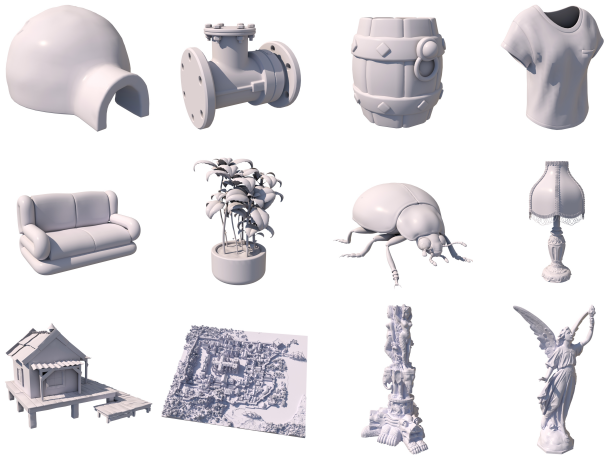


Figure 7: *Our mesh benchmark dataset ranges from 5k (*Igloo*) to 28M (*Stanford Lucy*) triangles.*

are computed in the vertex and forwarded to fragment shader invocations that compute the actual overlaps. We provide a complete implementation of our triangle-texel overlap technique as a Shadertoy demo https://www.shadertoy.com/view/3fV3Wz.

6. Results

One key aspect of our method is it that it efficiently runs on GPU without preprocessing. Our implementation uses Vulkan shaders [SK16]. All measurements were done with an RTX 4090 laptop GPU.

6.1. Comparisons

Sampling We compare our method to plain rejection sampling (RS) and naive triangle stratification as a baselines, and to FRS [ŠK13] which, to the best of our knowledge, is state-of-the-art. For fair comparisons, we implemented both methods on the GPU as well using Vulkan compute shaders. To evaluate performance over a range of use cases, we built a benchmark consisting of 9 density maps (Figure 6) and 12 meshes (Figure 7), yielding 108 different inputs for a benchmark. The Lucy and Statue models come from the Stanford Scanning Repository [LGCP05], the Rungholt model comes from the Computer Graphics Archive [McG17], and the other models come from the Substance 3D Assets library [Sub25].

In Figure 8, we report timing performances in ms, over the whole dataset, to generate a target sample count ranging from 1k to 100M samples. We can observe that at low sample count, i.e. less than a hundred of thousands, RS remains competitive, while FRS pays the price of its initial subdivision phase. However, as the sample count grows, the benefit of our method becomes more and more visible, and beyond the million of samples, we consistently outperform the other methods by at least an order of magnitude. Consequently, our method is the only one that stays within the real-time regime for a large sample count, taking less than 10ms to generate 100M samples. Figure 8 also shows that the performance of our method is more predictable than the competition, as its variance over the dataset is much lower.

In Figure 9, we show our method resilience on very large meshes containing dozens of millions of triangles and in Figure 10, we show that ultra-sparsity of the density can make our method slower than FRS in some cases. Since they do not rely on rejection sampling, finding the few non-zero density pixels is performed at preprocessing time rather than at sampling time. We present in the supplementary document a detailed analysis about how the triangle stratification and the per-triangle max densities significantly increase the expectancy of accepting a sample in practice over plain rejection sampling.

Triangle-texel overlap As reported in Figure 11, we compare our convolution approach against a standard polygon intersection routine [LF15], leading to significant performance degradations. We also compare to an approximate but computationally less expensive formula to compute the overlap, which has a limiting impact on performances. The formula we used is based on the 2D signed distance field (SDF) of the triangle:

$$\text{Area}(p \cap \mathcal{T}_{uv}) \approx \text{clamp}(0.5 + \text{SDF}(\mathcal{T}_{uv}, \text{center}(p)), 0.0, 1.0) \quad (6)$$

However, this approximation can lead to visible sampling artifacts (Figure 12).

Cumulative sum We compare in Figure 13 our *unordered* cumulative sum to the state of the art parallel cumulative sum method

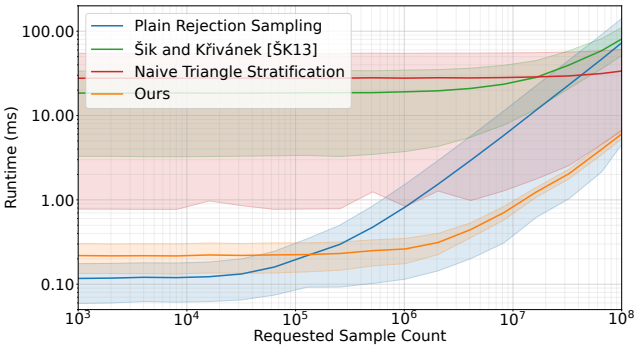


Figure 8: *Performance timings, averaged over the test dataset (in milliseconds, **logarithmic scale**), for a growing requested sample count, reported with standard deviation, for rejection sampling, FRS [ŠK13], rejection sampling with naive triangle stratification, and our method.*

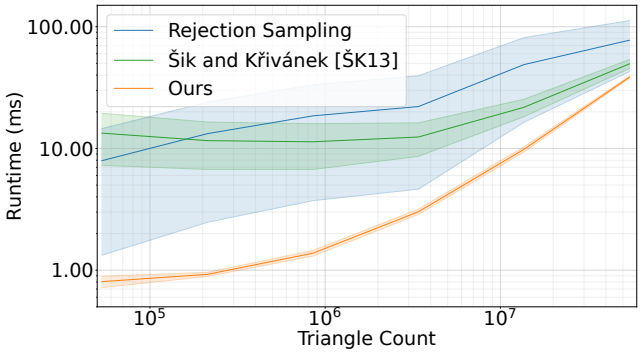


Figure 9: *Performance timings, averaged over the densities dataset (in milliseconds, **logarithmic scale**), for a fixed mesh (ladybug from Figure 7) and 10M sample count, with a growing number of uniform subdivision to increase the triangle count, reported with standard deviation, for rejection sampling, FRS [ŠK13], and our method.*

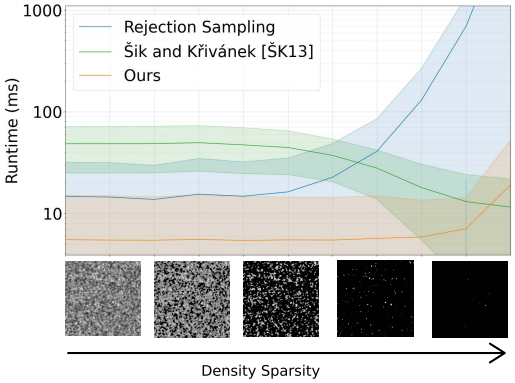


Figure 10: *Performance timings, averaged over the mesh dataset (in ms, **logarithmic scale**), for a fractal density map with different thresholding values that increase its sparsity, reported with standard deviation, for rejection sampling, FRS [ŠK13], and our method.*

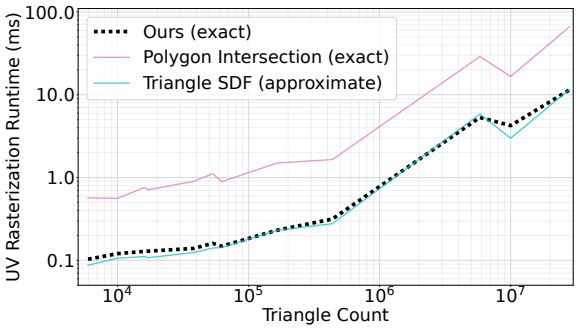


Figure 11: *UV rasterization performance, averaged over the test dataset (in milliseconds, **logarithmic scale**), for different methods to compute triangle-texel overlaps.*

[MG16], using a publicly available implementation [Lev21], which is substantially more complex to implement than ours, and to a method with a similar implementation complexity as ours [HSO07]. Our *unordered* cumulative sum performs significantly better at low triangle count, and offers similar performances to state of the art at high triangle count.

6.2. Ablation

We validate the different aspects of our method with an ablation, summarized in Figure 14. The different versions are either significantly slower, in particular at low sample count, or require additional implementation complexity without clear performance gains. We also add the previously mentioned texel-triangle overlap and cumulative sum comparisons in this figure to showcase their impact on the overall method.

Rasterization pipeline Replacing the rasterization pipeline with a compute pipeline parallelized over the triangles degrades performances the most (*"No Rasterization"*). This was expected since processing one triangle per compute thread leads to stalling threads waiting for the biggest triangle of the warp to be processed. Using a mesh shader instead of a vertex shader to reduce the amount of per-triangle computations has no visible impact on performances, confirming that fragment shader invocations are the bottleneck of the pipeline (*"Mesh Shader"*). Using finer-grained subgroup-level reductions in the fragment shader shows no clear benefit as well (*"Subgroup Partitions"*, *"Subgroup Clusters"*), while not using any Subgroup-level reduction significantly impacts performances (*"No Subgroup Reduction"*).

Inverse CDF The usage of an inverse CDF to accelerate the binary search for sampling the CDF degrades the performances at low to medium sample counts, and has negligible benefits for large sample counts (*"Inverse CDF"*). This is different from FSR, for which the usage of an inverse CDF is critical [ŠK13].

6.3. Applications

We show results in Figure 15, with high resolution density maps, on various mesh complexities, and across different applications.

Fur placement The fabric fuzz is distributed over a pattern structure. The distribution is controlled by a density map designed as a procedural graph which allows to synchronize fuzz density with material appearance. The sampling distribution is computed at each frame which allows to vary dynamically both the input mesh and the procedural material graph.

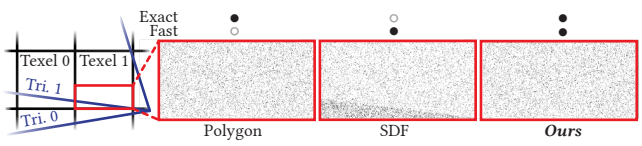


Figure 12: *Sampling with 3 different texel-tri. overlap methods.*

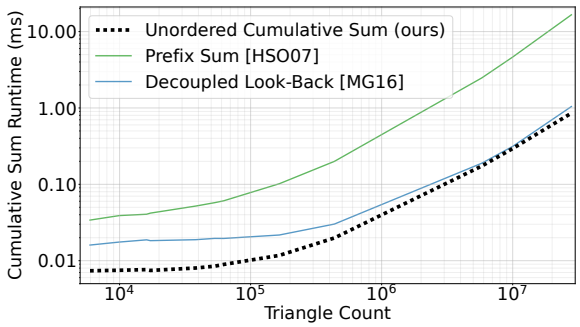


Figure 13: *Cumulative sum comparison performance, averaged over the mesh dataset (in milliseconds, **logarithmic scale**), for the methods of Harris et al. [HSO07], Merril et al. [MG16], and our unordered cumulative sum.*

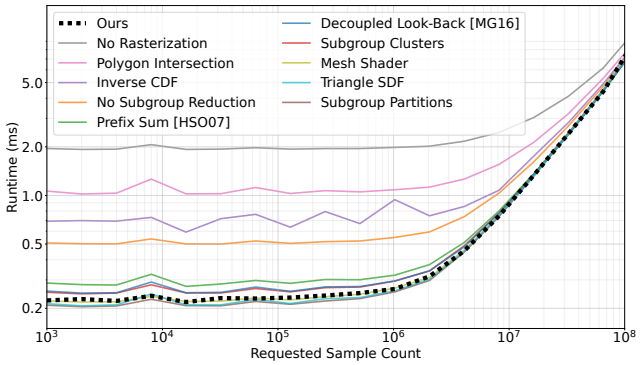


Figure 14: *Performance timings, averaged over the test dataset (in milliseconds, **logarithmic scale**), for a growing requested sample count, and different versions of our method. Each version differs from our final implementation by only one change, described in details in Section 6.2.*

Instancing The generated points serve as seeds for distributing small geometric instances over a surface. In our example, the instance parameters, such as scale and orientation, were randomised on-the-fly based on the sample index.

Light sampling By interpreting an the emissive map of a material as a density of light, our method computes the amount of emissive light per triangle, and generates a large number of point lights, approximating very well the emissive light distribution. This is very useful in a path tracing context, as it allows to finely importance sample portions of surface that actually emit light, thus greatly reducing the rendering noise variance caused by the light sampling.

7. Discussion, limitations and future work

Our mesh sampling technique exhibits high performances and parallel scalability, without requiring memory-expensive data structures. Its stratification is key for performances but using triangles as strata can be ineffective when the surface partitioning induced by triangles is very unbalanced, i.e. when a single triangle covers

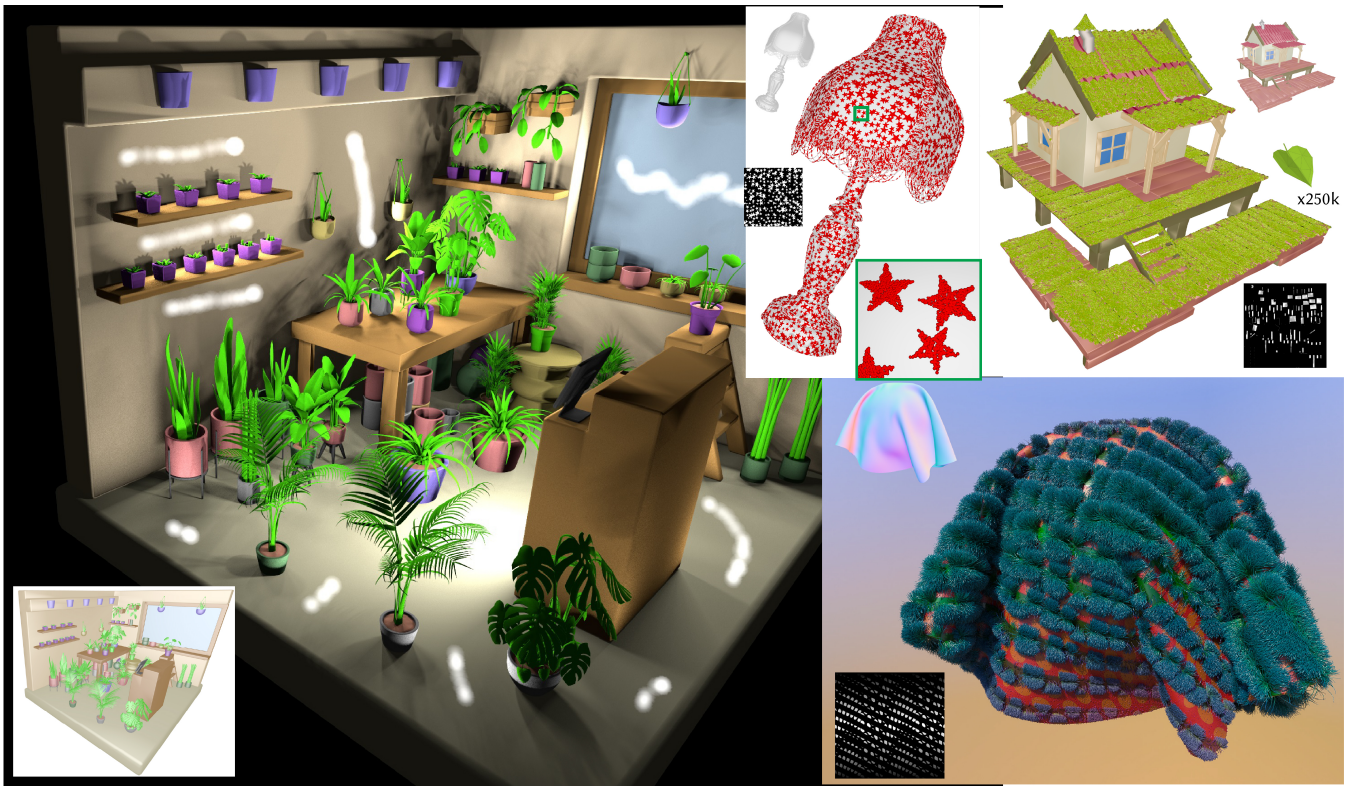


Figure 15: *Experimental downstream applications consuming the samples generated by our method in real time. For each use case, we show the 2k × 2k density map and the surface mesh. Left: emissive surface Monte Carlo rendering (1M samples, 1M triangles). Top middle: point set rendered as analytical spheres, with close-up in inset (10M samples, 400k triangles). For this example, the density map is tiled* 10 *times over the surface in both dimensions. Top right: geometry instancing seeding (250k samples, 60k triangles). Bottom right: fabric fuzz seeding (30M samples, 100k triangles). Additional examples provided as supplemental material.*

half the mesh area. Such cases may be addressed using an adaptive subdivision of the mesh, in the spirit of FRS [ŠK13] but with a stopping criterion less extreme than requiring a constant density per triangle.

Our method remains a rejection sampling approach, which does not cope well with extreme sparsity of the density, i.e. when out of millions of texels, only a couple of them have non-zero density. While we still handle those cases better than plain rejection sampling thanks to our stratification, finding those pixels will require many loop iterations. While this problem is rare in practical scenarios, efficient multi-resolution usage of the density map could be an interesting research direction to address it.

Considering the simplicity of our method, one can see it as drop-in replacement for any density-based random point sampling surfaces meshes. While our target scenarios are focused on consuming processes that request interactive feedback (Figure 15), a much wider set of 3D computer graphics applications may actually benefit from our sampler. Finally, we think that our texel-triangle overlap technique can be extended to 3D for fast intersection area computation between a mesh and a box.

Acknowledgments

We thank Andréa Machizaud for his kind, precious and efficient help with the GPU path tracer, David Farrel for his insightful suggestion regarding Vulkan memory, and the anonymous reviewers for their time and feedback.

References

[ASHW23] AHMED A. G. M., SKOPENKOV M., HADWIGER M., WONKA P.: Analysis and synthesis of digital dyadic sequences. 3

[BJFH19] BRANDT S., JÄHN C., FISCHER M., HEIDE F. M. A. D.: Visibility-Aware Progressive Farthest Point Sampling on the GPU. *CGF* (2019). 3

[BWWM10] BOWERS J., WANG R., WEI L.-Y., MALETZ D.: Parallel poisson disk sampling with spectrum analysis on surfaces. 3

[CRW09] CLINE D., RAZDAN A., WONKA P.: A comparison of tabular pdf inversion methods. In *CGF* (2009), vol. 28, pp. 154–160. 2

[ELF11] ELTEIR M., LIN H., FENG W.-C.: Performance characterization and optimization of atomic operations on amd gpus. In *IEEE Int. Conf. on Cluster Computing* (2011), pp. 234–243. 6

[Hei19] HEITZ E.: *A Low-Distortion Map Between Triangle and Square*. Tech. rep., Unity Technologies, 2019. 5

[Hen23] HENNING N.: Vulkan subgroup tutorial. https:

//www.khronos.org/blog/vulkan-subgroup-tutorial, 2023. Accessed: 2025-01-18. 6

[HSJ86] HILLIS W. D., STEELE JR G. L.: Data parallel algorithms. *Communications of the ACM 29*, 12 (1986), 1170–1183. 4

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with cuda. *GPU gems 3*, 39 (2007), 851–876. 4, 9

[Lev21] LEVIEN R.: Prefix sum. https://raphlinus.github.io/gpu/2020/04/30/prefix-sum.html, 2021. 9

[LF15] LÉVY B., FILBOIS A.: Geogram: a library for geometric algorithms. 4, 8

[LGCP05] LEVOY M., GERTH J., CURLESS B., PULL K.: The stanford 3d scanning repository. *URL http://www-graphics. stanford. edu/data/3dscanrep 5*, 10 (2005), 10. 7

[McG17] MCGUIRE M.: Computer graphics archive, July 2017. https://casual-effects.com/data. URL: https://casual-effects.com/data. 7

[MG16] MERRILL D., GARLAND M.: Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016). 4, 9

[MYK23] MEISTER D., YOSHIMURA A., KAO C.-C.: Gpu programming primitives for computer graphics. In *SIG. Asia Courses*. 2023, pp. 1–81. 6

[NS04] NEHAB D., SHILANE P.: Stratified point sampling of 3d models. In *Proc. Point-Based Graphics* (2004), pp. 49–56. 3

[OCON82] O'ROURKE J., CHIEN C.-B., OLSON T., NADDOR D.: A new linear algorithm for intersecting convex polygons. *Computer graphics and image processing 19*, 4 (1982), 384–391. 6

[PJH23] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. 2023. 2

[Por17] PORTSMOUTH J.: Efficient barycentric point sampling on meshes. *CoRR abs/1708.07559* (2017). 3

[QLLM13] QUINN J., LANGBEIN F., LAI Y.-K., MARTIN R.: Fast low-discrepancy sampling of parametric surfaces and meshes. 3

[ŠK13] ŠIK M., KŘIVÁNEK J.: Fast random sampling of triangular meshes. In *Pacific Graphics Short Papers* (2013). 3, 7, 8, 9, 10

[SK16] SELLERS G., KESSENICH J.: *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016. 7

[Sub25] Substance 3d assets. https://substance3d.adobe.com/assets, 2025. 7

[Vos91] VOSE M. D.: A linear algorithm for generating random numbers with a given distribution. *IEEE TSE 17*, 9 (1991), 972–975. 2

[WS18] WANG T., SUDA R.: Fast generation of poisson-disk samples on mesh surfaces by progressive sample projection. 3

[YGW*15] YAN D. M., GUO J. W., WANG B., ZHANG X. P., WONKA P.: A survey of blue-noise sampling and its applications. *Journal of Computer Science and Technology 30*, 3 (2015), 439–452. 3

[YSZ*06] YANG C., SHI P., ZAO W., WANG L., MENG X., WANG J.: New intersection algorithm of convex polygons based on voronoi diagrams. pp. 224–231. 6

[Yuk15] YUKSEL C.: Sample elimination for generating poisson disk sample sets. *CGF 34*, 2 (2015), 25–32. 3

[YWW14] YAN D.-M., WALLNER J., WONKA P.: Unbiased Sampling and Meshing of Isosurfaces . *IEEE TVCG 20*, 11 (2014). 3

[YXSH13] YING X., XIN S.-Q., SUN Q., HE Y.: An intrinsic algorithm for parallel poisson disk sampling on arbitrary surfaces. *IEEE TVCG 19*, 9 (2013), 1425–1437. 3