Visibility-Aware Progressive Farthest Point Sampling on the GPU

Sascha Brandt and Claudius Jähn and Matthias Fischer and Friedhelm Meyer auf der Heide

1 Heinz Nixdorf Institute, Paderborn University, Germany
2 DeepL GmbH, Cologne, Germany

Abstract
In this paper, we present the first algorithm for progressive sampling of 3D surfaces with blue noise characteristics that runs entirely on the GPU. The performance of our algorithm is comparable to state-of-the-art GPU Poisson-disk sampling methods, while additionally producing ordered sequences of samples where every prefix exhibits good blue noise properties. The basic idea is, to reduce the 3D sampling domain to a set of 2.5D images which we sample in parallel utilizing the rasterization hardware of current GPUs. This allows for simple visibility-aware sampling that only captures the surface as seen from outside the sampled object, which is especially useful for point-based level-of-detail rendering methods. However, our method can be easily extended for sampling the entire surface without changing the basic algorithm. We provide a statistical analysis of our algorithm and show that it produces good blue noise characteristics for every prefix of the resulting sample sequence and analyze the performance of our method compared to related state-of-the-art sampling methods.

CCS Concepts
• Computing methodologies → Rasterization; Point-based models; • Theory of computation → Computational geometry;

1 Introduction
Sampling of 3D surfaces is an essential technique in computer graphics with many applications, e.g., rendering, global illumination, object distribution, remeshing, texture synthesis, and Monte Carlo integration. One of the most important properties of sampling techniques is the blue noise property, i.e., a point distribution with high spacial uniformity and low regularity. This corresponds well with natural phenomena (e.g., retina cell distribution) and has a certain aesthetic appeal as blue noise sampling techniques tend to replace low frequency aliasing with high frequency noise [LWSF10; XHGL12]. A well-known distribution that has blue noise characteristics is the Poisson-disk distribution. A Poisson-disk distribution is a uniform, yet unstructured distribution of points where no two samples are placed closer together than its conflict radius r. It is called maximal if every point in the sample domain is within range r to the samples, i.e. disks of radius r placed at the sample locations cover the entire domain without gaps. Pro-
gressive samplings additionally have the property that they are ordered in a sequence such that every prefix of the sequence is a Poisson-disk distribution for some \( r \). This has the great advantage that we can get a multi-resolution sampling scheme, without needing to resample the surface again, by simply selecting a prefix from the sample sequence. Furthermore, this allows for a precise control over the number of samples, which is often difficult to achieve with traditional Poisson-disk sampling methods.

There are multiple ways of achieving progressive samplings with blue-noise characteristics. One way is to gradually decrease the disk radius used for Poisson-disk sampling [MREB12; MF92], which is prone to introduce sampling biases. Another way is to progressively eliminate samples from a sample set and order them in the reverse order of elimination [Yuk15]. A third way is the farthest point strategy, which our proposed method belongs to. Here, samples are added one-by-one from a sample set such that each sample maximizes the minimal distance to all previously chosen samples. In recent years, sampling methods that run on the GPU have gained a lot of popularity because of the massive parallelism which potentially allows to get high quality samplings in real-time. One core difficulty for blue-noise sampling on the GPU is managing the required spacial data structures and resolving conflicts of overlapping samples. For progressive samplings, another difficulty is that every sample in the sample sequence highly depends on the previous samples in that sequence. This adversely affects the parallelizability of sampling algorithms. To our knowledge, we are the first to propose a progressive sampling method for surfaces that runs entirely on the GPU.

We do this by reducing the problem of 3D surface sampling to a set of parallel 2.5D problems which we can efficiently sample using the GPUs rasterization hardware. The main idea is to render the surface data of an object from multiple directions into a set of G-buffer images and progressively compute the discrete 3D Voronoi diagram based on these images from which we choose multiple samples in parallel and order them by decreasing sample radius (Fig. 1a). The rasterization hardware allows us to efficiently process the neighborhood of a sample, without the need for explicit neighborhood searches, by simply rasterizing a circle into the G-buffer.

Our method has the following advantages:

- **High performance.** Utilizing the built-in 2D rasterization hardware of modern GPUs allows for very efficient sampling with interactive to real-time performance.
- **Object independent.** By sampling on a set of prerendered images, the sampling becomes almost independent of the complexity and type of the sampled object, i.e., we can sample triangle meshes as well as implicit surfaces of (almost) arbitrary complexity.
- **Progressive.** The progressive nature of our algorithm allows for a precise control of the number of samples with the possibility of dynamic adjustment of the sample density.
- **Visibility aware.** Using the rasterization hardware we are easily able to only sample the visible surfaces, which is especially useful for level-of-detail applications.
- **Easy implementation.** The proposed sampling method uses the standard rasterization pipeline together with a few simple compute shaders which are easy to implement and can be easily integrated into existing rendering pipelines.

2. Related work

We briefly review existing blue-noise sampling techniques. Most papers classify sampling algorithms into Poisson-disk sampling and its variations, relaxation-based sampling, and tile-based sampling. We further distinguish progressive sampling as a separate type, where each prefix of the sample list has good blue-noise properties. For a more detailed overview of blue-noise sampling and its applications we refer to the survey by Yan et al. [YGW*15].

**Poisson-Disk Sampling**

Many blue-noise sampling algorithms fall into the Poisson-disk sampling category with some variation of dart-throwing. A Poisson-disk sampling is typically characterized by its conflict radius \( r \), with the property that no two samples lie within this distance. It is called maximal if the disks of radius \( r \) placed at the sample locations, cover the whole sampling domain. Dart-throwing algorithms place random samples in the sample domain and accept or reject samples based on the distance to existing samples [Coo86]. This has also been adapted to surfaces [CJW*09]. These algorithms usually suffer from bad convergence rates and there has been a lot of work into optimization, e.g., by tracking and sampling the empty space between discs [Jon06; EMP*12; YW13; YGYZ15]. The first parallelized algorithm for Poisson-disk sampling was proposed by Wei [Wei08] which was extended to surfaces by Bowers et al. [BWWM10]. They use so called phase groups to parallelize the dart-throwing process and avoid conflicts. Ying et al. [YSXH13] assign each sample a priority to avoid using phase groups for parallelization. A method that utilizes the rasterization hardware for generating 2D Poisson-disk samples on the GPU was proposed by Ip et al. [IYLV13], which we adapted for our algorithm to resolve conflicts and draw the discrete Voronoi diagram. They use a geometry shader to draw discs from point primitives and use the depth buffer to resolve conflicts between overlapping discs.

**Relaxation-Based Sampling**

Another common strategy for achieving blue-noise samplings is relaxation-based sampling, which is usually based on some form of Lloyd iterations [Llo82]. Here, the core idea is to start with an arbitrary sampling and iteratively relocate the samples to their proper positions (relaxation) until the desired blue-noise quality (or other objective function) is satisfied. For the relocation, a Voronoi tessellation [BSD09], or the dual Delaunay triangulation [Xu11], can be used. This process has also been extended to surfaces by Xu et al. [XHGL12]. Schlömer et al. introduced farthest-point optimization [SHD11] where in each iteration the minimal distances between points are maximized. Their method was later parallelized by Chen et al. [CCW12] and generalized to surfaces by Yan et al. [YGI*14]. The resulting point set exhibits excellent blue-noise properties. Ahmed et al. [AGY*17] introduced an optimization-based algorithm that allows enforcement of spacial constraints, e.g., coverage of the sampling domain. A recent relaxation-based algorithm that runs on the GPU was proposed by Zayer et al. [ZMS18]. They relax a quasi-Voronoi tessellation over multiple iterations on
the GPU to generate natural tessellations on surfaces. In general, relaxation-based methods allow for precise control over the number of sample points. However, all these methods rely on iterative optimization/relaxation, which is not suitable for real-time applications due to bad convergence rates.

**Tile-Based Sampling**

To speed up the sampling process, an often used technique to generate Poisson-disk samplings is to use tiles of precomputed Poisson-disk patterns. Important work includes [CSHD03; LD05; KCODL06]. The tiles are placed in the sampling domain in a clever way such that the samples in different tiles don’t conflict which each other. This allows for a very fast generation of lower quality Poisson-disk patterns. Another category of tile-based sampling techniques uses a recursive subdivision of the sampling domain into self-similar tiles, like Polyominoes [Ost07]; Polyhexes [WPC+14], or Adaptive Regular Tiles [ANHD17], where each tile is associated with a single sample. These sampling techniques are also inherently progressive. In a more recent work, Wang et al. [WS18] proposed a very fast tile-based sampling method for 3D surfaces on the GPU. They use a set of orthogonal 2D planes with blue-noise patterns which are progressively projected on a mesh surface using ray tracing to generate a 3D Poisson-disk sampling. For conflict resolution they first resolve intra-batch conflicts between different projector planes using a simple priority based scheme, and then solve global conflicts on the mesh using a hash grid. Although their algorithm is fast, it still suffers from bad convergence rates and sample count prediction as it is basically a dart-throwing based method. Our algorithm uses similar ideas in the sense that we use static projector planes to initially sample the surface of an object.

**Progressive Sampling**

We call a sampling progressive when the samples are ordered in a list such that each prefix of the list has good blue-noise properties. We could achieve a progressive sampling by gradually decreasing the disk radius used in Poisson-disk sampling [MREB12; MP92]. However, the performance and quality highly depends on the update strategy for the radius and is prone to introduce sampling biases. Another way is Weighted Sample Elimination as proposed by Yüksel [Yuk15]. Given an arbitrary sampling, they assign weights to each sample based on the distance to their nearest neighbor and progressively remove the samples with the highest weight. A third way is the Farthest Point Strategy where each sample is progressively selected from the sampling domain such that it is as far as possible from all previously chosen samples according to some metric. These sequences are also called Greedy Permutations and have been excessively studied in theoretical computer science because of their versatile properties [Gon85; HPM06; EHS15]. An important property of Greedy Permutations, which we utilize in our algorithm, is that they are maximal for each prefix. The best candidate algorithm by Mitchell [Mit91] can be considered an approximate Greedy Permutation. In the context of computer graphics, the farthest point strategy was first studied by Eldar et al. [ELPZ97] for image compression where they used the discrete Voronoi diagram to find the sample with the farthest distance to all previous samples. They also showed its good blue-noise quality. An equivalent formulation of this algorithm, based on Delaunay triangulations, was proposed by Kanamori et al. [KSN11]. For 3D surface samplings, Moenning and Dodgson [MD03] proposed a method based on Fast Marching [MS96]. They embed the points in a Cartesian grid and progressively compute the Bounded Voronoi Diagram by simultaneously propagating fronts along a thin offset band. A main inspiration of our proposed method is the Greedy Cluster algorithm by Har-Peled and Mendel [HPM06]. The basic idea is to store the locally farthest point for each selected sample in a heap, extract the new sample with maximum distance and only update the local neighborhood of that sample for the next iteration. More recently, Ma et al. [MGY*18] introduced Instant Stippling. They use a precomputed Incremental Voronoi Set (basically a Greedy Permutation) on a toroidal domain, which they map into UV space of an object, to generate stippling patterns in real time.

### 3. Algorithm Overview

We will now give a short overview of our progressive farthest point sampling algorithm and will go into more detail for each phase of the algorithm in Section 4.

The basic idea of the overall process is outlined in Algorithm 1. Our goal is to incrementally compute the discrete 3D Voronoi diagram over a large set of initial surface samples, which we acquire through rasterization of the surface object \(O\). In each iteration, we associate one new sample per Voronoi cell by independently selecting the farthest point in each cell and add them to the sample list \(S\). We then eliminate newly added samples that are too close together and sort them by their distances to their Voronoi sites to get a progressive sample sequence. The result is an ordered list \(S\) of 3D sample points where every point is (approximately) the farthest point from all previous points in that list.

The initial sample set is stored as an array of G-buffer images \(I\), which we acquire through rasterization of the 3D surface data from object \(O\) from multiple directions \(D\) (Section 4.1). The Voronoi diagram is also stored as an array of G-buffer images \(V\), where each

---

**Algorithm 1** Progressive Farthest Point Sampling

**Input:** Surface object \(O\), target count \(n\), resolution \(R\), directions \(D\)

**Output:** Progressive sample sequence \(S\)

1. \(S \leftarrow \emptyset\), \(m \leftarrow 1\)
2. \(I \leftarrow \text{RASTERIZE}\(\text{SURFACE}(O, R, D)\)
3. \(S[0] \leftarrow \text{CHOOSERANDOM}\(\text{SAMPLE}(I)\)
4. \(\text{while } m < n\) do
5. \(\text{for all directions } D[i] \text{ in parallel do}\)
6. \(V[i] \leftarrow \text{DRAWVORONOI}(I[i], D[i], S[0 : m])\)
7. \(\text{end for}\)
8. \(S[m : 2m] \leftarrow \text{EXTRACTFARDEST}\(\text{PERCELL}(V, S[0 : m])\)
9. \(\text{for all directions } D[i] \text{ in parallel do}\)
10. \(P[i] \leftarrow \text{DRAWPOISSON}\(\text{DISKS}(I[i], D[i], S[m : 2m])\)
11. \(\text{end for}\)
12. \(S[m : 2m] \leftarrow \text{ELIMINATE}\(\text{CONFLICTS}(P, S[m : 2m])\)
13. \(S[m : 2m] \leftarrow \text{COMPACT}\(\text{AND}\text{SORT}(S[m : 2m])\)
14. \(m \leftarrow m+1\)
15. \(\text{end while}\)
pixel stores the id and distance to its Voronoi cell. We begin the sampling process by selecting a single random sample from \( I \) which we add to our sample sequence \( S \). We then iteratively perform the following steps until the desired sample count \( n \) is reached:

1. Compute the discrete 3D Voronoi diagram over all samples in \( I \) by computing the id and distance (Euclidean or Geodesic) of each sample in \( I \) to its closest site in \( S \) (Section 4.2).
2. For each site in \( S \), extract the sample from its 3D Voronoi cell with the maximum distance to its site and append it to \( S \), unless the distance falls below a certain threshold relative to the total maximum distance (Section 4.3).
3. Throw Poisson-disk darts onto each slice of \( I \) for each new sample in \( S \) using the distances to their Voronoi site as disk radius.
4. Resolve conflicting disks by accepting the sample with the largest disk radius (Section 4.4).
5. Compact the newly accepted samples in \( S \) and sort them by decreasing disk radius (Section 4.5).

The entire sampling process is also visualized in Figure 1a. Figure 2 shows a closeup of a single sampling iteration. Each of these steps are parallelizable on the GPU and are described in more detail in Section 4. The core idea, for the steps 1 & 3 (lines 7 & 11 in Algo. 1), is to use the rasterization hardware to reduce the problem of computing a 3D Voronoi diagram and performing the Poisson-disk step to a 2.5D problem. To further speed up the entire process, we also use a hierarchical sampling scheme based on a MIP hierarchy on the rasterized G-buffers, which is described in Section 4.6.

### 4. Progressive Sampling on the GPU

We will now present our progressive farthest point sampling algorithm in more detail. We begin by describing the necessary data structures we use throughout the algorithm. These need to be created only once and can be reused when sampling multiple objects.

As already mentioned above, we use the rasterization hardware to reduce the problem of computing a 3D Voronoi diagram and performing the Poisson-disk step to a 2.5D problem. For this, we need to allocate the corresponding layered G-buffer images; at least one for the 3D surface samples \( I \), two for the Voronoi diagram \( V \) (distance & id), and two for the Poisson-disk step \( P \). The resolution and number of array layers is the same for each of the G-buffers and depends on the initial rasterization step (Sec. 4.1). The format of the G-buffer images is described in the corresponding sections (4.1, 4.2, 4.4). We also allocate a MIP hierarchy for each of the G-buffers to improve the overall performance of each step (Sec. 4.6).

The Voronoi and Poisson-disk step (Sec. 4.2, 4.4) is done by rasterizing disks for each sample into each layer of the G-buffers and write either the Voronoi distance or the disk radius into the depth buffer. We can then find the farthest sample in each Voronoi cell using simple atomic operations (Sec. 4.3). The sample sequence \( S \) is stored as a buffer of size \( 2n \), where each sample is packed into a single 64-bit unsigned integer, with the sample radius stored in the most significant 32 bits and the texture coordinates, of where the samples originated from, into the least significant 32 bit. The texture coordinates are packed into a 12:12:5:3 format: 2 × 12 bits for the u,v-coordinates (integers), 5 bits for the texture layer, and 3 bits for the MIP level (Sec. 4.6). This allows for a G-buffer resolution of up to \( 4096 \times 4096 \) with 32 layers and 8 MIP levels. We copy the actual samples (3D position, normals, color, etc.) from the G-buffers into a vertex buffer after the sampling process is finished, although it can be easily done after each iteration. The directions \( D \) used in the rasterization step are stored in form of projection matrices in a single uniform buffer to be accessible in the other steps.

### 4.1. Sampling the Visible Surface by Rasterization

The initial step of our sampling method is a pre-sampling step that samples an object using rasterization. The goal is to capture the visible surface, as seen from outside the object, as good as possible. For this we place a set of multiple orthogonal projection planes around the object and then render it in a single pass, using layered rendering, into a set of G-Buffers, which encode the surface properties of the samples objects (3D positions, normals, colors, etc.; see Figure 3). For only the sampling process, the 3D positions are sufficient, which we encode in the 32-bit RGB channels of the G-buffer image. We use the alpha channel to mark a sample as valid, i.e., whether the corresponding pixel is part of the object or the background. The number of projection planes should be chosen such that most of the visible surface is covered. We are aware that it is very difficult to actually capture the entire visible surface using this method. However, using between 8-32 planes uniformly distributed around the object is usually enough to capture a reasonable amount of the visible surface. We use a pre-generated low-discrepancy sequence of samples on the unit sphere to dynamically distribute any conflicts of the samples objects (3D positions, normals, color, etc.) from the G-buffers to improve the overall performance of each step (Sec. 4.6).
The number of planes and the resolution of the G-buffers highly influences run-time and quality of the sampling process. Both parameters should be chosen according to the available hardware capabilities. In our experiments, we mostly used 16 planes with a resolution of $1024 \times 1024$ which empirically proved to be a good compromise between quality and speed (see Section 5.5). In general, the resolution should be high enough according to the expected sampling density for the desired sample count. We can estimate the required resolution by $2^{\lceil \log_2 \frac{D}{r_{\text{max}}} \rceil}$, where $D$ is the diameter of the bounding box and $r_{\text{max}}$ the densest packing radius defined as $r_{\text{max}} = \sqrt{\frac{\pi}{2}}$ (see [LD08]). $A$ is the area of the sampling domain and $N$ the number of desired samples.

### 4.2. Computing the Discrete Voronoi Diagram

Each iteration begins by computing the discrete 3D Voronoi diagram over all previously selected samples. In the first iteration, this is a single sample randomly selected from the position G-buffer $I$. Alternatively, one could start with any Poisson-disk sampling as long as it is maximal. The Voronoi diagram is stored in form of one layered G-buffer which stores the ids of the Voronoi cells (32-bit integer) and one G-buffer that stores the distance to the closest Voronoi site (32-bit float) for each sample in the position G-buffer $I$.

Using the rasterization hardware for computing the discrete Voronoi diagram is a well known process in computer graphics [HICK*00]. The basic idea is to rasterize hyperboloids for each sample and use the depth buffer to compute the Voronoi cells. Instead of computing the 3D Voronoi diagram slice-by-slice in a 3D grid, we compute it directly on the layers of the G-buffers. Furthermore, instead of rasterizing hyperboloids, we rasterize disks, in a similar fashion as [JP*10], and compute the Voronoi distances in a fragment shader, where they are written into the depth buffer. We reuse the Voronoi G-buffers from previous iterations and only render the disks for newly added samples, to improve the performance. Since we use a form of the Farthest Point Strategy, we can assume that the sampling is maximal and it is enough to use the current conflict radius $r$ as disk radius to avoid gaps in the Voronoi diagram. However, as we use only an approximate method, we use a slightly larger radius, i.e., the sample radius $r_i$ (computed in previous iterations; see Section 4.3) stored for each sample $s_i$ in the sample buffer $S$ (i-th sample in $S$). The shader pipeline for computing the Voronoi diagram is also illustrated in Figure 4.

In a vertex shader, we first read the 3D position of each sample $s_i$ from the position G-buffer, using the texture coordinates stored in the sample buffer $S$, and pass it, together with its sample radius $r_i$, to the geometry shader. In the geometry shader we emit equilat-
between run-time and quality (Sec. 5.2). Throwing away samples can be simply done by writing a 64-bit 0 into the sample buffer.

4.4. Conflict Removal using Poisson-disk Dart-throwing

Since we selected each sample from the Voronoi cells independently, it is highly likely that some samples are too close together, located at the shared Voronoi vertices (see, e.g., Figure 1a (2)). To resolve such conflicts we need to run a single Poisson-disk dart-throwing test and remove overlapping samples. For this, we again use almost the same rasterization method as described in the Voronoi step (Section 4.2) for each new sample. However, instead of writing the distance into the depth buffer, we write the sample radius stored in the MSB, and set the depth comparison function to greater-than to get the sample with the biggest radius. The used Poisson G-buffers \( P \) (for index and sample radius) are identical to the Voronoi G-buffers.

In a compute shader, we then run one thread for each new sample, where we read the index stored in the Poisson G-buffer at the samples position (using the stored texture coordinates) and compare it with the index of the sample buffer. When they don’t match, we reject the sample by writing 0 into the sample buffer. Otherwise the thread simply returns.

4.5. Compact & Sort

Our sample buffer \( S \) now contains multiple discarded samples scattered in the index range \([m, 2m]\), where \( m \) is the sample count of the previous step. We need to compact all non-zero samples and sort them by decreasing sample radius to achieve a progressive sample sequence. Since we stored the samples as 64-bit unsigned integers with the sample radius stored in the MSB, we can do this by a simple parallel radix sort on the 32 most significant bits. We use our own GLSL compute shader implementation of the radix sort algorithm by Satish et al. [SHG09] due to its simplicity and efficiency.

After this, we update the sample count \( m \) and continue with the next iteration, or stop if we have enough samples.

4.6. Hierarchical Sampling

Although the method above works already very well, it can be quite expensive, especially in the first few iterations. This is because we need to draw large disks, that cover the entire surface, at a high resolution. To further increase performance, we use a hierarchical sampling scheme based on a MIP hierarchy over the G-buffers.

For this, we re-compute the required image resolution after each iteration, similar as in Section 4.1, using the minimum sample radius so far instead of \( r_{\text{max}} \). From the resolution we choose the corresponding MIP level of all involved G-buffers and use it for this iteration. The only drawback is, that we need to recompute the entire Voronoi diagram when we switch between MIP levels.

Directly after the initial step, where we render an object into each layer of the G-buffers, we compute the MIP pyramid of each G-buffer based on the highest resolution. Here, we need to guarantee that each sample in the MIP pyramid lies on the surface of the sampled object. Instead of averaging the samples from lower levels, we use the medoid of the samples to guarantee this, i.e., the sample that closest to the average. Alternatively, we could simply re-render the object for each iteration into the used MIP level, which would increase the total computation time. The number of MIP levels is limited by 8, since we only use 3 bits to address the MIP level in the sample buffer \( S \) (Section 4).

5. Results and Analysis

We will now analyze the quality and performance of our progressive sampling algorithm. For statistical analysis we use the well-established radius statistics [LD08] and differential domain analysis [WW11], a form of spectral analysis suitable for surfaces. We show that our method can produce sampling sequences with high-quality blue noise characteristics for every prefix, which is close to a reference progressive sample sequence and better than sequences produced by sample elimination. As a reference we used the Greedy Cluster algorithm by Har-Peled and Mendel [HPM06], as it produces a maximal Greedy Permutation of the sample set, i.e., every sample in the sequence is the farthest from all previous samples.

We will also evaluate the performance of our method compared
We implemented our algorithm in our experimental rendering with a state-of-art GPU Poisson-disk sampling method, that does not compute a progressive order.

5.1. System Specification

We implemented our algorithm in our experimental rendering framework PADrend [EJP11], which is based on OpenGL. All tests were performed on a workstation PC with an Intel Core i7-7700K CPU and a NVIDIA GTX 1080 Ti graphics card. The source code is available as part of the BlueSurfels plugin of PADrend.

5.2. Radius Statistics

Radius statistics is a well-established measure which describes how densely a sample set is packed and therefore is an indicator of the quality of the distribution [LD08]. It is defined as the radius ratio $\rho = r / r_{\text{max}}$ of the minimum half distance between any two points of a sample set and $r_{\text{max}} = \sqrt{\frac{2}{3N \lambda}}$, the maximal possible radius to densely pack $N$ disks in a 2D domain of surface area $\Lambda$. Lagae and Dutré recommend $\rho$ values between 0.65 and 0.85 for high quality Poisson-disk distributions [LD08]. We will also use the mean radius ratio $\bar{\rho} = \bar{r} / r_{\text{max}}$, where $\bar{r}$ is the average half distance of each sample to its nearest neighbor.

Figure 5 shows the radius ratio $\rho$ (top) and the mean radius ratio $\bar{\rho}$ for every prefix of a sample sequence of 1M points sampled from the Stanford Bunny model [McG17] using our method compared to the reference (Greedy Cluster) and to Weighted Sample Elimination [Yuk15]. For the reference and Weighted Sample Elimination, we used the same sample set extracted from the G-Buffers of our method, for which we used 16 directions with a resolution of $1024 \times 1024$. The resulting sample set consisted of $\sim 6.3M$ samples.

We can see that both the radius ratio $\rho$ and the mean radius ratio $\bar{\rho}$ are close to the reference for the most part. If we look at the mean radius ratio $\bar{\rho}$ of our method, the quality is much smoother while still close to the reference. Apart from a few outliers, our method produces $\rho$ values that clearly lie in the targeted range of 0.65 and 0.85 which is a first indicator of the good blue noise quality of our method. Although Weighted Sample Elimination can produce higher quality samplings for the target count of 1M samples (compared to our method and the reference), the quality of the entire sequence in general is much better with our method.

The sawtooth pattern of our method comes from the individual
iterations where we select multiple samples at once and then sort them by decreasing radius. Another influencing factor is the radius threshold. We discard samples that fall below a radius threshold of 75% relative to the maximum radius in each iteration (see Section 4.3). The influence of this factor on \( \rho \) (and the run-time) is shown in Figure 6. We chose a threshold of 75% as a good compromise between run-time and quality.

5.3. Differential Domain Analysis

Another important analysis tool for blue noise sampled surfaces is Differential Domain Analysis (DDA) [WW11]. The DDA spectrum, together with the derived radial mean and anisotropy, can give information about the uniformity of the sample set and reveal structural artifacts and biases.

Figure 7 shows the Differential Domain Analysis of our method (top) for different sample prefixes \( n \) of the Stanford Bunny model compared to the reference (Greedy Cluster) (bottom). We can clearly see the typical radial pattern which is characteristic for blue noise distributions. In comparison, there seems to be no major differences between our method and the reference. At a prefix length of \( n = 500k \) and \( n = 1M \), we begin to see regular patterns and a decline in anisotropy for both, our sampling and the reference, for our sampling and the reference. These can be addressed to the relatively low resolution of \( 1024 \times 1024 \) in the initial rasterization step. We could easily reduce these patterns by increasing the resolution or the number of projection planes, in exchange for decreased performance (see Section 5.5). But, in our opinion, the quality of the sampling is still reasonably good.

5.4. Performance

We will now compare the performance of our algorithm with two CPU based algorithms, Greedy Cluster [HPM06] (our reference) and Weighted Sample Elimination [Yuk15], which both compute a progressive sample sequence similar to ours. We will also compare it with Progressive Sample Projection by Wang and Suda [WS18], a recent GPU algorithm for Poisson-disk sampling on mesh surfaces, that does not compute a progressive order.

Figure 8 shows the results of the comparisons for the Bunny model with a target sample size of 1M samples. We measured the total time after each iteration of our algorithm and the reference. For Weighted Sample Elimination, we only measured the total time to compute a progressive sample sequence of 100, 1k, 10k, 100k, and 1M samples from an initial sample set of \( \sim 6.3M \) samples, which we first reduced to 5 times the target sample count by random sampling. For Progressive Sample Projection, we recomputed the Poisson-disk sampling for each iteration of our algorithm and used its current radius ratio \( \rho \) as input to the Progressive Sample Projection algorithm, to get a fair comparison. A selection of the timings in Figure 8 can also be seen in Table 1. We can see that the speed of our algorithm is magnitudes faster than the CPU algorithms and even outperforms the GPU algorithm which only computes a Poisson-disk sampling without progressive ordering. We can compute an ordered sample sequence of 1M points in about 200ms, which corresponds to a sampling rate of 5 million samples per second. However, we should note here, that we used our own implementation of Progressive Sample Projection based on compute shaders and using the rasterization variant, as described by the authors, which can potentially reduce the overall performance, especially for higher sample counts. Using a more optimized implementation will probably perform much closer to our algorithm.

Table 1: Selected timings from Figure 8 in tabular form.

| Samples  | 100  | 1000 | 10000 | 100000 | 1000000 |
|----------|------|------|-------|--------|---------|
| Ref.     | 1.57s| 2.03s| 2.52s | 3.65s  | 10.16s  |
| Ours     | 16.2ms| 21.27ms| 38.18ms| 83.23ms| 192.0ms |
| Yuk15    | 0.45s| 0.45s| 0.61s | 3.42s  | 52.68s  |
| WS18     | 53.42ms| 50.58ms| 198.94ms| 1.44s  | 6.35s   |

Figure 9: Timings for each iteration of our algorithm. Each bar is divided into the 4 phases of each iteration (voronoi computation; extracting the farthest sample; resolving conflicts; compact & sort). The first bar is the rasterization time for the initial step.
5.5. Choice of Sampling Parameters

Our algorithm uses multiple input-parameters that can greatly influence the quality and run-time of our algorithm. The main parameters that influence the quality, are the number of directions during the initial rasterization step, and the sampling resolution. Figure 10 shows a matrix of different values for the number of directions and resolution measured from sampling the Bunny model with 100000 samples. When the resolution or direction count is low, we can see regular artifacts in the sample spectrum. When the resolution or direction count is high, the run-time is significantly higher. From this data, we recommend sampling from 16 directions with a resolution of 1024×1024, since it seems to be the best compromise between run-time and quality. Of course, the requirements can change for different sample counts and sampled objects. The data shown in Figure 10 should only be used as a guide to find the right parameters for the intended sampling purpose.

6. Extensions

We will now briefly discuss a few useful extensions of our sampling algorithm to handle different use cases.

6.1. Sampling the whole Surface

One might argue that a great disadvantage of our sampling algorithm is that we only sample the visible surface of an object. However, with a slight modification of the initial rasterization step, we can sample complete surfaces. For this, we can use a depth peeling approach for each rendered direction (e.g., [BM08]). Instead of rendering one image for each direction, we allocate a fixed number of additional G-buffer layers for each direction and render the object in multiple passes. In each pass, we render into one layer of the G-buffer and use the depth buffer from previous passes to discard fragments that are closer than the previous depth value. This way we can capture the entire surface layer-by-layer. The rest of our sampling algorithm doesn’t have to be changed for this approach.

A drawback of this method is that we might need a large number of layers, which can be difficult to predict and highly influences the runtime of our algorithm. Using similar techniques, that are normally used for order-independent transparency, might be helpful here. An example of a sampled object with our original approach compared to the depth-peeling approach is shown in Figure 11. In both cases the sampling time took ~85ms for 100000 sample.
6.2. Adaptive Progressive Sampling

Our method can also be extended to adaptive progressive sampling, i.e., a progressive sampling where we associate each sample with a weight which dictates the density of the sampling at this position. We do this by rasterizing weights, of the surface model that we want to sample, into a separate set of G-buffers during the initial rasterization step (Sec. 4.1). The weights can be stored in a surface texture or computed in a shader (e.g., surface curvature). We then read the weights from the G-buffers during the Voronoi and Poisson-disk step (Sec. 4.2, 4.4) and use them in a modified distance function.

Figure 12 shows examples of adaptively sampled objects based on precomputed ambient-occlusion maps for the weights. Darker areas have a higher density of samples than lighter areas.

7. Applications

We will now shortly outline some of the many applications of progressive sampling.

7.1. Continuous Level-of-Detail

One advantage of a progressive sampling is that any selected number of samples from the sample sequence are well distributed and cover the entire surface of the sampled object when placing disks of fixed radius at each sample. We can use this property for a simple but efficient point-based continuous level-of-detail method. The idea is to replace complex objects in a rendered 3D scene with sets of oriented 3D disks (surfels) rendered from progressive sample sequences. For this, we sample each object of a 3D scene and store the sample sequence as a vertex buffer referenced by the sampled object. During an interactive walk-through of the scene, we switch between original geometry and sample representations based on the camera position. We choose the rendered sample prefix for each object such that roughly every pixel of the rendered object is covered by at least one surfel disk without too much overlap.

Figure 13 shows a complex example scene consisting of a total of ~1 Billion triangles, rendered with this method. The scene consists of 27 Power plants (~12M triangles each [McG17]), 23 Emerald Squares (~6M triangles each [HB17]), ~10k trees (6 variations; 40k – 80k triangles each), and 1024 terrain tiles (~7k triangles each). The overall sampling took ~78 seconds.

7.2. Object Placement on Surfaces

Another application where a progressive sample sequence can be useful, is the placement of multiple classes of objects on a surface, where a visually pleasing layout without regularity artifacts is important. Figure 14a and 14b show the distribution of plants of varying sizes and species distributed on a sphere using progressive samples. Here, we emulate plant growth [DHL*98] by placing large trees at the first few samples and smaller trees at the following samples. Later in the sample prefix, we also place fern and grass. The trees in the scene of Section 7.1 were placed in a similar way (Fig. 14c). Here, we placed the trees using weighted sampling with a binary weight according to the slope of the terrain and a Perlin-noise function.

Figure 14d shows a different example, where we guide the placement of textures by using the Poisson-disk samples to project different shapes onto the surface of the object. We varied the shapes and sizes by selecting different prefixes of the sample sequences.

8. Discussion and Conclusion

We introduced a new progressive farthest point sampling algorithm for 3D surfaces that runs entirely on the GPU. We showed that...
we can produce high-quality progressive sample sequences in very short time.

Our algorithm can sample almost any type of object that can be rasterized on the GPU. However, objects that are very long in one dimension are not well represented when we rasterize them using a square viewport. The G-buffers will contain a lot of unused area and the resulting sample quality is possibly decreased. We might solve this by rasterizing the object using multiple low resolution tiles that only show a portion of the object. But, it might be difficult to correctly place the tiles such that the entire visible surface is covered. Similarly, we can place the viewpoints inside of an object (e.g., the interior of a room) and sample the surfaces “inside-out”. In this case we need to find an optimal placement of viewpoints such that every corner of the room is covered (art-gallery problem).

Another limitation of progressive farthest point sampling methods is, that we cannot easily control the overall quality of the sampling. For applications where high-quality samples are more important than the progressive nature, other existing sampling algorithms are probably more suitable. A possible extension of our algorithm could be to run a relaxation step after each iteration to improve the quality of the sampling.

Although the Euclidean distance metric is often sufficient for surface sampling, some applications require a sample distribution based on the Geodesic distance metric. Our algorithm currently supports only approximate Geodesic distances based on the surface normals [BWWM10]. For applications where exact Geodesic distances are required (e.g., Monte Carlo integration), this might not be sufficient. We could improve the approximation by propagating wave-fronts on the rendered image views. But, this would not guarantee that we can compute the exact Geodesic distances as we might hit discontinuities on the surface.

In future work, we want to explore further applications of our sampling method. We are working on integrating our method in a complete rendering system that allows on-the-fly approximation for real-time rendering of very large complex scenes. Another interesting application could be progressive remeshing of surfaces. By combining each iteration of our sampling algorithm with a triangulation step, we can potentially achieve multi-resolution meshes based on the sample prefix. It might also be interesting, how our method can perform in global illumination applications.

Acknowledgements
Most models used in this paper were downloaded from the McGuire Computer Graphics Archive [McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17]. The Emerald Square model and trees in Figure 14a were downloaded from the NVIDIA Emerald Square, McG17].

References

[AGY*17] AHMED, ARBALLA G. M., GUO, JIANWEI, YAN, DONG-MING, et al. “A Simple Push-Pull Algorithm for Blue-Noise Sampling”. IEEE Trans. Vis. Comput. Graph. 23.12 (2017), 2496–2508. DOI: 10.1109/TVCG.2016.2641963.

[ANHD17] AHMED, ARBALLA G. M., NIESE, TILL, HUANG, HUI, and DEUSSEN, OLIVER. “An adaptive point sampler on a regular lattice”. ACM Trans. Graph. 36.4 (2017), 138:1-138:13. DOI: 10.1145 / 3072959, 3073588 3.

[BM08] BAVOIL, LOUIS and MYERS, KEVIN. Order Independent Transparency with Dual Depth Peeling. Tech. rep. Santa Clara, CA, USA: NVIDIA Corporation, Feb. 2008.

[BSD09] BALZER, MICHAEL, SCHLÖMER, THOMAS, and DEUSSEN, OLIVER. “Capacity-constrained point distributions: a variant of Lloyd’s method”. ACM Trans. Graph. 28.3 (2009), 86:1–86:8. DOI: 10.1145/1531326.1531392.

[BWWM10] BOWERS, JOHN, WANG, RUI, WEI, LI-YI, and MALEZT, DAVID. “Parallel Poisson Disk Sampling with Spectrum Analysis on Surfaces”. ACM Trans. Graph. 29.6 (Dec. 2010), 166:1–166:10. ISSN: 0730-0301. DOI: 10.1145/1862261.1866188 2, 11.

[CCW12] CHEN, ZHONGGUI, CAO, JUAN, and WANG, WENPING. “Isotropic Surface Remeshing Using Constrained Centroidal Delaunay Mesh”. Comput. Graph. Forum 31.7–1 (2012), 2077–2085. DOI: 10.1111/j.1467-8659.2012.03200.x 2.

[CFW*09] CLINE, DAVID, JESCHKE, STEFAN, WHITE, K., et al. “Dart Throwing on Surfaces”. Comput. Graph. Forum 28.4 (2009), 1217–1226. DOI: 10.1111/j.1467-8659.2009.01499.x 2.

[CO06] COOK, ROBERT L. “Stochastic Sampling in Computer Graphics”. ACM Trans. Graph. 5.1 (1986), 51–72. DOI: 10.1145 / 7529.8927 2.

[CSHD03] COHEN, MICHAEL F., SHADE, JONATHAN, HILLER, STEFAN, and DEUSSEN, OLIVER. “Wang tiles for image and texture generation”. ACM Trans. Graph. 22.3 (2003), 287. DOI: 10.1145 / 882262. 882283.

[DH*98] DEUSSEN, OLIVER, HANRAHAN, PAT, LINTERMANN, BERND, et al. “Realistic Modeling and Rendering of Plant Ecosystems”. Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH ’98. ACM, 1998, 275–286. DOI: 10.1145/280814.280898 10.

[EHS15] EPPSTEIN, DAVID, HAR-Peled, SARIEL, and SIDIPOULOS, ANASTASIOS. “Approximate Greedy Clustering and Distance Selection for Graph Metrics”. CorR abs/1507.01555 (2015). URL: http: // arxiv.org/abs/1507.01555.

[EIP11] EIKEL, BENJAMIN, JAHN, CLAUDIUS, and PETRING, RAFAL. “PADrend: Platform for Algorithm Development and Rendering”. Augmented & Virtual Reality in the Produktentstehung. Vol. 295. HNI-Verlagsschreinerei. Heinz Nixdorf Institut, Universität Paderborn, May 2011, 159–170. URL: https: //www.padrend.de.7.

[ELPZ97] ELDAJ, Y., LINDENBAUM, M., PORAT, M., and ZEVI, Y. Y. “The farthest point strategy for progressive image sampling”. IEEE Trans. Image Process. 6.9 (1997), 1305–1315. ISSN: 1057-7149. DOI: 10.1109/83.623193 3.

[EMP*12] EBIEDA, MOHAMMAD S., MITCHELL, SCOTT A., PATNEY, ANJUL, et al. “A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions”. Comput. Graph. Forum 31.2 (2012), 785–794. DOI: 10.1111/j.1467-8659.2012.03059.x 2.

[GV] GONZALEZ, TEOFILIO F. “Clustering to minimize the maximum intercluster distance”. Theoretical Computer Science 38 (1985), 293 – 306. DOI: 10.1016/0304-3975(85)90224-5 3.

[GYI15] GUO, JIANWEI, YAN, DONG-MING, JIA, XIAOHONG, and ZHANG, XIAOPENG. “Efficient maximal Poisson-disk sampling and remeshing on surfaces”. Computers & Graphics (46) (2015), 72–79. DOI: 10.1016/j.cag.2014.09.019 2.

[HB17] HULL, NICHOLAS and BENTY, N. NVIDIA Emerald Square, Open Research Content Archive (ORCA). 2017. URL: http: //developer. nvidia.com/orca / nvidia-emerald-square 10, 11.

[HCK*00] HOFF III, KENNETH E., CULVER, TIM, KEYSER, JOHN, et al. “Fast computation of generalized Voronoi diagrams using graphics hardware”. Proceedings of the Sixteenth Annual Symposium on Computational Geometry. Clear Water Bay, Hong Kong, China: ACM, June 2000, 375–376. DOI: 10.1145/336154.336226 5.
