Parallel Poisson Disk Sampling

Li-Yi Wei

Microsoft Research Asia

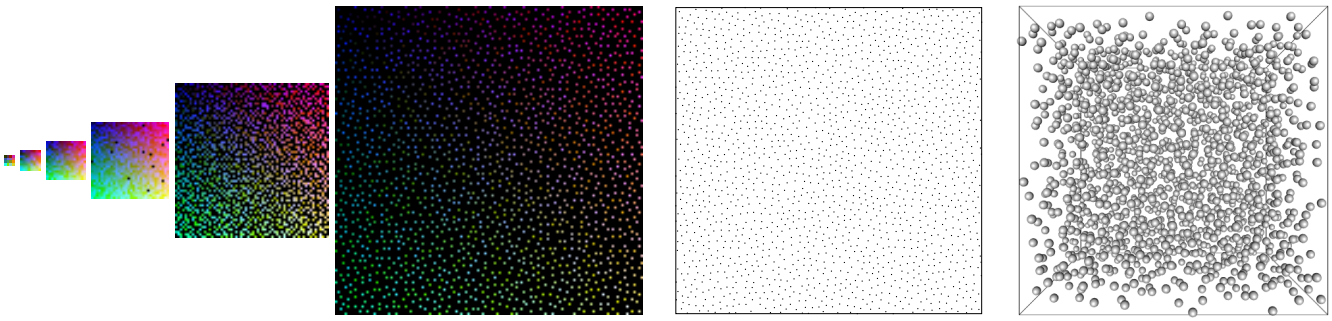


Figure 1: Poisson disk samples produced by our algorithm. The color images on the left are screen shots of our algorithm running on a GPU, producing 2D samples in a parallel and multi-resolution process. (Pixel colors represent sample locations with black color indicates absence of any sample.) The final set of 2D samples is shown in the middle. Our algorithm also works for arbitrary dimensions as demonstrated in the 3D case shown on the right (with samples visualized as small spheres). Our GPU implementation generates more than 4 million 2D samples/second and 555 thousand 3D samples/second. (2D case): r = 0.02, # samples = 1657. (3D case): r = 0.07, # samples = 1797. k = 4 for all images shown in this paper unless stated otherwise.

Abstract

Sampling is important for a variety of graphics applications include rendering, imaging, and geometry processing. However, producing sample sets with desired efficiency and blue noise statistics has been a major challenge, as existing methods are either sequential with limited speed, or are parallel but only through pre-computed datasets and thus fall short in producing samples with blue noise statistics. We present a Poisson disk sampling algorithm that runs in parallel and produces all samples on the fly with desired blue noise properties. Our main idea is to subdivide the sample domain into grid cells and we draw samples concurrently from multiple cells that are sufficiently far apart so that their samples cannot conflict one another. We present a parallel implementation of our algorithm running on a GPU with constant cost per sample and constant number of computation passes for a target number of samples. Our algorithm also works in arbitrary dimension, and allows adaptive sampling from a user-specified importance field. Furthermore, our algorithm is simple and easy to implement, and runs faster than existing techniques.

Keywords: Poisson disk, blue noise, sampling, parallel computation, GPU techniques, texture synthesis

1 Introduction

Sampling remains a core process in computer graphics, with a variety of applications ranging from rendering, imaging, and geometry processing. The number and distribution of the samples employed determine the speed of computation and quality of results. Cook [Cook 1986] observed that a Poisson disk sampling distribution usually yields superior image quality than alternative distributions with similar numbers of samples. Such a Poisson disk distribution has samples that are randomly located but remain at least a minimum distance r apart from one another (Figure 1). The Fourier spectrum of a set of Poisson disk samples exhibits *blue noise* property with low anisotropy and small amount of low frequency energy (Figure 4). In essence, a blue noise sampling produces visually pleasing results by replacing low frequency aliasing with high frequency noise, a less visually annoying effect. See [Lagae and Dutré 2008] for a comprehensive survey on recent techniques.

The desired properties of a sampling algorithm include (1) blue noise spectrum and (2) fast computation. Some earlier approaches that exhibit blue noise spectrum are often too slow for applications requiring a large number of samples (e.g. dart throwing [Cook 1986; McCool and Fiume 1992]). The efficiency has been improved by a variety of techniques either via pre-computed datasets [Cohen et al. 2003; Ostromoukhov et al. 2004; Lagae and Dutré 2005; Kopf et al. 2006; Ostromoukhov 2007] or by computing samples on the fly [Mitchell 1987; Jones 2006; Dunbar and Humphreys 2006; Bridson 2007; White et al. 2007]. However, despite their run-time efficiency, the pre-computed-dataset approaches may consume significant memory for storing data and could fall short in producing desired blue noise spectrums; see [Lagae and Dutré 2008] and Figure 6 for details. Even though some of the computation-on-the-fly approaches can produce blue noise power spectrums, to our knowledge they are sequential in nature. This imposes an upper limit on the achievable computation speed and prevents these algorithms from taking advantage of recent advances in parallel computing architectures such as GPUs and multi-core CPUs.

In this paper, we present a parallel algorithm that generates all samples on the fly with blue noise spectrums very similar to the ground truth produced by dart throwing [Cook 1986]. Our main idea is to subdivide the sample domain into square-shaped grid cells and we draw samples concurrently from multiple cells that are sufficiently far apart so that their samples cannot conflict with one another (i.e. to be within the specified minimum distance). Although this basic

idea is simple, care has to been taken in sampling and traversing the grid cells to ensure that the process does not introduce biases. We achieve this by a multi-resolution process inspired by [Popat and Picard 1997; Wei and Levoy 2000]. We present a parallel implementation of our algorithm running on a GPU (inspired by [Lefebvre and Hoppe 2005]) with constant cost per sample and constant number of computation passes for a target number of samples. To our knowledge, this is one of the first algorithms that allow a parallel generation of Poisson disk samples entirely on the fly.

As an added benefit, our algorithm also applies to dimensions higher than 2D. Although 2D sampling has a variety of important applications, sampling in higher dimensional space is required for other applications; examples include depth of field, motion blur [Akenine-Möller et al. 2007], and global illumination [Lawrence et al. 2005].[1] For these applications, using multiple slices of 2D Poisson samples is usually insufficient. One possible solution is to extend the pre-computed-dataset approaches in 2D to high dimensions (e.g. Wang cubes [Lagae and Dutré 2005]), but such approaches often incur combinatorial explosion in the amount of pre-computed datasets. A more feasible solution is to compute all samples on the fly just as the recent algorithm presented by [Bridson 2007], which works in arbitrary dimensions. This is the main source of inspiration for the arbitrary dimensionality of our algorithm.

Beyond quality, parallelism and arbitrary dimensionality, our algorithm also allows adaptive sampling as demonstrated in [Ostromoukhov et al. 2004; Kopf et al. 2006; Ostromoukhov 2007]. It also runs faster than existing techniques of which we are aware; see Section 6 for more details.

Our algorithm is quite simple; those who are primarily interested in implementation can start with Table 1 and 3. In the rest of the paper, we describe our algorithm in gradual steps: sequential uniform sampling (Section 2), parallel uniform sampling (Section 3), and adaptive sampling (Section 4). Along the way we also present the experiments and design choices we have made to come up with our algorithm. Although these are not essential for implementation, we believe doing so will facilitate readers' understanding and provide insights into the inner working of our algorithm.

2 Sequential Uniform Sampling

Our basic idea is to draw samples uniformly from square-shaped grid cells and perform this operation concurrently and independently for all grid cells that are sufficiently far apart. However, grid-based sampling can easily introduce bias. In the following, we present our sequential algorithm and discuss how we rid it of these biases. We then build our parallel algorithm on top of this, as described in the next section.

Given an n-dimensional sampling domain Ω and a minimum distance r between samples, we first build a grid around the domain with grid cell size bound by $\frac{r}{\sqrt{n}}$ so that each grid cell contains at most one sample. (This idea is from [Bridson 2007]; the importance of at-most-one-sample-per-cell will be apparent when we discuss GPU implementation.) We then traverse the grid cells in a certain order (to be discussed below). For each grid cell, we generate up to k new samples randomly drawn within the cell. For the first new sample that is not within a distance r from any existing sample, it is accepted as legal and added to the cell, and we move on to the next cell. If all k samples are rejected, we leave the cell empty. Even though this is a very simple process, we have found it nontrivial to design a traversal order that produces samples with blue noise

[1] However, as shown in [Mitchell 1991], Poisson disk sampling might not be the best option for all high-dimensional applications.

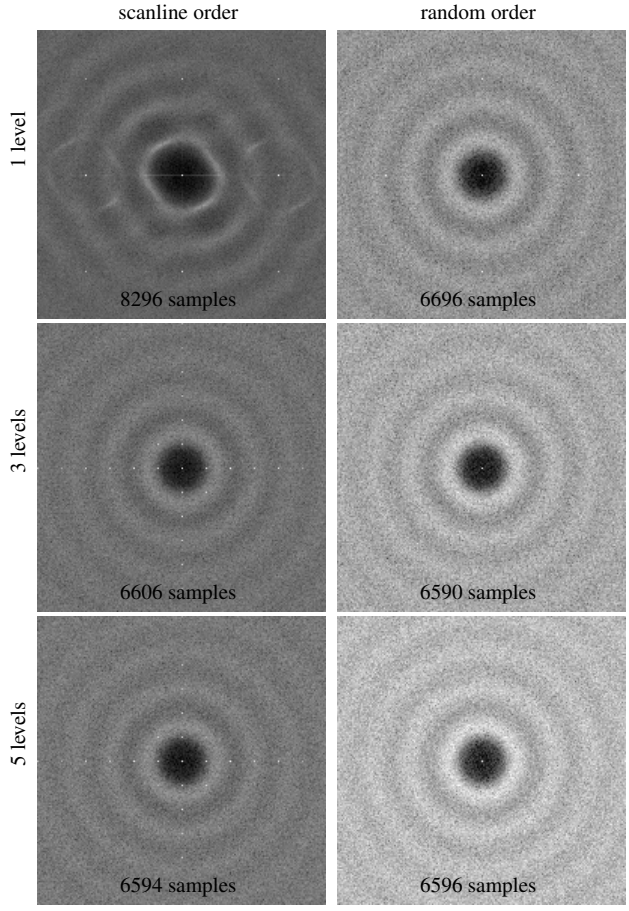


Figure 2: Comparison of different synthesis orders and number of levels. The biasing peaks on the Fourier spectrums are caused by (1) scanline order, (2) uniform sampling within grid cells, or both. Each image shows a power spectrum averaged over 10 runs with an identical order. $k = 10$, $r = 0.01$ and the average # of samples is indicated beneath each image. To make the peaks most obvious, we "tone-map" each individual spectrum by linearly scaling its values from [0 max] into [0 1] (where "max" is the per spectrum maximum value, not considering DC). Note that 1-level + scanline-order is able to pack more samples, causing twisted "rings" in the power spectrum.

spectrum. Below, we describe the traversal orders that we have attempted (as inspired by [Wei and Levoy 2000]), and conclude with a discussion.

Scanline order The first traversal order we tried is to visit the cells in a scanline order. It is not parallelizable, but we use it to illustrate potential sources of biases that can be caused by grid-based sampling. As shown in Figure 2 (top image on the left column), the result sample set contains obvious bias as manifested in the peaks and twisted "rings" of the corresponding Fourier spectrum. The peaks are located at frequencies equal to the number of grid cells per dimension. There are two possible reasons for such artifacts: (1) the grid cells are visited in a scanline order and (2) each sample is uniformly sampled from a grid cell. Below, we investigate these issues.

Random order + multi-resolution To attack issue (1), we simply randomize the traversal order (visiting each cell exactly once) with the result spectrum shown in Figure 2 (top image on the right column). Since now the scanline order bias is removed, the spectrum image does look better. However, there still exists some

bias in the spectrum image caused by issue (2) as manifested in the white spikes.

To attack issue (2), we use a multi-resolution approach so that the allowable sampling regions for samples start with the entire domain and gradually shrink to the cell size. The process is as follows:

Step 1 Initialize resolution level $L = 0$.

Step 2 Divide the sampling domain Ω into 2^{nL} sub-domains. Visit these sub-domains in a random order and produce up to k samples uniformly drawn within each sub-domain. For each sub-domain, when a sample is found to be at least r distance from all existing samples, insert it into the grid and move on to the next sub-domain.

Step 3 While sub-domain size $> \frac{r}{\sqrt{n}}$, increment L and go to Step 2. This moves the computation to a higher resolution.

Essentially, this algorithm attempts to produce one sample uniformly drawn from the entire domain, $(2^n - 1)$ samples from domains with smaller size ($1/2$ in each dimension), $(2^{2n} - 2^n)$ samples from domains with even smaller size ($1/4$ in each dimension), and so on. Note that the gradually-decreasing domain size is essential for the speed of convergence; if we were to keep the entire domain throughout all resolutions, our algorithm would reduce to dart throwing [Cook 1986]. Our approach also bears analogy to [McCool and Fiume 1992], but their method uses a gradually decreasing r whereas our approach uses a fixed r but changes the size of the random sampling sub-domains. Our random-order + multi-resolution algorithm eliminates the biasing artifacts (caused by issues 1 and 2 above); as demonstrated in Figure 2 (bottom two images on the right column) the corresponding Fourier spectrum exhibits blue noise properties.

To make sure both random-order and multi-resolution are needed, we have attempted to run the algorithm in scanline-order + multi-resolution. The results are shown in Figure 2 (bottom two images on the left column); notice the presence of bias even though it is reduced by multi-resolution.

Discussion Our approach bears strong resemblance to multi-resolution image processing and especially texture synthesis (e.g. [Popat and Picard 1997; Wei and Levoy 2000]), as we essentially use their framework and replace (1) their pixel color values with our sample locations and (2) their pixel neighborhood match with our sample conflict check. Even though traditional multi-resolution image processing theory cannot be directly applied to our method (pixel colors and sample locations are quite different beasts), it still provides us intuitive justifications as discussed below. Empirical evidence also confirms the quality of our approach; see spectrum results in Section 6.

We consider (1) regular traversal order and (2) grid-cell sampling the only two sources of biases as they are the only parts of our algorithm that exhibit any sampling regularity. Issue 1 is easier to grasp since a traversal order with directional bias will easily produce samples with spectrum alias (we use a scan-line order for extreme manifestation). A random order easily solves this. Issue 2 is more involved but our solution can be considered as a multi-resolution version of dart throwing [Cook 1986]. Specifically, in the original dart throwing algorithm [Cook 1986] each sample is drawn from the entire domain (maximum randomness per sample); this yields good sample statistics but is difficult to sample efficiently (and eventually to parallelize). Our single resolution algorithm, as another extreme, draws one sample from each grid cell; this allows easy sampling but yields grid-cell bias. Our multi-resolution algorithm strives for the right balance between quality and efficiency by drawing samples with gradually reduced randomness in a multi-resolution fashion.

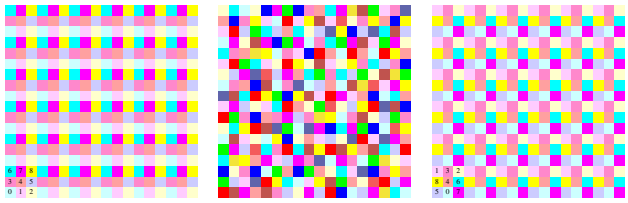


Figure 3: Phase group partition for parallel sample generation. Left: grid partition with scanline order. Middle: random partition. Right: grid partition with random order. Each phase group is illustrated with a unique color. For the two grid partition cases (left and right) the ordering is also marked at the lower left corner of the image. In this example, we use a rule that any two cells with the same id/color must be at least two cells apart.

In particular, samples generated at lower resolutions serve as constraints to reduce potential bias incurred by samples later generated at higher resolutions.

3 Parallel Uniform Sampling

Our sequential algorithm can be parallelized using the following key observation: for a group of grid cells sufficiently far away from one another, their corresponding samples cannot be closer than the minimum distance r. Consequently, samples drawn from these cells can be computed in parallel.

Our parallel algorithm works as follows. Similar to our sequential algorithm, the synthesis progresses in a multi-resolution fashion. Within each resolution, we partition the cells into disjoint phase groups so that cells within each group are at least r distance away from one another (and thus can be sampled in parallel). We produce all samples by visiting one phase group after another.

The key issue now is how we should perform such a phase group partition. There are several design decisions we have to make: (1) number of groups, (2) structure of each group, and (3) traversal order of these groups. Our goal is to achieve the best possible quality with fewest number of computation passes. Below we describe different options we have experimented with.

Grid partition + scanline order For n-dimensional space, it can be shown that the minimum number of phase groups is $\sim (\lceil \sqrt{n} \rceil + 1)^n$ and this can be achieved by a regular grid partition, as shown in Figure 3 (left). There are several possible orders to visit the phase groups. A naive method is a scanline order but this introduces similar bias artifacts to the scanline cell order in Figure 2.

Random partition A guaranteed method to eliminate the grid bias artifacts is to use a completely random phase partition as shown in Figure 3 (middle). This can be achieved by computing a unique group-id to sub-domains within each phase partition as follows:

Step 1 Produce a randomly ordered list of sub-domains belonging to a given resolution. Initialize the group-id for each sub-domain as 0. Initialize the active-list to contain all sub-domains. Initialize the current-id as 0.

Step 2 If the active list is not empty, visit its sub-domains in the pre-randomized order. If a sub-domain has group-id equals to current-id, removes it from the active list and update to (current-id+1) the group-ids of its neighboring sub-domains (within r distance) that are still on the active list.

Step 3 Increment current-id and go to Step 2.

After extensive experiments with different dimensions n and minimum distances r, we have found this heuristic produces number of

groups roughly twice the minimum achieved by the grid partition. (The ratio remains surprisingly constant around two with different n and r parameters.) So the increase in the number of computation passes is only about twofold. The heuristic also produces sufficiently random distribution of sub-domains within each group. By visiting the groups one by one (according to their group-id) while generating samples within each group in parallel, we are able to generate Poisson disk samples very efficiently. The result samples exhibit excellent blue noise spectrum similar to the ones shown in Figure 4.

However, one problem with this approach is that the computation of random phases remains a sequential process with computation time linearly proportional to the number of cells. This presents a potential bottleneck for parallel computation.

Grid partition + random order Our third option combines the good aspects of the two options presented above (speed and simplicity of grid partition + quality of random partition). It has a grid structure similar to our first option, but instead of a scanline order we visit the groups in a random order, as illustrated in Figure 3 (right). We have found a number of phase groups $\sim (2\lceil\sqrt{n}\rceil+1)^n$ (i.e. twice the minimum possible number of phase groups in each dimension) sufficient to produce samples with desired blue noise spectrum. Also, the random order of the phase groups can be pre-computed only once per dimension and stored as a small array, and consequently does not present a bottleneck to our parallel implementation. (We perform the pre-computation by generating a large number of random sequences and choose the one that produces an averaged power spectrum most similar to the ground truth produced by dart throwing.) We have chosen this option for our final implementation.

Discussion In our sequential algorithm we have shown that a random traversal for grid cells suppresses sampling alias. In our parallel algorithm we are attempting to tweak this traversal order to facilitate parallel grid cell sampling. Although our choice of grid partition + random order is not as random as a truly random permutation order, it is still sufficient to suppress grid traversal aliasing and we have not found any need for the sequence to satisfy rigorous statistical measurements. We have also tried to pre-compute random partition over a small hyper-cubical set of cells followed by tiling over the target domain; in theory this would provide better quality than grid partition + random order but empirically we have not found noticeable differences.

4 Adaptive Sampling

Our parallel uniform sampling algorithm can be extended for adaptive sampling as well. Unlike the uniform sampling case where each sample has to keep the same minimum distance r from one another, in adaptive sampling the user would supply a function $r(.)$ over the sampling domain Ω, specifying the minimum distance $r(s)$ for which sample $s \in \Omega$ has to be kept away from other samples.

Our parallel adaptive sampling algorithm is summarized in Table 1. Similar to our uniform sampling algorithm, our adaptive sampling algorithm utilizes an acceleration data structure around the domain Ω for generating samples. However, instead of a uniform grid, we use a hierarchical nd-tree structure for adaptive sampling. (*nd* means n-dimensional; our nd-tree is a high dimensional equivalence of quad-tree in 2D and oc-tree in 3D, for a lack of better term.) We build the nd-tree layer-by-layer on the fly after samples on the previous level are computed. The process is as follows. We begin with a single root node covering the entire domain Ω. For each node c on leaf level l of the tree, if it has no sample within, we try to generate one by uniform sampling from its domain $\Omega(c)$. This trial is

```
function ParallelAdaptiveSampling(Ω, r(.), k)
  // Ω: sampling domain in n-dimension
  // r(.): distance function defined over Ω
  // k: maximum number of trials per node
  T(0) ← BuildNDTreeRoot(Ω) // hypercube covering Ω
  l ← −1 // current level of T
  do // from coarse to fine, root to leaf of T
    l ← l + 1
    {p} ← ComputePhaseGroups(T(l))
    foreach phase group p in {p}
    // any two samples within two different nodes ∈ p
    // cannot conflict one another. See Section 3.
      parallel foreach node c in p
        if c has no sample
          s ← ThrowSample(T, Ω(c), r(.), k, l)
          if s is not null      add s to c      end
        end
      parallel end
    end
    T(l + 1) ← Subdivide(Ω, r(.), T(l))
  while T(l + 1) not ∅

function T(l + 1) ← Subdivide(Ω, r(.), T(l))
  parallel foreach node c of T(l)
    if ∃s ∈ c and √n μ(c) > r(s)
    // subdivide c only if likely to add another sample
    // μ(c) is the cell size of c
      subdivide c into 2^n child nodes // n is the dimension of Ω
      migrate s into the child c′ where s ∈ Ω(c′)
    end
  parallel end
  T(l + 1) ← newly created nodes
  return T(l + 1)

function s ← ThrowSample(T, Ω(c), r(.), k, l)
  foreach trial = 1 to k
    s ← sample uniformly drawn from Ω(c)
    if ∀s′ ∈ T  |s − s′| ≥ max(r(s), r(s′))
    // this can be done by examining only s′ ∈ neighbor nodes
    // within hyper-sphere of radius 3√n μ(l′) at level l′ = 0 to l
      return s
  end
  return null
```

Table 1: Pseudo-code of our algorithm. We can also use $\text{mean}(r(s), r(s'))$ (corresponding to geometric disks) instead of $\max(r(s), r(s'))$ in ThrowSample(). In that case, use $5\sqrt{n}\mu(l')$ instead of $3\sqrt{n}\mu(l')$ in ThrowSample() and $r(s)/2$ instead of $r(s)$ in Subdivide(). For clarify of presentation, we describe only $\max(r(s), r(s'))$ in the main text. See Appendix A for math details.

repeated for at most k times and we stop for the first sample that is not in conflict with any existing samples. It can be shown that this conflict check can be conducted by examining, at level l', for $l' = 0$ to l, existing samples at neighboring nodes $\{c'(l')\}$ whose centers are within $3\sqrt{n}\mu(l')$ ($\mu(l')$ indicates cell size at level l') distance from the center of the ancestor node $c(l')$ containing c . See Claim A.2 in Appendix A for proof.

After samples are deposited for level l, we perform subdivision at a subset of the nodes at that level. For each node c in level l, we subdivide it into 2^n uniformly-sized sub-nodes if c has a sample s within it (there can be at most one) and it is possible to add more samples within $\Omega(c)$ (this is true when $\sqrt{n}\mu(c) > r(s)$). If c is subdivided, we migrate its sample s to the child c' whose domain $\Omega(c')$ contains s. Consequently, interior nodes in the nd-tree possess no samples and each leaf node can possess at most one sample (similar to the uniform grid used in our uniform sampling algorithm). Note that our uniform sampling algorithm can be considered as a special

case of this adaptive algorithm with a complete nd-tree.

Our adaptive sampling algorithm can also be parallelized similar to our uniform sampling algorithm. For all nodes within the same level of the nd-tree, we perform a phase partition just like our parallel uniform sampling algorithm, and sample nodes within the same phase group concurrently. Similar to our uniform sampling algorithm, we use a grid partition + random order with twice the minimally possible number of phase groups at each dimension (i.e. $(2 * \lceil 3\sqrt{n} \rceil)^n$ since any two nodes $\lceil 3\sqrt{n} \rceil$ cells apart cannot have conflicting samples).

5 Implementation

5.1 Conflict check

For efficiency, care has to be taken when performing conflict check for every newly generated trial sample s. According to our adaptive sampling algorithm, for each new s uniformly randomly sampled from a node $c(l)$ at tree level l, we have to examine existing samples within each leaf node $c'(l')$ (with $0 \leq l' \leq l$) whose center is at most $3\sqrt{n}\mu(l')$ away from $c(l')$, the ancestor node at level l' whose domain contains s. One naive implementation is to examine a hyper-cube with size $6\sqrt{n}\mu(l')$ around $c(l')$, but this would be too computationally expensive as the cost is exponential in terms of the dimensionality n. In our implementation, we instead examine only a hyper-sphere with radius $3\sqrt{n}\mu(l')$ as the volumes of hyper-spheres grow much slower than hyper-cubes. Despite this, the potential number of neighbors that have to be checked can still get quite large; see Table 2.

n	hypercube	hypersphere	measured	
			best	worst
2D	81	61	3.92	4.38
3D	1331	619	8.15	9.66
4D	28561	6577	20.12	24.87
5D	371293	72797	58.44	69.02
6D	11390625	829201	177.57	201.30

Table 2: Neighborhood size for conflict checking. The table shows the number of neighborhood nodes that have to be checked for potential conflicts for different dimensions n and different neighborhood shapes. As shown, using a naive hyper-cube shaped neighborhood would incur much more computation than a hyper-sphere, especially at higher dimensions. The right-most two columns show the average number of neighbors visited at run-time for best and worst case scenarios. (Let r_t be a threshold r value for triggering cell subdivision as described in Table 1. The best case scenario is achieved with a r_b slightly larger than r_t, and the worst case with a r_w slightly smaller than r_t.) Notice that these numbers are much smaller than their theoretical counterparts shown on the same row.

Despite this potential large number of neighbors for conflict checking, we have found a useful trick to reduce the number of neighbors actually checked. For each conflict checking operation around a new sample, we visit its neighboring nodes in an inside-to-outside fashion. Since nearby nodes are more likely to contain conflicting samples, this would allow us to terminate the process when a nearby conflict is found. As show in Table 2 (column "measured"), at run time the real number of neighbors checked is much smaller than the worst case scenario across different dimensions.

5.2 GPU implementation

It is quite straightforward to implement our algorithm on a GPU as summarized in Table 3. Two major practical issues are (1) memory storage and (2) random number generation. For storage, we use framebuffer object (FBO) for generated sample locations as they are read-write data (i.e. output render target in one pass and input texture the next). We produce the samples from lower to higher nd-tree resolutions, and begin each resolution with initialization from lower resolution results. Within each resolution, we use two framebuffer objects to ping-pong the sample locations across different k trials and phase groups (i.e. one FBO serves as texture and another as render target, with their roles swapped in the next rendering pass). Since GPUs are designed mainly for 2D textures and render targets, currently we implement higher dimensional data structures by packing multiple 2D slices into a single 2D render target. For example, a 3D map is represented as a 1D stack of 2D regions, and a 4D map as a 2D grid of 2D regions.

```
function ParallelAdaptiveSamplingGPU
  foreach level l from coarse to fine
    construct framebuffer objects map(l, 0) and map(l, 1)
    σ ← 0 // source selection - ping pong between 2 maps
    // with map(l, σ) the input texture and
    // map(l, 1 − σ) the output render target
    if l > 0
      initialize map(l, σ) from map(l − 1, σ_{l−1})
      // initialization done similar to Subdivide() in Table 1
      // mask out nodes not subdivided from parents
    foreach trial from 0 to k − 1
      foreach phase group p
        map(l, 1 − σ) ← map(l, σ) // initialize render target
        foreach pixel s ∈ map(l, 1 − σ)
          // do following in the pixel shader
          if s ∉ p or s not empty or s masked out
            discard // fragment kill
          s ← random sample
          // similar to ThrowSample() in Table 1
          if s conflict any neighbor
            discard
          output pixel s
        end
        σ ← 1 − σ // next rendering pass
      end
    end
  end
```

Table 3: Pseudo-code for the GPU implementation of our algorithm. The final sample locations are available at the last map(l, σ_l).

As a further optimization, we layout pixels belonging to the same phase group spatially near each other (similar to [Lefebvre and Hoppe 2005]). This would incur some extra pixel location map arithmetic but result in faster total run time due to more coherent memory IO.

For random number generator, since current GPUs do not provide such routines we have to implement our own. In our current implementation we either use the hash-based method as presented in [Tzeng and Wei 2008] for GPUs supporting integer arithmetic (e.g. NVIDIA G80), or simply pre-compute the uniform random numbers and store the results in (up to k) textures for older generation GPUs.

Note that unlike in the CPU case where we perform up to k trials per grid cell before moving to the next one, here we perform only one trial per cell within a phase group before moving on to the next group, and repeat this process up to k times. We do this for two reasons. Quality-wise, this reduces a potential bias that favors more samples for earlier phase groups. Performance-wise, this would allow us to read one random number texture at one time, resulting in much better texture cache performance.

As discussed in Section 5.1, it is better to visit neighbor nodes in an inside-to-outside fashion during conflict check. We do so for our GPU implementation as well, and we discard the fragment immediately once we find its sample in conflict with existing ones. This

not only saves us from extra texture reads/writes but also permits us finishing a computation-pass earlier.

For adaptive sampling, instead of implementing a real nd-tree, we use a complete texture and mask out non-existing nd-tree nodes (via a special value) during the initialization step. Although it is possible to implement a real nd-tree on a GPU, we opt for this approach for simplicity and efficiency.

5.3 Order independence

In addition to being parallel, our algorithm is also order-independent, by which we mean that we can compute a subset of the samples while guaranteeing the results remain invariant as if all the samples were generated. This order-independence could be useful for situations where the entire sampling domain is huge but at a single instance of time or frame we only need a subset of the samples. This can be achieved by borrowing ideas from order-independent texture synthesis [Wei and Levoy 2002; Lefebvre and Hoppe 2005] since, similar to neighborhood match in these methods, our algorithm also examines only a small set of spatial neighbors for conflict check a new sample: those generated at lower resolutions, or those at the same resolution but at an earlier pass (either a smaller k or the same k but in an earlier phase group). Thus, the dependency group of each sample is constant, and we can pre-determine the minimally necessary set of samples per resolution from a given request set. At run time, we run our algorithm from low to high resolutions as discussed above, but at each resolution and each pass we compute only the minimal set instead of all samples covering the entire domain. The key to our implementation would be a random number generator that guarantees order-independence; this can be achieved via the hashing method in [Tzeng and Wei 2008] (using k as the key and texture coordinates as the input).

6 Results

Usage and parameters Our algorithm is pretty easy to use; aside from the mandatory parameters n and $r(.)$, the only user-tunable parameter is k. In our experience, picking k in the range of $[4\ 10]$ works well in practice as this would capture the majority of the samples. See Table 4 for details. (We have observed that k only affects the size of the "inner ring" as a result of different number of samples generated, not the quality, of the power spectrums.)

$n \setminus k$	1	2	3	4	5	6	7	8	9	10
2D	65	78	83	87	89	91	92	93	94	94
3D	62	74	80	84	86	88	90	91	92	93
4D	62	74	79	83	85	87	89	90	91	92
5D	64	74	79	83	85	87	88	89	90	91
6D	65	74	79	83	85	87	88	90	90	91

Table 4: Which k value to use? Each table entry indicates the percentage of samples generated with a particular k with respect to the maximum possible number of samples generated with a very large k_{inf}. Notice the diminishing returns when k increases.

n	r	r_{max}	ρ	# samples	spectrum size
2D	0.020	0.027	0.74	1606.5	512^2
3D	0.046	0.066	0.70	6533.5	128^3
4D	0.100	0.147	0.68	6731.5	64^4
5D	0.158	0.230	0.69	7321.4	32^5
6D	0.215	0.306	0.70	9185.0	16^6

Table 5: Detailed statistics for cases shown in Figure 4 & 5.

Quality We measure quality of Poisson disk sampling algorithms by two criteria (as detailed in [Lagae and Dutré 2008]): (1) the power spectrum and the associated radial mean and anisotropy measurements and (2) the relative radius ρ. ($\rho = \frac{r}{r_{max}}$ as defined in [Lagae and Dutré 2008]. r_{max} is the maximum average inter-sample distance computed from the maximum packing of a given number of samples.) For criterion (1), our algorithm produces samples exhibiting blue-noise power spectrums with desired radial mean and low anisotropy. As demonstrated in Figure 4 & 5, our sample distribution is very similar to brute force dart throwing. (Note: each image in Figure 4 & 5 is produced by 10 runs with identical phase group ordering.) For criterion (2), our algorithm usually produces distributions with ρ in the range $[0.65\ 0.85]$ as recommended by [Lagae and Dutré 2008]. See Table 5.

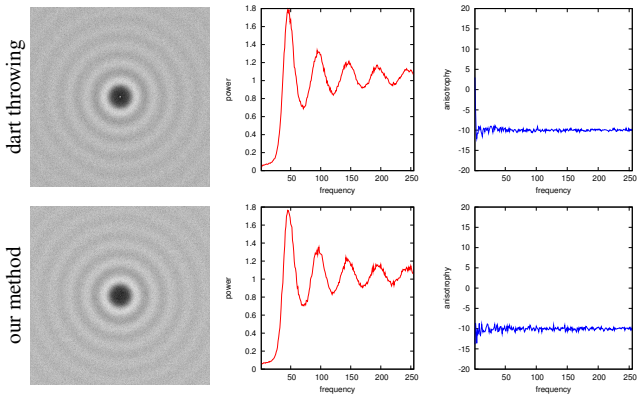


Figure 4: Spectrum comparison with dart throwing for 2D sampling. From left to right: power spectrum averaged over 10 runs, radial mean power, and radial variance/anisotropy. The radial mean is normalized against a reference white noise with mean power 1. An anisotropy of -10 dB is considered as the background noise [Lagae and Dutré 2008].

n	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
2D	4.06 M	3.56 M	3.19 M	2.85 M	2.57 M	2.37 M	2.20 M
3D	555 K	484 K	424 K	379 K	342 K	314 K	289 K
4D	42.9 K	38.3 K	34.7 K	31.7 K	29.3 K	27.1 K	25.3 K
5D	2.43 K	2.26 K	2.10 K	1.97 K	1.85 K	1.74 K	1.64 K
6D	179	170	162	155	147	142	136

Table 6: Performance of our algorithm on a GPU. All timings are measured on an NVIDIA Geforce 8800 GTX, and include the generation of samples at multiple resolutions and phases. Here we show the average number of samples per second (over 10 runs) for different combinations of n and k. (M = millions and K = thousands.)

Figure 6 compares the power spectrums with techniques that use pre-computed datasets [Kopf et al. 2006; Lagae and Dutré 2006; Ostromoukhov 2007]. As discussed in [Lagae and Dutré 2008], these approaches are prone to manifest the underlying tiling patterns in the power spectrum. This is most obvious in the anisotropy plots. We are able to reproduce such artifacts as shown in Figure 6. We have observed that such element-tiling artifacts are most obvious when (1) each element tile contains few samples and (2) a large output tiling is used. Approaches that compute all samples from scratch such as [Dunbar and Humphreys 2006; White et al. 2007] and our approach do not tend to have this problem.

Aside from the power spectrum issues, some pre-computed-dataset techniques such as [Ostromoukhov 2007] tend to produce samples with more uniform spatial distribution than our method. This can be desirable for some applications such as object placement.

Performance Table 6 lists the performance of our algorithm running on a commodity GPU. In the 2D case, our algorithm can

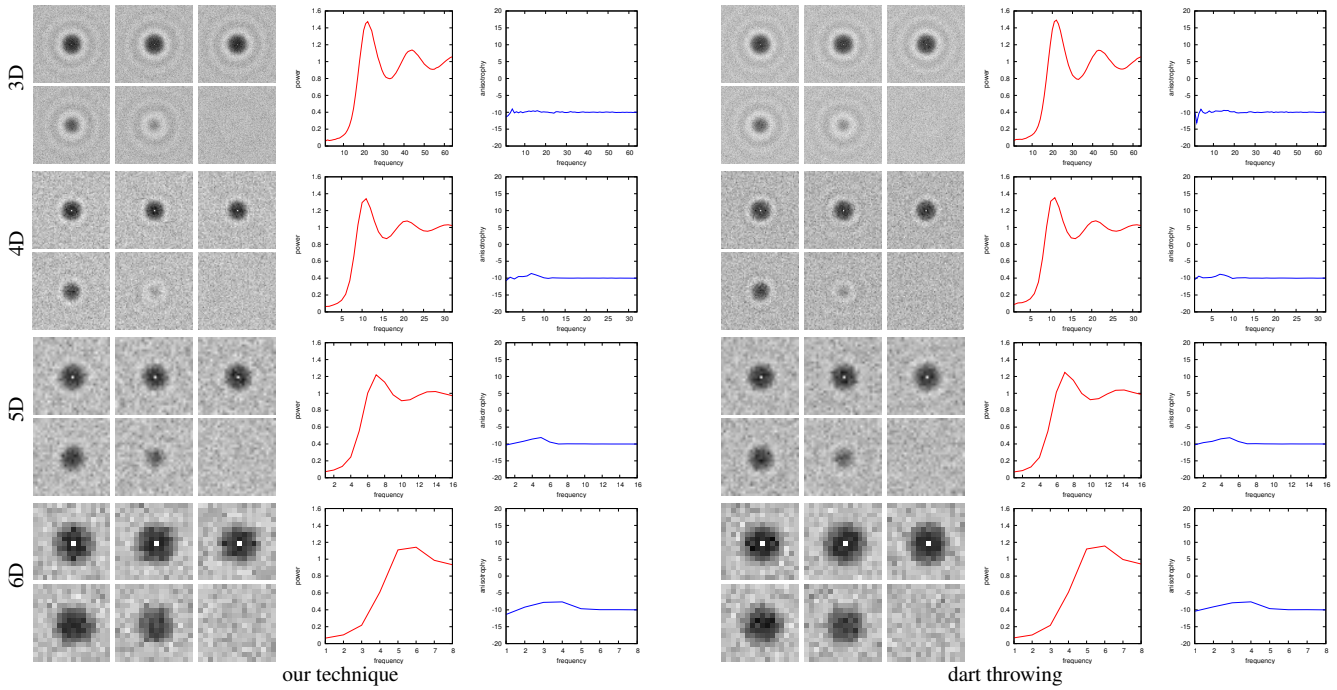


Figure 5: Spectrum results for sampling in different dimensions. Within each set of images generated by either dart throwing or our technique, the group on the left shows different 2D slices of the n-dimensional Fourier spectrum (with the top-row slices passing through the DC center and the bottom-row slices off the DC center with increasing offsets from left to right), and the group on the right shows the plots for radial mean and variance/anisotropy.

generate more than 4 million samples per second; this computation speed compares favorably with hierarchical dart throwing [White et al. 2007] (211 thousand samples per second) and boundary sampling [Dunbar and Humphreys 2006] (the fastest variation of all algorithms presented in that paper with about 200 thousand samples per second), two state-of-art techniques that also compute all samples from scratch as in our approach. We believe the performance gap between parallel and sequential algorithms will further widen on future GPUs and multi-core CPUs. The performance of our approach is in the same ballpark of techniques using pre-computed 2D datasets, such as recursive Wang tiles [Kopf et al. 2006] (between 1 million to 3 million samples per second) and Polyominoes [Ostromoukhov 2007] (more than 1 million samples per second), but our technique does not need to store any pre-computed dataset (which could consume significant amount of GPU memory, e.g. [Ostromoukhov 2007]). Due to the multi-pass rendering nature of our algorithm, it has lower samples-per-second ratio at lower resolutions and the optimal performance is achieved with sufficiently large number of samples. To reap the best performance of our algorithm, it is usually beneficial to use a CPU to compute the few samples at lower resolutions and then switch to a GPU for higher resolutions. For example, in the 2D case we compute the first 4 levels (up to texture size 8×8) on a CPU.

In higher dimensional cases, the performance of our algorithm is less than the performance of 2D case as shown in Table 6. This is caused by increased theoretical computational complexity (larger number of neighborhood nodes for conflict check, more computation phases, and longer coordinate vector length) as well as reduced texture cache coherence and other GPU-specific performance issues. The performance for 5D and 6D is further degraded by the fact that we have to use multiple render targets to store samples with dimensionality greater than 4.

Adaptive sampling Our algorithm is also applicable for adaptive sampling, as demonstrated in Figure 7 & 10. Given a user-specified importance field function $I(.)$, we compute a distance field $r(.) \propto I(.)^{\frac{-1}{n}}$ and use that to drive our adaptive sampling process.

7 Limitations and Future Work

We have presented an algorithm for parallel Poisson disk sample generation in arbitrary dimensions. We have empirically shown that this strategy works well, but we do not yet have any rigorous proof. We believe more theoretical insights can be gained by extending multi-resolution signal processing theory to incorporate location/domain instead of color/range information, and this may lead to further quality and/or speed improvements.

We also plan to improve the (quite slow) performance of our algorithm in 4D and higher dimensional cases. Currently, we do not know how much computation is necessary due to the curse of dimensionality, e.g. since the number of neighboring samples increases exponentially with respect to dimensionality, it might be difficult to reduce the number of neighborhood nodes that need to be checked for conflict.

Our algorithm is fastest for uniform sampling. The amount of slowdown for adaptive sampling will depend on how un-uniform the importance field is. Although we haven't tried this, we conjecture the worst case scenario would be an importance field that is very spiky (e.g. a narrow Gaussian), forcing the algorithm to do very little work at each resolution. Our current method also lacks the capability for fine-grain sample ranking [Kopf et al. 2006; Ostromoukhov 2007], a feature useful for applications such as continuous zoom-in.

For adaptive sampling a high-varying importance field, our method might prematurely terminate the subdivision of a node due to the rejection of early trial samples. This can be addressed by using a larger k value.

The sample sets generated by this technique are not guaranteed to

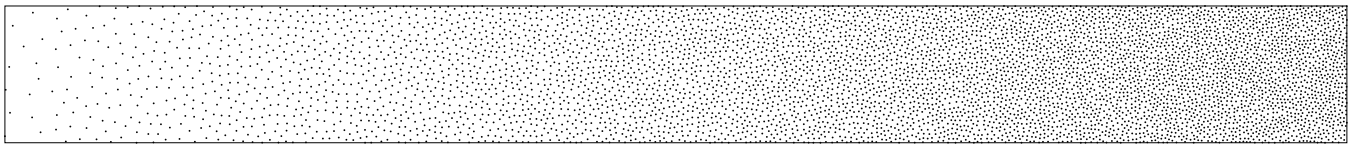


Figure 7: Adaptive sampling result. The samples are distributed with local density proportional to the input importance function, a linear ramp in this example. $r = 0.01$, # samples = 5973.

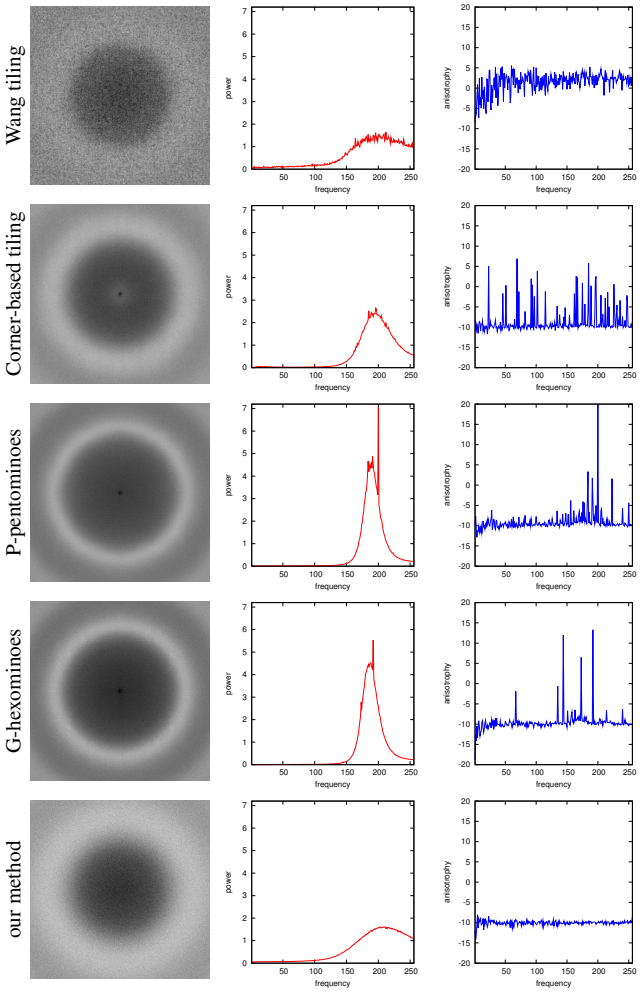


Figure 6: Spectrum comparison with techniques that use pre-computed datasets. From left to right: power spectrum averaged over 10 runs, radial mean power, and radial variance/anisotropy. From top to bottom: Wang tiling [Kopf et al. 2006], Corner-based tiling [Lagae and Dutré 2006], P-pentominoes [Ostromoukhov 2007], G-hexominoes [Ostromoukhov 2007], and our technique. The Wang tiling case consists of 32768 samples, generated by a 4×4 tiling with 2048 samples per tile. The corner-based tiling case consists of 33856 samples, generated by a 23×23 tiling with 64 samples per tile. The P-pentomino case consists of 32000 samples, generated by a 10×10 tiling of 20×20 patches with 2 levels of subdivision. The G-hexomino case consists of 31104 samples, generated via the deterministic tiling process as described in [Ostromoukhov 2007]. Our method ($r = 0.0044$, # samples $\sim$ 32K) produces better results as demonstrated both by the power spectrum image and the anisotropy plot. Note that under similar number of samples, results produced by [Ostromoukhov 2007] have larger inner ring than our technique due to the use of Lloyd relaxation. This translates to a more uniform spatial layout of samples; see Figure 8.

be maximal, and it can be difficult to control the exact number of samples generated. Also, the maximum number of samples that can be produced in a single run in our current implementation is limited by maximum texture size (about 1 million 2D samples per run via a 2Kx2K texture on the specific GPU we used). Producing more samples would require multiple runs but the order-independence nature of the algorithm would make the process consistent.

Our algorithm currently handles only Euclidean spaces. An interesting future work is to extend it for non-Euclidean spaces such as spheres [Ostromoukhov 2007] or general polygonal surfaces [Turk 1992]. We believe the ability to efficiently generate Poisson disk samples in parallel for such non-Euclidean surfaces would benefit a variety of sampling and geometry processing applications.

Acknowledgements Ares Lagae sent me a copy of his Ph.D. dissertation which inspired and initiated my pursue of this project, and provided data for his corner-based tiling method. Johannes Kopf provided his data and source code online. Victor Ostromoukhov answered questions regarding his Polyomino-based technique. Eric Andres answered questions regarding hyper-sphere lattice ordering. Zhouchen Lin and Ting Zhang provided great help on math details. Kun Zhou, Xin Tong, and Jian Sun challenged my project presentation. Stanley Tzeng helped with video dubbing. Eric Stollnitz, Dwight Daniels, and Jianwei Han commented on the writing. Baining Guo and Harry Shum supported my foray into MSRA's first single-authored SIGGRAPH paper. Reviewers made many valuable suggestions.

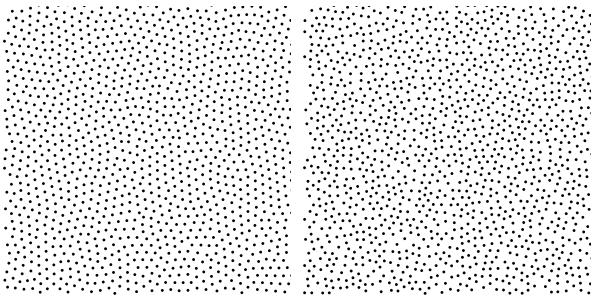


Figure 8: Spatial sample layout comparison. Here are zoom-in views of samples generated for Figure 6. Note that G-hexominoes [Ostromoukhov 2007] produces a more uniform spatial distribution than our method.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic rasterization using time-continuous triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, 7–16.

BRIDSON, R. 2007. Fast poisson disk sampling in arbitrary dimensions. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Sketches & Applications*.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 287–294.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Trans. Graph. 5*, 1, 51–72.

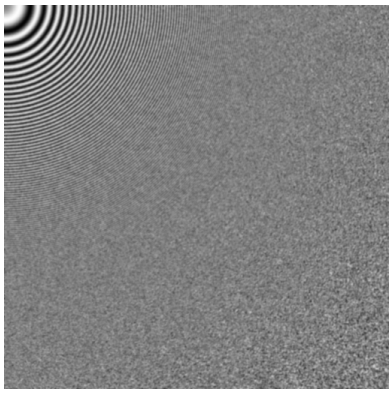


Figure 9: Anti-aliasing test via the zone plate pattern. The image is produced by sampling $\sin(x^2 + y^2)$ with a set of 266889 samples (roughly 1 sample per pixel) and filtering with a 3 pixel wide Gaussian kernel.

Figure 10: Adaptive sampling result. This set, consisting of 264346 samples, is produced from the importance image in [Kopf et al. 2006].

DUNBAR, D., AND HUMPHREYS, G. 2006. A spatial data structure for fast poisson-disk sample generation. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 503–508.

JONES, T. R. 2006. Efficient generation of poisson-disk sampling patterns. *journal of graphics tools 11*, 2, 27–36.

KOPF, J., COHEN-OR, D., DEUSSEN, O., AND LISCHINSKI, D. 2006. Recursive wang tiles for real-time blue noise. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 509–518.

LAGAE, A., AND DUTRÉ, P. 2005. A procedural object distribution function. *ACM Trans. Graph. 24*, 4, 1442–1461.

LAGAE, A., AND DUTRÉ, P. 2006. An alternative for wang tiles: colored edges versus colored corners. *ACM Trans. Graph. 25*, 4, 1442–1459.

LAGAE, A., AND DUTRÉ, P. 2008. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum*. to appear.

LAWRENCE, J., RUSINKIEWICZ, S., AND RAMAMOORTHI, R. 2005. Adaptive numerical cumulative distribution functions for efficient importance sampling. In *Eurographics Symposium on Rendering*.

LEFEBVRE, S., AND HOPPE, H. 2005. Parallel controllable texture synthesis. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, 777–786.

MCCOOL, M., AND FIUME, E. 1992. Hierarchical poisson disk sampling distributions. In *Proceedings of the conference on Graphics interface '92*, 94–105.

MITCHELL, D. P. 1987. Generating antialiased images at low sampling densities. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 65–72.

MITCHELL, D. P. 1991. Spectrally optimal sampling for distribution ray tracing. *SIGGRAPH Comput. Graph. 25*, 4, 157–164.

OSTROMOUKHOV, V., DONOHUE, C., AND JODOIN, P.-M. 2004. Fast hierarchical importance sampling with blue noise properties. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 488–495.

OSTROMOUKHOV, V. 2007. Sampling with polyominoes. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Papers*, 78.

POPAT, K., AND PICARD, R. W. 1997. Cluster based probability model and its application to image and texture processing. *IEEE Trans. Image Processing 6*, 2, 268–284.

TURK, G. 1992. Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, 55–64.

TZENG, S., AND WEI, L.-Y. 2008. Parallel white noise generation on a gpu via cryptographic hash. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, 79–87.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 479–488.

WEI, L.-Y., AND LEVOY, M. 2002. Order-independent texture synthesis. Tech. Rep. TR-2002-01, Computer Science Department, Stanford University.

WHITE, K., CLINE, D., AND EGBERT, P. 2007. Poisson disk point sets by hierarchical dart throwing. In *Symposium on Interactive Ray Tracing*, 129–132.

A Math Details

We first deal with conflict metric $\max(r(s), r(s'))$ in the following two claims. We then describe how to deal with $\mathrm{mean}(r(s), r(s'))$.

Claim A.1 *In our adaptive sampling algorithm, for a sample s located in a cell c with size μ in tree level l, we must have $r(s) \leq 2\sqrt{n}\mu$ (where n is the dimensionality of the sample space).*

Proof Sample s can either be migrated down from a parent cell of c or generated within c itself. In the former case, we know $r(s) \leq 2\sqrt{n}\mu$; otherwise the parent cell won't have been subdivided according to the cell subdivision criteria of our algorithm. In the latter case, since the parent cell has been subdivided, it must have a sample s' to begin with (again according to our cell subdivision criteria) and s' must be located within one of the sibling cells of c sharing the same parent. Consequently, we know $r(s) \leq 2\sqrt{n}\mu$, otherwise s and s' cannot co-exist. ∎

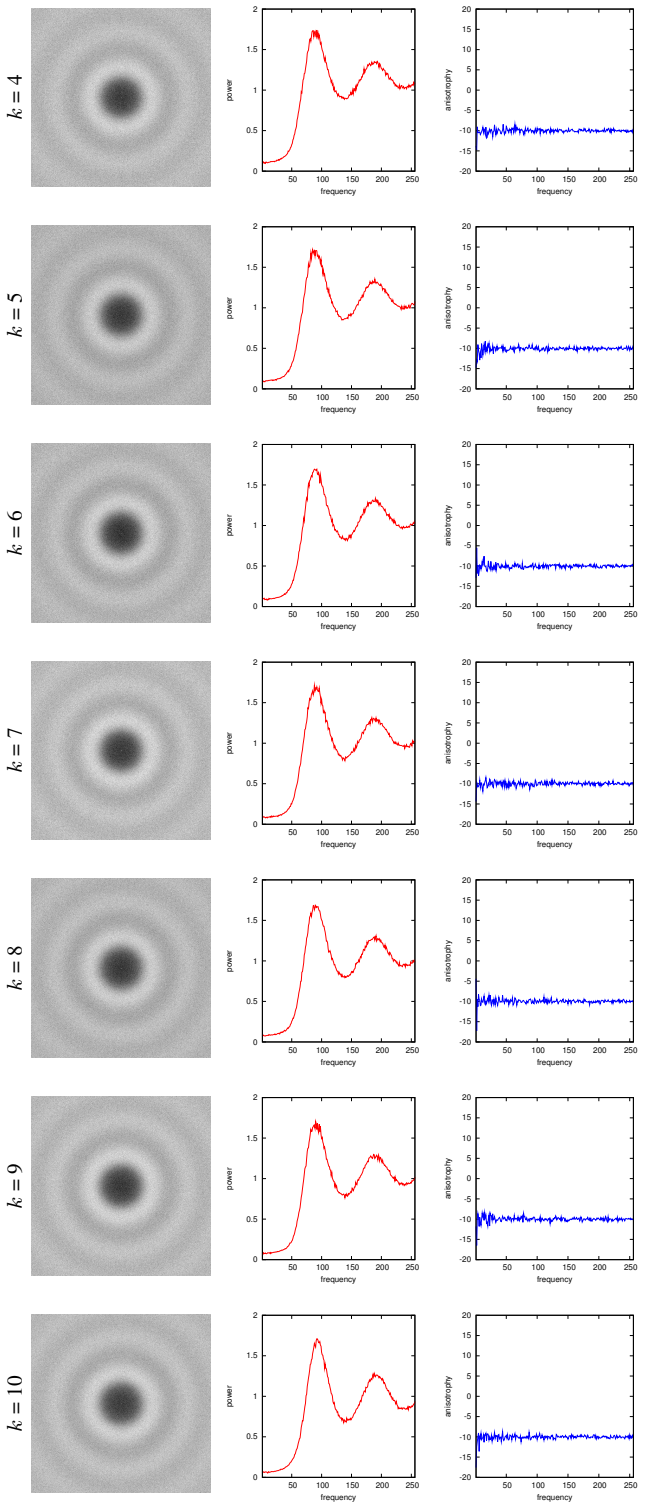
Claim A.2 *In our adaptive sampling algorithm, for each ancestor cell $c(l)$ at resolution l containing the current query sample s, we only need to look at neighboring cells whose centers are within $3\sqrt{n}\mu(l)$ (where $\mu(l)$ is the cell size at resolution l) distance from the center of $c(l)$.*

Proof From Claim A.1, we know that for each sample s' contained within a leaf node $c'(l)$ at level l, we have $r(s') \leq 2\sqrt{n}\mu(l)$. Since s can be at most $0.5\sqrt{n}\mu(l)$ away from the center of $c(l)$, we know s can conflict s' only if s' is within $2.5\sqrt{n}\mu(l)$ distance from the center of $c(l)$. Now, since s' itself can be at most $0.5\sqrt{n}\mu(l)$ from the center of $c'(l)$, we know the centers of $c(l)$ and $c'(l)$ can be at most $3\sqrt{n}\mu(l)$ apart.

Note that this works regardless of $r(s)$ as it must be $\leq 2\sqrt{n}\mu(l)$ for s to be accepted as a valid sample. ∎

For the conflict metric $\mathrm{mean}(r(s), r(s'))$, it can be shown via similar arguments that (1) $r(s) \leq 4\sqrt{n}\mu$ in Claim A.1 and (2) the conflict radius should be $5\sqrt{n}\mu(l)$ in Claim A.2.

Supplementary Materials



k	4	5	6	7	8	9	10
# samples	5595	5721	5833	5906	5983	6027	6070

Figure 11: Spectrum comparison with different k values. From left to right: power spectrum averaged over 10 runs, radial mean power, and radial variance/anisotropy. We have observed that k only affects the size of the "inner ring" (as a result of different number of samples), not the quality, of the power spectrums. ($r = 0.01$ for all cases.)

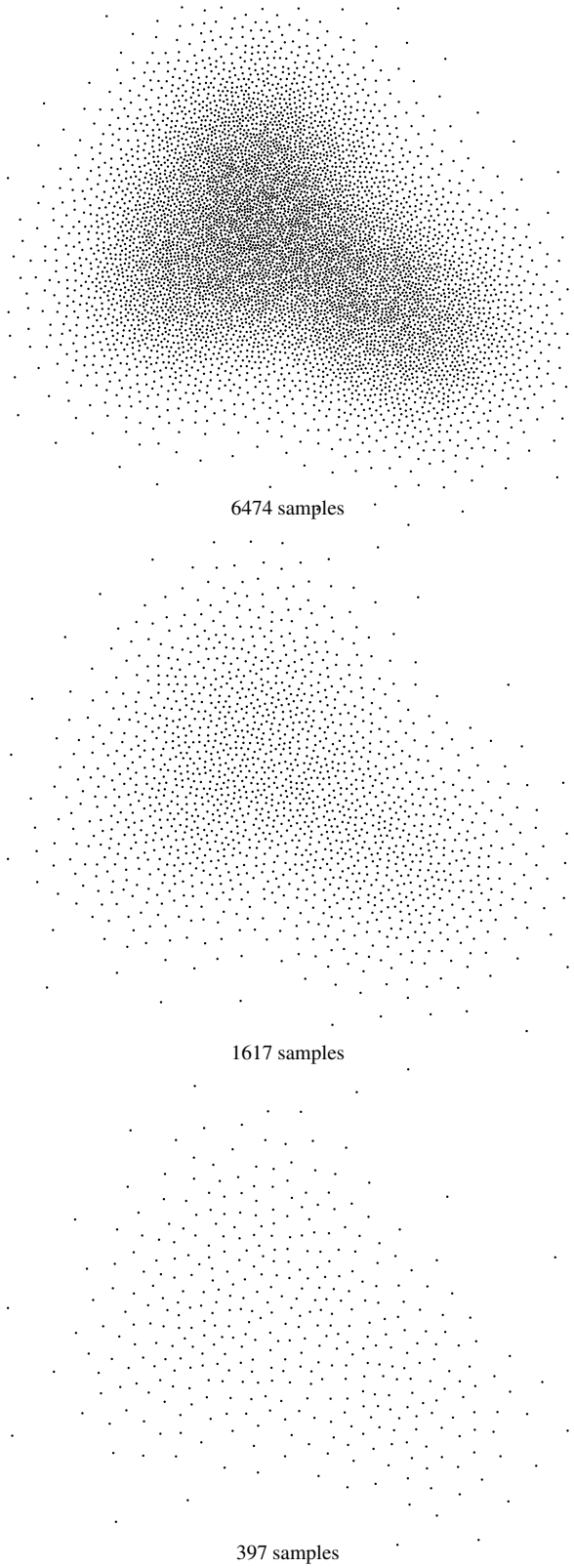


Figure 12: Adaptive sampling results. Here we show several sets with different number of samples. The source importance image is from [Ostromoukhov 2007].