Multi-Class Blue Noise Sampling

Li-Yi Wei

Microsoft Research

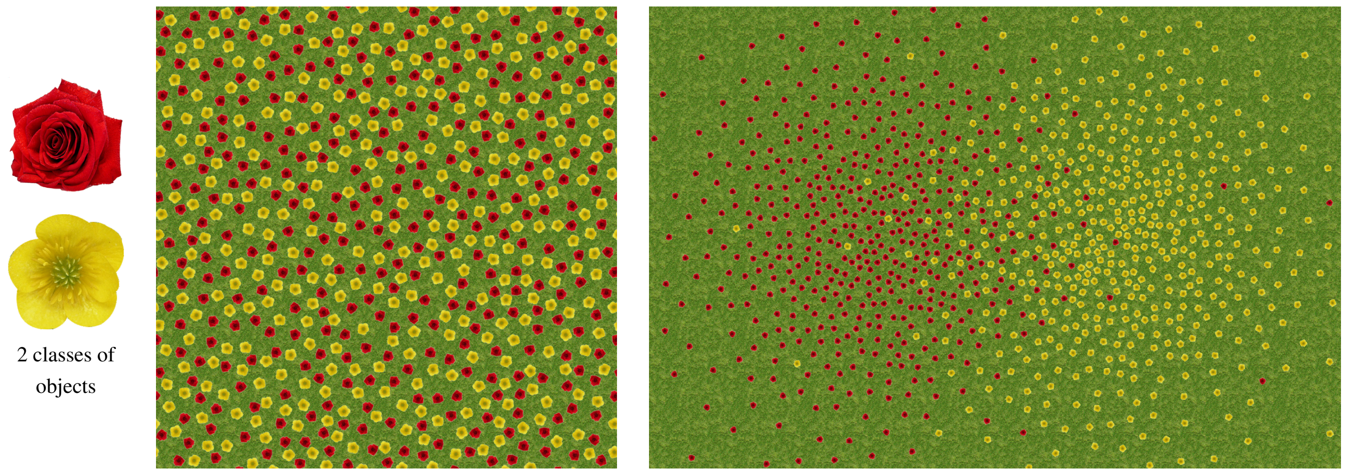


Figure 1: Object placement by multi-class blue noise sampling. Our algorithms can produce both uniform (middle) and adaptive (right) sampling results.

Abstract

Sampling is a core process for a variety of graphics applications. Among existing sampling methods, blue noise sampling remains popular thanks to its spatial uniformity and absence of aliasing artifacts. However, research so far has been mainly focused on blue noise sampling with a single class of samples. This could be insufficient for common natural as well as man-made phenomena requiring multiple classes of samples, such as object placement, imaging sensors, and stippling patterns.

We extend blue noise sampling to multiple classes where each individual class as well as their unions exhibit blue noise characteristics. We propose two flavors of algorithms to generate such multi-class blue noise samples, one extended from traditional Poisson *hard* disk sampling for explicit control of sample spacing, and another based on our *soft* disk sampling for explicit control of sample count. Our algorithms support uniform and adaptive sampling, and are applicable to both discrete and continuous sample space in arbitrary dimensions. We study characteristics of samples generated by our methods, and demonstrate applications in object placement, sensor layout, and color stippling.

Keywords: multi-class, blue noise, sampling, Poisson hard/soft disk, dart throwing, relaxation

1 Introduction

Sampling is important for a variety of graphics applications, including rendering, imaging, and geometry processing. Although different applications may favor different sampling patterns, blue noise sampling remains quite popular and is widely adopted. Inspired by the distribution of primate retina cells [Yellott 1983], a blue noise distribution contains samples that are randomly located but remain spatially uniform. The resulting sample set has a blue noise power spectrum, with a signature lack of low frequency energy and structural residual peaks. Beyond a blue noise power spectrum, additional desiderata for graphics applications include support for adaptive/importance sampling, efficient computation, as well as the ability to place samples in both discrete and continuous sample spaces.

Due to its importance, blue noise sampling has been researched extensively [Cook 1986; Mitchell 1987; McCool and Fiume 1992; Ostromoukhov et al. 2004; Jones 2006; Dunbar and Humphreys 2006; Kopf et al. 2006; Ostromoukhov 2007; Bridson 2007; White et al. 2007; Wei 2008; Fu and Zhou 2008; Balzer et al. 2009; Cline et al. 2009]. However, prior methods are mainly concerned about a single class of samples, and thus are not directly applicable to a variety of natural or man-made phenomena involving multiple classes of samples, such as the distribution of cone and rod cells in human retinas, the placement of multiple categories of objects, and the usage of multiple colored dots for stippling. In these situations, it is often desirable to have each individual class of samples as well as their union to exhibit blue noise distribution. Unfortunately, such simultaneous blue noise properties across multiple classes of samples are not guaranteed by previous single-class sampling methods. See Figure 2 for an example.

We present multi-class blue noise sampling algorithms that not only guarantee a blue noise spectrum for each individual class as well as their unions, but also allow samples to be placed in both discrete and continuous sample spaces. Our methods also support adaptive sampling and arbitrary sample space dimensionality.

To support different application scenarios, we present two flavors of algorithms: one derived from Poisson *hard* disk sampling [Cook 1986] for explicit control of sample spacing, and another based on our *soft* disk sampling for explicit control of sample count as in relaxation [Lloyd 1982]. (In a nutshell, a hard disk, centered on each sample, can neither deform nor intersect another, while a soft disk can intersect another, but subject to an energy penalty which, when minimized, produces uniform distribution.) Our main idea is to extend single-class dart throwing for multi-class soft/hard disk sampling by replacing the spacing parameter r in the former with a $c \times c$ symmetric matrix $\mathbf{r}$ for c sample classes. Since many

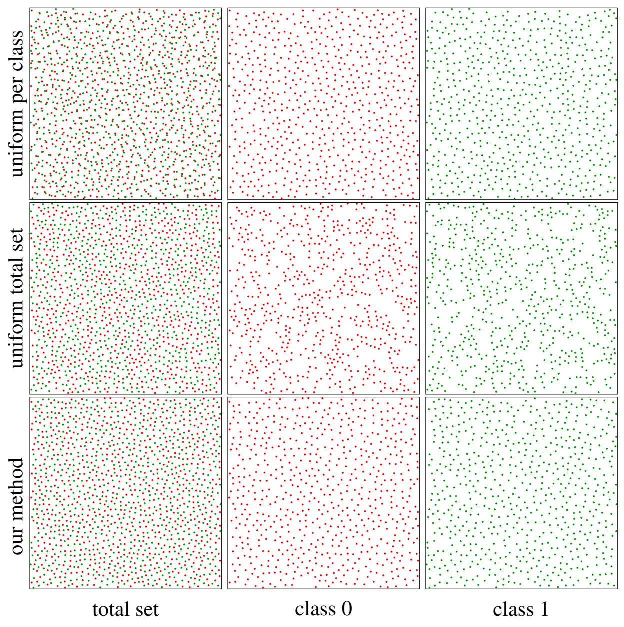


Figure 2: Comparison between single- and multi-class blue noise sampling. The top row is produced by applying single-class dart throwing to individual classes, but the total set is highly non-uniform. The middle row is produced by applying single-class dart throwing to the total set, but the individual classes are highly non-uniform. Our approach produces samples that exhibit blue noise distribution for each class as well as the total set. Each class contains $\sim$650 samples generated with r = 0.02.

blue-noise sampling methods are descendants of these two seminal algorithms [Lloyd 1982; Cook 1986] with different quality, performance, and usage tradeoffs, having multi-class extensions with both flavors could benefit different applications. Several such applications we show include object distribution [Cohen et al. 2003; Lagae and Dutré 2005], stippling [Kopf et al. 2006; Balzer et al. 2009], sensor layout and color filter design [Ben Ezra et al. 2007], involving both continuous/discrete sample space, uniform/adaptive sampling, control for sample spacing/count, and preferences for spatial-uniformity/spectrum-quality.

Our method is related to color halftoning and especially vector error diffusion [Baqai et al. 2005; Pang et al. 2008], which can also produce multiple classes of blue noise samples. However, our method differs from color halftoning in several significant ways. First of all, halftoning methods are mainly about computing colors for a given set of discrete samples, not for general purpose sampling in graphics that might require computing both color and position information in either a discrete or continuous domain. For halftoning methods that rely on limited neighborhood sizes such as dithering (predetermined masks) or error diffusion (predetermined distribution coefficients), such regular discretization could be undesirable (see e.g. [Alliez et al. 2003; Ostromoukhov et al. 2004]). Simply increasing the output resolution may not eliminate these discretization artifacts because this would reduce the effective spatial extent of the fixed neighborhoods. Thus, even in the traditional single class setting, halftoning is not a replacement for general blue noise sampling. Furthermore, even though certain halftoning techniques like error diffusion have implied blue noise properties, there is no guarantee that this will be carried over in the multi-class setting. To our knowledge, the best halftoning methods for generating multi-class blue noise samples rely on iterative optimization (e.g. the pioneering work of [Wang and Parker 1999]), which is often slow/complex and restricted to uniform/regular/discrete sampling.

2 Multi-Class Hard Disk Sampling

Dart throwing is a classical algorithm [Cook 1986] for Poisson *hard* disk sampling, a particular kind of blue noise distribution where samples are not only randomly and uniformly distributed but remain at least a minimum distance r away from each other. In dart throwing, a trial sample is drawn randomly from the entire domain. If the sample is not within a user-specified distance r from any other existing samples, it is accepted. Otherwise, it is rejected. This process is repeated until reaching certain termination criteria, e.g. a target number of samples and/or a maximum number of trials.

Our multi-class hard disk sampling algorithm follows a similar process, with necessary extensions to handle multiple classes of samples. Specifically, instead of a single number r, the user specifies a set of numbers $\{r_i\}_{i=0:c-1}$ for the c classes of samples. During the sample generation process, instead of checking whether a new trial sample is at least r away from all existing samples, we use a (symmetric) $c \times c$ matrix $\mathbf{r}$ for conflict check, where two samples in classes k and j have to be at least $\mathbf{r}(k, j)$ away from each other. This $\mathbf{r}$-matrix is built from $\{r_i\}$ with each diagonal entry $\mathbf{r}(i, i) = r_i$. Finally, we also need to determine the class for each new trial sample. Below we describe the algorithm in detail.

2.1 Sample class

For multi-class sampling, we have to decide from which class to sample for the next trial. To ensure that each class is well sampled throughout the entire process, we always pick the next trial sample from the class that is currently most under-filled. We measure this via *fill rate*, defined as the number of existing samples for a particular class over the target number of samples for that class. To maintain an equal fill rate across different classes, we define N_i, the target sample number of class i, as follows:

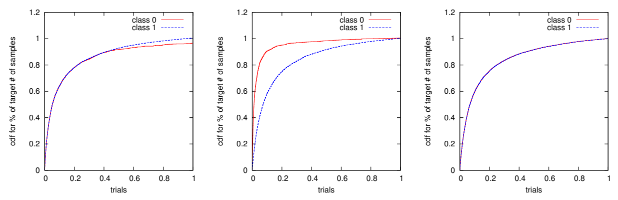
$$N_i = N \frac{\frac{1}{r_i^n}}{\sum_{j=0}^{c-1} \frac{1}{r_j^n}} \tag{1}$$

where N is the total number of target samples, n the sample space dimension, and $\{r_i\}$ the specified per-class minimum distances. Not maintaining equal fill rates among all classes can easily lead to non-uniform sample distribution.

2.2 Sample control

To ensure easy usage and a uniform sample distribution, it is desirable to generate the classes together (instead of one after another) and maintain a consistent fill rate among different classes throughout the sampling process. (We present a more detailed analysis in Section 4.) However, always drawing the next trial sample from the most under-filled class (Section 2.1) alone is not enough to achieve this goal, as it may be unable to find a new sample not in conflict with existing ones. This can happen quite early in the process when the output distribution is far from being uniform, so we cannot simply stop there. One possible remedy is to accept a trial sample s if it fails to be accepted for the most under-filled class but succeeds for another one. However, as shown in the left case of a simple 2-class experiment in Figure 3, even though the classes might maintain consistent fill-rates throughout the early part of the process, in the end the fill-rates may become unbalanced as eventually it becomes difficult for the class with a larger r value to compete with another with smaller r.

Another possibility is to tune the relative probability to sample from each class to achieve the desired fill rates at the end of the process, as shown in the middle case of Figure 3 (notice the two curves meet together at the end). However, there are two problems for this method. First, it is very tricky to come up with the right class proba-



247810.7 trials	240310.5 trials	157388.3 trials
0 killed	0 killed	1023.7 killed
237444.7 rejected	229944.5 rejected	145998.6 rejected
10366 accepted	10366 accepted	11389.7 accepted

Figure 3: Sampling history comparison. Here, we plot the fill-rates for two classes throughout the sampling process (with the number of trials normalized to [0 1]). $r_0 = 0.02$ and $r_1 = 0.00756$. The left case is generated by always drawing a new sample from the most under-filled class, the middle case with a constant class probability p_0 = 0.425 and p_1 = 0.575, and the right case by our final algorithm that allows removal of samples. We also measure # of trials as well as # of accepted, rejected, and killed samples (averaged over 10 runs). Notice that all 3 cases have the same final number of samples 10366 (= 11389.7 accepted - 1023.7 killed for the right case).

bilities; we estimated the numbers for the middle case in Figure 3 by exhaustively trying a vast number of possible combinations. Second, due to the stochastic nature of dart throwing, even with perfectly tuned probabilities, there is no guarantee that a different run with the same parameters would reach the same end condition, i.e. the two curves might still not meet in the end.

To resolve these issues, we allow the removal of existing samples $\mathbf{n}_s$ that are in conflict with a new trial sample s if (1) it is impossible to add another sample to class c_s (this can be figured out by tracking the still available spaces [Dunbar and Humphreys 2006] or by using a simple timeout mechanism), (2) each $s' \in \mathbf{n}_s$ (i.e. these in conflict with s) belongs to a class $c_{s'}$ with a smaller r than the class c_s for s and (3) each $c_{s'}$ is at least as filled as c_s. See Removable() in Program 1. Intuitively, this means that we only remove samples from classes that are easier to sample from (i.e. having a smaller r value) and are already as filled as the current class.

```
function bool Removable($\mathbf{n}_s, s, \mathbf{r}$)
  foreach $s' \in \mathbf{n}_s$
    if $\mathbf{r}(c_{s'}, c_{s'}) \geq \mathbf{r}(c_s, c_s)$ or FillRate($c_{s'}$) < FillRate($c_s$)
      return false
  return true
```

Program 1: Can we remove the set of conflict samples $\mathbf{n}_s$ for s?

Although it may sound unusual to allow removal of existing samples, we have found this essential to maintain an equal fill-rate across all classes at all times. As shown in the right case of Figure 3, the two classes maintain consistent fill-rate throughout the sampling process. Furthermore, even though killing samples may in theory increase the computation, in practice we have found that the number of killed samples usually far below the number of accepted and rejected samples. In fact, as shown in Figure 3, the efficiency brought by the sample removal may actually reduce the total number of trials, making the process even more efficient.

2.3 r-matrix construction

As discussed above, we fill the diagonal entries $\mathbf{r}(i,i)$ of $\mathbf{r}$ as r_i, the user specified intra-class minimum distance. But how should we compute the off-diagonal entries of $\mathbf{r}$? If we fill the off-diagonal entries with 0, our algorithm will reduce to decoupled single-class sampling (i.e. the top row in Figure 2). On the other hand, if we treat the samples as geometric disks and define the off-diagonal entries $\mathbf{r}(k,j)$ as $\frac{r_k + r_j}{2}$, we will get results as in the middle row of Figure 2, where the individual classes can be highly non-uniform

```
function r ← BuildRMatrix($\{r_i\}_{i=0:c-1}$)
  // $\{r_i\}$: user specified per-class values
  // c: number of classes
  for i = 0 to $c$-1
    $\mathbf{r}(i,i) \leftarrow r_i$ // initialize diagonal entries
  end
  sort the $c$ classes into priority groups $\{\mathbf{P}_k\}_{k=0:p-1}$ with decreasing $r_i$
  // classes in the same priority group have identical r values
  $C \leftarrow \emptyset$ // the set of classes already processed
  $D \leftarrow 0$ // the density of the classes already processed
  for k = 0 to $p$-1
    $C \leftarrow C \bigcup \mathbf{P}_k$
    foreach class $i \in \mathbf{P}_k$
      $D \leftarrow D + \frac{1}{r_i^n}$ // $n$ is the dimensionality of the sample space
    end
    foreach class $i \in \mathbf{P}_k$
      foreach class $j \in C$
        if $i \neq j$
          $\mathbf{r}(i,j) \leftarrow \mathbf{r}(j,i) \leftarrow \frac{1}{\sqrt[n]{D}}$ // r is symmetric
      end
    end
  end
  return r
```

Program 2: $\mathbf{r}$-matrix construction for uniform sampling.

caused by samples in other classes "getting in the way".

Our algorithm for computing $\mathbf{r}$ is shown in Program 2. To understand how it works, let's start with two classes ($c = 2$) only. Since each class i will have expected sample density proportional to $\frac{1}{r_i^n}$ in a n-dimensional sample space, the off-diagonal entries r_ϕ of $\mathbf{r}$ should be computed via the following formula so that the total set has the expected density $\sum_{i=0}^{c-1} \frac{1}{r_i^n}$:

$$\frac{1}{r_\phi^n} = \sum_{i=0}^{c-1} \frac{1}{r_i^n} \qquad (2)$$

The bottom row in Figure 2 is produced by $\mathbf{r}$ constructed in this fashion. It can be seen, both experimentally and intuitively, that an r_ϕ value deviating from the one computed via Equation 2 will produce worse results, i.e. a smaller value will produce a less uniform total set as in the top row of Figure 2, whereas a larger value will produce less uniform individual classes as in the middle row of Figure 2. The method described above could also be applied to compute a uniform off-diagonal $\mathbf{r}$ matrix entry value for $c > 2$ classes if they share an identical r value.

However, for $c > 2$ classes with different r values, computing a uniform off-diagonal entry value via Equation 2 will produce suboptimal results. Recall that a Poisson disk sample set possesses a blue noise power spectrum, with an inner ring radius $\frac{1}{r}$ within which the power spectrum has very low energy. However, in multi-class blue noise sampling, the power spectrum of a class c_j with parameter r_j could interfere with the power spectrum of another class c_i with $r_j > r_i$, as the noise/energy outside frequency $\frac{1}{r_j}$ of class c_j would fall within the inner ring $\frac{1}{r_i}$ of class c_i. Thus, to minimize the pollution inside its inner ring $\frac{1}{r_i}$, each class c_i would need to ensure that the union of all classes $\{c_j\}$ with $r_j > r_i$ has as uniform a joint distribution as possible.

We achieve this goal by the algorithm described in Program 2. Below is an intuitive explanation. We begin by assigning the diagonal entries of $\mathbf{r}$ from the user specified parameters $\{r_i\}$. To compute the off-diagonal entries, we first sort the classes by their r values in a decreasing order. We then add them in that order to the set of already considered classes C, while simultaneously computing the

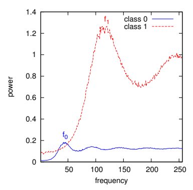


Figure 4: Inter-class spectrum interference. Here we show the radial mean plots for 2 classes with different r values. Each class has its energy peak around frequency $f = \frac{1}{r}$. Note that the peak energy of c_0 falls into the inner ring of c_1, since $f_0 < f_1$ ($r_0 > r_1$).

off-diagonal entries that involve the newly added classes. The computation is performed to ensure that at any moment the set C is as uniform as possible, according to the merit of Equation 2.

Discussion An $\mathbf{r}$-matrix built by the aforementioned method can exhibit discontinuous changes in the off-diagonal entries when the r values of a group of classes change from being identical to slightly different. However, we have not found this to impact distribution quality. For real application scenarios, we believe it will be more common to use classes with either identical or sufficiently different r values (to avoid this issue). In addition, the user can choose to group the classes differently from the default behavior in Program 2.

2.4 Adaptive sampling

So far we have described multi-class dart throwing only for uniform sampling. Here we describe how to extend it for adaptive sampling. The main difference between uniform and adaptive sampling is that the user-specified constants $\{r_i\}_{i=0:c-1}$ for the former could be general spatially-varying functions $\{r_i(.)\}_{i=0:c-1}$ for the latter. This requires us to make corresponding spatially-varying extensions for the $\mathbf{r}$-matrix, the conflict check metric, and the criterion in determining if a sample s' is removable relative to s:

- For building $\mathbf{r}$-matrix, we simply apply the algorithm in Program 2 for every sample location s, i.e. $\mathbf{r}(s) = \text{BuildRMatrix}(\{r_i(s)\}_{i=0:c-1})$.
- For conflict check, we use $\hat{\mathbf{r}}(s, s') = \frac{\mathbf{r}(s,c_s,c_{s'})+\mathbf{r}(s',c_{s'},c_s)}{2}$ instead of $\mathbf{r}(c_s, c_{s'})$. This is analogous to the use of $\frac{r(s)+r(s')}{2}$ instead of r for single-class adaptive sampling.
- For Removable(), we use the sample location s in addition to its class number c_s in Program 1; see Program 3.

```
function bool Removable($\mathbf{n}_s$, $s$, $\mathbf{r}(.)$)
    foreach $s' \in \mathbf{n}_s$
        if $\mathbf{r}(s', c_{s'}, c_{s'}) \geq \mathbf{r}(s, c_s, c_s)$ or FillRate($c_{s'}$) < FillRate($c_s$)
            return false
    return true
```

Program 3: Removable() for adaptive sampling. The colored portions highlight differences from the uniform sampling algorithm in Program 1.

3 Multi-Class Soft Disk Sampling

In Section 2 we extend single-class hard disk sampling for multiple classes of samples. Here, we extend the method further by placing a soft disk centered on each sample. These soft disks behave like energy blobs with local support. They do not have hard boundaries and thus can intersect each other, but the amount of overlap is subject to an energy penalty which, when minimized, produces a uniform distribution. The main advantage of our soft disk sampling is that it serves as a good complement to hard disk sampling, offering explicit control for sample count ($\{N_i\}_{i=0:c-1}$ for c classes) while producing distributions with more spatial uniformity. Even though relaxation [Lloyd 1982] can typically achieve these benefits, we have found that it might not produce good results for the multi-class setting.

Below, we define the notion of a soft disk and an energy function measuring distribution uniformity, followed by a soft dart throwing method that minimizes this energy for blue noise sampling.

3.1 Uniformity measurement

We quantify sample *uniformity* via the following formula:

$$\mathbf{E}(s) = \sum_{s' \in S} \omega(c_s, c_{s'}) \phi_{s',\sigma(s,s')}(s)$$
$$\phi_{s',\sigma(s,s')}(s) = e^{-\frac{(s-s')^2}{\sigma(s,s')^2}} \qquad (3)$$

where s is the query sample for energy value, s' any sample in the sample set S, $\omega(c_s, c_{s'})$ a user specifiable scalar weight factor for class combination $(c_s, c_{s'})$, $\phi_{s',\sigma}$ a Gaussian blob with center s' and width σ that depends on the sample pair (s, s'). Intuitively, $\mathbf{E}(.)$ tends to be smaller for an S with a more uniform sample distribution. Below we provide more details about the parameters:

ω This parameter $\omega(c_s, c_{s'})$ allows the user to specify different importance to different class combinations. We have found it adequate to simply set ω = 1, treating all classes equally.

σ The width σ of a blob $\phi_{s',\sigma(s,s')}(s)$ depends not only on its center s' but also the query sample s. This follows naturally from our multi-class hard disk sampling algorithm presented in Section 2, as the desired spacing between two samples s' and s depends on not only their respective class ids but also their locations for adaptive sampling. Intuitively, $\sigma(s, s')$ should be proportional to $\hat{\mathbf{r}}(s, s')$ in Section 2.4, so that the blob $\phi_{s',\sigma(s,s')}(s)$ properly measures the energy according to the desired distance between s and s'. In our experiments we have found that $\sigma(s, s')$ = 0.25 × $\hat{\mathbf{r}}(s, s')$ works well.

r To evaluate $\hat{\mathbf{r}}$ we will need to know the sample spacing parameters $\{r_i\}$. These are not given explicitly in soft disk sampling, but can be estimated from the specified sample counts $\{N_i\}$ by setting $r_i = r_{i,max}$, the average inter-sample distance computed from the maximum packing of N_i samples.

3.2 Soft dart throwing

One possible method to generate samples minimizing Equation 3 is to extend the multi-class dart throwing algorithm in Section 2 for soft disk samples. Our soft dart throwing algorithm is similar to its hard counterpart, with the major difference being that instead of rejecting a new trial sample s' when it is too close to any existing samples, we always accept s'. This feature allows the user to exactly control the final number of samples across all classes. To help ensure the sample uniformity, we perform multiple attempts of s' and pick the one with minimum energy $\mathbf{E}(s')$ among all trials. (A similar idea is used in best candidate dart throwing [Mitchell 1991].) During the initial phase of the algorithm when the domain Ω is sparsely populated, it might be wasteful to perform many trials. To speed this up, the user can optionally specify a threshold energy $\mathbf{E}_t$ to allow early termination of trials when a trial sample s' with $\mathbf{E}(s') < \mathbf{E}_t$ is found. We have found that $\mathbf{E}_t$ = 0.01 works well. Note that Equation 3 measures only spatial uniformity and could favor a regular distribution. Our soft dart throwing method, due to its stochastic nature, avoids such potential regularity.

4 Analysis

We use the methods in [Lagae and Dutré 2008] to analyze the spatial and spectrum properties of sample distributions. For spatial uniformity, we utilize the relative radius $\rho = \frac{r}{r_{max}}$, where r is the minimum spacing between any pair of samples and r_{max} is the average inter-sample distance computed from the maximum packing

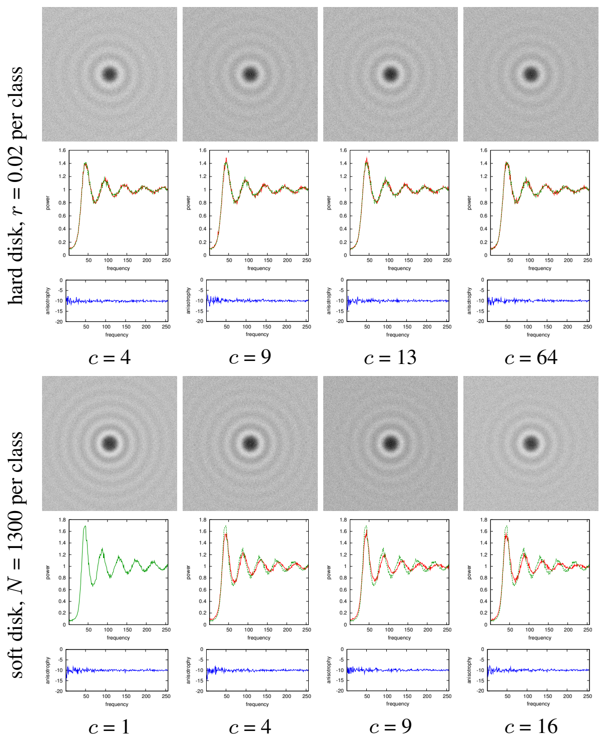


Figure 5: Spectrum results for different number of classes. From top to bottom (within each group): power spectrum, radial mean power, and radial variance/anisotropy. Each column is produced with a different number of classes c as indicated. For easy comparison, we overlay the ground truth mean curve ($c = 1$) in green color with all other cases ($c > 1$).

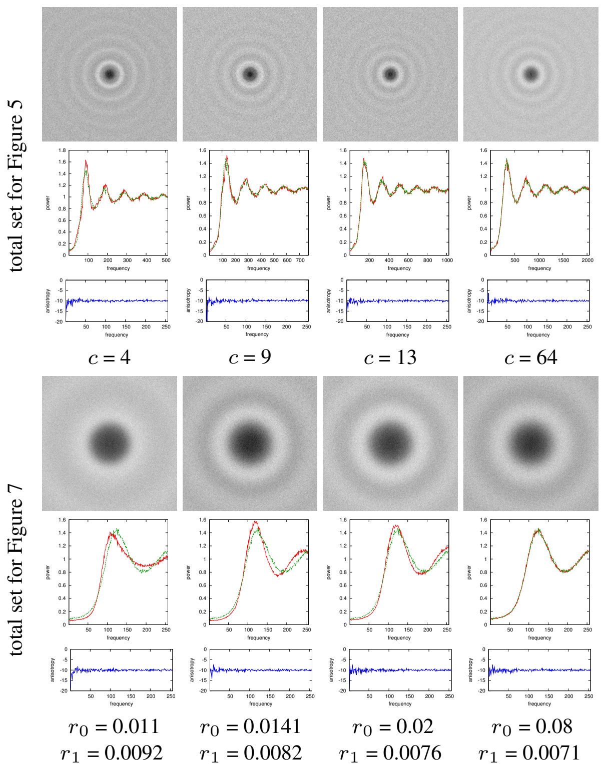


Figure 6: Total sample set distribution. For the radial mean plot, we overlay the ground truth produced by single-class dart throwing as green curves for easy comparison. Shown here are only hard disk results; the soft disk ones have similar quality.

of a given number of samples. For spectrum analysis, we compute the Fourier power spectrum and measure the radial mean and variance/anisotropy, all averaged over 10 runs.

Single-class soft dart throwing Under the single-class setting, soft dart throwing produces results exhibiting good spatial uniformity with $\rho = 0.75$. The power spectrum analysis also confirms the quality, as shown by the $c = 1$ case in Figure 5. Due to its stochastic and non-iterative nature, soft dart throwing does not tend to settle down into hexagonal shaped local minimums as in traditional Lloyd relaxation [Lloyd 1982] and thus could be used as an alternative for applications that require an exact sample count.

Number of classes We start our analysis for multi-class sampling with the simplest case where all the classes have the same parameters, i.e. r for hard disk sampling and N for soft disk sampling. As shown Figure 5, the multi-class statistics remain similar to the single-class setting across a variety of c numbers.

As recommended by [Lagae and Dutré 2008], ρ should be in the range $[0.65\ 0.85]$ for single-class blue noise sampling. However, for the multi-class setting, we have found that $\rho \in [0.65\ 0.70]$ is achievable but beyond that may require excessive number of trials or iterations. Fortunately, this issue does not seem to worsen progressively with the increasing number of classes; due to our $\mathbf{r}$-matrix construction algorithm, the inter-class r values decrease with the increasing number of classes, thus they tend to cancel each other out in terms of imposing additional constraints. All results shown in the paper have $\rho \geq 0.67$ unless indicated otherwise.

Non-uniformity Next we examine what happens if the classes have different parameters. We start with the simplest case of only two classes as shown in Figure 7. We produce several sets of 2-class sample sets with different r_0 and r_1 values so that the total is a sample set with the same r value (where $\frac{1}{r^n} = \frac{1}{r_0^n} + \frac{1}{r_1^n}$). We start with similar r_0 and r_1 on the left and with increasing disparity towards the right. Here, we can observe several interesting facts:

- Class 0 remains indistinguishable from single-class sampling. However, class 1 might deviate from the single-class results, as manifested by the small "humps" between their radial mean curves. These humps are caused by the spectrum peaks of class 0, centered at frequency $\frac{1}{r_0}$ as discussed in Figure 4.
- The deviation between class 1 and ground truth is less obvious when r_0 and r_1 are either sufficiently similar or sufficiently dissimilar. For the former, the hump will happen around the existing peak of class 1, making it non-obvious. For the latter, class 0 simply has too few samples to have a major impact on the power spectrum of class 1. However, even for the case with maximum discrepancy ($r_0 = 0.02$ and $r_1 = 0.0076$) we have not found noticeable differences in sampling results.

Class priority As we discussed in Section 2.2, generating all classes together allows us to maintain consistent fill-rates and thus distribution quality among all classes. Generating the classes sequentially does not allow us to do this, and would require us to specify criteria for stopping the generation of one class and starting another one. This might not be easy as it is very hard to predict if an earlier class would over-constrain the generation of a later one. Furthermore, generating the classes sequentially might actually harm the distribution quality; as illustrated in Figure 8, when classes c_0 and c_1 are produced prior to c_2, c_2 might have little rooms left, resulting in a non-uniform distribution. This effect is particularly pronounced for the soft disk sampling case. However, we wish to emphasize that both our hard and soft disk sampling algorithms

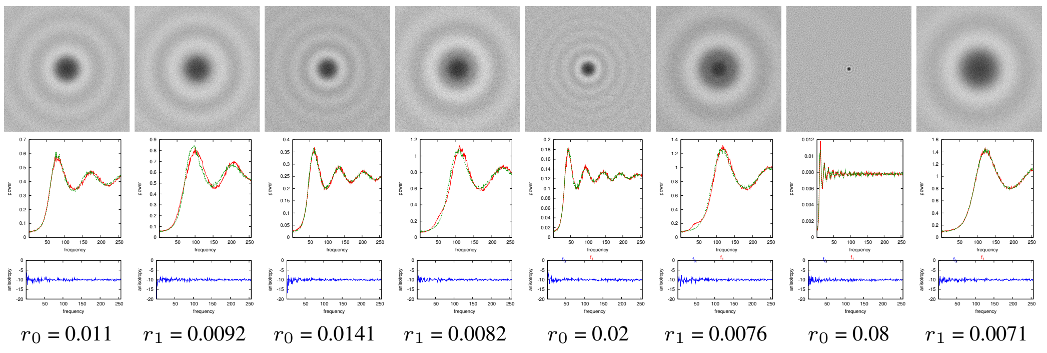


Figure 7: Spectrum results for two classes with different r values. Here we show the hard disk results for easy explanation; the soft disk results have similar but slightly more pronounced effects. Each pair of columns is produced together with different r_0 and r_1 values so that their total is a sample set with $r = \frac{0.01}{\sqrt{2}}$. For the radial mean plots, we overlay the single-class ground truth as green curves for easy comparison.

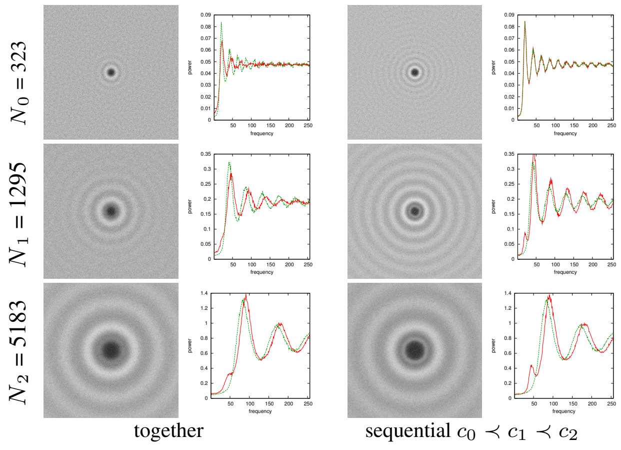


Figure 8: Class priority. Here we generate 3 classes of samples either together (left group) or sequentially (right group). We use soft- rather than hard-disk sampling due to more pronounced effects. In the radial mean plots, the red/green curves correspond to multi/single-class results. Note that generating the classes sequentially would introduce more low frequency noise for the classes produced latter.

can be used to generate the sample classes sequentially if the user desires so.

Total sample set Figure 6 shows the power spectrums for the total sample distribution from selected sets of results in Figure 5 and Figure 7. As shown, the power spectrums of the total sets stay close to the ground truth blue noise profiles. The deviation is most obvious when there are a small number of classes and/or when the classes have similar r values. When there are a larger number of classes, the probability that a sample has a neighbor in a different class is higher, thus geometrically making the entire sample set more similar to a single-class distribution; see the progression in the top cases. When the classes have dissimilar r values, the class with larger r has fewer samples to impact the overall power spectrum; see the progression in the bottom cases.

Performance Using a simple grid/tree data structure [Bridson 2007; Wei 2008] for storing samples and checking conflicts, our current implementations are able to achieve reasonable performance, as tabulated in Table 1. For soft disk sampling, we usually cut off the Gaussian blobs beyond 3σ, thus localizing all energy updates and evaluations. The performance decreases with an increasing number of classes since the sample placement is more constrained, incurring more computations during the generation process. Since a grid/tree data structure allows us to perform the conflict check in constant time, the total time for generating N samples is χN, where χ is the ratio of the total number of trials over N. We are not yet able to determine an accurate formula for χ since it depends on not only the number of classes but also certain implementation details, e.g. how accurately the available space is tracked to reduce futile trials. (We measure the timing in Table 1 via uniform random sampling as in classical dart throwing without such empty-space tracking.) We thus believe that χ could be better determined along with more definite future accelerations of our algorithms.

# classes	1	2	3	4	5	6	7	8
hard disk	9.85	3.80	2.74	2.24	1.36	1.15	0.99	0.65
soft disk	0.23	0.17	0.135	0.13	0.13	0.12	0.086	0.084

Table 1: Performance of our algorithms. All performance numbers are in K-samples/second, and measured on a laptop with a 2.50 GHz CPU + 2 GB RAM. The 1-class case serves as a reference for others.

5 Applications

Here, we show several applications for multi-class blue noise sampling. Depending on the particular application needs, either hard-disk or soft-disk sampling may be more suitable. Specifically, the former is better for applications that have strict sample spacing requirement and are flexible with regard to the number of samples, and the latter for applications that prefer more uniform spatial distribution and exact specification for the number of samples. Hard disk sampling is also more natural for continuous sample spaces whereas soft disk sampling suitable for both continuous and discrete settings. In addition, hard dart throwing is usually computationally faster as it is easier to accelerate.

5.1 Object distribution

Uniform object placement is often desirable for both scientific (e.g. biological distribution) and artistic (e.g. procedural texture [Lagae and Dutré 2005]) applications. Such a uniform distribution can be achieved by blue noise sampling [Cohen et al. 2003; Lagae and Dutré 2005; Kopf et al. 2006], but existing methods do not explicitly consider the presence of multiple classes of objects. We can apply our approach for this purpose. An example is shown Figure 1 for placing two classes of objects in either uniform or adaptive distribution. Due to the desire to keep minimum distances between objects, we opt for hard disk sampling for this application. Our method could also be applied to place 3D objects (e.g. flowers [Cohen et al. 2003]) for scene design or 2D motifs for pattern generation (e.g. [Lagae and Dutré 2005]).

5.2 Color stippling

In addition to object placement, blue noise sampling can also be employed for stippling with visually pleasing pointillism effects (see e.g. [Kopf et al. 2006; Kim et al. 2008; Balzer et al. 2009]). However, existing stippling results are mostly black-and-white since traditional blue noise sampling can handle only a single class of samples. We can apply our algorithms for multi-color stip-

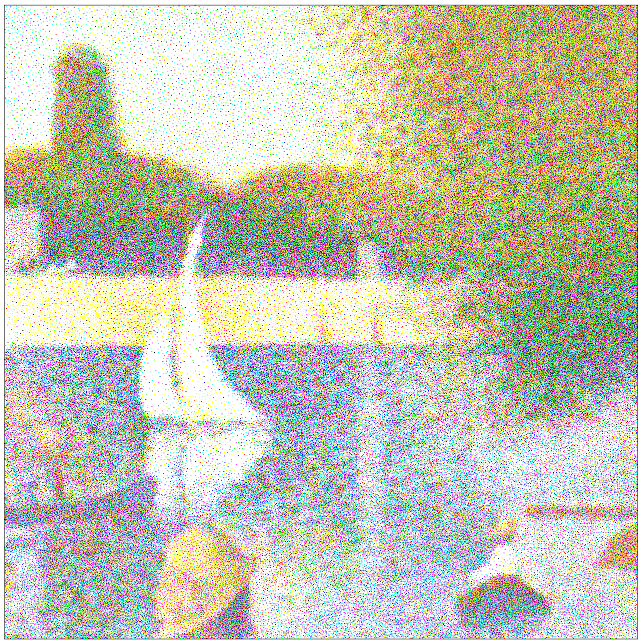


Figure 9: Color stippling result. Using a color image (a corner from Seurat's "A Sunday Afternoon on the Island of La Grande Jatte") as the input importance field, our method produces an adaptive sample set with $\sim$290K color dots in 7 classes (red, green, blue, cyan, magenta, yellow, black) over a white background. (Note: this image might not show up well in print; try it on a computer display and vary the viewing distance.)

pling by using a color image as the input importance field, treating each color channel as a separate class and producing a multi-class output sample set accordingly. Unlike color halftoning which mainly targets discrete regular sample sets, our method allows samples to be placed anywhere and thus provides more of a free-style pointillism effect. As shown in Figure 9, our method can produce reasonably complex color stippling; the colored dots not only follow the input importance field but also maintain a blue noise distribution.

5.3 Sensor layout

The layout of a color sensor array determines the quality of the sampling results as well as subsequent reconstruction algorithms, such as super-resolution. The most widely used layouts usually deploy the RGB sensor elements in a regular grid (or variations thereof); as pointed out in [Ben Ezra et al. 2007], grid layouts are subject to a variety of sampling and reconstruction issues, and the authors recommended the use of Penrose pixels. However, as pointed out in [Kopf et al. 2006; Lagae and Dutré 2008], a Penrose sample layout, even after quality improvement via jittering [Ostromoukhov et al. 2004], still exhibits visible spectrum bias.

Following an analogous line of thinking, we wonder if it is possible to further improve the quality of Penrose pixel layout for color sensors [Ben Ezra et al. 2007] via our approach, treating the RGB sensors as three classes of samples. The comparison is shown in Figure 10. For [Ben Ezra et al. 2007], we use the randomized 3-coloring algorithm in [McClure 2002] to assign the RGB sensor locations. For our result, we use soft disk sampling to specify the exact number of samples. As shown, our result has no bias in the power spectrum as well as no aliasing in a spatial sampling for the zone-plate pattern, a commonly used stress test for evaluating sampling quality [Kopf et al. 2006; Ostromoukhov 2007; Wei 2008].

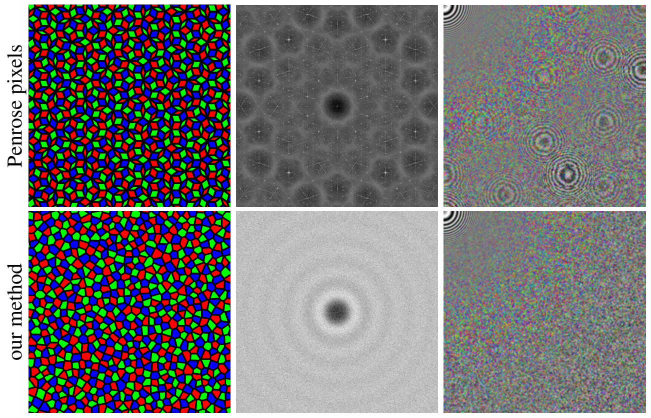


Figure 10: Comparison with Penrose pixels for RGB sensor layout. From left to right: spatial sensor layout, power spectrum, and spatial sampling via the zone-plate pattern. The spectrum results are produced by one of the classes while the spatial sampling via all 3 classes.

5.4 Color filter array design

Penrose pixels [Ben Ezra et al. 2007] and our method would produce better spectrum quality than traditional regular grid sensor layout. However, a regular layout is easier to fabricate, especially for wiring [Ben Ezra et al. 2007]. We have found it possible to maintain the regular layout for the sensor cells, but apply our technique to de-regularize the color filter placement so that the spectrum quality is still improved. This can be achieved by applying our multi-class soft disk sampling algorithm to a pre-determined regular set of samples. Due to the desire to specify the exact number of samples per (color channel) class, this is an application where soft-disk would be more suitable than hard-disk sampling.

As shown in Figure 11, a regular layout (Bayer mosaic in that particular case) causes significant aliasing as expected. One possible solution to reduce aliasing is to place the samples randomly. Even though randomization cannot remove aliasing caused by the underlying regular grid structure, the sampling quality is still improved by de-regularizing the color filter elements. However, this reduction in aliasing is achieved at the expense of more noise. Our soft disk approach, in contrast, reduces aliasing compared to a regular layout while introducing less noise than a random layout. We have also shown result produced by our hard disk approach with a discrete sample domain. To ensure that all sensor elements are utilized, we gradually decrease the r parameters throughout the sampling process similar to [McCool and Fiume 1992]. As shown, even though the hard disk result is better than random placement, it is still worse than the soft disk one. Furthermore, hard disk sampling cannot guarantee an exact number of samples per class.

6 Limitations and Future Work

We have mainly focused on the basic algorithms for multi-class sampling and only lightly touched on the issues of acceleration. Since our algorithms are extensions of dart throwing, we believe they can benefit from a repertoire of previous acceleration techniques, such as [Jones 2006; Dunbar and Humphreys 2006; White et al. 2007; Wei 2008]. Our method is also applicable for constructing multi-class sample tiles [Cohen et al. 2003; Ostromoukhov et al. 2004; Kopf et al. 2006; Lagae and Dutré 2006; Ostromoukhov 2007] as another way to save run-time computation.

Although we have only demonstrated results in 2D, our algorithm is directly applicable to higher dimensional spaces [Bridson 2007; Wei 2008] for scenarios like 3D object distribution. It would also be interesting to extend our approach to sample non-Euclidean do-

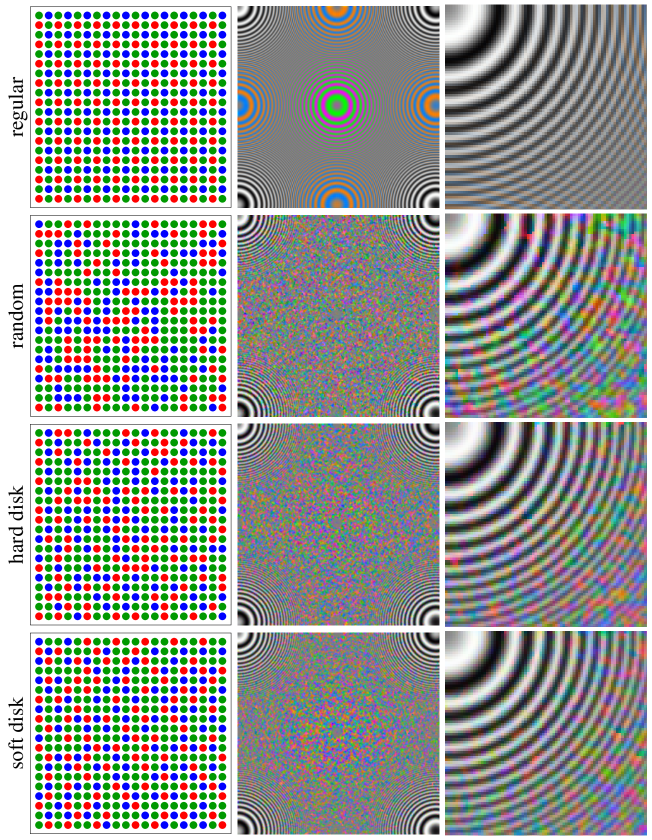


Figure 11: Color filter array design. From left to right: spatial filter array layout, zoneplate sampling, and zoom-in of the low frequency corner. The bottom 3 methods can remove aliasing caused by regular color layout, but not aliasing by the underlying sensor grid (i.e. near the zoneplate corners). Compared to other methods, our soft disk sampling produces more uniform spatial layouts, translating to less noisy sampling results (right column).

mains such as manifold surfaces [Turk 1992; Fu and Zhou 2008; Cline et al. 2009] for rendering and texturing applications.

Acknowledgements Yin Li inspired my pursuit of this project by asking if it is possible to extend [Wei 2008] to place multiple classes of objects. Kun Zhou, Xin Tong, Eric Stollnitz, Brandon Lloyd, Bill Baxter, and the anonymous reviewers provided valuable suggestions. The OPAL Group fixed a SIS bug that once made my submission disappear. I would like to thank Naga Govindaraju, John Manferdelli, Dan Reed, and other incubation team members for their support. Flower images courtesy of `http://www.gif-favicon.com/`. Grass texture courtesy of `http://store.got3d.com/`.

References

Alliez, P., de Verdière, É. C., Devillers, O., and Isenburg, M. 2003. Isotropic surface remeshing. In *Shape Modeling International*, 49–58.

Balzer, M., Schlömer, T., and Deussen, O. 2009. Capacity-constrained point distributions: A variant of Lloyd's method. In *SIGGRAPH '09*, 86:1–8.

Baqai, F., Lee, J.-H., Agar, A., and Allebach, J. 2005. Digital color halftoning. *Signal Processing Magazine, IEEE 22*, 1 (Jan.), 87–96.

Ben Ezra, M., Lin, Z., and Wilburn, B. 2007. Penrose pixels super-resolution in the detector layout domain. In *ICCV '07*, 1–8.

Bridson, R. 2007. Fast Poisson disk sampling in arbitrary dimensions. In *SIGGRAPH '07 Sketches & Applications*.

Cline, D., Jeschke, S., Razdan, A., White, K., and Wonka, P. 2009. Dart throwing on surfaces. In *EGSR '09*, 1217–1226.

Cohen, M. F., Shade, J., Hiller, S., and Deussen, O. 2003. Wang tiles for image and texture generation. In *SIGGRAPH '03*, 287–294.

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

Dunbar, D., and Humphreys, G. 2006. A spatial data structure for fast Poisson-disk sample generation. In *SIGGRAPH '06*, 503–508.

Fu, Y., and Zhou, B. 2008. Direct sampling on surfaces for high quality remeshing. In *SPM '08*, 115–124.

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

Kim, D., Son, M., Lee, Y., Kang, H., and Lee, S. 2008. Feature-guided image stippling. *Computer Graphics Forum 27*, 4, 1209–1216.

Knuth, D. E. 1987. Digital halftones by dot diffusion. *ACM Trans. Graph. 6*, 4, 245–273.

Kopf, J., Cohen-Or, D., Deussen, O., and Lischinski, D. 2006. Recursive Wang tiles for real-time blue noise. In *SIGGRAPH '06*, 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 21*, 1, 114–129.

Lloyd, S. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory 28*, 2, 129–137.

McClure, M. 2002. A stochastic cellular automaton for three-coloring penrose tiles. *Computers & Graphics 26*, 3, 519–524.

McCool, M., and Fiume, E. 1992. Hierarchical Poisson disk sampling distributions. In *Graphics Interface '92*, 94–105.

Mitchell, D. P. 1987. Generating antialiased images at low sampling densities. In *SIGGRAPH '87*, 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*, 488–495.

Ostromoukhov, V. 2007. Sampling with polyominoes. In *SIGGRAPH '07*, 78:1–6.

Pang, W.-M., Qu, Y., Wong, T.-T., Cohen-Or, D., and Heng, P.-A. 2008. Structure-aware halftoning. In *SIGGRAPH '08*, 89:1–8.

Turk, G., and Banks, D. 1996. Image-guided streamline placement. In *SIGGRAPH '96*, 453–460.

Turk, G. 1992. Re-tiling polygonal surfaces. In *SIGGRAPH '92*, 55–64.

Wang, M., and Parker, K. 1999. Properties of combined blue noise patterns. *ICIP 4*, 328–332.

Wei, L.-Y. 2008. Parallel Poisson disk sampling. In *SIGGRAPH '08*, 20:1–9.

White, K., Cline, D., and Egbert, P. 2007. Poisson disk point sets by hierarchical dart throwing. In *Symposium on Interactive Ray Tracing*, 129–132.

Yellott, J. I. J. 1983. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science 221*, 382–385.

Zhou, B., and Fang, X. 2003. Improving mid-tone quality of variable-coefficient error diffusion using threshold modulation. In *SIGGRAPH '03*, 437–444.

Supplementary Materials

(For the electronic version of the paper not the final proceedings.)

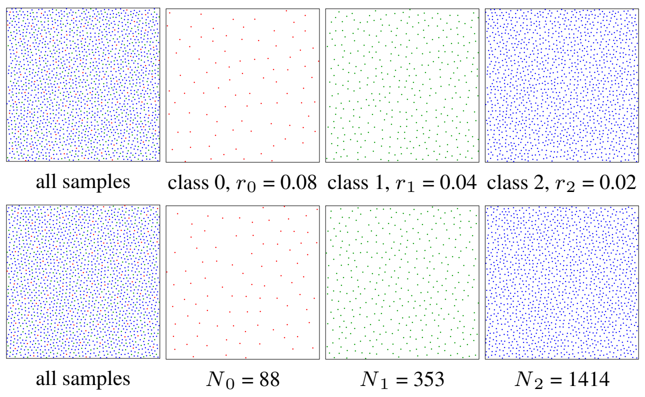


Figure 12: Multi-class blue noise sampling. We propose two flavors of algorithms, Poisson hard disk sampling for explicit control of sample spacing, and soft disk sampling for explicit control of sample count. Here we generate 3 classes of samples and visualize them in different colors. Each class could have its own density, controlled by sample spacing r_i in hard disk sampling (top) or sample count N_i in soft disk sampling (bottom).

In Section A, we provide pseudo-codes for our main algorithms, including hard/soft dart throwing for uniform/adaptive sampling. In Section B, we explain why Lloyd relaxation [Lloyd 1982] may not work well for the multi-class setting. In Section C, we provide an alternative method, termed dart shuffling, that could achieve similar effects to relaxation without its problems. Dart shuffling can be considered as a post process of our soft dart throwing algorithm for further improvement of distribution uniformity. In Section D, we provide additional results. In Section E we describe some implementation details, along with relevant math proofs in Section F. In Section H, we suggest a heuristic to decompose a color image into c channels for c-class color stippling.

A Pseudocode

Here we provide pseudo-codes for our main algorithms, including hard dart throwing for uniform sampling (Program 5), hard dart throwing for adaptive sampling (Program 6), and soft dart throwing (Program 4).

```
function $S \leftarrow$ MultiClassSoftDartThrowing($\Omega, \{N_i\}_{i=0:c-1}, \mathbf{E}(.), \mathbf{E}_t$)
    // $\Omega$: sampling domain
    // $\{N_i\}$: user specified sample count per class
    // $c$: number of classes
    // $\mathbf{E}(.)$: energy function defined in Equation 3
    // $\mathbf{E}_t$: (optional) user-specified energy threshold for speedup
    $S \leftarrow \emptyset$ // final set of samples
    while not enough samples in $S$
        $c_s \leftarrow \arg\min_c$ FillRate $(c)$ // choose the most under-filled class
        $s \leftarrow$ null
        $\mathbf{E}_{min} \leftarrow \infty$
        while $\mathbf{E}_{min} \geq \mathbf{E}_t$ and not enough trials attempted
            $s' \leftarrow$ new sample uniform-random drawn from $\Omega$
            if $\mathbf{E}(s') < \mathbf{E}_{min}$
                $s \leftarrow s'$
                $\mathbf{E}_{min} \leftarrow \mathbf{E}(s')$
            end
        end
        add s to $S$
    end
    return $S$
```

Program 4: Multi-class soft dart throwing.

```
function $S \leftarrow$ MultiClassHardDartThrowing($\Omega, \{r_i\}_{i=0:c-1}$)
    // $\Omega$: sampling domain, $c$: number of classes
    // $\{r_i\}$: user specified parameters for intra-class sample spacing
    // $\mathbf{r}$: $c \times c$ matrix controlling inter-class sample spacing
    $\mathbf{r} \leftarrow$ BuildRMatrix($\{r_i\}_{i=0:c-1}$) // see Program 2
    $S \leftarrow \emptyset$ // final set of samples
    while not enough trials attempted and not enough samples in $S$
        $s \leftarrow$ new sample uniform-random drawn from $\Omega$
        $c_s \leftarrow \arg\min_c$ FillRate $(c)$ // choose the most under-filled class
        if $\forall s' \in S \quad |s - s'| \geq \hat{\mathbf{r}}(s, s')$ // conflict check
            add s to $S$
        else if impossible to add another sample to $c_s$
            // try to remove the set of conflicting samples $\mathbf{n}_s$
            $\mathbf{n}_s \leftarrow \bigcup s' \in S$ where $|s - s'| < \hat{\mathbf{r}}(s, s')$
            if Removable($\mathbf{n}_s, s, \mathbf{r}$)
                remove $\mathbf{n}_s$ from $S$
                add s to $S$
            end
        end
    end
    return $S$

function float $\hat{\mathbf{r}}(s, s')$
    return $\mathbf{r}(c_s, c_{s'})$

function float FillRate($c$)
    return $\frac{\text{\# of existing samples} \in c}{\text{target \# of samples for } c}$ // see Equation 1

function bool Removable($\mathbf{n}_s, s, \mathbf{r}$)
    foreach $s' \in \mathbf{n}_s$
        if $\mathbf{r}(c_{s'}, c_{s'}) \geq \mathbf{r}(c_s, c_s)$ or FillRate($c_{s'}$) < FillRate($c_s$)
            return false
    return true
```

Program 5: Multi-class hard dart throwing for uniform sampling.

```
function $S \leftarrow$ MultiClassHardDartThrowing($\Omega, \{r_i(.)\}_{i=0:c-1}$)
    // $\{r_i(.)\}$: spatially-varying parameters for intra-class sample spacing
    $\mathbf{r}(.) \leftarrow$ BuildRMatrix($\{r_i(.)\}_{i=0:c-1}$) // see Program 2
    // ... see Program 5 for the rest ...

function float $\hat{\mathbf{r}}(s, s')$
    return $\frac{\mathbf{r}(s, c_s, c_{s'}) + \mathbf{r}(s', c_{s'}, c_s)}{2}$

function bool Removable($\mathbf{n}_s, s, \mathbf{r}(.)$)
    foreach $s' \in \mathbf{n}_s$
        if $\mathbf{r}(s', c_{s'}, c_{s'}) \geq \mathbf{r}(s, c_s, c_s)$ or FillRate($c_{s'}$) < FillRate($c_s$)
            return false
    return true
```

Program 6: Multi-class hard dart throwing for adaptive sampling. The colored portions highlight differences from the uniform sampling in Program 5.

B Multi-Class Lloyd Relaxation

Lloyd relaxation [Lloyd 1982] is a classical method to generate blue noise samples from a given initial configuration. Similar to our soft disk sampling method, it offers explicit control for sample counts. Unfortunately, even though relaxation works well in the traditional single-class setting, we have found it problematic for handling multiple classes of samples. Below, we first describe our extension for the multi-class setting, then demonstrate the issues we have found.

Let S be a set of samples (i.e. sites in the jargon of [Balzer et al. 2009]), $\mathcal{V}$ the Voronoi tessellation generated from S, and V_i the Voronoi region corresponding to sample $s_i \in S$. The uniformity of S can be measured by the following energy function

$$\mathbf{E}(S, \mathcal{V}) = \sum_i \int_{V_i} \varrho(p) \, |p - s_i|^2 \, dp \qquad (4)$$

where p indicates a point in the sample domain Ω and ϱ is a non-negative density function defined over Ω. Lloyd relaxation [Lloyd 1982] minimizes this energy function by iterating the following two steps until meeting some termination criterion:

Voronoi generate the Voronoi tessellation $\mathcal{V}$ from the sample set S

Centroid move each sample $s_i \in S$ to the centroid m_i of the corresponding Voronoi region $V_i \in \mathcal{V}$

$$m_i = \frac{\int_{V_i} \varrho(p) p dp}{\int_{V_i} \varrho(p) dp} \quad (5)$$

For multi-class sampling, we can extend Equation 4 as follows:

$$\mathbf{E}(S, \mathcal{V}, C) = \sum_{j=0}^{2^c-1} \omega(C_j) \sum_{s_i \in S_j} \int_{V_{ij}} \varrho(p_j) \left| p_j - s_i \right|^2 dp_j \quad (6)$$

where (as in Section 2) C indicates the set of classes, c the number of classes, C_j the j^{th} possible class combination (out of 2^c total possibilities), $\omega(C_j)$ the weight factor for class combination C_j (to be explained below), $\mathcal{V}_j$ the Voronoi tessellation formed from the sample set $S_j = \{s \mid c(s) \in C_j\}$, and $V_{ij} \in \mathcal{V}_j$ the Voronoi region corresponding to s_i in the context of class combination C_j.

To minimize Equation 6, we can extend traditional Lloyd relaxation as follows:

Voronoi for each class combination C_j (2^c total for c classes), generate the Voronoi tessellation $\mathcal{V}_j$ from the set of samples S_j with class ids belong to C_j, i.e. $S_j = \{s \mid c(s) \in C_j\}$

Centroid move each sample $s_i \in S$ to the centroid m_i of the corresponding Voronoi regions $\{V_{ij} \in \mathcal{V}_j\}_{j=0:2^c-1}$

$$m_i = \frac{\sum_{j=0}^{2^c-1} \omega(C_j) \int_{V_{ij}} \varrho(p_j) p_j dp_j}{\sum_{j=0}^{2^c-1} \omega(C_j) \int_{V_{ij}} \varrho(p_j) dp_j} \quad (7)$$

This class weight parameter, ω, plays a very similar role to the ω in Equation 3. Here, for clarity of presentation, we formulate $\omega(C_j)$ in the most general form for all possible class combinations C_j, even though for practical reasons it might suffice to consider only ≤ 2 classes as in Equation 3. However, this does not really matter here, as we use a very simple 2-class example below to show that relaxation is problematic.

Failure case Multi-class relaxation might get stuck in local minimums with insufficient sample uniformity for a simple reason: for a given sample s, different class combinations may have very different opinions about the desired centroid location to which s should move. A simple 2-class example is shown in Figure 13. There, we first generate an initial sample set via our soft dart throwing algorithm. We then try to improve its uniformity via either multi-class relaxation or dart shuffling. In the first case of relaxation, we set ω = 1. As shown, even though the uniformity of the individual classes improves, the total sample set becomes less uniform. This is evident from both visual inspection as well as the reduced ρ [Lagae and Dutré 2008]. One might conjecture that this is due to the smaller Voronoi region for each sample in the total class, so in the second relaxation case we compensate for this by normalization with respect to the Voronoi region areas. However, even though the uniformity of the total set improves slightly, both the individual classes become less uniform as a consequence. In particular, class 1 actually becomes less uniform than the initial condition. We have found that no matter how we tune the parameters, relaxation never produces better results than the initial condition. We have also tried to generate the classes on after another, i.e. generating one class first via relaxation, keep the samples fixed, and then generate another class following the formulation above, but this still does not improve the problems. Our dart shuffling method, in contrast, improves uniformity for all class combinations as shown in Figure 13.

In addition to traditional relaxation [Lloyd 1982], we have also tried capacity constraint [Balzer et al. 2009] but the same problem persists. This is not surprising, as capacity constraint mainly addresses the overly-regular problem of [Lloyd 1982] by changing the Voronoi step, not the centroid step that is causing problems in the multi-class setting.

C Dart shuffling

Our soft dart throwing algorithm as described in Section 3.2 can generate fairly good initial sample distributions. However, the uniformity of the produced sample sets often leaves room for further improvement (Figure 13). As explained in Section B, one classical method for such improvement is Lloyd relaxation [Lloyd 1982; Balzer et al. 2009], where sample uniformity is gradually improved through an iterative process alternating between Voronoi and centroid computations. Unfortunately, despite its efficacy in single class setting, extension of Lloyd relaxation to multi-class scenario might not work well. The reason is that, unlike in the traditional single class setting where each sample s belongs to only one Voronoi region, s would belong to different Voronoi regions computed through different class combinations. Thus, during the centroid stage different Voronoi regions corresponding to s might have conflicting opinions on where to move it, trapping the computation in a non-uniform local minimum.

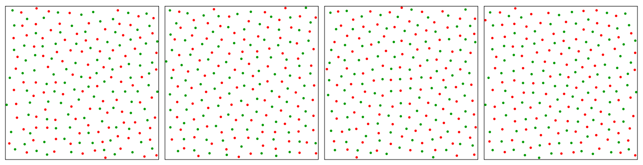


Figure 13: Comparison between multi-class relaxation and dart shuffling. From left to right: initial set generated by soft dart throwing (with 100 samples per class), relaxation by equal class weighting, relaxation by normalization with Voronoi region area, and dart shuffling. Note that the right 3 results are all produced from the left-most case as the initial condition. The uniformity measures ρ are also indicated below each case.

Figure 13 demonstrates a typical scenario. Starting with an initial distribution computed by our soft dart throwing, we aim to improve its uniformity through multi-class relaxation. We perform two trials with different weightings. The first trial weighs all classes equally; as shown, the uniformity of the individual classes improves at the expense of the total union. To improve this, in the second trial we normalize the weights in the centroid computation to favor the total union (which has smaller Voronoi regions). However, even though the uniformity of the total union turns out better, one of the classes becomes worse than the initial condition. We have found that no matter how we tune the parameters, relaxation never produces better results than the initial condition.

To address this issue, we present an alternative method, *dart shuffling*, where a sample set is iteratively improved from a given initial distribution. Unlike relaxation where samples are only locally jittered, we allow samples to be moved anywhere in the domain as long as the movement improves the uniformity of distribution. Specifically, to shuffle a sample s, we draw several candidate locations $\{s'\}$ randomly from the domain Ω and pick the one s' that most minimizes $\max(\mathbf{E}(.))$ among samples around s and s' if these two swap locations. (If Ω is continuous, s' would almost always be empty, and thus the swap reduces to moving s to the location of s'. However, if Ω is discrete, s' might actually be a real sample, thus we need to swap the locations of s and s' to preserve sample counts.) (This process can be considered a form of random descent [Turk and Banks 1996].) Since dart shuffling is primarily used as a refinement from a reasonably uniform initialization by soft dart throwing, we have found it sufficient to shuffle each sample once, in the order of decreasing energy value. We have also found that directly applying the dart shuffling process to a multi-class white noise initialization leads to tougher convergence issues than from our soft dart throwing initialization.

D Additional Results

Here, we provide additional comparisons between multi-class Lloyd relaxation and our dart shuffling method as described in Section C. As shown in Figure 14, we generate 2 classes of samples via different possible relaxation methods: concurrent, where all samples are relaxed together from an initial white noise distribution; sequential, where the classes are produced one after another; and a last case where the samples are initialized from a single class of already relaxed samples. Compared to the result produced by single-class relaxation, it is obvious that none of the multi-class results appear to be nearly as uniform. However, our method still produces more uniform distributions than relaxation, both visually (Figure 14) and by ρ measurement (Table 2).

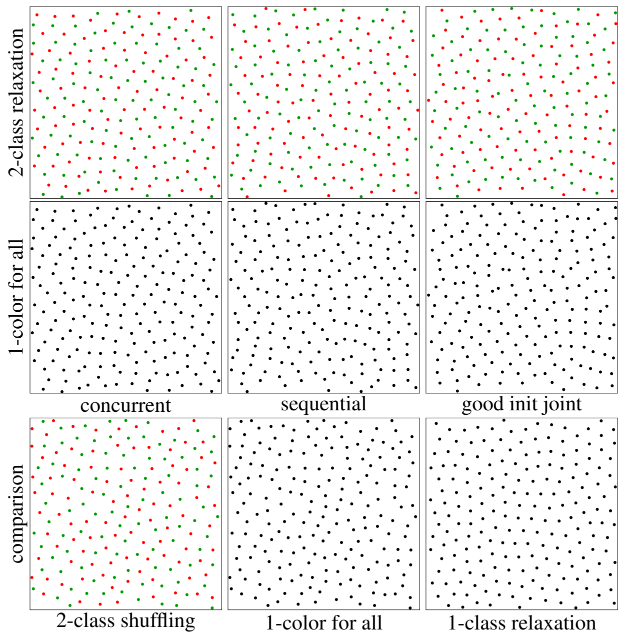


Figure 14: Comparison between multi-class Lloyd relaxation and dart shuffling. On the top row, we use various options to generate 2 classes of samples via multi-class Lloyd relaxation, with single coloring in the middle row for easy comparison. As shown, the results lack sufficient uniformity as compared to multi-class dart shuffling (bottom row). The ground truth provided by traditional (1-class) relaxation is also shown (bottom row right).

	concurrent	sequential	joint	dart shuffle	1-class relax
c_0	0.79	0.72	0.71	0.71	NA
c_1	0.78	0.79	0.77	0.72	NA
all	0.50	0.59	0.43	0.71	0.77

Table 2: The ρ measurements for the results shown in Figure 14.

Figure 16 compares our method with color halftoning. Figure 17 shows that halftoning does not guarantee a general blue noise sampling even in the traditional single class setting. Figure 18, 19, 20, 21, 22, 23 are additional results for Section 4. Figure 24 visualizes a soft disk sampling energy function. Figure 25 provides a larger color stippling result.

Linear subset In addition to the total sample set, due to our $\mathbf{r}$-matrix construction algorithm, we have found that subsets containing classes with r over a certain value would also exhibit satisfactory blue noise properties, as shown in Figure 15. However, in general this is not true for an arbitrary collection of classes; our current method does not enforce this, and we believe it might not be possible to enforce simultaneous blue noise properties for all possible class combinations.

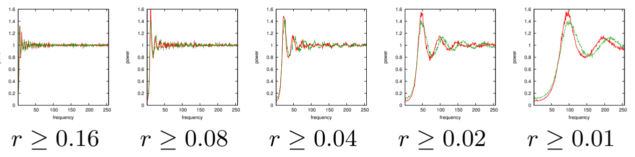


Figure 15: Partial class union. Here we have a 5-class set with r values indicated above. The red radial mean curves are from our results whereas the green ones from 1-class ground truth.

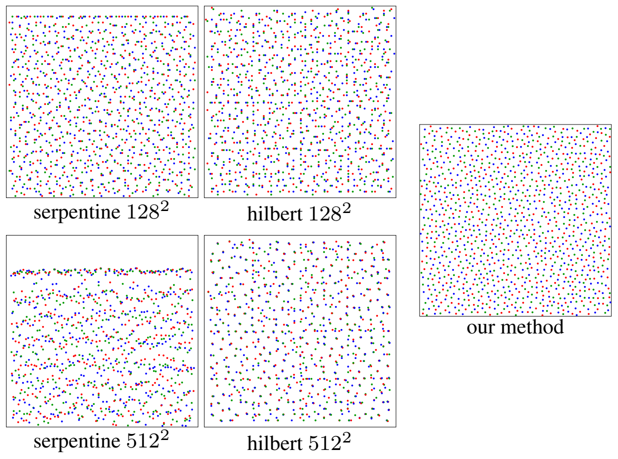


Figure 16: Comparison with color halftoning. The left column is produced by extending [Zhou and Fang 2003] for color/vector error diffusion in a serpentine order, with different output resolutions (128^2 top and 512^2 bottom). The middle column is produced by [Knuth 1987] + a Hilbert curve ordering to reduce the serpentine order artifact.. The right column is produced by our soft disk sampling method in a continuous space without any discretization.

$\mathbf{r}$-matrix computation Figure 22 demonstrates the effects of the $\mathbf{r}$-matrix on sample quality for a 3-class scenario. In the left case, we compute the off-diagonal entries of $\mathbf{r}$ uniformly via Equation 2. In the middle case, we compute $\mathbf{r}$ via Program 2. (We have to use more than 2 classes because otherwise these two cases would be equivalent.) Note that in all cases class c_0 have very similar results to the ground truth; this is to be expected as c_0 has the largest r value, causing it to have the smallest inner ring in the power spectrum and thus immune from contaminations by the other two classes. However, for classes c_1 and c_2, the results are quite different. Since the $\mathbf{r}$ is computed in a prioritized order in the middle case, it clearly has a better distribution for c_2 than the left case. The middle case does have a slightly worse class c_1 than the left case (because c_1 is more constrained), but the difference is quite minor.

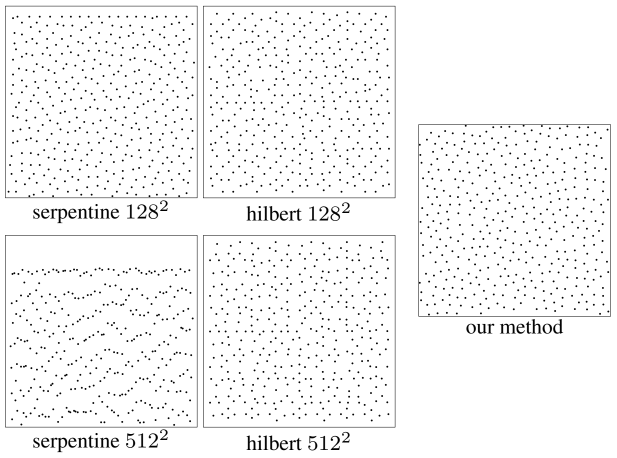


Figure 17: Comparison with scalar halftoning. This is similar to Figure 16, except that we have only one class of samples. Notice the problematic sample distribution produced by halftoning methods.

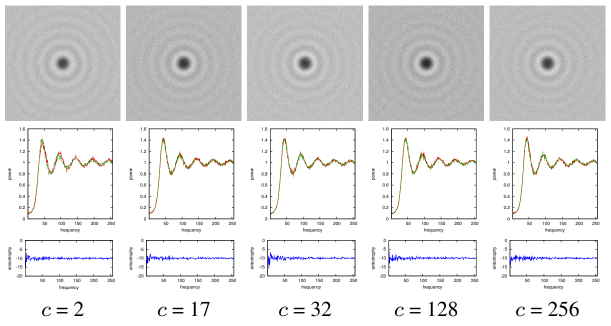


Figure 18: Spectrum results for different number of classes. More results for Figure 5.

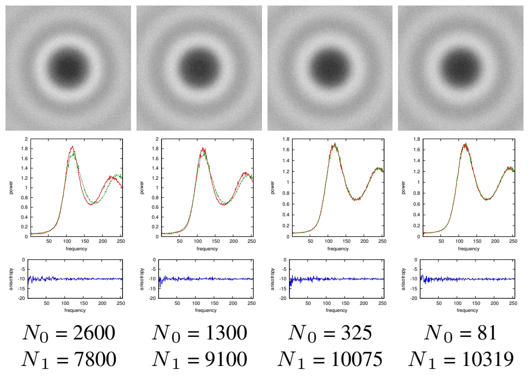


Figure 19: Total sample set distribution. More results from soft disk sampling for Figure 6.

E Implementation

So far we have only described how our algorithms work using the abstract, high-level metaphor of dart throwing. Even though dart throwing can provide high quality, it is also well known for its slow computation speed. Here we provide more implementation details about acceleration and other issues.

E.1 Single-resolution grid

One possibility to accelerate dart throwing is via a grid data structure as described in [Bridson 2007; White et al. 2007; Wei 2008]. The basic idea there is to subdivide the sample domain Ω into grid cells so that each cell can contain at most one sample. Thus, for

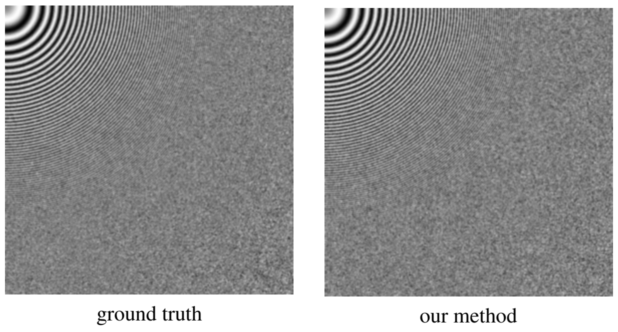


Figure 20: Spatial sampling quality comparison via the zone plate pattern $\sin(x^2 + y^2)$. Each image is produced with 166400 samples (roughly 1 sample per pixel) and filtering with a 3 pixel wide Gaussian kernel. The left image is produced from the single-class ground truth and the right image from our method as shown in the third case in Figure 7 where the two distributions deviate the most.

uniform sampling, one only needs to examine a constant number of cells surrounding a new trial sample for conflict check. This grid data structure can be trivially extended for our multi-class uniform sampling algorithm, and we could also use the grid cells to track the available remaining space for each class of samples (for the purpose of estimating impossibility to add samples in Program 5). However, since the grid has to be fine enough to accommodate the minimum value in our $\mathbf{r}$-matrix while the conflict check has to be conservative enough for the maximum value in the $\mathbf{r}$-matrix, the computation cost of a single-resolution grid implementation of our algorithm will increase linearly with respect to the ratio of the $\max$ and $\min$ values of $\mathbf{r}$. This situation is further exacerbated in adaptive sampling.

E.2 Multi-resolution tree

The aforementioned issues for a single-resolution grid could be addressed by a multi-resolution tree structure as described in [Wei 2008]. The basic idea there is to store larger samples (in terms of $r(.)$) at a lower resolution of the tree while the smaller ones at higher resolutions, so that the conflict check can be performed by looking at a constant number of tree nodes at each resolution.

The multi-resolution single-class algorithm in [Wei 2008] could be combined with our single-resolution multi-class algorithm in Program 6. The hybrid algorithm for multi-resolution multi-class sampling is summarized in Program 7. Our main idea is to use c individual trees to store each class of samples, and for each new trial sample s we perform conflict check across multiple trees for all existing samples that have potential for conflict. (The algorithm can be visualized as the spatial overlay of c individual trees for each class of samples.) Since the number of samples stored in each tree can be quite different, we also subdivide the trees on demand instead of all together. This means that at any time the trees can have a different number of levels. We traverse the nodes of any tree level in a randomized order to avoid bias (as discussed in [Wei 2008]).

The main difference between our algorithm and [Wei 2008] is the cross conflict check between a trial sample s and a tree T for different classes. Specifically, when s is generated from its own (same class) tree T_s, s must be drawn from a highest resolution leaf cell $\square_s$ and thus all the math properties of [Wei 2008] hold. However, when s is conflict-checked against a different tree T, $\square_s$ might not even have a twin cell in T, or the twin cell exists but is not on the highest resolution of T. Thus, the algorithm in [Wei 2008] cannot be directly applied. To handle these situations, we extend the approach of [Wei 2008] as follows. Let the trial sample s be generated at level l from T_s. In [Wei 2008], the conflict check is performed

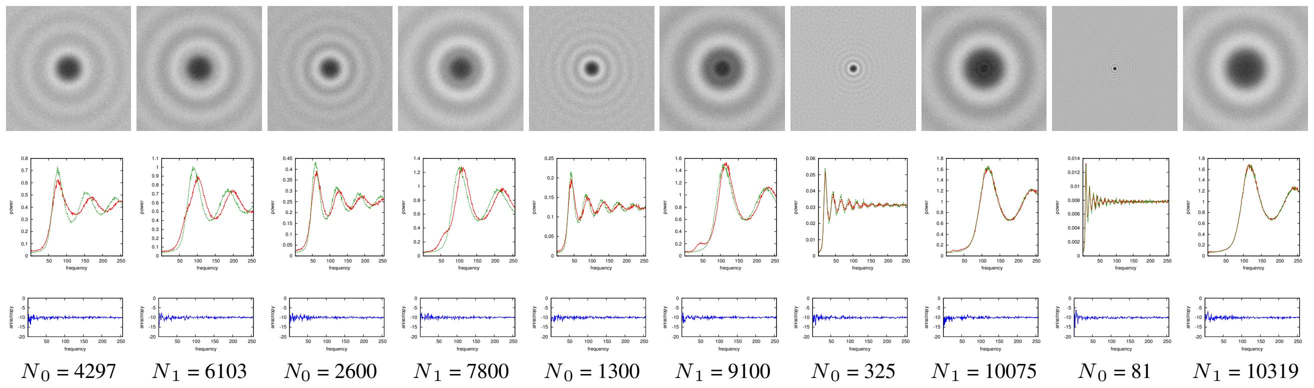


Figure 21: Spectrum results for two classes with different N values. Here we show the soft disk results for Figure 7. Each pair of column is produced together with different N_0 and N_1 values so that their total is a sample set with $N = 10400$.

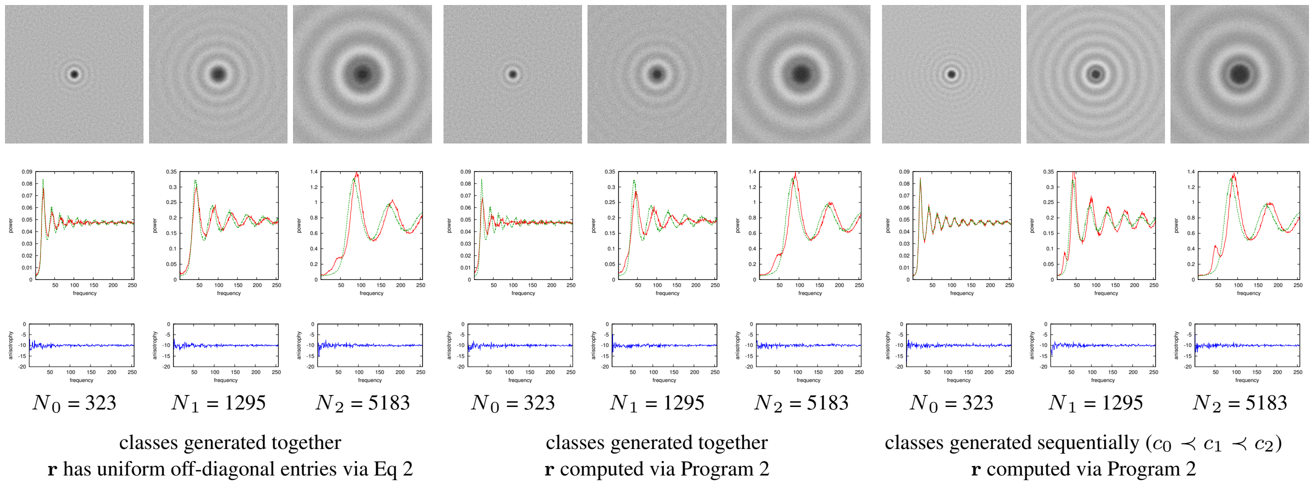


Figure 22: $\mathbf{r}$-matrix computation and class priority. Here we show the soft- rather than hard-disk results because the effects are more pronounced. In the radial mean plots, the red curves correspond to multi-class results, and the green curves the single-class ones. Note that a $\mathbf{r}$-matrix with uniform off-diagonal entries would cause noiser power spectrum for c_2 (compare the radial mean plots between the left and middle cases for c_2 in frequency range $[0\ 50]$). Generating the classes sequentially would introduce low frequency noise for the classes produced latter (compare the middle and right cases).

by looking at, for each resolution l' from l to 0, a set of cells within $3\sqrt{n}\mu(l')$ distance from the ancestor cell $\square(l')$ containing s. In our approach, we simply look at cells within the same vicinity of the ancestor cells not only within the same tree T_s but also at all other trees T. To accommodate for the aforementioned situations, we make the following modifications: (1) we only examine cells that actually exist in T and (2) if $l_{max}(T) > l$, for any non-leaf cell $\square$ at level l of T, we have to examine all samples contained within its sub-tree. (This situation never happens in T_s as $l_{max}(T_s) = l$.)

To mathematically prove that the algorithm is correct, simply follow the math proofs in [Wei 2008] and take into account the fact that $\forall s \in \Omega$, the off-diagonal entries in $\mathbf{r}(s, ., .)$ are all smaller than its diagonal entries according to our $\mathbf{r}$-matrix construction algorithm in Program 2. See Section F for math details.

E.3 Impossibility Estimation

A core component of our algorithm is to estimate when it is impossible to add samples to a specific class. For uniform sampling, this can be precisely estimated by tracking the remaining available space for each class in the merit of [Dunbar and Humphreys 2006]: for each newly added sample s in class i, it will strike out a spherical region centered at s with radius $\mathbf{r}(i, j)$ from the remaining available space for class j. However, it is not clear how to extend this strategy for adaptive sampling where $\mathbf{r}(.)$ can be spatially-varying. In addition, the algorithm can be quite complex to implement, especially for high dimensional spaces [Bridson 2007; Wei 2008].

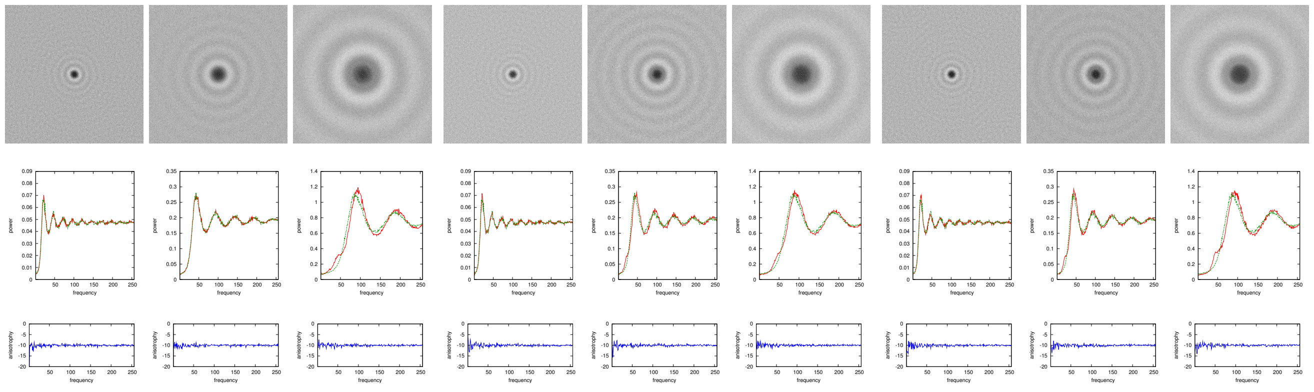
Fortunately, for our algorithm, all we need is a rough estimation of impossibility rather than an 100% exact measurement. In our current implementation, we simply use the grid/tree cells to track available regions: for each newly added sample s in class i, we strike out cells for class j that are entirely within the spherical region centered at s with radius $\mathbf{r}(s, i, j)$. (Note that this works for both uniform and adaptive sampling, and we strike out cells only for the sake of impossibility estimation, not really removing them from the candidate list $\{\square\}_j^{l_j}$ in Program 7.) We have found this strategy work well in practice.

F Math Details

Claim F.1 *The hyper-sphere radius utilized in Program 7 is conservative enough to check all potential conflicts (for both the* mean *and* max *metrics).*

Proof For clarity, in the discussion below we mainly use the max metric (as in [Wei 2008]); claims about the mean metric could be proved analogously.

Let's consider the conflict check between a new sample s in class c_s and an existing sample s' in class $c_{s'}$. When $c_s = c_{s'}$, the situation reduces to the single-class algorithm in [Wei 2008]. When



$r_0 = 0.04$ $r_1 = 0.02$ $r_2 = 0.01$ $r_0 = 0.04$ $r_1 = 0.02$ $r_2 = 0.01$ $r_0 = 0.04$ $r_1 = 0.02$ $r_2 = 0.01$

classes generated together
r has uniform off-diagonal entries via Eq 2

classes generated together
r computed via Program 2

classes generated sequentially ($c_0 \prec c_1 \prec c_2$)
r computed via Program 2

Figure 23: **r**-matrix computation and class priority. Here we show the hard disk sampling results for Figure 22.

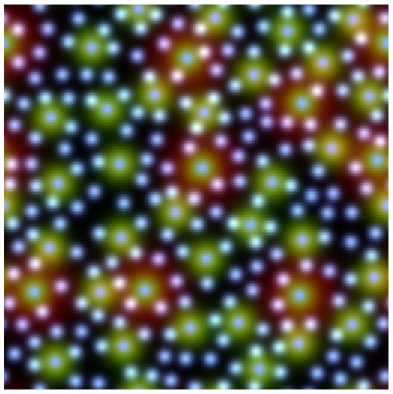


Figure 24: Soft disk sampling energy visualization. This example contains 3 classes with number of samples 8 (red), 32 (green), and 196 (blue).

$c_s \neq c_{s'}$, note that (1) Claim A.1 in [Wei 2008], which states that for any existing sample s generated in tree level l we have $\mathbf{r}(s, c_s, c_s) \leq 2\sqrt{n}\mu(l)$, still holds true for each individual class, and (2) for each trial sample s produced in level l, $\mathbf{r}(s, c_s, c_s)$ must be $\leq 2\sqrt{n}\mu(l')$ for each $l' = 0$ to l, in order to survive the conflict check within its own tree T_{c_s}, as in the last statement of Claim A.2 in [Wei 2008]. These two properties, together with the fact that for each s', $\mathbf{r}(s', c_{s'}, c_s) < \mathbf{r}(s', c_{s'}, c_{s'})$ (i.e. the inter-class distances are smaller than intra-class ones due to our construction algorithm in Program 2), inform us that to check conflict between a trial sample s and a node $\square(l') \in$ level l' ($\leq l$ of s) of tree $T_{c_{s'}}$ of a different class $c_{s'}$, the distance $3\sqrt{n}\mu(l')$ would be enough to conflict-check s with all samples $\in \Omega(\square(l'))$. This, in turn, allows us to check all samples contained in non-leaf cells at level l as well as all leaf cells at level $l' < l$ of tree $T_{c_{s'}}$. ▌

G Original Dart Shuffling Algorithm

Below we describe an older version of our dart shuffling algorithm that we designed in order to apply to an arbitrary initialization, notably multi-class white noise distribution. However, we have found that doing so would incur excessive number of iterations for the sample set to converge to a good multi-class blue noise distribution. Eventually, we resolved this issue via our soft disk sampling algorithm as a much better initialization. Below, we briefly describe our old algorithm for bookkeeping purposes. It is also summarized in Program 8 for easy reference.

The dart shuffling is an iterative process. Within each iteration, we first identify the sample s with maximum energy $\mathbf{E}(s)$ among all samples. We then move the sample to another location so that $\max(\mathbf{E}(.))$ is most reduced. This can be achieved by attempting multiple trials locations s' and pick the one where swapping the locations of s and s' would most reduce $\max(\mathbf{E}(.))$. The precise meaning of *swap* depends on whether there is a sample at s' (if the sample space Ω is continuous s' should never collocate with an existing sample, but if Ω is discrete this might happen); if so we swap the locations for the two samples, and if not we simply move s to s'. We iterate this process until the uniformity of sample distribution barely changes or when a maximum number of iterations are reached. The former can be achieved by either a threshold value for sample movement or a threshold ρ_t value measuring uniformity [Lagae and Dutré 2008].

H Color Channel Decomposition

Let I be the input multi-channel importance field (e.g. a color image). Our goal is to decompose it into multiple mono-color importance fields $\{I_i\}_{i=0:c-1}$ so that each I_i serves as the importance field for a particular sample class. Let $\vec{c}_i$ denotes the (normalized vector valued) color of I_i. We can then factor I_i into $\vec{c}_i \Upsilon_i$ where Υ_i is a scalar importance field. We can formulate all these into the following equation:

$$I = \sum_{i=0}^{c-1} I_i = \sum_{i=0}^{c-1} \vec{c}_i \Upsilon_i \tag{8}$$

subject to the hard constraint that all Υ_i must be ≥ 0. When the number of output channels $c >$ the number of input channels, this decomposition becomes under-constrained and multiple possible solutions exist. To make sure all sample classes are well represented, our goal is to perform a decomposition so that the element-sum $|\Upsilon_i|$ are as similar to each other as possible. (Thus, our goal is quite different from traditional color decomposition methods.)

To achieve this goal, we use the following simple heuristic. For each input sample/pixel, we sort its color channels in order of increasing values. We then visit the input channels in that order, subtract away values from output channels that have already been assigned, and split the remaining value equally among all output channels that are not yet assigned. See Program 9 for a summary of our algorithm. We have found this heuristic works well in practice, and have used it to produce color stippling, as shown Figure 9.

```
function MultiClassAdaptiveSampling($\Omega$, $\mathbf{r}(.)$, $k$)
  // $\Omega$: sampling domain in $n$-dimension
  // $\mathbf{r}(.)$: $c \times c$ $\mathbf{r}$-matrix defined over $\Omega$; see Program 6
  // $k$: maximum number of trials per node
  // use separate trees to track each class of samples
  $\{T_i(0)\}_{i=0:c-1} \leftarrow$ BuildNDTreeRoots($\Omega$) // hypercubes covering $\Omega$
  foreach class i   $l_i \leftarrow 0$   // track the leaf level number for each $T_i$
  foreach class i   $\{\square\}_i^{l_i} \leftarrow$ randomized list of (leaf) nodes $\in T_i(l_i)$
  while not enough trials attempted and not enough samples in $\{T_i\}$
    $\jmath \leftarrow \arg\min_c$ FillRate $(c)$ // choose the most under-filled class
    if $\{\square\}_\jmath^{l_\jmath} = \emptyset$ // no more leaf nodes to sample from; try subidivide $T_\jmath$
      $T_\jmath(l_\jmath + 1) \leftarrow$ Subdivide($\Omega$, $\mathbf{r}(.)$, $T_\jmath(l_\jmath)$)
      if $T_\jmath(l_\jmath + 1) = \emptyset$   break   // impossible to add another sample
      $l_\jmath \leftarrow l_\jmath + 1$
      $\{\square\}_\jmath^{l_\jmath} \leftarrow$ randomized list of (leaf) nodes $\in T_\jmath(l_\jmath)$
    end
    $\square \leftarrow$ PopFront($\{\square\}_\jmath^{l_\jmath}$) // take the head of the randomized list
    $s \leftarrow$ ThrowSample($\{T_i\}$, $\Omega(\square)$, $\mathbf{r}(.)$, $k$, $l_\jmath$)
    if $s$ is not null
      add $s$ to $\square$
    else if impossible to add another sample to class $\jmath$
      // try to remove the set of conflicting samples $\mathbf{n}_s$
      $\mathbf{n}_s \leftarrow \bigcup s' \in S$ where $s$ and $s'$ are in conflict
      if Removable($\mathbf{n}_s, s, \mathbf{r}(.)$) // see Program 6
        remove $\mathbf{n}_s$ from $\{T_i\}$
        add s to $\square$
      end
    end
  end

function $T(l+1) \leftarrow$ Subdivide($\Omega$, $\mathbf{r}(.)$, $T(l)$)
  $i \leftarrow$ class number for $T$
   foreach node $\square$ of $T(l)$
    if $\exists s \in \square$ and $\sqrt{n}\mu(\square) > \mathbf{r}(s, i, i)$
    // subdivide $\square$ only if likely to add another sample
    // $\mu(\square)$ is the cell size of $\square$
      subdivide $\square$ into $2^n$ child nodes // $n$ is the dimension of $\Omega$
      migrate $s$ into the child $\square'$ where $s \in \Omega(\square')$
    end
   end
  $T(l+1) \leftarrow$ newly created nodes
  return $T(l+1)$

function $s \leftarrow$ ThrowSample($\{T_i\}$, $\Omega(\square)$, $\mathbf{r}(.)$, $k$, $l$)
  foreach trial = 1 to $k$
    $s \leftarrow$ sample uniformly drawn from $\Omega(\square)$
    if $\forall s' \in T \;\; |s - s'| \geq \max\left(\mathbf{r}(s, c_s, c_{s'}), \mathbf{r}(s', c_{s'}, c_s)\right)$
    // this can be done by examining only $s' \in$ neighbor nodes in $T_{c_{s'}}$
    // within hyper-sphere of radius $3\sqrt{n}\mu(l')$ at level $l'$ = 0 to $l$
    // for each node in level $l$ of $T_{c_{s'}}$ look at samples under its subtree
    // can also use the mean metric $\frac{\mathbf{r}(s,c_s,c_{s'})+\mathbf{r}(s',c_{s'},c_s)}{2}$ in Program 6
    // in this case, use hyper-sphere radius $5\sqrt{n}\mu(l')$ above
    // and $\frac{1}{2}\mathbf{r}(s, i, i)$ in Subdivide()
      return $s$
  end
  return null

function bool Removable($\mathbf{n}_s, s, \mathbf{r}(.)$)
  foreach $s' \in \mathbf{n}_s$
    if $\mathbf{r}(s',\ c_{s'}, c_{s'}) \geq \mathbf{r}(s,\ c_s, c_s)$ or FillRate($c_{s'}$) < FillRate($c_s$)
      or level(s') < level(s) // add on beyond Program 6
      return false
  return true
```

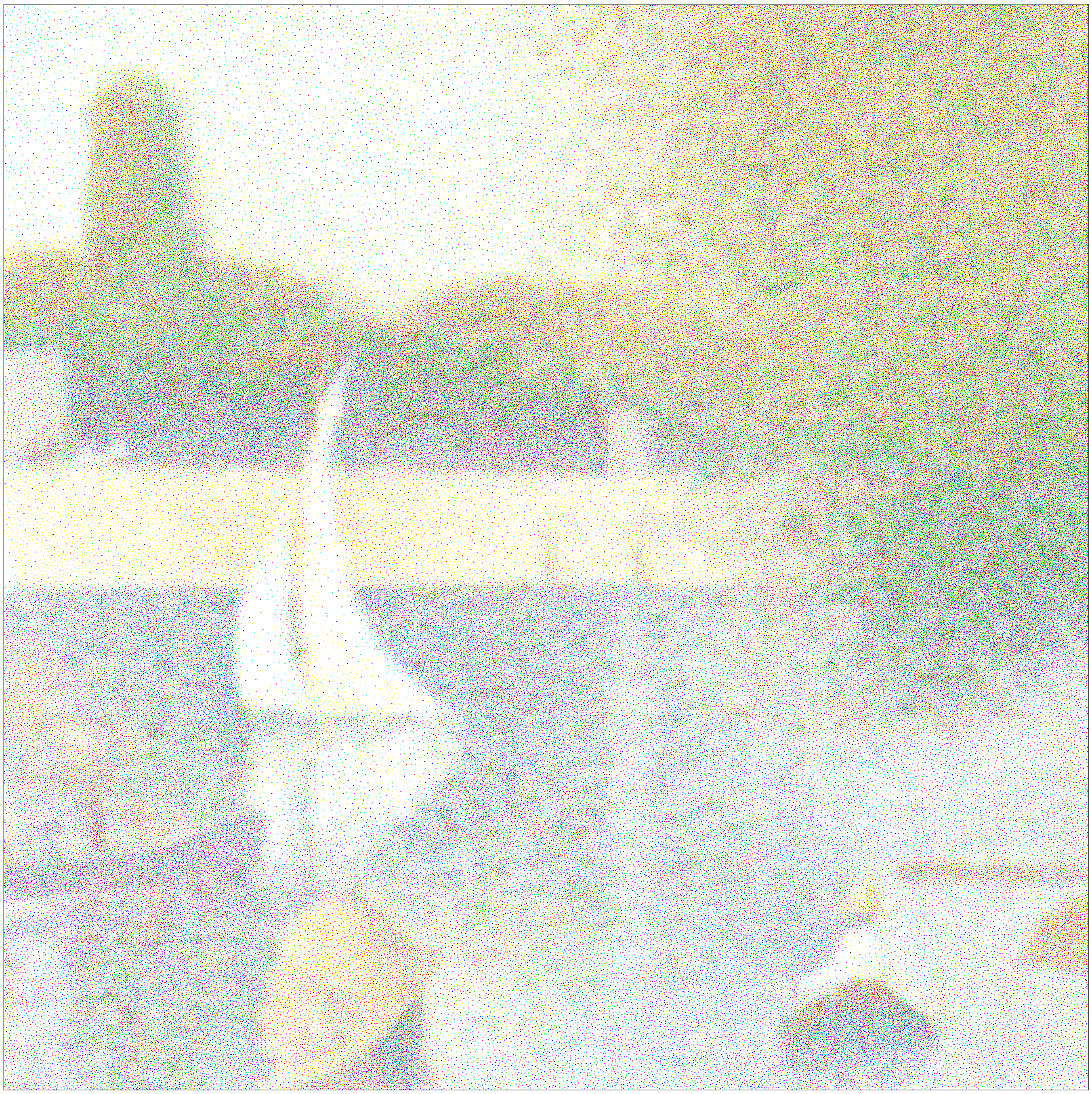
Program 7: Multi-resolution adaptive sampling using separate trees to track each class of samples. The main code is a hybrid of the multi-resolution algorithm in [Wei 2008] and our adaptive sampling algorithm in Program 6. The functions Subdivide() and ThrowSample() resemble the ones in [Wei 2008]; for easy comparison, we highlight the main differences.

```
function $S' \leftarrow$ MultiClassDartShuffling($\Omega, S, \mathbf{E}(.), \rho_t$)
  // $\Omega$: sampling domain
  // $S$: initial samples in c classes
  // $\mathbf{E}(.)$: energy function defined in Equation 3
  // $\rho_t$: user threshold value for $\rho$
  done $\leftarrow$ false
  while not done and $\rho < \rho_t$ and not enough trials attempted
    change $\leftarrow$ false
    foreach $s \in S$ from high to low energy $\mathbf{E}(s)$
      $s' \leftarrow$ FindTrough($s, \Omega, S, \mathbf{E}(.)$)
      if $s'$ is not null
        Swap($s$, $s'$) // swap locations of $s$ and $s'$ but keep their class ids
        change $\leftarrow$ true
        break
      end
    end
    if not change
      done $\leftarrow$ true
    end
  end
  return $S$

function $s_{min} \leftarrow$ FindTrough($s, \Omega, S, \mathbf{E}(.)$)
  $\mathbf{E}_{min} \leftarrow \mathbf{E}(s)$
  $s_{min} \leftarrow$ null
  while not enough trials attempted
    $s' \leftarrow$ uniform random sample from $\Omega$
    Swap($s$, $s'$)
    $\mathbf{E}_{new} \leftarrow \max\{\mathbf{E}(\text{FindPeak}\,(s, S, \mathbf{E})), \mathbf{E}(\text{FindPeak}\,(s', S, \mathbf{E}))\}$
    if $\mathbf{E}_{new} < \mathbf{E}_{min}$
      $\mathbf{E}_{min} \leftarrow \mathbf{E}_{new}$
      $s_{min} \leftarrow s'$
    end
    Swap($s$, $s'$) // undo previous swap
  end
  return $s_{min}$

function $s' \leftarrow$ FindPeak($s, S, \mathbf{E}(.)$)
  // find the peak sample $s' \in \mathbf{n}_s$, the neighbor samples near (including) $s$
  return $\arg\max_{s'}\{\mathbf{E}(s'), s' \in S$ and $s' \in \mathbf{n}_s\}$
```

Program 8: Multi-class dart shuffling.

```
function $\{\Upsilon_i\}_{i=0:c-1} \leftarrow$ ColorChannelDecomposition($I, \{\vec{c}_i\}_{i=0:c-1}$)
  // $I$: input importance field, e.g. a color image
  // $\{\vec{c}_i\}_{i=0:c-1}$: output colors in c classes
  // $\{\Upsilon_i\}_{i=0:c-1}$: the corresponding weight fields for the output colors
  foreach pixel/sample $p \in I$
    $\{\Upsilon_i(p)\}_{i=0:c-1} \leftarrow$ PixelDecomposition($I(p), \{\vec{c}_i\}_{i=0:c-1}$)
  end
  return $\{\Upsilon_i\}_{i=0:c-1}$

function $\{\upsilon_i\}_{i=0:c-1} \leftarrow$ PixelDecomposition$\left(\{p_j\}_{j=0:c'-1}, \{\vec{c}_i\}_{i=0:c-1}\right)$
  // $\{p_j\}_{j=0:c'-1}$: input pixel in $c'$ color channels
  // $\{\upsilon_i\}_{i=0:c-1}$: output weight pixel
  $\{\text{assigned}_i\}_{i=0:c-1} \leftarrow 0$
  foreach $p_j$ in ascending order of value
    $\upsilon \leftarrow p_j$ - $\sum_i \text{assigned}_i \upsilon_i \vec{c}_{ji}$
    $w \leftarrow \sum_i(\text{assigned}_i == 0$ and $\vec{c}_{ji} > 0)$
    foreach $i$ from 0 to $c - 1$
      if $\text{assigned}_i == 0$
        $\upsilon_i \leftarrow \frac{\upsilon}{w\vec{c}_{ji}}$
        $\text{assigned}_i \leftarrow 1$
      end
    end
  end
  return $\{\upsilon_i\}_{i=0:c-1}$
```

Program 9: Color channel decomposition.



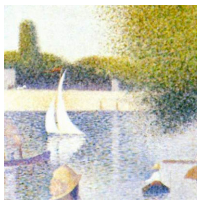


Figure 25: Color stippling result. This is the same sample set as in Figure 9, but we plot the dots in a smaller size and enlarge the image to make the sample locations more distinguishable. (Doing so makes the stippling effect whiter than usual.) The source image, shown on the left, is from `http://scienceblogs.com/cognitivedaily/upload/2009/02/visual_illusion_may_explain_th/monnier2.jpg`.