FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Analysis of Forced Random Sampling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Daniel Cornel
Matrikelnummer 0726194

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr.techn. Robert F. Tobler

Wien, 7.8.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Analysis of Forced Random Sampling

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Daniel Cornel
Registration Number 0726194

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr.techn. Robert F. Tobler

Vienna, 7.8.2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Daniel Cornel
Rüdengasse 12/11, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Abstract

Stochastic sampling is an indispensable tool in computer graphics which allows approximating complex functions and integrals in finite time. Applications which rely on stochastic sampling include ray tracing, remeshing, stippling and texture synthesis. In order to cover the sample domain evenly and without regular patterns, the sample distribution has to guarantee spatial uniformity without regularity and is said to have blue-noise properties. Additionally, the samples need to be distributed according to an importance function such that the sample distribution satisfies a given sampling probability density function globally while being well distributed locally. The generation of optimal blue-noise sample distributions is expensive, which is why a lot of effort has been devoted to finding fast approximate blue-noise sampling algorithms. Most of these algorithms, however, are either not applicable in real time or have weak blue-noise properties.

Forced Random Sampling is a novel algorithm for real-time importance sampling. Samples are generated by thresholding a precomputed dither matrix with the importance function. By the design of the matrix, the sample points show desirable local distribution properties and are adapted to the given importance. In this thesis, an efficient and parallelizable implementation of this algorithm is proposed and analyzed regarding its sample distribution quality and runtime performance. The results are compared to both the qualitative optimum of blue-noise sampling and the state of the art of real-time importance sampling, which is Hierarchical Sample Warping. With this comparison, it is investigated whether Forced Random Sampling is competitive with current sampling algorithms.

The analysis of sample distributions includes several discrepancy measures and the sample density to evaluate their spatial properties as well as Fourier and differential domain analyses to evaluate their spectral properties. With these established methods, it is shown that Forced Random Sampling generates samples with approximate blue-noise properties in real time. Compared to the state of the art, the proposed algorithm is able to generate samples of higher quality with less computational effort and is therefore a valid alternative to current importance sampling algorithms.

Kurzfassung

Stochastisches Sampling ist ein für die Computergrafik unerlässliches Hilfsmittel, das die Annäherung komplexer Funktionen und Integrale in endlicher Zeit ermöglicht. Anwendungen, die auf stochastisches Sampling zurückgreifen, sind unter anderem Ray Tracing, Remeshing, Stippling und die Textursynthese. Um den abzutastenden Bereich gleichmäßig und ohne regelmäßige Muster abzudecken, muss die Verteilung räumliche Gleichmäßigkeit ohne Regelmäßigkeit garantieren, was als Blue-Noise-Charakteristik bezeichnet wird. Zusätzlich sollen Samples auch entsprechend einer Gewichtsfunktion verteilt werden (Importance Sampling), sodass die Verteilung der Samples global einer gegebenen Dichtefunktion genügt und lokal noch immer gleichmäßig und irregulär ist. Die Erzeugung optimaler Blue-Noise-Verteilungen ist aufwändig, weshalb in der Vergangenheit vor allem an schnelleren Verfahren mit ähnlich guter Qualität gearbeitet wurde. Die meisten dieser Algorithmen sind jedoch entweder nicht für Echtzeitanwendungen geeignet oder haben deutlich schlechtere Verteilungseigenschaften.

Forced Random Sampling ist ein neuartiger Algorithmus für Importance Sampling in Echtzeit. Samples werden erzeugt, indem eine vorberechnete Dithermatrix mit einer auf der Gewichtsfunktion basierenden Schwellwertfunktion verglichen wird. Aufgrund der Konstruktion der Matrix weisen die resultierenden Samples lokal wünschenswerte Verteilungseigenschaften auf und sind zudem global an die gegebene Gewichtsfunktion angepasst. In dieser Arbeit wird eine effiziente und parallelisierbare Implementierung dieses Algorithmus vorgestellt und hinsichtlich der Verteilungsqualität und Laufzeit analysiert. Die Ergebnisse werden sowohl mit dem qualitativen Optimum für Blue Noise Sampling als auch mit dem aktuellen Stand der Technik auf dem Gebiet des Importance Sampling in Echtzeit – dem Hierarchical Sample Warping – verglichen. Mithilfe dieses Vergleichs soll untersucht werden, ob Forced Random Sampling mit gängigen Sampling-Algorithmen konkurrieren kann.

Die Analyse von Sampleverteilungen umfasst einerseits mehrere Diskrepanzmaße sowie die Dichte zur Beurteilung der räumlichen Verteilungseigenschaften und andererseits Fourier- und Differentialbereichanalysen zur Beurteilung der spektralen Verteilungseigenschaften. Mit diesen bewährten Methoden wird gezeigt, dass Forced Random Sampling zur Erzeugung von Samples mit annähernder Blue-Noise-Charakteristik in Echtzeit verwendet werden kann. Verglichen mit dem derzeitigen Stand der Technik liefert das vorgestellte Verfahren Samples höherer Qualität mit weniger Berechnungsaufwand und ist deshalb eine echte Alternative zu gängigen Importance-Sampling-Algorithmen.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Sampling is a ubiquitous task in computer graphics, whose goal is to discretize continuous functions. The most common application might be the output of continuous content on an ordinary display. As the display's pixels are discrete, the content has to be sampled regularly at discrete positions. In general, the *sampling theorem* states that after sampling, a signal can only be reconstructed correctly if the original signal is bandlimited, meaning that its highest frequency is bounded, and has no frequencies above half the sampling frequency. In the case of a display, the sampling frequency and therefore the maximum frequency that can be displayed are fixed and determined by the pixels' size. Higher frequencies cannot be displayed correctly and instead alias as lower frequencies. Since discontinuities in a signal have an unbounded frequency, aliasing is very common in computer graphics and its visually apparent artifacts such as Moiré patterns and jaggies are well known. If the content exhibits discontinuities, these artifacts can only be reduced, e.g., by supersampling, but they cannot be completely removed with regular sampling. Instead, *stochastic sampling* can be applied, which replaces the artifacts of aliasing with noise which is much less distracting to the human eye.

Stochastic sampling has become an indispensable tool in computer graphics, not only for anti-aliasing. The numerical evaluation of complex integrals such as the rendering equation with Monte Carlo integration lays the foundation for realistic lighting algorithms. Many artifacts caused by regular sampling can be avoided by sampling a function at random locations, but completely random sampling does not guarantee the function to be sampled *spatially uniformly*. This means that the random samples do not cover the sample domain evenly, but tend to cluster and form holes randomly. In order to properly approximate the sample domain – in the following denoted with Ω – with a finite number of samples, spatial uniformity is an important property of a sample distribution. An ideal sample distribution would have a high spatial uniformity and a low regularity. One distribution to meet these conditions is the *Poisson disk* distribution. An exemplary sample set is depicted in Figure 1.1 in between regular and uniform random sample sets. Unlike regular samples, Poisson disk samples do not exhibit any regularity, but still

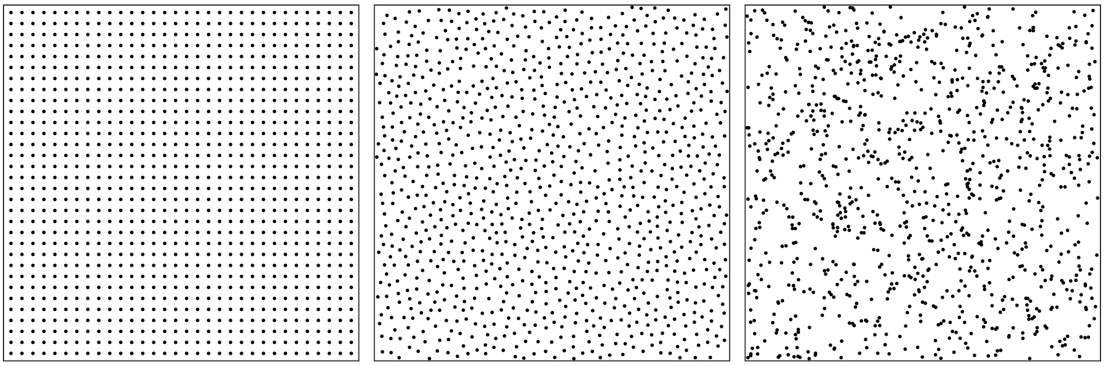
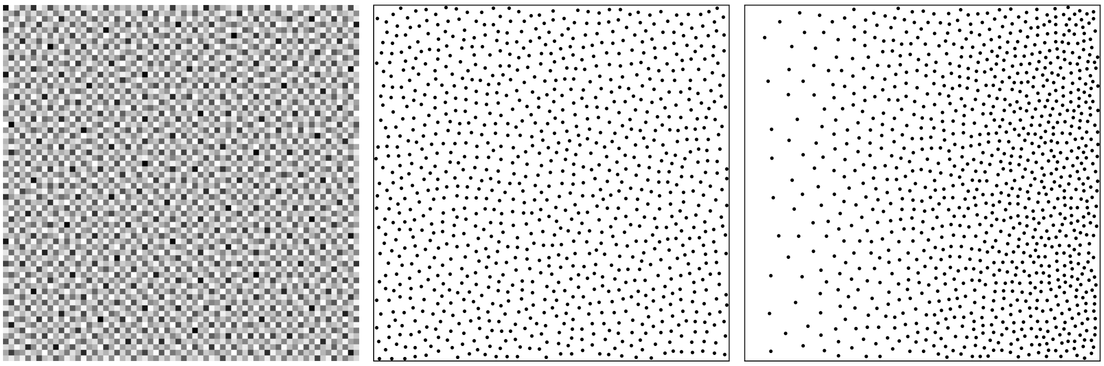

(a) Regular
(b) Poisson disk
(c) Uniform random

Figure 1.1: Different sample distributions

have a much higher spatial uniformity than uniform random samples. The generation of such a distribution is discussed in detail in this thesis. Its power spectrum – a representation of the energy of a signal as a function of frequency, in this context estimated from individual sample sets – is very distinctive and is said to have *blue-noise* properties. This means that it is isotropic, has no concentrated spikes and almost no low-frequency energy. For higher frequencies, the energy concentrates in periodic annuli and finally transitions into noise. Sampling with such a distribution is called *blue-noise sampling*. Because of the good results achievable with blue-noise sampling, the quality of a sample distribution is usually given by the correspondence of its power spectrum to the blue-noise power spectrum.

Although blue-noise sampling as such works well, it does not include any knowledge of Ω. Often, an *importance function* is given or can be estimated, e.g., from the light contribution of an environment map to the scene or reflection properties of a material. This importance function can be used to distribute samples such that regions of high importance are more likely to be sampled. As the number of samples is finite, sampling in general introduces an inevitable error. However, it can be reduced by sampling from the regions which contribute most to the result. This makes *importance sampling* a powerful extension to ordinary stochastic sampling, especially in real-time applications. In a broader sense, the term *sampling* is also used to describe the placement of points in a k-dimensional domain following a specific distribution, even if the points are not used to actually sample a signal. Applications very similar to blue-noise sampling are halftoning and stippling, which approximate an input image by a set of points for either a reduction of color depth or artistic, non-photorealistic rendering. Other applications include the placement of objects in the domain, e.g., for vegetation and natural phenomena, the placement of patterns in textures for texture synthesis, and the remeshing of geometry. In these cases, often the terms *sample density* or *point density* are used for the function that controls the distribution of samples instead of *importance*, because the latter has its roots in statistics and corresponds to an actual probability. This is why throughout this thesis, the more general term *adaptive sampling* instead of importance sampling is used when referring to sampling according to a non-uniform density or importance function. In contrast, sampling based on a constant function is referred to as *uniform sampling*.

Forced Random Sampling (FRS) is a previously unpublished real-time algorithm for adaptive sampling developed by Robert Tobler for the photon tracer used in fast lightmap computation [LTH$^+$13]. The basic algorithm is conceptually very simple and inherently parallelizable, which makes it suitable for real-time applications. The main idea behind FRS is to threshold a large precomputed dither matrix with an importance function. Although the algorithm works with any threshold matrix, a *Forced Random Dithering* [PTG94] matrix is used for its unique properties that allow progressive sampling. The matrix is created by randomly adding points into a repulsive force field near the location of lowest energy, which results in a spatially very uniform, but irregular placement. Thus, thresholding it with a constant c will leave the first c points, which are well distributed and can be used for sampling. Similar to dithering, using a function rather than a constant value for thresholding allows the generation of samples with a local density matching the given importance. A threshold matrix of size 64×64 is illustrated in Figure 1.2 together with results of uniform and adaptive sampling.



(a) Threshold matrix (b) Uniform sampling (c) Adaptive sampling

Figure 1.2: An exemplary Forced Random Dithering matrix (a) and results of Forced Random Sampling with a uniform (b) and linear (c) importance function

Relying on a precomputed matrix allows the expensive computation for the sample distribution to be done offline, while only the lightweight thresholding for sample selection needs to be performed online. However, this also means that sampling is completely dependent on the precomputed matrix. The number of unique samples that can be drawn from this matrix is finite and their locations are discrete. A sample set drawn from such a matrix is likely to exhibit repetitive and regular patterns. The more samples are drawn, the more sampling degenerates to regular sampling. Therefore, a compromise has to be made between the maximum number of samples drawn from a matrix and the distribution quality.

1.2 Problem Statement

Ever since Yellott [Yel83] showed that the distribution of photoreceptors in the eyes of primates roughly follows a Poisson disk distribution, it is assumed to be the ideal distribution for sampling without aliasing and apparent artifacts. The generation of Poisson disk samples is conceptually simple, but computationally expensive. The naive algorithm proposed by Cook [Coo86] works by creating uniform random samples sequentially and discarding them if they are closer than a predefined minimum distance to an already existing sample. This can be sped up by spatially guided or tile-based algorithms, but most of them either reduce the distribution quality of the samples or restrict sampling to few dimensions or to uniform importance functions. Adaptive sampling in real time is possible, but requires even more serious tradeoffs between quality and performance.

The problem is that there is currently no real-time sampling algorithm that combines quality, performance and usability in a satisfying way. Fast real-time adaptive sampling algorithms are not suitable for blue-noise sampling, while algorithms that produce optimal sample distributions are not feasible in real time. Existing accelerations in between are still too slow to be competitive with fast sampling algorithms and are often difficult to implement. As the focus in real-time application is high performance, a novel adaptive sampling algorithm should be as fast and simple as current algorithms, but achieve a higher sample quality.

Forced Random Sampling, a new algorithm for adaptive sampling in real time, could be such an algorithm. As only basic calculations are needed at runtime, it is very fast. The quality of the sample distribution is assumed to be high, based on the visual similarity to Poisson disk samples, but is strongly dependent on the threshold matrix and other parameters. Until now, however, the quality and performance of FRS have not been quantified and compared to current algorithms. Moreover, it has not been investigated how to implement the basic concept of FRS efficiently.

1.3 Research Question

The main question addressed in this thesis is whether Forced Random Sampling is competitive to existing sampling algorithms in terms of quality and runtime performance. A fair comparison between FRS and existing sampling algorithms is only possible if both properties are quantified, which has not been done for FRS yet. The quality of FRS is influenced by matrix generation, thresholding and sample placement, which is why it is necessary to find the ideal parameters for artifact-free sampling first. Likewise, it has to be determined how the concept of FRS can be implemented most efficiently for CPU and GPU to minimize the runtime.

1.4 Methodology

To answer the research questions, the state-of-the-art algorithm for fast adaptive sampling and the unmodified Poisson disk sampling algorithm are implemented and analyzed in the same test environment as FRS. The results of this analysis are supposed to allow an objective comparison and classification of the algorithms based on established measures.

When considering the quality of a sample distribution, this quality first of all needs to be quantified in a meaningful way. For the evaluation of a sample distribution's quality, several established methods exist, which can be divided into *spatial* and *spectral* methods. While spatial analysis is concerned with an estimation of the distribution's spatial uniformity, spectral analysis is used to evaluate the distribution's blue-noise properties and to reveal hidden structures, patterns and flaws. With these methods, the quality of a sample distribution can be expressed in a comparable way. Therefore, the quality and the benefit of FRS can be assessed in comparison to existing algorithms for blue-noise and adaptive sampling.

The creation of the threshold matrix used for FRS itself as well as the thresholding process involve several parameters which are likely to affect the sampling quality. In order to find the right parameters to maximize the quality of FRS, qualitative analyses are performed for various parameter choices. This aims at improving the spectral properties of the sample distributions while also reducing the expected regularity artifacts stemming from the thresholding process. A promising and inexpensive approach to eliminate the regularity artifacts is by jittering samples after thresholding, which is why an additional *jittered FRS* algorithm is discussed in this thesis.

Similarly, in order to optimize the runtime performance of FRS, several different implementation concepts for FRS that emerged before and during the course of this thesis are analyzed regarding their computational effort. Although CPU implementations are provided and benchmarked, too, the GPU implementations are strongly emphasized because of their relevance for real-time sampling. The different FRS implementations are likewise compared to the state-of-the-art real-time sampling algorithm, which is *Hierarchical Sample Warping* (HSW) [CJAMJ05].

The analyses of performance and quality of the relevant algorithms include both uniform and adaptive sampling, where adaptive sampling is performed for different well-known importance functions. First, this aims to investigate whether FRS is a versatile sampling algorithm, and second, it is supposed to allow a general, conclusive statement about all tested algorithms.

1.5 Contributions

1. A formal description of *Forced Random Sampling*, an algorithm for adaptive sampling with results exhibiting approximate blue-noise properties based on thresholding a pre-computed dither matrix.

2. Fast, parallelizable CPU and GPU implementations of Forced Random Sampling for real-time application.

3. A tool for the evaluation of the spatial uniformity of point sets using five different discrepancy measures.

4. A comparison of the proposed FRS implementations to implementations of state-of-the-art algorithms in order to evaluate their overall performance. References for quality are dart throwing for uniform and relaxation dart throwing for adaptive sampling. The reference for speed is the Hierarchical Sample Warping algorithm with Halton samples.

1.6 Outline

In the first part of Chapter 2, a more detailed introduction to sampling is given with a discussion of related work, focusing on the combination of blue-noise sampling and adaptive sampling in real time. In the second part, different methods for the analysis of sample distributions used in this thesis are reviewed. A recapitulation of Forced Random Dithering and especially the matrix creation is given in Chapter 3, followed by a general and formal description of FRS in Chapter 4. Different implementations of FRS are presented and discussed in Chapter 5. For the scope of this thesis, only implementations for sampling in two dimensions are considered. Chapter 6 encapsulates the results of several analyses regarding the suitability of FRS for real-time blue-noise sampling. The results of quality and performance analyses of different FRS implementations and other relevant sampling algorithms are compared and discussed. A detailed description of every analysis and test case is provided as well. The thesis closes with a conclusion and a summary of open problems in Chapter 7.

CHAPTER 2

Related Work

2.1 Stochastic Sampling

The human eye has only a finite number of photoreceptors and therefore also has a Nyquist limit. Yet, humans usually do not experience aliasing artifacts. Yellott [Yel83] studied the eyes of rhesus monkeys and observed that the photoreceptors in the primates' retinas are distributed irregularly with a minimum distance between each other. Such a distribution – a Poisson distribution with a minimum distance constraint between samples – is called a *Poisson disk distribution* with radius r. Its power spectrum is characterized by a lack of concentrated spikes and a lack of low-frequency energy – the *blue-noise properties*. Yellott concluded that because of the special distribution of receptors in the eye, low-frequency aliasing is replaced by much less conspicuous broadband noise, and proposed its use for artificial imaging.

Cook [Coo86] investigated the use of stochastic sampling for ray tracing in order to render complex phenomena such as motion blur, depth of field, soft shadows and glossy objects. He seized Yellott's idea and sketched a simple algorithm to generate Poisson disk sample sets: Pseudo-random samples are generated and inserted into a list. For each new sample, the distance to every sample already inserted is calculated and compared to the minimum distance. If the new sample is too close to an inserted sample, it is rejected, otherwise it is inserted. This is repeated until no more samples can be added. Cook understood that the algorithm, called *dart throwing*, is very expensive and thus recommended the jittering of a regular grid as discussed by Dippé and Wold [DW85] for distributed ray tracing, despite inferior spectral properties.

Since then, blue-noise sampling has been an active field of research with a strong focus on performance. While dart throwing produces unbiased Poisson disk samples, it is infeasible for more than a few thousand samples even in offline application. For real-time application, compromises have to be made between the runtime and the distribution's blue-noise properties. It is also crucial for a wide range of applications to adaptively distribute samples according to a given non-uniform importance function, which strict dart throwing is not capable of. In stippling and halftoning, for example, points need to be denser in dark regions of a given image than in bright ones, and samples for environment map sampling should be placed in the regions of the highest light contribution to the scene. Still, all samples should be well distributed locally.

Quasi-Random Sampling

Because of its simplicity, quasi-random sampling is used frequently as an inexpensive alternative to Poisson disk sampling. Quasi-random sampling, as opposed to (pseudo-)random sampling, is not concerned with creating samples of uniform probability, but of uniform spatial distribution. This is based on the discrepancy theory outlined below and aims at minimizing the error made by approximating an arbitrary space by a finite number of points in a deterministic manner. The most famous low-discrepancy sequence is the Halton sequence S [HS64], which generalizes the van der Corput sequence to k dimensions. Let $b_1, ..., b_k \geq 2$ be prime numbers and

$$i = \sum_{j=0}^{m} a_j(i) b^j, \tag{2.1}$$

the b-ary representation of integer i. From this, the radical inverse function in base b is given as

$$\phi_b(i) = \sum_{j=0}^{m} a_j(i) b^{-j-1}. \tag{2.2}$$

Figuratively speaking, the expansion of i in base b is mirrored by the decimal point. Although the calculation is completely deterministic, the resulting radical inverse appears to be random in $(0,1)^k$. Besides this direct calculation, an incremental calculation of $\phi_b(i)$ based on $\phi_b(i-1)$ has been proposed by Keller [Kel97]. The first n elements of the Halton sequence are then defined as

$$S = x_1, ..., x_n, \quad x_i = (\phi_{b_1}(i), ..., \phi_{b_k}(i)) \in [0,1]^k, \quad i \in \mathbb{N}. \tag{2.3}$$

The closely related Hammersley set for n elements is defined as

$$S = x_1, ...x_n, \quad x_i = \left(\frac{i}{n}, \phi_{b_1}(i), ..., \phi_{b_{k-1}}(i)\right) \in [0,1]^k, \quad i \in \mathbb{N}, \tag{2.4}$$

which has a lower quasi-Monte Carlo error bound than the first n elements of the Halton sequence, but requires n to be know beforehand and does not allow progressive sampling. A third choice for quasi-random sampling is the conceptually very different Sobol sequence [BF88]. It has been investigated along with Halton points by Secord et al. [SHS02] for use in their stippling algorithm, but has been dismissed for its tendency to form undesirable patterns, which is why an insight in its construction is omitted here. For more information on low-discrepancy sampling, the reader is referred to Niederreiter [Nie92].

Blue-Noise Sampling

An approximate Poisson disk algorithm called *point diffusion* has been proposed by Mitchell [Mit87]. Mitchell recognized the strong relationship between sampling and halftoning and deduced a sampling algorithm from the Floyd-Steinberg error diffusion algorithm. The algorithm processes a grid in a scan-line order and maintains a diffusion value for each position. The decision whether to create a sample at a location is the comparison of a constant value to the weighted sum of the previous diffusion values and a random number used to control the density of the samples. Finally, uniform jitter is added to the samples to reduce the regularity introduced by the grid. Contrary to Mitchell's own analysis, Klassen [Kla00] was not able to reproduce Mitchell's results and detect distinct blue-noise properties in the algorithm's results.

To tackle the vague termination criterion of strict dart throwing, Mitchell [Mit91] also proposed the *best candidate* dart-throwing algorithm. Multiple potential random samples are created per iteration and their distance to the closest inserted sample is calculated. The best candidate is the one farthest away from all inserted samples while still fulfilling the minimum distance constraint. The algorithm terminates if no such candidate could be found.

As the validation of the minimum distance constraint of potential samples is the main effort of dart throwing, several spatially guided algorithms have been proposed to limit sampling or validation to a subdomain of the sample domain. Dunbar and Humphreys [DH06] proposed the *scalloped regions* to express the overlapping of the samples' disks and to sample from the free neighborhood around them. Bridson [Bri07] proposed to place samples in the spherical annulus around each sample and keep track of free space in a grid. Jones [Jon06] proposed to maintain the free space of the sample domain in a weighted tree corresponding to the Delaunay triangulation of the inserted sample points. Sampling is then done by randomly traversing the tree. White et al. [WCE07] and Gamito and Maddock [GM09] proposed a quadtree for the maintenance of the free space, Ebeida et al. [EMP$^+$12] used a *flat quadtree*.

Relaxation dart throwing proposed by McCool and Fiume [MF92] performs usual dart throwing with a large r until no more samples can be inserted, i.e., a finite number of samples has been discarded. r is then decreased by a small fraction and dart throwing continues. New samples are added progressively until the desired number of samples has been generated. By allowing varying radii over the sampling domain, relaxation dart throwing can also be used for adaptive sampling: In regions of higher importance, the minimum distance between samples can be reduced such that samples are denser in important regions of the domain. A further contribution of McCool and Fiume was the application of Lloyd's relaxation algorithm [Llo82] to sample sets in order to optimize their distribution properties. In each iteration, a Voronoi tessellation of the sample set is generated and each sample is moved to the centroid of its corresponding Voronoi region. After several iterations, this converges to a centroidal Voronoi tessellation with a minimum-energy configuration of the sample points. By regarding a non-uniform importance function for the calculation of the centroids as done by Secord [Sec02], the algorithm also allows adaptive sampling.

Although Lloyd's relaxation converges faster if the initial sample set already has approximate blue-noise properties, every other distribution can be used as well. For the task of image stippling closely related to sampling, Deussen et al. [DHvOS00] generated an initial point set by pulse-density modulation halftoning of the grayscale image to be stippled. Secord [Sec02] used rejection sampling for a similar task. Since the relaxation converges to a hexagonal lattice, it has to be terminated before the regularity of the sample set becomes too high. Balzer et al. [BSD09] therefore extended Lloyd's method by a capacity constraint. The *capacity-constrained Voronoi tessellation* (CCVT) ensures that all samples have the same capacity, which is computed as the area of a sample's corresponding Voronoi cell weighted by the underlying importance function. This effectively prevents the convergence to a regular structure, but leads to more computational effort. Even with accelerated implementations proposed by Li et al. [LNW$^+$10], Xu et al. [XLGG11], Chen et al. [CYC$^+$12] and de Goes et al. [dGBOD12], CCVT takes seconds up to minutes to generate a few thousand samples.

A conceptually different and parallelizable optimization algorithm based on the physical principles of electrostatics has been proposed by Schmaltz et al. [SGBW10]. In each iteration, the movement of charged particles considering mutual repulsion among samples and attraction based on an importance function is simulated. A very simple optimization algorithm trying to maximize the mutual distances of the samples that converges very fast has been proposed by Schlömer et al. [SHD11].

In order to separate sample generation and sampling, Hiller et al. [HDK01] proposed a sample generation algorithm based on Wang tiles. Wang tiles are quadratic tiles with colors assigned to the four edges. A plane can be tiled by connecting the tiles such that the shared edges between all neighboring tiles have the same color. Depending on the set of Wang tiles and their selection, the tiling can be completely aperiodic or at least without obvious repetitions over an arbitrarily large plane. By filling the tiles with Poisson disk samples, a larger set of samples can be obtained with little computation. The eight sample tiles used by Hiller et al. are created offline with a modified version of Lloyd's relaxation such that the samples of all tiles fit together at the tile borders. The online effort for sampling is then limited to the selection of the right sample sets for aperiodic tiling. Shade et al. [SCM02] proposed a very similar tiling with eight sample tiles created by a modified dart-throwing algorithm. To make the tiled sample set seamless, the minimum distance constraint of samples is also evaluated in neighboring tiles. They pointed out, however, that this modification leads to repetitive artifacts because significantly less samples are placed in the corners of the tiles. Cohen et al. [CSHD03] observed the same artifacts and suggested Lloyd's relaxation as described by Hiller et al. [HDK01] for tile generation. They also proposed a stochastic tile selection algorithm to ensure non-periodic tiling. Each tile is chosen randomly out of all fitting ones, but requires all previous tiles to be chosen already. The stochastic tiling by Lagae and Dutré [LD05] avoids this constraint, allowing a local tile selection. Lagae and Dutré [LD06] also proposed a sample tile generation that subdivides the tile in edge, corner and interior regions. Special edge and corner tiles for each color combination are created and are then used to constrain the dart throwing in the interior regions of the tiles. This way, repetitive artifacts in edge and corner regions of the tiled sample set can be avoided.

A comprehensive survey on methods for the generation of Poisson disk distributions has been published by Lagae and Dutré [LD08] in 2008, which includes every notable algorithm up to that time. They concluded that algorithms based on tiling are the only option for real-time blue-noise sampling, although their resulting sample sets' spectral properties are suboptimal.

The first algorithm to offer real-time applicability and sample distribution of high quality has been proposed by Wei [Wei08]. First, the k-dimensional sample domain is subdivided into grid cells with a size bound by $r/\sqrt{k}$ such that every cell can contain at most one sample. As samples in cells farther away from each other than r do not interact with each other, dart throwing can be performed simultaneously in these cells. This is why multiple independent cells can be grouped together in *phase groups* and are processed in parallel, only the different phase groups are processed sequentially. To avoid artifacts from uniformly sampling in a regular grid, Wei also proposed a multi-resolution approach that allows the first dart-throwing samples to be drawn from the whole sample domain and then subdivides the sample domain into several subdomain cells based on the current level. The resulting algorithm is capable of uniform and adaptive sampling on the GPU at a rate of several million samples per second. However, Gamito and Maddock [GM09] and Xiang et al. [XXSH11] argued that the predetermined processing order of the phase groups is a violation of the uniform sampling condition of strict dart throwing, which also leads to artifacts. Xiang et al. proposed an unbiased but slower parallel dart-throwing algorithm for uniform sampling: First, a dense set of potential sample points is generated. In parallel, each thread then randomly picks one potential sample and assigns a unique random priority to it. If a potential sample is closer than r to an accepted sample, it is rejected. If it is closer than r to another potential sample, the one with the lower priority is rejected. Otherwise it is accepted. The algorithm terminates if all samples of the initial set are either accepted or rejected.

Adaptive Sampling

Adaptive sampling can be done by either distributing new samples adaptively based on an importance function or transforming existing samples to match the given sample density. Analogous to dart throwing in the uniform case, relaxation dart throwing as proposed by McCool and Fiume [MF92] serves as unbiased ground truth for adaptive sampling. The global optimization algorithms based on Lloyd's relaxation or electrostatic halftoning allow adaptive sampling by weighting the centroid calculation resp. the attraction of particles with the importance function. For real-time sampling, Wei's [Wei08] parallel implementation can be used with a subdivision tree for the cells rather than a regular grid. Most other variants of dart throwing, especially the spatially guided ones, cannot be easily extended to adaptive sampling because they rely on a constant r. This is also true for the aforementioned tile-based approaches with few exceptions.

Kopf et al. [KCODL06] proposed the use of recursive Wang tiles with toroidal, self-similar, progressive sample sets. The self-similarity allows tiles to be subdivided locally to increase the sample density based on precomputed subdivision rules. Their progressive generation with relaxation dart throwing allows the sample density to be adjusted accurately to the given importance function. A completely different tiling approach proposed by Ostromoukhov et al. [ODJ04] uses a modified Penrose tiling to tile the sample domain aperiodically. Instead of precomputing a sample set per tile, special pentagonal tiles themselves correspond to the location of a sample. As the tiling is aperiodic, so is the placement of these pentagonal tiles and thus of the resulting samples. To adapt the tiling to an importance function, predefined subdivision rules are used to replace the different tiles by one or more smaller ones. An algorithm proposed by Ostromoukhov [Ost07] uses polyominoes instead of Penrose tiles. Again, the sample domain is tiled aperiodically and tiles are subdivided recursively to match an underlying importance function. Each polyomino corresponds to one sample of which location is predetermined by the polyomino's type. All three adaptive tiling algorithms are able to generate more than one million samples per second.

Secord et al. [SHS02] proposed a stippling algorithm which allows the redistribution of existing sample points in real time, based on the *inversion method* [Nie92] known from statistics.

In the one-dimensional case, a probability density function $p(x)$ is either given or estimated. The cumulative density function of $p(x)$,

$$P(x) = \int_{0}^{x} p(t)\mathrm{d}t, \tag{2.5}$$

is then inverted and each input sample x_i is transformed to $x'_i = P^{-1}(x_i)$. In two dimensions, the transformation in one dimension can be obtained by reducing the second one to a function of average values using the marginal density function of $p(x, y)$,

$$m(y) = \int_{0}^{1} p(x, y)\mathrm{d}x. \tag{2.6}$$

With the cumulative density function of $m(y)$,

$$M(y) = \int_{0}^{y} m(t)\mathrm{d}t, \tag{2.7}$$

the first one-dimensional transformation of each input sample (x_i, y_i) is $y'_i = M^{-1}(y_i)$. The second dimension's cumulative density function is conditional,

$$P(x|y'_i) = \int_{0}^{x} \frac{p(t, y'_i)}{m(y'_i)}\mathrm{d}t, \tag{2.8}$$

the transformation is given by $x'_i = P^{-1}(x_i|y'_i)$. Although the two-dimensional inversion method is fast, the decoupling of dimensions does not preserve the distribution properties of input samples. The results are therefore inferior to those of algorithms which directly distribute samples according to a given importance function instead of transforming them afterwards, as shown by Ostromoukhov et al. [ODJ04] and Kopf et al. [KCODL06]. In contrast to tile-based approaches, however, it is easily possible to use any sample set as input, including uniform random, Poisson disk and low-discrepancy samples. Secord et al. pointed out that for the task of non-photorealistic stippling, Halton samples are an equivalent alternative to Poisson disk samples.

Another transformation method and the current state of the art in real-time adaptive sampling has been proposed by Clarberg et al. [CJAMJ05]. The *wavelet importance sampling* algorithm combines adaptive sampling from a BRDF and an environment map by sparsely evaluating their product represented in the Haar wavelet basis. Following their notation, the Haar basis expansion of a one-dimensional image H of size 2^l is defined as

$$H(x) = \sum_{t} H^{0}_{l,t}\phi^{l}_{t}(x) = H^{0}_{0,0}\phi^{0}_{0}(x) + \sum_{l}\sum_{t} H^{1}_{t,l}\psi^{l}_{t}(x), \tag{2.9}$$

$$\phi_t^l(x) := 2^{l/2}\phi(2^l x - t), \quad \phi(x) := \begin{cases} 1, & 0 \leq x < 1 \\ 0 & \text{otherwise} \end{cases}, \tag{2.10}$$

$$\psi_t^l(x) := 2^{l/2}\psi(2^l x - t), \quad \psi(x) := \begin{cases} 1, & 0 \leq x < \frac{1}{2} \\ -1, & \frac{1}{2} \leq x < 1 \\ 0 & \text{otherwise} \end{cases}, \tag{2.11}$$

where $\phi_t^l(x)$ are the normalized scaling functions and $\psi_t^l(x)$ the orthogonal wavelet basis functions. l is the level and $t = 0, ..., 2^l - 1$ the translation of the functions. $H_{0,0}^0$ is the first scaling coefficient, $H_{t,l}^1$ are the detail coefficients. Simply put, the normalized Haar wavelet basis allows the hierarchical representation of an image as the sum of details at different frequencies. As explained by Clarberg et al., this representation can be used to evaluate the product of two wavelets sparsely. For more information, the reader is referred to Clarberg et al. [CJAMJ05] and Clarberg [Cla12]. The result – also in the Haar wavelet basis – is then used to transform an existing set of samples hierarchically. When reconstructing the product, the scaling coefficients $H_{l,t}^0$ corresponding to the image's local averages are calculated where needed. The probability $P(x \in s)$ of a sample being placed in a region $s = (l, t)$ of the product is then

$$P(x \in s) = 2^{-2l}\frac{H_{l,t}^0}{H_{0,0}^0}. \tag{2.12}$$

The initial sample set is now warped recursively according to these probabilities for each region s in order to adapt the density of the samples to the local conditional probabilities, which the authors call *Hierarchical Sample Warping* (HSW). As with the inversion method, all k dimensions are treated independently for warping. One warping step in two dimensions is illustrated in Figure 2.1. The subset of the sample set corresponding to s is first partitioned along one dimension such that the volume of the two partitions corresponds to the absolute probabilities of each half of the first dimension. The samples of both subsets are then linearly scaled such that the partition lies in the center of s. Each subset is then split and warped analogously along the second dimension. The expected number of samples in the resulting subsets is now proportional to the probabilities of the four quadrants of s. The algorithm then recurses for each of the quadrants. The local density of the final sample set satisfies the given importance function, which in the case of wavelet importance sampling is the product of a BRDF and an environment map. However, Wei and Wang [WW11] pointed out that HSW is not able to preserve the blue-noise properties of the initial sample set, which leads to apparent artifacts when adapting samples to a discontinuous importance function. Clarberg et al. [CJAMJ05] used a Hammersley point set as input for HSW, but again, any sample distribution is possible. In an improved version of wavelet importance sampling which needs lesser precalculations, Clarberg and Akenine-Möller [CAM08] used Poisson disk samples, but did not propose substantial changes to the warping step of the algorithm. An extension to incremental wavelet importance sampling has been proposed by Huang et al. [HCTW07], in which the number of samples varies based on a variance estimation. For the progressive sampling, samples from the Halton sequence are used. Cline et al. [CETC06] proposed an algorithm using HSW with a summed-area table rather than wavelets.

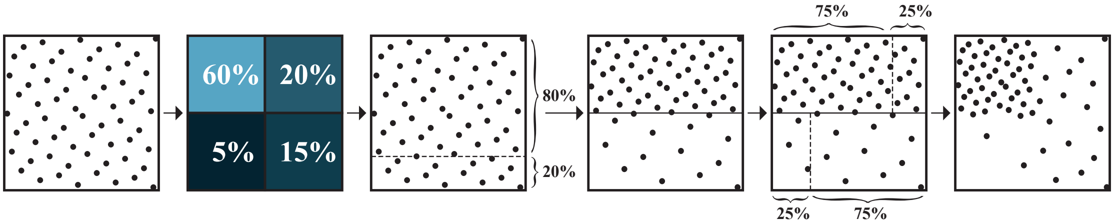


Figure 2.1: Hierarchical Sample Warping. The initial sample set is partitioned and warped once in the first and twice in the second dimension to match the probabilities given by the wavelet tree. The algorithm then recurses for each quadrant, using the warped result as input sample set. Image retrieved from [Cla12].

2.2 Sample Distribution Analysis

In order to evaluate the quality of sample sets and hence the distribution algorithm they originate from, an objective criterion for *quality* has to be found. Some applications of sampling such as ray tracing allow the comparison of results to a ground truth and a judgment based on the variance or the mean squared error. To assess the quality of a sample set directly and in a general manner, several methods for spatial and spectral analysis of the distributions have been proposed. Their least common denominator is the assumption that *spatial uniformity* is a desirable property of a sample distribution, while *regularity* is not, which coincides with the blue-noise properties.

Spatial Analysis

Discrepancy

The most popular measurement of spatial uniformity is the *discrepancy* from statistics, which has been proposed for the use in computer graphics by Shirley [Shi91]. In general, the discrepancy of a point set X of size n expresses how good an arbitrary space is approximated by this point set. Let A be an arbitrary subdomain of the sample domain $\Omega = [0,1)^k$ and let $\|X \cap A\|$ be the number of samples in A. Then, the *local discrepancy* at point (x, y) is

$$d_X(x,y) = \left| \frac{\|X \cap A\|}{n} - \lambda_k(A) \right|, \tag{2.13}$$

where $\lambda_k(A)$ is the Lebesgue measure or k-dimensional volume of A. In the following, only the case of $k = 2$ will be considered. Simply put, the local discrepancy measures the absolute difference between the actual number of samples in A and the expected number of samples that should fall into A, assuming an equiprobable distribution, which in the unit square then equals the area of A. For the calculation of what is known as *discrepancy*, different norms can be used, of which the most important ones are $\mathcal{L}_2$ (*average discrepancy*) and $\mathcal{L}_\infty$ (*worst-case discrepancy*):

$$D_2(X) = \left(\int_0^1 \int_0^1 d_X(x,y)^2 \mathrm{d}x\mathrm{d}y \right)^{\frac{1}{2}}, \quad D_\infty(X) = \sup_{(x,y)\in\Omega} d_X(x,y). \tag{2.14}$$

Dobkin et al. [DEM96] pointed out the strong relation of the worst-case discrepancy to the integration error for a function $f(x)$ by Koksma's theorem,

$$\left| \int_0^1 f(x)\mathrm{d}x - \frac{1}{n}\sum_{i=1}^{n} f(x_i) \right| \leq \frac{1}{n} V(f) D_\infty(X), \tag{2.15}$$

where $V(f)$ is the bounded variation of f and independent from X. The inequality shows that the integration error of the Monte Carlo method is bounded by the worst-case discrepancy of the sample set. The low-discrepancy sequences discussed above are therefore constructed to minimize $D_\infty(X)$ and thus the integration error. However, as can be seen in Equation 2.13, the local discrepancy is dependent on the subdomain A.

For low-discrepancy sequences, only a subset $\mathcal{A}$ of possible subdomains of Ω is considered, namely rectangles $A(x, y) \in \mathcal{A}$ with one corner in the origin and the opposite corner in (x, y). In this case, the discrepancy is more specifically called *star discrepancy*. Although the star discrepancy is the best-known discrepancy, Matoušek [Mat10] argued that it does not capture the properties of a uniform distribution too well. For the average star discrepancy, the weight of points is non-uniform, because points close to the origin contribute more to the discrepancy than points farther away. Also, $\mathcal{A}$ is not rotationally invariant.

For a more general discrepancy measurement, the class of geometric shapes $\mathcal{A}$ has to be re-generalized. Shirley [Shi91] proposed to use axis-parallel boxes $A(u, v, x, y)$ with the two opposite corners (u, v) and (x, y). The class of axis-parallel boxes is invariant to right angle rotations and eliminates the non-uniform weighting of points for the average discrepancy. Following the notation of Dobkin and Gunopulos [DG94], this will be referred to as *rectangle discrepancy*. Still, $\mathcal{A}$ is limited to axis-parallel shapes, which in sampling usually only appear in artificial cases. A general statement of a point set's quality and its distribution function can hardly be made.

To evaluate the approximation of *arbitrary* geometric shapes by a point set, $\mathcal{A}$ has to be rotationally invariant and more general. Shirley [Shi91] mentioned the possible benefit of circles instead of rectangles. Matoušek [Mat10] listed several possible classes, including arbitrary triangles, arbitrary quadrilaterals and arbitrary ellipsoids. In the following, the *quadrilateral discrepancy* and the *circle discrepancy* will be considered. Dobkin et al. [DEM96] proposed the *halfspace discrepancy*, which is calculated by dividing Ω with an arbitrary edge.

Density

Lagae and Dutré [LD08] proposed a spatial measure corresponding to the density of points. Given an arbitrary sample set X of size n, the minimum distance between any two samples is

$$d_X = \min_{x_i, x_j \in X, i \neq j} d(x_i, x_j), \tag{2.16}$$

where $d(.)$ denotes the Euclidean distance between two samples. If X is a Poisson disk sample set, then $d_X \approx r$. The largest minimum distance of a sample set of size n is reached in a hexagonal lattice and is given as

$$d_{max} = \sqrt{\frac{2}{\sqrt{3}n}}. \tag{2.17}$$

The density ρ of X is then defined as

$$\rho_X := \frac{d_X}{d_{max}}. \tag{2.18}$$

Schlömer et al. [SHD11] also proposed an average density $\bar{\rho}_X$ using the average minimum distance of the sample set. According to Lagae and Dutré, $\rho_X \in [0.65, 0.85]$ is a *good* density, based on observations from Poisson disk sample sets. A lower density indicates a lack of spatial uniformity, a higher density indicates regularity. However, this definition is a bit vague. Schlömer et al. argued that point sets optimized with their proposed farthest-point optimization have excellent blue-noise properties with $\rho_X \approx 0.930$. Interestingly, this is very close to the density of a regular grid, $\rho_X = (\sqrt{3}/2)^{1/2} \approx 0.9306$. This suggests that the density ρ_X might be an adequate measure for the spatial uniformity of a point set, but not for its regularity. In contrast to discrepancy, however, ρ_X can also be used for non-uniform point sets, as proposed by Wei and Wang [WW11].

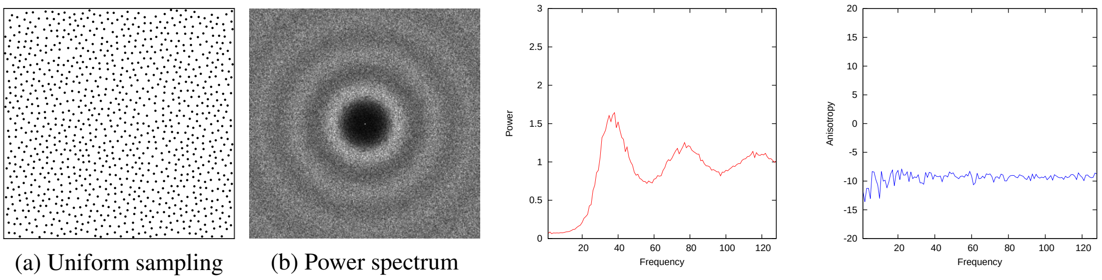
Spectral Analysis

Spectral analysis is the oldest method for analyzing sample distributions. While Yellott [Yel83] still used an optical transform to characterize the blue-noise properties, others [Coo86, Mit87, MF92] used the Fourier transform $\mathcal{F}$. For each sample set X resp. its frequency vector f, a periodogram

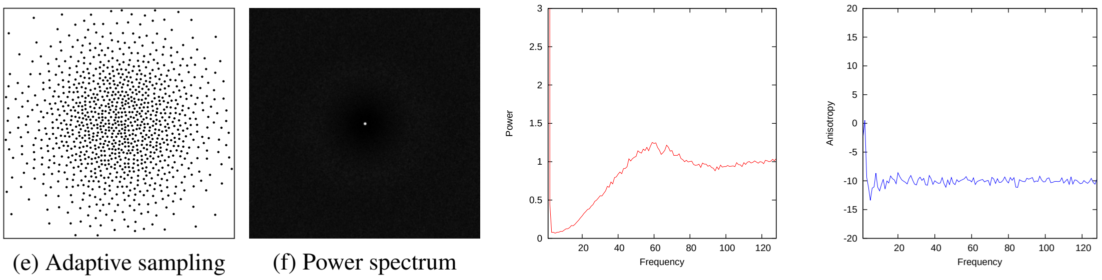
$$P_X(f) = |\mathcal{F}(f)|^2 = \left| \sum_{m=0}^{n-1} e^{-2\pi i (f \cdot x_m)} \right|^2 \tag{2.19}$$

can be computed. When averaging several periodograms, the distribution's power spectrum $\hat{P}(f)$ can be estimated, as proposed by Ulichney [Uli87] for the study of dither patterns. The estimated power spectra of distributions (see Figure 2.2b for an example) allow a very fast visual comparison and have since been used in almost every publication on blue-noise sampling. Ulichney also proposed the calculation of two radially averaged statistics from $\hat{P}(f)$, namely the radially averaged power spectrum or *radial mean* and the *anisotropy*. He also pointed out that the anisotropy – the estimation of the power spectrum's radial symmetry – is related to the number of periodograms used to estimate $\hat{P}(f)$. In the following, every power spectrum is estimated by ten periodograms. Thus, an anisotropy of -10 dB should be considered *background noise* and is – for the scope of this thesis – the reference value for low anisotropy. The radial mean gives the best indication of blue-noise properties and has been studied comprehensively by Lagae and Dutré [LD08] and Heck et al. [HSD13]. The reference radial mean for blue-noise sampling is computed from samples created by dart throwing (see Figure 2.2c). The peak of the DC component is followed by a low-energy annulus and a sharp transition corresponding to the inverse of the minimum radius r. Periodic annuli of higher, but decreasing energy corresponding to multiples of this frequency follow, until transitioning into noise.

However, the usual Fourier analysis does not provide useful results for non-uniform sample distributions, as seen in Figure 2.2f. Although the samples are well distributed locally, the power spectrum has little resemblance to the blue-noise spectrum. To overcome this, Wei and



(a) Uniform sampling (b) Power spectrum (c) Radial mean (d) Anisotropy



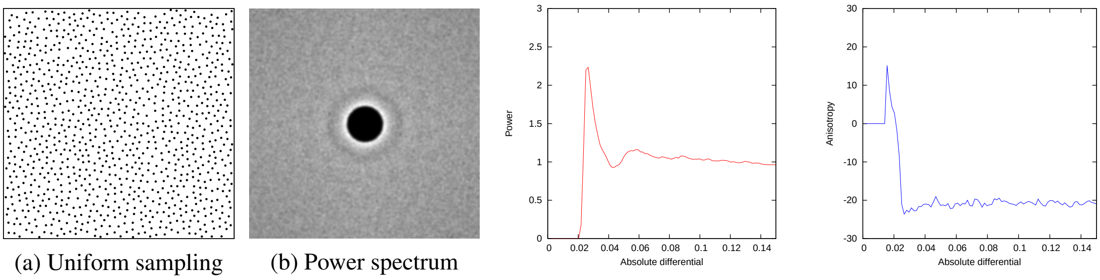
(e) Adaptive sampling (f) Power spectrum (g) Radial mean (h) Anisotropy

Figure 2.2: Poisson disk distribution (1024 samples). Fourier power spectrum estimated with 10 sample sets. Top row: Uniform sampling with dart throwing. Bottom row: Adaptive sampling with relaxation dart throwing using the Gaussian blob importance function.

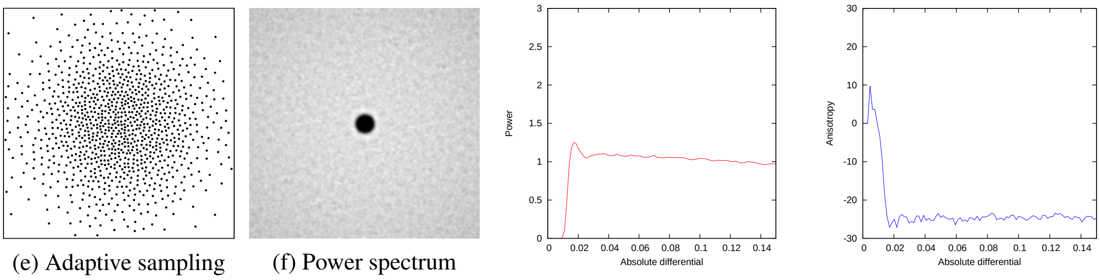
Wang [WW11] proposed the *differential domain analysis* (DDA) derived from the Fourier analysis. The DDA is based on the observation that Equation 2.19 can be reformulated to

$$P_X(f) = \frac{1}{n} \sum_{l,m=0}^{n-1} \cos\left(2\pi (f \cdot d_{l,m})\right), \tag{2.20}$$

where $d_{l,m} = x_l - x_m$ is the pairwise sample location *differential*. Thus, the transformation only depends on the distribution of differentials rather than the sample locations itself. For the complete derivation, further generalizations and extensions carried out by Wei and Wang, the reader is referred to the paper [WW11]. Analogous to the power spectrum, the two radially averaged measures *radial mean* and *anisotropy* can be calculated, although both statistics depend on the absolute differential, $|d_{l,m}|$, rather than the frequency. The DDA results are very similar to those of traditional Fourier analysis, as can be seen when comparing Figures 2.2 and 2.3. The blue-noise DDA power spectrum has a strong resemblance to a full solar eclipse with a disk of low energy surrounded by an annulus of high energy, corresponding to the minimum distance r, which transitions into noise. The radial mean shows the same peaks as in the Fourier power spectrum, the anisotropy also shows the same behavior. Following Wei and Wang, the anisotropy in the range smaller than r is ignored, as no samples fall into this region. It can be seen in Figure 2.3 that apart from the different minimum radii for uniform and adaptive sampling, the DDA power spectrum for both sample distributions is the same. Thus, DDA is a powerful tool for the



(a) Uniform sampling (b) Power spectrum (c) Radial mean (d) Anisotropy



(e) Adaptive sampling (f) Power spectrum (g) Radial mean (h) Anisotropy

Figure 2.3: Poisson disk distribution (1024 samples). DDA power spectrum estimated with 10 sample sets. Top row: Uniform sampling with dart throwing. Bottom row: Adaptive sampling with relaxation dart throwing using the Gaussian blob importance function.

analysis of sample distributions independent from their sample domain or importance function. As already mentioned above, Wei and Wang also proposed an extension of the spatial density ρ_X to non-uniform sampling by choosing d_X out of the transformed differentials $d_{l,m}$. An extensive set of tools for the analysis of sample distributions, including Fourier and differential domain analysis, has been made available online by Wei [Wei11].

CHAPTER 3

Forced Random Dithering

3.1 Construction

Dithering is a color-reduction technique that is used to display or print an image with arbitrarily many different colors on a device with a limited color palette. The most prominent application for dithering is the conversion of grayscale images to binary images for early monochrome computer monitors and printers. Instead of simply thresholding the intensity with a fixed value, in which case all halftone information is lost, the aim is to preserve this information in the density of fixed-sized black dots or pixels. Furthermore, their arrangement should be random in order to hide the inherent quantization error of the color reduction in noise. There exists a wide range of dithering algorithms, of which *ordered dithering* [Bay73] is particularly interesting. The decision whether to draw a dot or not is independent from all other dots, which is why ordered dithering can be parallelized very easily. To threshold each pixel with another value, a *threshold matrix* is created beforehand. As a result, the amount of regularity or randomness of the dot placement is only controlled by the threshold matrix. Besides completely random or regular matrices, matrices can for example be constructed to minimize the quantization error or to reduce repetitive patterns in the dithered result.

Forced Random Dithering proposed by Purgathofer et al. [PTG94] is a variant of ordered dithering that aims at a spatially uniform placement of dots without regularity artifacts. The creation of the threshold matrix $\mathcal{M}$ uses the principle of repulsion to control the placement of threshold values. To create a k-dimensional matrix of size $S_M \times \cdots \times S_M$, the values $0, \ldots, S_M^k - 1$ are inserted one by one into a discrete, k-dimensional force field. In this force field, all values that have already been inserted repulse new values according to a force-field function f. For the most even distribution of values in $\mathcal{M}$, the location at which the next value should be inserted is the global minimum of the force field. However, this deterministic choice could possibly lead to regular patterns, which is why a local minimum is chosen randomly instead of the global minimum. In the proposed algorithm, this is done by randomly selecting half of all free locations of the force field and choosing the location with the minimal accumulated repulsion out of these. If more or less than half of the locations are considered, the spectral properties of the resulting matrices slightly differ, which is discussed in Section 3.2.

The design of the force-field function f follows from the requirements formulated by Purgathofer et al. For isotropic images, f should be radially symmetric, and to avoid clumping of values, f should penalize closeness. Thus, as a function of the distance

$$r = \sqrt{(x_{1,d_1} - x_{2,d_1})^2 + \cdots + (x_{1,d_k} - x_{2,d_k})^2} \tag{3.1}$$

of any location x_1 to an already set location x_2 in the force field, the repulsion is expressed as

$$f(r) = \exp\left(-\left(\frac{r}{s}\right)^p\right). \tag{3.2}$$

p and s are parameters to control the steepness and deviation of f. For both values, Purgathofer et al. suggest to use $1/2$, which leads to the force-field function

$$f(r) = \exp\left(-\sqrt{2r}\right), \tag{3.3}$$

which is used for all dither matrices throughout this thesis.

Pseudo-code for the generation of a Forced Random Dithering matrix is provided in Listing 3.1. The random selection of half of the free locations in $\mathcal{M}$ in Lines 6 and 7, the search for the location with the minimum force-field value in this subset in Lines 10 to 12 as well as the update of the force field in Lines 14 to 17 can be parallelized.

```
 1 freeLocations = $\{x | x \in [0, S_M - 1]^k\}$;
 2 forceField = new Tensor($S_M \times \cdots \times S_M$);
 3 $\mathcal{M}$ = new Tensor($S_M \times \cdots \times S_M$);
 4
 5 for (ditherValue = 0; ditherValue < $S_M^k$; ditherValue++) {
 6   P = randomPermutation(freeLocations);
 7   halfP = takeFirstHalfSubset(P);
 8   minimum = $\infty$; minimumLocation;
 9
10   foreach (location in halfP)
11     if (forceField[location] < minimum)
12       minimumLocation = location;
13
14   foreach (cell in forceField) {
15     $r$ = toroidalMinimumDistance(cell.location, minimumLocation);
16     forceField[cell.location] += $f(r)$;
17   }
18
19   freeLocations.remove(minimumLocation);
20   $\mathcal{M}$[minimumLocation] = ditherValue;
21 }
```

Listing 3.1: Generation of Forced Random Dithering matrix

36	160	58	181	84	175	46	215	144	198	8	138	238	91	114	227
228	98	189	221	18	156	111	194	60	32	126	80	27	193	133	247
140	112	44	119	163	105	3	244	180	42	155	212	101	218	67	0
94	31	174	201	25	72	224	75	141	235	170	54	16	231	48	236
143	251	63	137	90	254	50	135	200	29	113	74	162	150	78	178
168	9	106	151	15	208	95	7	187	103	134	253	190	117	24	86
223	56	211	184	226	166	197	87	59	241	13	202	83	5	249	216
65	127	30	132	109	52	179	123	159	34	182	69	165	145	45	120
173	148	76	239	35	100	252	21	203	43	214	125	53	157	206	28
255	12	186	229	158	1	217	85	96	169	61	2	191	88	219	71
154	97	124	49	68	220	115	246	147	177	130	240	136	38	234	116
33	176	23	192	139	39	171	55	92	110	19	199	99	195	17	107
47	232	81	243	204	102	26	233	10	209	82	51	222	77	161	245
131	164	6	129	73	210	79	128	167	248	142	188	149	4	57	207
89	152	121	196	146	14	237	205	37	153	66	40	122	242	104	183
70	11	250	41	225	64	118	93	22	213	185	108	172	62	20	230

Table 3.1: A 16×16 Forced Random Dithering matrix. The first elements are highlighted to visualize their even distribution within the matrix.

An exemplary Forced Random Dithering matrix of size 16×16 is shown in Table 3.1. The first ten values that have been inserted into the matrix have been highlighted in yellow to emphasize their even distribution. The next ten values have been highlighted in orange to emphasize that additional values are added in the gaps between the yellow values. As a result, both the first ten and the first twenty values are distributed evenly within $\mathcal{M}$. This is an important observation that allows the matrix to be used for approximate blue-noise dithering, because independent from the intensity levels of an image, the pixels of the dithered result are distributed spatially uniformly. From the highlighted values it can also be seen that $\mathcal{M}$ is toroidal, meaning that it can be repeated seamlessly, which reduces the regular artifacts observable with other threshold matrices. The toroidal topology of $\mathcal{M}$ is achieved by simply joining the left and right as well as the top and bottom edges of the force field.

It is obvious that the quality of the distribution of dither values in $\mathcal{M}$ increases with the matrix size S_M, as the discrete location of each local minimum in the force field can be determined more precisely. In practice, the dither matrix has to be much larger than 16×16 pixels.

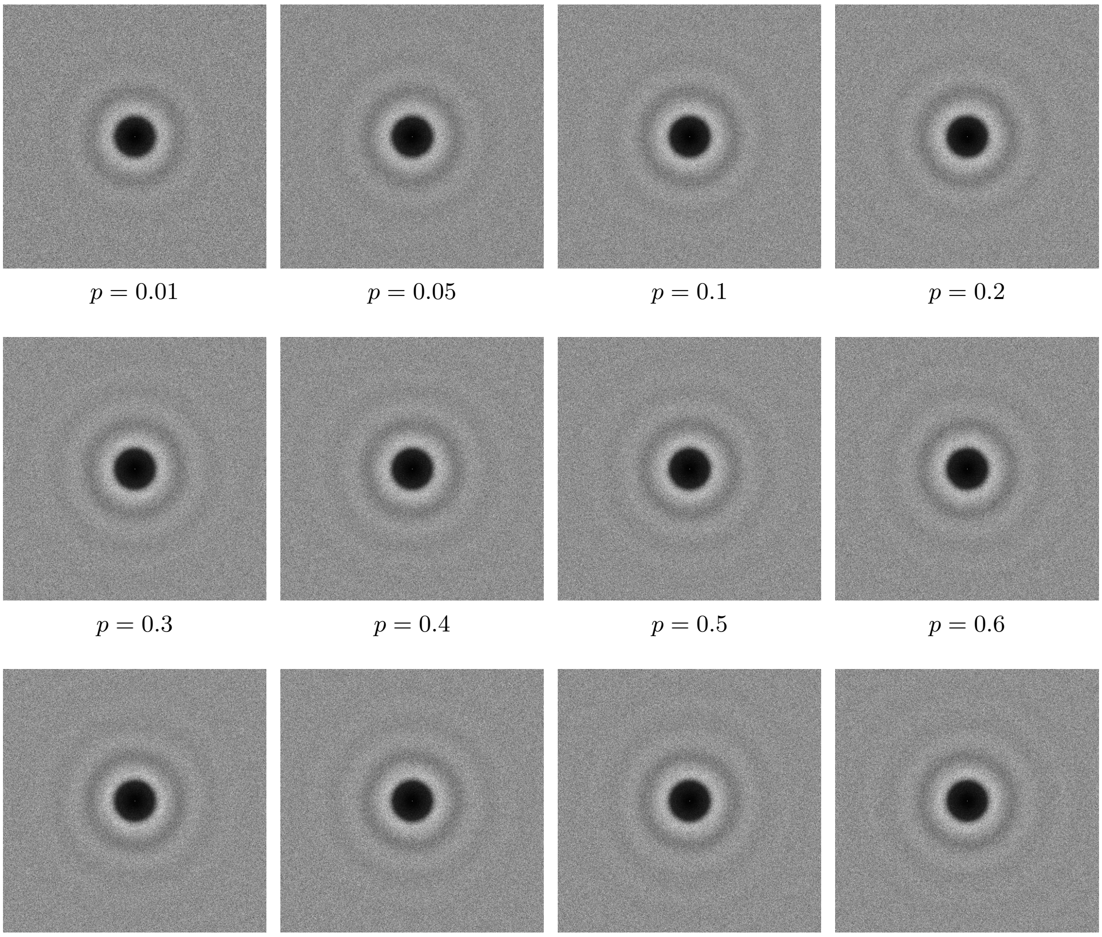
A qualitative comparison of different dither matrix sizes is given in Section 4.7. At the time, Purgathofer et al. suggested a size of 300×300 pixels as a compromise between image quality and the computation time for the matrix. As the influence of all inserted values on every empty matrix location has to be evaluated for every new value to be inserted, the creation of the matrix has a runtime complexity of $\mathcal{O}(S_M^{2k})$. The calculation can be optimized by storing the influence of all inserted values, because it will not change after insertion. Further, in an isotropic domain, the force function can be precomputed for each location, such that the values only have to be added to the stored sum. Still, the generation of larger matrices remains expensive and is not feasible on the CPU. Even with all parallelizable operations done on the GPU, the matrix generation is far too expensive for online generation. On a test system using an NVIDIA GeForce GTX 680 video card, the generation of the 2048×2048 dither matrix used throughout this thesis took over four hours.

3.2 Randomized Minimum Selection

Purgathofer et al. [PTG94] proposed to insert each dither value at a local rather than the global minimum of the force field to avoid deterministic behavior. This is why in Listing 3.1, the minimum accumulated force is searched in only half of all free matrix locations. The selection of which locations are compared for the minimum force-field value is randomized. While this algorithm already produces the desired Forced Random Dithering matrices, it is interesting to know how changing this selection of locations influences the matrix and its spectral properties. For both dithering and sampling, the power spectrum of locations of matrix elements that remain after a constant threshold comparison should have approximate blue-noise properties. This means that the locations of the values in the matrix should neither be too random nor too regular. It can be assumed that the placement of values in $\mathcal{M}$ becomes more regular if more than half of all free locations are compared. It will be fully deterministic if all locations are considered. Likewise, the placement becomes more random if less than half of all locations are compared. If only one location is chosen randomly in each iteration, $\mathcal{M}$ will be a completely random threshold matrix.

A parameter $p \in (0, 1]$ is introduced to control the percentage of free locations that are considered for the calculation of a force-field minimum. For each location in $\mathcal{M}$, a uniform random number $r \in [0, 1)$ can be generated and compared to p. The selection of a candidate for the placement of the next dither value is then determined by whether $r < p$. If, for example, $p = 0.5$, on average half of all free locations in $\mathcal{M}$ are randomly selected, which corresponds to the selection proposed by Purgathofer et al.

In Figure 3.1, the power spectra of results of Forced Random Dithering are presented for matrices generated with different values of p. Each power spectrum has been estimated by 10 matrices of size 512×512. The input image has a size of 512×512, such that the dither matrix covers it completely, and a constant intensity of $97.5\ \%$. As shown already by Purgathofer et al. for $p = 0.5$, the power spectra of the results show very distinct blue-noise properties. As expected, the placement of values in $\mathcal{M}$ becomes more random as p decreases. For $p \leq 0.05$, the outer annulus of high energy is almost completely lost in uncorrelated noise. More surprising are the blue-noise properties of matrices for $p > 0.5$. Even if all locations are considered for the determination of the minimum, i.e., if each dither value is placed at the location of the global minimum of the force field, there are no noticeable regular patterns or other anisotropic artifacts. This suggests that the random selection of only a subset of all locations is not necessary and can be omitted in an implementation. However, all instead of only half of all free locations have to be compared then, which can be more or less expensive than the random selection, depending on the cost of the random number generation. If a very simple pseudo-random number generator such as the LCG variant proposed by Park and Miller [PM88] is used, the random selection is considerably faster. In a test implementation of Forced Random Dithering, the parallelized random selection was about 20 % faster than comparing all free locations.



$p = 0.01$ $p = 0.05$ $p = 0.1$ $p = 0.2$

$p = 0.3$ $p = 0.4$ $p = 0.5$ $p = 0.6$

$p = 0.7$ $p = 0.8$ $p = 0.9$ $p = 1.0$

Figure 3.1: Estimated power spectra of Forced Random Dithering results with matrices generated with different values of p

CHAPTER 4

Forced Random Sampling

4.1 Overview

Forced Random Sampling is the application of Forced Random Dithering for k-dimensional adaptive sampling. Instead of halftoning an image, an importance function $\mathcal{I}$ defined in the sample domain $\Omega \subseteq [0, 1)^k$ is considered. Using halftoning methods for sampling has already been proposed by Mitchell [Mit87]. However, Mitchell's point diffusion algorithm, which is based on error diffusion, cannot be parallelized reasonably. Methods based on ordered dithering such as Forced Random Dithering, in contrast, can be parallelized very well, because thresholding of each element is independent from all others. Additionally, distribution properties such as randomness can be included in the dither matrix itself and do not need to be computed at runtime. Forced Random Dithering even includes spatial uniformity of elements with similar values. Therefore, sampling based on ordered dithering is very similar to tile-based Poisson disk sampling reviewed in Chapter 2. The main advantage of these methods is the separation of sample generation and sampling into an expensive offline and an inexpensive online computation step. Although very fast, using tiles with precomputed Poisson disk samples is not very flexible and difficult to use for adaptive sampling. When using a dither matrix instead of actual samples for tiling, the sample tiles are basically created at runtime by thresholding. The dither matrix $\mathcal{M}$ can be seen as a large set of possible sample distributions, including adaptive ones. One set of samples X is drawn from $\mathcal{M}$ by thresholding, where the importance function $\mathcal{I}$ controls the density of samples. The entire sampling algorithm is explained in the following sections in detail, actual implementations of FRS are provided in the next chapter.

4.2 Uniform Sampling

To introduce the idea of FRS, only the special case of uniform sampling is considered, meaning that $\mathcal{I} = 1$ over the entire sample domain. Let $\mathcal{M}$ be of size $S_M \times \cdots \times S_M$ and let $p_{\mathcal{M}} \in [0, S_M - 1]^k$ be the index of an arbitrary matrix element with the precomputed dither value $\mathcal{M}(p_{\mathcal{M}})$. The dither values stem from the order of insertion at matrix creation and range from 0 to $S_M^k - 1$. This means that comparing $\mathcal{M}$ to a constant threshold function

$$\mathrm{T} = n, n \in \mathbb{N}, \tag{4.1}$$

will leave the first n elements that have been inserted. The index $p_{\mathcal{M}}$ of each of these elements corresponds to a point $p_{\mathcal{I}} = p_{\mathcal{M}}/S_M$ in the sample domain Ω, which can be used for sampling. By the design of the matrix, these points are distributed spatially uniformly within Ω.

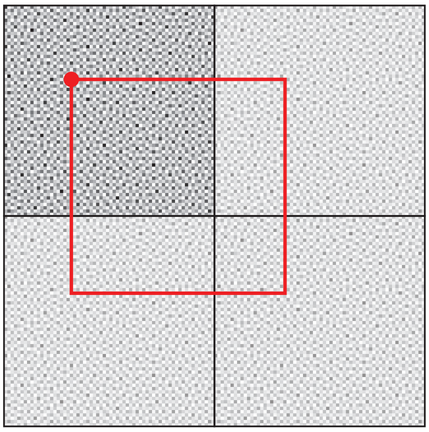


Figure 4.1: Dither matrix window. The dither matrix is repeated to tile an infinite plane. A window is cut out at a random offset.

From the dither matrix $\mathcal{M}$, a maximum of S_M^k samples can be generated with the described thresholding, which is one for every element of the matrix. To generate more than S_M^k samples without using a larger matrix, a tiling of $\mathcal{M}$ can be used. Since $\mathcal{M}$ is toroidal, it can be repeated seamlessly. From the theoretically infinite tiling of $\mathcal{M}$, a finite section is used for thresholding. This section, in the following called a *window* of $\mathcal{M}$, has to be large enough to provide the desired number of samples, but should be as small as possible to minimize the computational effort of thresholding. Figure 4.1 illustrates such a window. Here, a 2×2 tiling of $\mathcal{M}$ of size 64×64 is shown. The values of the matrix are actually in the range $[0, 64^2 - 1]$ and have been scaled to $[0, 255]$ for display. The red circle marks the origin of the window, which has been offset relative to the origin of $\mathcal{M}$. This offset Δ_W can be used to vary the sample distribution for the same $\mathcal{I}$ and $\mathcal{M}$ over several sampling runs. It does not influence thresholding itself, as it is only used to select a different portion of $\mathcal{M}$. Its value can therefore be arbitrary and is chosen randomly from $[0, S_M - 1]^k$.

The size S_W of the window depends on multiple parameters and determines the quality and performance of sampling. The smallest possible size is $S_W = \sqrt[k]{n} \times \cdots \times \sqrt[k]{n}$. In this case, every element of the window would have to pass thresholding, which would result in a completely regular sample placement. This degeneration to regular sampling has to be prevented by enforcing an average distance between samples, which is achieved with the *sparsity* σ. It introduces a constraint that on average, only one out of σ^k window elements passes thresholding. The choice of σ influences the quality of the resulting samples. If it is too low, the samples exhibit strong regularity artifacts. If it is too high, the window becomes large and too many elements are compared for thresholding, which reduces the runtime performance. σ should therefore be as small as possible, but as large as necessary to avoid artifacts. To find the ideal value, a spectral analysis of samples for several values of σ is given in Section 4.7. With σ, the optimal size S_W of the window can be calculated, which is explained in detail in Section 4.4.

Having a window larger than the dither matrix itself slightly changes thresholding because the same dither values occur multiple times in the window. Considering only the case of a two-dimensional importance, $\sigma = 1$ and a window of size $S_W = S_{W,1} \times S_{W,2} = S_M \times S_M$, then the first n elements of the dither matrix would pass thresholding. If, however, $S_{W,1} = S_{W,2} = 2S_M$, each value $\in [0, S_M^2 - 1]$ of the dither matrix would appear four times in the window. Thus, the threshold needs to be scaled to a quarter of its original value to account for that or $4n$ samples would be created. Similarly, if $S_{W,1} = S_{W,2} = S_M/2$, the window would contain only a quarter of all possible dither values, which is why the threshold would have to be multiplied by four. If the size of the window does not match the size of $\mathcal{M}$, the threshold function T of Equation 4.1 therefore has to be extended to

$$\mathrm{T} = n \cdot \frac{S_M^k}{\prod_{i=1}^k S_{W,i}}. \tag{4.2}$$

4.3 Adaptive Sampling

Until now, only uniform sampling has been discussed. In this case it suffices to threshold the window with T as shown in Equation 4.2 to get n samples. But if $\mathcal{I}$ is a non-uniform function, more elements of the window should remain in regions of higher importance than in regions of low importance such that the density of the resulting samples approximates $\mathcal{I}$. Thus, in the general case, the threshold of each window element p_W depends on the importance $\mathcal{I}(p_\mathcal{I})$ at the corresponding point $p_\mathcal{I} = p_W / S_W$ in the sample domain, i.e.,

$$\mathrm{T}(p_\mathcal{I}) = \mathcal{I}(p_\mathcal{I}) \cdot \tau, \tag{4.3}$$

where τ is a global, constant scaling factor. If $\tau = n \cdot S_M^k / \prod_{i=1}^k S_{W,i}$ as in the uniform case, less than n samples would be created if $\mathcal{I}(p_\mathcal{I}) < 1$ for any $p_\mathcal{I}$. To create approximately n samples, τ needs to account for the average importance

$$\mu_\mathcal{I} = \int_\Omega \mathcal{I}(\omega_1, \ldots, \omega_k) d\omega_1 \ldots d\omega_k, \tag{4.4}$$

leading to a scaling factor

$$\tau_\mathcal{I} = \frac{n}{\mu_\mathcal{I}} \cdot \frac{S_M^k}{\prod_{i=1}^k S_{W,i}}. \tag{4.5}$$

The complete threshold inequation of FRS, $\mathcal{M}(p_\mathcal{M}) < \mathcal{I}(p_\mathcal{I}) \cdot \tau_\mathcal{I}$, is illustrated for $k = 2$ and $n = 32$ in Figure 4.2 with a Gaussian blob importance function. The dither value of each element p_W of the window (orange) is compared to the product of the corresponding importance $\mathcal{I}$ at point $p_\mathcal{I}$ (green) and the scaling factor $\tau_\mathcal{I}$. Approximately n elements pass this comparison, and their locations (blue) are used for sampling. The resulting samples are adapted to the importance function while still being well distributed locally.

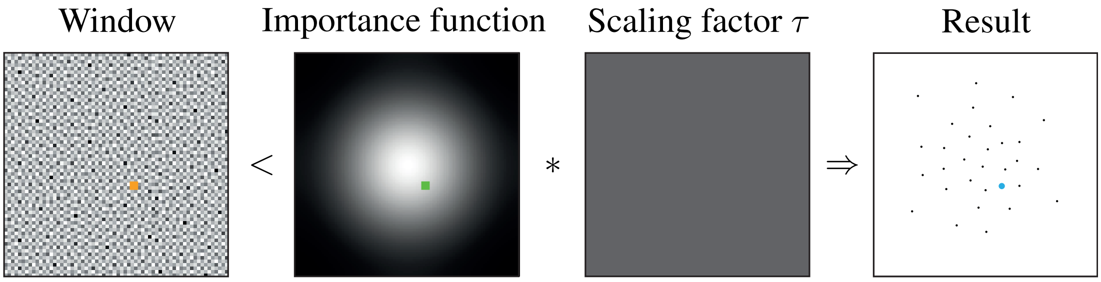


Figure 4.2: Illustration of the thresholding inequation of Forced Random Sampling

4.4 Window Calculation

The left side of the equation depicted in Figure 4.2 is the window itself. The window is the section of an infinite tiling of $\mathcal{M}$ that is large enough to provide the desired number of samples n through thresholding. Its size $S_W = S_{W,1} \times \cdots \times S_{W,k}$ is a result of the sparsity constraint introduced in Section 4.2, which ensures that on average, only one out of σ^k window elements passes thresholding, and thus prevents the sample placement from being too regular. For a uniform importance $\mathcal{I} = 1$, the sparsity constraint can be expressed as

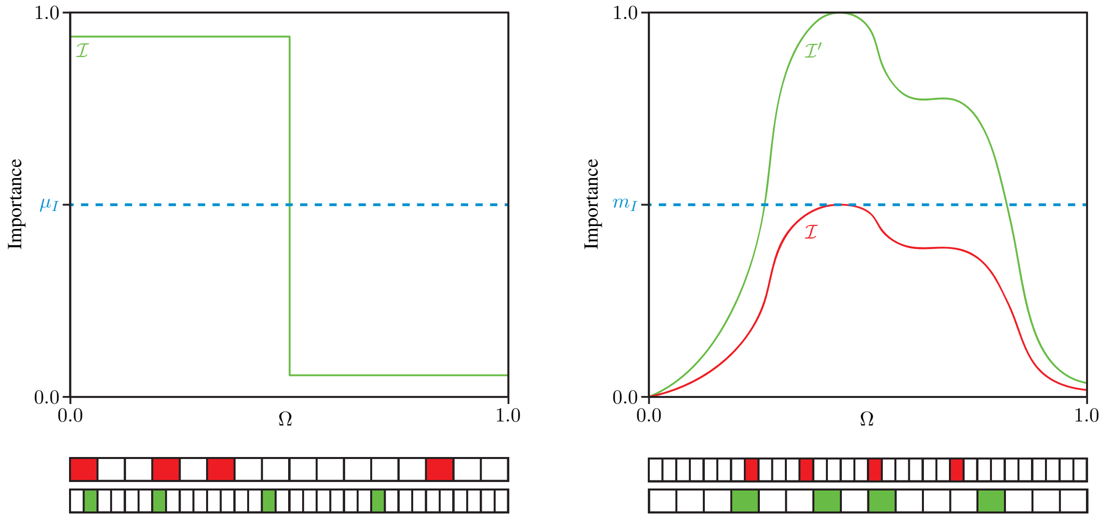
$$n\sigma^k = \prod_{i=1}^{k} S_{W,i}. \tag{4.6}$$

This means that a dither matrix window has to have a size of at least $S_{W,1} \times \cdots \times S_{W,k}$ to provide n samples for a given σ. For non-uniform $\mathcal{I}$, the sample density should vary according to the local importance $\mathcal{I}(p_{\mathcal{I}})$. If $\mathcal{I}(p_{\mathcal{I}}) < 1$, less dither values than one out of σ^k should pass thresholding in these regions. Instead of having a probability of $1/\sigma^k$ to pass thresholding like in the uniform case, the probability that window element p_W passes thresholding should be $\mathcal{I}(p_{\mathcal{I}})/\sigma^k$. From this follows that with the $\prod_{i=1}^{k} S_{W,i}$ window elements for the entire sample domain, only approximately

$$\frac{\prod_{i=1}^{k} S_{W,i}}{\sigma^k} \int_{\Omega} \mathcal{I}(\omega_1, \ldots, \omega_k) d\omega_1 \ldots d\omega_k = n\mu_{\mathcal{I}} \tag{4.7}$$

samples could be created, which is less than desired. To still provide n samples in total, more samples would need to be placed in the more important regions, which would violate the sparsity constraint. Thus, the minimum window size that follows from Equation 4.6 is not sufficient in this case. Instead, the window size has to be scaled by $1/\mu_{\mathcal{I}}$ beforehand to compensate for the low importance in some regions of $\mathcal{I}$.

This problem is illustrated in Figure 4.3a for one dimension. Here, $\mathcal{I}$ (green) is an importance function with half the values close to 1 and the other half of the values close to 0, such that the average importance $\mu_{\mathcal{I}} = 0.5$. Let $n = 4$ and $\sigma = 4$, then the window size according to Equation 4.6 should be $S_{W,1} = 16$. Due to the high importance of the lower values of $\mathcal{I}$, all four



(a) Violation of the sparsity constraint (red samples) in regions of high importance can be avoided by scaling the window size by $1/\mu_{\mathcal{I}}$ (green samples)

(b) A too large window size (red samples) can be avoided by scaling the original importance function (red) by $1/m_{\mathcal{I}}$ (green samples)

Figure 4.3: Influence of average and maximum importance on the window size

samples should be placed there. If all four samples are obtained from the first eight dither values (red samples), the sparsity constraint is violated and the samples will exhibit regularity artifacts. If the window size is scaled to $16/\mu_{\mathcal{I}} = 32$, there are 16 dither values for the lower values of $\mathcal{I}$, which can be thresholded to obtain the four samples (green samples), which satisfies the sparsity constraint.

However, this assumes that the supremum of $\mathcal{I}$,

$$m_{\mathcal{I}} = \sup_{(\omega_1,\dots,\omega_k)\in\Omega} \mathcal{I}(\omega_1,\dots,\omega_k), \tag{4.8}$$

is 1, which is not necessarily true. An exemplary importance function for the one-dimensional case is illustrated in Figure 4.3b (red) with a maximum importance of 0.5. Because of the low average value, the window size would be scaled up to 32 or even 64. This is to ensure that in regions of $\mathcal{I}(p_{\mathcal{I}}) = 1$, only one out of 4 dither values passes thresholding, although there are no such regions in $\mathcal{I}$. Thresholding would thus compare too many dither values for the specified constraints (red samples). It suffices to ensure the sparsity constraint for the maximum importance, even if it is smaller than 1. This is achieved by scaling the average importance by $1/m_{\mathcal{I}}$, which can be seen as a transformation $[0, m_{\mathcal{I}}] \mapsto [0, 1]$ of importance values, such that the transformed $\mathcal{I}'$ has a maximum $m'_{\mathcal{I}} = 1$ (green). It then becomes apparent that it is sufficient to threshold only 16 dither values (green samples). Including both average and maximum importance into Equation 4.6, the sparsity constraint for sampling with a non-uniform importance function can be expressed as

$$n\sigma^k \cdot \frac{m_{\mathcal{I}}}{\mu_{\mathcal{I}}} = \prod_{i=1}^{k} S_{W,i}. \tag{4.9}$$

As the window is thresholded with the importance function, their aspect ratios should match. In the following, the extents of $\mathcal{I}$ are denoted with $S_{I,i}$ for $i = 1, \dots, k$, which are the extents of the domain of definition Ω in the continuous case and the pixel size of the importance map in the discrete case. In both cases, the aspect ratios are the same. Without loss of generality, it is assumed that $S_{I,1}$ is the smallest extent of the importance function. Then $S_{W,1}$ should be also the smallest extent of the window. With the aspect ratios

$$a_i = S_{I,i}/S_{I,1} = S_{W,i}/S_{W,1} \geq 1, \tag{4.10}$$

all k extents of the window can be expressed as a function of $S_{W,1}$. With these, Equation 4.9 can be rewritten as

$$n\sigma^k \cdot \frac{m_{\mathcal{I}}}{\mu_{\mathcal{I}}} = S_{W,1}^k \prod_{i=1}^{k} a_i, \tag{4.11}$$

from which follows

$$S_{W,1} = \sqrt[k]{\frac{n m_{\mathcal{I}}}{\mu_{\mathcal{I}} \prod_{i=1}^{k} a_i}}\,\sigma, \quad S_{W,i} = a_i S_{W,1}. \tag{4.12}$$

This means that the window has to have a size of at least $S_{W,1} \times \cdots \times S_{W,k}$ to provide enough dither matrix elements for thresholding $\mathcal{I}$ with sparsity σ.

4.5 Sample Placement

Once the size of the window has been determined, every element p_W of the window can be thresholded with $\mathrm{T}(p_{\mathcal{I}})$. First, its coordinates relative to the origin of the dither matrix,

$$p_{\mathcal{M}} = (\Delta_W + p_W) \bmod S_M, \tag{4.13}$$

are calculated, which are used to retrieve the dither value $\mathcal{M}(p_{\mathcal{M}})$ at that point. The dither value is compared to the threshold function and a sample $x \in \Omega$ is created if

$$\mathcal{M}(p_{\mathcal{M}}) < \mathrm{T}(p_{\mathcal{I}}). \tag{4.14}$$

The location of the sample is

$$x = p_{\mathcal{I}} S_I + \frac{\Delta_p}{S_W}, \tag{4.15}$$

where $\Delta_p \in [0,1)^k$ is an offset inside the window element. In the following, $\Delta_p = (0.5, \dots, 0.5)$, the center of p_W, is used for FRS, while a random offset is used for jittered FRS.

Since the dither matrix is repeated in the case of any $S_{W,i} > S_M$, repetitions in the arrangement of the samples can occur, depending on $\mathcal{I}$. In the case of uniform $\mathcal{I}$, this is certain.

Repetitions cannot be avoided completely in general, but they can be reduced by either applying random jittering to the samples or increasing the size of the dither matrix. In the following, all qualitative tests will be performed on both jittered and unjittered samples. Also, if not denoted otherwise, a dither matrix of size $S_M = 2048$ is used. This size is a fair compromise between the expensive matrix creation and the number of unique samples that can be created. An analysis of the sample quality depending on the dither matrix size is given in Section 4.7.

For an actual implementation of FRS, thresholding can be parallelized because Equation 4.14 does not depend on any intermediate results other than the scaling factor $\tau_{\mathcal{I}}$ and the window size. Similar to other sampling techniques, it is not guaranteed that exactly n samples are retrieved, but the deviation is very small.

4.6 Progressive Sampling

Forced Random Sampling can be extended to progressive sampling, meaning that new samples can be created and added to an existing sample set such that the distribution properties of the sample set are not impaired. This is possible due to the uniform distribution of similar dither values within the dither matrix, as illustrated in Table 3.1 in Chapter 3. Both the first n and the first $n+1$ dither values are distributed uniformly in $\mathcal{M}$ and thus in the window, such that adding an additional sample to a sample set of size n leads to the same well-distributed sample set as would creating $n+1$ samples at once. It has to be ensured that the window is large enough to provide $n+1$ samples without violating the sparsity constraint. This is why the window size (Equation 4.12) has to be calculated with respect to the maximum number of samples that might be required at some point. By adjusting n in Equation 4.5 to calculate $\tau_{\mathcal{I}}$, the actual number of samples to pass thresholding can be controlled.

However, the new sample cannot be created directly without thresholding the entire window again. Thresholding with the modified scaling factor creates the entire sample set of size $n+1$. Rather than using a single $\tau_{\mathcal{I}}$ for thresholding, a range $[\tau_{\mathcal{I},old}, \tau_{\mathcal{I},new})$ has to be used, where $\tau_{\mathcal{I},old}$ is the scaling factor used to create the first n samples and $\tau_{\mathcal{I},new}$ the adjusted one. Then, the threshold comparison in Equation 4.14 can be extended to

$$\tau_{\mathcal{I},old} \cdot \mathcal{I}(p_{\mathcal{I}}) \leq \mathcal{M}(p_{\mathcal{M}}) < \tau_{\mathcal{I},new} \cdot \mathcal{I}(p_{\mathcal{I}}), \tag{4.16}$$

such that only new samples pass.

4.7 Parameter Choices

Dither Matrix Size S_M

The size S_M of the dither matrix does not influence the runtime complexity of FRS, but the quality of the samples. As the dither matrix is a regular grid, samples obtained from it always exhibit regularity to some extent because of their discrete element positions. Furthermore, only a finite number of unique samples can be obtained from one dither matrix. Using Equation 4.6 for $k = 2$, this number is

$$n_{max} = \frac{S_M^2}{\sigma^2}. \tag{4.17}$$

For the parameters used in this thesis ($S_M = 2048$, $\sigma = 8$), $n_{max} = 65536$. Further results are given in Table 4.1. If the desired number of samples exceeds n_{max}, the dither matrix is repeated inside the window, which leads to repetitions in the sample set. Thus, increasing the matrix size reduces the amount of repetitive patterns. However, the dither matrix has to be stored in the memory for thresholding, which at least for GPU implementations is a valid argument for using a matrix of smaller size. Also, the matrix creation is very expensive and seems not feasible for $S_M > 4096$ on current hardware.

	128	256	512	1024	2048	4096	8192
4	1024	4096	16384	65536	262144	1048576	4194304
8	256	1024	4096	16384	65536	262144	1048576
16	64	256	1024	4096	16384	65536	262144

Table 4.1: Number of unique samples that can be obtained from one dither matrix depending on the matrix size (columns) and the sparsity (rows)

It is important to use a matrix that leads to samples without obvious artifacts or patterns, but it has to be computable with reasonable effort. The former has been evaluated with a differential domain analysis (DDA) [WW11] of sample sets for several sizes S_M. The results are shown in Figure 4.4. Each power spectrum has been estimated by 10 sets of 1024 samples. In the first row, the power spectrum of dart throwing is given as a reference. For small matrix sizes, the power spectra of FRS exhibit strong regularities, which is caused by the finite number of possible distances between any two elements of the dither matrix. Since the DDA only depends on the distribution of the differentials instead of the sample locations, this shortcoming of FRS is very apparent in the power spectra. It can be avoided by using a random sample offset Δ_p in Equation 4.15 for jittering. Jittering adds variation to the differentials and leads to a more random distribution. Still, the results of $S_M \leq 256$ strongly deviate from the reference and seem too random. From a matrix size of 512 on, the results of FRS closely match those of jittered FRS and are also close to the reference. Therefore, the matrix size should be chosen such that the desired number of samples can be created with it according to Table 4.1, but with a minimum of $S_M = 512$.

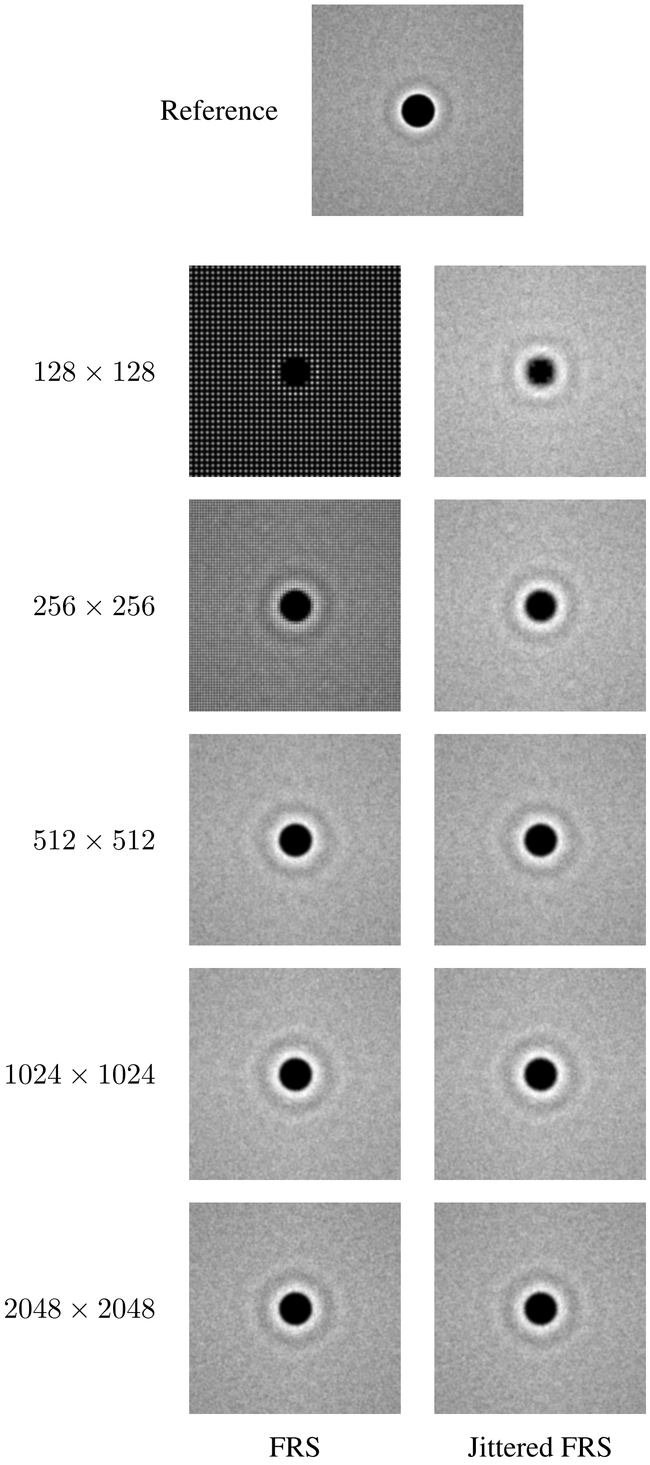
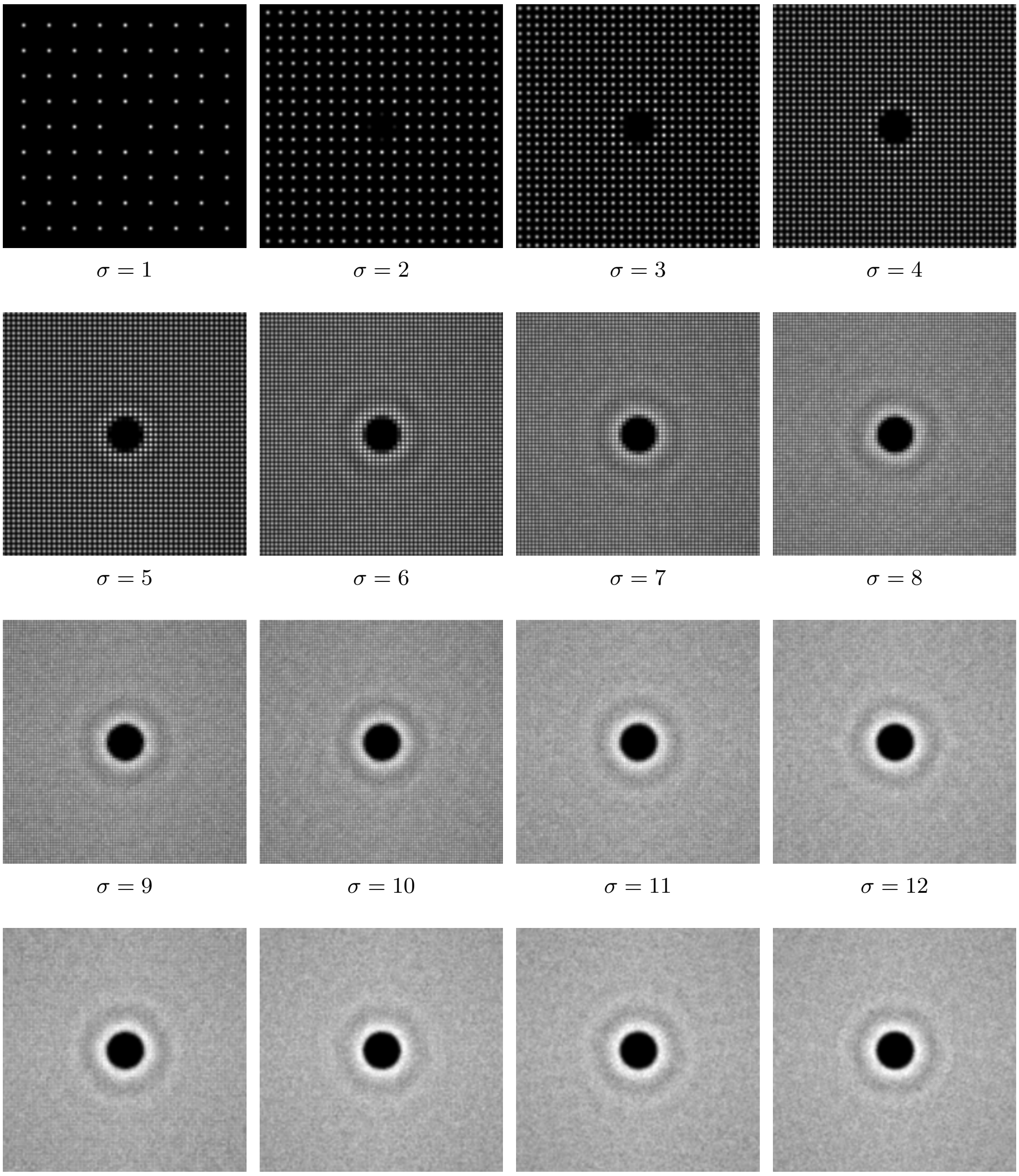


Figure 4.4: Estimated DDA power spectra of FRS and jittered FRS for different dither matrix sizes

Sparsity σ

The sparsity σ has been introduced to avoid that the window elements which pass thresholding are too close to each other. In the worst case, every element could have a value smaller than the threshold, in which case FRS would degenerate to regular sampling, which equals $\sigma = 1$. σ scales the threshold such that on average, only one out of σ^k window elements passes thresholding. Instead of using the whole range $[0, S_M^k - 1]$ of dither values for thresholding, only the range $[0, S_M^k/\sigma^k - 1]$ is used, which is the same as thresholding a *sparse* dither matrix with only the first $S_M^k/\sigma^k - 1$ elements set. In Table 4.1, the maximum number of samples obtainable from a dither matrix has been given dependent on S_M and σ. Assuming a fixed S_M, increasing σ decreases this number, which means that the window has to increase, too, to obtain the desired number of samples. So, on the one hand, σ preserves the even distribution of samples, but on the other hand, it increases the window size, which can lead to obvious repetitive patterns in the sample set for too small S_M as well as increase the computational effort of thresholding. This is why σ should just be large enough to remove obvious regular sample arrangements.

As these regular arrangements lead to a finite number of possible distances between any two samples, again the DDA is well suited to visualize them. In Figure 4.5, the power spectra of two-dimensional FRS, each estimated by 10 sets of 1024 uniform samples, are shown for different values of σ and $S_M = 512$. Figure 4.6 shows the corresponding results of jittered FRS. Unsurprisingly, the power spectra for $\sigma = 1$ equal those of regular sampling resp. jittered grid sampling. Especially in the case of unjittered FRS, it can be seen how the regularity of the samples decreases as σ increases. However, it has to be noted that even in the case of $\sigma = 16$ or a larger value, the sample positions are always discrete and therefore exhibit regularities. They are just lost in the finite resolution of the plot of the power spectrum. Therefore, an ideal σ cannot be found for FRS. Jittering, in contrast, adds randomness to the sample positions and thus variation to the differential distribution, which is why the power spectrum of jittered FRS looks acceptable for smaller σ as compared to FRS. Although the high-energy annulus in the power spectrum is a bit frayed, $\sigma = 8$ seems like a fair compromise between quality and computational effort and is used for the results throughout this thesis.



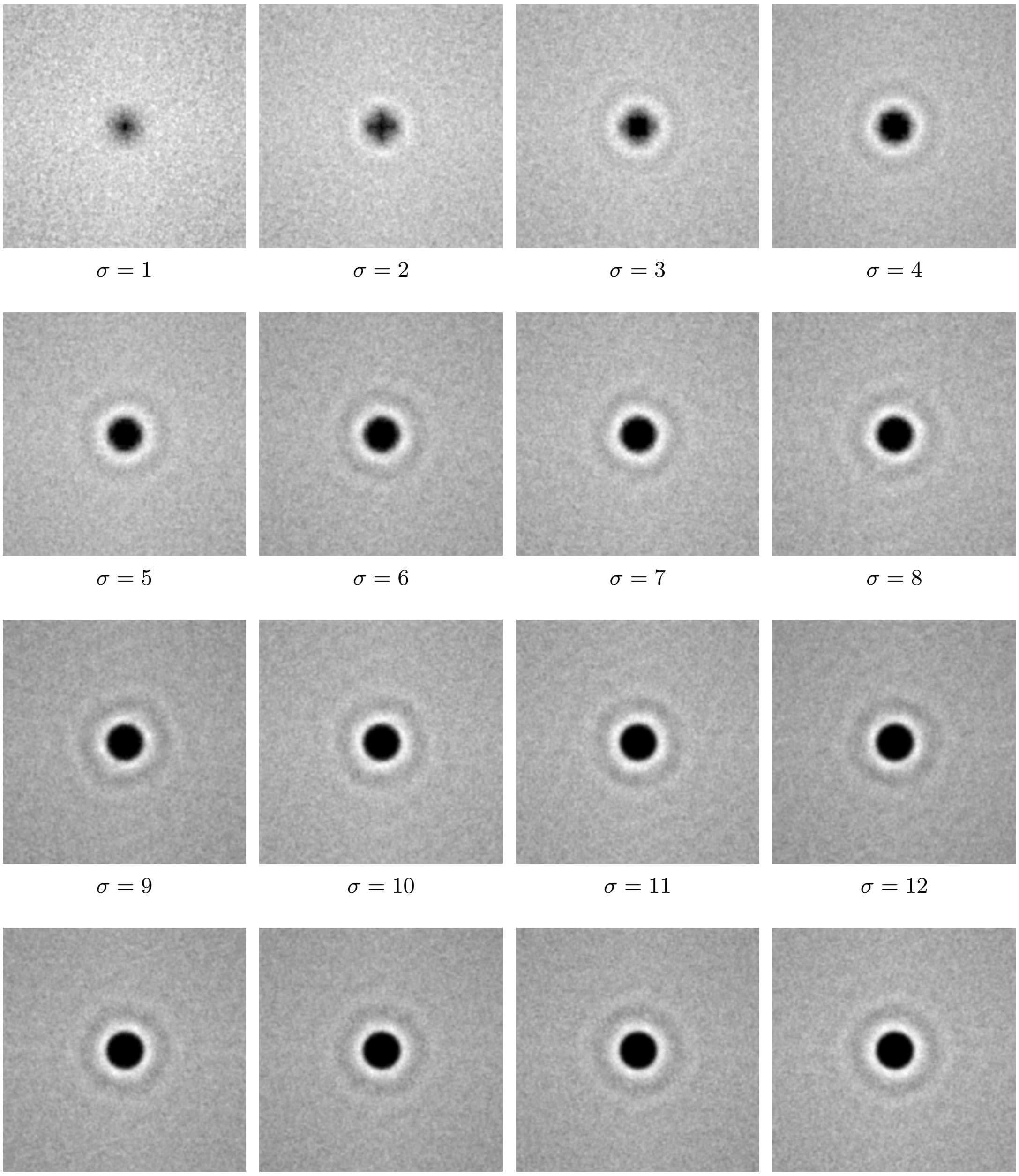
$\sigma = 1$ $\sigma = 2$ $\sigma = 3$ $\sigma = 4$

$\sigma = 5$ $\sigma = 6$ $\sigma = 7$ $\sigma = 8$

$\sigma = 9$ $\sigma = 10$ $\sigma = 11$ $\sigma = 12$

$\sigma = 13$ $\sigma = 14$ $\sigma = 15$ $\sigma = 16$

Figure 4.5: Estimated DDA power spectra of FRS for different values of σ



$\sigma = 1$ $\sigma = 2$ $\sigma = 3$ $\sigma = 4$

$\sigma = 5$ $\sigma = 6$ $\sigma = 7$ $\sigma = 8$

$\sigma = 9$ $\sigma = 10$ $\sigma = 11$ $\sigma = 12$

$\sigma = 13$ $\sigma = 14$ $\sigma = 15$ $\sigma = 16$

Figure 4.6: Estimated DDA power spectra of jittered FRS for different values of σ

CHAPTER 5

Implementation

5.1 Overview

In this chapter, an implementation of FRS is presented for both CPU and GPU application. As shown in the previous chapter, thresholding of the window can be parallelized, because the calculations are independent from each other. The window size S_W and the scaling factor τ can be calculated in a preceding setup step, which is explained shortly in the next section.

Although sampling in higher dimensions is possible, only the two-dimensional case is implemented for the sake of simplicity. Furthermore, the importance function $\mathcal{I}$ is assumed to be a discrete *importance map*, which allows a more efficient implementation of the thresholding step. In Figure 5.1, a Gaussian blob importance map of size $S_I = 8 \times 8$ is used to illustrate this. The relation of the window size S_W to S_I is emphasized by overlaying the window with the pixel grid of $\mathcal{I}$. Each pixel $p_{\mathcal{I}} = (p_{\mathcal{I},x}, p_{\mathcal{I},y}) \in [0, S_{\mathcal{I},x} - 1] \times [0, S_{\mathcal{I},y} - 1]$ of $\mathcal{I}$ (green) corresponds to 8×8 elements – *subpixels* – of the window (orange). Each subpixel inside the orange selection is compared to the product of the importance $\mathcal{I}(p_{\mathcal{I}})$ and the scaling factor.

If nearest-neighbor interpolation is used to look up $\mathcal{I}(p_{\mathcal{I}})$, the importance is the same for each subpixel of $p_{\mathcal{I}}$, which is why the work can be partitioned into chunks for each $p_{\mathcal{I}}$. On the CPU, this leads to an outer loop for each $p_{\mathcal{I}}$ and an inner loop for thresholding each subpixel. On the GPU, one OpenCL thread is created for each $p_{\mathcal{I}}$, thresholding is done in a kernel. In both cases, only a few subpixels (blue) pass thresholding and are then used to generate samples.

If linear interpolation is used instead, there are still known upper and lower bounds for the importance inside $p_{\mathcal{I}}$ which can be used to accelerate thresholding of the corresponding block of window elements. Unless $n \gg S_{\mathcal{I},x} \cdot S_{\mathcal{I},y}$, however, there is no significant difference between nearest-neighbor and linear interpolation of $\mathcal{I}$ in terms of sample distribution quality. Thus, nearest-neighbor interpolation is preferred due to its lower computational effort.

Additional to the naive implementation which performs thresholding for each subpixel, four optimizations are presented which try to reduce the number of threshold comparisons with different representations of the dither matrix $\mathcal{M}$. The implementations Stack, Level N and Flat N are based on a minimum map of $\mathcal{M}$, the Sorted implementation relies on a list of block-wise

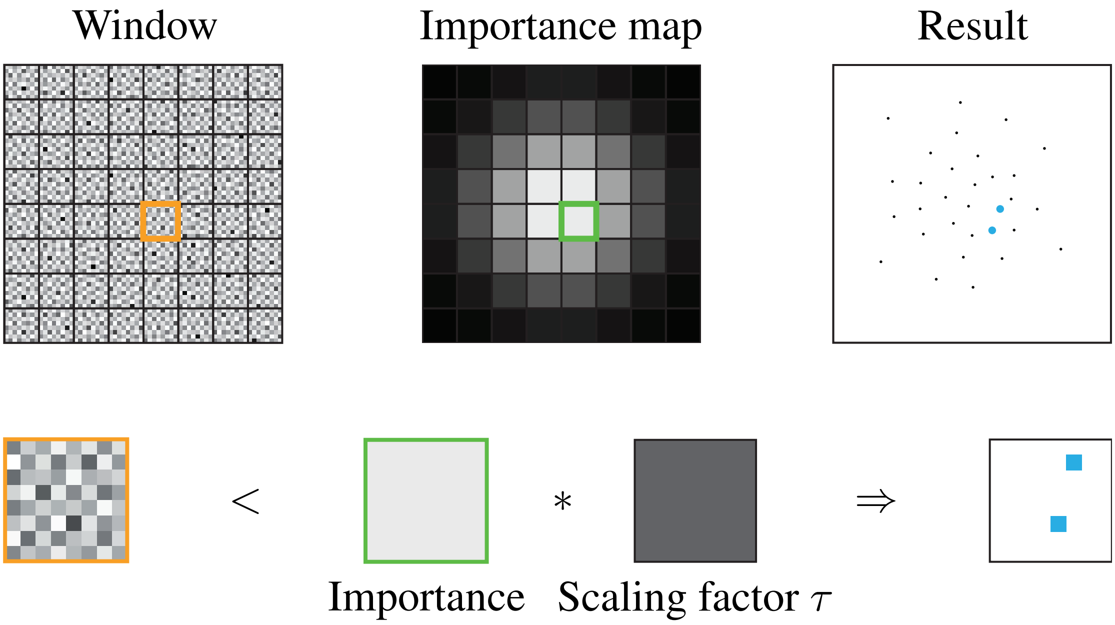


Figure 5.1: Forced Random Sampling for a discrete importance function

sorted dither values. All implementations produce the same result, but differ strongly in terms of runtime, which is analyzed in Chapter 6. It is briefly anticipated here that the Sorted implementation is by far the most efficient FRS implementation for both CPU and GPU.

When referring to one of the implementations in the following, a monospace font is used for clarity, i.e., `Naive`, `Stack`, `Level N`, `Flat N` and `Sorted`.

5.2 Setup

The minimum window size $S_W = S_{W,1} \times \cdots \times S_{W,k}$ can be calculated with Equation 4.12. In general, this is not an integer size, which makes an implementation cumbersome and prone to numerical errors. To avoid rounding errors and unnecessarily expensive calculations, a deviation from the formal explanation of FRS in the previous chapter is made. The window size is restricted to be an integer multiple of S_I. Thus, the actual window size is typically larger than necessary. For the calculation of the final window size,

$$S'_{W,1} = S_{\mathcal{I},1}\phi\left(\frac{S_{W,1}}{S_{\mathcal{I},1}}\right), \quad S_{W,i} = \left\lceil a_{i,1} S'_{W,1} \right\rceil, \tag{5.1}$$

there are two options, depending on the function $\phi(x)$. If the ceiling function

$$\phi(x) := \min(z \in \mathbb{Z} | z \geq x) \tag{5.2}$$

is applied, the window size will be the next integer multiple of the importance map size. If, instead, $\phi(x)$ is chosen to return the next power-of-2 number $\geq x$,

$$\phi(x) := 2^{\max(0,\lceil \log_2 x \rceil)}, \quad x > 0, \tag{5.3}$$

the window size will be a power-of-2 multiple of the importance map size. This also means that the number of subpixels per pixel of $\mathcal{I}$,

$$n_s = \frac{S_{W,1}}{S_{\mathcal{I},1}}, \tag{5.4}$$

is a power of 2. On the one hand, this leads to computational overhead because the window will typically be larger, but on the other hand, it allows a more efficient implementation of certain calculations with bit-wise operations. Additionally, most of the implementations discussed in this chapter rely on the number of subpixels to be a power of 2. For the scope of this thesis, all calculations have only been performed with the power-of-2 function.

Pseudo-code of the simple setup of FRS for $k = 2$ is provided in Listing 5.1. The distinction between *small* and *large* is made because the window size depends on the smaller of the two extents of the importance map.

```
$\mu_{\mathcal{I}}$ = mean($\mathcal{I}$);
$m_{\mathcal{I}}$ = max($\mathcal{I}$);

small = min($S_{\mathcal{I},x}$, $S_{\mathcal{I},y}$);
large = max($S_{\mathcal{I},x}$, $S_{\mathcal{I},y}$);
$a$ = large / small;

$w_{min}$ = ceil($\sigma$ * sqrt($n$ * $m_{\mathcal{I}}$ / ($\mu_{\mathcal{I}}$ * $a$)));
$w_{small}$ = small * pow(2, max(0, ceil(log2($w_{min}$ / small))));
$w_{large}$ = ceil($w_{small}$ * $a$);
$S_W$ = ($S_{\mathcal{I},x}$ > $S_{\mathcal{I},y}$) ? ($w_{large}$, $w_{small}$) : ($w_{small}$, $w_{large}$);

$n_s$ = $S_{W,x}$ / $S_{\mathcal{I},x}$;

$\Delta_W$ = floor(rand() * $S_M$ / $n_s$) * $n_s$;

$\tau$ = $n$ * $S_M$ * $S_M$ / ($S_{W,x}$ * $S_{W,y}$ * $\mu_{\mathcal{I}}$);
```

Listing 5.1: Setup of FRS

Since FRS does not always generate exactly n samples, a data structure slightly larger than n should be used to store the samples. For a CPU implementation, a linked list is used for this task. On the GPU, however, a usual buffer has to be used, as anything else would greatly influence the runtime performance. The buffer is global, thus access of the concurrent threads to it has to be regulated. One possible solution is to assign a fixed range of buffer indices to each thread, such that each thread only accesses its own part of the buffer. Then the question arises how large this range should be. It follows from Equation 4.6 that the average number of samples obtained from one block of $n_s \times n_s$ subpixels is n_s^2/σ^2. As the dither matrix is random, the actual number of samples slightly varies and can also be larger than this. To prevent samples from being discarded, the maximum number of samples allowed per thread is twice this size,

$$n_t = \frac{2n_s^2}{\sigma^2}. \tag{5.5}$$

This means that the buffer on the GPU has to have the size $n_t \cdot t$, where

$$t = S_{\mathcal{I},x} \cdot S_{\mathcal{I},y} \tag{5.6}$$

is the number of threads, which is much larger than actually needed for the computation of n samples. It also means that the buffer is only filled sparsely with the resulting samples. If the samples are not to be used in the thresholding kernel itself, they should be passed on in a compact form. Benchmarks of a compaction step with parallel prefix sum [HSO07] showed that for large n, compacting the samples is just as expensive as thresholding itself.

A better solution for the storage of samples in the video memory is to allow all threads synchronized access to any part of the global buffer. OpenCL 1.1 provides the *atomic_inc* function, which can be used to increment a global index of the next free spot in the buffer. Although synchronization itself is more costly than if each thread was writing to its own partition of the buffer, compacting is no longer required. This is why in total, using synchronized access is faster. In conclusion, the setup of a GPU implementation of FRS additionally requires a buffer of size n or slightly larger – to account for the randomness – and a global index variable for synchronization.

5.3 Naive

The straightforward implementation of Forced Random Sampling compares every dither value of the window with the threshold function (Equation 4.3). On the CPU, this is done by simply iterating over each pixel of the importance map in an outer loop and over each subpixel of the corresponding block of subpixels in the window in an inner loop. A sample is created at the position of the subpixel if its dither value is smaller than the threshold function. If the sample is to be jittered, a random subpixel offset is generated, else the sample is moved to the center of the subpixel and output. In all implementations, the random offsets are two-dimensional Halton points, which are very inexpensive in their generation and suffice for the application of jittering.

Pseudo-code for the `Naive` CPU implementation of FRS is shown in Listing 5.2. $\mathcal{M}$ and $\mathcal{I}$ are assumed to be given as matrices or textures that can be accessed with two-dimensional element coordinates. It can be seen that apart from *rand()* for the Halton or uniform random numbers, only very basic calculations are needed for thresholding at runtime. The floating point divisions in Lines 5 and 17 can be replaced by a multiplication with the precomputed constant fractions. The integer modulo operation in Line 12 can be replaced by a faster bit-wise AND, because

$$a \bmod b = a \,\&\, (b - 1), \tag{5.7}$$

```
samples = new List($n$ * 1.1);

for ($y$ = 0; $y$ < $S_{\mathcal{I},y}$; $y$++) {
  for ($x$ = 0; $x$ < $S_{\mathcal{I},x}$; $x$++) {
    relativeCoord = ($x$, $y$) / $S_\mathcal{I}$;
    windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

    weightedThreshold = $\tau$ * $\mathcal{I}$($x$, $y$);

    for ($p_{W,y}$ = 0; $p_{W,y}$ < $n_s$; $p_{W,y}$++) {
      for ($p_{W,x}$ = 0; $p_{W,x}$ < $n_s$; $p_{W,x}$++) {
        ditherCoord = (windowCoord + ($p_{W,x}$, $p_{W,y}$)) mod $S_M$;

        if ($\mathcal{M}$(ditherCoord) < weightedThreshold) {
          $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

          samples.add(relativeCoord + (($p_{W,x}$, $p_{W,y}$) + $\Delta_p$) / $S_W$);
        }
      }
    }
  }
}
```

Listing 5.2: `Naive` CPU implementation

if b is a power of 2, which is true for S_M. For better readability, however, such optimizations are omitted in the listing. *samples*, the data structure in which the samples are stored, can be initialized with a capacity slightly larger than n in order to avoid dynamic resizing.

The GPU implementation shown in Listing 5.3 looks very similar. The main difference is that the outer loops over all t pixels of the importance map have been replaced by a single call that enqueues t threads on the GPU with the thresholding kernel. Inside the kernel, the importance map coordinates are determined with the help of the individual *threadID*. The second difference to the CPU implementation is that $\mathcal{M}$ and $\mathcal{I}$ are now assumed to be one-dimensional buffers, which is why access has to be done with an index rather than with coordinates. After thresholding, the samples are written to a global buffer of fixed size, where the already mentioned *atomic_inc* function is used for synchronization. It reads the current value – here the next available index in the buffer – from *index*, increments it by one and then returns the previous value. After all threads have finished work, the value of *index* is the number of samples in the buffer, so a sub-buffer of *samples* of the range 0 to *index* -1 can be passed to the actual sampling application.

The `Naive` implementation of FRS is very simple and can easily be extended to higher dimensions by including an additional outer and inner loop for each dimension. However, it performs a threshold comparison for each subpixel of the window regardless of prior knowledge of $\mathcal{M}$ and the maximum allowed number of samples per subpixel block, n_t. If it is known that on average only one out of σ^2 subpixels passes thresholding, there is no need to perform all comparisons, but `Naive` does not allow the thread to terminate early.

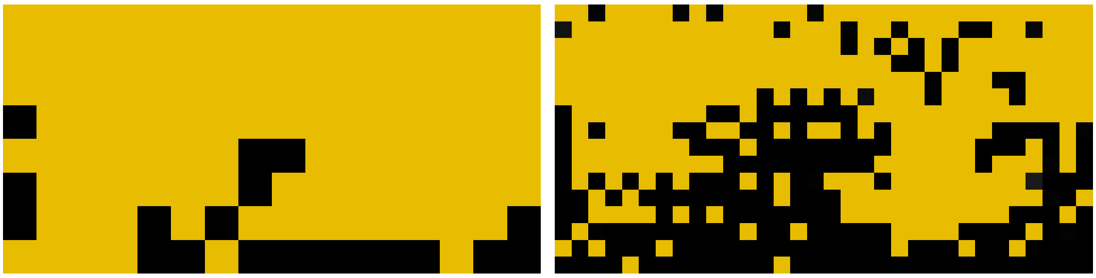
```
samples = new GPU_Buffer($n$ * 1.1);
index = new GPU_Integer(0);

kernel = {
  $y$ = threadID / $S_{\mathcal{I},x}$; // integer truncation intended
  $x$ = threadID - $y$ * $S_{\mathcal{I},x}$;
  relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
  windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

  weightedThreshold = $\tau$ * $\mathcal{I}$[$x$ + $S_{\mathcal{I},x}$ * $y$];

  for ($p_{W,y}$ = 0; $p_{W,y}$ < $n_s$; $p_{W,y}$++) {
    for ($p_{W,x}$ = 0; $p_{W,x}$ < $n_s$; $p_{W,x}$++) {
      ditherCoord = (windowCoord + ($p_{W,x}$, $p_{W,y}$)) mod $S_M$;

      if ($\mathcal{M}$[ditherCoord.x + $S_M$ * ditherCoord.y] < weightedThreshold) {
        $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

        samples[atomic_inc(index)] = relativeCoord + (($p_{W,x}$, $p_{W,y}$) + $\Delta_p$) / $S_W$;
      }
    }
  }
};

kernel.Call($t$);
```

Listing 5.3: `Naive` GPU implementation

5.4 Stack

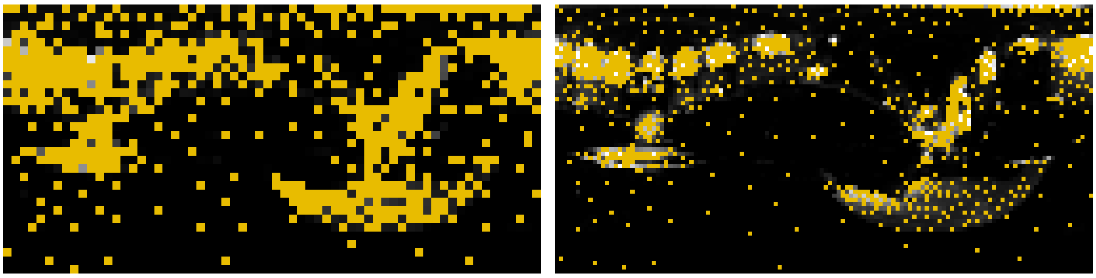
The early termination of unpromising decision branches is crucial for the performance of FRS. As the dither matrix is known beforehand, it can be preprocessed in order to restructure the contained information. The representation of $\mathcal{M}$ by a minimum map, or *minmap*, has the advantage of being hierarchical. If level 0 is the original dither matrix, then the size of level 1 is only $S_M/2 \times S_M/2$ and every element in level 1 has the minimum value of the corresponding 2×2 block of values of level 0. If four elements of $\mathcal{M}$ are to be thresholded with $\mathrm{T}(p_{\mathcal{I}})$ and the level 1 value of these four elements is already greater than $\mathrm{T}(p_{\mathcal{I}})$, then no other comparison has to be performed. None of the remaining three values can be smaller than $\mathrm{T}(p_{\mathcal{I}})$. This becomes even more beneficial when looking at level 2 of the minmap, because 4×4 elements can potentially be discarded at once. If the level 2 minimum is smaller than $\mathrm{T}(p_{\mathcal{I}})$, the level 1 tests for the four 2×2 blocks are performed. At least one block will lead to a sample, but the other three might still be discarded based on their level 1 values, meaning that only three instead of 12 buffer reads and comparisons would be necessary for them.

With an increasing level, the size of the block that can potentially be discarded grows. However, it becomes also more likely that a value lower than $\mathrm{T}(p_{\mathcal{I}})$ is in a larger block of pixels, as the values are distributed very evenly in the dither matrix. As on average, one out of σ^2 pixels should pass thresholding, the highest useful minmap level is $\log_2 \sigma$. This is illustrated in Figure 5.2, where lookups into the minmap for 1024 samples are visualized by orange squares for $\sigma = 8$. For better visualization, all minmap levels have been scaled to the same size, such that the size of each square corresponds to the block of dither values it covers. It can be seen that almost every threshold of minmap level 4 succeeds, meaning that the retrieved value is $< \mathrm{T}(p_\mathcal{I})$, in which case the block cannot be discarded. Thus, the additional level 4 lookup actually leads to a performance loss instead of gain. Even the lookup at level 3 succeeds in most of the cases, such that only less than half of the blocks can be discarded. At level 2 and 1, most of the blocks can be discarded, but the total number of comparisons is four times as high as in the previous level.



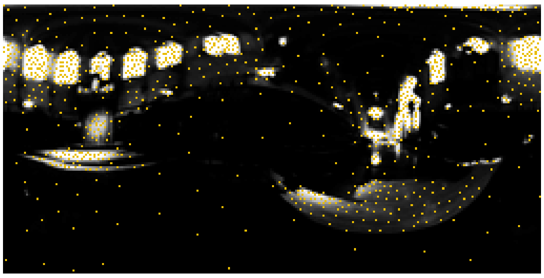
(a) Level 4

(b) Level 3



(c) Level 2

(d) Level 1



(e) Level 0

Figure 5.2: Hierarchical lookups into the minmap of the dither matrix

A second restriction on the maximum level is given by the number of subpixels n_s. Each pixel $p_\mathcal{I}$ of the importance map corresponds to $n_s \times n_s$ subpixels of the window, so $\mathrm{T}(p_\mathcal{I})$ is the same for every subpixel of this block. Larger blocks of subpixels could not be discarded with a single comparison of the minimum value because the importance needed for the threshold would not be constant. This also explains why in Chapter 4, n_s has been restricted to a power of 2 for all implementations but `Naive`. It ensures that the window of the minmap at level $\log_2 n_s$ coincides with the importance map, such that each value of this minmap level is the minimum value of all subpixels that can be thresholded with the same importance $\mathcal{I}(p_{\mathcal{I}})$.

For an actual implementation of FRS using a minmap, a stack-based implementation is a very obvious choice. For each pixel $p_{\mathcal{I}}$ of the importance map, a stack is initialized with the corresponding minimum value m of minmap level $\log_2 \min(n_s, \sigma)$. It is thresholded with $\mathrm{T}(p_{\mathcal{I}})$ and if $m \geq \mathrm{T}(p_{\mathcal{I}})$, no further comparisons have to be performed for this $p_{\mathcal{I}}$. If $m < \tau(p_{\mathcal{I}})$, the minimum values of the four quadrants of minmap level $\log_2 \min(n_s, \sigma) - 1$ are pushed to the stack. If level 0 is reached, a sample is created. Once the stack is empty, thresholding terminates. Pseudo-code for the CPU implementation `Stack` is provided in Listing 5.4. Here, the minmap is assumed to be stored in a data structure such that each level $\mathcal{M}'$ is an individual matrix of size $S_{M'}$. As the minmap only depends on the dither matrix, it can either be created along with the dither matrix itself and stored in a file or it can be calculated when loading the dither matrix, which only takes a split second.

The stack is initialized with an upper bound of the number of elements that can be pushed to it. Each element has three integer components, which are the two-dimensional offset of the quadrant inside the $n_s \times n_s$ block and the number of subpixels inside this quadrant. To account for the different sizes of the minmap levels, the calculation of the coordinates for dithering in Equation 4.13 is extended as shown in Line 19. Again, this implementation can be optimized by replacing the divisions and modulo operations by bit-wise shifts and ANDs. The *log2* operation in Line 18 can be implemented with a lookup table, as only power of 2 values $\leq n_s$ need to be considered.

The GPU implementation is essentially the same as the CPU implementation, which is why its code is omitted here. The only notable difference is that all levels of the dither minmap are assumed to be stored in the same buffer such that the origin of the individual levels can be calculated from S_M and the desired lookup level. This offset simply has to be added to *ditherCoord* in Line 19.

Comparing `Stack` to `Naive`, it is obvious that less dither matrix accesses and less threshold comparisons are necessary. However, an additional read access to the stack buffer is required in each iteration. Even worse, cheap read accesses are traded for expensive write accesses when pushing elements onto the stack, which is disadvantageous especially on the GPU.

```
samples = new List($n * 1.1$);
stack = new Stack($n_s * 4 + 1$);

for ($y$ = 0; $y$ < $S_{\mathcal{I},y}$; $y$++) {
  for ($x$ = 0; $x$ < $S_{\mathcal{I},x}$; $x$++) {
    relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
    windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

    weightedThreshold = $\tau$ * $\mathcal{I}(x, y)$;

    stack.push((0, 0, $n_s$));

    while (!stack.empty) {
      block = stack.pop();
      blockCoord = (block.x, block.y);
      $n'_s$ = block.z;

      $\mathcal{M}'$ = minmap[log2($n'_s$)];
      ditherCoord = ((windowCoord mod $S_M$) / $n'_s$ + blockCoord) mod $S_{M'}$;

      if ($\mathcal{M}'$(ditherCoord) < weightedThreshold) {
        if (blockSubpixels == 1) {
          $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

          samples.add(relativeCoord + (blockCoord + $\Delta_p$) / $S_W$);
        }
        else {
          stack.push((2 * blockCoord, $n'_s$ / 2));
          stack.push((2 * blockCoord + (1, 0), $n'_s$ / 2));
          stack.push((2 * blockCoord + (0, 1), $n'_s$ / 2));
          stack.push((2 * blockCoord + (1, 1), $n'_s$ / 2));
        }
      }
    }
  }
}
```

Listing 5.4: `Stack` CPU implementation

5.5 Level N

The idea of the `Level N` implementations is to implement hierarchical thresholding for FRS in a stack-free way to avoid write accesses to a buffer. For this, the `Naive` implementation is extended by additional inner loops for each minmap level > 0. As the thresholding code differs depending on the highest minmap level N, it is generated automatically according to a template. In the following, only the simplest variant, `Level 1`, will be discussed, as it sufficiently illustrates the idea of this implementation. Theoretically, the highest useful minmap level is $N = \log_2 n_s$, but in practice, only `Level 1` and `Level 2` are feasible implementations. The GPU implementations of higher levels cannot be compiled due to the large number of nested loops. `Level 3` and higher levels run generally slower on the CPU than the first two levels. On the GPU, the runtime of `Level 1` and `Level 2` is very similar, which is why `Level 1` is also the only `Level N` variant considered for the performance analysis in Chapter 6.

Pseudo-code of the CPU implementation of `Level 1` is provided in Listing 5.5. If thresholding should start from $N = 2$, n_s would have to be divided by 4 instead of 2 in Line 10. Lines 14 to 21 would have to be repeated with the necessary subscript modifications in order to threshold the level 2 dither value first. It is obvious that the implementation becomes more confusing with every additional level. Still, the aim of implementing hierarchical thresholding without a stack has been achieved, which makes this implementation much more suitable for a GPU.

It is possible to further optimize this code, which has been omitted in Listing 5.5 for better readability. Each dither value of the minmap level 1 is the minimum of four corresponding dither values of level 0. These values can be sorted offline, e.g., at minmap creation, and their order can be stored in another matrix. There are 24 possible orders of the four values, which can be encoded in one byte, for example. If the minmap lookup at Line 17 additionally to the dither value returns the order of the four quadrants of the next minmap level, it can be used to early terminate the loops in Lines 18 and 19. As soon as the dither value of one of the four quadrants is greater than the threshold, none of the remaining quadrants has to be considered. When analyzing the performance of all FRS implementations, it became clear, however, that this modification does not actually pay off for `Level 1` because of the additional lookup for the order. `Level 2` slightly benefits from it regarding runtime, but the readability of the generated code deteriorates drastically.

```
samples = new List($n$ * 1.1);

for ($y$ = 0; $y$ < $S_{\mathcal{I},y}$; $y$++) {
  for ($x$ = 0; $x$ < $S_{\mathcal{I},x}$; $x$++) {
    relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
    windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

    weightedThreshold = $\tau$ * $\mathcal{I}$($x$, $y$);

    $n'_s$ = $n_s$ / 2;

    for ($y_1$ = 0; $y_1$ < $n'_s$; $y_1$++) {
      for ($x_1$ = 0; $x_1$ < $n'_s$; $x_1$++) {
        $\mathcal{M}'$ = minmap[1];
        level1Coord = ((windowCoord mod $S_M$) / $n'_s$ + ($x_1$, $y_1$)) mod $S_{M'}$;

        if ($\mathcal{M}'$(level1Coord) < weightedThreshold) {
          for ($y_0$ = 0; $y_0$ <= 1; $y_0$++) {
            for ($x_0$ = 0; $x_0$ <= 1; $x_0$++) {
              q0Offset = 2 * ($x_1$, $y_1$) + ($x_0$, $y_0$);
              level0Coord = (windowCoord + q0Offset) mod $S_M$;

              if ($\mathcal{M}$(level0Coord) < weightedThreshold) {
                $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

                samples.add(relativeCoord + (q0Offset + $\Delta_p$) / $S_W$);
              }
            }
          }
        }
      }
    }
  }
}
```

Listing 5.5: `Level 1` CPU implementation

5.6 Flat N

The `Level N` implementations can be rewritten to iterate over all minmap levels in an outer loop and then perform thresholding for each level individually. The aim is to have simpler thresholding steps without loops that are easy to implement and to maintain. Also, only one minmap level is accessed at the time, which might allow a better use of the caches than the erratic minmap accesses of `Stack` and `Level N`. In a first step, thresholding is performed for minmap level N. Instead of writing samples, however, only an encoded index of each pixel that passed thresholding is stored. For the sake of simplicity, here the index is assumed to be a four-component vector which holds the coordinates (x, y) of $p_{\mathcal{I}}$ as well as the subpixel coordinates $(p_{W,x}, p_{W,y})$ relative to $p_{\mathcal{I}}$. These indices are then used in separate thresholding steps for the next lower level. At each level, the indices of the previous level correspond to 2×2 blocks of pixels in the current level. At least one of these pixels passed thresholding at the previous level, so all of them are thresholded again. The refined indices of the pixels that passed thresholding are then stored again for the next level until level 0. Finally, the indices are translated to actual sample coordinates and stored. Since the overall number of operations increases, but their complexity decreases, it is assumed to be well suited for GPU implementations. This is why in the following, only the GPU implementation will be considered.

Starting thresholding at level N, each pixel $p_{\mathcal{I}}$ of the importance map corresponds to $n_s^2/2^{2N} = n_s'^2$ elements of the minmap at level N. The result buffer must be large enough to store $t = S_{\mathcal{I},x} S_{\mathcal{I},y} / n_s'^2$ indices. In the next step, four quadrants for each of these indices have to be thresholded, which in the worst case all need to be stored as well. This leads to a total of $S_{\mathcal{I},x} S_{\mathcal{I},y} / n_s^2$ indices that need to be stored at level 0 in the worst case. In Listing 5.6, pseudo-code for the first kernel is provided. One thread is created for each of the t necessary thresholds, the assignment of each thread to a subpixel is again done with the unique *threadID*.

After the first thresholding step, $n' \leq t$ subpixel indices are stored in the *indices* buffer, where n' is the current value of the synchronization variable *index*. Since all four quadrants of every index need to be thresholded in the next step, $t' = 4n'$ new threads have to be started. This is repeated until the level 0 thresholding is finished. Pseudo-code for the level $N - 1$ kernel is provided in Listing 5.7.

Once thresholding of minmap level 0 is completed, the remaining n' indices are translated to sample locations and optional jittering is added, as shown in Listing 5.8.

Putting it all together, the `Flat N` implementation of FRS as shown in Listing 5.9 requires $N + 1$ separate OpenCL kernel calls. N kernels require synchronized access to the global memory and at least two separate buffers for input and output of the indices are needed. The complexity reduction inside a kernel as compared to `Level N` is traded for a larger OpenCL overhead as well as more memory consumption. Although each individual kernel is easy to understand and to implement, three individual kernels have to be maintained. Anticipating the results of the performance analysis in Chapter 6, this implementation performs poorly due to the multiple kernel calls, which makes it generally unfavorable.

```
indices = new GPU_Buffer($t$);
index = new GPU_Integer(0);

startKernel = {
  $n'_s$ = $n_s$ / $2^N$;
  blockID = threadID / $n'^2_s$; // integer truncation intended
  subID = threadID - blockID * $n'^2_s$;

  $y$ = blockID / $S_{\mathcal{I},x}$; // integer truncation intended
  $x$ = blockID - $y$ * $S_{\mathcal{I},x}$;
  $p_{W,y}$ = subID / $n'_s$; // integer truncation intended
  $p_{W,x}$ = subID - $p_{W,y}$ * $n'_s$;

  windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;
  $\mathcal{M}'$ = minmap[N];
  levelNCoord = ((windowCoord mod $S_M$) / $n'_s$ + ($p_{W,x}$, $p_{W,y}$)) mod $S_{M'}$;

  weightedThreshold = $\tau$ * $\mathcal{I}$[$x$ + $S_{\mathcal{I},x}$ * $y$];

  if ($\mathcal{M}'$[levelNCoord.x + $S_{M'}$ * levelNCoord.y] < weightedThreshold) {
    indices[atomic_inc(index)] = ($x$, $y$, $p_{W,x}$, $p_{W,y}$);
  }
}
```

Listing 5.6: `Flat N` start kernel

```
indices = new GPU_Buffer($t'$);
index = new GPU_Integer(0);

levelNM1Kernel = {
  indexID = previousIndices[threadID / 4]; // integer truncation intended
  quadrantID = threadID mod 4;
  ($x$, $y$) = (indexID.x, indexID.y);
  ($p_{W,x}$, $p_{W,y}$) = (indexID.z, indexID.w) * 2;
  $p_{W,x}$ += (quadrantID $\in \{1,3\}$) ? 1 : 0; // 0 1
  $p_{W,y}$ += (quadrantID $\in \{2,3\}$) ? 1 : 0; // 2 3

  windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;
  $\mathcal{M}'$ = minmap[N - 1];
  levelNM1Coord = ((windowCoord mod $S_M$) / $n'_s$ + ($p_{W,x}$, $p_{W,y}$)) mod $S_{M'}$;

  weightedThreshold = $\tau$ * $\mathcal{I}$[$x$ + $S_{\mathcal{I},x}$ * $y$];

  if ($\mathcal{M}'$[levelNM1Coord.x + $S_{M'}$ * levelNM1Coord.y] < weightedThreshold) {
    indices[atomic_inc(index)] = ($x$, $y$, $p_{W,x}$, $p_{W,y}$);
  }
}
```

Listing 5.7: `Flat N` level $N-1$ kernel

```
samples = new GPU_Buffer($n'$);

sampleGenerationKernel = {
  indexID = previousIndices[threadID];
  ($x$, $y$) = (indexID.x, indexID.y);
  ($p_{W,x}$, $p_{W,y}$) = (indexID.z, indexID.w);

  relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
  $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

  samples[threadID] = relativeCoord + (($p_{W,x}$, $p_{W,y}$) + $\Delta_p$) / $S_W$;
}
```

Listing 5.8: `Flat N` sample generation kernel

```
index = new GPU_Integer(0);
level = N;
$t$ = $S_{\mathcal{I},x}$ * $S_{\mathcal{I},y}$ / ($n_s^2$ / $2^{2N}$);
indices = startKernel.Call($t$);

for (level = N - 1; level >= 0; level--) {
  $t'$ = index * 4;
  index = 0;
  previousIndices = indices;

  indices = levelNM1Kernel.Call($t'$);
}

samples = sampleGenerationKernel.Call(index);
```

Listing 5.9: `Flat N` GPU implementation

5.7 Sorted

Up to this point, `Level N` is the best implementation on the GPU, which is hard to implement in a tidy way. It can be observed that the hierarchical processing is the main cause for its disadvantages, which is why the `Sorted` implementation uses another dither matrix representation. As mentioned above, the maximum number of samples that can be obtained from the $n_s \times n_s$ dither matrix elements corresponding to the importance map pixel $p_{\mathcal{I}}$ can be estimated as $n_t = 2n_s^2/\sigma^2$ (Equation 5.5). This value has not been taken into account in any of the previous implementations because the dither values in the window and thus the minmap have no order. If, however, the dither values of a whole $n_s \times n_s$ block were sorted, thresholding could stop after n_t comparisons. Even further, an early termination after the first element greater than the threshold would be possible.

To achieve this, the dither matrix has to be sorted *block-wise*, where each block has the size $n_s \times n_s$. For each block, a list of n_s^2 dither values in ascending order can be obtained. Additionally, an element index $\in [0, n_s^2 - 1]$ corresponding to the former position of the pixel in the block is kept for each value. All of these block-wise sorted lists can be stored in one buffer, in the following denoted by $\mathcal{L}$, of length S_M^2, which is the size of the original dither matrix. Now that the dither values are sorted and only a maximum of n_t values per block will be used, only the first n_t elements of each block need to be stored, adding up to $n_t S_{\mathcal{I},x} S_{\mathcal{I},y}$ elements for the whole dither matrix. This is especially useful for the GPU implementation, as only a small portion of the dither matrix needs to be uploaded to and stored in the video memory. Pseudo-code for the generation of $\mathcal{L}$ is provided in Listing 5.10.

```
$n_t$ = 2 * $n_s^2$ / $\sigma^2$;
numBlocksPerDim = $S_M$ / $n_s$;
numBlocks = numBlocksPerDim * numBlocksPerDim;
$\mathcal{L}$ = new List();

for (block = 0; block < numBlocks; block++) {
  blockCoord = (block mod numBlocksPerDim, block / numBlocksPerDim);
  subMatrix = $\mathcal{M}$(blockCoord * $n_s$, $n_s \times n_s$);

  list = new List();

  for (index = 0; index < $n_s^2$; index++) {
    list.add((subMatrix[index], index));
  }

  list.sortAscendingByXComponent();

  $\mathcal{L}$.addRange(list);
}
```

Listing 5.10: Block-wise sorted list generation

Thresholding $\mathcal{L}$ is much easier than thresholding the minmap, because the values in $\mathcal{L}$ are already sorted. The first task in the thresholding step is to select the right n_t elements of $\mathcal{L}$ for the current pixel $p_{\mathcal{I}}$ of the importance map. In the setup (see Listing 5.1), the window offset Δ_W has been chosen such that it is a multiple of n_s, which is required in this case. The start index of the sought elements in $\mathcal{L}$ for $p_{\mathcal{I}}$ can then be calculated easily. Once the start index is found, the next n_t elements are compared to the threshold. For each element which passes thresholding, a sample is created. As soon as the dither value of one element is greater than the threshold, thresholding can terminate, as the elements are sorted by their dither values and none of the remaining elements can have a smaller value. Pseudo-code of the `Sorted` FRS implementation for CPU and GPU is provided in Listings 5.11 and 5.12. Once again, all divisions and modulo operations inside the kernel can be avoided by either precomputing the fraction or by replacing them with bit-wise operations.

```
samples = new List($n * 1.1$);

for ($y$ = 0; $y$ < $S_{\mathcal{I},y}$; $y$++) {
  for ($x$ = 0; $x$ < $S_{\mathcal{I},x}$; $x$++) {
    relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
    windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

    weightedThreshold = $\tau * \mathcal{I}(x, y)$;

    blockWindowSize = $S_W$ / $n_s$;
    blockMatrixSize = $S_M$ / $n_s$;
    blockOffset = $\Delta_W$ / $n_s$;

    // block location inside the dither matrix
    blockCoord = (($x$, $y$) + blockOffset) mod blockMatrixSize;

    // block location in the flattened list $\mathcal{L}$
    blockStartIndex = (blockCoord.y * blockMatrixSize + blockCoord.x) * $n_t$;

    for (j = 0; j < $n_t$; j++) {
      blockElement = $\mathcal{L}$[blockStartIndex + j]; // (ditherValue, subpixelIndex)

      if (blockElement.x < weightedThreshold) {
        $p_{W,y}$ = blockElement.y / $n_s$; // integer truncation intended
        $p_{W,x}$ = blockElement.y - $p_{W,y}$ * $n_s$;
        $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

        samples.add(relativeCoord + (($p_{W,x}$, $p_{W,y}$) + $\Delta_p$) / $S_W$);
      }
      else {
        break;
      }
    }
  }
}
```

Listing 5.11: `Sorted` CPU implementation

```
samples = new GPU_Buffer($n$ * 1.1);
index = new GPU_Integer(0);

kernel = {
  $y$ = threadID / $S_{\mathcal{I},x}$; // integer truncation intended
  $x$ = threadID - $y$ * $S_{\mathcal{I},x}$;
  relativeCoord = ($x$, $y$) / $S_{\mathcal{I}}$;
  windowCoord = $\Delta_W$ + ($x$, $y$) * $n_s$;

  weightedThreshold = $\tau$ * $\mathcal{I}$[$x$ + $S_{\mathcal{I},x}$ * $y$];

  blockWindowSize = $S_W$ / $n_s$;
  blockMatrixSize = $S_M$ / $n_s$;
  blockOffset = $\Delta_W$ / $n_s$;

  // block location inside the dither matrix
  blockCoord = (($x$, $y$) + blockOffset) mod blockMatrixSize;

  // block location in the flattened list $\mathcal{L}$
  blockStartIndex = (blockCoord.y * blockMatrixSize + blockCoord.x) * $n_t$;

  for (j = 0; j < $n_t$; j++) {
    blockElement = $\mathcal{L}$[blockStartIndex + j]; // (ditherValue, subpixelIndex)

    if (blockElement.x < weightedThreshold) {
      $p_{W,y}$ = blockElement.y / $n_s$; // integer truncation intended
      $p_{W,x}$ = blockElement.y - $p_{W,y}$ * $n_s$;
      $\Delta_p$ = (jitter) ? (rand(), rand()) : (0.5, 0.5);

      samples[atomic_inc(index)] = relativeCoord + (($p_{W,x}$, $p_{W,y}$) + $\Delta_p$) / $S_W$;
    }
    else {
      return;
    }
  }
}

kernel.Call($S_{\mathcal{I},x}$ * $S_{\mathcal{I},y}$);
```

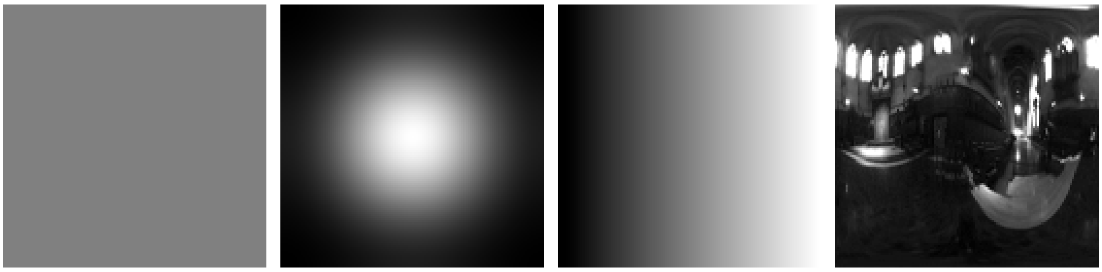
Listing 5.12: `Sorted` GPU implementation

CHAPTER 6

Analysis

6.1 Overview

In this chapter, different tests for the evaluation of quality and performance of sample distributions are presented together with comparisons and discussions of the results. The quality of FRS, jittered FRS, regular, uniform random, Poisson disk and Halton sampling is assessed by the use of the spatial measures discrepancy and density. This is complemented by a Fourier analysis in the uniform and a differential domain analysis in the adaptive case, which helps to identify harmful patterns that did not come out in the spatial analysis. The performance of the different FRS implementations is measured for adaptive sampling and compared to the performance of HSW for several importance maps depicted in Figure 6.1.



(a) Constant (b) Gaussian Blob (c) Linear Gradient (d) Environment

Figure 6.1: Importance maps used for adaptive sampling. (d) is based on the Grace Cathedral light probe retrieved from [Deb98].

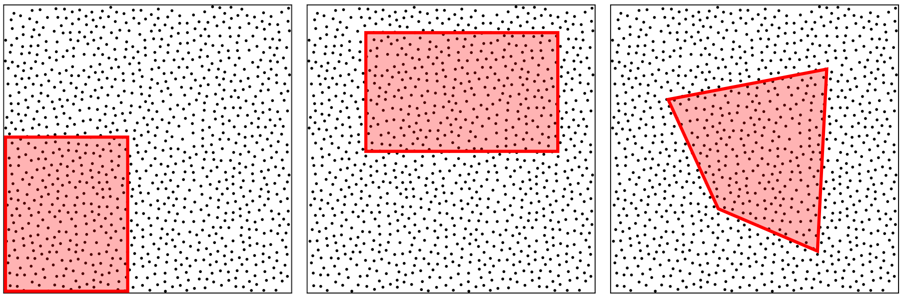
6.2 Spatial Analysis of Sample Distributions

Discrepancy

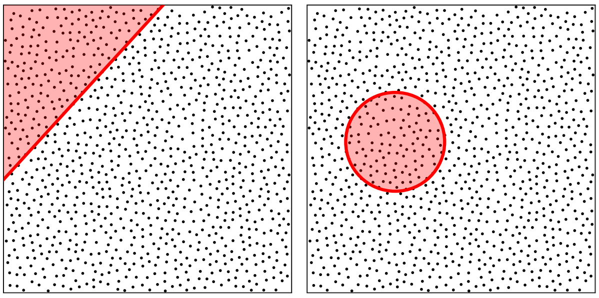
For the spatial analysis of sample distributions in two dimensions, a tool has been developed to calculate the average discrepancy $D_2(X)$ and worst-case discrepancy $D_\infty(X)$ of sample sets, which both depend on the local discrepancy d_X. The local discrepancy itself depends on a geometric shape $A \in \mathcal{A}$, where $\mathcal{A}$ is a class of geometric shapes such as rectangles or circles. It is defined for every point in the sample domain $\Omega = [0,1)^2$, which makes an analytic solution very difficult. Instead, Ω is discretized into a regular grid $\hat{\Omega} \subset \Omega$ of size $s \times s$, for which the discrepancies can be calculated easily in finite time. Likewise, a finite subset $\mathcal{A}$ of geometric shapes in Ω is used. Equations 2.14 can then be reformulated to

$$D_2(X) = \sqrt{\frac{1}{s^2} \sum_{A \in \mathcal{A}} d_X(A)^2} \quad \text{and} \quad D_\infty(X) = \max_{A \in \mathcal{A}} d_X(A), \tag{6.1}$$

$$\text{where} \quad d_X(A) = \left| \frac{\|X \cap A\|}{n} - \lambda_2(A) \right|. \tag{6.2}$$



(a) Star (b) Rectangle (c) Quadrilateral



(d) Half-space (e) Circle

Figure 6.2: Different geometric shapes used for the geometry tests

The grid size s has been chosen very conservatively for each $\mathcal{A}$ such that no significant changes in the resulting discrepancy could be observed and that the GPU resources are used to capacity. For a finite number of shapes $A \in \mathcal{A}$, $d_X(A)$ is calculated. In the following, the calculation of $d_X(A)$ for a single set of samples X and shape A is called a *geometry test*. As the geometry tests vary for different classes of shapes $\mathcal{A}$ (see Figure 6.2), the implementations are explained separately below. The local discrepancy of the grid points leads to the calculation of one sample set's $D_2(X)$ and $D_\infty(X)$. To estimate D_2 and D_∞ of the sample distribution itself, the results of m sample sets or *runs* are used:

$$D_2 = \frac{1}{m}\sum_{i=0}^{m-1} D_2(X_i), \quad D_\infty = \max_{i=0\ldots m-1} D_\infty(X_i). \tag{6.3}$$

The resulting average and worst-case discrepancies of a sample distribution are now only dependent on the number of samples n instead of a concrete set of samples. In the following, D_2 and D_∞ calculated from $m = 1000$ runs are plotted for different values of n from 2^2 to 2^{13} to visualize the discrepancies' asymptotic behavior. Although a single value of D_∞ has no meaning for itself, the slope of D_∞ gives an indication of how fast the integration error of the Monte Carlo method diminishes with an increasing number of samples.

Star Discrepancy

The simplest and most important class of geometric shapes for the calculation of discrepancy is the class $\mathcal{A}_{Star}$ of rectangles with one corner in the origin $(0, 0)$ and the opposite corner in the point (x, y). For higher dimensions, an infinite number of boxes of varying size concentrates at the origin, which is why the resulting discrepancy is often referred to as *star discrepancy*. The individual rectangle $A(x, y) \in \mathcal{A}_{Star}$ only depends on the coordinates of the corner farthest away from the origin, which is why the discrepancy calculation is trivial. For each grid point $(x, y) \in \hat{\Omega}$, the area of $A(x, y)$ is

$$\lambda_2(A(x, y)) = xy \tag{6.4}$$

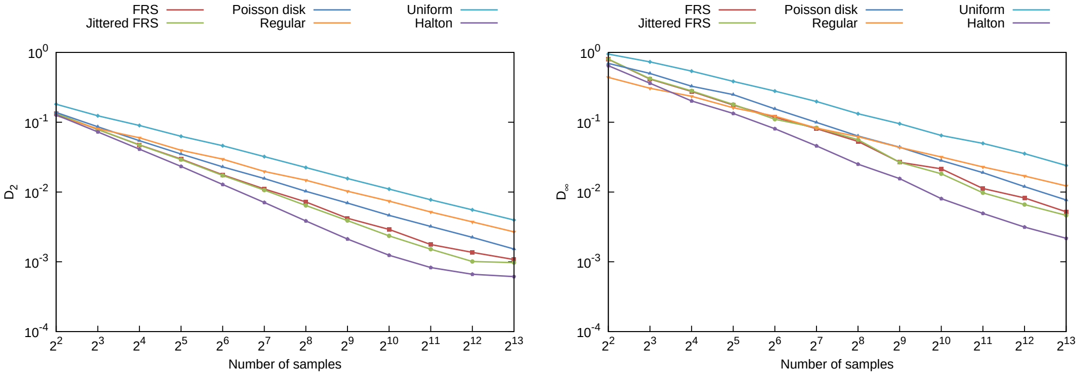
and the number of samples in A is

$$\|X \cap A\| = \sum_{i=0}^{n-1} n(i), \quad n(i) = \begin{cases} 1, & x_{i,x} < x, \quad x_{i,y} < y, \\ 0 & \text{otherwise.} \end{cases} \tag{6.5}$$

Since the locations of the samples are constant for all $A(x, y)$, Equation 6.5 can be implemented efficiently by using a summed-area table SAT_X of size $s \times s$ to store the discretized sample locations. The local discrepancy then simplifies to

$$d_X(A(x, y)) = \left| \frac{SAT_X(x, y)}{n} - xy \right|. \tag{6.6}$$

The results of the star discrepancy test are shown in Figure 6.3 in logarithmic scale. The grid size was chosen as $s = 3000$, hence 3000^2 different geometry tests have been performed for each n, sample distribution and sample set. Unsurprisingly, the samples drawn from the Halton sequence have a very low star discrepancy, because this is what low-discrepancy sequences are designed for. The generally high discrepancy of the uniform random distribution was likewise expected, because the samples tend to form clusters and holes randomly. More surprising is the rather poor performance of the the Poisson disk distribution, considering it is the assumed optimal distribution for sampling. Both FRS and the slightly better jittered FRS range well between Poisson disk and Halton sampling.



(a) Average star discrepancy

(b) Worst-case star discrepancy

Figure 6.3: Results of the star discrepancy tests

Rectangle Discrepancy

The star discrepancy can be generalized by testing all axis-aligned rectangles in $\hat{\Omega}$ instead of only the ones with one corner in the origin. In contrast to $\mathcal{A}_{Star}$, the class of all axis-aligned rectangles $\mathcal{A}_{Rectangle}$ is invariant to right angle rotations and does not overemphasize sample points close to the origin. One rectangle $A(E, F)$ depends on the coordinates of the corner $E = (E_x, E_y)$ and its opposite corner $F = (F_x, F_y)$. For each pair of grid points $(E, F) \in \hat{\Omega}$, its area is

$$\lambda_2(A(E,F)) = (F_x - E_x)(F_y - E_y) \tag{6.7}$$

and the number of samples in A is

$$\|X \cap A\| = \sum_{i=0}^{n-1} n(i), \quad n(i) = \begin{cases} 1, & E_x \leq x_{i,x} < F_x, \quad E_y \leq x_{i,y} < F_y, \\ 0 & \text{otherwise.} \end{cases} \tag{6.8}$$

Similar to the star discrepancy test, the rectangle discrepancy test can be implemented efficiently by using a summed-area table to store the sample locations:

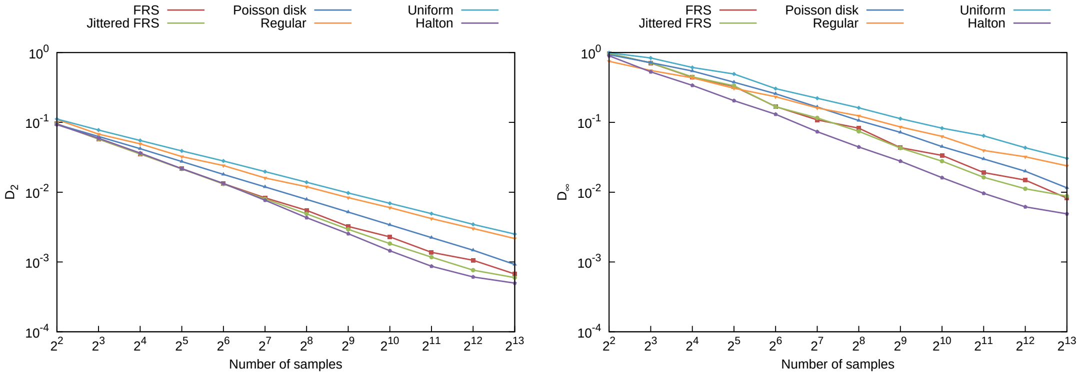
$$\|X \cap A\| = SAT_X(F_x, F_y) - SAT_X(E_x, F_y) - SAT_X(F_x, E_y) + SAT_X(E_x, E_y). \tag{6.9}$$

From this follows the local discrepancy

$$d_X(A(E,F)) = \left| \frac{SAT_X(E,F)}{n} - (F_x - E_x)(F_y - E_y) \right|. \tag{6.10}$$

If grid point E defines the first corner of the rectangle, then every grid point F, $F_x \geq E_x$, $F_y \geq E_y$, is a valid opposite corner, thus a geometry test is performed. This is implemented by starting s^2 concurrent threads corresponding to the grid points E and calculating the local discrepancy for each valid F per thread, which leads to a total of $(s-1)^2 s^2/4$ geometry tests.

The results of the rectangle discrepancy tests for $s = 500$ are shown in Figure 6.4 in logarithmic scale. The asymptotic behavior of the rectangle discrepancies for an increasing n is similar to the behavior of the star discrepancies, but the slope of the Halton samples' discrepancy is less steep. Still, Halton samples can be considered the best choice for approximating axis-aligned structures by a finite number of samples.



(a) Average rectangle discrepancy

(b) Worst-case rectangle discrepancy

Figure 6.4: Results of the rectangle discrepancy tests

Quadrilateral Discrepancy

An even more general class of geometric shapes is the class $\mathcal{A}_{Quadrilateral}$ of all convex quadrilaterals. $\mathcal{A}_{Quadrilateral}$ is a rotationally invariant superset of $\mathcal{A}_{Rectangle}$ in Ω that additionally includes all non-rectangular convex quadrilaterals. Its discrepancy test gives a good indication of how good arbitrarily shaped and oriented objects can be approximated by a given point set and is a more realistic estimate for the quality of samples used for rendering, for example. Each quadrilateral $A(E,F,G,H)$ is determined by the coordinates of its four corners $E = (E_x, E_y)$, $F = (F_x, F_y)$, $G = (G_x, G_y)$ and $H = (H_x, H_y)$. Its area can be calculated easily as the sum of two triangles spanned by the corners,

$$A_{\triangle EFG} = \frac{1}{2}|(F_x - E_x)(G_y - E_y) - (G_x - E_x)(F_y - E_y)|, \tag{6.11}$$

$$A_{\triangle EGH} = \frac{1}{2}|(G_x - E_x)(H_y - E_y) - (H_x - E_x)(G_y - E_y)|, \tag{6.12}$$

$$\lambda_2(A(E,F,G,H)) = A_{\triangle EFG} + A_{\triangle EGH}. \tag{6.13}$$

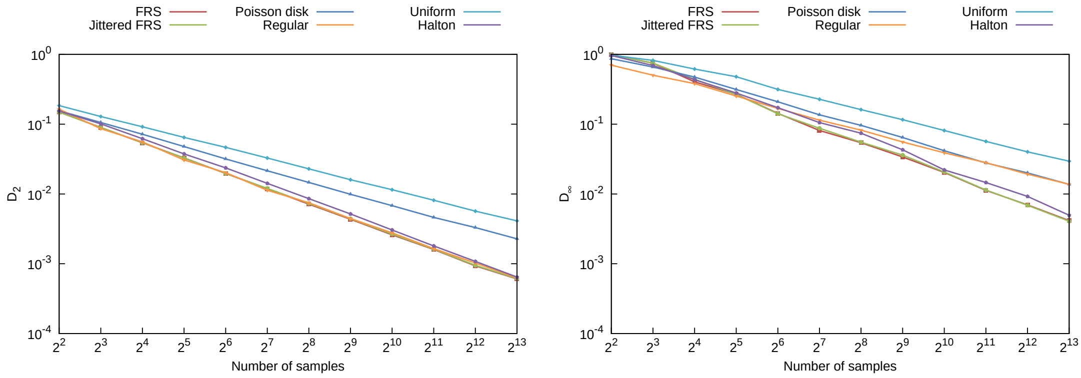
Likewise, the decision if a sample x_i is inside of $A(E, F, G, H)$ can be made by calculating the areas of four triangles sharing x_i as a common point and comparing their sum to the area of the quadrilateral:

$$n(i) = \begin{cases} 1, & A_{\triangle EFx_i} + A_{\triangle FGx_i} + A_{\triangle GHx_i} + A_{\triangle HEx_i} = A_{\triangle EFG} + A_{\triangle EGH}, \\ 0 & \text{otherwise}, \end{cases} \tag{6.14}$$

$$\|X \cap A\| = \sum_{i=0}^{n-1} n(i). \tag{6.15}$$

To perform the quadrilateral discrepancy test in feasible time, only a random subset of quadrilaterals is tested. However, during each of the m different test runs, the same subset is used for all different sample distributions to ensure equal test conditions. First, the four corner coordinates of a quadrilateral are placed randomly in Ω and a counterclockwise order E, F, G, H is assumed. The diagonals $\overline{EG}$ and $\overline{FH}$ are then intersected to check if the quadrilateral is concave, in which case it is discarded and the next set of random coordinates is tested. Otherwise, the point-in-quadrilateral test according to Equation 6.14 is performed.

The results of the quadrilateral discrepancy test for $s = 3000^2$ quadrilaterals per run are shown in Figure 6.5 in logarithmic scale. Again, the average discrepancies of FRS and jittered FRS are much closer to the results of Halton sampling than of Poisson disk sampling. For the class $\mathcal{A}_{Quadrilateral}$, FRS and jittered FRS have even lower average and worst-case discrepancies than Halton sampling, indicating a high suitability for spatially uniform sampling in general. However, it is apparent that the discrepancy does not account for the regularity of a sample distribution and the possibly resulting artifacts. This is why the suitability of any of the distributions for blue-noise sampling in particular cannot be inferred from these results alone.



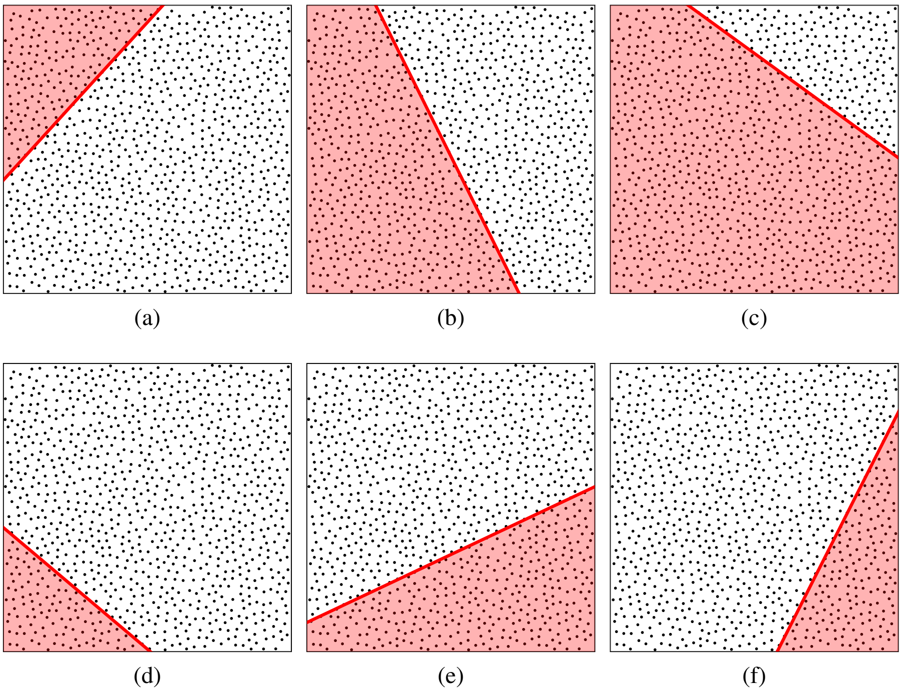
(a) Average quadrilateral discrepancy

(b) Worst-case quadrilateral discrepancy

Figure 6.5: Results of the quadrilateral discrepancy tests

Halfspace Discrepancy

The most general rotationally invariant class of geometric shapes used for the discrepancy is the class of halfspaces $\mathcal{A}_{Halfspace}$ of Ω partitioned by an arbitrary edge $\overline{EF}$, where $E = (E_x, E_y)$ and $F = (F_x, F_y)$ denote two points on the edge. There are six cases of partitions by edges that need to be considered in an implementation, as illustrated in Figure 6.6.


(a) (b) (c)

(d) (e) (f)

Figure 6.6: Space partitions by edges that need to be distinguished. Edges intersecting (a) the top and left border of $\hat{\Omega}$, (b) the top and bottom border, (c) the top and right border, (d) the left and bottom border, (e) the left and right border and (f) the bottom and right border.

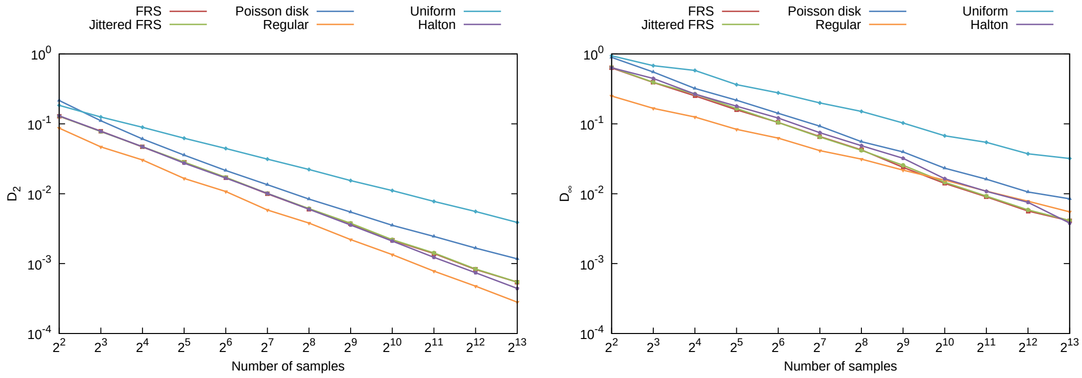
The local discrepancy of a halfspace only depends on the edge, so discretizing the borders of $\hat{\Omega}$ and testing each possible connection of border points according to the six cases is sufficient. The shape of the halfspace is the union of a triangle and a rectangle, its area $A(E, F)$ depends on the type of partition and can be calculated by either the sum of the individual areas or by its complement's area:

$$\lambda_2(A(E,F)) = \begin{cases} A_\triangle, & \text{case (a), (d) or (f),} \\ \min(E_x, F_x) + A_\triangle, & \text{case (b),} \\ 1 - A_\triangle, & \text{case (c),} \\ \min(E_y, F_y) + A_\triangle, & \text{case (e),} \end{cases} \qquad A_\triangle = \frac{1}{2}|(F_x - E_x)(F_y - E_y)|. \tag{6.16}$$

The number of samples in A is determined by the position of each sample x_i relative to the edge,

$$\|X \cap A\| = \sum_{i=0}^{n-1} n(i), \quad n(i) = \begin{cases} 1, & (E_x - F_x)(x_{i,y} - F_y) - (E_y - F_y)(x_{i,x} - F_x) > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{6.17}$$

As every unique pair of border points is considered for a space partition, a total of $6s^2$ different geometry tests have to be performed. The results of the halfspace discrepancy tests for $s = 3000$ per run are shown in Figure 6.7 in logarithmic scale. They match the results of Dobkin et al. [DEM96] to the effect that regular sampling has a significantly lower average halfspace discrepancy than random, Poisson disk or low-discrepancy sample distributions. Again, the average discrepancies of FRS and jittered FRS closely match the results of Halton sampling. Their worst-case discrepancies are even lower than the one of Halton sampling. In contrast to the previous results, jittered FRS has a slightly higher worst-case discrepancy than FRS.



(a) Average halfspace discrepancy

(b) Worst-case halfspace discrepancy

Figure 6.7: Results of the halfspace discrepancy tests

Circle Discrepancy

So far, only polygonal shapes have been considered for the discrepancy tests. Therefore, an obvious choice for another $\mathcal{A}$ is a class of shapes without straight line segments. When proposing the star discrepancy as a quality measure for sample sets, Shirley [Shi91] already considered that the class of circles in Ω, $\mathcal{A}_{Circle}$, could be an equivalent or even better measure. Its calculation is very easy and is – in contrast to the star and rectangle discrepancies – not prone to regular structures in the sample sets. Similar to the quadrilateral discrepancy, only a random subset of all circles in Ω is considered in the implementation. For each geometry test, three random numbers $r_1, r_2, r_3 \in [0, 1)$ are generated, of which two are used to define the random center $E = (r_1, r_2)$ of the circle $A(E, r_c)$. As the circle has to fit in Ω, its largest possible radius is calculated from the minimum distance of the center to the borders of Ω,

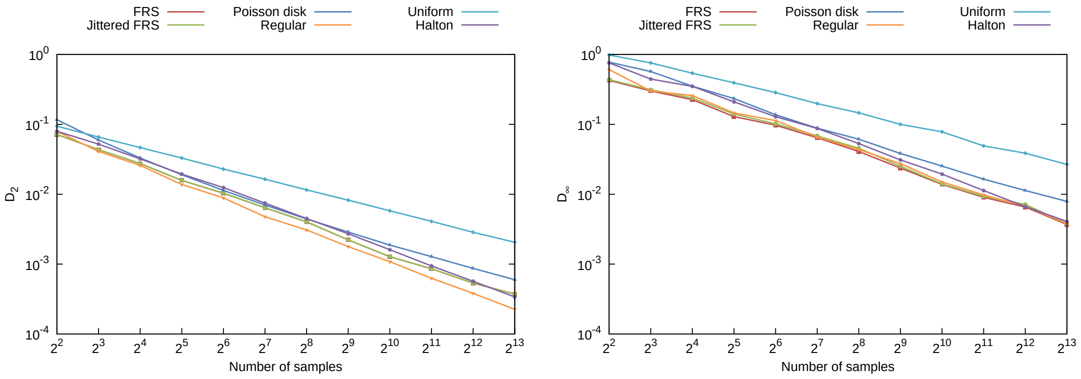
$$r_{max} = \min(r_1, 1.0 - r_1, r_2, 1.0 - r_2). \tag{6.18}$$

The actual radius of the circle is a random part of this distance, $r_c = r_3 r_{max}$. The computations of the area and the number of samples in $A(E, r_c)$ are trivial:

$$\lambda_2(A(E, r_c)) = \pi r_c^2, \tag{6.19}$$

$$\|X \cap A\| = \sum_{i=0}^{n-1} n(i), \quad n(i) = \begin{cases} 1, & \|x_i - E\| < r_c, \\ 0 & \text{otherwise.} \end{cases} \tag{6.20}$$

The results of the circle discrepancy tests for $s = 3000^2$ circles per run are shown in Figure 6.8 in logarithmic scale. As with the other generalized discrepancies, the regular sample distribution has a remarkably low average and worst-case discrepancy. It is followed by FRS and jittered FRS, which both have discrepancies slightly lower than Halton sampling and significantly lower than Poisson disk sampling.



(a) Average circle discrepancy

(b) Worst-case circle discrepancy

Figure 6.8: Results of the circle discrepancy tests

Summary

Discrepancy is an important measure for the quality of sample sets because it is directly linked to the sampling error. The more evenly spread the samples are in the sample domain, the better can an arbitrary integral be approximated by them. It is therefore generally desirable to minimize the discrepancy of a sample distribution to minimize the sampling error. However, the results above show that the discrepancy as a measure for spatial uniformity favors regular configurations, which cover Ω very evenly by design. This can be seen best for $\mathcal{A}_{Circle}$. The geometric classes $\mathcal{A}_{Star}$ and $\mathcal{A}_{Rectangle}$ do not show this behavior, which can be explained by the common axis-parallel alignment of the regular grid used as input. As the tested shapes are also axis-parallel, a slight change of the rectangle's area can have a large effect on the number of samples inside the rectangle. If an arbitrarily rotated regular grid or lattice had been used instead, the samples would have very low star and rectangle discrepancies, as shown by Matoušek [Mat10].

The problem with regular sampling is that it produces obvious artifacts, as opposed to stochastic sampling, which transforms the sampling error into uncorrelated noise. Although regular sampling leads to a low sampling error, the type of the error is more important than its magnitude. The best sample distribution is therefore not the one with the lowest discrepancy, but the one with the lowest discrepancy that does not lead to visible sampling artifacts. The latter is usually evaluated by a spectral analysis of the sample distribution, as done in Section 6.3. Thus, a conclusive statement regarding the sample distributions' quality cannot be made at this point. However, some general remarks can be made.

First of all, Halton sampling representing the *low-discrepancy sequences* has a low discrepancy, but only excels in the cases of $\mathcal{A}_{Star}$ and $\mathcal{A}_{Rectangle}$, for which it has been designed. If more general shapes are approximated, FRS and regular sampling are valid alternatives. It generally seems that the higher generalizations of geometric shapes, namely $\mathcal{A}_{Quadrilateral}$, $\mathcal{A}_{Halfspace}$ and $\mathcal{A}_{Circle}$, allow a more realistic estimation of a sample distribution's spatial uniformity, since the axis-parallel shapes can interfere with regular structures in the sample set.

Shirley [Shi91] already pointed out the surprisingly high star and rectangle discrepancy of Poisson disk sampling, which could be reproduced for more general cases as well. Although it has a high spatial uniformity, the asymptotic behavior of its discrepancy is very similar to the one of uniform random sampling. The samples have been created by dart throwing, so the sample sets are not necessarily maximal Poisson disk sets, which would mean that no further points can be added. This might lead to a higher discrepancy. Generally, it can be observed that sample distributions with a random component always have a higher discrepancy than deterministic ones.

The results of the FRS variants are much closer to the results of Halton sampling than of Poisson disk sampling. This is surprising because the visual and spectral (see Section 6.3) similarity of FRS to Poisson disk sample sets is much higher than to Halton sample sets. It can be assumed that this is due to the higher regularity of FRS samples originating in the matrix thresholding and the overall deterministic generation. When not using axis-parallel shapes, adding a random component to the samples by jittering seems to impair the discrepancy. For the more general shapes, the worst-case discrepancies of FRS and jittered FRS are even lower than the discrepancy of Halton sampling, indicating a high suitability for spatially uniform sampling.

Density

Because of its simplicity, the density ρ_X of sample set X is an increasingly popular measure for the quality of a sample distribution. Lagae and Dutré [LD08] proposed the calculation of the density based on the minimum Euclidean distance between any two samples in X, which only returns meaningful results if X is distributed according to a uniform importance function. By using the minimum transformed differential d_{min} after the differential domain transform as proposed by Wei and Wang [WW11], the density can also be calculated for non-uniform sample distributions:

$$\rho_X := \frac{d_{min}}{d_{max}}, \quad d_{max} = \sqrt{\frac{2}{\sqrt{3}n}}. \tag{6.21}$$

As already explained in Section 2.2, one problem with ρ_X is the interpretation of values regarding blue-noise sampling. It is favorable to have a sample distribution with samples far away from each other to cover the sample domain with as few samples as possible, but without exhibiting any regularity. A density $\rho_X \in [0.65, 0.85]$ is generally considered a good density for blue-noise sampling, where a lower value indicates randomness and a higher value indicates regularity. However, there exist algorithms [SHD11] to generate blue-noise sample sets with $\rho_X \approx 0.93$, which is very close to the density of a regular grid. At least for high values, the density seems to be ambiguous. This is why a meaningful conclusion regarding the different point sets' quality based on the density alone cannot be made. Still, the behavior of the density over a set of different importance functions can be used to investigate the constancy of a distribution algorithm.

The density values for the different sample distributions in Table 6.1 have been calculated using Equation 6.21 and averaged over 10 different sample sets, $n \approx 1024$, per distribution and importance map. By averaging over m sample sets, the average density

$$\rho := \frac{1}{m} \sum_{i=0}^{m-1} \rho_{X_i}. \tag{6.22}$$

of a sample distribution can be approximated. The reference sample distribution with a target density $\rho = 0.7$ is dart throwing in the uniform case and relaxation dart throwing in the non-uniform case. For the non-uniform case, the uniform random, regular, Halton and Poisson disk sample sets have been warped using HSW. It can be seen that independent from the input distribution, HSW is not able to preserve the density of the samples when warping non-uniformly. The density of FRS is very similar for all importance maps and ranges in the lower end of the interval suggested by Lagae and Dutré. The density of jittered FRS tends to be slightly lower than the density of FRS, which can be explained by the small amount of randomness added by jittering.

	Uniform	Constant	Gaussian Blob	Linear Gradient	Environment
Reference	0.700	0.700	0.700	0.700	0.700
FRS	0.650	0.681	0.643	0.662	0.644
Jittered FRS	0.621	0.668	0.634	0.617	0.661
Poisson Disk	0.700	0.700	0.261	0.370	0.142
Random	0.014	0.003	0.004	0.001	0.008
Regular	0.930	0.930	0.318	0.439	0.357
Halton	0.170	0.066	0.149	0.138	0.041

Table 6.1: Sample distribution densities ρ for different importance maps, averaged over 10 sample sets each

6.3 Spectral Analysis of Sample Distributions

Fourier Analysis

For uniform sampling, the Fourier analysis is the most important tool for the evaluation of a sample distribution's quality. Regularities and hidden structures are apparent in the power spectrum of a sample distribution. The ideal power spectrum for alias-free sampling is assumed to be the blue-noise spectrum as introduced in Section 2.2. As the dart-throwing sample distribution has such a spectrum, it is used as a reference for the quality of uniform sample sets. Thus, by comparing the estimated power spectrum of any sample distribution to the power spectrum of dart throwing, a statement about its blue-noise properties can be made. Additionally, the radial mean and anisotropy of the spectra are compared. As all power spectra in this thesis have been estimated with 10 different sample sets, the anisotropy of an artifact-free sample distribution should be -10 dB. The results of uniform random and regular sampling give an impression of how randomness and regularity manifest in the power spectrum of a sample distribution. Halton sampling is included as a representative of sampling with low-discrepancy sequences, which is also used as input for HSW. It is obvious that the distribution quality of a warped sample set largely depends on the quality of the input sample set. Therefore, the sampling method cannot perform better in the adaptive than in the uniform case.

The estimated power spectra of FRS and jittered FRS ($\sigma = 8$) are shown in Figure 6.9 together with the results of the other sample distributions for $n = 1024$. It is apparent that the power spectra of both FRS variants have distinct blue-noise characteristics, which indicates a high suitability for alias-free sampling. The spectra show the typical annuli of low and high energy and are very similar to the reference. The transition between the low-energy annulus around the center to the first high-energy annulus is slightly sharper in the case of FRS and jittered FRS, the anisotropy is lower. The most interesting observation is that both variants show a higher randomness than dart throwing, despite the inherent regularity of the thresholding process. This can be seen best when comparing the radial means of FRS and dart throwing. While the first peak is almost identical in both plots, the second and third peak corresponding to the second and third high-energy annuli of the power spectra are significantly weaker in the case of FRS. In its power spectrum, the third annulus is barely distinguishable from uncorrelated noise. This observation coincides with the density ρ of FRS (see Table 6.1), which is slightly lower than the density of dart throwing and also indicates a slightly more random distribution.

A second observation made in the spatial analysis is the similar behavior of the discrepancy of FRS and Halton sampling. This is surprising, as the associated power spectra have no similarity at all. Halton sampling is highly anisotropic and exhibits regular patterns which are very clearly visible in both the power spectrum and the sample set itself. However, it has to be noted that Halton sampling is not intended for blue-noise sampling in the first place. Therefore, the qualitative comparison of dart-throwing and Halton samples as possible input samples to HSW is only made to emphasize this fact. Regular and uniform random sampling, two further choices for fast sampling, are only included for completeness' sake, as they are well known to be poor choices for blue-noise sampling.

A last important observation is that there is no significant difference between the results of FRS and jittered FRS. For each of the 10 sample sets, the same random window offsets have been

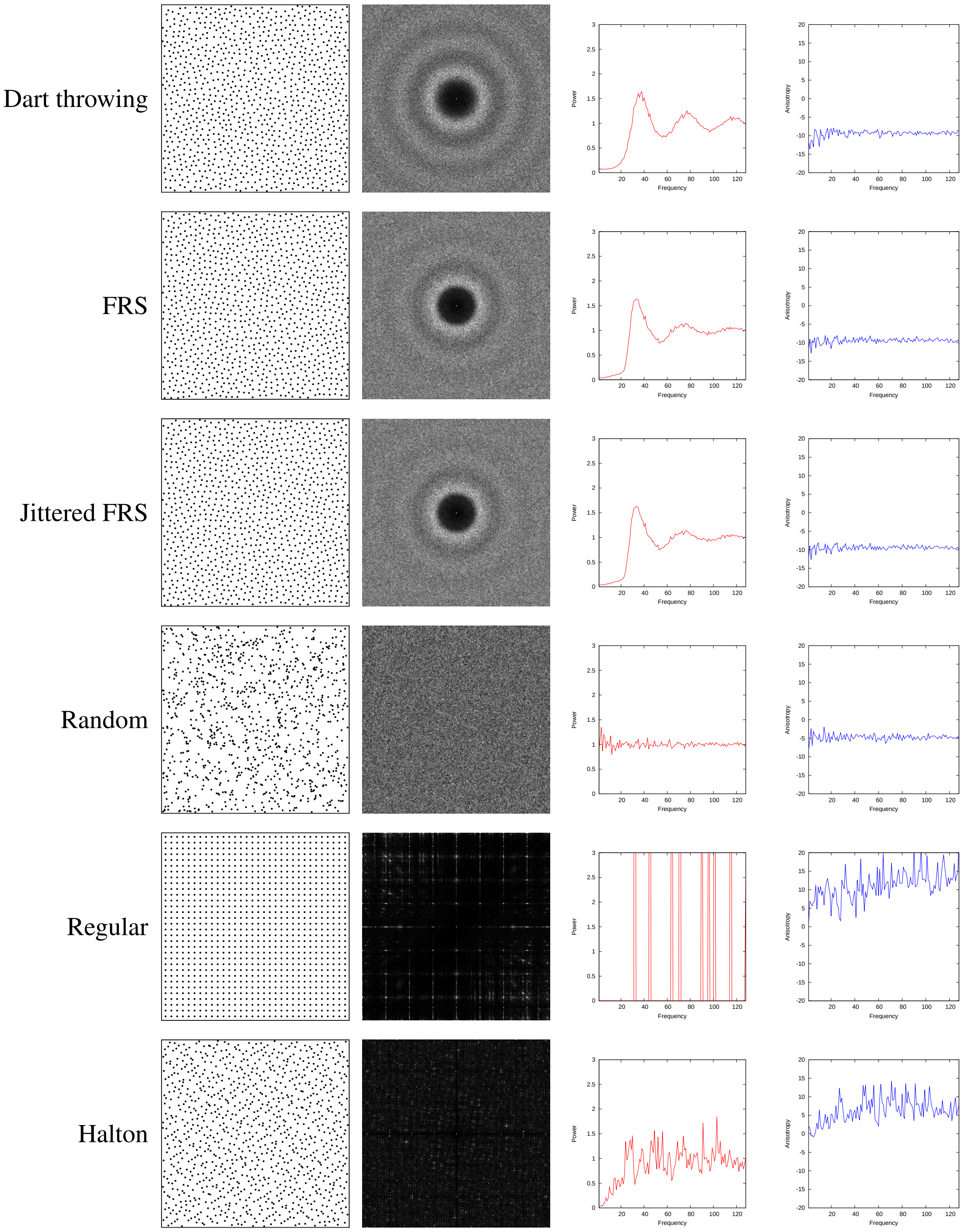


Figure 6.9: Estimated power spectra with radial mean (red) and anisotropy (blue) of different sample distributions for uniform sampling

used for both variants to account for the subpixel jittering only, which is why their exemplary sample sets look almost the same. The Fourier analysis suggests that both algorithms produce samples of high quality and are equally well suited for blue-noise sampling.

Differential Domain Analysis

In order to evaluate the quality of a sample distribution for adaptive sampling, a differential domain analysis is performed. As with the Fourier analysis, the results of this method can be shown in a spectral plot. Although the measured magnitude is not necessarily *power*, the term *power spectrum* is used for the spectral plot in analogy to the Fourier analysis. The quality of a sample distribution is assessed by comparing its power spectrum to the power spectrum of relaxation dart throwing. Distinctive patterns are identified and interpreted according to the reference spectra given by Wei and Wang [WW11]. The optimal spectrum is isotropic, has a low-energy annulus around the center followed by a narrow high-energy annulus corresponding to the minimum differential of any two samples and then transitions into noise. As the distance varies according to the importance map and is the lowest at the most important region, the radii of the annuli also vary between different importance maps. Their behavior, however, should be the same.

The estimated power spectra of FRS and jittered FRS for different importance maps are shown in Figure 6.10 together with the results of HSW with different input sample distributions. Since uniform sampling is just a special case of adaptive sampling, the results of uniform sampling are identical to those of adaptive sampling with a constant importance map and are omitted here. All power spectra have been estimated with 10 different sets of 1024 samples each. For FRS and jittered FRS, the sparsity $\sigma = 8$ has been chosen. The DDA has been performed with the tool provided by Wei [Wei11], using the parameters used in the paper by Wei and Wang [WW11] for an easy comparison. Specifically, the computation kernel κ is a Gaussian kernel, the neighborhood size for the range selection is $\epsilon = 12$.

The DDA results extend the results of the Fourier analysis to the general case of adaptive sampling. The power spectra of relaxation dart throwing and FRS are very similar for all importance maps. This again suggests the use of FRS for blue-noise sampling as a fast alternative to relaxation dart throwing. Also, it shows that FRS is suited very well for adaptive sampling, as its good spectral properties are preserved locally when adapting to a non-uniform importance function. The power spectrum of FRS also clearly shows the low-energy annulus around the center followed by a high-energy annulus, which is slightly fringed outwardly as compared to the sharp transition in the reference spectrum. This can best be seen when comparing its radial mean for uniform sampling with the reference, as shown in Figure 6.11. Once again, this indicates a higher variation of the differentials and thus more randomness of FRS as compared to Poisson disk sampling. A second difference to the reference spectrum is the clearly visible second peak in the radial mean corresponding to the double of the minimum differential. In the power spectrum, this shows as a second annulus of high energy around the first one.

While FRS seems to be generally well suited for uniform and adaptive sampling, HSW struggles to preserve the quality of an input sample distribution when warping. It is obvious that uniform random and regular sampling are not useful for blue-noise sampling. The interesting observation is that the power spectra look different for each importance map, although the

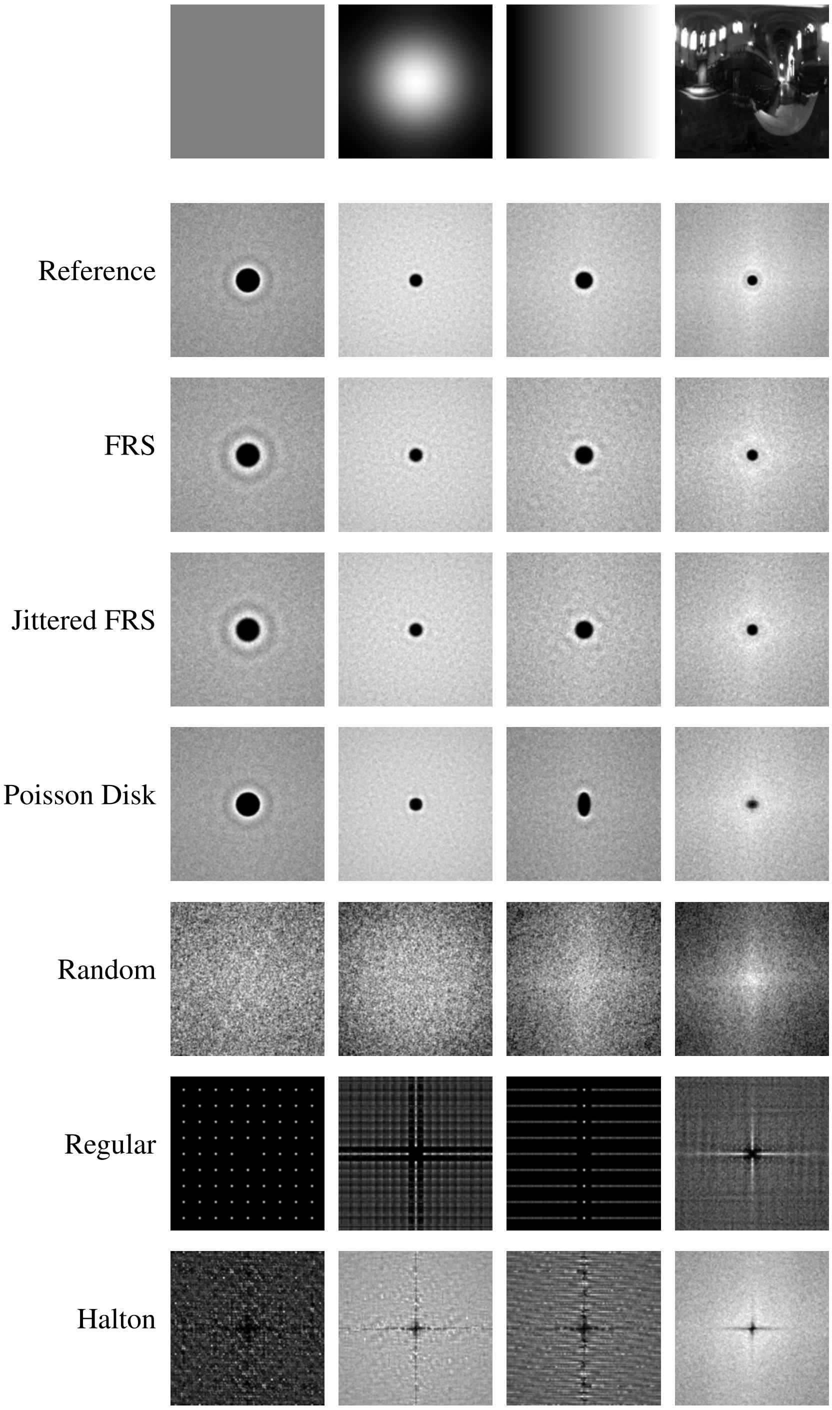


Figure 6.10: Estimated DDA power spectra of different sample distributions for adaptive sampling. The reference sample distribution is relaxation dart throwing. Random, regular, Halton and Poisson disk indicate the distribution of input samples for HSW.

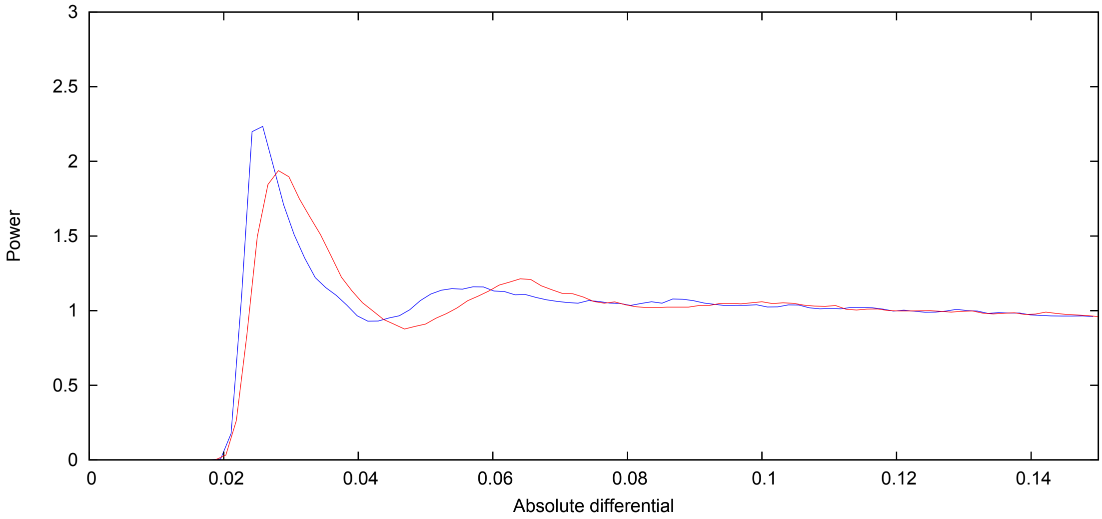
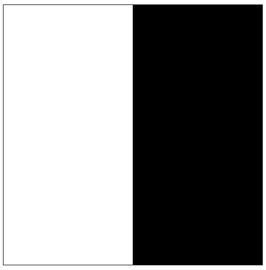


Figure 6.11: Radial means of relaxation dart throwing (blue) and FRS (red)

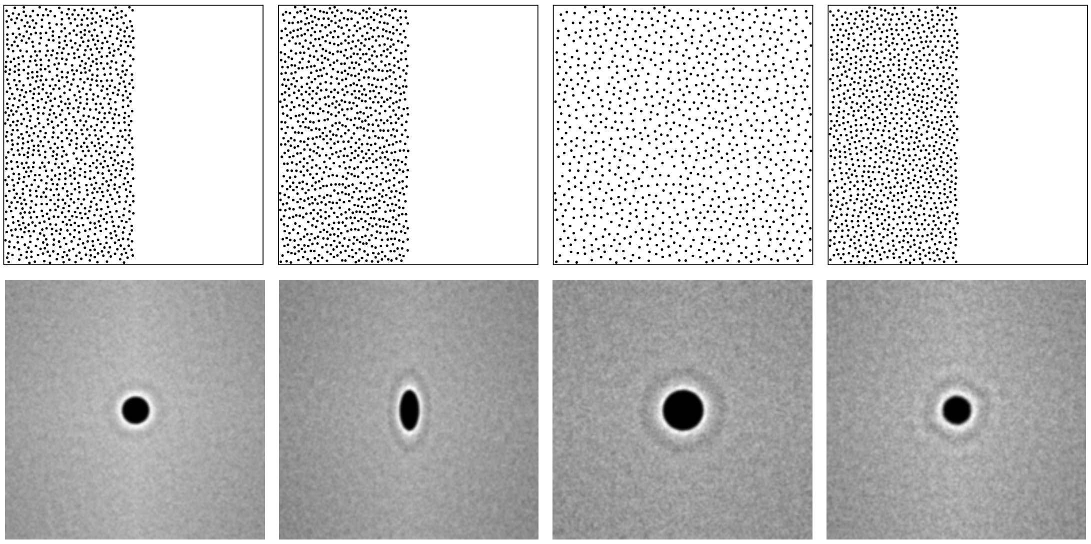
differential domain analysis compensates for the importance map itself. Thus, the differences in the power spectra stem from the sample warping itself, as already indicated by the varying densities in Section 6.2. When looking at the results of HSW with Halton samples, the inability to preserve the distribution properties may seem even desirable. At least the power spectra for the Gaussian blob and the environment importance maps are much closer to the reference than in the uniform case, although regular structures are clearly visible. With the results of Poisson disk input samples, however, it becomes obvious that this is a major shortcoming of HSW. Even with blue-noise input samples, adaptive blue-noise sampling with HSW is impossible. The reason for this is that warping only works well when the importance map is isotropic. Otherwise, the isotropic sample set is distorted anisotropically. In Figure 6.12, this is emphasized with an artificial, highly anisotropic importance function. Although the input sample distribution has optimal blue-noise properties, the sample distribution after warping is highly anisotropic. FRS, in contrast, is able to achieve a result close to the reference.

Summary

The spectral analysis has revealed that FRS sample distributions have blue-noise properties and are well suited for alias-free sampling, confirming the results of the spatial analysis. Their power spectra indicate a slightly higher noise than in Poisson disk distributions, but no isolated peaks of high energy or any other anisotropic artifacts could be observed. Although all samples are obtained from a matrix and thus have discrete positions, this does not impair the distribution quality as long as the sparsity is set to a sufficiently large value, i.e., $\sigma \geq 8$. In this case, it is also unnecessary to jitter the obtained samples. FRS is able to preserve the distribution quality even in adaptive distributions, which makes it very suitable for sampling with arbitrary importance functions. With HSW, in contrast, the quality of the warped sample distributions is impaired by anisotropic distortions even if HSW is used with Poisson disk samples as input.



Importance Map



Reference HSW HSW Input Samples FRS

Figure 6.12: Adaptive sampling with a highly anisotropic importance map

6.4 Performance Analysis

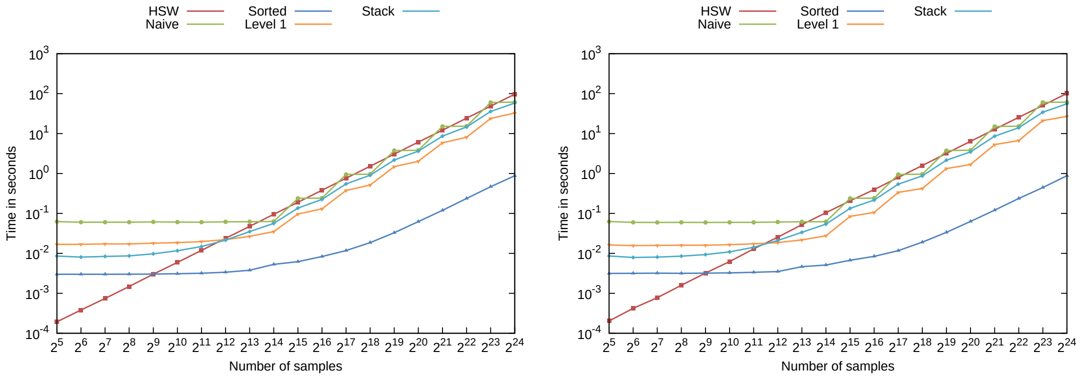
All used sampling algorithms have been implemented in the same application written for the .NET Framework 4.0, using OpenCL 1.2 for the GPU implementations. The benchmarks presented in this section have been obtained by counting the CPU ticks between the start and the end of a sampling task and dividing them by the CPU frequency. For the GPU implementations, the elapsed CPU ticks from the enqueuing of the kernel (*clEnqueueNDRangeKernel*) to the arrival of the finish event (*clWaitForEvents*) have been counted. The benchmarks should not be considered accurate measurements of time spans, but numeric values related to time spans that are only meaningful when compared to each other. The tests have been performed in a 64-bit environment on a system with an Intel Q6600 2.4 GHz quad-core processor and an NVIDIA GeForce GTX 680 video card. The CPU sampling implementations work with double precision, the GPU implementations with single precision.

In the following, the benchmarks of all FRS implementations are compared to HSW with Halton points, the state of the art of adaptive sampling in real time, for the four different importance maps depicted in Figure 6.1.

Single-Threaded CPU Implementations

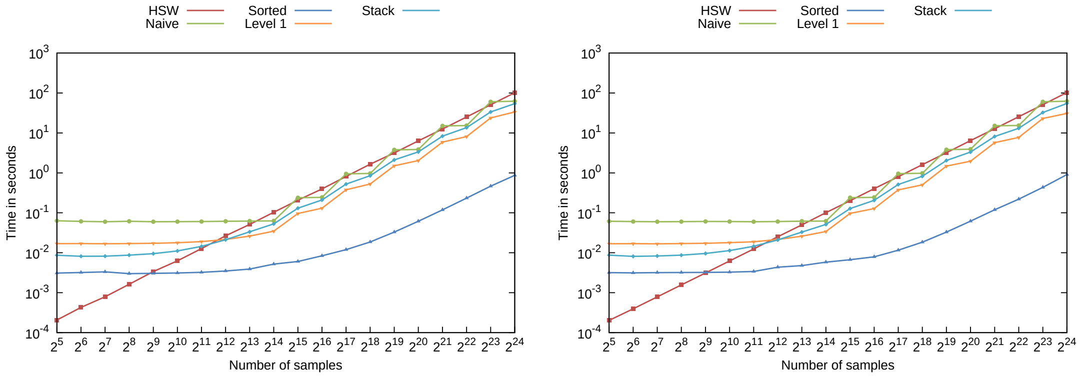
Samples	`Naive`	`Level 1`	`Stack`	`Sorted`	HSW
2^5	0.062	0.016	0.008	0.003	0.000
2^6	0.060	0.016	0.008	0.003	0.000
2^7	0.059	0.016	0.008	0.003	0.000
2^8	0.060	0.016	0.008	0.003	0.001
2^9	0.060	0.017	0.009	0.003	0.003
2^{10}	0.060	0.017	0.011	0.003	0.006
2^{11}	0.060	0.018	0.014	0.003	0.012
2^{12}	0.061	0.020	0.021	0.003	0.025
2^{13}	0.061	0.025	0.033	0.004	0.050
2^{14}	0.062	0.032	0.053	0.005	0.100
2^{15}	0.238	0.093	0.132	0.006	0.201
2^{16}	0.243	0.123	0.213	0.008	0.393
2^{17}	0.946	0.365	0.530	0.011	0.793
2^{18}	0.966	0.491	0.862	0.018	1.589
2^{19}	3.772	1.449	2.119	0.033	3.149
2^{20}	3.850	1.920	3.440	0.062	6.304
2^{21}	15.078	5.694	8.417	0.119	12.555
2^{22}	15.317	7.657	13.820	0.231	25.044
2^{23}	60.148	22.944	33.950	0.452	50.233
2^{24}	61.674	30.965	55.637	0.883	100.434

Table 6.2: Average time consumption of the CPU implementations in seconds



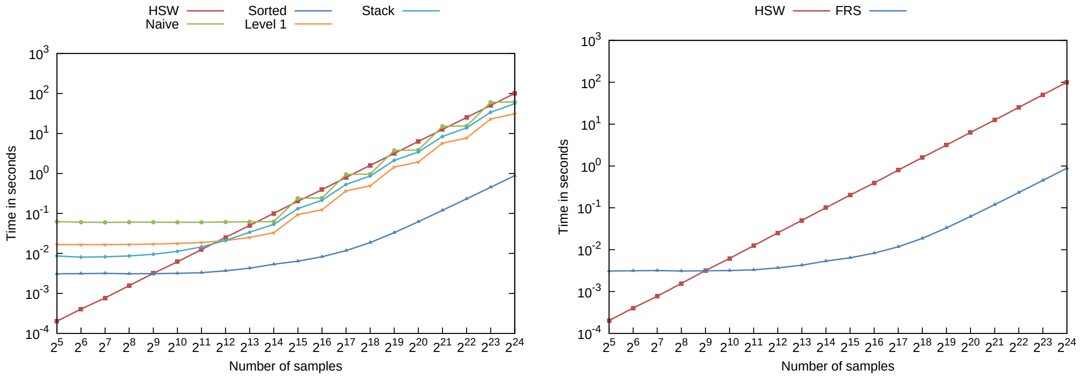
(a) Constant

(b) Gaussian Blob



(c) Linear Gradient

(d) Environment



(e) Average

(f) Average, `Sorted` and HSW only

Figure 6.13: Time consumption of the FRS CPU implementations `Naive`, `Sorted`, `Level 1` and `Stack` compared to HSW for different importance maps

The benchmarks of the CPU implementations on the test system have been obtained by averaging the elapsed CPU cycles of 100 runs and are shown in Figure 6.13 in logarithmic scale. The plots (e) and (f) have been created by averaging the results of the four importance maps. The average values are also presented in Table 6.2. An important observation is that the results for all four importance maps are almost identical, meaning that the performance of both HSW and FRS is independent from the underlying importance function. Up until $n = 2^9$ samples, HSW is the best among the tested algorithms. Its behavior is very predictable, as its computational effort grows linearly with the number of samples. The FRS implementations, in contrast, show a transition from constant to linear growth and – excluding `Sorted` – also a distinctive oscillating behavior.

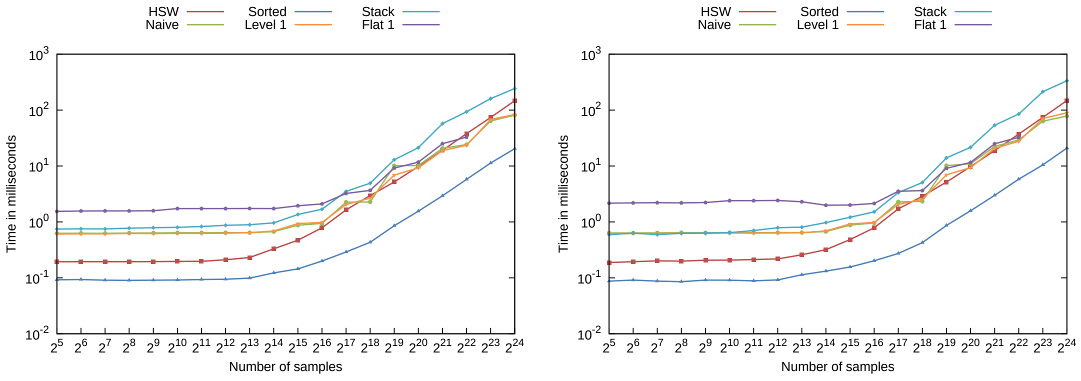
The runtime of FRS mainly depends on the number of subpixels rather than the number of samples itself, which in turn is directly related to the window size S_W and to the sparsity σ. In order to prevent sampling from being too regular, σ has been set to the realistic value of 8 for the performance tests. For the importance maps of size $S_I = 128 \times 128$, this results in a minimum window size $S_W = 1024$. Following Equation 4.6, a maximum of $n = 2^{14}$ samples can be generated for this window size. This is why the computational effort of the FRS implementations is almost constant up to this number of samples. For higher n, a larger window has to be used, which explains the stair-stepping in the results of the `Naive`, `Level 1` and `Stack` implementations.

The `Level 1` and `Stack` implementations perform slightly better than `Naive` because they rely on the minmap. Its hierarchical evaluation slightly reduces the number of necessary comparisons for thresholding. However, the use of a stack prevents an efficient use of caches and trades the relatively cheap thresholding operations for more expensive write accesses to the memory, which is why the overall performance of `Stack` is worse than the one of `Level 1`. The most efficient FRS implementation on the CPU is `Sorted`, which is also significantly faster than HSW for $n > 2^9$. As can be seen in Table 6.2, `Sorted` FRS was able to create over 16 million samples on the CPU in less than one percent of the time it took HSW.

GPU Implementations

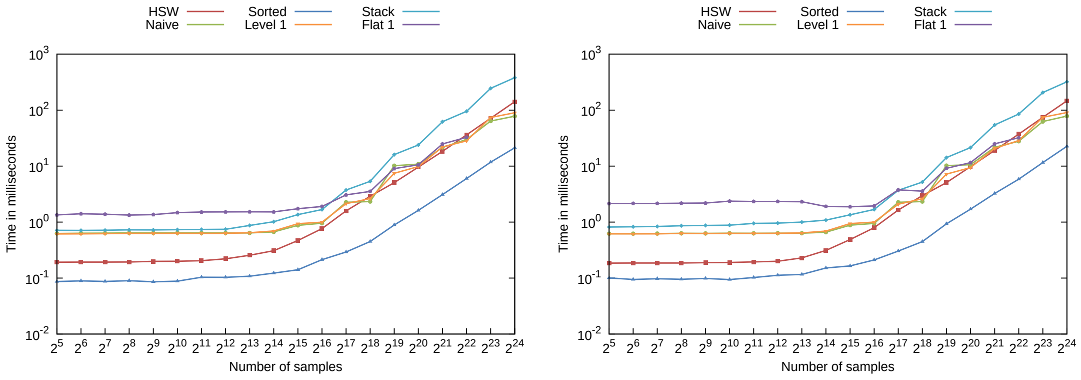
The benchmarks of the GPU implementations on the test system have been obtained by averaging the elapsed CPU cycles of 1000 runs and are shown in Figure 6.14 in logarithmic scale. Again, an average runtime is provided, the numeric values are shown in Table 6.3. Additionally to the FRS implementations tested in the CPU case, the `Flat 1` implementation has been tested on the GPU. However, the results show a rather poor performance of this implementation. Due to the large number of threads created per iteration, sampling was limited to $n \leq 2^{22}$ on the test system. Also, it has a large overhead due to the many OpenCL kernel calls. To a moderate extent, the overhead of kernel calls also influences the other implementations, which is much larger than the actual computation time for smaller n. However, as every implementation except `Flat 1` calls two kernels – sampling and jittering for FRS, Halton sample generation and warping for HSW –, this overhead is assumed to be approximately the same for the implementations.

In contrast to the CPU implementation, the computational effort of HSW now also seems to be constant for smaller values of n and then slowly transitions to the linear growth as observed above. A possible explanation for this is the low number of threads created in the case of small n.



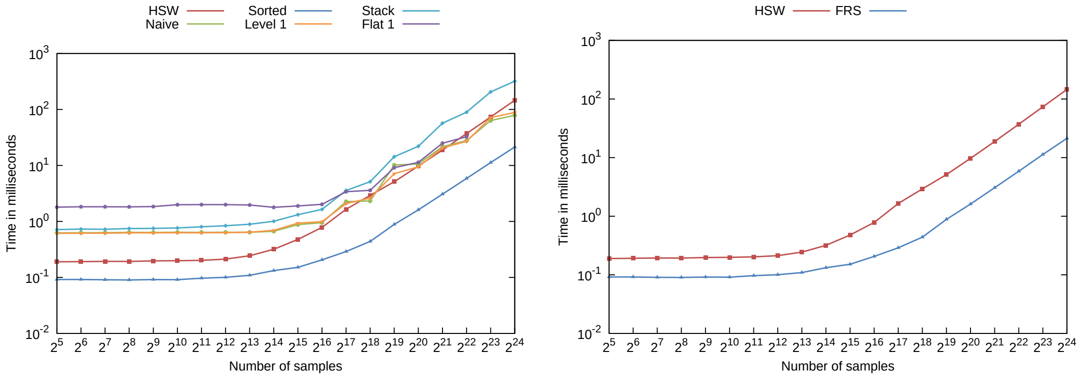
(a) Constant

(b) Gaussian Blob



(c) Linear Gradient

(d) Environment



(e) Average

(f) Average, `Sorted` and HSW only

Figure 6.14: Time consumption of the FRS GPU implementations `Naive`, `Sorted`, `Level 1`, `Stack` and `Flat 1` compared to HSW for different importance maps

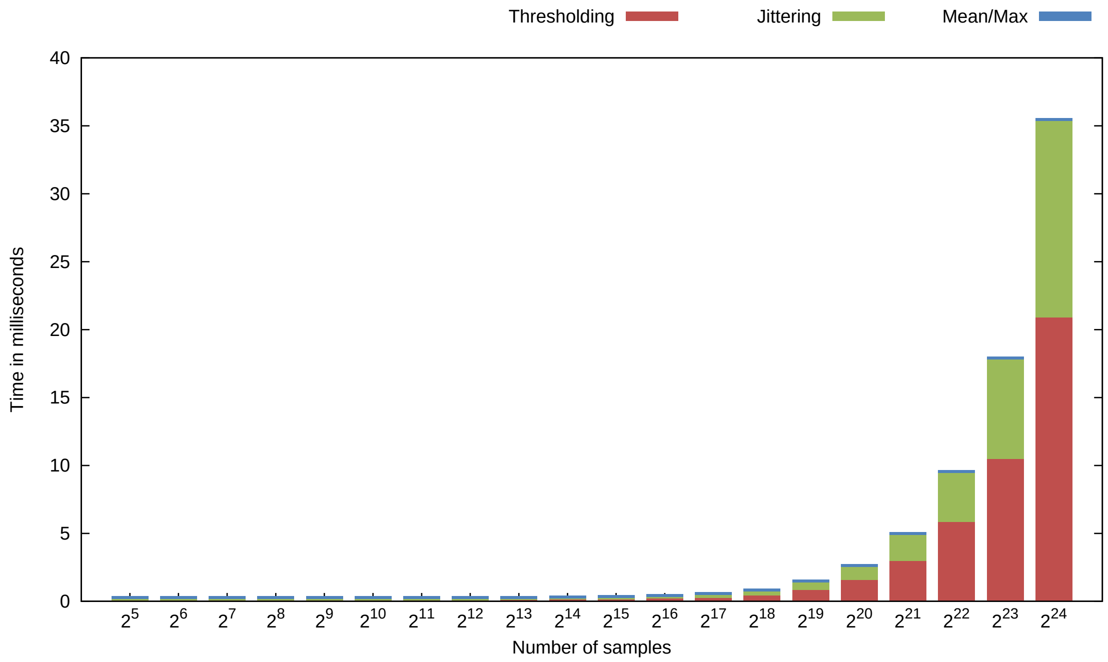
Samples	`Naive`	`Level 1`	`Flat 1`	`Stack`	`Sorted`	HSW
2^5	0.62	0.61	1.79	0.71	0.09	0.18
2^6	0.62	0.61	1.82	0.72	0.09	0.19
2^7	0.63	0.61	1.82	0.72	0.09	0.19
2^8	0.63	0.62	1.81	0.74	0.09	0.19
2^9	0.63	0.62	1.84	0.75	0.09	0.19
2^{10}	0.63	0.62	1.99	0.76	0.09	0.19
2^{11}	0.63	0.62	1.99	0.80	0.09	0.20
2^{12}	0.64	0.63	2.00	0.83	0.10	0.21
2^{13}	0.64	0.64	1.96	0.89	0.10	0.24
2^{14}	0.66	0.69	1.78	1.00	0.13	0.31
2^{15}	0.87	0.93	1.89	1.31	0.15	0.47
2^{16}	0.95	0.99	2.01	1.63	0.20	0.78
2^{17}	2.26	2.08	3.39	3.57	0.28	1.64
2^{18}	2.31	2.61	3.59	5.12	0.43	2.92
2^{19}	10.15	7.12	9.10	14.27	0.88	5.11
2^{20}	10.64	9.45	11.34	22.01	1.61	9.72
2^{21}	21.58	20.57	24.97	56.84	3.06	18.83
2^{22}	27.78	26.98	32.59	89.56	5.85	37.14
2^{23}	63.34	71.62		206.06	11.27	73.49
2^{24}	78.88	88.31		319.48	21.31	145.53

Table 6.3: Average time consumption of the GPU implementations in milliseconds

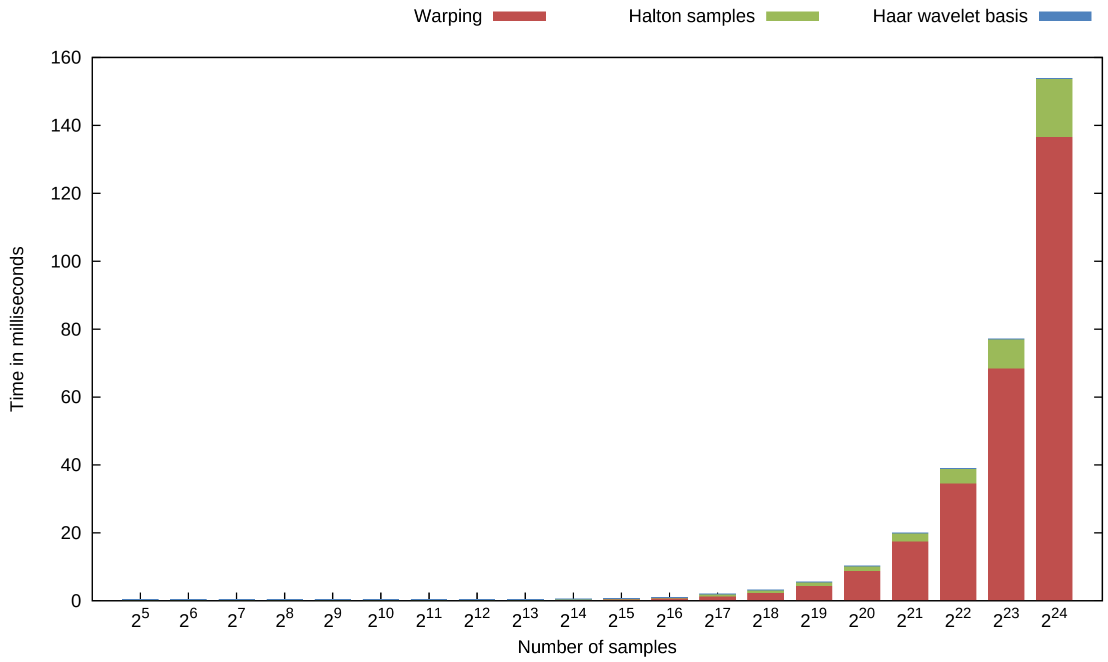
For each sample to be warped, an own thread is created and enqueued. If there are less threads than the GPU can execute concurrently, warping takes just as long as the slowest individual thread. Once the number of samples exceeds this limit, threads have to wait for execution and the overall execution time of warping increases with n.

Although this behavior also occurs in the case of the FRS implementations, it is not clearly visible because the sparsity $\sigma = 8$ leads to a constant number of subpixels for small n anyways. This is why the results of the FRS implementations for both CPU and GPU look very similar. The major difference between them is that neither `Stack` nor `Level 1` are significantly faster than the `Naive` implementation on the GPU. The reason for this is the erratic read access to the global memory for hierarchical thresholding, which inhibits efficient caching. The `Stack` implementation additionally performs a lot of expensive write accesses, which is why it is even slower than the `Naive` implementation. The fastest among all tested implementations is `Sorted`, which is also significantly faster than HSW for all tested n. It is also the GPU implementation of FRS that needs the least memory, as only a small portion of the dither matrix is needed instead of the whole matrix for `Naive` or even the minmap hierarchy for `Level 1`, `Flat 1` and `Stack`.

Figure 6.15 illustrates the proportional time consumption of the different tasks of `Sorted` with $\sigma = 4$ and HSW with Halton points. The calculations of the Haar wavelet basis (HSW) and the mean and maximum value (FRS) of the importance map are included as well, although



(a) Jittered `Sorted`



(b) HSW with Halton points

Figure 6.15: Proportional time consumption of the different tasks for adaptively sampling the Gaussian blob importance map

they have to be performed only once if the importance map changes. The results show that for small n, these calculations are the most expensive tasks of sampling with both methods, although done on the GPU. It can also be seen that subpixel jittering for FRS takes more than a third of the overall computation time. For an improvement of the sampling quality it is therefore faster to threshold with a higher sparsity ($\sigma \geq 8$) than to apply subpixel jittering.

Summary

The results show that the best way to implement FRS for both CPU and GPU is with the `Sorted` approach. It is significantly faster than the other implementations and requires the least memory. Due to the ordered threshold values and the possible early exit from a thread, its runtime is more dependent on the actual number of samples rather than the number of subpixels. Therefore, its performance is stable even if the window size has to be increased and does not exhibit the stair-stepping behavior of the `Naive` implementation.

On the CPU, all FRS implementations are outperformed by HSW for small sample sets in terms of runtime, but are competitive for larger values of n. On the GPU, in contrast, `Sorted` performs significantly better than HSW for every n and is applicable in real time. However, only Halton samples have been considered as input to HSW in the performance tests, which do not allow blue-noise sampling at all. Using Poisson disk samples as input as done for the spectral analysis is not feasible in real time. Although generally slower than `Sorted`, the `Naive` implementation of FRS is also suitable for adaptive sampling in real time and comparable to HSW for larger sample sets in terms of runtime. It is very easy to implement and also the implementation that can most easily be extended to higher dimensions. `Level 1`, `Flat 1` and `Stack`, however, are inferior to `Sorted` in all aspects, at least in the two-dimensional case.

CHAPTER 7

Conclusion

Forced Random Sampling is a versatile algorithm for fast k-dimensional adaptive progressive sampling by thresholding a precomputed dither matrix. Sample distributions obtained from it have been shown to have spectral properties close to the optimal blue-noise spectrum of dart throwing and relaxation dart throwing, independent from the underlying importance function. In contrast to the state-of-the-art algorithm for real-time adaptive sampling, Hierarchical Sample Warping, FRS is also suited for anisotropic importance functions. The spatial analysis of FRS further revealed that its discrepancy is similar to the discrepancy of the Halton low-discrepancy sequence, which means that FRS is well suited for general sampling applications and not limited to blue-noise sampling.

The implementation of FRS is very simple and does not require any complex calculations at runtime. The one-time generation of the Forced Random Dithering matrix is computationally expensive, but conceptually very simple as well. With little effort, the matrix can be restructured to sort the matrix elements block-wise, in which case the `Sorted` implementation can be used. This implementation has been shown to be the fastest implementation of FRS in the two-dimensional case. On the CPU, FRS scales much better than HSW and is therefore a valid alternative for generating a larger number of samples. As it is highly parallelizable, it is ideally suited for sampling on the GPU, in which case it outperforms HSW and makes real-time blue-noise sampling possible. For the few comprehensible parameters needed for the implementation, realistic values have been proposed in order to maximize the sampling performance without impairing the sample distribution quality. It has further been shown that jittering the obtained samples does not significantly influence the quality.

In summary, this answers the research question whether Forced Random Sampling is competitive to the state-of-the-art adaptive sampling algorithm in terms of quality and runtime performance. Not only is FRS capable of adaptive blue-noise sampling at a very high quality, which is not possible with Hierarchical Sample Warping, but it does so at an overall better performance. For real-time sampling, FRS is a serious alternative to existing algorithms. For offline sampling, FRS is a reasonable choice as long as runtime performance matters, but it cannot compete with the distribution quality of unbiased relaxation dart throwing and other Poisson disk implementations.

The analysis of sample distribution quality indicates that FRS differs from the assumed optimum of blue-noise sampling in being too random. A question that remains is whether this randomness can be reduced with a modified threshold matrix creation. Apart from trivial choices such as a completely random or regular dither matrix, it has not been investigated how changes to the force-field function influence the distribution properties of the obtained samples. Matrices with properties similar to those of Forced Random Dithering might be producible with related halftoning techniques such as electrostatic halftoning [SGBW10] in the discrete case.

A second problem that has not been addressed in this thesis is sampling with FRS from an arbitrary surface. Like other real-time sampling algorithms, FRS with the current threshold matrix creation only allows sample distributions on a plane without distortions. A generalization to distribute samples correctly on arbitrary manifold surfaces such as spheres or triangle meshes would be possible, but would require the force-field function to account for the geodesic distance between inserted dither values.

Finally, it has not been investigated how FRS performs in higher dimensions. While it can be assumed that the quality of sample distributions will be the same as in the two-dimensional case, no definite statement can be made on how the different possible implementations perform in terms of runtime. The `Naive` implementation can be extended to higher dimensions very easily, but the runtime complexity grows exponentially with the dimension. An extension of `Stack`, `Level N` and `Flat N` to higher dimensions appears to be unfeasible because of the code complexity. It can be assumed that `Sorted` scales rather well with higher dimensions, as thresholding itself is independent from the number of dimensions due to the reduction of the sorted values to a one-dimensional list. A more substantial challenge than the modification of the sampling step is the generation of the a higher-dimensional dither matrix itself. While the algorithm explained in Chapter 3 can generate two-dimensional matrices of practical size in a few hours, three-dimensional matrices would likely take days or weeks to generate.

Bibliography

[Bay73] Bryce E. Bayer. An Optimum Method for Two-Level Rendition of Continuous-Tone Pictures. *IEEE International Conference on Communication, Conference Record*, pages (26–11)–(26–15), 1973.

[BF88] Paul Bratley and Bennett L. Fox. Algorithm 659: Implementing Sobol's Quasi-random Sequence Generator. *ACM Transactions on Mathematical Software*, 14(1):88–100, 1988.

[Bri07] Robert Bridson. Fast Poisson Disk Sampling in Arbitrary Dimensions. In *ACM SIGGRAPH 2007 Sketches*. ACM, 2007.

[BSD09] Michael Balzer, Thomas Schlömer, and Oliver Deussen. Capacity-Constrained Point Distributions: A Variant of Lloyd's Method. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)*, 28(3):86:1–86:8, 2009.

[CAM08] Petrik Clarberg and Tomas Akenine-Möller. Practical Product Importance Sampling for Direct Illumination. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(2):681–690, 2008.

[CETC06] David Cline, Parris K. Egbert, Justin F. Talbot, and David L. Cardon. Two Stage Importance Sampling for Direct Lighting. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, pages 103–113. Eurographics Association, 2006.

[CJAMJ05] Petrik Clarberg, Wojciech Jarosz, Tomas Akenine-Möller, and Henrik Wann Jensen. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, 24(3):1166–1175, 2005.

[Cla12] Petrik Clarberg. *Hierarchical Variance Reduction Techniques for Monte Carlo Rendering*. PhD thesis, Dept. of Computer Science, Lund University, 2012.

[Coo86] Robert L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang Tiles for Image and Texture Generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, 22(3):287–294, 2003.

[CYC+12] Zhonggui Chen, Zhan Yuan, Yi-King Choi, Ligang Liu, and Wenping Wang. Variational Blue Noise Sampling. *IEEE Transactions on Visualization and Computer Graphics*, 18(10):1784–1796, 2012.

[Deb98] Paul Debevec. Light Probe Image Gallery. Personal website, 1998. Available from: http://www.pauldebevec.com/Probes/ (accessed 27 September 2013).

[DEM96] David P. Dobkin, David Eppstein, and Don P. Mitchell. Computing the Discrepancy with Applications to Supersampling Patterns. *ACM Transactions on Graphics*, 15(4):354–376, 1996.

[DG94] David P. Dobkin and Dimitrios Gunopulos. Computing the Rectangle Discrepancy. In *Proceedings of the 10th Annual Symposium on Computational Geometry*, pages 385–386. ACM, 1994.

[dGBOD12] Fernando de Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. Blue Noise through Optimal Transport. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012)*, 31(6):171:1–171:11, 2012.

[DH06] Daniel Dunbar and Greg Humphreys. A Spatial Data Structure for Fast Poisson-Disk Sample Generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):503–508, 2006.

[DHvOS00] Oliver Deussen, Stefan Hiller, Cornelius van Overveld, and Thomas Strothotte. Floating Points: A Method for Computing Stipple Drawings. *Computer Graphics Forum*, 19:40–51, 2000.

[DW85] Mark A. Z. Dippé and Erling Henry Wold. Antialiasing Through Stochastic Sampling. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pages 69–78. ACM, 1985.

[EMP+12] Mohamed S. Ebeida, Scott A. Mitchell, Anjul Patney, Andrew A. Davidson, and John D. Owens. A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions. *Computer Graphics Forum*, 31(2):785–794, 2012.

[GM09] Manuel N. Gamito and Steve C. Maddock. Accurate Multidimensional Poisson-Disk Sampling. *ACM Transactions on Graphics*, 29(1):8:1–8:19, 2009.

[HCTW07] Hao-da Huang, Yanyun Chen, Xing Tong, and Wen-cheng Wang. Incremental Wavelet Importance Sampling for Direct Illumination. In *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, pages 149–152. ACM, 2007.

[HDK01] Stefan Hiller, Oliver Deussen, and Alexander Keller. Tiled Blue Noise Samples. In *Proceedings of the Vision Modeling and Visualization Conference 2001*, pages 265–272. Aka GmbH, 2001.

[HS64] J. H. Halton and G. B. Smith. Algorithm 247: Radical-Inverse Quasi-Random Point Sequence. *Communications of the ACM*, 7(12):701–702, 1964.

[HSD13] Daniel Heck, Thomas Schlömer, and Oliver Deussen. Blue Noise Sampling with Controlled Aliasing. *ACM Transactions on Graphics*, 32(3):25:1–25:12, 2013.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel Prefix Sum (Scan) with CUDA. *GPU Gems*, 3(39):851–876, 2007.

[Jon06] Thouis R. Jones. Efficient Generation of Poisson-Disk Sampling Patterns. *Journal of Graphics Tools*, 11(2):27–36, 2006.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive Wang Tiles for Real-Time Blue Noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):509–518, 2006.

[Kel97] Alexander Keller. Instant Radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 49–56. ACM Press/Addison-Wesley Publishing Co., 1997.

[Kla00] R. Victor Klassen. Filtered Jitter. *Computer Graphics Forum*, 19(4):223–230, 2000.

[LD05] Ares Lagae and Philip Dutré. A Procedural Object Distribution Function. *ACM Transactions on Graphics*, 24(4):1442–1461, 2005.

[LD06] Ares Lagae and Philip Dutré. An Alternative for Wang Tiles: Colored Edges Versus Colored Corners. *ACM Transactions on Graphics*, 25(4):1442–1459, 2006.

[LD08] Ares Lagae and Philip Dutré. A Comparison of Methods for Generating Poisson Disk Distributions. *Computer Graphics Forum*, 27(1):114–129, 2008.

[Llo82] Stuart Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[LNW$^+$10] Hongwei Li, Diego Nehab, Li-Yi Wei, Pedro V. Sander, and Chi-Wing Fu. Fast Capacity Constrained Voronoi Tessellation. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 13:1–13:1. ACM, 2010.

[LTH$^+$13] Christian Luksch, Robert F. Tobler, Ralf Habel, Michael Schwärzler, and Michael Wimmer. Fast Light-Map Computation with Virtual Polygon Lights. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 87–94. ACM, 2013.

[Mat10] Jiří Matoušek. *Geometric Discrepancy. An Illustrated Guide.* Dordrecht: Springer, 2nd edition, 2010.

[MF92] Michael McCool and Eugene Fiume. Hierarchical Poisson Disk Sampling Distributions. In *Proceedings of the Conference on Graphics Interface '92*, pages 94–105. Morgan Kaufmann Publishers Inc., 1992.

[Mit87] Don P. Mitchell. Generating Antialiased Images at Low Sampling Densities. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–72. ACM, 1987.

[Mit91] Don P. Mitchell. Spectrally Optimal Sampling for Distribution Ray Tracing. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, pages 157–164. ACM, 1991.

[Nie92] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, 1992.

[ODJ04] Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. Fast Hierarchical Importance Sampling with Blue Noise Properties. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, 23(3):488–495, 2004.

[Ost07] Victor Ostromoukhov. Sampling with Polyominoes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3), 2007.

[PM88] Steven K. Park and Keith W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[PTG94] Werner Purgathofer, Robert F. Tobler, and Manfred Geiler. Forced Random Dithering: Improved Threshold Matrices for Ordered Dithering. In *Proceedings of the 1st IEEE International Conference on Image Processing*, volume 2, pages 1032–1035. IEEE, 1994.

[SCM02] Jonathan Shade, Michael F. Cohen, and Don P. Mitchell. Tiling Layered Depth Images. Technical Report 02-12-07, University of Washington, Department of Computer Science and Engineering, 2002.

[Sec02] Adrian Secord. Weighted Voronoi Stippling. In *Proceedings of the 2nd International Symposium on Non-Photorealistic Animation and Rendering*, pages 37–43. ACM, 2002.

[SGBW10] Christian Schmaltz, Pascal Gwosdek, Andrés Bruhn, and Joachim Weickert. Electrostatic Halftoning. *Computer Graphics Forum*, 29(8):2313–2327, 2010.

[SHD11] Thomas Schlömer, Daniel Heck, and Oliver Deussen. Farthest-Point Optimized Point Sets with Maximized Minimum Distance. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 135–142. ACM, 2011.

[Shi91] Peter Shirley. Discrepancy as a Quality Measure for Sample Distributions. In *Proceedings of Eurographics ’91*, pages 183–194. Elsevier Science, 1991.

[SHS02] Adrian Secord, Wolfgang Heidrich, and Lisa Streit. Fast Primitive Distribution for Illustration. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 215–226. Eurographics Association, 2002.

[Uli87] Robert Ulichney. *Digital Halftoning*. MIT Press, 1987.

[WCE07] Kenric B. White, David Cline, and Parris K. Egbert. Poisson Disk Point Sets by Hierarchical Dart Throwing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 129–132. IEEE, 2007.

[Wei08] Li-Yi Wei. Parallel Poisson Disk Sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, 27(3):20:1–20:9, 2008.

[Wei11] Li-Yi Wei. BlueMath. Public SVN repository of Li-Yi Wei, Revision 7, 1 September 2011. Available from: http://svn.liyiwei.org/public/BlueMath/src/ (accessed 13 September 2013).

[WW11] Li-Yi Wei and Rui Wang. Differential Domain Analysis for Non-Uniform Sampling. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, 30(4):50:1–50:10, 2011.

[XLGG11] Yin Xu, Ligang Liu, Craig Gotsman, and Steven J. Gortler. Capacity-Constrained Delaunay Triangulation for Point Distributions. *Computers and Graphics*, 35(3), 2011.

[XXSH11] Ying Xiang, Shi-Qing Xin, Qian Sun, and Ying He. Parallel and Accurate Poisson Disk Sampling on Arbitrary Surfaces. In *SIGGRAPH Asia 2011 Sketches*, pages 18:1–18:2. ACM, 2011.

[Yel83] John I. Yellott Jr. Spectral Consequences of Photoreceptor Sampling in the Rhesus Retina. *Science*, 221:382–385, 1983.