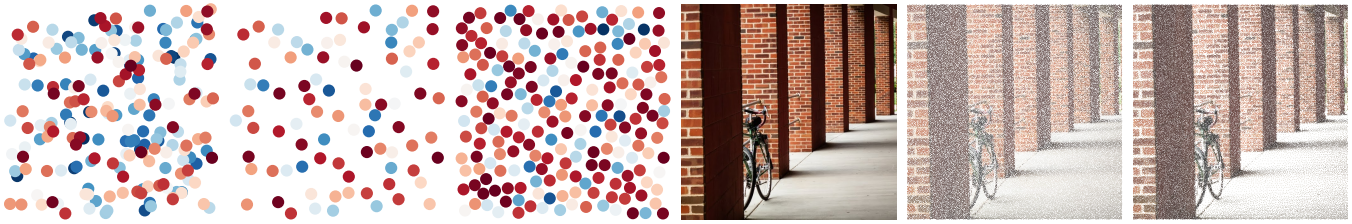
PixelPie: Maximal Poisson-disk Sampling with Rasterization

Cheuk Yiu Ip[1] M. Adil Yalçin[1] David Luebke[2] Amitabh Varshney[1]
University of Maryland, College Park[1] NVIDIA Research[2]



(a) (b) (c) (d) (e) (f)

Figure 1: *We perform maximal Poisson-disk sampling by rasterization and occlusion culling. First, we rasterize random disks of distinct depths (red is close, blue is far) in (a). Second, we cull the occluded disks to remove conflicting samples in (b). Third, we iterate this process on the empty regions to obtain a maximal Poisson-disk distribution in (c). (e) and (f) show uniform and importance sampling on (d).*

Abstract

We present PixelPie, a highly parallel geometric formulation of the Poisson-disk sampling problem on the graphics pipeline. Traditionally, generating a distribution by throwing darts and removing conflicts has been viewed as an inherently sequential process. In this paper, we present an efficient Poisson-disk sampling algorithm that uses rasterization in a highly parallel manner. Our technique is an iterative two step process. The first step of each iteration involves rasterization of random darts at varying depths. The second step involves culling conflicted darts. Successive iterations identify and fill in the empty regions to obtain maximal distributions. Our approach maps well to the parallel and optimized graphics functions on the GPU and can be easily extended to perform importance sampling. Our implementation can generate Poisson-disk samples at the rate of nearly 7 million samples per second on a GeForce GTX 580 and is significantly faster than the state-of-the-art maximal Poisson-disk sampling techniques.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Antialiasing I.4.1 [Computer Graphics]: Image Processing and Computer Vision—Digitization and Image Capture-Sampling;

Keywords: Poisson-disk sampling, GPGPU, dart throwing, maximal sampling

1 Introduction

This paper shows how to leverage the modern graphics hardware functions (shaders/depth culling) to solve a popular computer graphics problem, the Poisson-disk distribution. Poisson-disk distributions are highly desired in many computer graphics applications, such as anti-aliasing, stochastic ray tracing, object placement, and global illumination.

Poisson-disk distribution consists of random samples in a domain that are at least distance r apart. In the equations below, X is the set of samples i in domain D. We follow and extend the formulations of Gamito and Maddock [2009] and Mitchell et al. [2012].

$$\forall i \in X, \forall S \supseteq D : P(i \in S) = \int_S di \quad (1)$$

$$\forall i, j \in X, i \neq j : \|i - j\| \geq r \quad (2)$$

$$\mathcal{S}(X) = \{j \in D : \|i - j\| \geq r, i \in X\}, \mathcal{S}(X) = \emptyset \quad (3)$$

Equation 1 shows Poisson-disk distribution is uniform. The probability $P(i \in S)$ of a sample i falling in any subset S of D is proportional to the area of S, $\int_S di$.

Equation 2 shows the minimum distance requirement. Every pair of samples must be at least r away from each other. This eliminates the high frequency components from an unrestricted random sample distribution. If we draw disks of radius r around the samples, no disk should intersect with another sample.

Equation 3 represents the maximal coverage property. $\mathcal{S}(X)$ represents empty regions for inserting new samples without violating the minimum distance requirement. A distribution is maximal when $\mathcal{S}(X)$ is empty.

1.1 Poisson-disk Sampling on the Graphics Pipeline

One of the most popular approaches to generating a Poisson-disk sample set is the dart-throwing method. The traditional dart-throwing Poisson-disk sampling algorithms sequentially throw random darts on to the target domain and accept the new dart if it is r away from all the previously accepted darts. Since the acceptance of a new dart depends on the positions of all the previously accepted darts, this process is inherently sequential. Recent parallel approaches [Wei 2008; Ebeida et al. 2011; Ebeida et al. 2012] throw darts in a non-conflicting pattern and then meticulously fill in the remaining empty space using spatial data-structures such as quad-trees. If the darts are not thrown in a non-conflicting pattern then the darts that are closer than a distance r need to be identified and appropriately culled. This check is computationally intensive and also relies on spatial hierarchical data structures. In this paper, we present PixelPie, an alternative, simpler, technique that leverages the rasterization hardware on modern many-core GPUs to carry out this process in a highly parallel fashion. Unlike previous work, PixelPie does not rely on hierarchical data-structures for efficiency. Instead, its simplicity allows it to efficiently leverage the

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HPG 2013, July 19 – 21, 2013, Anaheim, California.
Copyright © ACM 978-1-4503-2135-8/13/07 $15.00

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

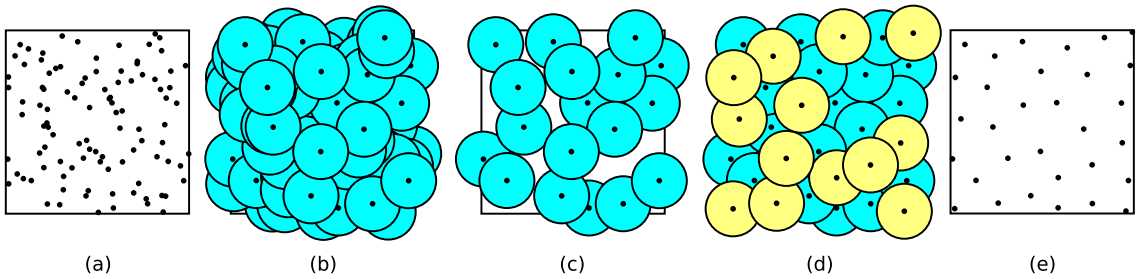


Figure 2: *Concept overview: (a) We throw random darts into the sample domain in (a) by drawing them as solid disks (b). We remove the disks with occluded center in (c) as they violate the minimum distance requirement. We repeat this process on the empty regions in (d) until we obtain a maximal Poisson-disk distribution in (e).*

highly-optimized graphics pipeline on GPUs. Figure 2 presents a overview of PixelPie. We use the graphics pipeline to perform 1) dart throwing and 2) conflict removal in parallel.

1. We throw random darts into empty regions by rasterizing them as circular disks into a depth map. We cast the problem of conflict checks as an occlusion detection problem. We use the depth test feature of the GPU to perform this efficiently.

2. We read the depth map to identify and remove the occluded disks. The unoccluded disk centers become part of the final sample set. We iterate this process on the empty regions to attain maximal coverage.

1.2 Contributions

1. We present a simple algorithm, PixelPie, for generating Poisson-disk distribution by casting it as an occlusion-culling problem using programmable shaders and rasterization.

2. PixelPie efficiently leverages the highly-optimized and parallel graphics operations on the modern GPUs. We attain maximal sampling coverage with a small number of iterations.

3. PixelPie is easily extensible to importance sampling by simply changing the radii of the disks based on the underlying importance map.

4. We present angular distribution of nearest neighbors to quantify the bias of a Poisson-disk distribution. We show how subpixel-centered samples can substantially reduce the bias for discrete sampling approaches.

PixelPie approximates circular disks with anti-aliased discrete disks and uses the GPU to efficiently perform parallel conflict checks. In comparison with recent algorithms [Bowers et al. 2010; Ebeida et al. 2011; Ebeida et al. 2012; Xiang et al. 2011] that use the GPU as a parallel computer with CUDA and OpenCL, PixelPie uses the hardware rasterizer to achieve a very high performance.

We hope that this will show the potential of casting other problems to the graphics hardware for significant performance gains.

The rest of the paper is organized as follows: Section 2 reviews the previous work on Poisson-disk sampling and geometric computing on GPUs. Section 3 presents the details of PixelPie. Section 4 discusses the quality of our results. Section 5 presents the performance and memory requirements of PixelPie.

2 Related Work

2.1 Poisson-Disk Sampling

Poisson-disk sampling has been an active area of research in computer graphics since the 1980s. Yellott [1982] showed that the distribution of photo receptors in the human eyes follows a Poisson-disk distribution. Dippe and Wold [1985] introduced this distribution in the stochastic sampling patterns to address aliasing artifacts. Cook [1986] showed that Poisson-disk sampling can be applied to a variety of computer graphics applications such as ray tracing and generating motion blur. Cook was also the first to show how to generate a bias-free Poisson-disk distribution by the dart-throwing method. Lagae and Dutre [2008] provide a nice survey of Poisson-disk sampling.

Several approximation schemes for Poisson-disk sampling have been proposed. Cook [1986] approximated the Poisson-disk sampling by jittering a grid of samples. Wang tiles [Lagae and Dutré 2005; Lagae and Dutré 2006a; Kopf et al. 2006] and polyominoes [Ostromoukhov 2007] are popular approximations but they introduce bias. The most recent schemes include replicating a sample spectrum [Kalantari and Sen 2012] and solving a constrained farthest-point optimization problem [Balzer et al. 2009; Chen and Gotsman 2012; de Goes et al. 2012].

Spatial data structures have been used to accelerate dart throwing by helping identify sparse regions. McCool and Fiume [1992] throw darts from large to small to cover the domain and then apply the Lloyd relaxation procedure to move the darts to the Voronoi cell centers. Jones [2006; 2011] used Voronoi diagrams and grids to select sparse regions for dart throwing. Dunbar and Humphreys [2006] used a new spatial data structure bounded by circular arcs, *the scalloped sector data structure*, to assist in throwing darts. White et al. [2007] used a quadtree to throw darts into empty regions only. Gamito and Maddock [2009] generalize the use of subdivision-assisted dart throwing to produce maximal samples in higher dimensions.

Recent work has introduced parallel dart throwing on GPUs. Wei [2008] introduced the first multi-resolution approach, however, it does not guarantee maximal and bias-free Poisson-disk samples. Ebeida et al. [2011; 2012] have shown maximal and bias-free sampling with CUDA by first sampling on a coarse grid then filling in the empty regions.

Poisson-disk distribution have been generalized to higher-dimensions [Gamito and Maddock 2009; Ebeida et al. 2012; Lagae and Dutré 2006b; Wei 2008]. Applications of high-dimensional Poisson-disk distributions include procedural texture and volumetric mesh generation for simulations. Poisson-disk sampling on mesh surfaces is used for remeshing, texturing, and object place-

ment. Cline et al. [2009] threw darts directly onto a mesh surface. Bowers et al. [2010] and Xiang et al. [2011] sampled a mesh surface on the GPU in parallel. Li et al. [2010] showed anisotropic sampling on a surface according a vector field.

Poisson-disk sampling has been extended to handle importance sampling. Ostromoukhov et al. [2004; 2007] used a hierarchy of tiles to perform importance sampling. Kopf et al. [2006] showed a real-time implementation with recursive Wang tiles. Wei [2010] presented a dart throwing approach for importance sampling on the GPUs. Kalantari and Sen [2011] threw darts as discretized disks of different radii. Fattal [2011] sampled according to different Gaussian kernels. The constrained farthest-point approaches [Balzer et al. 2009; Chen et al. 2012; de Goes et al. 2012] can also perform importance sampling. Zhou et al. [2012] showed how to sample according to different spectra.

In this paper we present a maximal and parallel dart-throwing approach that leverages the highly optimized graphics pipeline on the modern GPUs. We also present a simple extension to our approach that can handle importance sampling using the graphics pipeline.

2.2 General-Purpose GPU Computing

In the past decade a number of approaches have successfully leveraged the computing power of GPUs to solve general problems [Owens et al. 2007]. In addition to computer graphics problems, such as global illumination [Hachisuka 2005] and ray tracing [Parker et al. 2010], there have been notable successes in other, more traditional computation areas, such as geometric computing, signal processing, and physics-based simulation. Work on general-purpose GPU computing now spans the fields of scientific and high-performance computing. Much of this work is implemented on NVIDIA's CUDA platform [Nickolls et al. 2008] and various CUDA libraries, such as Thrust [Bell and Hoberock 2011], which implements basic parallel programming operations, such as map, reduce, scan, sort, and compact.

While most recent work in GPU computing simply treats the GPU as a massively-parallel processor, there is a long history of computing Voronoi diagrams and distance transforms with graphics pipelines. Hoff et al. [1999] used the fixed graphics pipeline to compute discrete Voronoi diagrams. Sigg et al. [2003] evaluated distance in the fragment shader and Sud et al. [2004] used the occlusion queries to cull out irrelevant primitives when computing distance fields. Fischer and Gotsman [2006] showed how to use the GPU screen-space tangent plane algorithm for computing high-order Voronoi diagrams. Rong et al. [2006; 2011] introduced a jump flooding approach that flood fills a texture in parallel. It can be use to compute distance transforms, Voronoi diagrams, and centroidal Voronoi tessellation.

3 PixelPie: Approximate Poisson-disk Sampling by Rasterization and Occlusion Culling

We use the dart-throwing method [Cook 1986] to generate a maximal Poisson-disk distribution. Each dart is rasterized as a disk of radius r in the graphics pipeline. If the distance between two disk centers is less than r, one disk will occlude the center of the other disk. We remove one of the occluded disks to resolve this minimum distance violation. The graphics pipeline approximates the circular disks by discrete pixels for efficient processing.

We have implemented this approach using the OpenGL GLSL shaders and the CUDA Thrust Library. We rasterize darts as solid disks on two 2D textures as target domains and keep a list of empty

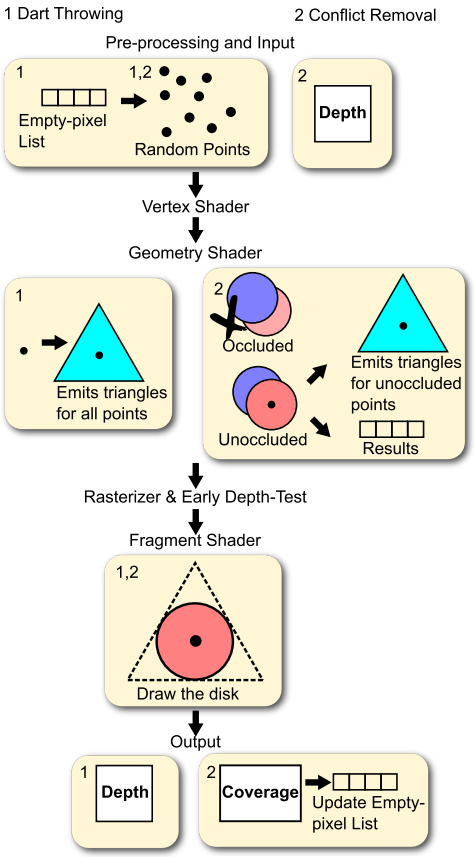


Figure 3: *PixelPie Pipeline overview: We use programmable shaders to perform Poisson-disk sampling. The vertex shader passes the random darts to the geometry shader. The geometry shader emits triangles for the darts and the fragment shader trims the triangles into disks. In Step 1: Dart-Throwing, we render disks for all the darts into the output depth map. In Step 2: Conflict-Removal, we read the depth map from step 1 to identify the acceptable unoccluded darts and render them into the coverage map. This two-step process iterates until we obtain a maximal Poisson-disk distribution.*

pixels. The first texture is a depth map for recording occluded darts. The second texture is a coverage map for tracking empty regions. We illustrate the PixelPie implementation in Figure 3. The vertex shader passes the input dart to the geometry shader without any modification. We discuss the geometry and fragment shaders in detail.

3.1 Dart Throwing

Input and Pre-processing The goal of this step is to generate a set of random samples in the empty regions. For this we maintain a list of pixels that are empty (not covered by a previously thrown dart). We use a CUDA kernel to generate a random number and use it to select a pixel position from the empty-pixel list. We use the default `minstd` random number generator provided by the CUDA Thrust library[1]. We dispatch this CUDA kernel in parallel. Each 2D dart position contains two 16-bit floating-point numbers (x, y). Each 2D position is packed into a 32-bit unsigned integer for data transfer. We initialize the coverage texture map to zero once at the

[1] We have also experimented with the Mersenne Twister random number generator [Saito and Matsumoto 2006] but we have not observed significant quality difference in the results.

beginning of the entire algorithm and reset the depth map texture to the maximum depth at the start of every iteration.

Geometry Shader The geometry shader emits one equilateral triangle per dart whose incircle has a radius r as in Equation 2. For each dart, we compute its 3D position by unpacking its 2D (x, y) coordinate from the unsigned integer dart data with `unpackUnorm2x16` and we assign the dart's unique `gl_PrimitiveIDIn` as its depth z. In addition to the triangle vertices, the geometry shader also emits normalized triangle coordinates for disk trimming by the fragment shader. The normalized coordinates are relative to a disk whose center is at $(0, 0)$ and radius $r = 1$. The rasterizer converts each dart triangle into fragments and interpolates the normalized coordinates across each fragment.

Fragment Shader The fragment shader is used to trim dart triangles into pixel-precise disks. This is done by simply discarding those fragments whose normalized coordinates have a magnitude greater than 1. Such fragments lie outside the incircle of the triangle representing the dart. At the end of this step, the fragment shader renders the disks into the depth map for proper occlusions.

We have also experimented with other rendering primitives, namely quads, point sprites, and smooth points. We have not observed a significant difference in the performance. Rendering disks from quads may increase performance as it discards fewer pixels than rendering disks from triangles, yet we suspect the overhead of additional geometry processing for multiple triangles offsets this benefit. The sizes of point sprites and smooth points are constrained to a small range and this limits their use in importance sampling.

Depth Test and Occlusions We enable depth testing to ensure that the disks properly occlude one another and optimize this approach's performance. Fragments of the disks that are rasterized early are likely to have lower primitive IDs and depth values. By using the "less-than" depth function, the GPU can efficiently reject the later occluded fragments. This is similar to rendering in an ascending depth order. We would like to point out that the GPU can perform early depth culling before invoking the fragment shader as the shader does not modify the depth.

3.2 Conflict Removal

Note that we had encoded the dart's unique ID `gl_PrimitiveIDIn` as its depth z. This was because we would like to reconstruct the IDs of the darts that survive the occlusion test by reading back the depth buffer and removing duplicates. However, this step needs to be carried out carefully. At the end of the the *Dart Throwing* step it is possible that some of the dart centers are occluded while their fragments survive the depth occlusion from neighboring darts. We are only interested in identifying the darts whose centers survive the occlusion check as representing a partial set of the final Poisson-disk distribution. To accomplish this efficiently we carry out the *Conflict Removal* step of each iteration after the *Dart Throwing* step.

Input and Pre-processing The output depth map of the *Dart Throwing* step is used as the input depth map for this step. We re-throw the same batch of darts as in the *Dart Throwing* step to identify those darts whose centers remain unoccluded.

Geometry Shader For each dart with coordinates (x, y, z), the geometry shader reads the depth d_{xy} at location (x, y) from the depth map. If the depth d_{xy} is closer than the dart's depth z, it means that the center of this dart is occluded by another dart. Such occluded darts violate the minimum distance requirement and are discarded by this shader. If the depth d_{xy} is equal to the dart's depth z, it means that the center of this dart is not occluded. In that case the geometry shader emits a triangle and records the dart's 2D position (x, y) into the results buffer by using the OpenGL transform feedback mechanism. The 2D positions of such unoccluded darts comprise a part of the final Poisson-disk distribution.

Fragment Shader Similar to the *Dart Throwing* step we use the fragment shader to trim dart triangles into pixel-precise disks. In this step however we do not have to do any depth testing and we render the disks on the coverage map.

3.3 Post-processing

The goal of this post-processing step is to use the binary coverage map to remove the newly-covered pixels from the empty pixel list. An elegant way to accomplish this is to use the parallel *stream-compaction* from the CUDA Thrust library. Stream compaction takes an input vector v and a predicate $\mathcal{P}$, and outputs those components of v for which $\mathcal{P}(v)$ is true [Horn 2005; Harris et al. 2007]. Stream compaction consists of a scan and a scatter operation. In our case the empty-pixel list is the input vector and the binary coverage map represents the predicate. The scan stage involves generating a temporary vector that represents the coverage for every entry in the empty-pixel list. The scan of this temporary vector generates the destination address for every uncovered pixel in the output vector. The scatter stage then copies the uncovered pixels using the destination addresses from the previous stage into the output empty-pixel list. Modern GPUs have a native scatter capability that makes this stream compaction very efficient [Harris et al. 2007].

We iteratively fill the remaining empty pixels until there are no empty pixels left. The results buffer contains a maximal Poisson-disk distribution. In subsequent iterations, we decrease the number of darts to match the number of remaining pixels. We would like to use the GPU by throwing a large number of darts in parallel, however, rejecting darts after filling up the domain is a waste of computing power. We will show how to empirically find a reasonable number of darts in Section 5.

Optimizations for an Empty Domain We substitute expensive memory accesses with cheap computation when the domain is empty during the first iteration in two ways. First, we generate the dart positions by directly converting 32-bit random numbers to the 2D coordinates. Second, in the post-processing step for the first iteration, we initialize the empty-pixel list by simply enumerating all the pixels locations instead of iterating memory lookups over an empty coverage map. This optimization is significant because the initial empty-pixel list is also the largest one.

Summary Figure 4 shows an example of the entire sampling procedure. Figure 4(a) shows the depth map after the dart-throwing step. Figure 4(b) shows the coverage map after the conflict-removal step. Figure 4(c) shows the remaining empty regions. Figure 4(d-f) show new darts are thrown at the empty regions in the subsequent iterations. This example terminates after four iterations.

3.4 Importance Sampling

PixelPie is versatile enough to readily extend to importance sampling. We vary the sampling densities by emitting disks of different radii; the constant r in Equation 2 is replaced by a radius function $r(i)$ as shown by the *importance map* in grayscale in Figure 5. We represent high importance by low (darker) intensities and small disks. Similarly, we represent low importance by high (lighter) intensities and large disks. The geometry shader reads the importance

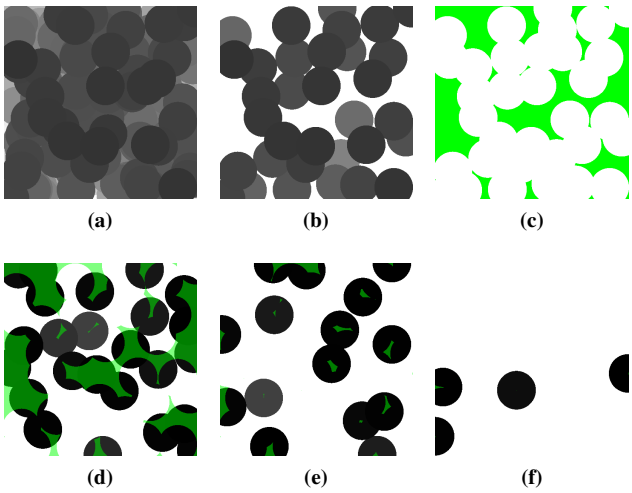


Figure 4: *Example: The first iteration: (a) shows the depth map of darts after the dart-throwing step. The intensity of the darts reflects their ID and depth. (b) shows the coverage map of non-conflicting darts. (c) shows the remaining empty regions. The subsequent iterations: (d), (e), and (f) show darts are thrown only into the empty regions during the second, third, and forth iterations. The entire domain is covered in four iterations.*

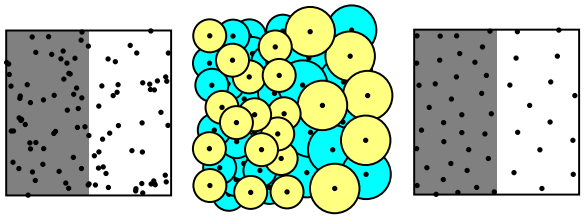


Figure 5: *Importance sampling: We throw darts of different sizes to vary the sampling densities according to an importance map. This figure corresponds to Figure 2(a), (c), and (e) in an importance sampling setup.*

texture and scales the disk radii accordingly. The changes in the geometry shader are presented in Figure 6.

The above process works well within regions of similar importance but additional care needs to be taken when dealing with boundaries between regions with significantly different importance. Here we follow the approach by Mitchell et al. [2012] that works well in an iterative setting. A sample j satisfies the empty disk property if

$$\forall i \in X, i \neq j : \|i - j\| \geq f(i,j) \tag{4}$$

Here $f(i,j)$ is a function of $r(\cdot)$ evaluated at a previously accepted sample i and a later candidate sample j. A candidate j is accepted if $\|i - j\| \geq f(i,j), \forall i \in X$ so far. Given that the sample i arrives before sample j, the importance function $f(i,j)$ may operate in the following ways [Mitchell et al. 2012]:

$$f(i,j) = r(i) \qquad \text{Prior Disk} \tag{5}$$
$$f(i,j) = r(j) \qquad \text{Current Disk} \tag{6}$$
$$f(i,j) = \max(r(i), r(j)) \qquad \text{Bigger Disk} \tag{7}$$
$$f(i,j) = \min(r(i), r(j)) \qquad \text{Smaller Disk} \tag{8}$$

We can implement each of these cases by using the depth-test functions and by encoding the importance into the depth. Since the Current Disks have a greater depth than the Prior Disks, we use the

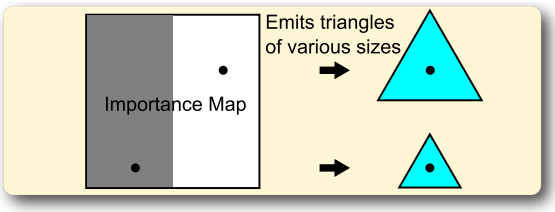


Figure 6: *Geometry shader for importance sampling: It emits triangles of different sizes according to the importance map.*

"greater-than" depth test function to ensure that the Current Disks occlude and reject the Prior Disks regardless of importance. Similarly, we use the "less-than" depth test function to favor the Prior Disks. For the other two cases, we can encode the disk radii into the most significant bits of the depth to select either the Bigger or the Smaller Disks. This ensures the depth of the Smaller Disks are less than the Bigger Disks. Once again, by using the "greater-than" or "less-than" depth tests, we can consistently pick the Bigger or Smaller disks.

4 Quality Analysis

In this section, we show our sampling approach produces high-quality Poisson-disk samples.

4.1 Sampling Correctness

Non-conflicting Samples from the Same Iteration In the dart-throwing step, if any distance between two samples is less than r, one of the disks must occlude the other sample. The conflict-removal step discards the occluded sample and removes this conflict. By encoding each dart with a unique depth and using orthogonal projection for rendering, we ensure that the occlusions are well-defined and the rasterized disks are free of Z-fighting.

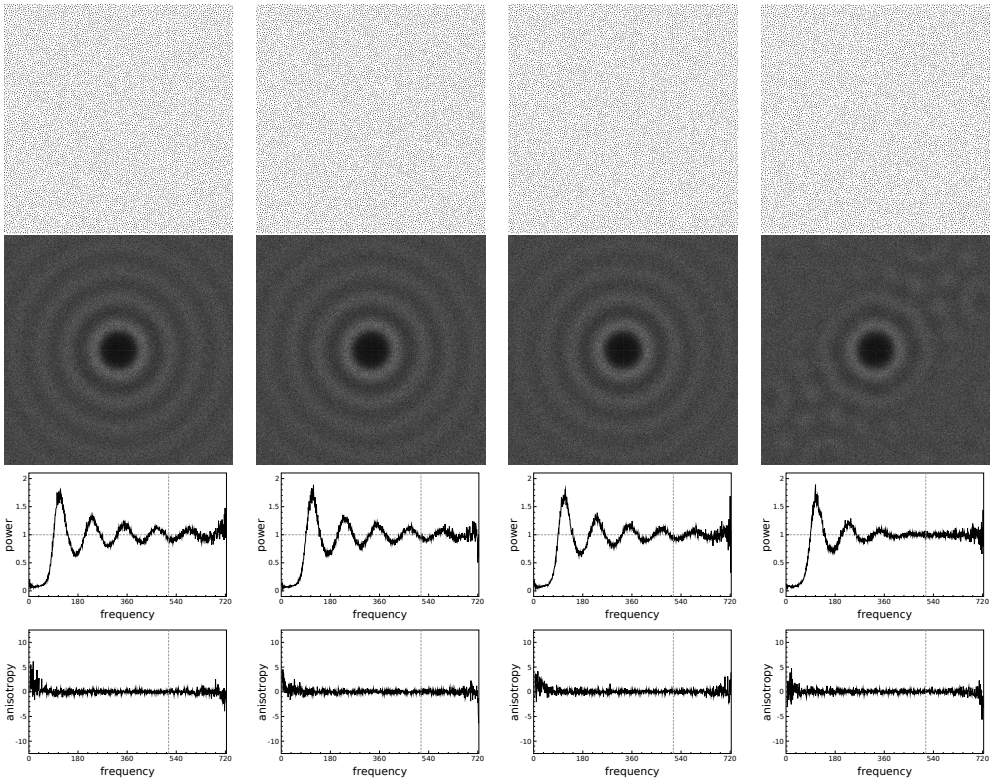
Non-conflicting Samples from Different Iterations The coverage map covers pixels that are r away from the accepted samples. Since we only throw darts on to the uncovered pixels in subsequent iterations, the distance of a new sample and any previously accepted sample must be at least r.

Convergence to Maximal Sampling Our iterative sampling approach converges to a maximal sample set by monotonically reducing the empty pixel count after each iteration. The sample with the smallest depth must be accepted as it cannot be occluded by any other sample. Also, any accepted sample must cover at least one previously empty pixel. This ensures the number of empty pixels reduces monotonically.

4.2 Spectral Analysis

Figure 7 shows spectral analysis of our sample point sets. Each point set consists of $10K$ samples at different texture resolutions. We used the pointset analysis tool [Schlömer and Deussen 2011] to compute this spectral analysis. We estimate the required radius, r following Gamito and Maddock [2009]: $r = 0.7766\sqrt{\frac{2}{\sqrt{3}N}}$, where N is the desired number of samples. For N = 10k, $r \approx 0.0083$.

Spectral analysis shows that samples generated by our method are comparable to Gamito and Maddock's accurate approach [2009]. The power spectrum periodograms are smooth at high resolutions of 4096^2 and 2048^2. Further, the anisotropy plots show that our spectra are radially symmetric. However, the quality of the samples deteriorates at low resolution 1024^2 where the power spectrum periodogram shows a biased pattern.



(a) *Gamito and Maddock* **(b)** 4096^2 **(c)** 2048^2 **(d)** 1024^2

Figure 7: *The figures above compare the quality of* $10K$ *Poisson-disk distribution samples at various target texture resolutions. From top to bottom, the rows show the generated samples, the power spectrum periodogram, the radially averaged power spectrum, and the radial anisotropy plot. Our results on high resolution textures are comparable to Gamito and Maddock [2009]'s accurate approach in (a). However, the power spectrum periodogram of low resolution samples at* 1024^2 *shows a biased pattern in (d).*

4.3 Dart Samples at Subpixel Locations

Discretization can be a source of bias if the texture resolution is too low. We examine the 1024^2 samples from Figure 7(d). The radius of such samples is $0.0083 \times 1024 = 8.5$ pixels per disk. We examine such a discretized disk in Figure 8(a). The center of this disk is at the center of a pixel and its shape is closer to an octagon than a circular disk. The polygonal shape affects the distribution of samples resulting in aliasing artifacts.

We reduce aliasing by creating multiple variations of discretized disks based on the sub-pixel location of their centers. Figure 8(b) shows the 4 different shapes of disks created by 2×2 subpixel centers. We show the average subpixel-centered disks of 2×2 and 4×4 in Figure 8(c). The average subpixel-centered disks are visually close to a circular anti-aliased disk and provide a better probabilistic coverage of the boundaries over all such disks.

We use the high precision of our (x, y) dart coordinates to center them with subpixel accuracy. The 16-bit precision of our (x, y) dart coordinates is often higher than the (x, y) resolutions of our textures. For example, the integer pixel coordinates for a 8192^2 texture only require 13 bits and we can use the remaining 3 bits for subpixel accuracy.

Angular Distribution of Nearest Neighbors We show that the subpixel-centered samples substantially reduce the discretization bias by evaluating the angle distribution of neighboring samples. The angles among nearest-neighbor samples should ideally be uniformly distributed in a Poisson-disk distribution. However, the discrete polygons create an uneven distribution of neighbors. We compute the angular statistics among the neighboring samples to quantify this bias. We first construct a Delaunay triangulation of the samples to find the nearest neighbors. Then we compute the angles between the neighboring samples and the x-axis. We plot the resulting distribution of angles in a 360-bin histogram as shown in Figure 9. In Figure 9(a), the pixel-centered samples produce a lot of spikes. This is because an axis-aligned octagonal shape of the dart reduces sample neighbors at multiples of $45°$. In comparison, the angle distribution of the subpixel-centered samples is more uniform. Their angle distribution is comparable to the distribution produced by Gamito and Maddock's [2009] approach in Figure 9(b). Since one can expect the angles to be evenly distributed around the mean frequency, we can quantify the angular bias by computing the standard deviation of the angular distribution. The standard deviations of the pixel-centered samples, the subpixel-centered samples, and the Gamito and Maddock [2009] samples are 73.2, 16.7, and 16.5 respectively. The subpixel-centered samples have substantially reduced the angular bias of the distribution.

We show the spectral analysis of $10K$ samples with subpixel centers in Figure 10. In contrast with Figure 7(d), there is no visibly biased pattern in the power spectrum periodogram in Figure 10(a).

5 Performance

In this section, we evaluate the performance of PixelPie. PixelPie can generate high-quality Poisson-disk distribution at a rate of 6.8

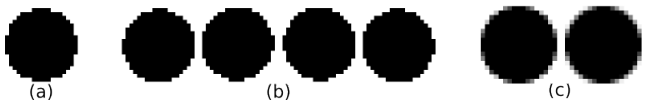
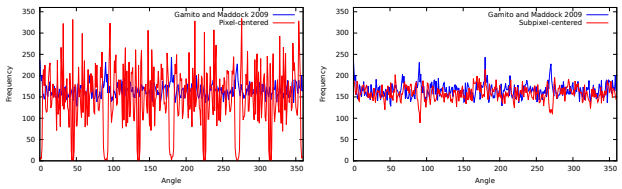


Figure 8: *Comparison of pixel-centered and subpixel-centered disks: (a) shows the shape of a discretized pixel-centered disk, (b) shows variations of disks centered at 2×2 subpixel locations, and (c) shows the average of the subpixel-centered disks at 2×2 and 4×4 subpixel locations. The average of subpixel sampling effectively generates anti-aliased disks.*



(a) **(b)**

Figure 9: *Angular distributions: The angles between neighboring samples should be evenly distributed in a Poisson-disk distribution. We plot this angular distribution with 360-bin histograms in which the x axis shows the angle between the neighboring samples and the y axis shows the frequency. We plot our distributions in red and the Gamito and Maddock [2009] distributions in blue. (a) shows the bias of the pixel-centered samples results in several spikes with a standard deviation of 73.2. Subpixel-centered samples substantially reduce this bias in (b) to 16.7 that compares favorably to the Gamito and Maddock's [2009] standard deviation of 16.5.*

million samples per second. We have implemented PixelPie on the Windows 7 OpenGL platform with Visual C++ and NVIDIA CUDA Thrust. We ran our experiments on a NVIDIA GeForce GTX 580 GPU and an Intel Core i7 2.8 GHz CPU.

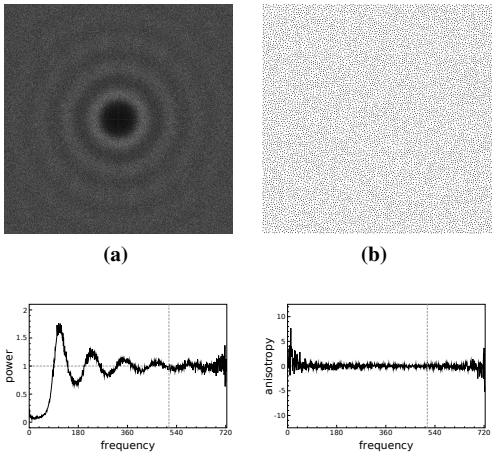
5.1 Dart Batch Size

The GPU usage is maximized when processing large batches of darts, yet rejecting darts after filling up the domain wastes computation power. We have performed experiments to estimate the optimal batch size. Similar to the quality assessment section, we generate $N = 10K$ samples by throwing darts in different batch sizes on to textures of various resolutions. We throw batches of $N/8$ to $2N$ ($1.25K$ to $20K$) darts. In Figure 11, we plot the running time against different batch sizes. As can be seen, the running time drops rapidly as the batch size increases from $1.25K$ to $6K$ darts and rises slowly thereafter as the batch size further increases to $20K$ darts. This behavior is consistent across textures of different resolutions. This study shows that a batch size of roughly $N/2$ is the most efficient.

5.2 Performance Characteristics

The performance of PixelPie is depended on the resolution of the target texture. We plot its running time against different sample sizes in Figure 12. The sample sizes N range from 0.1% to 1% of the target texture size. The texture size instead of the sample size determines the performance because the GPU has to render every pixel of the target texture.

In Figure 13 and Table 1, we summarize the performance of PixelPie. We throw darts of a small absolute radius (8.5 pixels) with the optimal batch size $N/2$ on textures of different sizes to obtain samples of different sizes. We plot the running time of PixelPie against the number of samples generated in Figure 13. Table 1 details the texture size, batch size, sample sizes, relative dart radius, timings and memory usage. We present the timings in three parts: (1) Dart-throwing step (S1), (2) Conflict-removal step (S2) , and (3) the post-processing stream compaction (PP).



(a) **(b)**

(c) **(d)**

Figure 10: 1024^2 *subpixel-centered samples: Subpixel-centered samples reduce the discretization bias. In comparison with Figure 7(d), biased patterns are not visible on this power spectrum periodogram.*

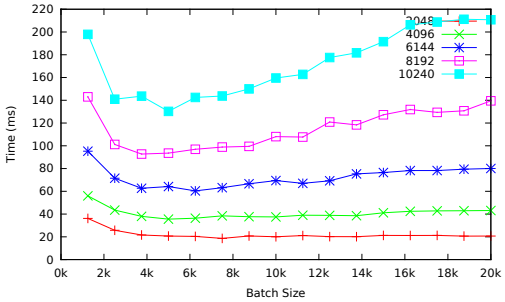


Figure 11: *Batch size: We throw darts to sample a Poisson-disk distribution with $N = 10K$ samples in batches of different sizes. The batch sizes range from $N = 1.25K$ to $2N = 20K$. Each line represents a target texture of a different resolution. The running time drops rapidly from $1.25K$ to $6K$ darts and slowly rises further. A batch size around $N/2$ is consistently efficient.*

Figure 14 shows PixelPie's sampling performance with darts of different radii. The performance increases rapidly when the radius decreases, therefore it is important to identify the minimum required radius for the optimal performance.

Comparison We present a comparison of PixelPie and the recent dart-throwing maximal Poisson-disk sampling approaches in Table 2. Gamito and Maddock's test program [2009] can produce $1M$ samples at a rate of $22K/s$ on our platform. We ran PixelPie on a GeForce GTX 460 GPU to compare against the performance reported in [Ebeida et al. 2011; Ebeida et al. 2012]. We note that their precision is dependent on the resolution of the flat quadtree. PixelPie can produce $800K$ samples at a rate of $4M/s$, which is four times faster than the previously reported performance.

Samples	Parameters			Memory	Performance Timing (ms)					Rate
N	Domain	Batch #	Radius r	(MB)	Iter.	S1	S2	PP	Total	(sample/s)
9.81K	1024^2	5.05K	8.30E-03	8.09	8	4.71	3.25	8.37	16.34	0.60M
39.05K	2048^2	20.21K	4.15E-03	32.45	9	6.11	3.53	10.56	20.20	1.93M
87.77K	3072^2	45.48K	2.77E-03	72.43	9	8.92	4.57	13.43	26.92	3.26M
155.96K	4096^2	80.86K	2.08E-03	128.75	10	12.97	6.27	17.08	36.33	4.29M
243.28K	5120^2	126.34K	1.66E-03	200.88	10	18.44	7.87	21.53	47.83	5.09M
350.17K	6144^2	181.93K	1.38E-03	289.13	10	24.73	9.81	26.26	60.80	5.76M
476.03K	7168^2	247.62K	1.19E-03	393.38	11	33.33	12.47	33.72	79.52	5.99M
624.35K	8192^2	323.43K	1.04E-03	513.63	11	42.90	15.71	39.87	98.48	6.34M
787.27K	9216^2	409.34K	9.22E-04	650.00	11	52.85	18.02	47.12	117.99	6.67M
973.22K	10240^2	505.35K	8.30E-04	802.38	11	64.88	21.47	55.55	141.91	6.86M

Table 1: *This table summarizes the parameters, performance, and memory consumption of our experiments presented in Figure 13. Radius r denotes the relative radius in the $(0, 1)$ domain. Timing section includes number of iterations to convergence, the cumulative timings of dart-throwing step (S1), conflict-removal step (S2), and stream-compaction (PP), and the total time.*

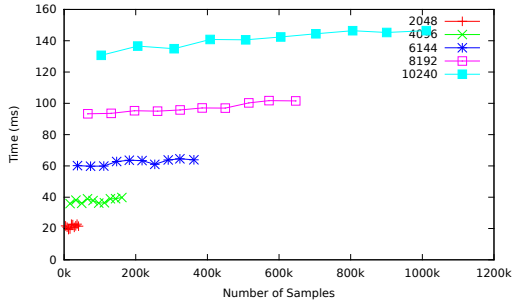


Figure 12: *Sample size: The performance of PixelPie is mainly dependent on the resolution of the target textures. The running time is essentially the GPU time for rendering all the pixels. The change of the sample size affects the quality of the samples rather than the running time.*

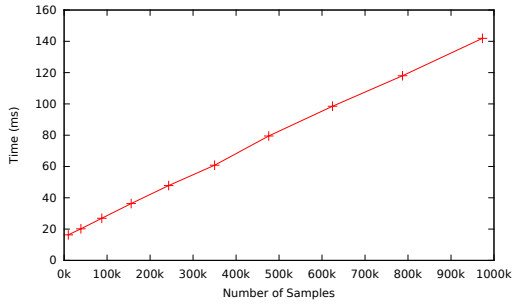


Figure 13: *Performance: We throw darts with an absolute radius of 8.5 pixels and a batch size of $N/2$ to textures of various sizes to obtain high quality Poisson-disk distributions.*

5.3 Memory Consumption

Table 1 shows our GPU memory usage and Table 2 shows a comparison of PixelPie and the other state-of-the-art dart throwing approaches. Our memory usage is mainly determined by the size of the target textures. The profiled memory consumption in Figure 1 shows PixelPie requires approximately 8 bytes per pixel. The target textures and the empty pixel-list use about 7 bytes per pixel. We use 32-bit pixels (4 bytes) for the depth map and unsigned char pixels (1 byte) for the coverage map. We reserve an empty-pixel list (2 bytes) that is half the size of the domain with 32-bit elements. Additional memory resources required include the random dart source buffer, the results buffer, and any extra memory used by the CUDA Thrust operations. PixelPie does not require a significant amount of CPU main memory.

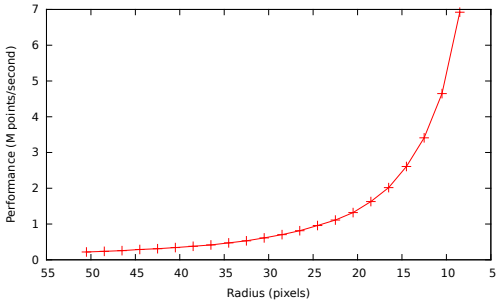


Figure 14: *Radii: We throw darts of various absolute radii to generate Poisson-disk distributions on $10K \times 10K$ textures. This plot shows the sampling performance (y-axis) increases as the absolute radius decreases (x-axis).*

Method	Type	Samples	Rate
[Gamito and Maddock 2009]	CPU	1M/0.5GB	22K/s
[Ebeida et al. 2011] (GTX460)	GPU	2M/1 GB	224K/s
[Ebeida et al. 2012] (GTX460)	GPU	0.6M/1GB	1M/s
PixelPie (GTX460)	GPU	0.78M/0.65GB	4M/s
PixelPie (GTX580)	GPU	0.97M/0.8GB	6.9M/s

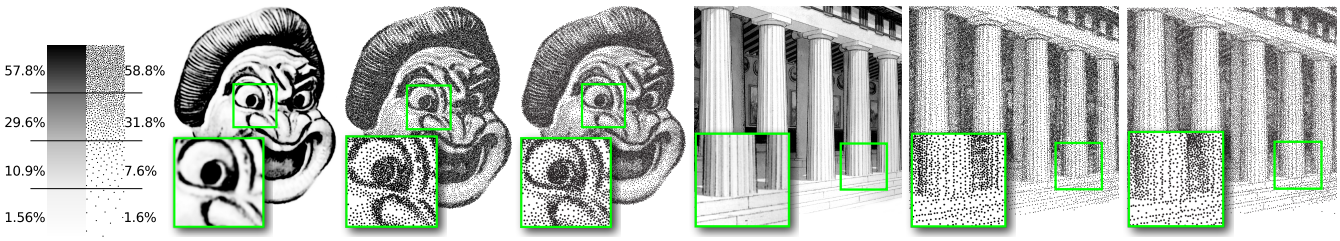
Table 2: *Performance comparison: PixelPie is four times faster than the state-of-the-art CPU and GPU dart throwing approaches with comparable memory usage.*

5.4 Importance Sampling

We show the results of importance sampling in Figure 15. Figure 15 (a) shows the importance sampling of a quadratic gradient by PixelPie. Figure 15 (b-g) show a comparison of PixelPie against Kalantari and Sen [2011]'s serial and discrete dart throwing approach. Each example contains approximately $20K$ points. While both approaches produce samples of similar quality, our GPU parallel PixelPie approach runs 400 times faster than Kalantari and Sen [2011]'s serial approach on a CPU. PixelPie samples were generated on a GTX 580 with 2048^2 textures. Their results were generated on a Core i7 2.8 GHz CPU [Kalantari and Sen 2011]. We select the *Smaller Disk* option at the boundaries between different importance regions. The importance maps are provided by Kopf et al. [2006].

6 Limitations

One limitation of PixelPie is the bias introduced by the discretization. While this is a limitation, we have shown that the angular distribution of our subpixel-centered samples is comparable to the samples produced by the Gamito and Maddock's [2009] exact ap-



(a) **(b)** **(c)** *0.053 seconds* **(d)** *24 seconds* **(e)** **(f)** *0.031 seconds* **(g)** *25 seconds*

Figure 15: *Importance Sampling: (a) shows PixelPie sampling on a quadratic gradient. (b-g) show a comparison of PixelPie against Kalantari and Sen [2011]'s discrete approach. (b) and (e) show the importance maps. (c) and (f) show PixelPie results. (d) and (g) show their results. Both approaches produce samples of similar quality while our parallel GPU PixelPie approach runs 400 times faster than their serial CPU approach. PixelPie was tested on a GTX 580 GPU. Their approach was tested on a Core i7 2.8GHz CPU [Kalantari and Sen 2011].*

proach. A second limitation of PixelPie is that the precision of PixelPie is constrained by the resolution of the target texture. However the use of discrete textures allows us to use rasterization and depth-test hardware to achieve very high sampling performance. PixelPie trades precision with performance gain. We believe this approach is particularly suitable for screen-space interactive computer graphics applications, since the need for precision depends on the resolution of the discrete framebuffer.

7 Conclusions

In this paper, we have shown how to use the programmable graphics pipeline to generate 2D maximal Poisson-disk distributions. The key to our approach, PixelPie, is to identify and remove conflicting darts by culling occluded disks. Our implementation leverages the optimized graphics functions of the modern GPU to perform parallel dart-throwing and conflict-removal in large batches. PixelPie can be extended to perform importance sampling by programming the geometry shader to emit disks of different radii. We have also presented angular distribution of nearest neighbors to quantify the bias of Poisson-disk distributions. We have introduced subpixel-centered samples to substantially reduce this bias for discrete sampling approaches. Our experiments have shown that PixelPie can produce maximal Poisson-disk samples at the rate of nearly 7 million samples per second on a GeForce GTX 580. The surprising insight of our work is that hardware-oriented solutions can significantly outperform the computational geometric state-of-the-art algorithms for Poisson-disk sampling by a factor of 4 or more with no perceptible difference in quality. Our approach indicates that modern geometry and tessellation shaders have the potential to significantly accelerate proximity-driven geometry computations.

Acknowledgements

We thank the anonymous reviewers for their constructive comments and suggestions. This work has been supported in part by the NSF grants: CCF 05-41120, CMMI 08-35572, CNS 09-59979 and the NVIDIA CUDA Center of Excellence. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the research sponsors.

References

BALZER, M., SCHLÖMER, T., AND DEUSSEN, O. 2009. Capacity-constrained point distributions: A variant of lloyd's method. *ACM Transactions on Graphics 28*, 3, 86:1–86:8.

BELL, N., AND HOBEROCK, J. 2011. *GPU Computing Gems, Jade Edition*. Morgan Kaufmann, ch. Thrust: A Productivity-Oriented Library for CUDA.

BOWERS, J., WANG, R., WEI, L. Y., AND MALETZ, D. 2010. Parallel Poisson disk sampling with spectrum analysis on surfaces. *ACM Transactions on Graphics 29*, 6, 166:1–166:10.

CHEN, R., AND GOTSMAN, C. 2012. Parallel blue-noise sampling by constrained farthest point optimization. *Computer Graphics Forum 31*, 5, 1775–1785.

CHEN, Z., YUAN, Z., CHOI, Y.-K., LIU, L., AND WANG, W. 2012. Variational blue noise sampling. *IEEE Transactions on Visualization and Computer Graphics 18*, 10, 17841796.

CLINE, D., JESCHKE, S., WHITE, K., RAZDAN, A., AND WONKA, P. 2009. Dart throwing on surfaces. *Computer Graphics Forum 28*, 4, 1217–1226.

COOK, R. L. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*, 1, 51–72.

DE GOES, F., BREEDEN, K., OSTROMOUKHOV, V., AND DESBRUN, M. 2012. Blue noise through optimal transport. *ACM Transactions on Graphics 31*, 6 (Nov.), 171:1–171:11.

DIPPÉ, M. A. Z., AND WOLD, E. H. 1985. Antialiasing through stochastic sampling. *Computer Graphics 19*, 3, 69–78.

DUNBAR, D., AND HUMPHREYS, G. 2006. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics 25*, 3, 503–508.

EBEIDA, M. S., DAVIDSON, A. A., PATNEY, A., KNUPP, P. M., MITCHELL, S. A., AND OWENS, J. D. 2011. Efficient maximal Poisson-disk sampling. *ACM Transactions on Graphics 30*, 4, 49:1–49:12.

EBEIDA, M. S., MITCHELL, S. A., PATNEY, A., DAVIDSON, A. A., AND OWENS, J. D. 2012. A simple algorithm for maximal Poisson-disk sampling in high dimensions. *Computer Graphics Forum 31*, 2, 785–794.

FATTAL, R. 2011. Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics 30*, 4, 48:1–48:12.

FISCHER, I., AND GOTSMAN, C. 2006. Fast approximation of high-order voronoi diagrams and distance transforms on the gpu. *Journal of Graphics, GPU, and Game Tools 11*, 4, 39–60.

GAMITO, M. N., AND MADDOCK, S. C. 2009. Accurate multidimensional Poisson-disk sampling. *ACM Transactions on Graphics 29*, 1, 8:1–8:19.

HACHISUKA, T. 2005. *GPU Gems 2*. Addison-Wesley, ch. High-Quality Global Illumination Rendering Using Rasterization.

HARRIS, M., SENGUPTA, S., AND OWENS, J. D. 2007. *GPU Gems 3*. Addison-Wesley, ch. Parallel Prefix Sum (Scan) with CUDA.

HOFF, III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH*, ACM, 277–286.

HORN, D. 2005. *GPU Gems 2*. Addison-Wesley, ch. Stream Reduction Operations for GPGPU Applications.

JONES, T. R., AND KARGER, D. R. 2011. Linear-time poisson-disk patterns. *Journal of Graphics, GPU, and Game Tools 15*, 3, 177–182.

JONES, T. R. 2006. Efficient generation of Poisson-disk sampling patterns. *Journal of Graphics, GPU, & Game Tools 11*, 2, 27–36.

KALANTARI, N. K., AND SEN, P. 2011. Efficient computation of blue noise point sets through importance sampling. *Computer Graphics Forum 30*, 4, 1215–1221.

KALANTARI, N. K., AND SEN, P. 2012. Fast generation of approximate blue noise point sets. *Computer Graphics Forum 31*, 4, 1529–1535.

KOPF, J., COHEN-OR, D., DEUSSEN, O., AND LISCHINSKI, D. 2006. Recursive Wang tiles for real-time blue noise. *ACM Transactions on Graphics 25*, 3, 509–518.

LAGAE, A., AND DUTRÉ, P. 2005. A procedural object distribution function. *ACM Transactions on Graphics 24*, 4, 1442–1461.

LAGAE, A., AND DUTRÉ, P. 2006. An alternative for Wang tiles: colored edges versus colored corners. *ACM Transactions on Graphics 25*, 4, 1442–1459.

LAGAE, A., AND DUTRÉ, P. 2006. Poisson sphere distributions. In *Vision, Modeling, and Visualization*, 373–379.

LAGAE, A., AND DUTRÉ, P. 2008. A comparison of methods for generating Poisson disk distributions. *Computer Graphics Forum 27*, 1, 114–129.

LI, H., WEI, L. Y., SANDER, P. V., AND FU, C. W. 2010. Anisotropic blue noise sampling. *ACM Transactions on Graphics 29*, 6, 167:1–167:12.

MCCOOL, M., AND FIUME, E. 1992. Hierarchical Poisson disk sampling distributions. In *Graphics Interface*, vol. 92, CHCCS, 94–105.

MITCHELL, S. A., RAND, A., EBEIDA, M. S., AND BAJAJ, C. 2012. Variable radii Poisson-disk sampling. In *Canadian Conference on Computational Geometry*, CCCG.

NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. 2008. Scalable parallel programming with CUDA. *Queue 6*, 2 (Mar.), 40–53.

OSTROMOUKHOV, V., DONOHUE, C., AND JODOIN, P. M. 2004. Fast hierarchical importance sampling with blue noise properties. *ACM Transactions on Graphics 23*, 3, 488–495.

OSTROMOUKHOV, V. 2007. Sampling with polyominoes. *ACM Transactions on Graphics 26*, 3, 78:1–78:6.

OWENS, J. D., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRÜGER, J., LEFOHN, A. E., AND PURCELL, T. J. 2007. A survey of general-purpose computation on graphics hardware. *Computer graphics forum 26*, 1, 80–113.

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 4, 66:1–66:13.

RONG, G., AND TAN, T.-S. 2006. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, ACM, 109–116.

RONG, G., LIU, Y., WANG, W., YIN, X., GU, X. D., AND GUO, X. 2011. Gpu-assisted computation of centroidal voronoi tessellation. *IEEE Transactions on Visualization and Computer Graphics 17*, 3, 345–356.

SAITO, M., AND MATSUMOTO, M. 2006. SIMD-oriented fast mersenne twister: a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods*, Springer, 607–622.

SCHLÖMER, T., AND DEUSSEN, O. 2011. Accurate spectral analysis of two-dimensional point sets. *Journal of Graphics, GPU, and Game Tools 15*, 3, 152–160.

SIGG, C., PEIKERT, R., AND GROSS, M. 2003. Signed distance transform using graphics hardware. In *Visualization*, IEEE, 83–90.

SUD, A., OTADUY, M., AND MANOCHA, D. 2004. Difi: Fast 3d distance field computation using graphics hardware. *Computer Graphics Forum 23*, 3, 557–566.

WEI, L. Y. 2008. Parallel Poisson disk sampling. *ACM Transactions on Graphics 27*, 3, 20:1–20:9.

WEI, L.-Y. 2010. Multi-class blue noise sampling. *ACM Transactions on Graphics 29*, 4, 79:1–79:8.

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

XIANG, Y., XIN, S.-Q., SUN, Q., AND HE, Y. 2011. Parallel and accurate poisson disk sampling on arbitrary surfaces. In *SIGGRAPH Asia Sketches*, 18:1–18:2.

YELLOTT JR, J. I. 1982. Spectral analysis of spatial sampling by photoreceptors: topological disorder prevents aliasing. *Vision Research 22*, 9, 1205–1210.

ZHOU, Y., HUANG, H., WEI, L.-Y., AND WANG, R. 2012. Point sampling with general noise spectrum. *ACM Transactions on Graphics 31*, 4 (July), 76:1–76:11.