Fast Maximal Poisson-Disk Sampling by Randomized Tiling

Tong Wang
IST, University of Tokyo

Reiji Suda
IST, University of Tokyo

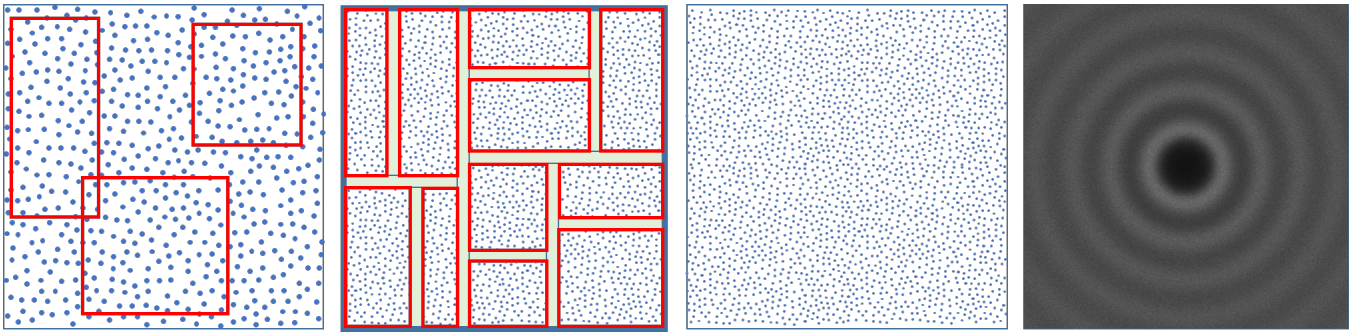


Figure 1: (a) Original maximal Poisson-disk sample pattern with *n* points generated by any MPS(Maximal Poisson-disk sampling) method. (b) Example of randomized tiling. Tile randomly clipped tiles from previous pattern, to form the target sample set with *4n* points. The size and position of each clipping are determined by a spatial data structure (KD-tree, etc.). (c)Eliminate conflict points in margin area of each random tile, and insert new points in the gap caused by the elimination to ensure maximal coverage property. (d) The power spectrum of Poisson disk distributions generated with our algorithm.

ABSTRACT

It is generally accepted that Poisson disk sampling provides great properties in various applications in computer graphics. We present KD-tree based randomized tiling (KDRT), an efficient method to generate maximal Poisson-disk samples by replicating and conquering tiles clipped from a pattern of very small size. Our method is a two-step process: first, randomly clipping tiles from an MPS(Maximal Poisson-disk Sample) pattern, and second, conquering these tiles together to form the whole sample plane. The results showed that this method can efficiently generate maximal Poisson-disk samples with very small trade-off in bias error. There are two main contributions of this paper: First, a fast and robust Poisson-disk sample generation method is presented; Second, this method can be used to combine several groups of independently generated sample patterns to form a larger one, thus can be applied as a general parallelization scheme of any MPS methods.

CCS CONCEPTS

•Computing methodologies →Computer graphics;

KEYWORDS

Poisson-disk Sampling

HPG '17, Los Angeles, CA, USA

DOI: 10.1145/3105762.3105778

ACM Reference format:
Tong Wang and Reiji Suda. 2017. Fast Maximal Poisson-Disk Sampling by Randomized Tiling. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105778

1 INTRODUCTION

This paper shows how to use a divide-and-conquer based method to generate Poisson-disk distributed samples efficiently.

Sampling with Poisson-disk distribution is considered one of the most important sampling pattern in image processing and computer graphics. As it is generally believed that a wide range of applications in these field can make a profit from the Poisson-disk sampling pattern with blue noise feature. For example, rendering can benefit from it as this pattern is considered a good placement similar to layout of natural retina cells [Pharr et al. 2016], which is proved to generate less aliasing in synthesized image and prevent artifacts. Other applications include object placement [Wei 2010], texture synthesis [Cohen et al. 2003], remeshing on surfaces [Guo et al. 2015], etc.

The Poisson-disk sampling pattern is a group of samples with no two of them within a specified radius, and maximal means that no more points can be added to the group of samples, which means that there are no *gaps* that are not covered by a sample disk. We follow the definition in [Gamito and Maddock 2009] and the extended version in [Ip et al. 2013], denote sample set as X in domain D, with each sample as i, according to definition that maximal Poisson-disk samples have following properties:

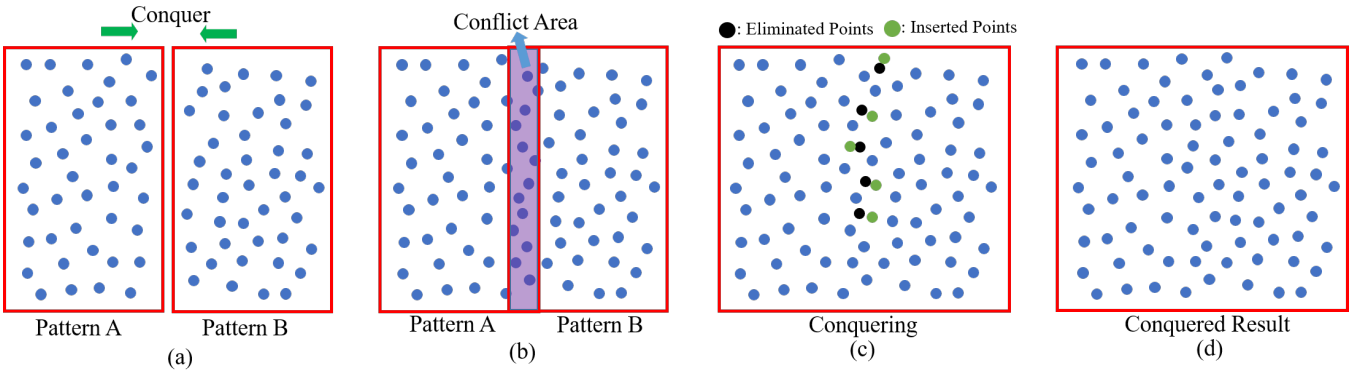


Figure 2: Overview of the concept: (a) Two maximal Poisson-disk sample patterns. (b) Combine them together, samples in the middle strip area will not satisfy minimum distant (conflict-free) requirements. (c) While eliminating conflict points in the conflict area, keep inserting new points to ensure maximal coverage property. (d) The conquered result.

$$\forall i \in X, \forall S \subseteq D : P(i \in S) = \int_S di \tag{1}$$

$$\forall i, j \in X, i \neq j : \|i - j\| \geq r \tag{2}$$

$$G = \{j \in D | \forall i \in X, \|i - j\| \geq r\} = \emptyset \tag{3}$$

Equation 1 shows that the possibilities of sample placements on sampling domain is uniform. This is also referred as the *bias-free* property. Equation 2 is the conflict-free requirement, that no pairs of points can be closer than a minimum distance. Equation 3 is the maximal coverage requirement, that no other samples can be inserted in the sample set without breaking the minimum distant requirement. To meet the requirements of a wide range of applications, our goal is to generate a large group of good quality maximal Poisson-disk sample set in interactive speed.

There are extensive studies on how to generate samples with some or all above properties. Most of them focused on using dart throwing based method ([Dippé and Wold 1985] [Cook 1986]) to insert as many points in the sampling domain as possible, with some arbitrary techniques on detecting and filling gaps to satisfy maximal properties ([Gamito and Maddock 2009], [Ebeida et al. 2012]). But because of the complexity nature of dart throwing, these methods often fail to meet performance requests of interactive applications.

In this paper, we present a divide-and-conquer method that can efficiently generate maximal Poisson-disk samples with only minor error. The idea came from a very straightforward observation: Combining two already generated Poisson-disk sample sets to form a larger one is always easier than completely generate the larger one itself from scratch.

As presented in Figure 1, The algorithm is a two-step process:

(1) Divide the sampling space into random tiles, and fill the tiles with points clipped from a random rectangle with the same shape in the pre-generated MPS pattern. Here we use KD-tree to split the whole sampling space.

(2) Eliminate conflict points in margin area of each tile, and properly insert new points in the gaps caused by the elimination.

Step (1) efficiently generates sample-point-tiles from a very small pattern, with required randomness. Step (2) guarantee conflict-free and maximal coverage property of the final result. The following article will discuss various aspects on more details of this method, including background, correctness, implementation, performance, and result analysis. All current work and discussion are based on 2-dimensional sampling, but it's straightforward to extend the method to multi-dimensional space.

We call this method a KD-tree based Randomized Tiling method, which will be constantly abbreviated as KDRT in the following article.

2 BACKGROUND

Enormous efforts have been placed on this topic since 80s of last century. Some important early work including [Dippé and Wold 1985] and [Cook 1986] started the exploration of stochastic sampling patterns in anti-aliasing and various other applications in computer graphics. They proposed the most popular and the simplest method to generate Poisson-disk samples: dart throwing. With a predefined disk radius r, point samples are continuously inserted into sample domain. Point will be accepted if not conflicting previous accepted points. One obvious disadvantage of this method is computation complexity. As the termination condition cannot be determined with minor efforts, the acceptance rate of samples will drop significantly as the sampling domain filled by more and more samples, which makes this algorithm extremely slow to converge. Generate point set with maximal coverage property is even nearly impossible using only dart throwing. A thorough survey of some earlier work in this field is presented by [Lagae and Dutré 2008].

More recent works focusing on dart-throwing methods tends to use some specific techniques to reduce sampling area after each dart accepted, to increase the acceptance rate of the next dart. [Gamito and Maddock 2009] try to keep all disk-free area in a quad-tree data structure, throw darts against the tree, then update the

data structure after each dart accepted. [Ebeida et al. 2011] uses a similar grid data structure, and keeps tracking a convex polygonal approximating the void inside the grid. [Ebeida et al. 2012] keeps record of an implicit flat quad-tree of active grids, then continuously splits and tests sub-grid against each dart. The quad tree is kept as flat as possible, so memory consumption is significantly reduced. [Yan and Wonka 2013] provides a deep study on concepts of general gap analysis in Poisson-disk sample sets with varying radii. This category of methods can be orthogonal to ours.

Tile based methods can generate Poisson-disk samples much faster, but with serious sacrifice in sample quality or trade-off in biasness property. The basic idea is to generate small sample sets and tile them in the sampling domain to form a large sample set. Some significant work including [Cohen et al. 2003], [Lagae and Dutré 2005], [Kopf et al. 2006] and [Wachtel et al. 2014]. But quality loss of these tile based methods are sometimes not acceptable, and few tile-based methods focused on the maximal coverage property of sample domain. KDRT can also be classified as a tile based method, but the error of KDRT is much less as more degrees of freedom in randomization are exploited in KDRT. Also few of tile-based method insist a maximal coverage property. Comparing with [Kalantari and Sen 2012] who also used a tile-based method fetching from a small pattern, our method is different in several ways. First, we randomized the tiling scheme on sampling domain with a KD-tree based manner, which ensures a more random result. Second, our method met the conflict-free requirement that most applications will need, and third, KDRT guaranteed the final result sample set a maximal coverage property.

A Poisson-disk sample sets can also be obtained by repositioning samples. Reposition base methods are generally referred as relaxation based methods. Samples generated by many previous Lloyd relaxation based methods tend to be too regular for many applications, until capacity-constrained relaxation methods are generally applied such as in [Balzer et al. 2009] and [Xu et al. 2011]. A more recent method using optimal transport is discussed in [De Goes et al. 2012], which will also be compared to our method in this paper. This category of methods can also be orthogonal to ours.

Most recent solutions for Poisson-disk sampling would utilize a parallel computation pattern to boost its performance. [Wei 2008] started the randomized phase group based parallelization pattern, which is enhanced to deal with 2D manifold sampling in following works such as [Bowers et al. 2010] and [Ying et al. 2013]. Our method also used some of their methods for GPU-friendly execution. Other new and inspiration methods including [Ebeida et al. 2014] which is effective in high dimensional space, and [Yuksel 2015] with a simple and elegant elimination method to generate neat Poisson disk samples on 2D manifolds.

Almost all previous literatures discuss the quality of Poisson-disk samples or other sampling patterns in a frequency domain analysis framework. See [Lagae and Dutré 2008] for a more detailed disscussion. More theoretical and practical details are provided by [Durand 2011], [Subr and Kautz 2013] and [Pilleboue et al. 2015] on analysis of sampling quality. Our method used a similar analysis tool as discussed in [Schlmer and Deussen 2011]

Our method is also partly inspired by [Kalantari and Sen 2012]. They try to tile the sampling domain with rotated square pattern. Although they reached very fast sampling speed, the cost of low sampling quality and failed conflict-free and maximal coverage requirements sometimes are not unacceptable. Our method instead provides a more accurate solution that met the requirements of conflict-free and maximal coverage, with bounded error in biasness.

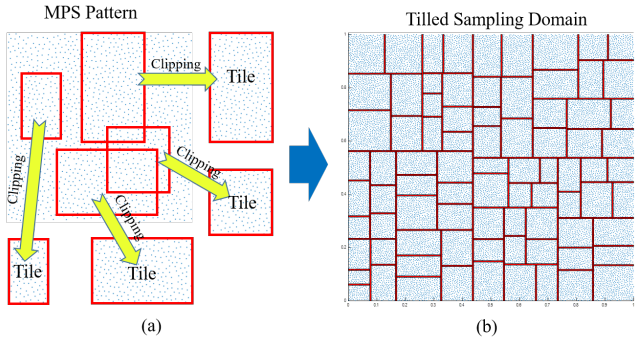


Figure 3: (a) Clipping tiles from a pre-generated MPS pattern. The shape and size of each tile is determined by each leaf node of the KD-tree that subdivide the sampling domain. (b) Tile all the tiles to corresponding position in the sampling domain.

To provide an overview and comparison, this paper has the following contribution:

(1) We present a fast and robust tile-based maximal Poisson-disk sampling method (KDRT) by randomly dividing sample domain into tiles, tiling each tile with a clipping from pre-generated pattern, and gluing all the tiles together to generate a large sample set.
(2) We show that this method can be a proved framework to do divide-and-conquer parallelization of almost any other Poisson-disk sampling methods.

3 ALGORITHM OVERVIEW

The randomized tiling algorithm is a two-step process:

(1) Clipping tiles from a small pre-generated MPS pattern, and tile them on sampling domain to form the large sample set.
(2) Eliminate conflict points in margin area of each clipping, and properly insert points to cover gaps caused by elimination.

3.1 Clipping and Tiling

3.1.1 Notations. There are some notes and abbreviations that will be constantly referred, including:

- *MPS*: Maximal Poisson-disk Sampling.
- n: Size of the output MPS set.
- *Pattern*: A pre-generated MPS set with size k, $k << n$.
- *Clipping*: The operation of fetching a subset of samples within a specified rectangle (2-D) of the pattern.
- $\mathcal{S}$: The set of Poisson-disk samples.
- $\mathcal{C}$: The set of clipping and tiling result (containing conflict points).
- $\mathcal{D}$: The whole sampling domain.
- $\mathcal{P}$: The whole pattern domain.

- r: Poisson-disk radius (minimum distance for samples).
- *disk*: The circle with a sample as center, and r as radius.
- *threshold*: The minimum edge length of the leaf node, as a pre-defined parameter.
- *ratio*: The number of samples of the result MPS sample set divided by the number of samples in pattern set, roughly means how many times more samples a user want to produce from the pattern set.

3.1.2 Clipping. *Clipping* means fetching a subset of samples within a specified bounding rectangle. The first step of this algorithm would be dividing $\mathcal{D}$ into randomized rectangles which will be later used as clipping unit, as illustrated by Figure 3. We call these rectangles building blocks. One convenient way to do this is just to fetch each leaf node of the KD-tree that subdivide domain $\mathcal{D}$.

Algorithm 1: Generate leaf rectangles as building blocks

Input : The bounding box of $\mathcal{D}$: b, current splitting axis: $axis$, threshold: Len
Output: Array of bounding box: BBoxes

```
Function GetBuildingBlocks (b, axis);
if DimensionCheck(b) == ALL or
  DimensionCheck(b) == axis then
    BoundingBox leftBox ← RandomSplitLeft(b, axis);
    BoundingBox rightBox ← RandomSplitRight(b, axis);
    GetBuildingBlocks(leftBox, nextAxis);
    GetBuildingBlocks(rightBox, nextAxis);
else if DimensionCheck(b) == nextAxis then
    GetBuildingBlocks(b, nextAxis)
else
    BBoxes.add(b);
end
Function DimensionCheck ( b, axis, Len);
if b.xmax − b.xmin > Len and b.ymax − b.ymin > Len then
    return ALL;
else if b.xmax − b.xmin > Len then
    return xaxis;
else if b.ymax − b.ymin > Len then
    return yaxis;
else
    return leaf;
end
```

The KD-tree is generated from top to bottom, as shown in Algorithm 1. The function GetBuildingBlocks() will generate an array of bounding boxes as building blocks. Each bounding box contains the maximal and minimum position for each dimension, in 2-D they are $xmin, xmax, ymin, ymax$. We choose the dividing plane for each subdividing dimension randomly, and also within a range that can satisfy a minimum dimension length parameter, so that each dividing in this procedure will ensure all building blocks' edge length be larger than the threshold. The function DimensionCheck() is going to check if the current bounding box is ready for subdivide on the specific axis. If a specific axis is not suitable for further subdivide, the subdivision should only happen on other axes.

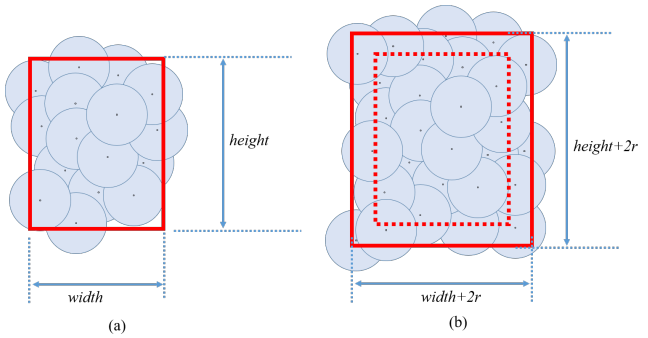


Figure 4: (a) Clipping tiles itself may not satisfy maximal coverage requirements. This tile is clipped from a MPS pattern, but there might be gaps near margin of the clipped tile. (b) If we enlarge the clipping domain by $2r$ in each dimension, and include all samples in outer margin domain, then the original clipping is ensured to be maximal coverage.

Bear this in mind, it's obvious that the subdividing depth as a parameter plays an important role in the final quality of this algorithm. A deep depth means that each tile is small, which add more randomness than using a flat KD-tree to clip a big portion of the pattern as the building tile. As a limitation of this algorithm, the minimum length of each dimension for a leaf rectangle is $6r$, and we find that using $20r$ to $40r$ as the leaf node threshold a good trade-off between speed and quality. When subdividing procedure detects that the dimension will be smaller than threshold, the recursive subdividing on the particular axis will stop. The whole process will stop until all edge length of every leaf node cannot be further subdivided.

Next is to prepare a pre-generated pattern that contains samples at least larger than the biggest leaf node in the KD-tree. This pattern can be generated by any other methods that met the bias-free, conflict-free and maximal coverage requirement. Here we use algorithms in [Ebeida et al. 2012] to generate the pattern.

Note that when a rectangle is being clipped from the pattern, the samples in the rectangle may not satisfy the maximal coverage requirements. As the margins of the rectangle are arbitrarily selected, and the points that originally covered the gaps may not be included in the clipping. We define a *shrinked domain* a domain that's been reduced by $2r$ in each dimension, and an *enlarged domain* a domain that's been extended by $2r$ in each dimension.

To always get a building block with maximal coverage property, we have to prove that:

Theorem 3.1. *(maximal coverage sub-domain): Let $\hat{\mathcal{D}}$ be the shrinked domain of $\mathcal{P}$. $\forall \mathcal{S} \subset \hat{\mathcal{D}}$, Let $\mathcal{E}$ be the enlarged domain of $\mathcal{S}$, the disks clipped by $\mathcal{E}$ covers $\mathcal{S}$.*

Proof. To prove it by contradiction, assume that a point $i \in \mathcal{S}$ not covered by disks clipped by $\mathcal{E}$. By definition of enlarged domain, we know that $\forall p \in \mathcal{E}^C$, the distance between i and p is larger than r, thus no disks in $\mathcal{E}^C$ can cover i. And with the assumption we know that no disks in $\mathcal{E}$ can cover i, which means that i is in a

gap. But as $i \in \mathcal{S} \subset \mathcal{P}$, and $\mathcal{P}$ is a domain with maximal coverage property, contradiction exist. □

Refer to 4 for a simple visualization. We know from this that for all the building blocks acquired from Algorithm 1, we have to clip an enlarged block in the pattern to ensure maximal coverage of the sub-domain.

3.1.3 Tiling. With discussions above, the pseudocode to generate tiles will be straightforward. Note that as $k << n$, so when clips are clipped from pattern, they have to be scaled by a pre-defined parameter $ratio$ to fit in the building block. We also have to translate each clipping to the correct corresponding position of each building blocks in the final sampling domain. This procedure is described in Algorithm 2.

Algorithm 2: Psudocode of Generating Tiles

```
Input  : BBoxes BoxArray, Pattern P, Ratio ratio
Output : Sample set C that contains conflict samples
Function GenerateTiles(BoxArray, P, ratio) ;
forall b of BoxArray do
    e ← Enlarge b by 2r;
    e ← Scale e by ratio;
    Point op ← Original position of e in KD-tree;
    Point rp ← Random position in pattern;
    e ← Translate e to rp;
    ClippedSamples ← Clipping pattern using e;
    TransSamples ← Translate ClippedSamples to op ;
    TransSamples ← Scale TransSamples by 1/ratio;
    C.Add(TranslatedSamples);
end
```

After each tile being tiled on the building blocks of $\mathcal{D}$, now we get a point set C with maximal coverage, as Figure 5(a). But the enlarged boundary does not satisfy the requirements of minimum distance, as each tile may have several conflicts in margin area with its neighbors. Next we need to do elimination and insertion to resolve this problem.

3.2 Elimination and Insertion

Currently as we have a clipping and tilling result with maximal coverage, the most important task now is to resolve conflicts inside this sample set. Elimination procedure is simple and intuitive, for each sample point being processed, just mark the conflict points as INVALID. After each elimination, a gap may or may not appear due to the elimination. We call the gaps left by each elimination the *Generated Gap*. We call the neighbors within a certain radius of the eliminated points the *Text Buffer*, as these samples in the buffer are candidate samples for gap detection. The main goal of elimination and insertion step is to eliminate all conflict sample points, and insert new sample points to cover all generated gaps caused by sample elimination.

3.2.1 Generated Gap Analysis. Drawing inspiration from [Yan and Wonka 2013], we know that:

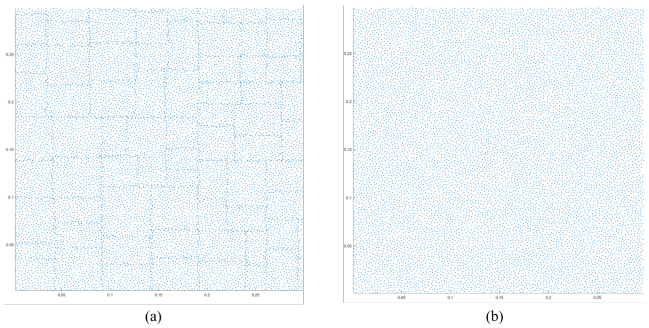


Figure 5: (a) A portion of the clipping and tiling result. The tiling result is maximal coverage as each tile satisfy maximal coverage condition according to theorem 3.1. But we can see the redundant boundary of all the enlarged tiles that violate conflict-free requirements. (b) The final result after elimination and insertion.

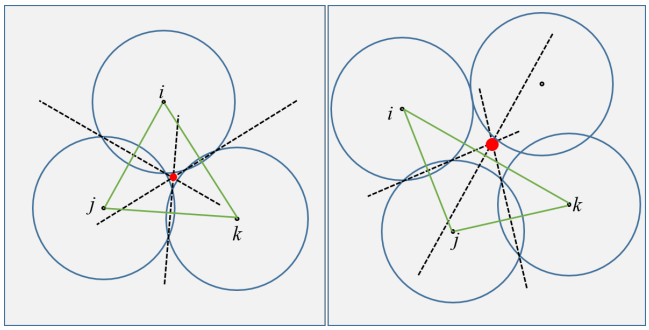


Figure 6: If the circumcenter of the triangle i, j, k is not in a gap, then there are no gaps exist between these three disks.

Theorem 3.2. *(Gap-detection): A gap exist among three disks centered at i, j, k ($i, j, k \in C$), iff the circumcenter c of the triangle i, j, k is not covered by any disks.*

This is also illustrated in Figure 6. Using the above theorem, we can do the actual gap detection given a sample buffer called TestBuffer. We define a gap detection and insertion procedure in Algorithm 3.

Note that if GapDetectionAndInsertion being applied on all sample points $p \in C$, it is not guaranteed that all gaps could be covered. In Algorithm 3, only one point is inserted in each gap detected, which is not enough for some special cases, as illustrated in Figure 7. So we introduce a concept *GapDetector* to help locate all gaps in multiple iterations after the first elimination loop.

We define the point currently being processed as *pivotpoint*. when a sample point is eliminated from the original set due to conflicts with pivot point, we mark all disks intersecting the eliminated sample point as *GapDetectors*. It is obvious that all gaps consist of arcs from gap detectors.

3.2.2 Eliminate and Insert Algorithm. There are two main loops in the function: the first loop eliminates each conflict points, and while elimination, inserting only once inside any gap that can be

Algorithm 3: Psudocode of Gap Detection and Insertion

```
1 Function GapDetectionAndInsertion(TestBuffer,i,C);
  Input  : Sample Point i, TestBuffer, C
  Output : C with conflicts removed and new samples inserted
2 forall every two samples m, n in TestBuffer do
3 |   center ← Circumcenter(i, m, n);
4 |   if center in a gap then
5 |   |   Insert(center, C);
6 |   else
7 |   |   continue;
8 |   end
9 end
```

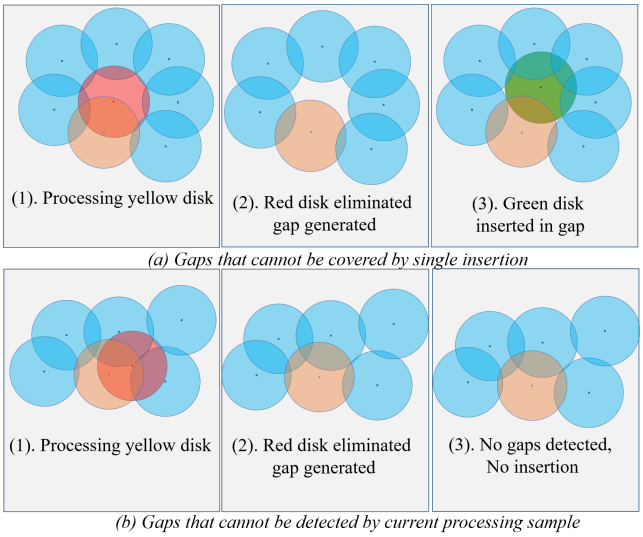


Figure 7: (a) Not all gaps can be covered by a single insertion. (b) Not all gaps can be simply detected by currently processing sample point.

detected directly by pivot point. We know from Figure 7 that the gap might not being covered completely with a single insertion, and potential gaps might still exist after each insertion.

To resolve this problem, the second loop is introduced. For each gap detector, do GapDetectionAndInsertion() for a TestBuffer with increased search range for from $2r$ to $4r$ to include all sample disks that consist the gap in case that gap detectors might fail to detect neighboring gaps (In practice, we find that only search gap detectors in range $2r$ is enough for gap detection. But here we cannot provide a strict proof to claim so.) From theorem 3.2 we know that at least one gap detector can successfully detect the gap, and insert a point inside the gap. Mark the inserted point as gap detector again (as the inserted point will be the most possible candidate to help in detecting the clipped gaps caused by the insertion). After the whole loop finished, a stream compaction will gather all gap detectors, and start the next iteration. See Algorithm 4.

3.2.3 Termination. The first for loop of Algorithm 4 will terminate as there are finite number of samples in C. The following while

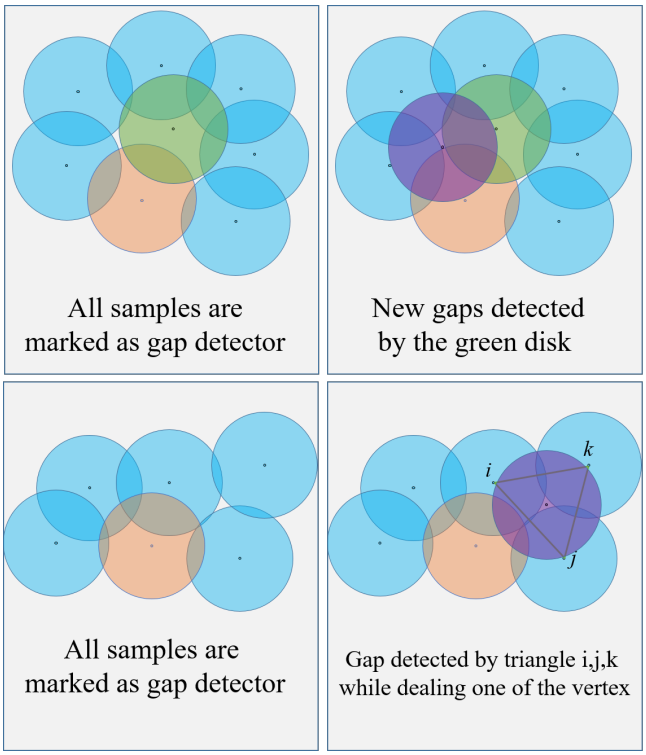


Figure 8: Solving problem stated in Figure 7 by adding another loop on gap detectors.

loop will terminate, as for each insertion in gap, the new inserted sample disk will cover finite area, until there are no gaps detected, and no samples inserted. The algorithm will converge after enough iterations. We can also prove the termination in this way:

Proof. It can be concluded that for each iteration of processing gap detectors, big gaps partly covered by inserted samples turn into smaller gaps, small gaps completely covered by inserted points disappear. If we define a gap that need at most N inserted points to cover as *type N gap*, then each iteration solves the *type 1 gap*, and reduce the type of other gaps by 1. As each gap must have a finite number of type, in this case, the algorithm is able to meet a termination condition, which is all gap type decreased to 1, and then solved by a single insertion loop. □

Figure 9 illustrated this proof.

We tested iterations needed for convergence by generating 10 million MPS results for 10 times, and actually all the generation needed only 2 iterations to finish the gap type decreasing, special cases that need more insertion rarely showed out.

Also note that we don't have to run this algorithm on all $p \in C$, only operate points in conflict area is enough, as all other points are conflict-free with maximal coverage property (see Figure 4), thus no points will be eliminated, and no generated gaps can be detected in those area.

Algorithm 4: Psudocode of Gap Detection and Insertion

```
Function EliminateAndInsert C;
  Input  : Clipping and Tiling result C
  Output : Maximal Poisson Disk Sample Set S
forall Sample p in conflict area of C do
    Mark p as GapDetector;
    ConflictBuffer ← RadiusSearch(p, r);
    forall conflictPoint in ConflictBuffer do
        Eliminate conflictPoint from C;
        TestBuffer ← RadiusSearch(conflictPoint,2r);
        Mark all samples in TestBuffer as GapDetector;
        GapDetectionAndInsertion(TestBuffer, p, C);
    end
end
Mark all inserted samples as GapDetector;
while True do
    GapDetectorArray← StreamCompaction(C,
     isGapDetector());
    forall Sample p in GapDetectorArray do
        TestBuffer ← RadiusSearch(conflictPoint, 4r);
        GapDetectionAndInsertion(TestBuffer, p, C);
        Mark all inserted samples as GapDetector
    end
    if No new sample point inserted then
        break;
    else
        continue;
    end
end
```

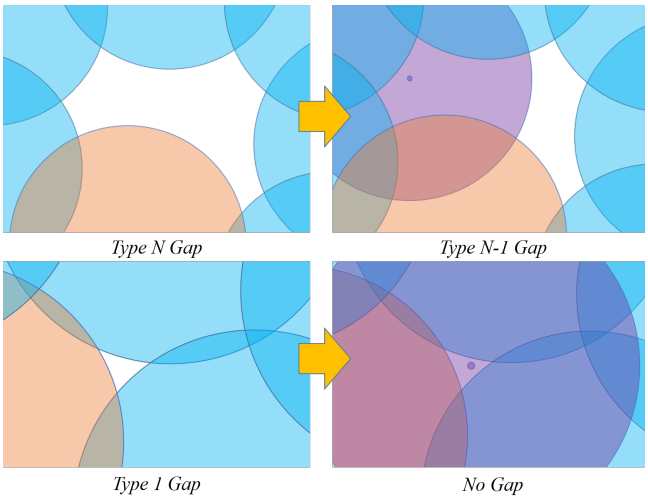


Figure 9: Gap type means the largest number of samples we could insert into the gap without breaking conflict-free requirement. Each iteration processing gap detector reduces the gap type for each gap by 1.

4 IMPLEMENTATION AND RESULT ANALYSIS

4.1 CPU and GPU Implementation and result

Our testing machine is a PC with a moderate spec: Intel Core i7-4930K Quad-core CPU with 32GB of RAM. GPU test is on a single NVIDIA GTX 1080. There are no intrinsic parallelization patterns of KDRT, we divide the sampling domain into grids, and apply a similar phase group based parallelization pattern as stated in [Wei 2008]. That is, dealing samples separated by a long distance as independent samples, and process them in parallel. Refer [Wei 2008] for more details. We used a uniform grid as radius search data structure on CPU and GPU. There is a clear upper-bound of amount of samples stored for each uniform grid, so the size of each uniform grid can be fixed, and future data will overwrite invalid data in the grid. Note that no atomic operation needed, as the operation on exist samples is marking. A stream compaction after each iteration will manage all the elimination.

4.1.1 Sampling Correctness. The randomized tiling is a tile-based method with the tiling process being completely randomized, thus the results have very similar spectral quality compared to reference. Refer to Figure 13 for more details. KDRT is proved to satisfy the minimum distance (conflict-free) requirement and maximal coverage requirement. Notice that this method is biased, but the biasness is bounded, See 5.1 for more details.

4.1.2 Quality. The most important parameter in KDRT is the minimum dimension length of the KD-tree used in clipping and tiling: R_{leaf}. Increasing R_{leaf} can reduce the area of confliction, reduce the number of necessary operations of elimination and inserting. But larger tiles tend to lead the tiling result more regular, as more similar portion can be clipped from the pattern. Fortunately, KDRT have other degree of freedom in randomness: the position of each building block is random, and the position of clipping on pattern is also random. As presented in Figure 13, we did not observe serious quality drop by even increasing the minimum dimension length of KD-tree leaf node to a size closer to the size of the MPS pattern itself.

4.1.3 Comparison. We compared our result with BNOT([De Goes et al. 2012]) and MPS([Ebeida et al. 2012]). [Ebeida et al. 2012] is an unbiased Poisson-disk sampling method with maximal coverage feature, and also being one of the fastest high performance MPS generating method. [De Goes et al. 2012] is also famous for its good performance in applications require blue noise samples. We know from various theoretical analysis that suppressed low frequency domain power is critical in applications such as Monte-Carlo integration, which explains the advantages that BNOT([De Goes et al. 2012]), MPS([Ebeida et al. 2012]) and our methods over uniform random sampling in Figure 10. The first row in Figure 10 is 2D point samples, the second row is the power spectrum of frequency domain, the third row shows how a zone plate $z = sin(x^2 + y^2)$ is rendered by 1 million of each samples, and the last row shows the San Miguel scene (modeled by Guillermo M. Leal Llaguno) rendered with a bi-directional path tracer with 200 million camera ray sampled by each algorithm. We could observe similar anti-aliasing and denoising quality of [De Goes et al. 2012], [Ebeida et al. 2012]

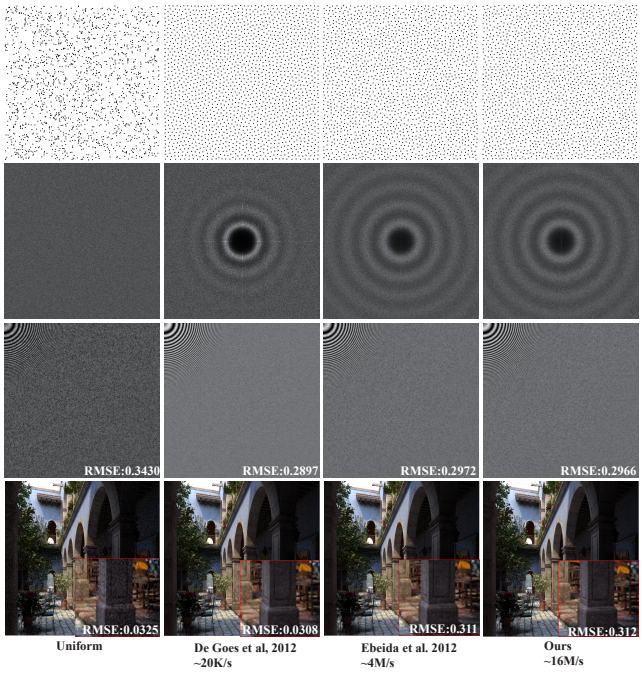


Figure 10: Comparison of sample quality. Row 1: 2D samples. Row 2: Power spectrum. Row 3: Zone plate function $z = sin(x^2 + y^2)$ sampled by 1 million samples each with Gaussian filter. Row 4: San Miguel scene rendered with 200 million camera rays (generated by each method).

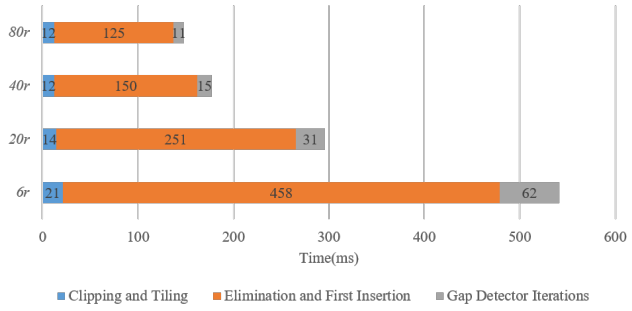
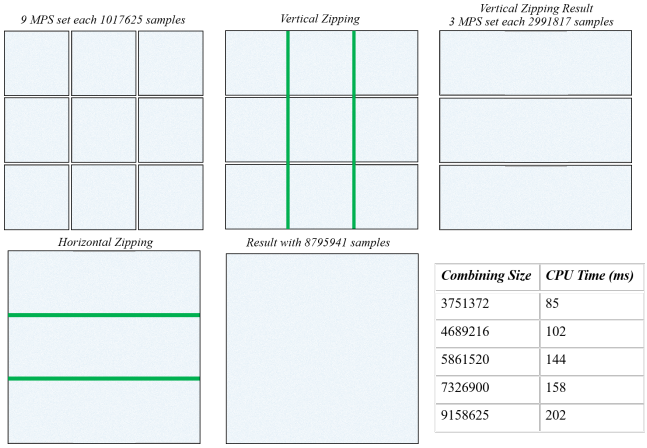


Figure 11: Time taken by each procedure to generate 1 million maximal Poisson-disk samples. The elimination and first insertion part takes longest, as it is operated against the most amount of samples.

and our algorithm, with our algorithm having a clear advantage in sample generation speed.

We compared our implementation to GPU implementation of [Ebeida et al. 2012] for performance. As their result is an implementation of unbiased Poisson-disk samples with maximal coverage feature, and also being one of the fastest high performance MPS generating method. In the grouped Figure 13, we calculated 10 average of various radius requirements and different parameter



Combining Size	CPU Time (ms)
3751372	85
4689216	102
5861520	144
7326900	158
9158625	202

Figure 12: Example of combining independently generated MPS samples by applying our elimination and insertion framework. Here is an example of combining 9 MPS sets into one. With a vertical and horizontal *zipping*, all points are glued together by elimination and insertion, with only minor loss of samples in margin area.

of R_{leaf} (the threshold of tile size) for each unit to compare. Frequency power spectrum of samples with 0.01 and 0.005 radius are scaled to the same level as radius 0.0025 for better visualization, with radial mean of the power spectrum calculated on the resolution according to the scaling. From the results we can see that even combining big tiles at a level of $80r$, we can barely see any regularity patterns that often appears in other tiling based method. Also noted that KDRT provide 2.5x to 4x sampling rate for high performance applications.

4.2 Combining Large MPS Patterns

Besides generating MPS using KDRT, the elimination and insertion framework can also be applied in combining independently generated MPS patterns together to form a complete sample set. Sometimes these MPS patterns are generated by different cores or different nodes in network all in parallel. Using the elimination and insertion framework can effectively turn every serial algorithm into a parallel version, with only very small overhead. See Figure 12 for a simple example of this application.

5 CONCLUSION

In this paper we present KDRT, an efficient and robust method to generate maximal Poisson-disk samples. This method replicate tiles clipped from a very small pattern on a randomly subdivided KD-tree based sampling domain, then eliminate all conflicts in margin area, and insert new samples to ensure maximal coverage. This method is intuitive and easy to implement. The elimination and insertion framework can also be used orthogonally to parallelize almost any other MPS generation methods. The rough sketch in Figure 12 illustrated a process of combining 9 really large sample sets together with our method.

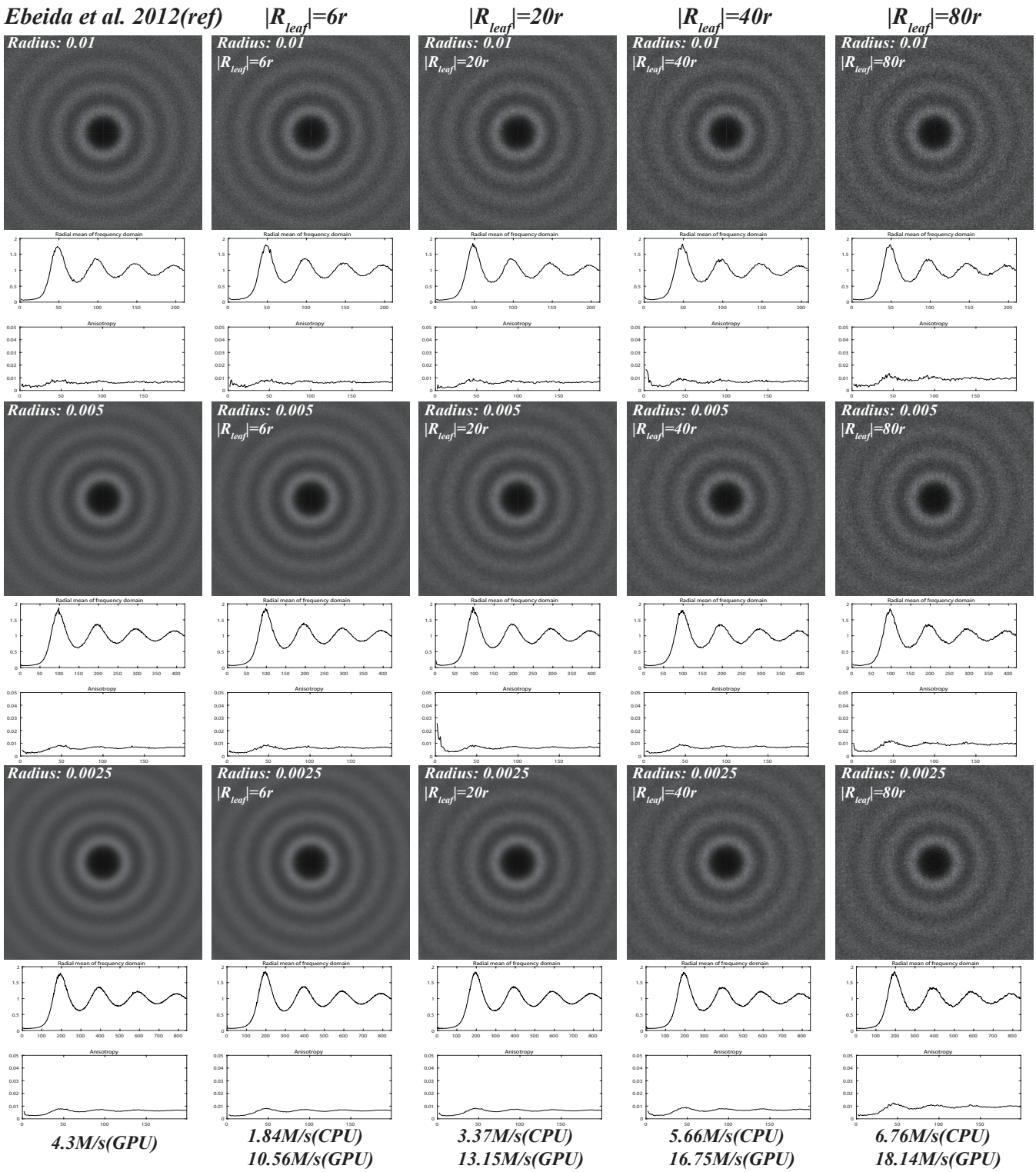


Figure 13: KDRT sampling result. Each row of data are generated under the same radius condition. The fist column is generated by using a reference method developed by Ebeida et al. Other columns are generated using KDRT, with different minimum KD-tree dimension length. The last row is the generation rate measured by using each method to generate 2 million MPS samples.

5.1 Limitation

A clear limitation is that this method cannot generate MPS from scratch, a pre-generated pattern with relatively high quality is needed.

Another limitation is in parallelization of insertion and elimination process. There are no intrinsic parallelization patterns of this method, as sample markings are directly related to insertion and elimination during the process. Hence parallelize KDRT take efforts in dealing with scattered memory access and irregular operations on data structure.

One important issue of our method that has been mentioned many times is that this method has bias in conflict area, which is introduced by the deterministic sample insertion step. But it can be observed that the number of samples introducing bias is bounded, thus the total error of the tiling result is bounded. Consider the minimum tile edge length being Nr, r being the radius of Poisson disk in the final sample set. With tiling redundancy introduced by our method, the area that each tile actually covers in the sample domain is $(Nr + r)^2$. Thus the portion of area that the elimination and insertion happening will take:

$$\frac{(Nr + r)^2 - (Nr)^2}{(Nr + r)^2} = \frac{2N + 1}{(N + 1)^2} \tag{4}$$

which is inversely proportional to $N(N > 0)$, and will be less than 10 percent with $N > 19$. So as long as the tiling unit is not too small, the bias error in the conflicting area can be controlled easily.

5.2 Future work

The tiling process is KD-tree based, hence very easy to extend the method to high dimensions. Elimination and insertion involves higher dimensional neighbor search, which may suffer from curse of dimensionality. With our experience and early stage experiments, this method can at least maintain good quality and performance in 3D space sampling.

Also we see potential of adaptive sampling with this method, as the KD-tree based tiling has the nature of recursive and subdivisive. It would be really interesting to see this method being used on applications such as image stippling.

ACKNOWLEDGMENTS

The first draft of this paper is not in very good condition, we would like to express our gratitude, to all the anonymous reviewers for their invaluable advices on the original manuscript that help to shape the paper to be much better.

REFERENCES

Michael Balzer, Thomas Schlömer, and Oliver Deussen. 2009. *Capacity-constrained point distributions: a variant of Lloyd's method.* Vol. 28. ACM.

John Bowers, Rui Wang, Li-Yi Wei, and David Maletz. 2010. Parallel Poisson disk sampling with spectrum analysis on surfaces. In *ACM Transactions on Graphics (TOG)*, Vol. 29. ACM, 166.

Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. *Wang tiles for image and texture generation.* Vol. 22. ACM.

Robert L Cook. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG)* 5, 1 (1986), 51–72.

Fernando De Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. 2012. Blue noise through optimal transport. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 171.

Mark AZ Dippé and Erling Henry Wold. 1985. Antialiasing through stochastic sampling. *ACM Siggraph Computer Graphics* 19, 3 (1985), 69–78.

Fredo Durand. 2011. A frequency analysis of Monte-Carlo and other numerical integration schemes. (2011).

Mohamed S Ebeida, Andrew A Davidson, Anjul Patney, Patrick M Knupp, Scott A Mitchell, and John D Owens. 2011. Efficient maximal Poisson-disk sampling. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 49.

Mohamed S Ebeida, Scott A Mitchell, Muhammad A Awad, Chonhyon Park, Laura P Swiler, Dinesh Manocha, and Li-Yi Wei. 2014. Spoke darts for efficient high dimensional blue noise sampling. *arXiv preprint arXiv:1408.1118* (2014).

Mohamed S Ebeida, Scott A Mitchell, Anjul Patney, Andrew A Davidson, and John D Owens. 2012. A Simple Algorithm for Maximal Poisson-Disk Sampling in High Dimensions. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 785–794.

Manuel N Gamito and Steve C Maddock. 2009. Accurate multidimensional Poisson-disk sampling. *ACM Transactions on Graphics (TOG)* 29, 1 (2009), 8.

Jianwei Guo, Dong-Ming Yan, Xiaohong Jia, and Xiaopeng Zhang. 2015. Efficient maximal Poisson-disk sampling and remeshing on surfaces. *Computers & Graphics* 46 (2015), 72–79.

Cheuk Yiu Ip, M Adil Yalçin, David Luebke, and Amitabh Varshney. 2013. Pixelpie: Maximal poisson-disk sampling with rasterization. In *Proceedings of the 5th High-Performance Graphics Conference*. ACM, 17–26.

Nima Khademi Kalantari and Pradeep Sen. 2012. Fast generation of approximate blue noise point sets. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1529–1535.

Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. 2006. *Recursive Wang tiles for real-time blue noise.* Vol. 25. ACM.

Ares Lagae and Philip Dutré. 2005. A procedural object distribution function. *ACM Transactions on Graphics (TOG)* 24, 4 (2005), 1442–1461.

Ares Lagae and Philip Dutré. 2008. A comparison of methods for generating Poisson disk distributions. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 114–129.

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation.* Morgan Kaufmann.

Adrien Pilleboue, Gurprit Singh, David Coeurjolly, Michael Kazhdan, and Victor Ostromoukhov. 2015. Variance analysis for Monte Carlo integration. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 124.

Thomas Schlmer and Oliver Deussen. 2011. Accurate Spectral Analysis of Two-Dimensional Point Sets. *Journal of Graphics, GPU, and Game Tools* 15, 3 (2011), 152–160. DOI:http://dx.doi.org/10.1080/2151237X.2011.609773 arXiv:http://dx.doi.org/10.1080/2151237X.2011.609773

Kartic Subr and Jan Kautz. 2013. Fourier analysis of stochastic sampling strategies for assessing bias and variance in integration. *To appear in ACM TOG* 32 (2013), 4.

Florent Wachtel, Adrien Pilleboue, David Coeurjolly, Katherine Breeden, Gurprit Singh, Gaël Cathelin, Fernando de Goes, Mathieu Desbrun, and Victor Ostromoukhov. 2014. Fast Tile-based Adaptive Sampling with User-specified Fourier Spectra. *ACM Trans. Graph.* 33, 4, Article 56 (July 2014), 11 pages. DOI:http://dx.doi.org/10.1145/2601097.2601107

Li-Yi Wei. 2008. Parallel Poisson disk sampling. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 20.

Li-Yi Wei. 2010. Multi-class blue noise sampling. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 79.

Yin Xu, Ligang Liu, Craig Gotsman, and Steven J Gortler. 2011. Capacity-constrained Delaunay triangulation for point distributions. *Computers & Graphics* 35, 3 (2011), 510–516.

Dong-Ming Yan and Peter Wonka. 2013. Gap processing for adaptive maximal Poisson-disk sampling. *ACM Transactions on Graphics (TOG)* 32, 5 (2013), 148.

Xiang Ying, Shi-Qing Xin, Qian Sun, and Ying He. 2013. An intrinsic algorithm for parallel poisson disk sampling on arbitrary surfaces. *IEEE transactions on visualization and computer graphics* 19, 9 (2013), 1425–1437.

Cem Yuksel. 2015. Sample Elimination for Generating Poisson Disk Sample Sets. *Comput. Graph. Forum* 34, 2 (May 2015), 25–32. DOI:http://dx.doi.org/10.1111/cgf.12538