Fast Maximal Poisson-Disk Sampling by Randomized Tiling

TONG WANG, REIJI SUDA,
IST, University of Tokyo
Maximal Poisson Disk Sampling
Poisson-disk sampling

- Definition: Poisson-disk sampling pattern is a group of samples with no two of them within a specified distance $r$.
  - Considered a good property in many applications.

- Essentials in many computer graphics applications.
  - Texture generation, Object placement, Rendering and anti-aliasing, Image stippling, etc.
Blue noise property

- One reason that Poisson-disk sampling is good for many applications is because of its blue noise spectral properties [1].

Maximal Poisson-disk samples (MPS)

- A Poisson-disk sample set is maximal coverage *iff*.
  \[ G = \{ j \in D | \forall i \in X, \|i - j\| \geq r \} = \emptyset \]
  For sample set as \( X \) in domain \( D \).

- Guaranteed the quality of samples.
  - No big gaps and clusters.
  - Evenly distributed sampling density.
Generate Poisson-disk samples

- Dart throw
- Relaxation

- Often need many iterations to reach maximal coverage.
Tile based method

Tile Set

• Preprocessing.
  • Generate tile set, etc.

• Choose a tile from tile set.

• Tile on the plane.

Sample Plane
Tile based method

• Extremely fast to fill the sample plane.

But

• Preprocessing takes a long time.

• Regular artifacts are not desirable in many applications.
Add more randomness in tiling

- Design the tile sets and tiling techniques: [1][2].
  - Affect preprocessing and tiling performance.

- Transform and rotate tiles to make tiles non-periodic: [3].
  - But [3] didn’t deal with sample conflicts, which is not acceptable in many applications.

- Randomly subdivide the space and fetch tiles on-the-fly.
  - Our method.

Performance and Regularity

• For Poisson-disk sample generation:

Performance and Regularity Artifact

Approximate Tiling\(^{[3]}\)
Ours
PixelPie\(^{[2]}\)
Flat Quad-tree
Dart Throw\(^{[5]}\)
Wang Tiles\(^{[1]}\)
Regular Tiling
Regular Sampling
Regularity Artifact
Capacity Constrained Relaxation\(^{[4]}\)
Lloyd’s Relaxation

Our algorithm
Algorithm Outline

• Prepare a pre-generated MPS pattern.

• KD-tree based randomized tiling (KDRT) is divide-and-conquer based.
  • Divide: Subdivide the sampling plane using KD-tree.
    • Each leaf node square (2d case) is called a building block of the entire sampling plane.
  • Tile: Fill each building block with points clipped from a pre-generated MPS pattern.
  • Conquer: Eliminate conflicts on the edge, and insert new samples in the gap to ensure maximal coverage property.

• Divide and clipping from MPS pattern are randomized, no regular artifacts.

• Conquering process is determined and easy to convergence.
Prepare pre-generated pattern

- Use any unbiased Poisson-disk sampling method to generate a pattern.
  - Here we use algorithm described in [1] to generate a pattern.

- Define a scale factor *ratio* as how many times samples that the final sampling plane contains compared with the pattern.
  - The final sample set size is controlled by manipulating this parameter.

Divide: Subdivide the sampling plane

- Building blocks of the sampling plane.
- Conflict area that will be processed in conquering.
Tiling: Filling building block with samples
Tiling: Filling building block with samples

MPS Pattern

Tilled Sampling Domain
Tiling: Filling building block with samples
Conquering: Outline

Tile A

Tile B

Conflict Area

Tile A

Tile B

Conquering

Conquered Result

●: Eliminated  ○: Inserted
Conquering: Necessary Redundant

(a) width

height

(b) width + 2r
Conquering: Gap detection

- Gap detection\textsuperscript{[1]}: A gap exist among three disks centered at $i,j,k$ ($i,j,k \in C$), if the circumcenter $c$ of the triangle $i,j,k$ is not covered by any disks.

Conquering: Elimination

- Processing yellow point.
- Eliminate red point to avoid conflicts.
- Insert blue point to ensure maximal coverage.

**Algorithm 3:** Psuedocode of Gap Detection and Insertion

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

- Notice that there are circumstances that if GapDetectionAndInsertion being applied on all sample points $p \in \mathbb{C}$, it is not guaranteed that all gaps could be covered.

(a) Gaps that cannot be covered by single insertion

1. Processing yellow disk
2. Red disk eliminated gap generated
3. Green disk inserted in gap
Conquering: Insertion

- Notice that there are circumstances that if GapDetectionAndInsertion being applied on all sample points $p \in C$, it is not guaranteed that all gaps could be covered.

(b) Gaps that cannot be detected by current processing sample
Conquering: Insertion

All samples are marked as gap detector

New gaps detected by the green disk

All samples are marked as gap detector

Gap detected by triangle $i, j, k$ while dealing one of the vertex

---

**Algorithm 4: Pseudocode of Gap Detection and Insertion**

1. Function EliminateAndInsert $C$;
2. Input : Clipping and Tiling result $C$
3. Output: Maximal Poisson Disk Sample Set $S$
4. **forall** Sample $p$ in conflict area of $C$ **do**
5.   Mark $p$ as GapDetector;
6.   ConflictBuffer $\leftarrow$ RadiusSearch($p, r$);
7.    **forall** conflictPoint in ConflictBuffer **do**
8.      Eliminate conflictPoint from $C$;
9.      TestBuffer $\leftarrow$ RadiusSearch(conflictPoint, $2r$);
10.     Mark all samples in TestBuffer as GapDetector;
11.     GapDetectionAndInsertion(TestBuffer, $p, C$);
12. **end**
13. **end**
14. Mark all inserted samples as GapDetector;
15. **while** True **do**
16.    GapDetectorArray $\leftarrow$ StreamCompaction($C$, isGapDetector($i$));
17.    **forall** Sample $p$ in GapDetectorArray **do**
18.      TestBuffer $\leftarrow$ RadiusSearch(conflictPoint, $4r$);
19.      GapDetectionAndInsertion(TestBuffer, $p, C$);
20.      Mark all inserted samples as GapDetector
21. **end**
22. **if** No new sample point inserted **then**
23.    break;
24. **else**
25.    continue;
Conquering: Insertion

- All gaps can be detected this way.
Results
Results

RMSE: 0.3430
RMSE: 0.2897
RMSE: 0.2972
RMSE: 0.2966
RMSE: 0.0325
RMSE: 0.0308
RMSE: 0.311
RMSE: 0.312

Uniform
De Goes et al., 2012
Ebeida et al., 2012
Ours

~20K/s
~4M/s
~16M/s
Results
Results

Time taken by each procedure to generate 1 million samples.
Combining MPS sample sets

9 MPS set each 1017625 samples

Vertical Zipping

3 MPS set each 2991817 samples

Horizont:al Zipping

Result with 8795941 samples

Vertical Zipping Result

3 MPS set each 2991817 samples

Combining Size

3751372
4689216
5861520
7326900
9158625

CPU Time (ms)

85
102
144
158
202
Conclusion

• KDST is a tile based method that can generate maximal Poisson-disk samples.

• KDST is a divide and conquer method, including:
  • Divide: Randomly divide the sample plane into building blocks.
  • Tiling: Fill the building blocks with samples clipped from a pre-generated pattern.
  • Conquer: Eliminate conflict points and insert new points in the gap.

• Conquering step can be used to combine MPS sample sets together to form a larger one.

• KDST performs fast on CPU and GPU, and is very easy to implement.
• Thanks for all reviewers for your constructive comments.

• Special thanks to Anjul Patney for his efforts to help the author to get the U.S. visa.