Fast Poisson Disk Sampling in Arbitrary Dimensions

Steven Yu-Chih Lin
National Taiwan University
R04922170

Andrew Yu-Chian Wu
National Taiwan University
R05922103

Abstract

In this project, we implemented a fast Poisson-Disk generating algorithm, introduced by Robert Bridson in 2007. This algorithm is a slightly revised version of dart throwing, and it generated samples in linear time, i.e., O(N) complexity. The capability of being easily extended to and implemented in arbitrary dimension is one of the most important characteristics of this algorithm. We generated 4 sets of samples and demonstrated the results in our evaluation.

Introduction

Sampling is an indispensable, yet essential step in rendering. In PBRT, we have seen and reviewed several strategies for sampling, including stratified sampling, best-candidate sampling and adaptive sampling.

Poisson-Disk sampling is highly praised for its well-distributed samples, where all samples are at least γ apart without leaving any holes in any regions, γ being a user-determined parameter. This sort of distribution, often referred to as Blue Noise, is generally considered ideal for applications in rendering. A naïve algorithm to generate Poisson-Disk sampling is the dart-throwing algorithm. However, in dart-throwing, generating random samples and rejecting until a valid one is created might lead to an endless loop and result in undesirable computational cost. Moreover, dart-throwing algorithm potentially under-samples and creates holes in some regions. Many existing algorithms that guarantee better sampling quality in a less time complexity, on the other hand, are not easily generalized to higher dimensions.

Therefore, for applications that require three or more dimensions, e.g., motion blur and depth-of-field, this method proposed by Robert Bridson shed light on generating Blue Noise. The algorithm is guaranteed to run in O(N) time, where N is the number of Poisson-Disk samples. We generated candidates the same way as that introduced by Dunbar and Humphreys in 2006, but instead of calculating the allowed scalloped region, we rejected invalid candidates.

Implementation

We implemented the algorithm in 2D, though it can be easily extended to an arbitrary dimension. The algorithm can be divided into three steps: (i) setting up utility data structures, (ii) initialization for the loop of sample generation, and (iii) the main loop.

(i) Setting up utility data structures

We first created a n-dimensional grid, called background grid, in order to accelerate the spatial searches and calculation. n is the number of dimensions in which we wanted to generate samples, and in our implementation, we set it to two. The size of each cell in the grid is $\gamma/\sqrt{n}$, and therefore each cell would contain at most one sample.

(ii) Initialization for the loop of sample generation

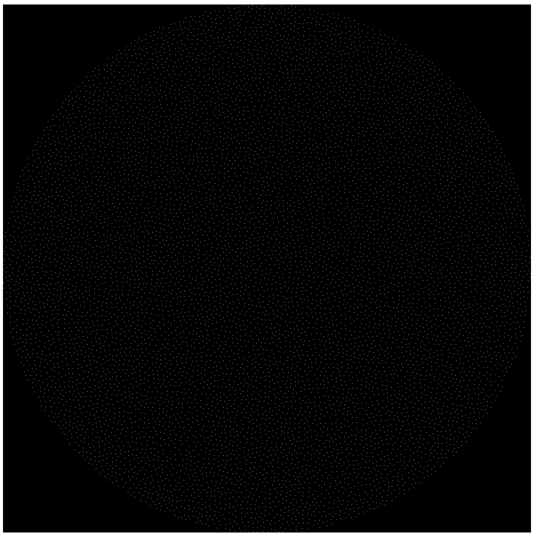
In this step, a randomly created sample is chosen uniformly from our domain, say x_0. We inserted x_0 into the corresponding cell in the background grid and prepared a container, *"active list"*, as mentioned in the original paper, to accommodate samples. The sample x_0 is then inserted into the active list.

(iii) The main loop

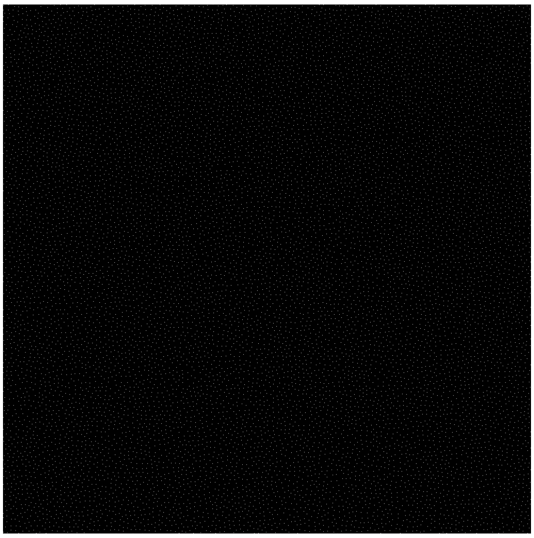
For the main loop of generating samples, while the active list is not empty, we pop a random sample inside the active list, say x_i. And then, we generated k points around x_i by uniformly choosing from the annulus region between radius γ and $2\times\gamma$. For each of the k points, we used the background grid to check whether it was within distance γ of any existing samples. With background grid, we had to test the nearby samples only, which took constant time rather than linear. If a point is sufficiently far from all existing samples after the test, we inserted it into the background grid as well as active list.

Results

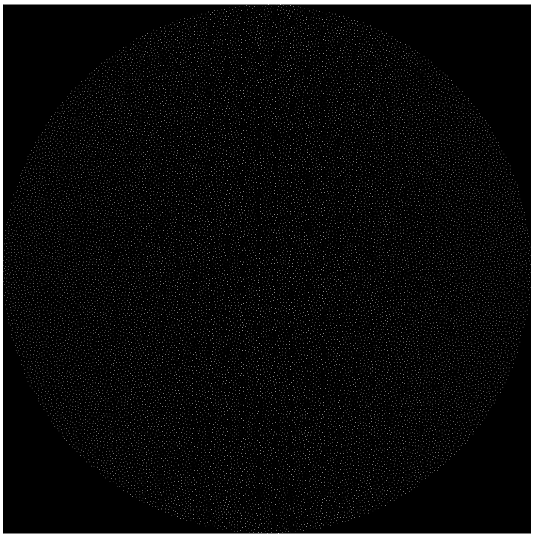
We demonstrated the sampling algorithm in 4 images. The execution time and number of samples generated are both recorded.



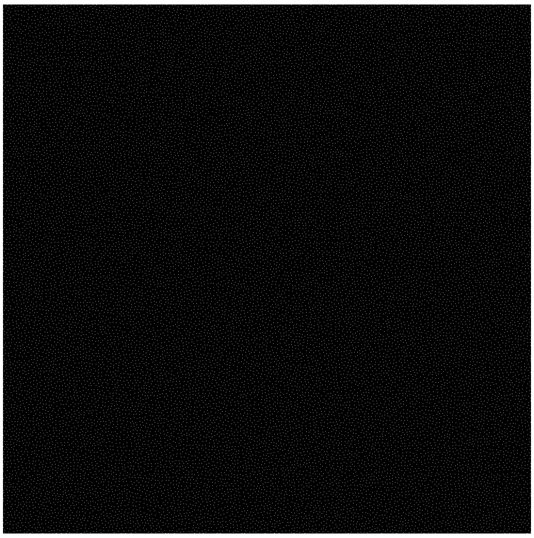
shape: circle
total # of samples: 9832
time for generation: 0.316 sec
(time for drawing: 0.133 sec)



shape: rectangle
total # of samples: 12480
time for generation: 0.406 sec
(time for drawing: 0.227 sec)



shape: circle
total # of samples: 19607
time for generation: 0.633 sec
(time for drawing: 0.183 sec)



shape: rectangle
total # of samples: 24900
time for generation: 0.806 sec
(time for drawing: 0.230 sec)

Usage

With the properly composed `Makefile`, compiling and executing the program is simple and easy. Basically there are two main source files, Poisson.cpp and drawPoisson.py, for generating samples and drawing the results respectively. The following 3 steps help running the program:

1. Use `$make` to compile the C++ program first.

2. Execute with `$make run` command.
3. Lastly, draw the result using `$make draw`.

By default, the C++ program generated samples in a shape of circle, but it can be easily adjusted by setting boolean constant, `Circle`, to `false` in `Poisson.cpp` (line #14). And the constant `k` is set to `30` (line #15) as suggested in the paper.
In addition, to draw the result with white background and black samples, use the command `$make draw-white` instead of `$make draw` in the third step described earlier in this section.

Environment

Operating System: macOS Sierra (10.12.2)
CPU: 2.6 GHz Intel Core i5
RAM: 8GB 1600 MHz DDR3
GPU: Intel Iris 1536 MB
Disk: 256 GB PCIe SSD

References

[1] Daniel Dunbar, Greg Humphreys. A Spatial Data Structure for Fast Poisson-Disk Sample Generation. Proceedings of SIGGRAPH 2006.

[2] Robert Bridson. Fast Poisson Disk Sampling in Arbitrary Dimensions. Proceedings of SIGGRAPH 2007.