2007-09-01

Poisson Disk Point Sets by Hierarchical Dart Throwing

David Cline
clinedav@gmail.com

Parris K. Egbert
egbert@cs.byu.edu

Kenric B. White

Original Publication Citation

David Cline, Kenric B. White, Parris Egbert, "Poisson Disk Point Sets by Hierarchical Dart Throwing", Symposium on Interactive Ray Tracing, pp. 129-132, Sept. 27.

Poisson Disk Point Sets by Hierarchical Dart Throwing

Kenric B. White* David Cline† Parris K. Egbert‡
Brigham Young University Brigham Young University
Arizona State University

Poisson disk point sets are "ideally" generated through a process of dart throwing. The naive dart throwing algorithm is extremely expensive if a maximal set is desired, however. In this paper we present a hierarchical dart throwing procedure which produces point sets that are equivalent to naive dart throwing, but is very fast. The procedure works by intelligently excluding areas known to be fully covered by existing samples. By excluding covered regions, the probability of accepting a thrown dart is greatly increased. Our algorithm is conceptually simple, performs dart throwing in $O(N)$ time and memory, and produces a maximal point set up to the precision of the numbers being used.

Keywords: Poisson disk, sampling

Index Terms: I.3 [Computer Graphics]

1 Introduction

A Poisson-disk point set [1] is defined as a set of points taken from a uniform distribution in which no two points are closer to each other than some minimum distance, r. The point set is said to be *maximal* if there is no empty space left in the sampling domain where new points can be placed without violating the minimum distance property. Poisson-disk point sets are useful in a number of applications, including Monte Carlo sample generation, the placement of plants and other objects in terrain modeling, and image halftoning.

The most straightforward way to create a Poisson disk distribution is through a process of "dart throwing". Samples are drawn successively from a uniform distribution, and those samples that obey the minimum distance property with respect to the samples currently in the set are kept, while the others are discarded. Unfortunately, in the naive dart throwing approach, each dart can hit anywhere in the sampling domain, so that as the sampling area fills up, the likelihood of finding an open spot goes down. This can be particularly problematic if a maximal set is desired, since the probability of throwing an accepted dart goes to zero as the algorithm approaches a maximal set.

Owing to the difficulty of producing maximal Poisson disk distributions by standard dart throwing, researchers have devised a number of other distributions that are easier to generate, but display similar statistical properties [4, 5, 6, 7]. In this article, however, we concentrate on methods that produce distributions which are exactly equivalent to those made by dart throwing.

Recently, several researchers have proposed modified dart throwing algorithms that can produce maximal Poisson disk point sets in $O(N \log N)$ time, where N is the number of points in the set. Dunbar and Humphreys [2] describe a mathematically elegant system to generate Poisson disk point sets. Their algorithm relies on encoding the boundary of an expanding set of points as a group of *scalloped sector* regions. To add a point to the set, they choose one of the scalloped sectors, with probability proportional to its area, and randomly choose a point inside it. They then update the boundary representation to reflect the new point and repeat the process. By choosing a point within a known safe zone (the scalloped sectors), Dunbar and Humphreys' algorithm does not need to explicitly check thrown darts against existing points to ensure the minimum distance requirement. Since a search is required to choose a scalloped sector, Dunbar and Humphreys' algorithm runs in $O(N \log N)$ time. In addition to the scalloped sector algorithm, Dunbar and Humphreys describe a fast *boundary sampling* variant that reduces processing time by shrinking the scalloped sectors to circular arcs, making them much easier to sample. Complexity is further reduced by choosing an arc on which to sample at random rather than according to arc length. While very fast, this variant of the algorithm is not equivalent to standard dart throwing because each disk always abuts at least one other disk, which is not the case in a Poisson disk distribution.

Jones [3] presents a different dart throwing technique that produces Poisson disk distributions in $O(N \log N)$ time. Jones' method works by keeping a Voronoi diagram of the growing point set. New darts are thrown by choosing one of the Voronoi cells with probability proportional to the empty area within it. A random point is then generated within the chosen cell such that, with high probability, the point will obey the minimun distance constraint. As with the method of Dunbar and Humphreys, however, an $O(\log N)$ search is needed to find the right voronoi cell in which to sample. Thus, Jones' algorithm also runs in $O(N \log N)$ time.

In this paper, we present a hierarchical dart throwing technique that is similar to the methods presented by Jones and Dunbar and Humphreys in that it excludes zones from the sample space where darts can be thrown. However, our geometric spatial partitioning allows for faster indexing, yielding a constant time search operation to determine in which region to sample next. Our algorithm can exactly mimic the results of standard dart throwing, and we have strong evidence that the new algorithm runs in $O(N)$ time. In practice our algorithm realizes more than a 30 fold speed increase over current methods.

2 Hierarchical Dart Throwing

Our hierarchical dart throwing algorithm is based on a quadtree subdivision of the sampling domain. We begin by dividing the sampling domain into equal-sized *base level* squares that are placed on an *active list*, meaning that they are not known to be completely covered by samples already in the point set. To maximize perfor-

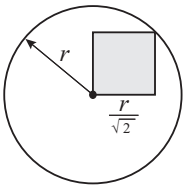


Figure 1: The width of the base level squares should be less than $r/\sqrt{2}$ so that an accepted dart thrown anywhere within the square will completely cover it.

*e-mail: cavkenr@gmail.com
†e-mail: clinedav@gmail.com
‡e-mail: egbert@cs.byu.edu

IEEE/EG Symposium on Interactive Ray Tracing 2007
10 - 12 September, Ulm, Germany
978-1-4244-1629-5/07/$25.00 ©2007 IEEE

Hierarchical Dart Throwing

```
Put base level squares on active list 0 (the base level).
Initialize the point set to be empty.
While there are active squares
  Choose an active square, S, with prob. proportional to area.
  Let i be the index of the active list containing S.
  Remove S from the active lists.
  If S is not covered by a point currently in the point set
    Choose a random point, P, inside square S.
    If P satisfies the minimum distance requirement
      Add P to the point set.
    Else
      Split S into its four child squares.
      Check each child square to see if it is covered.
      Put each non-covered child of S on active list i+1.
```

Figure 2: Hierarchical dart throwing agorithm

mance, the base level squares should be as large as possible while still being covered by a dart thrown anywhere inside them (see figure 1). As a practical matter, however, the base level squares should evenly divide the sampling domain, so we set the size of the base level squares, b_0, to

$$b_0 = \frac{h}{\lceil h\sqrt{2}/r \rceil} \tag{1}$$

where h is the width of the sampling domain.

The algorithm works by throwing darts only in active squares, and deactivating squares as often as possible to reduce dart rejections. When a dart sample is accepted, it gets added to the point set, and the square in which the dart was thrown is removed from the active lists. This square *cannot* have another accepted dart sample in it. On the other hand, if the dart was not accepted, the square gets subdivided, and only those subregions that are not completely covered by an existing point are added back to the active lists. Thus, with each dart thrown−hit or miss−more area is likely to be excluded.

Dart throwing proceeds as follows: First, the algorithm chooses an active square with probability proportional to the square's area, and removes it from its active list. If the square is not covered by (the minimum distance disk around) an existing point, the algorithm generates a new point inside the square with uniform probability (i.e. it throws a dart). The dart is then tested to see if it meets the minimum distance requirement with respect to the current point set. If the minimum distance requirement is met, the algorithm adds the dart to the point set. Otherwise, it subdivides the square into four child squares in quadtree fashion and tests each child to see if it is covered by an existing sample. Those child squares that are not covered are then added back to the active lists. Sampling terminates when there are no more active squares. Figure 2 provides pseudocode for hierarchical dart throwing.

3 Implementation Details

3.1 Storing the points for fast lookup.

In the basic algorithm, we store a copy of the point set in an acceleration grid of cell width r to help with mimimum distance checks. This allows distance checks to be performed within a 3×3 neighborhood of cells. Based on a spacing of r, each grid cell can contain at most 4 points out of the point set, one on each corner. Consequently, we store each grid cell as an array of four points, rather than using a linked structure of some kind, to improve cache performance.

Since the minimum distance check is the main speed bottleneck, we defined a second "accumulator" variant of the algorithm that optimizes the distance check by storing a copy of the points not only in the cell in which they fall, but in the neighboring cells as well. This allows the distance check to be performed by testing against the points in a single grid cell in the acceleration grid, but it quadruples the memory usage of the grid cells.

3.2 Optimizing the square inside circle test.

One frequent operation in the algorithm is to determine if an active square is covered by the minimum distance disk around a point. Rather than testing the disk against each corner of the square, we note that the square will be inside the disk precisely if the corner that is furthest away from the point lies within the disk. The distance from the point to the farthest corner, d_{fc}, can be calculated as follows:

$$d_{fc}^2 = (|x_c - x_p| + b/2)^2 + (|y_c - y_p| + b/2)^2 \tag{2}$$

where $P = (x_p, y_p)$ is the location of the point, $C = (x_c, y_c)$ is the center of the square, and b is the width of the square. Figure 3 shows the geometry for the square inside circle test. Note that any point on the square, such as one of the corners, can be used to look up the neighborhood of points that could cover the square. Note also that the algorithm never tests to see if the disks from several points together cover a square. Instead, we simply subdivide the square if no one point covers it, trusting that each of the subpieces will eventually be covered by the disks from single points.

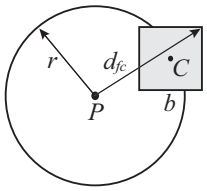


Figure 3: Geometry of the square inside circle test.

3.3 Storing the active squares.

The active squares are stored in a group of lists, with one list for each different size of square. The largest "base level" squares are placed in list 0 at the beginning of the algorithm, and each successive list holds squares that are half the width of those in the previous level. Since each list contains squares of the same size, a square can be encoded as a single point, in our case the minimal xy corner. Thus, each list of active squares is really just a list of points.

In theory, the number of active square levels could be unbounded, but in practice we only need enough levels to account for the precision of the numbers being used. This boils down to having about the same number of lists as bits of precision, 23 for single precision and 52 for double precision. In implementation, the algorithm simply discards any squares that get subdivided beyond some predetermined level. (It has been our experience that squares sometimes get discarded when working with single precision, but we have never had to discard squares in the double precision version of the algorithm. In fact, we have never seen any squares beyond level 32 in any run of the algorithm.)

Along with the lists of squares, we keep a running total of the area of all the active squares, a_{tot}. To choose a square in which to sample, the algorithm generates a random number x in $[0,1)$ and multiplies it by a_{tot}. A linear search is then performed to find the list

for which the area of the list and previous lists exceeds the number. In other words, find m such that

$$\sum_{i=0}^{m-1} a_i \;\le\; a_{tot}x \;<\; \sum_{i=0}^{m} a_i, \tag{3}$$

where m is the index of the list for which we are searching and a_i is the area of all the squares in list i. The algorithm then chooses a random square within list m in which to sample.

While the linear search just described may seem inefficient, it has several advantages. First, because the number of lists depends on the precision of the numbers rather than the total number of squares, the search runs in constant time for a given precision. Even if unlimited precision were available, however, the average number of checks would still be bounded since (based on all of our studies) the number of squares generated at each level drops off exponentially after the first few levels. As a consequence hierarchical dart throwing can create a maximal point set in $O(N)$ time rather than $O(N \log N)$. Furthermore, the average number of list checks that must be performed during a search is quite small (less than 3 in our tests).

4 Results

4.1 Speed Comparison

Figure 4 shows the number of points generated per second using both the standard, low memory version of our algorithm, and a higher memory, accumulator version. These results are plotted along with similar numbers for the Voronoi-diagram based method of Jones [3], and the scalloped sector and boundary sampling methods presented by Dunbar and Humphreys [2]. As can be seen in the figure, our method handily outperforms both of the other methods that are equivalent to naive dart throwing. For example, the faster version of our algorithm was able to generate a point set with 100,000 points at a rate of more than 211,000 points per second, while Jones' method produced about 4,050 points per second, and the scalloped sector method of Dunbar and Humphreys produced only 778 points per second. In fact, the faster version of our algorithm is on par with Dunbar and Humphreys' boundary sampling method in terms of speed, but we produce a true Poisson disk distribution whereas they only approximate one.

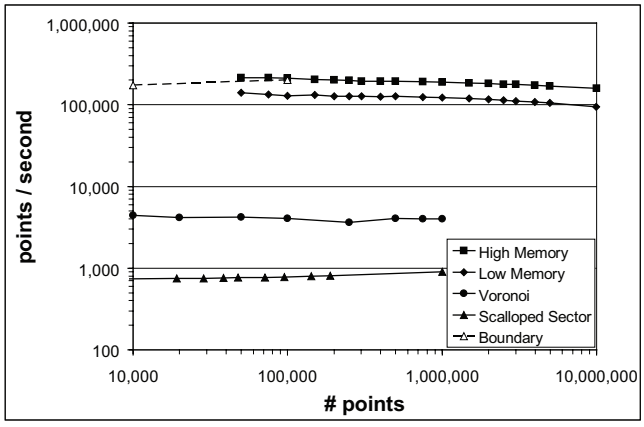


Figure 4: Timings for hierarchical dart throwing compared with the methods of Jones (Voronoi) and Dunbar and Humphreys (Scalloped Sector, Boundary). The graph plots the number of points generated per second for different sized point sets. Results shown are for a 2.13 Ghz Intel Core 2 Duo.

4.2 Memory usage

The main data structures used in hierarchical dart throwing, besides the point set itself, are the lists of active squares and the acceleration grid. Our studies indicate that the total number of active squares generated during a run of the algorithm is consistently between 9 and 10 times the final number of points in the point set (see figure 6. Additionally, with a radius of r the number of cells in the acceleration grid is roughly 1.25 times the final number of points. Based on these figures, the memory overhead of hierarchical dart throwing per accepted dart ranges from 120 bytes per point for the low memory, single precision version, to 416 bytes per point for the high memory, double precision version.

4.3 Wavefront Sampling

Using hierarchical dart throwing as described in sections 2 and 3 we were able to produce point sets with up to about 10 million points on a PC with 2 gigabytes of memory. With the goal of making larger point sets, we modified the algorithm in several ways to decrease its memory footprint.

First, we increased the size of the cells in the acceleration grid, using the accumulator variant. Instead of allocating memory for the maximum number of points that could be in a cell, we only allocated enough memory for the maximum number *likely* to be in the cell. For those few cells that exceeded this number, we created an overflow buffer to contain the extra points. Restructuring the acceleration grid in this manner can reduce its memory requirements to just a few percent more than simply storing the final point set, depending on the size of the grid cells. One surprising result was how little impact changing the acceleration grid cell size had on the performance of the algorithm. Even with cells set to size $20r \times 20r$, each of which contained more than 360 points, the algorithm ran only about 2.5 times slower than with $r \times r$ cells which only contained 16 points.

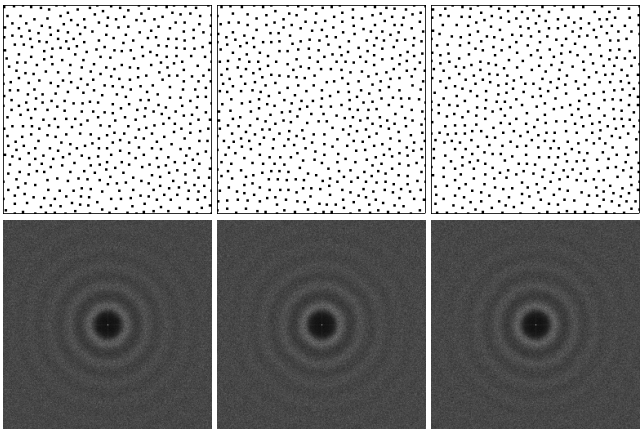
To reduce the memory used by active squares, we changed the way in which the base level squares are added to the active lists. Our approach was inspired by the scalloped sector algorithm of Dunbar and Humphreys. Rather than adding all of the base level squares up front, we add them in groups of 3×3 cells as needed. The algorithm starts by adding the 3×3 group of cells at the bottom corner of the sampling domain. Then, whenever a dart is accepted, the base level squares in the nine 3×3 groups surrounding the dart are added to the active lists, if they haven't been added already. This produces an expanding wavefront of active base level cells, so we call the method "wavefront sampling". In this manner all area within $2r$ of the current point set is included in the active sampling area. We keep track of which 3×3 cell groups have been added to the active lists using a map that contains one bit per cell group. Note that even though explicitly we are only changing the manner in which base level cells are created, the maximum number of active cells at higher levels is also reduced appreciably.

Using the wavefront sampling method, we have been able to produce point sets with up to 95 million points on our 2 gigabyte test machine, and up to 750 million points on a machine with 32 gigabytes of RAM. We believe this to be by far the largest true Poisson disk distribution ever produced.

4.4 Spectral Equivalence to Naive Dart Throwing

Figure 5 shows example point sets and averaged Fourier transforms for naive dart throwing compared with hierarchical dart throwing and our wavefront sampling method. Based on the Fourier transforms, the spectral properties of hierarchical darth throwing and wavefront sampling both appear to be identical to naive dart throwing. This should not be surprising for hierarchical dart throwing, since it produces accepted darts in an equivalent manner to naive dart throwing (i.e. Darts are thrown at random within the entire open domain.) Wavefront sampling, on the other hand, does not throw

darts within the entire open domain. Even so, point sets generated by the wavefront method appear to be indistinguishable spectrally from standard dart throwing.



Naive dart throwing Hierarchical dart throwing Wavefront sampling

Figure 5: Example point sets and averaged Fourier transforms for naive dart throwing, and our hierarchical dart throwing and the wavefront sampling methods.

4.5 Assertion of Linear Order

While we cannot provide a rigorous proof, we are convinced that hierarchical dart throwing is $O(N)$ in both space and time on average, even ignoring precision issues. This result would follow directly if we could demonstrate either that

- the total number of squares at each successive level decreases exponentially,

or that

- the average number of squares generated per accepted dart is constant and the average number of checks needed to choose a square is also constant.

All of our statistics suggest that both of these conditions hold. Figure 6 shows the second condition visually, plotting the average number of squares per accepted dart and the average number of checks needed to select an active square for different point set sizes. As can be seen, both the number of squares per accepted dart and the number of checks to choose a square remain essentially constant without regard to the point set size.

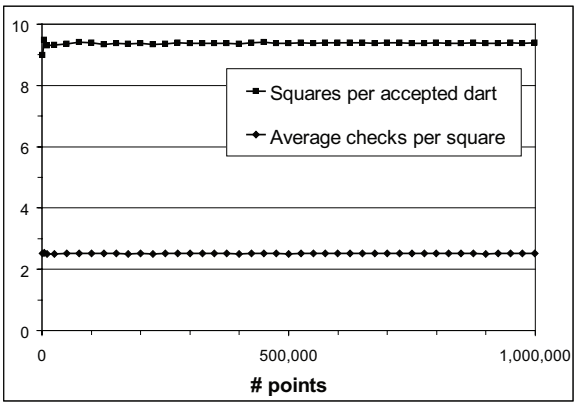


Figure 6: The graph plots the total number of active squares per accepted dart and the average number of checks needed to choose a square in which to sample, for different point set sizes.

5 Conclusion

This paper presents a new hierarchical dart throwing algorithm for creating Poisson disk point sets. The new procedure creates maximal point sets that are exactly equavalent to naive dart throwing, while being both simple and very fast. Compared with current state of the art dart throwing methods, hierarchical dart throwing is one to two orders of magnitude faster.

References

[1] Robert L. Cook. Stochastic sampling in computer graphics. *Computer Graphics (Proceedings of SIGGRAPH '86)*, 5(1):51–72, 1986.

[2] Daniel Dunbar and Greg Humphreys. A spatial data structure for fast poisson-disk sample generation. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, 25(3):503–508, 2006.

[3] Thouis R. Jones. Efficient generation of poisson-disk sampling patterns. *Journal of Graphics Tools*, 11(2):27–36, 2006.

[4] S. Lloyd. An optimization approach to relaxation labeling algorithms. *Image and Vision Computing*, 1(2):85–91, May 1983.

[5] Don P. Mitchell. Generating antialiased images at low sampling densities. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, pages 65–72, New York, NY, USA, 1987.

[6] Don P. Mitchell. Spectrally optimal sampling for distribution ray tracing. *Computer Graphics (Proceedings of SIGGRAPH '91)*, 25(4):157–164, 1991.

[7] Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. Fast hierarchical importance sampling with blue noise properties. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004)*, 23(3):488–495, 2004.