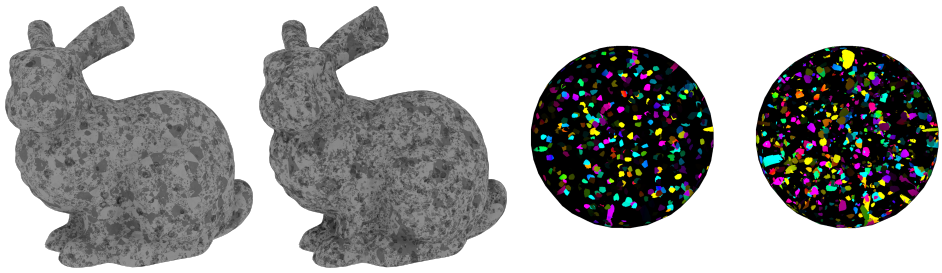
Implementation of Fast and Adaptive Procedural Cellular Noise

Théo Jonchier
OLM Digital Inc.
ASALI-SIR, XLIM

Alexandre Derouet-Jourdan
OLM Digital Inc.

Marc Salvati
OLM Digital Inc.



(a) (b) (c) (d)

Figure 1. 3D cellular noise renderings. Rock shader with (a) uniform and (b) non-uniform cellular noise. Flake shader with (c) uniform and (d) non-uniform cellular noise. Non-uniform cellular noise increases variation in the output.

Abstract

Cellular noise as defined by Worley is a useful tool to render natural phenomena, such as skin cells, reptiles scales, or rock minerals. It is computed at each position in texture space by finding the closest feature points in a grid. In a preliminary work, we showed in 2D space how to obtain non-uniform distribution of points by subdividing cells in the grid and that there is an optimal traversal order of the grid cells. In this paper, we generalize these results to higher dimensions, and we give details about their implementation. Our optimal traversal in 3D proves to be 15% to 20% faster than the standard Worley algorithm.

1. Introduction

In his seminal paper [Worley 1996], Steven Worley introduced the so-called procedural cellular noise to render several natural phenomena, such as skin cells, reptiles scales, or rock minerals. The procedural cellular noise is constructed using a set of feature points spread through space and a set of basis functions F_n that associate to

a location x, the distance $F_n(x)$ to the nth-closest feature point. The basis functions F_n are linearly combined to form the final result.

The procedural cellular noise is designed to distribute the feature points in a grid's cells, each cell containing a random number of points. This number, as well as the positions of the feature points in the cell are computed on-the fly based on the location of the cell. For a given location x, it is possible to compute F_1 to F_4 by visiting a restricted neighborhood (a disk of radius 2) of cells around the one containing x.

In a preliminary work restricted to a two-dimensional space [Jonchier et al. 2016], we showed that it was possible to optimize the order in which the neighbor cells are visited to compute F_n. Moreover, by using a quadtree structure, where a cell either contains a feature point or is cut in a 2×2 sub-grid, we allowed for non-uniform distribution of feature points while still conserving a fast computation.

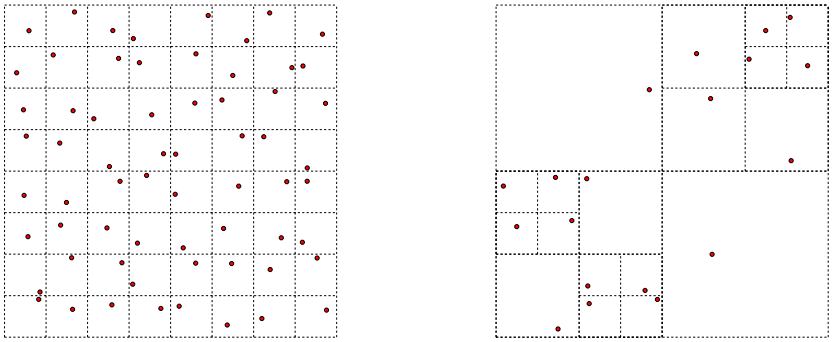
In the present paper, we show how the traversal of the adaptive grid can be optimized in 3D and 4D by using the same type of optimal order of neighbor cell traversals. We detail how we implement this optimization and, in particular, how to automatically pre-compute the traversal orders.

2. Procedural Cellular Noise

In this section, we recall the basic idea about procedural cellular noise and its adaptive version. We will start with a regular grid before explaining how to extend the concept to a multi-resolution grid. In this description, we focus on the case of one feature point per grid cell. The case of multiple feature points per grid cell as originally presented in [Worley 1996] is similar.

2.1. Cellular Noise with Regular Grid

Typical cellular noise uses a regular grid (see Figure 2 (a)) where querying a point is straightforward and given in the following steps.



(a) (b)

Figure 2. (a) Regular grid; (b) quadtree grid.

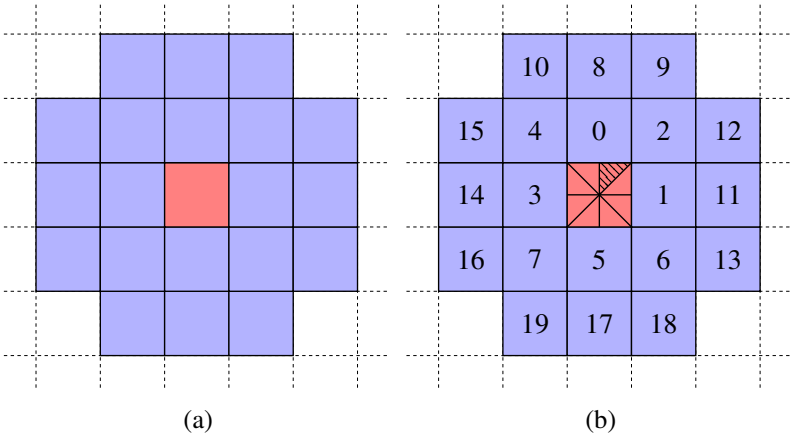


Figure 3. (a) Neighborhood (in blue) of a square s (in red); (b) Optimal order of neighbor traversal for query located in the shaded area of the central square. The order is symmetrical for all other regions.

1. Compute the square (i, j) that contains the query point (typically divide the coordinates by the square size or use the integer part);

2. Use a 2D hash function to obtain the feature point of the square (i, j) and compute the distance to the query point;

3. Visit the neighborhood (see Figure 3 (a)) and use the hash function to compute the feature points and their respective distance to the query point.

To reduce the number of visited neighbors, [Worley 1996] proposes to compute the distance from the query point to the boundary of rows of neighbors. If the distance to the closest current point is smaller, then the entire row can be discarded. The simple idea of [Jonchier et al. 2016] is to use the distance to each neighboring square boundary and visit each from the closest to the furthest (see Figure 3 (b)). As soon as the feature point is closer than a given neighbor boundary, the rest of the neighbors can be discarded. We call this distance a *threshold* as it triggers a break in the visit of the neighbors. For efficiency, the threshold is computed as the Chebyshev distance ($d(p, q) = \max |p_i - q_i|$) from the query point to the closest point on the neighbor-cell boundary. The cell boundary being axis-aligned, computing this distance boils down to computing the maximum coordinate of the query point in absolute value. What makes this technique work efficiently is that there is a limited number of possible neighbor traversal orders. They are given by the first and second neighbor squares. It allows us to cut the square containing the query point into eight regions (see Figure 3 (b)), each one associated to a precomputed traversal order.

2.2. Cellular Noise with Quadtree Grid

To obtain non-uniformity in the distribution of feature points, we use a quadtree structure (see Figure 2 (b)). Each square can either contain a point or be split in four. First we start by locating the query point in the quadtree structure, meaning finding the square that contains the query point and a feature point:

1. Start at level $k = 0$;
2. Compute the square (i, j, k) to which the query point belongs;
3. Use a 3D hash function for key (i, j, k) to determine if the square contains a point or need to be split;
4. If the square contains a point, use a 3D hash function to obtain the feature point of the square (i, j, k);
5. If the square need to be split, iterate from step 2 with $k = k + 1$.

When computing the closest feature point, we need to visit neighbors of the square where the query point is located. We go through the neighbors at the same level l as the square in the optimal order as defined in Figure 3 (b).

1. Compute the center of the neighbor at the level l;
2. Starting with $k = 0$, use a hash function to know if the corresponding square splits or not;
3. If it splits and $k < l$ iterate from step 2, with $k = k + 1$;
4. If it splits and $k >= l$, process the four children in optimal order;

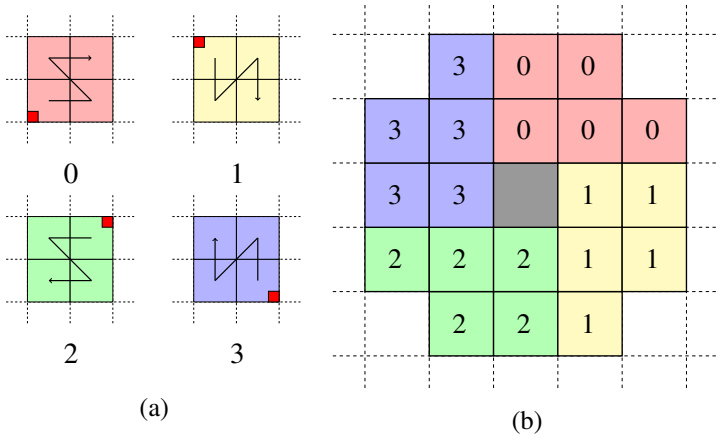


Figure 4. (a) Traversal patterns for sub-square; (b) Relative position to the square containing the query point (in gray). The number determines the traversal pattern to use when visiting the children of the squares in the neighborhood.

5. If it does not split, use a hash function to compute the feature point and update the threshold if necessary.

The optimal order in Step 4 to process children of a square is shown in Figure 4. It follows the same principle, that is, to visit the squares in the increasing order of their distance to the query point. The recursive process of the children is similar to that of the regular grid with the update of the threshold; the process stops when we are sure that there are not any closer points.

3. Implementing Optimal Traversal

In this section we explain how to implement efficiently the optimal order of traversal as given in Section 2 . Section 3.1 considers the 2D case, and Section 3.2 treats its generalization to higher dimensions.

3.1. 2D implementation

We implement the optimal order using the following tables:

- Optimal traversal orders table: $orders[orderId]$, the sequence of the neighbor ids (see Figure 5) for the given order id. Computed statically.
- Relative coordinates table: $neighbors[neighborId]$, offsets to compute the neighbor coordinates in the virtual grid. Computed statically.
- Depth order table: $depth\_orders[neighborId]$: the sequence of traversal orders of children of a square according to Figure 4. Computed statically.
- Threshold table: $lower\_bounds[neighborId]$. Computed dynamically.

In Figure 5, we introduce the primary and secondary neighbors. From the Chebyshev distance properties, we deduce that we never need to compute the thresholds of

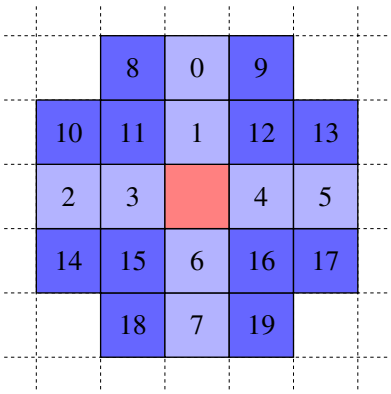


Figure 5. Medium blue indicates a primary neighbor, dark blue indicates a secondary neighbor, with the static index for primary neighbors (0 to 7) and secondary neighbors (8 to 19).

```
float F1(Point q) {
    Cell c = retrieve_cell(q);
    float u = q.u - c.i;
    float v = q.v - c.j;
    int traversal_order_id = traversal_id(u, v);
    float d = distance(q, point_at(c));
    for(int i = 0 ; i < neighbors_count; ++i) {
        int neighbor_id = orders[traversal_order_id][i];
        Cell offset = neighbors[neighbor_id];
        Cell neighbor = c + offset;
        float lower_bound = 0.0;
        if (is_primary(neighbor_id))
            lower_bound = lower_bounds[neighbor_id];
        if (d > lower_bound) {
            d = min(compute_distance(q, point_at(neighbor)), d);
        } else {
            break;
        }
    }
    return d;
}
```

Listing 1. Distance to the closest feature point, following the optimal order.

the secondary neighbors. Indeed, the threshold of cell 12 in Figure 5 is equal to the max of the thresholds of cells 1 and 4. It means that when we visit both of these cells, then, necessarily, we visit cell 12, making the threshold of 12 inconsequential. Then, given the tables, Listing 1 explains how to compute the distance to the closest feature point using optimal order. Most of the tables can be pre-computed, reducing what is computed dynamically and therefore the overall computation time by 10% to 20% as shown in Section 4. The function $is\_primary$ returns whether a neighbor is a primary neighbor or not based on its id. The integer $neighbors\_count$ represents the number of neighbors we may visit to find the closest feature point. It is equal to 20 in 2D.

3.2. Generalization to Higher Dimension

To extend the cellular noise to 3D/4D, we just need to compute the 3D/4D version of the tables introduced in Section 3.1. Listing 1 may then be used with very few trivial modifications. In 2D, a square is divided in eight different regions (see Figure 3 (b)), for which we need to create the optimal order of the 20 neighbors which is manageable by hand. In 3D, the number of regions is six times bigger: 48 traversal orders of 116 neighbors. In 4D, the number gets even larger with 384 possible traversal orders of 608 neighbors. Those tables need to be automatically generated. The relative coordinates table generation is straightforward. We compute the optimal traversal orders table by sorting the cubes of the neighborhood by their Chebyshev distance to

the center of each region (corresponding to a traversal order). When two neighbors have the same Chebyshev distance, the primary neighbor should appear before the secondary. The primary and secondary neighbors definition in higher dimension is a straightforward generalization of the 2D case where the primaries are located on the axis (see Figure 5).

As for the depth order table, the idea is similar to the neighbor optimal traversal order. We create an order with increasing Chebyshev distance of the children to the query-point square.

For more details, a Python 3 script that computes all the necessary optimal order tables, as well as a C program illustrating how to use them by generating a picture of a 2D cellular noise are found in the supplemental materials. The C program contains Listing 1 with only minor modifications necessary for a practical implementation.

4. Results

For our experiments we use a Dell Precision T1700 with a XeonE3-1241v3 3.5GHz CPU. The production results are rendered using Arnold 4.2.13.4.

The hash function we use is based on a multiplicative congruential pseudo-random number generator (MCG) [Park and Miller 1988]. We seed it by multiplying the input numbers by large prime and perform a xor operation. The result of the hash function is given by the first iteration of the MCG.

We compare our optimal traversal order to the traversal order defined by Worley in [Worley 1996]. We restrict the cellular noise to using only one feature point per grid cell. We show the results in Table 1. With our optimal order, we reduce the computation time by around 15%.

We have two production shaders (Rock and Flakes) that intensively use 3D cellular noise. Thanks to our generalization of [Jonchier et al. 2016] to 3D, we improve the rendering time and enable more variation in the details with the adaptive feature. Comparison of our production quality rendering using the adaptive feature are shown in Figure 1.

In Table 2, we compare our new cellular noise implementation to our own standard Worley implementation. We also compare with another hash function based on

n	Worley	Ours
1	7.8s	6.5s ($\times$0.83)
2	10.44s	8.93s ($\times$0.86)
3	13.69s	11.15s ($\times$0.81)
4	18.27s	14.41s ($\times$0.79)

Table 1. Computation times (in seconds, for 16M queries) of F_n in 3D following Worley traversal and the optimal traversal.

Hash Function	Permutation Table		MCG	
traversal order	Worley	Ours	Worley	Ours
Rock ($4k$, 4spp)	103s	95s ($\times 0.92$)	113s	103s ($\times 0.91$)
Flake ($4k$, 1spp)	66s	54s ($\times 0.82$)	85s	72s ($\times 0.85$)

Table 2. Computation times (in seconds) of our previous and new cellular noises. We compare two hash functions: linear congruential MCG and permutation table.

a permutation table inspired by [Perlin 2002]. We compute the seed the same way as the MCG-based hash, but the final result is obtained by performing look-ups in a permutation table. Using any hash function, we get roughly a 10% to 20% improvement in computation time, depending on the number of calls to the cellular function. We can also see the importance of the hash function in the computation time.

We also compare our implementation to the reference implementation of the production renderer Arnold. The shader API provides a simple function (AiCellular) to create 3D cellular noise, that is stated as a Worley noise implementation. The computation times are given in Table 3. The difference in the hash function computation time becomes less noticeable because of the renderer overhead (more than 85%). However, we can clearly see that Arnold could benefit from our optimal order to increase its cellular noise performance. We get 20% improvement for the overall rendering. The renderer overhead set aside, our noise proves to be a lot faster. With two levels in the adaptive grid, we get computation times similar to Arnold ones. Our cellular noise is then more efficient, while still giving more expressiveness to the artists.

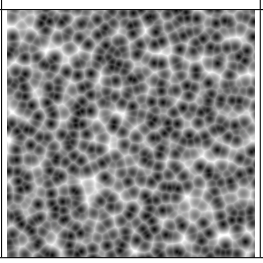
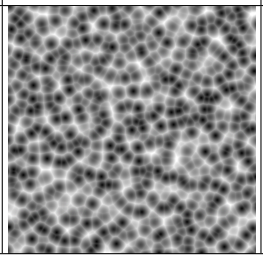
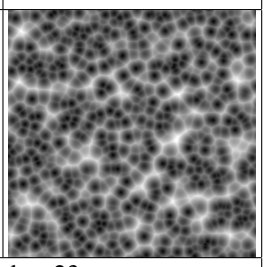
Shaders	Arnold	Ours with Table / MCG	Ours with Two Levels MCG
Time(s) 4k, 4spp	33	26 / 27	31
Time(s) no overhead	10	3 / 4	8
Output			
Render time for Arnold, 4k, 4spp, empty shader: 23s			

Table 3. Comparison between our algorithm and Arnold implementation.

5. Conclusion

We presented in this paper how to generalize and implement the optimal order traversal of neighbors in the computation of grid-based cellular noise. The optimal order reduces the computation time of the cellular noise by around 15%.

We believe there is still room for further optimization. We could compute more precise thresholds during the depth traversal depending on the query-point position and then reduce unnecessary distances computations. Also, when we visit the neighbor squares, we might encounter a square that has a lower level. In this case, we try to visit a square that is smaller than the actual neighbor containing feature points. The problem in that case is that we might visit the actual neighbor square with a lower level several times. Finding a way to reduce the number of visits would improve the overall computation time.

References

JONCHIER, T., SALVATI, M., AND DEROUET-JOURDAN, A. 2016. Procedural Non Uniform Cellular Noise. In *Symposium on Mathematical Progress in Expressive Image Synthesis (MEIS2016)*, Springer, Berlin, vol. 69 of *MI Lecture Note Series*, 28–39. URL: https://www.springer.com/us/book/9789811328497. 36, 37, 41

PARK, S. K., AND MILLER, K. W. 1988. Random number generators: Good ones are hard to find. *Commun. ACM 31*, 10 (Oct.), 1192–1201. URL: https://dl.acm.org/citation.cfm?id=63042. 41

PERLIN, K. 2002. Improving noise. *ACM Transactions on Graphics (TOG) 21*, 3, 681–682. URL: https://dl.acm.org/citation.cfm?id=566636. 42

WORLEY, S. 1996. A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, SIGGRAPH '96, 291–294. URL: http://doi.acm.org/10.1145/237170.237267. 35, 36, 37, 41

Index of Supplemental Materials

We provide a Python 3 script `optimal_order_by_sort.py` at http://www.jcgt.org/published/0008/01/02/code.zip that implements the computation of all the optimal orders in dimensions 2, 3 and 4. We also provide a C implementation of the runtime, restricted to the 2D case, to illustrate how the optimal orders can be used.

Some results of the adaptive cellular noise can be seen in a short video at http://www.jcgt.org/published/0008/01/02/optimal_cellular_noise_video.mp4.

Author Contact Information

Théo Jonchier
OLM Digital Inc. /
ASALI-SIR, XLIM
123, avenue Albert Thomas
87060 Limoges CEDEX
France
theo.jonchier@unilim.fr

Alexandre Derouet-Jourdan
OLM Digital Inc.
Mikami Bldg, 2F 1-18-10
Wakabayashi
Setagaya-ku, Tokyo 154-0023
Japan
alexandre.derouet-jourdan@olm.co.jp

Marc Salvati
OLM Digital Inc.
Mikami Bldg, 2F 1-18-10
Wakabayashi
Setagaya-ku, Tokyo 154-0023
Japan
salvati.marc@olm.co.jp

Théo Jonchier, Alexandre Derouet-Jourdan, Marc Salvati, Implementation of Fast and Adaptive Procedural Cellular Noise, *Journal of Computer Graphics Techniques (JCGT)*, vol. 8, no. 1, 35–44, 2019
http://jcgt.org/published/0008/01/02/

Received:	2018-06-07		
Recommended:	2018-10-08	Corresponding Editor:	Marc Olano
Published:	2019-01-17	Editor-in-Chief:	Marc Olano