

Long-Period Hash Functions for Procedural Texturing

Ares Lagae & Philip Dutré

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200 A, B-3001 Heverlee, Belgium
Email: {ares.lagae, philip.dutre}@cs.kuleuven.be

Abstract

Procedural texturing is commonly used to increase visual complexity and realism in rendered scenes. Procedural texture basis functions, such as Perlin’s noise function, are often built on hash functions defined over the integer lattice. These hash functions are usually based on permutation tables and have a short period. This causes procedural textures to repeat. In this paper, we present a method for constructing hash functions with long periods. Our method is based on techniques for constructing long-period pseudo-random number generators. The hash functions we propose are almost as efficient as the traditional ones, but allow to generate procedural textures with very large periods. This increases the visual complexity and realism of procedural textures, and enables robust implementations of texture basis functions.

1 Introduction

Since the introduction of solid texturing in 1985 by Perlin [7] and Peachy [6], procedural texturing and modeling has become an active area of research within computer graphics. With the advent of complex and realistic scenes, procedural techniques are gaining importance and popularity [11, 1]. Because of their widespread use, industrial-strength implementations of procedural tools are needed. Almost all procedural methods, such as Perlin’s noise function and the procedural object distribution function of Lagae and Dutré [2, 3], require some form of pseudo-randomness. This is often introduced using a hash function defined over the integer lattice, based on a permutation table. These hash function have short periods (typically 256), which causes the generated textures to repeat. In this paper we introduce long-period hash functions to address this problem. These improved hash functions increase the quality of implementation of existing techniques for procedural texturing and modeling, and enable

more realistic and complex scenes.

2 Hash Functions based on Permutation Tables

Traditional hash functions are typically based on permutation tables. A permutation table P of size N contains a random permutation of the integers $0, 1, \dots, N - 1$.

A random permutation of the elements $0, 1, \dots, N - 1$ can easily be computed by starting with the permutation $\{0, 1, \dots, N - 1\}$, and then exchanging the i th element with a randomly selected element from the first i elements, for i in $0, 1, \dots, N - 2$ [9].

A one-dimensional hash function $hash$ is then defined as

$$hash(x) = P[x \% N], \quad (1)$$

where x is an integer, $P[i]$ denotes the i th element of P , and $\%$ indicates the modulo operation. The hash function is a periodic function with period N . The range of the hash function is N .

A two-dimensional hash function $hash$ can be defined using two permutation tables P_x and P_y of size N

$$hash(x, y) = (P_x[x \% N] + P_y[y \% N]) \% N. \quad (2)$$

The hash function is a periodic function with period (N, N) . The range of the hash function is N .

In order to avoid the storage of two permutation tables, the same permutation table is often used twice

$$hash(x, y) = P[(P[x \% N] + y) \% N]. \quad (3)$$

This is also a periodic function with period (N, N) and range N . This technique extends to multiple dimensions and is used to implement Perlin’s noise function [7, 8], the procedural object distribution function of Lagae and Dutré [2, 3], and has several other applications.

3 Long-Period Hash Functions based on Permutation Tables

The key observation for constructing long-period hash functions is that hash functions based on permutation tables are periodic functions, and that the addition of periodic functions yields a new periodic function with a larger period.

A one-dimensional long-period hash function *hash* is defined as

$$\text{hash}(x) = \left(\sum_{i=1}^M P_i[x \% N_i] \right) \% N_j, \quad (4)$$

where x is an integer, $P_1 \dots P_M$ are M permutation tables with size $N_1 \dots N_M$, and N_j is one of these sizes.

A two-dimensional long-period hash function *hash* is defined similarly

$$\text{hash}(x, y) = \left(\sum_{i=1}^M P_i[(P_i[x \% N_i] + y) \% N_i] \right) \% N_j. \quad (5)$$

We also call these long-period hash functions combined hash functions.

A similar technique was used by L'Ecuyer [4] in 1988 to construct a long-period pseudo-random number generator by combining several linear congruential generators. However, with the advent of recent pseudo-random number generators [5], this technique has largely become obsolete in its original context.

3.1 Period and Range

The period of a combined hash function is the least common multiple of the periods of the combining hash functions. In order to maximize the period of the combined hash function, the periods of the combining hash functions should be relatively prime.

The range of the combined hash function is determined by the final modulo divisor N_j , which is one of $N_1 \dots N_M$. Note that, in contrast with traditional hash functions, the range and period of the combined hash functions is different.

3.2 Distribution

Traditional hash functions produce uniformly distributed values over the integer lattice, and most applications of these hash functions rely on this property. The following theorem shows that combined hash functions will also produce uniformly distributed values.

Theorem 1. *If $X_1 \dots X_N$ are N independent discrete random variables, such that X_1 is uniform between 0 and $d-1$, where d is a positive integer, then*

$$X = \left(\sum_{i=1}^N X_i \right) \% d \quad (6)$$

follows a discrete uniform probability law between 0 and $d-1$.

Note that there are no requirements on the distribution of the variables $X_2 \dots X_N$. This theorem was first hinted at by Wichmann and Hill [10], and proved later by L'Ecuyer [4].

For long-period hash functions, all combining hash functions are uniformly distributed. Therefore, the final modulo divisor N_j can be any one of $N_1 \dots N_M$.

However, some care must be taken in selecting the appropriate permutation table sizes. Suppose a combined hash function is build from a permutation table of size 128 and a permutation table of size 3. The period of the combined hash function is 384. However, if the final modulo divisor is 128, then the period will contain three almost identical parts. This is because the ranges of the two combining hash functions are too different. Note that, if the final modulo divisor is 3, this will not be the case. From this, we learn that the sizes of the permutation tables should not differ too much.

3.3 Efficiency

The time needed to evaluate a combined hash function is roughly proportional to the number of combining hash functions. This does not mean that an application that uses a long-period hash function consisting of N combining hash function will be N times slower than the same application using a traditional hash function. In most applications, the evaluation of the hash function is only a small part of the total computation time. Also note that combined hash functions typically have a smaller memory footprint, which improves the cache efficiency of multiple permutation table lookups.

3.4 Design

The design of a combined hash function is determined by four factors: the required range of the hash function, the period of the hash function, the memory footprint of the combined permutation tables, and the time needed to evaluate the hash func-

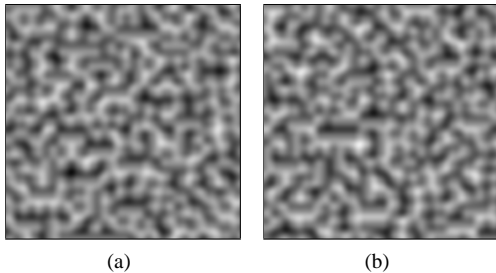


Figure 1: Two-dimensional Perlin noise generated using (a) the hash function proposed in [8] and (b) the long-period hash function proposed in this paper. As expected, the two images have the same appearance. However, the noise function on the left repeats every 256 units, while the noise function on the right has a period of 739.024.

tion. We recommend the strategy outlined below to design a long-period hash function.

First, determine the required range of the hash function. This fixes the final modulo divisor N_j and thus the size of one permutation table.

The required range of the hash function is typically small. For example, Perlin’s noise function [8] uses the lower 4 bits of the hashes (16 values) to choose amongst one of 12 vectors at each integer lattice point. The procedural object distribution function of Lagae and Dutré [2, 3] uses the hash function to choose a random color at each integer lattice point in order to construct a random Wang tiling or corner tiling. The number of colors is typically very small (2, 3, 4, 6 or 8).

If the range of the hash function is too small, or if the range should be adjustable at runtime, for example for the texture basis function of Lagae and Dutré [2, 3], then use a multiple of the range of the hash function and apply an additional final modulo divisor.

Next, choose a number of permutation table sizes for the rest of the combining hash function. The sizes should not differ too much, in order to ensure a uniform distribution. The sizes should also be relatively prime, and as close as possible, to maximize the period. An easy way to choose the permutation table sizes is to take the primes closest to the range of the hash function. If the primes are not a factor of the range, then the period of the combined hash function will be the product of the permutation table sizes.

The number of permutation tables will determine

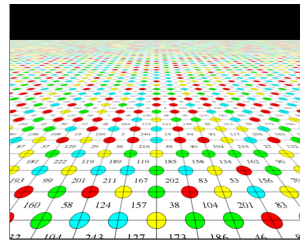


Figure 2: A tiling generated with the long-period hash function proposed in this paper.

the time needed to evaluate the hash function, and the joint size of the permutation tables will determine the memory footprint of the hash function. Period length can be traded for evaluation time and memory footprint.

4 Examples

In this section we propose long-period hash functions for Perlin’s noise function [7, 8] and for the procedural object distribution function of Lagae and Dutré [2, 3].

4.1 Perlin Noise

As mentioned in the previous section, the required range of the hash function is 16. For the sizes of the other permutation tables we select the primes 11, 13, 17 and 19. The period of the combined hash function is $11 \times 13 \times 16 \times 17 \times 19 = 739.024$. The size of the combined permutation table is $11 + 13 + 16 + 17 + 19 = 76$. Compared to Perlin’s implementation [8], the period is increased with a factor of almost 3000, the memory footprint is decreased with a factor of almost 3.5, and the evaluation time is increased with a factor of about 5. The total evaluation time of the modified noise function is increased with a factor of about 2.5. Figure 1 shows the original and the modified implementation. As expected, the two images have the same appearance. Figure 3 shows an example implementation of the combined hash function. Note that all the permutation tables are packed into a single array.

4.2 A Procedural Object Distribution Function

We select 24 for the range of the hash function. By applying an additional modulo divisor at runtime we are able to adjust the range of the hash function to 2, 3, 4, 6, and 8. For the sizes of the other permutation tables we select the primes 17, 19, 23, 29, 31

```

1 int p[76];
2
3 void randomize()
4 {
5     for (int i = 0; i < 11; ++i) {
6         p[i] = i;
7     }
8     for (int i = 0; i < (11 - 1); ++i) {
9         int j = i + (std::rand() % (11 - i));
10        std::swap(p[i], p[j]);
11    }
12
13    for (int i = 0; i < 13; ++i) {
14        p[11 + i] = i;
15    }
16    for (int i = 0; i < (13 - 1); ++i) {
17        int j = i + (std::rand() % (13 - i));
18        std::swap(p[11 + i], p[11 + j]);
19    }
20
21    for (int i = 0; i < 16; ++i) {
22        p[24 + i] = i;
23    }
24    for (int i = 0; i < (16 - 1); ++i) {
25        int j = i + (std::rand() % (16 - i));
26        std::swap(p[24 + i], p[24 + j]);
27    }
28
29    for (int i = 0; i < 17; ++i) {
30        p[40 + i] = i;
31    }
32    for (int i = 0; i < (17 - 1); ++i) {
33        int j = i + (std::rand() % (17 - i));
34        std::swap(p[40 + i], p[40 + j]);
35    }
36
37    for (int i = 0; i < 19; ++i) {
38        p[57 + i] = i;
39    }
40    for (int i = 0; i < (19 - 1); ++i) {
41        int j = i + (std::rand() % (19 - i));
42        std::swap(p[57 + i], p[57 + j]);
43    }
44 }
45
46 int hash(int x, int y)
47 {
48     return (p[ ((p[ x % 11] + y) % 11)]
49            + p[11 + ((p[11 + (x % 13)] + y) % 13)]
50            + p[24 + ((p[24 + (x % 16)] + y) % 16)]
51            + p[40 + ((p[40 + (x % 17)] + y) % 17)]
52            + p[59 + ((p[59 + (x % 19)] + y) % 19)]) % 16;
53 }

```

Figure 3: Implementation of the combined hash function for Perlin’s noise function.

and 37. The period of the combined hash function is $17 \times 19 \times 23 \times 24 \times 29 \times 31 \times 37 = 5.930.659.848$. Note that this is more than the 32-bit integer range. The size of the combined permutation table is only $17 + 19 + 23 + 24 + 29 + 31 + 37 = 180$. Figure 2 shows a tiling generated with this hash function. The supplemental video shows that also in very large tilings, no repetition is visible.

5 Conclusion

In this paper we have constructed long-period combined hash functions, using a technique originally introduced in the context of pseudo-random number generators. Because of their widespread use, procedural techniques require robust implementations. We believe that long-period hash functions help to

accomplish this.

Acknowledgments

The first author is funded as a Research Assistant by the Fund of Scientific Research - Flanders, Belgium (Aspirant F.W.O.- Vlaanderen). We would like to thank the anonymous reviewers and the people of our research group for their constructive remarks.

References

- [1] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., 2002.
- [2] A. Lagae and P. Dutré. A procedural object distribution function. *ACM Transactions on Graphics*, 24(4), 2005.
- [3] A. Lagae and P. Dutré. An alternative for Wang tiles: Colored edges versus colored corners. *ACM Transactions on Graphics*, 2006. Accepted.
- [4] P. L’Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–751, 1988.
- [5] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
- [6] D. R. Peachy. Solid texturing of complex surfaces. *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19(3):279–286, 1985.
- [7] K. Perlin. An image synthesizer. *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19(3):287–296, 1985.
- [8] K. Perlin. Improving noise. *ACM Transactions on Graphics*, pages 681–682, 2002.
- [9] S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, 1990.
- [10] B. A. Wichmann. and I. D. Hill. An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.
- [11] S. Worley. A cellular texture basis function. In *Proceedings of ACM SIGGRAPH 1996*, pages 291–294, 1996.