Amortized Noise

Ian Parberry

Technical Report LARC-2013-03

Laboratory for Recreational Computing
Department of Computer Science & Engineering
University of North Texas
Denton, Texas, USA

September, 2013

Amortized Noise

Ian Parberry
Department of Computer Science & Engineering
University of North Texas
Denton, TX, 76203–5017
URL: http://larc.unt.edu/ian

***Abstract*—A new noise generation algorithm called *amortized noise* generates smooth noise using dynamic programming techniques to amortize the cost of floating point multiplications over neighboring points. The amortized 2D noise algorithm uses a factor of $17/3 \approx 5.67$ fewer floating point multiplications than the 2D Perlin noise algorithm, resulting in a speedup by a factor of 3 in practice on current desktop computing hardware. The amortized 3D noise algorithm uses a factor of $40/7 \approx 5.71$ fewer floating point multiplications than the 3D Perlin noise algorithm, but the increasing overhead for the initialization of tables limits the speedup factor achieved in practice to around 2. In addition to being significantly faster, amortized noise is less prone to repeat at regular intervals than Perlin noise, and it is smoother since it uses quintic splines in place of cubic splines.**

***Index Terms*—Amortized analysis, dynamic programming, infinite noise, Perlin noise, Simplex noise.**

I. Introduction

Perlin noise [1] was developed as a source of smooth random noise for use in applications such as procedural texture generation and modeling (see, for example, Ebert et al. [2]). The code is heavily optimized since applications typically call it often. Although the results of each call to the Perlin noise function are computed independently of every other call, most applications use it to fill in a grid of noise values at regularly spaced points. We are able to take advantage of this typical call pattern to amortize the computation cost of Perlin noise using dynamic programming techniques[1]. We will refer to our new algorithm as *amortized noise*.

The remainder of this paper is divided into six main sections. Section II gives some definitions and notation to be used throughout the rest of the paper. Section III describes the amortized 2D noise generator in some detail and includes a sketch of a correctness proof. Section IV describes the amortized 3D noise generator in slightly less detail. Section V contains some observations about infinite amortized noise. For more information about amortized noise, including source code, see Parberry [4].

II. Notation

Let $\mathbb{R}$ denote the set of real numbers, and $\mathbb{N}$ denote the set of natural numbers starting at zero. Let $\mathbb{U}$ denote the real line segment from 0 to 1 and $\mathbb{U}^{\pm}$ denote the real line segment from -1 to 1, that is, $\mathbb{U} = \{x \in \mathbb{R} \mid 0 \le x \le 1\}$, $\mathbb{U}^{\pm} = \{i/n \mid i \in \mathbb{N}\} \cap \mathbb{U}$. Suppose $n \in \mathbb{N}$. Let $\mathbb{U}_n$ be the set of $n+1$ evenly-spaced points from $\mathbb{U}$, that is, $\mathbb{U}_n = \{i/n \mid i \in \mathbb{N}\} \cap \mathbb{U}$. This notation is summarized in Table I for the reader's convenience.

[1]Cormen *et al.* [3] attribute the term *amortized* to Danny Sleator and Robert Tarjan, and the term *dynamic programming* to R. Bellman in 1955.

Set	Definition
$\mathbb{N}$	Natural numbers $\cup\{0\}$
$\mathbb{R}$	Real numbers
$\mathbb{U}$	$\{x \in \mathbb{R} \mid 0 \le x \le 1\}$
$\mathbb{U}^{\pm}$	$\{x \in \mathbb{R} \mid -1 \le x \le 1\}$
$\mathbb{U}_n$	$\{i/n \mid i \in \mathbb{N}\} \cap \mathbb{U}$

TABLE I: The sets used in this paper and their definitions.

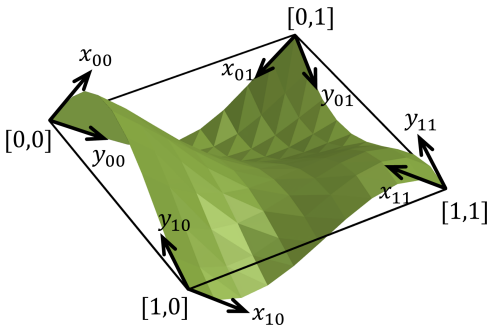

Fig. 1: 2D Perlin noise interpolates and smooths between pseudorandom gradients at integer grid points.

If S is a set, then for $n \ge 1$, S^n denotes the n-wise Cartesian product of S, $S^n = S \times S \times \cdots \times S$.

III. 2D Noise

The 2D Perlin noise algorithm computes a function $\mathscr{P}_2 : \mathbb{U} \times \mathbb{U} \to \mathbb{U}^{\pm}$. To compute $\mathscr{P}_2(x, y)$ it chooses pseudorandom gradients $\vec{g}_{00} = [x_{00}, y_{00}]$, $\vec{g}_{01} = [x_{01}, y_{01}]$, $\vec{g}_{10} = [x_{10}, y_{10}]$, and $\vec{g}_{11} = [x_{11}, y_{11}]$ at integer points $[0,0]$, $[0,1]$, $[1,0]$, and $[1,1]$ (respectively) and interpolates and smooths between them as shown in Fig. 1. Table II shows some pseudocode for one octave of the 2D Perlin noise generator on input $\vec{p} = [x, y] \in \mathbb{U} \times \mathbb{U}$. It uses a cubic spline function s_curve$(x) = x^2(3 - 2x)$ and a linear interpolation function lerp$(\epsilon, x, y) = x + \epsilon(y - x)$. While the Perlin noise algorithm computes $u_a(\vec{p})$ for any vector $\vec{p} \in \mathbb{U} \times \mathbb{U}$, in practice we only need to compute it for $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$ for some $n \in \mathbb{N}$. Call n the noise *granularity* and let $\delta = 1/n$, $\delta \in \mathbb{R}$.

The remainder of this section is divided into three subsections. Section III-A sketches the mathematical background

0.	Input $\vec{p} = [x, y]$
1.	$s_x = \text{s\_curve}(x)$
2.	$s_y = \text{s\_curve}(y)$
3.	$a = \text{lerp}(s_x, \vec{p} \cdot \vec{g}_{00}, \vec{p} \cdot \vec{g}_{01})$
4.	$b = \text{lerp}(s_x, \vec{p} \cdot \vec{g}_{10}, \vec{p} \cdot \vec{g}_{11})$
5.	Output $\text{lerp}(s_y, a, b)$

TABLE II: The 2D Perlin noise algorithm.

for amortized 2D noise and a proof of its correctness. Section III-B describes the implementation of amortized 2D noise. Section III-C contains both a theoretical and an experimental analysis of the run-time of amortized 2D noise compared to 2D Perlin noise.

A. Amortized 2D Noise

Amortized 2D noise follows the general structure of the Perlin noise algorithm shown in Table II, with the following optimizations. The s_curve function calls in Lines 1 and 2 are replaced with a table look-up, since only the $n + 1$ values of s_curve(x) for $x \in \mathbb{U}_n$ are needed. In Lines 3–4 the dot products are replaced by additions as follows.

Definition 1. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, define $u_a(\vec{p})$ as follows.*

1) $u_a(0, 0) = 0$.
2) *For all* $\gamma \in \mathbb{U}_n$, $u_a(0, \gamma) = \gamma x_{00}$.
3) *For all* $\gamma \in \mathbb{U}_n$, $u_a(\gamma, 0) = \gamma y_{00}$.
4) *For all* $\gamma \in \mathbb{U}_n$,
$$u_a(0, \gamma) = u_a(0, \gamma - \delta) + u_a(0, \gamma).$$
5) *For all* $\gamma \in \mathbb{U}_n$,
$$u_a(\gamma, 0) = u_a(\gamma - \delta, 0) + u_a(\gamma, 0).$$
6) *For all* $\gamma_0, \gamma_1 \in \mathbb{U}_n$,
$$u_a(\gamma_0, \gamma_1) = u_a(\gamma_0, 0) + u_a(0, \gamma_1).$$

Theorem 1. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, $u_a(\vec{p}) = \vec{p} \cdot \vec{g}_{00}$.*

Proof: We are required to prove that for all $i, j \in \mathbb{N}$ such that $0 \le i, j \le n$, $u_a(i\delta, j\delta) = (x_{00}i + y_{00}j)\delta$. When $i = j = 0$ the claim is true by Definition 1.1.

We will now prove the claim for $i = 0$ and all $j > 0$, that is, for all $j \in \mathbb{N}$ such that $1 \le j \le n$, $u_a(0, j\delta) = y_{00}j\delta$. The proof is by induction on j. When $j = 1$ the claim is true by Definition 1.3. Now suppose $j \ge 1$ and that the claim is true for j, that is, $u_a(0, j\delta) = y_{00}j\delta$. We are required to prove that the claim is true for $j+1$, that is, $u_a(0, (j+1)\delta) = y_{00}(j+1)\delta$. Now,

$$\begin{array}{lll} & u_a(0, (j+1)\delta) & \\ = & u_a(0, j\delta) + u_a(0, j\delta) & \text{(by Definition 1.4)} \\ = & u_a(0, j\delta) + y_{00}\delta & \text{(by Definition 1.3)} \\ = & y_{00}j\delta + y_{00}\delta & \text{(by induction)} \\ = & y_{00}(j+1)\delta & \text{(as required).} \end{array}$$

The proof of the claim for $i > 0$, $j = 0$ is similar to the above, substituting Definition 1.5 and Definition 1.2 for Definition 1.4) and Definition 1.3, respectively.

Now suppose $0 < i, j \le n$. Then,

$$\begin{array}{lll} & u_a(i\delta, j\delta) & \\ = & u_a(i\delta, 0) + u_a(0, j\delta) & \text{(by Definition 1.6)} \\ = & x_{00}i\delta + y_{00}j\delta & \text{(by the above)} \\ = & g_{00} \cdot [i, j]\delta & \text{(as required).} \end{array}$$

This completes the proof. ∎

Definition 2. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, define $v_a(\vec{p})$ as follows.*

1) $v_a(0, 1) = 0$.
2) *For all* $\gamma \in \mathbb{U}_n$, $v_a(0, 1 - \gamma) = \gamma x_{10}$.
3) *For all* $\gamma \in \mathbb{U}_n$, $v_a(\gamma, 1) = \gamma y_{01}$.
4) *For all* $\gamma \in \mathbb{U}_n$,
$$v_a(\gamma, 1) = v_a(\gamma - \delta, 1) + v_a(\gamma, 1).$$
5) *For all* $\gamma \in \mathbb{U}_n$,
$$v_a(0, \gamma) = v_a(0, \gamma + \delta) + v_a(0, 1 - \delta).$$
6) *For all* $\gamma_0, \gamma_1 \in \mathbb{U}_n$,
$$v_a(\gamma_0, \gamma_1) = v_a(\gamma_0, 1) + v_a(0, \gamma_1).$$

Theorem 2. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, $v_a(\vec{p}) = \vec{p} \cdot \vec{g}_{01}$.*

The proof is similar to that of Theorem 1.

Definition 3. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, define $u_b(\vec{p})$ as follows.*

1) $u_b(1, 0) = 0$.
2) *For all* $\gamma \in \mathbb{U}_n$, $u_b(1, \gamma) = \gamma x_{10}$.
3) *For all* $\gamma \in \mathbb{U}_n$, $u_b(1 - \gamma, 0) = \gamma y_{10}$.
4) *For all* $\gamma \in \mathbb{U}_n$,
$$u_b(\gamma, 0) = u_b(\gamma + \delta, 0) + u_b(1 - \gamma, 0).$$
5) *For all* $\gamma \in \mathbb{U}_n$,
$$u_b(1, \gamma j) = u_b(1, \gamma - \delta) + u_b(1, \gamma).$$
6) *For all* $\gamma_0, \gamma_1 \in \mathbb{U}_n$,
$$u_b(\gamma_0, \gamma_1) = u_b(\gamma_0, 1 - \delta) + u_b(1, \gamma_1).$$

Theorem 3. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, $u_b(\vec{p}) = \vec{p} \cdot \vec{g}_{10}$.*

The proof is similar to that of Theorem 1.

Definition 4. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, define $v_b(\vec{p})$ as follows.*

1) $v_b(1, 1) = 0$.
2) *For all* $\gamma \in \mathbb{U}_n$, $v_b(1, 1 - \gamma) = \gamma x_{11}$.
3) *For all* $\gamma \in \mathbb{U}_n$, $v_b(1 - \gamma, 1) = \gamma y_{11}$.
4) *For all* $\gamma \in \mathbb{U}_n$,
$$v_b(\gamma, 1) = v_b(\gamma + \delta, 0) + v_b(\gamma, 0).$$
5) *For all* $\gamma \in \mathbb{U}_n$,
$$v_b(1, \gamma) = v_b(1, \gamma + \delta) + v_b(1, 1 - \gamma).$$
6) *For all* $\gamma_0, \gamma_1 \in \mathbb{U}_n$,
$$v_b(\gamma_0, \gamma_1) = v_b(\gamma_0, 1 - \delta) + v_b(1, \gamma_1).$$

Theorem 4. *For all $\vec{p} \in \mathbb{U}_n \times \mathbb{U}_n$, $v_b(\vec{p}) = \vec{p} \cdot \vec{g}_{11}$.*

The proof is similar to that of Theorem 1.

Array	From	To
uax	$u_a(0,0)$	$u_a(0,1)$
uay	$u_a(0,0)$	$u_a(1,0)$
vax	$v_a(0,0)$	$v_a(0,1)$
vay	$v_a(0,1)$	$v_a(1,1)$
ubx	$u_b(1,0)$	$u_b(1,1)$
uby	$u_b(0,0)$	$u_b(1,0)$
vbx	$v_b(1,0)$	$v_b(1,1)$
vby	$v_b(0,1)$	$v_b(1,1)$

TABLE III: Interpolated 2D gradient tables and their contents using the values defined in Section III.

In Lines 3–4 of Table II the dot products are replaced by additions as described in Definitions 1–4 and Theorems 1–4, respectively. In each of Definitions 1–4, parts 1–5 are precomputed in two 1-dimensional tables of $d+1$ floating-point entries, and part 6 is computed on demand.

B. Implementation of Amortized 2D Noise

The following C code generates an n × n grid of 2D Perlin noise. We need some initialization done once per octave, starting with a spline table.

```
float spline[n+1];
for(int i=0; i<=n; i++)
  spline[i] = s_curve((float)i/n);
```

Notice that we can replace Perlin's original cubic spline function $s(t) = 3t^2 - 2t^3$ with a quintic spline function $6t^5 - 15t^4 + 10t^3$ as proposed in [5] without increasing the runtime of amortized noise. The quintic spline function has the advantage that its first and second derivative at 0 and 1 are zero.

We need 8 arrays to store the interpolated gradient tables, 2 for each edge of a square. Table III shows the arrays and their contents.

```
float uax[n+1], uay[n+1];
float vax[n+1], vay[n+1];
float ubx[n+1], uby[n+1];
float vbx[n+1], vby[n+1];
```

Four of these tables need to be filled in from bottom to top, and the others from top to bottom. This is done with the following two helper functions.

```
void FillUp(float* t, float f){
  t[0] = 0.0f; t[1] = f/n;
  for(int i=2; i<=n; i++)
    t[i] = t[i-1] + t[1];
} //FillUp
```

```
void FillDn(float* t, float f){
  t[n] = 0.0f; t[n-1] = -f/n;
  for(int i=n-2; i>=0; i--)
    t[i] = t[i+1] + t[n-1];
} //FillDn
```

These are used to initialize the interpolated gradient tables as follows.

```
FillUp(uax, x00); FillDn(vax, x01);
FillUp(ubx, x10); FillDn(vbx, x11);
FillUp(uay, y00); FillDn(uby, y10);
FillUp(vay, y01); FillDn(vby, y11);
```

Once initialization is complete, we can generate an n × n noise grid with the following function.

```
void anoise2(float*** cell){
  float u, v, a, b;
  for(int i=0; i<=n; i++)
    for(int j=0; j<=n; j++){
      u = uax[j] + uay[i];
      v = vax[j] + vay[i];
      a = lerp(spline[j], u, v);
      u = ubx[j] + uby[i];
      v = vbx[j] + vby[i];
      b = lerp(spline[j], u, v);
      cell[i][j] =
        lerp(spline[i], a, b);
    } //for
} //anoise2
```

Notice that the only floating point multiplications used are for the three linear interpolations.

C. Analysis of Amortized 2D Noise

The 2D Perlin noise algorithm uses 17 floating point multiplications per point per octave (see Table IV), and thus requires $17n^2$ floating point multiplications per octave to find noise values for an $n \times n$ grid. The techniques described in the previous section replace the cubic splines and dot products in Table IV with table lookups and floating point additions. We are left with a single floating point multiplication per point for each of three linear interpolations. The number of floating point multiplications required to generate noise on an $n \times n$ grid is therefore $3n^2 + \mathrm{O}(n)$ per octave, a reduction in the number of floating point multiplications by a factor of $17/3 \approx 5.67$ over 2D Perlin noise.

Since a good deal of the computation time used by Perlin noise is taken up by floating point multiplications, we can achieve a significant speedup factor in practice. Whether we can come close to the theoretical factor of 5.7 depends on the speed of floating point multiplications compared to other

Task	Lines	Number	Mults	Total
Cubic spline	1, 2	2	3	6
Linear interpolation	3–5	3	1	3
Dot product	3, 4	4	2	8
Total				17

TABLE IV: Number of floating point multiplications used by 2D Perlin noise per point per octave. The line numbers in the second column refer to the algorithm in Table II.

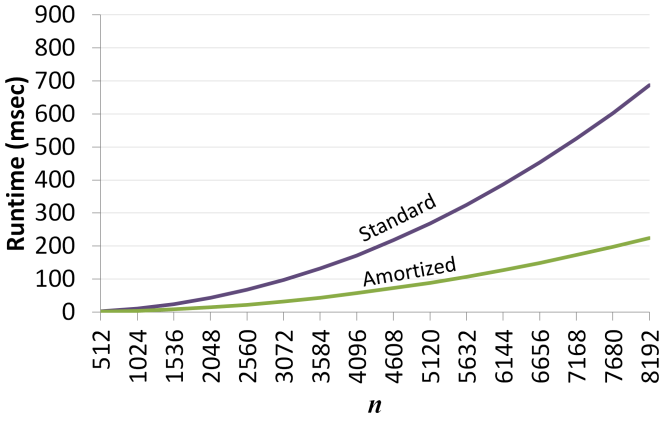

Fig. 2: Running time for the Perlin and amortized noise algorithms measured in seconds for computing 2D noise values for grids of $n \times n$ points, where $512 \leq n \leq 8192$ using single precision floating point arithmetic.

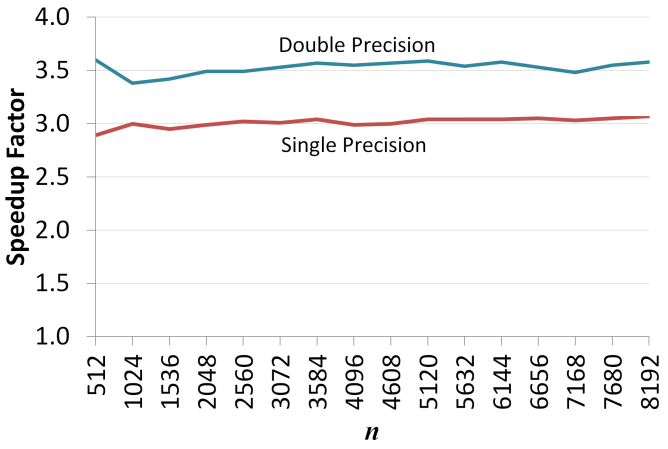

Fig. 3: Speedup factor for the amortized noise algorithm over the Perlin noise algorithm for a grid of $n \times n$ points, where $512 \leq n \leq 8192$.

operations. To test this, we measured the runtime of our algorithm using both single and double precision floating point arithmetic on an Intel® Core™ i7-3930K running at 4.2 GHz. We found that on this hardware amortized noise is approximately 3–3.5 times faster than Perlin noise.

Fig. 2 shows the run-time in milliseconds for computing 2D Perlin and amortized noise on an $n \times n$ grid for $512 \leq n \leq 8192$ in steps of 512 using single-precision floating point arithmetic. Fig. 3 shows the ratio of Perlin noise runtime divided by the amortized noise runtime for both single and double precision arithmetic.

The overhead of initializing the new arrays becomes an increasingly significant fraction of the overall runtime as n decreases. Repeating our experiments for small n, we found that amortized 2D noise is faster for all $n > 1$, but by a smaller factor than for large n. For example, Fig. 4 shows the ratio of the runtime of Perlin noise divided by that of amortized noise

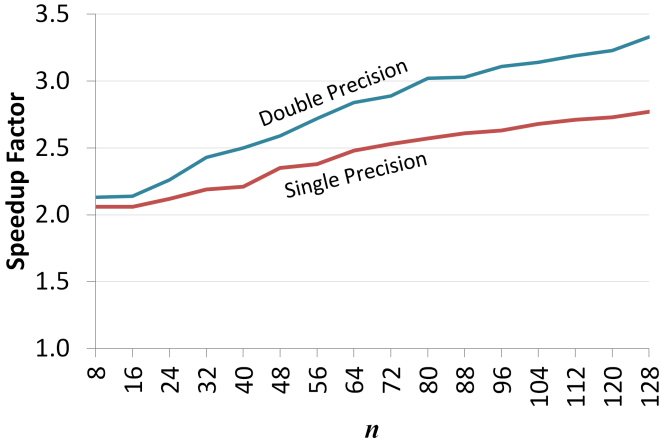

Fig. 4: Speedup factor for the amortized noise algorithm over Perlin noise for a grid of $n \times n$ points, where $8 \leq n \leq 128$.

on an $n \times n$ grid for $8 \leq n \leq 128$ in steps of 8.

We can see from Fig. 3 and Fig. 4 that amortized noise has more of an advantage over Perlin noise when both algorithms use double precision floating point arithmetic because double precision multiplications cost more time than single precision ones. We conjecture that as floating point multiplications become more expensive, amortized 2D noise will achieve a speedup factor approaching 5.

IV. 3D Noise

The 3D Perlin noise algorithm computes a function $\mathscr{P}_3 : \mathbb{U}^3 \to \mathbb{U}^{\pm}$. To compute $\mathscr{P}_3(x, y, z)$ it picks pseudorandom gradients $\vec{g}_{ijk} = [x_{ijk}, y_{ijk}, z_{ijk}]$ at the 8 integer points $[i, j, k]$ (respectively), for $i, j, k \in \{0, 1\}$, and interpolates and smooths between them. The amortized 2D noise algorithm from Section IV generalizes to 3D in a fairly straightforward manner (code is provided online in Parberry [4]). Instead of 8 interpolated gradient tables, there are 24.

The 3D Perlin noise algorithm uses 40 floating point multiplications per point per octave (see Table IV), and thus requires $40n^3$ floating point multiplications per octave to find noise values for an $n \times n \times n$ grid. The techniques described in the previous section replace the cubic splines and dot products in Table V with table lookups and floating point additions. We are left with a single floating point multiplication per point for each of 7 linear interpolations. The number of floating point multiplications required to generate noise on an $n \times n$ grid is therefore $7n^2 + \mathrm{O}(n)$ per octave, a reduction in the number of floating point multiplications by a factor of $40/7 \approx 5.71$ over 2D Perlin noise.

Fig. 5 shows the run-time in milliseconds for computing 3D Perlin and amortized noise on an $n \times n \times n$ grid for $32 \leq n \leq 512$ in steps of 32. Fig. 6 shows the ratio of the Perlin noise runtime divided by the amortized noise runtime. We can see that the overhead for initializing the large number of arrays is beginning to become a more significant fraction of the run-time. The results were much the same for both single and

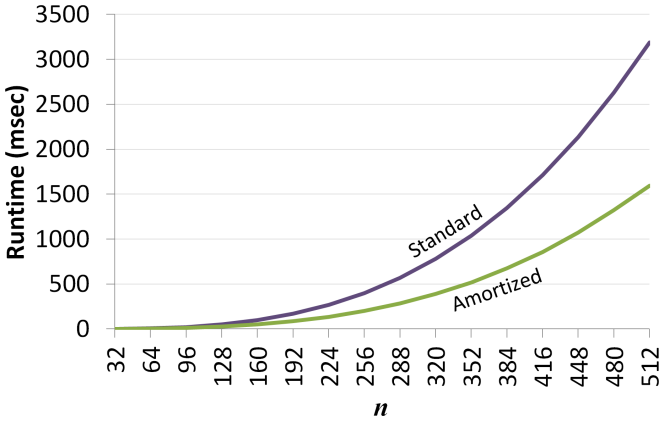

Fig. 5: Running time for the 3D Perlin noise and amortized 3D noise algorithms measured in seconds for computing 3D noise values for grids of $n \times n \times n$ points, where $32 \leq n \leq 512$.

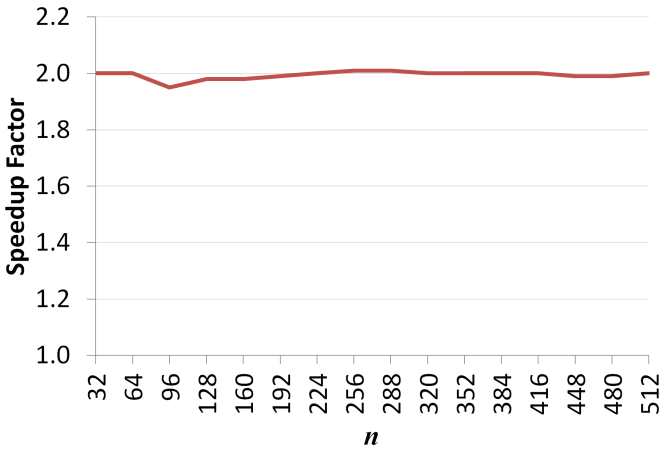

Fig. 6: Speedup factor for the amortized 3D noise algorithm over Perlin noise for a grid of $n \times n \times n$ points, where $32 \leq n \leq 512$.

double precision floating point operations. One can conjecture from this result that amortized noise will be of little or no use for higher dimensional noise.

V. Infinite Noise

Perlin noise repeats with period nB, where B is the size of Perlin's permutation and gradient tables, and is usually equal to 256. This defect in Perlin noise could be remedied on an m-bit computer by computing the gradients at integer grid points using a minimal perfect hash function for m-bit integers, that is, a hash function whose domain and range are exactly the set of m-bit integers. (For more information about hash functions, see, for example, Knuth [6].) However, this would slow down the Perlin noise generator significantly.

Task	Number	Mults	Total
Cubic spline	3	3	9
Linear interpolation	7	1	7
Dot product	8	3	24
Total			40

TABLE V: Number of floating point multiplications used by 3D Perlin noise per point per octave.

Amortized noise takes less of a speed hit since the cost of computing gradients at integer grid points is amortized over the computation at the remaining points. Take, for example, amortized 2D noise as described in Section III. Let $h : \mathbb{N} \to \mathbb{N}$ be a hash function. For all $n \in \mathbb{N}$, define $h : \mathbb{N}^2 \to \{0, 1, \ldots, n-1\}$ to be $h(x, y) = h(h(x) + y) \mod n$. Compute gradients $g(\vec{p}) \in \mathbb{N} \times \mathbb{N}$ at integer points $\vec{p} \in \mathbb{N} \times \mathbb{N}$ as follows. Choose an *angular granularity* $n \in \mathbb{N}$ and let $\{\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{n-1}\}$ be the set of n uniformly spaced vectors around the unit circle. We then define $\vec{g}([x, y]) = \hat{u}_{h(x,y)}$. The resulting noise will be, if not infinite, then as close as possible to being infinite depending on the quality of the hash function. We have had particular success with a quadratic hash function $h(x) = px^2$ for small prime numbers p.

VI. Conclusion

While the Perlin noise algorithm provides random access to a source of smooth noise, amortized noise saves computation by computing noise values in a nearby neighborhood. Amortized noise is only useful as a sequential computation running on a traditional von Neumann architecture CPU. The running time of shader implementations of Perlin noise (such as Green [7]) is dominated by the cost of texture addressing rather than floating point multiplication, and hence will see little or no benefit from amortization.

Interesting open problems include a full analysis of candidate hash functions for infinite amortized noise. The quadratic hash function suggested in this paper does better than linear congruential hash functions, but it does produce some visual artifacts. A quadratic congruential hash function $h(x) = x^2 \mod p$ for prime numbers p around 2^{32} for 32-bit arithmetic has fewer visual artifacts , but costs slightly more to compute.

References

[1] K. Perlin, "An image synthesizer," in *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 1985, pp. 287–296.

[2] D. Ebert, S. Worley, F. Musgrave, D. Peachey, and K. Perlin, *Texturing & Modeling, a Procedural Approach*, 3rd ed. Elsevier, 2003.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and S. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[4] I. Parberry. (2013) Amortized Noise. [Online]. Available: http://larc.unt.edu/ian/research/amortizednoise/

[5] K. Perlin, "Improving noise," in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. ACM, 2002, pp. 681–682.

[6] D. E. Knuth, *Sorting and Searching*, 2nd ed., ser. The Art of Computer Programming. Addison-Wesley, 1998, vol. 3.

[7] S. Green, "Implementing improved Perlin noise," in *GPU Gems 2*. CRC Press, 2005.