Amortized Noise

Ian Parberry
University of North Texas

Figure 1. A texture generated from 2D noise.

Abstract

Perlin noise is often used to compute a regularly spaced grid of noise values. The *amortized noise* algorithm takes advantage of this regular call pattern to amortize the computation cost of floating-point computations over interpolated points using dynamic programming techniques. The 2D amortized 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 approximately 3.6–4.8 in practice on available desktop and laptop computing hardware. The 3D amortized noise algorithm uses a factor of $40/7 \approx 5.71$ fewer floating-point multiplications than the 3D Perlin noise algorithm; however, the increasing overhead for the initialization of tables limits the speedup factor achieved in practice to around 2.25. Improvements to both 2D Perlin noise and 2D amortized noise include making them infinite and non-repeating by replacing the permutation table with a perfect hash function, and making them smoother by using quintic splines instead of cubic splines. While these improvements slow down 2D Perlin noise down by a factor of approximately 32–92, they slow 2D amortized noise by a negligible amount.

1. Introduction

Perlin noise [Perlin 1985] was developed as a source of smooth random noise for use in applications such as procedural texture generation and modeling (see, for example, the book by Ebert et al. [2003]). Figure 1 shows an example of a texture rendered from 2D noise by converting the noise value calculated at each pixel into a grayscale value. Perlin's 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 ev-

 ISSN 2331-7418

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 1. The sets used in this paper and their definitions.

ery other call, many applications (such as textures and height maps) use it to compute a grid of noise values at regularly spaced points. The *amortized noise* algorithm takes advantage of this regular call pattern to amortize the computation cost using dynamic programming techniques. These are the kinds of optimizations that developers make in practice, and although they may have been applied to noise generation in the past, the results do not appear to have been published in the open literature. This is the first publication to systematically describe and evaluate the performance gains to be obtained from amortizing the Perlin noise algorithm.

The remainder of this paper is divided into six main sections. Section 2 gives some definitions and notation to be used throughout the rest of the paper. Section 3 describes the 2D amortized noise generator in some detail. Section 4 describes the 3D amortized noise generator in slightly less detail. Section 5 shows that some improvements to the noise quality that slow down 2D Perlin noise by a factor of 63 have negligible effect on the running time of 2D amortized noise.

2. 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 inclusive and $\mathbb{U}^{\pm}$ denote the real line segment from -1 to 1 inclusive. That is, $\mathbb{U} = \{x \in \mathbb{R} \mid 0 \le x \le 1\}$, $\mathbb{U}^{\pm} = \{x \in \mathbb{R} \mid -1 \le x \le 1\}$. 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 1 for the reader's convenience. If S is a set, then for $n \ge 1$, S^n denotes the n-wise Cartesian product of S,

$$S^n = \underbrace{S \times S \times \cdots \times S}_{n \text{ times}}.$$

3. 2D Noise

The 2D Perlin noise algorithm computes a function $\mathscr{P}_2 : \mathbb{U} \times \mathbb{U} \to \mathbb{U}^{\pm}$. To compute $\mathscr{P}_2(x,y)$ the algorithm 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

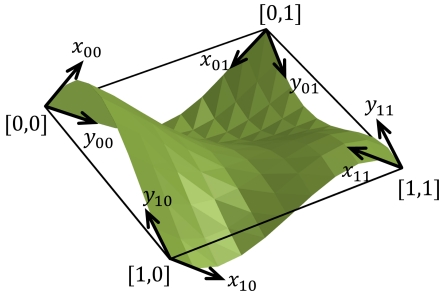

Figure 2. 2D Perlin noise interpolates and smooths between pseudorandom gradients at integer grid points.

$[1,1]$, respectively, and interpolates and smooths between them as shown in Figure 2. Table 2 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 $\text{s\_curve}(x) = x^2(3-2x)$ and a linear interpolation function $\text{lerp}(\varepsilon,x,y) = x + \varepsilon(y-x)$. While the 2D 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 $n \in \mathbb{N}$. For example, Figure 3 shows a square grid with $n = 5$, with the integer gridpoints shown as black dots and the non-integer gridpoints as white dots. 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 3.1 sketches the 2D amortized noise algorithm by investigating how 2D Perlin noise works on an example. Section 3.2 describes the implementation of 2D amortized noise. Section 3.3 contains both a theoretical and an experimental analysis of the run-time of 2D amortized noise compared to 2D Perlin noise.

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

Table 2. The 2D Perlin noise algorithm.

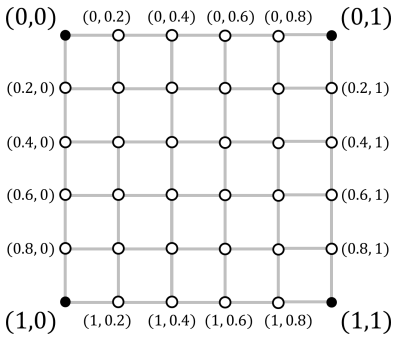

Figure 3. A grid of regularly spaced points. The corner points (black dots) have integer coordinates, while the interior points (white dots) have rational coordinates. The corners of this grid correspond to the corners of the square in Figure 2.

3.1. 2D Amortized Noise

Suppose, for example, that the gradient vectors at the corners of the grid $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$ shown in Figure 3 are, respectively,

$$\begin{aligned}\vec{g}_{00} &= [-0.53,-0.848]\\ \vec{g}_{10} &= [-0.4472,0.8944]\\ \vec{g}_{01} &= [0.9285,0.3714]\\ \vec{g}_{11} &= [-0.9578,0.2873].\end{aligned}$$

Note that all four vectors have unit length. Let $\mathbb{P} = \{0, 0.2, 0.4, 0.6, 0.8, 1\}$.

Line 3 of the 2D Perlin noise algorithm in Table 2 computes the vector dot product $\vec{g}_{00} \cdot \vec{p}$ for all $\vec{p} \in \mathbb{P} \times \mathbb{P}$. If we store this in a table $u_a(x,y) = \vec{g}_{00} \cdot [x,y]$ for all $x,y \in \mathbb{P}$ we get the results shown in Table 3. For example, the entry for $[0.8, 0.6]$ is

$$\begin{aligned}u_a(0.8,0.6) &= \vec{g}_{00} \cdot [0.8,0.6]\\ &= [-0.53,-0.848] \cdot [0.8,0.6]\\ &= -0.9328.\end{aligned}$$

	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0	-0.1060	-0.2120	-0.3180	-0.4240	-0.5300
0.2	-0.1696	-0.2756	-0.3816	-0.4876	-0.5936	-0.6996
0.4	-0.3392	-0.4452	-0.5512	-0.6572	-0.7632	-0.8692
0.6	-0.5088	-0.6148	-0.7208	-0.8268	-0.9328	-1.0388
0.8	-0.6784	-0.7844	-0.8904	-0.9964	-1.1024	-1.2084
1.0	-0.8480	-0.9540	-1.0600	-1.1660	-1.2720	-1.3780

Table 3. A table of $u_a(y,x) = \vec{g}_{00} \cdot [x,y]$ for all $x,y \in \mathbb{P}$. Notice that in the first row and the first column, the cyan entries are integer multiples of the red entry.

Although the entries of Table 3 were computed independently of each other during different calls to the 2D Perlin noise algorithm, some clear patterns emerge. Obviously the top-left entry is zero since $u_a(0,0) = \vec{g}_{00} \cdot \vec{0} = 0$. Looking at the top row of Table 3, we see that

$$u_a(0.0, 0.2) = \vec{g}_{00} \cdot [0.2, 0] = -0.53 \times 0.2 = -0.106,$$

(highlighted in cyan) and $u_a(0.0, 0.2i) = -0.106i$ for all $1 \leq i \leq 5$ (highlighted as a group in cyan). Instead of using five floating-point multiplications to compute these values we can use five additions by observing that $u_a(0.0, 0.2)$ equals 0.2 times the x-coordinate of $\vec{g}_{00}$ and for all $2 \leq i \leq 5$,

$$\begin{aligned} u_a(0.0, 0.2i) &= u_a(0.0, 0.2(i-1)) - 0.106 \\ &= u_a(0.0, 0.2(i-1)) + u_a(0.0, 0.2). \end{aligned}$$

Similarly, looking at the left column of Table 3, we see that

$$u_a(0.2, 0.0) = \vec{g}_{00} \cdot [0, 0.2] = -0.848 \times 0.2 = -0.1696,$$

(highlighted in cyan) and $u_a(0.2j, 0.0) = -0.1696j$ for all $1 \leq j \leq 5$ (highlighted as a group in cyan). Again, instead of using five floating-point multiplications to compute these values we can use five additions by observing that $u_a(0.2, 0.0)$ equals 0.2 times the y-coordinate of $\vec{g}_{00}$ and for all $2 \leq j \leq 5$,

$$\begin{aligned} u_a(0.2j, 0.0) &= u_a(0.2(j-1), 0.0) - 0.1696 \\ &= u_a(0.2(j-1), 0.0) + u_a(0.2, 0.0). \end{aligned}$$

Furthermore, as shown in Table 4, each of the remaining entries (which were also computed using a single floating-point multiplication each) equals the sum of the entry in the same row, column 0, and the same column, row 0. For example, $u_a(0.6, 0.8)$ (highlighted in blue) equals the sum of $u_a(0.0, 0.8)$ and $u_a(0.6, 0.0)$ (both highlighted in green).

	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0	-0.1060	-0.2120	-0.3180	-0.4240	-0.5300
0.2	-0.1696	-0.2756	-0.3816	-0.4876	-0.5936	-0.6996
0.4	-0.3392	-0.4452	-0.5512	-0.6572	-0.7632	-0.8692
0.6	-0.5088	-0.6148	-0.7208	-0.8268	-0.9328	-1.0388
0.8	-0.6784	-0.7844	-0.8904	-0.9964	-1.1024	-1.2084
1.0	-0.8480	-0.9540	-1.0600	-1.1660	-1.2720	-1.3780

Table 4. A table of $u_a(y, x) = \vec{g}_{00} \cdot [x, y]$ for all $x, y \in \mathbb{P}$. Notice that the blue entry is equal to the sum of the green entries.

	0.0	0.2	0.4	0.6	0.8	1.0
0.0	-0.4472	-0.3578	-0.2683	-0.1789	-0.0894	0
0.2	-0.2683	-0.1789	-0.0894	0.0000	0.0894	0.1789
0.4	-0.0894	0.0000	0.0894	0.1789	0.2683	0.3578
0.6	0.0894	0.1789	0.2683	0.3578	0.4472	0.5367
0.8	0.2683	0.3578	0.4472	0.5367	0.6261	0.7155
1.0	0.4472	0.5367	0.6261	0.7155	0.8050	0.8944

Table 5. A table of $v_a(y,x) = \vec{g}_{10} \cdot [x,y]$ for all $x,y \in \mathbb{P}$.

x	0.0	0.2	0.4	0.6	0.8	1.0
s_curve(x)	0.0000	0.1040	0.3520	0.6480	0.8960	1.0000

Table 6. Values of the cubic spline function $\texttt{s\_curve}(x) = 3x^2 - 2x^3$ at points $x \in \mathbb{P}$.

Similar observations can be made about v_a in Table 5, using the last column instead of the first.

The value a is computed from u_a and v_a in Line 5 of the 2D Perlin noise algorithm in Table 2 using a linear interpolation and a cubic spline. Since we only need cubic splines for values in $\mathbb{P}$, we can pre-compute these and store them in a table as shown, for example, in Table 6. From Line 5 of Table 2,

$$a(y,x) = (1 - \texttt{s\_curve}(x))u_a(y,x) + \texttt{s\_curve}(x)v_a(y,x).$$

Table 7 shows the values of $a(y,x)$ for our example.

The process for computing u_b, v_b, and b in Lines 6–8 of Table 2 is similar, differing mostly in which rows and columns are greyed out. The final result computed in Line 9 of Table 2 for input (x,y) is

$$(1 - \texttt{s\_curve}(y))a(y,x) + \texttt{s\_curve}(y)b(y,x).$$

There is no need to explicitly store the tables for a and b since each entry is used only once. Furthermore, we only need to store one row and one column of u_a, v_a, u_b, and v_b since the remaining entries are used only once each. We are left with eight one-dimensional arrays of six entries each (five if the zero is not stored explicitly).

	0.0	0.2	0.4	0.6	0.8	1.0
0.0	0	-0.1322	-0.2318	-0.2279	-0.1242	0
0.2	-0.1696	-0.2655	-0.2788	-0.1716	0.0184	0.1789
0.4	-0.3392	-0.3989	-0.3257	-0.1154	0.1610	0.3578
0.6	-0.5088	-0.5323	-0.3726	-0.0592	0.3037	0.5367
0.8	-0.6784	-0.6656	-0.4196	-0.0030	0.4463	0.7155
1.0	-0.8480	-0.7990	-0.4665	0.0532	0.5890	0.8944

Table 7. Computing a.

3.2. Implementation of 2D Amortized Noise

This section describes a C implementation of the amortized noise algorithm. The code described here is simplified slightly from the ideal implementation in order to fit within the confines of the page width and still be human-readable. A full implementation in C++ is provided in the supplementary material. We begin with some defines lifted directly from Perlin's original code[1]:

```
#define B 0x100
#define BM 0xff
#define N 0x1000

#define lerp(t, a, b) (a + t*(b - a))
```

We also use Perlin's gradient and permutation tables.

```
float g2[B][2]; //Perlin gradient table.
int p[B]; //Perlin permutation table.
```

These tables are initialized in essentially the same way that they are in Perlin's code.

```
void initPerlinNoiseTables(){
  //random normalized gradient vectors
  for(int i=0; i<B; i++){
    g2[i][0] = (float)((rand()%(B + B)) - B)/B;
    g2[i][1] = (float)((rand()%(B + B)) - B)/B;
    float m = sqrt(g2[i][0]*g2[i][0] + g2[i][1]*g2[i][1]);
    g2[i][0] /= m; g2[i][1] /= m;
  } //for

  //identity permutation
  for(int i=0; i<B; i++)
    p[i] = i;

  //random permutation
  for(int i=B-1; i>0; i--){
    int tmp = p[i];
    int j = rand()%(i+1); //minor correction to Perlin's code
    p[i] = p[j]; p[j] = tmp;
  } //for
} //initPerlinNoiseTables
```

The preceding initialization is done once at the start of the program. Next we need some initialization done once per octave, starting with the spline table.

[1] http://mrl.nyu.edu/~perlin/doc/oscar.html#noise.

```
float spline[n+1];

void initSplineTable(const int n){
  for(int i=0; i<n; i++){
    float t = (float)i/n;
    spline[i] = (t * t * (3.0f - 2.0f*t));
  } //for
} //initSplineTable
```

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

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 8. Interpolated 2D gradient tables and their contents using the values defined in Section 3.1.

```
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, int n){
  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, int n){
  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.

```
void initEdgeTables(int x0, int y0, int n){
  //compute gradients at corner points
  unsigned int b0 = h(x0,   y0);
  unsigned int b1 = h(x0,   y0+1);
  unsigned int b2 = h(x0+1, y0);
  unsigned int b3 = h(x0+1, y0+1);

  //fill inferred gradient tables from corner gradients
  FillUp(uax, g2[b0][0], n); FillDn(vax, g2[b1][0], n);
  FillUp(ubx, g2[b2][0], n); FillDn(vbx, g2[b3][0], n);
  FillUp(uay, g2[b0][1], n); FillUp(vay, g2[b1][1], n);
  FillDn(uby, g2[b2][1], n); FillDn(vby, g2[b3][1], n);
} //initEdgeTables
```

The function `h` in the above code is Perlin's hash function:

```
unsigned int h(unsigned int x, unsigned int y){
  return p[(p[x & BM] + y) & BM];
} //h
```

Once initialization is complete, a single point of noise can be generated using the following function:

```
float getNoise(int i, int j){
  float u, v, a, b;
  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);
  return lerp(spline[i], a, b);
} //getNoise
```

An `n` $\times$ `n` noise grid can be generated by iterating this process.

```
void getNoise(int n, int i0, int j0, float** cell){
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      cell[i0 + i][j0 + j] = getNoise(i, j);
} //getNoise
```

Notice that the only floating-point multiplications used are for the three linear interpolations per point in `getNoise(int, int)`.

Finally, the following function generates octaves `m0` through `m1` into `cell`, whose top-left point is assumed to be at integer coordinates `(x, y)`.

```
float generate(int x, int y, int m0, int m1, int n, float** cell){
  int r = 1; //side of cell divided by side of subcell.

  //Skip over unwanted octaves.
  for(int i=1; i<m0; i++){
    n /= 2; r += r;
  } //for

  //generate first octave directly into cell
  initSplineTable(n);
  for(int i0=0; i0<r; i0++)
    for(int j0=0; j0<r; j0++){
      initEdgeTables(x + i0, y + j0, n);
      getNoise(n, i0*n, j0*n, cell);
    } //for

  float scale = 1.0f; //scale factor

  //add the other octaves into cell
  for(int k=m0; k<m1 && n>=2; k++){
    n /= 2; r += r;  x += x; y += y; scale *= 0.5f; //rescale
    initSplineTable(n);
    for(int i0=0; i0<r; i0++)
      for(int j0=0; j0<r; j0++){
        initEdgeTables(x + i0, y + j0, n);
        addNoise(n, i0*n, j0*n, scale, cell);
      } //for
  } //for each octave

  //Compute 1/magnitude and return it.
  //Multiply noise by this to bring it to [-1,1].
  return M_SQRT2/(2.0f - scale);
} //generate
```

The return value `M_SQRT2/(2.0f - scale)` may at first seem obscure. Note that `scale` is always a power of 2. A single octave of Perlin noise returns a value of magnitude at most $1/\sqrt{2}$. Adding magnitudes over all scaled octaves gives a total magnitude of

$$\frac{1}{\sqrt{2}}(1+\tfrac{1}{2}+\tfrac{1}{4}+\ldots+\texttt{scale}) = \frac{2-\texttt{scale}}{\sqrt{2}}$$

(using the standard formula for the sum of a geometric progression). The inverse magnitude is therefore

$$\frac{\sqrt{2}}{2-\texttt{scale}}.$$

In order to avoid the unnecessary expense of zeroing out memory, the first octave is generated directly into the cell using `getNoise` described above, while subsequent octaves are scaled and added into the cell using `addNoise`:

```
void addNoise(int n, int i0, int j0, float scale, float** cell){
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      cell[i0 + i][j0 + j] += scale * getNoise(i, j);
} //addNoise
```

3.3. Analysis of 2D Amortized Noise

The 2D Perlin noise algorithm uses 17 floating-point multiplications per point per octave (see Table 9), 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 9 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 2D 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 operations. To test this, we measured the run-time of our algorithm on various desktop and laptop computers and found that in practice 2D amortized noise is approximately 3.6–4.8 times faster than 2D Perlin noise (see Figure 4).

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 9. 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 2.

4. 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 eight integer points $[i,j,k]$, respectively, for $i,j,k \in \{0,1\}$, and interpolates and smooths between them. The 2D amortized noise algorithm from Section 4 generalizes to 3D in a fairly straightforward manner. Instead of eight interpolated gradient tables, there are 24.

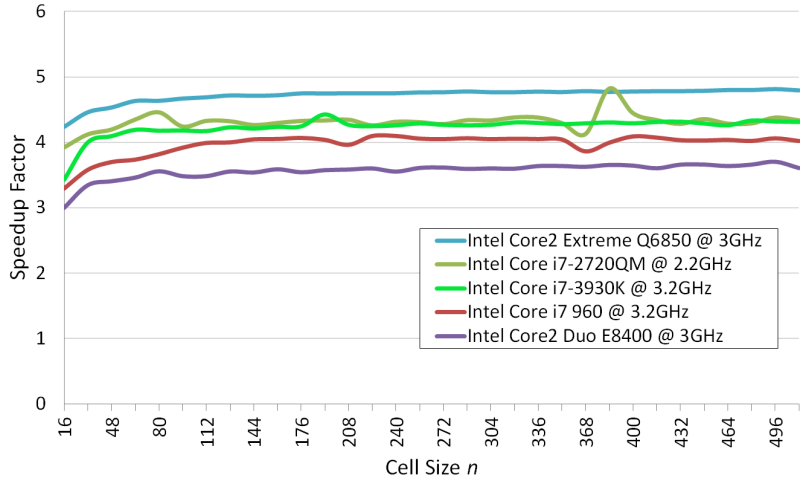

Figure 4. Speedup factor obtained by amortized noise over Perlin noise for computing 2D noise values for grids of $n \times n$ points, where $16 \leq n \leq 512$.

The 3D Perlin noise algorithm uses 40 floating-point multiplications per point per octave (see Table 9), 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 10 with table lookups and floating-point additions. We are left with a single floating-point multiplication per point for each of seven 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 3D Perlin noise.

We found that, on the hardware described above, 3D amortized noise is approximately 2.25 times faster than 3D Perlin noise. Figure 5 shows the ratio of the Perlin noise CPU time divided by the amortized noise CPU time on an Intel Core i7-3930K @ 3.2GHz. We can see that the overhead for initializing the large number of arrays is beginning to become a more significant fraction of the CPU time. One can conjecture from this result that amortized noise will likely be of little or no use for higher-dimensional noise.

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

Table 10. Number of floating-point multiplications used by 3D Perlin noise per point per octave.

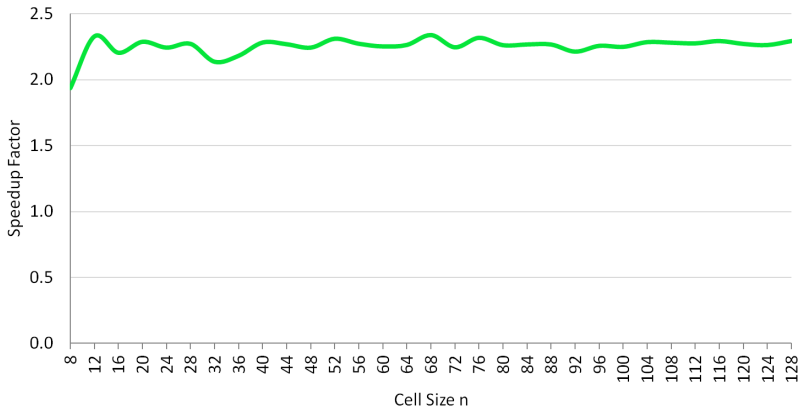

Figure 5. Speedup factor obtained by amortized noise over Perlin noise for computing 3D noise values for grids of $n \times n \times n$ points, where $8 \leq n \leq 128$.

5. Infinite Smooth 2D Noise

There are improvements to the Perlin noise algorithm that increase the variability and smoothness of the noise, but unfortunately increase its running time by a large amount. These improvements have negligible effect on the running time of the amortized noise algorithm. The first improvement is in *variability*. Perlin noise repeats with period nB, where B is the size of Perlin's permutation and gradient tables (which was originally 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 perfect hash function for m-bit integers; that is, a hash function whose domain and range are the set of m-bit integers. (For more information about hash functions, see, for example, the standard textbook by Knuth [1998].)

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 MurmurHash3[2].

For example, the hashed gradient at a corner point can be computed from its integer coordinates as follows:

```
float h(unsigned int x, unsigned int y){
  unsigned int result;
  unsigned long long key = ((unsigned long long)x<<32) | y;
  MurmurHash3_32(&key, 8, seed, &result);
  return (float)result;
} //h
```

[2] Open source, https://code.google.com/p/smhasher/wiki/MurmurHash3.

The edge tables are then computed as follows:

```
void initEdgeTables(int x, int y, int n){
  //compute gradients at corner points
  float b0=h(x, y), b1=h(x, y+1), b2=h(x+1, y), b3=h(x+1, y+1);

  //fill inferred gradient tables from corner gradients
  FillUp(uax, cosf(b0), n); FillDn(vax, cosf(b1), n);
  FillUp(ubx, cosf(b2), n); FillDn(vbx, cosf(b3), n);
  FillUp(uay, sinf(b0), n); FillUp(vay, sinf(b1), n);
  FillDn(uby, sinf(b2), n); FillDn(vby, sinf(b3), n);
} //initEdgeTables
```

The second improvement is in *smoothness*. Perlin [2002] has proposed replacing his original cubic spline function $3t^2 - 2t^3$ with a quintic spline function $6t^5 - 15t^4 + 10t^3$. The quintic spline function has the advantage that its first and second derivatives at 0 and 1 are both zero, leading to smoother noise with no discontinuities at integer points.

These two improvements, when implemented in the obvious manner, slow the 2D Perlin noise generator drastically by a factor of around 8.9–19.5, as shown in Figure 6. As Figure 7 shows, 2D improved amortized noise is only negligibly slower than 2D amortized noise since the cost of computing gradients at integer grid points is amortized over the computation at the remaining points, and is therefore about 32–92 times faster than 2D improved Perlin noise.

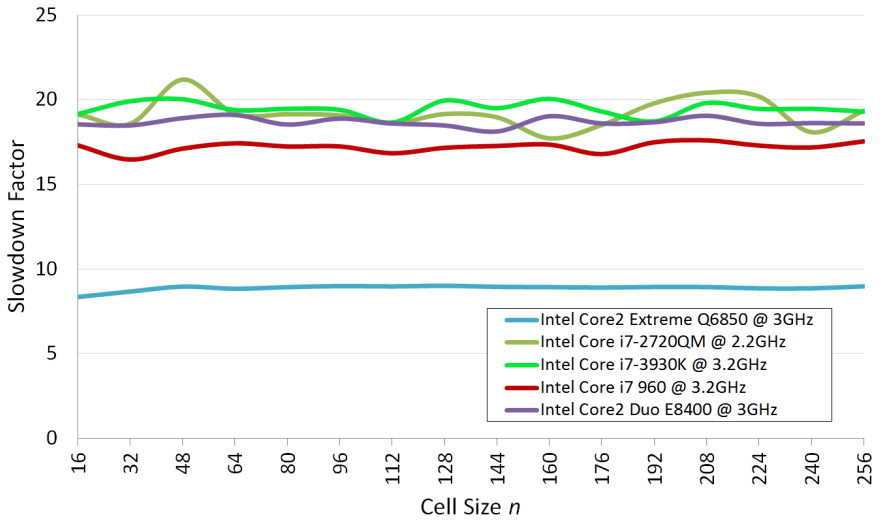

Figure 6. Slowdown factor for the 2D improved Perlin noise algorithm over Perlin noise for a grid of $n \times n$ points, where $16 \leq n \leq 256$.

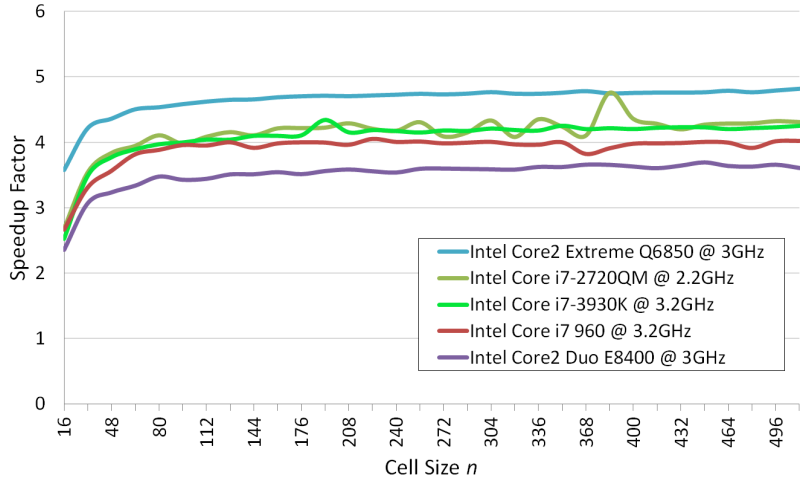

Figure 7. Speedup factor for the 2D improved amortized noise algorithm over Perlin noise for a grid of $n \times n$ points, where $16 \leq n \leq 512$.

6. Conclusion

We have seen that, on available desktop and laptop computing hardware, the 2D amortized noise algorithm is approximately 3.6–4.8 times faster than the 2D Perlin noise algorithm, and that the 3D amortized noise algorithm is approximately 2.25 times faster than the 3D Perlin noise algorithm. We have also seen that improvements to the noise quality that cause the 2D Perlin noise algorithm to slow down by a factor of 32–92 cause almost no slowdown of the 2D amortized noise algorithm.

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, and is therefore only useful when noise values are computed in an evenly-spaced grid such as a texture or height map. The amortized noise algorithm is likely only useful as a sequential computation running on a traditional von Neumann architecture CPU. The running time of a shader implementation of Perlin noise (such as the one by Green in *GPU Gems 2* [2005]) is typically dominated by factors such as the the speed of texture addressing rather than the speed of floating-point multiplication, and hence will see little or no benefit from amortization.

Interesting open problems include a full investigation of candidate hash functions for 2D improved amortized noise.

References

EBERT, D., WORLEY, S., MUSGRAVE, F., PEACHEY, D., AND PERLIN, K. 2003. *Texturing & Modeling, a Procedural Approach*, 3rd ed. Morgan Kaufmann, San Francisco. 31

GREEN, S. 2005. Implementing improved Perlin noise. In *GPU Gems 2*, CRC Press, Natick, MA. 45

KNUTH, D. E. 1998. *Sorting and Searching*, second ed., vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA. 43

PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '85, 287–296. http://doi.acm.org/10.1145/325334.325247. 31

PERLIN, K. 2002. Improving noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '02, 681–682. http://doi.acm.org/10.1145/566570.566636. 44

Index of Supplemental Materials

Source code and data is available under the GNU All-Permissive License in a zipfile on the JCGT website (as of the publication date) for this paper, and maintained after publication at https://github.com/Ian-Parberry/AmortizedNoise. The contents of the supplemental data are as follows:

`2D Evaluator` has a Microsoft Visual Studio 2012 project, an iOS Xcode project, and a Unix makefile for the evaluator used to measure the running time of amortized noise compared to Perlin noise.

`2D Generator` has a Microsoft Visual Studio 2012 project, an iOS Xcode project, and a Unix makefile for a generator that will save a grayscale image of 2D finite or infinite amortized noise.

`3D Generator` has a Microsoft Visual Studio 2012 project, an iOS Xcode project, and a Unix makefile for a generator that will save grayscale images of 3D finite or infinite amortized noise.

`Data` has files `Data2D.xlsx` and `Data3D.xlsx` containing the test data in Microsoft Excel format.

`Examples` has four image files as follows:

File Name	Size	Type	Dims.	Finity
`i0s512o36s9999r23c14.png`	512×512	png image	2D	finite
`i1s512o36s9999r23c42.png`	512×512	png image	2D	infinite
`finite.gif`	256×256	animated gif	3D	finite
`infinite.gif`	256×256	animated gif	3D	infinite

The code quoted in this paper differs slightly from the code in the GitHub archive in that the former is simplified to fit within the tight width restrictions of this medium, while the latter is engineered for comprehension and use by programmers in the real world. Links to Doxygen-generated documentation of the source code can be found at http://larc.unt.edu/ian/research/amortizednoise/.

Author Contact Information

Ian Parberry
Dept. of Computer Science and Engineering
University of North Texas
1155 Union Circle #311366
Denton, Texas 76203–5017

http://larc.unt.edu/ian

Ian Parberry, Amortized Noise, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 2, 31–47, 2014
http://jcgt.org/published/0003/02/0/2

Received:	2014-02-02		
Recommended:	2014-03-22	Corresponding Editor:	Andrew Wilmot
Published:	2014-06-05	Editor-in-Chief:	Morgan McGuire