Tobler's First Law of Geography, Self Similarity, and Perlin Noise: A Large Scale Analysis of Gradient Distribution in Southern Utah with Application to Procedural Terrain Generation

Ian Parberry

Technical Report LARC-2014-04

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

June 2014

UNT
UNIVERSITY OF
NORTH TEXAS

Tobler's First Law of Geography, Self Similarity, and Perlin Noise: A Large Scale Analysis of Gradient Distribution in Southern Utah with Application to Procedural Terrain Generation

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

***Abstract*—A statistical analysis finds that in a 160,000 square kilometer region of southern Utah gradients appear to be exponentially distributed at resolutions from 5m up to 1km. A simple modification to the Perlin noise generator changing the gradient distribution in each octave to an exponential distribution results in realistic and interesting procedurally generated terrain. The inverse transform sampling method is used in the amortized noise algorithm to achieve an exponential distribution in each octave, resulting in the generation of infinite non-repeating terrain with the same characteristics.**

***Index Terms*—$1/f$ noise, Digital Elevation Model (DEM), exponential distribution, fractal, geospatial analysis, gradient distribution, heightmap, Perlin noise, procedural content generation, procedural terrain generation, self-similarity, sigmoid, United States Geographical Survey (USGS), Utah.**

I. Introduction

Games such as Minecraft[1] take place in an artificially generated terrain. The task of generating this content is called *procedural terrain generation*. As noted in Smelik et al. [1], procedural terrain generation is often based on fractal noise generators such as Perlin noise (for more details see, for example, Ebert et al. [2]). We show how to make terrain generated from fractal noise more realistic by incorporating observations from a statistical analysis of gradients in Utah.

Tobler's First Law of Geography [3] states that "*All things are related, but nearby things are more related than distant things*". In this spirit, we investigate how related nearby and distant terrain gradients are over an area of 160,000 square kilometers in Utah. Utah was chosen because it has both mountainous and flat terrain, and because the Utah Automated Geographic Reference Center[2] has correlated elevation data for the state of Utah from the United States Geological Survey with digital satellite images. It is thus easy to tell at a glance whether a region is mountainous or flat. We make some interesting observations about gradients in Utah, including that the average gradient when plotted as a function of (exponentially) increasing distance between sample points describes a sigmoid curve, and that the gradients conform to an exponential distribution.

In earlier work we analyzed elevation data from a much smaller area of Utah and showed how the height distribution from a specific area chosen by a designer for its interest can be incorporated into a version of Perlin noise called *value noise* [4]. The work in this paper, in contrast, shows how realistic terrain can be generated by varying the exponent of an exponential gradient distribution, resulting in direct control of the ruggedness of the generated terrain.

More information about the subject of this paper, including more terrain images, can be found at http://larc.unt.edu/ian/research/tobler/.

II. Analysis of Gradients in Southern Utah

We performed a statistical analysis of gradients computed from elevation data in the Digital Elevation Model (DEM) format at 5m resolution from the Utah Automated Geographic Reference Center. Our data set consists of 400 text files each containing the elevation of points on a 4000×4000 grid at 5 meter resolution, giving a total of 6.4 billion elevations over the square of side 400km within the southern part of the US state of Utah shown in Fig. 1. Each elevation value is given to one decimal place in meters. These files occupy a total of 51GB on disk. The analysis was performed by a program written by the author in he programming language C++.

A. Analysis at 5m Resolution

While coping with a 51GB data set is not a trivial matter, it is not out of range of the current generation of desktop computers. It took 41 minutes to read and parse the data with `fscanf` into a square array of `floats` occupying 25.6GB of memory on an Intel® Core™ i7-3930K CPU @ 3.2GHz with 32GB of RAM and a solid state hard drive. After doing so for the first time we found that the minimum elevation in the area sampled is 436.3 meters, the maximum elevation is 4120.9 meters, and the average elevation is 1888.7 meters.

[1] https://minecraft.net/
[2] http://gis.utah.gov/data/

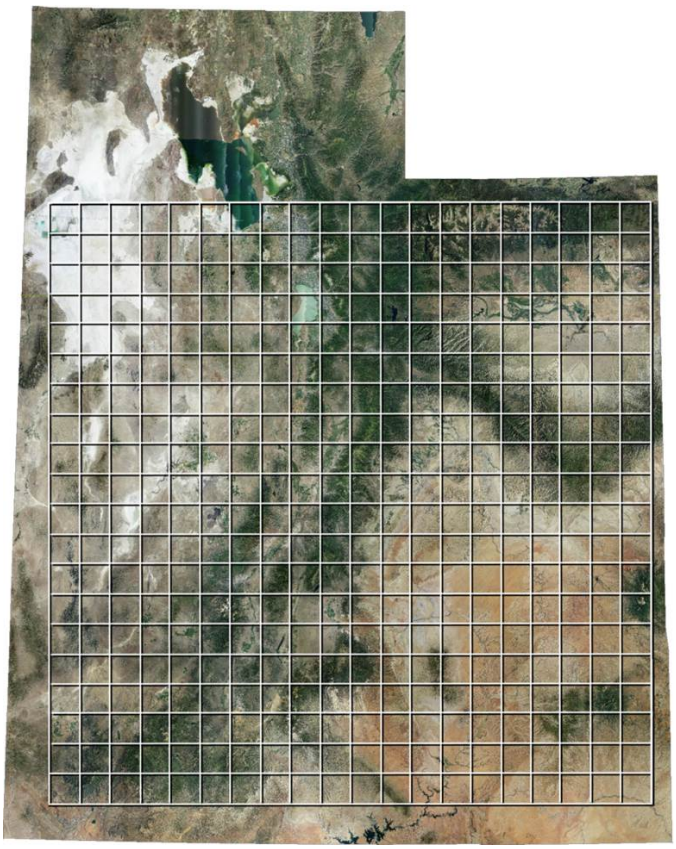

Fig. 1: The area chosen for the gradient study in the US state of Utah. Each of the 400 squares represents a single DEM file containing height data for a 2000×2000 grid of points.

This means that each height value in decimeters will fit into an `unsigned short` which requires 2 bytes as opposed to the 4 bytes used for a `float`, reducing the memory footprint to 12.8GB. Saving the elevation data as a raw binary file halved the amount of disk space required and allowed us to read it with a single call to `fread` in under 5 seconds.

The magnitude[3] of the average gradient over the 160,000 square kilometer area inside the white box in Figure 1 is quite low at 0.1461. The distribution of gradients might be expected to be a normal distribution centered around the average gradient, but it is actually quite close to an exponential distribution as shown in Figure 2.

B. Analysis at 5m–164km Resolution

Motivated by the study of $1/f$ noise (see, for example, [5], [6]) we will look for self-similarity in gradient by analyzing the elevation data in a series of *octaves*. Octave 1 will be the original data at 5m separation. Octave 2 samples every second point, giving a 10m separation. For $1 \leq i \leq 16$, octave i samples every 2^ith point, giving a separation of 5×2^i meters

[3]Since the designation of terrain gradient as being *uphill* or *downhill* is dependent on the orientation of the observer, we will treat gradient as an unsigned value and for the sake of convenience we will omit the word "magnitude" for the rest of this paper.

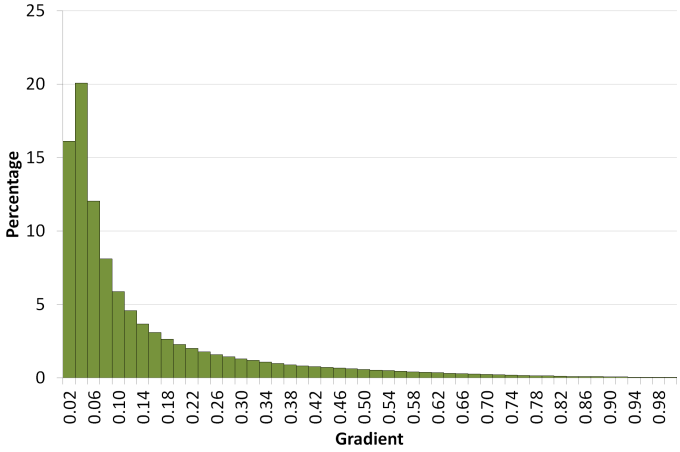

Fig. 2: Gradient distribution at 5m resolution.

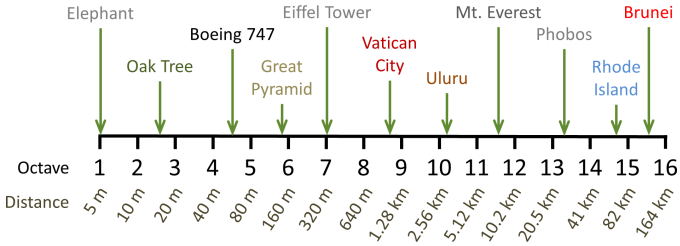

Fig. 3: A list of octaves, the distance between sample points in that octave, and some examples of objects of that size.

(see Figure 4). To give a sense of scale, Figure 3 gives a list of octaves, the corresponding distance between sample points, and some examples of objects of that size.

To avoid loss of statistical significance with each successive octave, each grid size was sampled four times, once as described, and once each offset by half the separation distance horizontally, vertically, and diagonally. The number of sampled grids therefore increases exponentially with octave, offsetting the exponential decrease in the number of points in each grid (see Figure 5). Suppose, for example, the first octave has an $n \times n$ grid of n^2 points, where n is odd (the case where n is even is similar and is left as an exercise for the reader). The second octave has four grids, one of dimension $\lceil n/2 \rceil \times \lceil n/2 \rceil$ (the blue grid in Figure 5), one of dimension $\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor$ (the gray grid in Figure 5, left), one of dimension $\lceil n/2 \rceil \times \lfloor n/2 \rfloor$ (the gray grid in Figure 5, center), and one of dimension $\lfloor n/2 \rfloor \times \lceil n/2 \rceil$ (the gray grid in Figure 5, right), giving a total of $\lceil n/2 \rceil^2 + 2\lceil n/2 \rceil \lfloor n/2 \rfloor + \lfloor n/2 \rfloor^2 = n^2 - O(n)$ sample points for the second octave, etc.

Clearly the average gradient cannot increase from one octave to the next. The average gradient may stay constant from one octave to the next in the contrived case of terrain that slopes uniformly. For example, consider the terrain shown in Figure 6 (top). When sampled over the octave shown in Figure 6 (center), the gradient in each interval is 0.25, and hence the average gradient for the octave is also 0.25. When

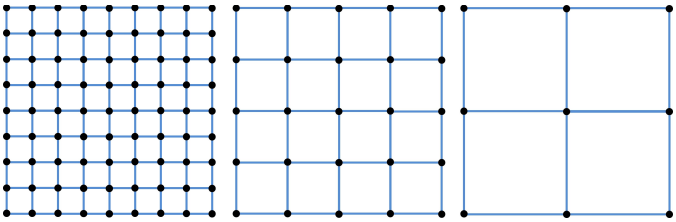

Fig. 4: Example of sampling grids for three successive octaves.

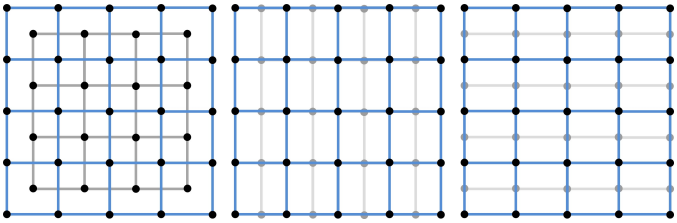

Fig. 5: To avoid loss of statistical significance with each successive octave, each grid is sampled four times in total, offset by half the grid separation distance.

sampled over the next octave, which is shown in Figure 6 (bottom), the gradient in each interval is again 0.25, and hence the average gradient for the octave is also 0.25. The average gradient in this example therefore remains constant at 0.25 from one octave to the next.

The average gradient may drop quite precipitously from one octave to the next in the contrived case of a series of mountains or hills with a wavelength equal to the distance between sample points. For example, consider the terrain

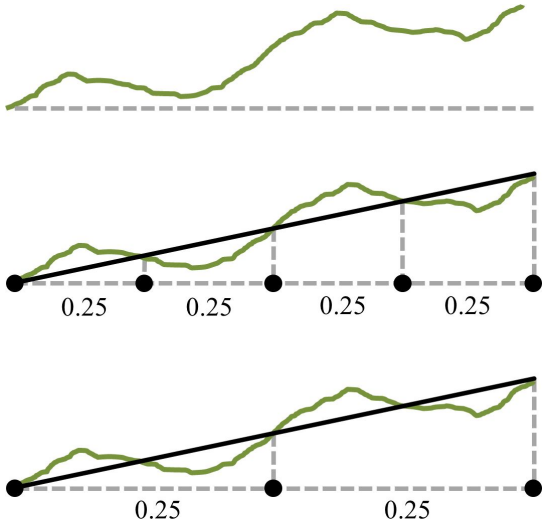

Fig. 6: Example of terrain with gradient that remains constant from one octave to the next. Top: The terrain. Center: When sampled at the points in the first octave, the average gradient is 0.25. Bottom: When sampled at the points in the next octave, the average gradient is still 0.25.

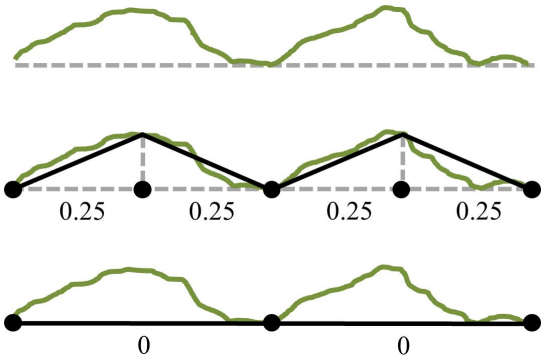

Fig. 7: Example of terrain with gradient that decreases drastically from one octave to the next. Top: The terrain. Center: When sampled at the points in the first octave, the average gradient is 0.25. Bottom: When sampled at the points in the next octave, the average gradient falls to zero.

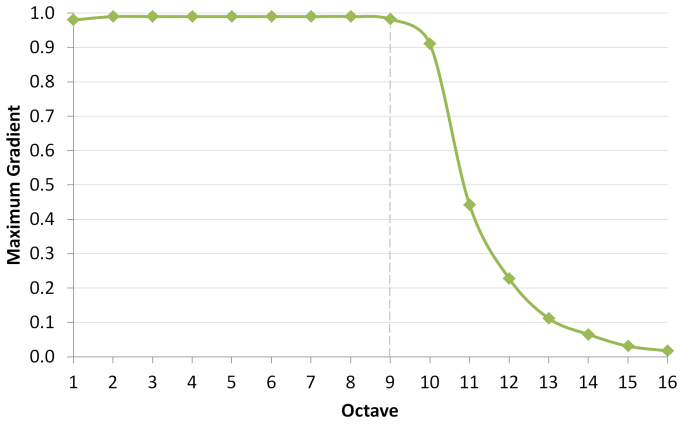

Fig. 8: Maximum gradient by octave.

shown in Figure 7 (top). When sampled over the octave shown in Figure 7 (center), the gradient in each interval is 0.25, and hence the average gradient for the octave is also 0.25. When sampled over the next octave, which is shown in Figure 7 (bottom), the gradient in each interval is 0, and hence the average gradient for the octave is also 0. The average gradient has in this example has therefore dropped from 0.25 in one octave to zero in the next. In general though, we expect the gradient to decrease from one octave to the next between these two extremes.

This analysis took under 16 minutes elapsed time on an Intel® Core™ i7-3930K CPU @ 3.2GHz with 32GB of RAM and a solid state hard drive. Figure 8 shows the maximum gradient by octave, which remains fairly steady for octaves 1–9. At octave 10 it starts to drop because the distance between the sample points is $5 \times 2^10 = 5120$ meters, while the maximum height difference is at most $4121 - 436 = 3685$ meters, giving a maximum gradient of approximately $3685/5120 \approx 0.72$. Thereafter the maximum gradient drops exponentially with octave since the horizontal distance between sample points increases exponentially yet the maximum height difference can

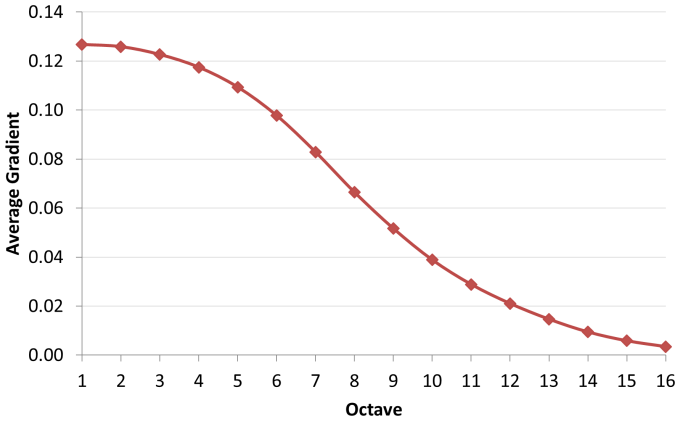

Fig. 9: Average gradient by octave.

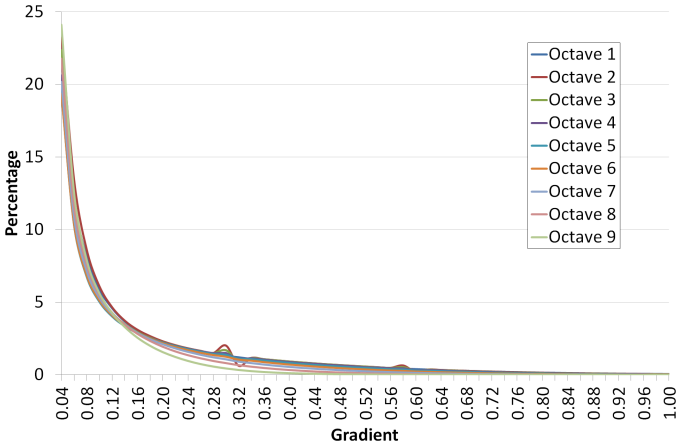

Fig. 10: Gradient distribution for octaves 1–9, at resolutions from 5m to 1.28km.

be no more than 3685 meters.

The average gradient is in general quite small, and when plotted by octave it appears to follow an s-shaped curve (see Figure 9). Octave 8 is a turning point at which the second derivative changes sign from positive to negative. Gradient distributions for octaves 1–9 are shown in Figure 10 (the gradient distribution for octave 1 has already been seen in Figure 2). All appear at least superficially to be an exponential curve.

III. Perlin Noise

Perlin noise was developed by Ken Perlin [5], [6] as a source of continually varying smooth random noise. It has found use in many applications including texture generation and terrain generation.

A. Standard Perlin Noise

Formally, the 2D Perlin noise function $h : \mathbb{R}^2 \to [-1, 1]$. The Perlin noise algorithm starts by pre-computing a table $\mathtt{g2}$ of unit gradients and a random permutation $\mathtt{p}$ to be used as a hash function. To find a noise value y at $(x, z) \in \mathbb{R}^2$, it first finds the four closest integer points $(\lfloor x \rfloor, \lfloor z \rfloor)$, $(\lfloor x \rfloor + 1, \lfloor z \rfloor)$, $(\lfloor x \rfloor, \lfloor z \rfloor + 1)$, and $(\lfloor x \rfloor + 1, \lfloor z \rfloor + 1)$, where for all $x \in \mathbb{R}^+$, $\lfloor x \rfloor \in \mathbb{Z}^+$ is the smallest integer that does not exceed x. It then combines values from $\mathtt{g2}$ evaluated at these using the pre-computed permutation table $\mathtt{p}$, interpolates between them to (x, z) and smooths using cubic splines.

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 I: The 2D Perlin noise algorithm.

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

TABLE II: The 2D Perlin noise algorithm with gradient magnitude.

To obtain interesting noise we sum noise values from the 2D Perlin noise function at various frequencies and amplitudes in a process called $1/f$ *noise* or *turbulence*. Noise at a single frequency is called an *octave*. The amplitude is multiplied by the *persistence* (usually 0.5) from one octave to the next. The frequency is multiplied by the *lacunarity* (usually 2) from one octave to the next.

For terrain generation, a Perlin noise value $y = h(x, z)$ is normalized to $[0, 1]$, multiplied by a scale value Δ and used as the height of the terrain $h'(x, z) = \Delta(h(x, z) + 1)/2$ for each point (x, z) in the 2D Cartesian plane. Figure 11 shows terrain generated with 8 octaves of Perlin noise. (All images of terrain in this paper were rendered offline by Terragen 3[4] from a DEM file generated using the algorithms described here.) Although 2D Perlin noise is invariably described as having range $[-1, 1]$, it is relatively easy to prove that the range is more properly $[-\gamma, \gamma]$, where $\gamma = 1/\sqrt{2} \approx 0.7071$. This is easily rectified by multiplying the output by $\sqrt{2} \approx 1.4142$. Table I gives pseudocode for Perlin's `noise2` function.

B. Exponentially Distributed Perlin Noise

We add an array of scalar magnitudes of the same size as the gradient array, and preload it with the appropriate values. Let m_{00}, m_{10}, m_{01}, and m_{11} be the magnitudes at the integer corner points with gradients $\vec{g}_{00}$, $\vec{g}_{10}$, $\vec{g}_{01}$, and $\vec{g}_{11}$, respectively. We then modify Perlin's 2D noise function as shown in Table II. Since we learned in Section II that the gradient distribution in every octave of southern Utah is an exponential curve, we fill the magnitude table with exponentially decreasing values using the following snippet

[4] http://planetside.co.uk/products/terragen3

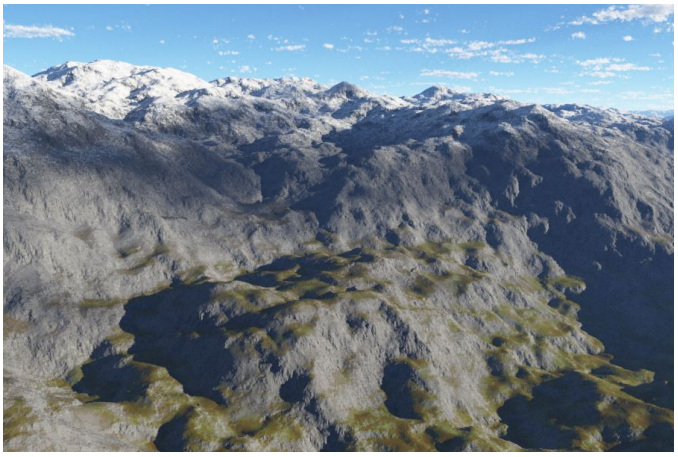

Fig. 11: Terrain generated using exponentially distributed Perlin noise with $\mu = 1$, which is identical to classical Perlin noise.

of C code, in which `MU` is a floating point value μ such that $\mu \geq 1$.

```
float m[B]; //magnitude table
const float MU = 1.01f; //for example
float s = 1.0f; //current magnitude
for(int i=0; i<B; i++){ //for each entry
  m[i] = s; //set i'th magnitude
  s /= MU; //decrease next magnitude
} //for
```

The smallest value in the gradient magnitude table `m`, which is `m[B-1]` $= \mu^{B-1}$, should be one that actually be represented as a floating point number. The smallest normalized floating point value is $2^{-(2^f-2)}$, where f is the number of bits in the exponent ($f = 7$ for a `float`, $f = 10$ for a `double`). We therefore want $\mu^{B-1} \leq 2^{2^f-2}$, that is, $\mu \leq 2^{(2^f-2)/(B-1)}$. Using the standard implementation of Perlin noise with `floats` and $B = 256$ means that we should ensure that $\mu \leq 2^{126/256} < 1.1637$.

Figures 11, 12, 13, 14, and 15 show terrain generated using 8 octaves of exponentially distributed Perlin noise and, respectively, $\mu = 1$, 1.002, 1.007, 1.02, and 1.05. These examples were generated using the same random number seed and were rendered from the same location with the same camera orientation, and therefore they clearly illustrate the effect of changing the parameter μ. Notice that although the terrain gets generally flatter as μ increases, not all gradients are decreased at the same rate. Figure 16 shows the maximum and minimum elevations for a random 4096×4096 terrain with $\Delta = 5000$ against μ. Notice that the maximum and minimum elevations are at least 800 meters apart for $1 \leq \mu \leq 1.16$.

IV. Amortized Noise

Perlin noise repeats with period nB, where B is the size of Perlin's gradient table (commonly chosen to be 256),

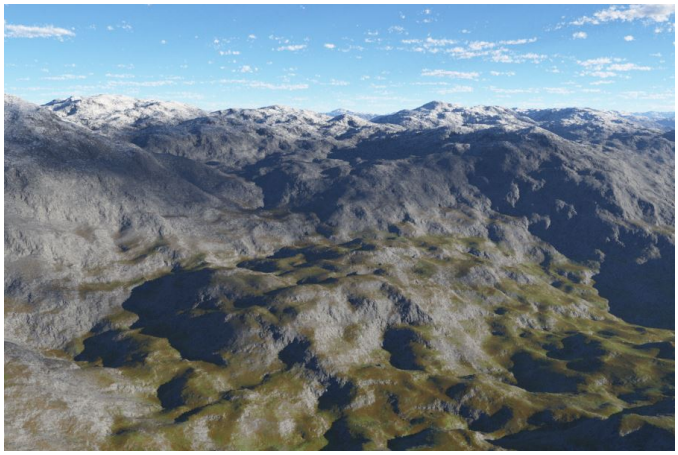

Fig. 12: Terrain generated using exponentially distributed Perlin noise with $\mu = 1.002$.

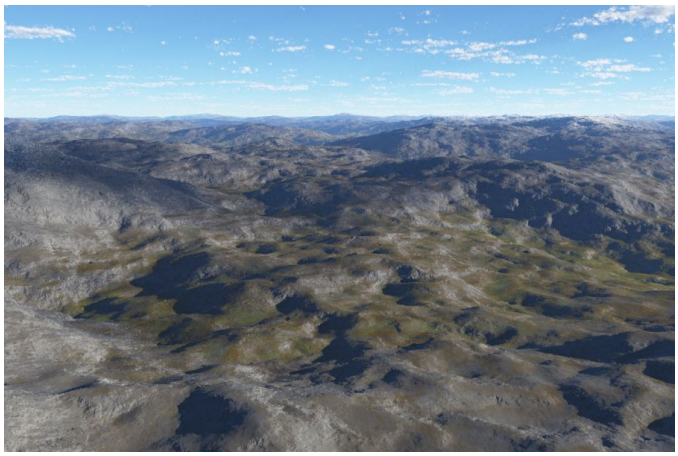

Fig. 13: Terrain generated using exponentially distributed Perlin noise with $\mu = 1.007$.

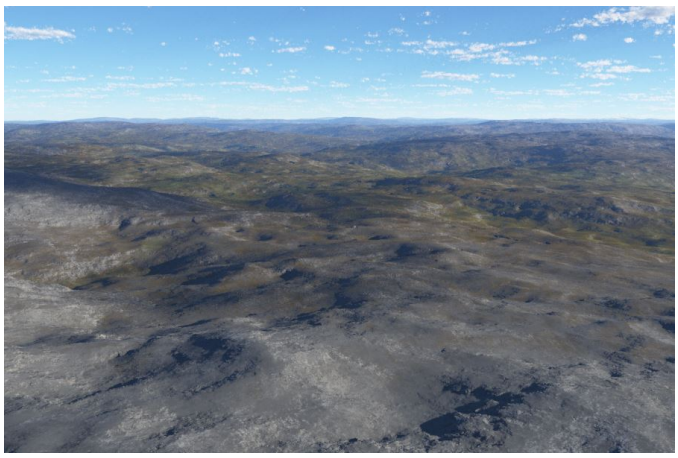

Fig. 14: Terrain generated using exponentially distributed Perlin noise with $\mu = 1.02$.

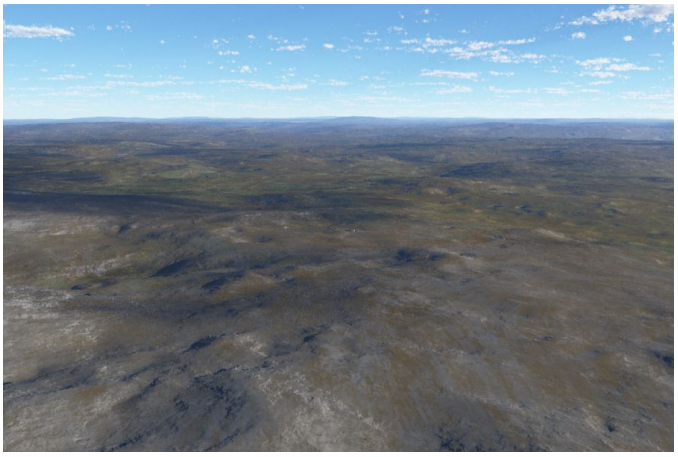

Fig. 15: Terrain generated using exponentially distributed Perlin noise with $\mu = 1.05$.

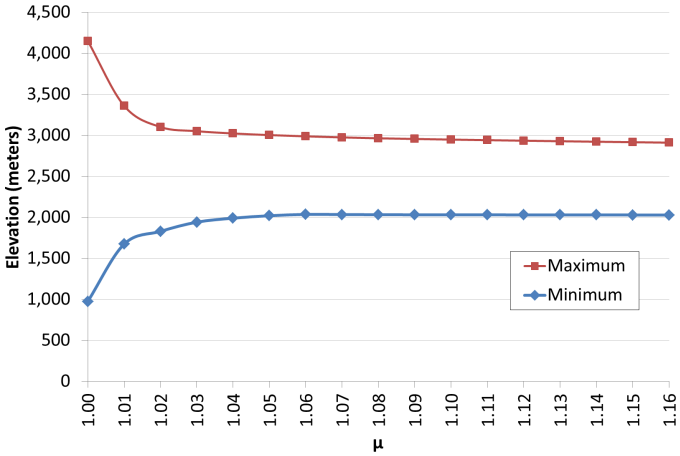

Fig. 16: Maximum and minimum elevations for a random 4096×4096 terrain with $\Delta = 5000$.

and n is the number of points between integer values. This means that any supposedly "infinite terrain" generated from Perlin noise will in fact repeat. The author of this paper has shown recently [7] a fast method for generate potentially infinite non-repeating 2D noise called *amortized noise*. Infinite amortized noise uses gradient $\vec{g}([x, y]) = [\cos\theta, \sin\theta]$, where $\theta = h(2^{32}x + y)$ and h is the the open source hash function MurmurHash3[5]. The magnitudes of the gradient vectors are, as with Perlin noise, always equal to unity.

A. Exponentially Distributed Hash Functions

The modification to amortized noise requires the computation of a gradient magnitude for each grid point, that is, we need a function that hashes the (x, y) co-ordinates of integer points into the real interval $(0, 1)$ with an exponential distribution. As far as the author is aware, the term *hash function* has until now meant a function that maps its domain pseudorandomly into its range with as near to a uniform distribution as possible. Here we generalize this concept to a function that maps its domain pseudorandomly into its range with as near to a given *target distribution* as possible. For definiteness, we will use the term *uniform hash function* for the standard hash function whose target is the uniform distribution, and *exponentially distributed hash function* for one whose target is the exponential distribution.

[5] https://code.google.com/p/smhasher/wiki/MurmurHash3

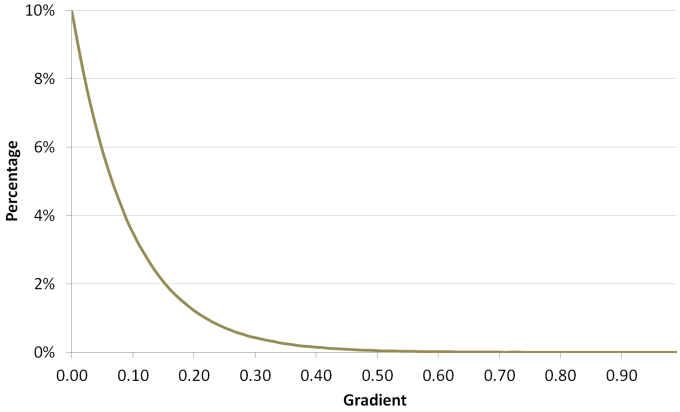

Fig. 17: The distribution of 10^7 calls to `ExpHash(rand(), MAX_RND)`.

We will use the following `typedef` to make sure that our code does not wrap over the end of a line.

```
typedef unsigned int uint;
```

Function `NHash` takes a uniformly hashed value `x` between zero and `max` and normalizes it to the interval $(0, 1)$. For MurmurHash3, `max` is the largest 32-bit `unsigned int`, that is, $2^{32} - 1 =$ `0xFFFFFFFF`.

```
float NHash(uint x, uint max){
 return ((float)x+1.0f)/((float)max+2.0f);
} //NHash
```

Function `ExpHash` takes a uniformly hashed value `x` between zero and `max` and transforms it to a value chosen from the interval $(0, 1)$ using an exponential distribution. The *inverse transform sampling method* says that to achieve an exponential distribution, apply the inverse of the cumulative probability density function (that is, the logarithm) to values sampled uniformly from $(0, 1)$.

```
float ExpHash(uint x, uint max){
 static const float s =
   1/log(0.5f*((float)max + 2.0f));
 return -s*log(0.5f*(float)x+1.0f) + 1.0f;
} //ExpHash
```

Figure 17 shows the distribution of 10^7 calls to `ExpHash(rand(), MAX_RND)`.

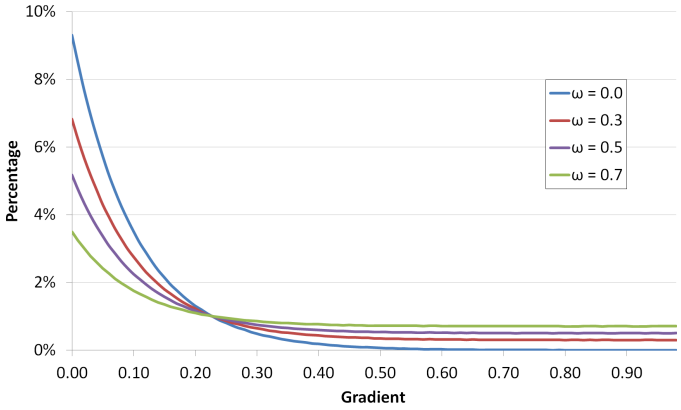

Fig. 18: The distribution of 10^7 calls to `ExpHash(rand(), MAX_RND, ω)`.

The second version of `ExpHash` has a floating point parameter `omega` used to raise the tail of the exponential distribution by a fraction ω with $0 < \omega \leq 1$.

```
float ExpHash(uint x, uint y, uint max,
  float omega)
{
 return NHash(y, max) < beta?
   NHash(x, max): ExpHash(x, max);
} //ExpHash
```

Figure 18 shows the distribution of 10^7 calls to `ExpHash(rand(), MAX_RND, omega)` for `omega`$= 0.0, 0.3, 0.5, 0.7$.

B. Exponentially Distributed Amortized Noise

The amortized noise algorithm [7] uses the following function `initEdgeTables` to initialize a set of tables called *edge tables*.

```
void initEdgeTables(const int x0,
 const int y0, const int n)
{
 //compute gradients at corner points
 uint b00 = h(x0,   y0);
 uint b01 = h(x0,   y0+1);
 uint b10 = h(x0+1, y0);
 uint b11 = h(x0+1, y0+1);

 //fill gradient tables from corners
 FillUp(uax, cosf((float)b00), n);
 FillDn(vax, cosf((float)b01), n);
 FillUp(ubx, cosf((float)b10), n);
 FillDn(vbx, cosf((float)b11), n);
 FillUp(uay, sinf((float)b00), n);
 FillUp(vay, sinf((float)b01), n);
 FillDn(uby, sinf((float)b10), n);
 FillDn(vby, sinf((float)b11), n);
} //initEdgeTables
```

It calls the following 2D hash function `h` based on MurmurHash3 on a stored `seed`.

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

We modify this code as follows. Instead of using a single hash seed `seed`, we use three seeds `s0`, `s1`, and `s2`. `s0`, like the original `seed`, is a hash seed for gradient orientation, `s1` is a hash seed for gradient magnitude, and `s2` is a hash seed for the tail of the gradient magnitude distribution. Function `h2` is modified to use the seed as a parameter.

```
uint h(const uint x, const uint y,
 const uint s)
{
 uint result;
 unsigned long long key =
  ((unsigned long long)x<<32) | y;
 MurmurHash3_32(&key, 8, s, &result);
 return result;
} //h
```

The initial block of code in function `initEdgeTables` above for computing `b00`, `b01`, `b10`, and `b11` is replaced by the following.

```
 //compute gradients at corner points
 uint b00 = h(x0,   y0,   s0);
 uint b01 = h(x0,   y0+1, s0);
 uint b10 = h(x0+1, y0,   s0);
 uint b11 = h(x0+1, y0+1, s0);
```

We then insert the following new block of code for computing gradient magnitudes `m00`, `m01`, `m10`, and `m11`.

```
 //compute magnitudes at corner points
 const uint M = 0xFFFFFFFF;
 float m00 = ExpHash(h(x0, y0, s1),
   h(x0, y0, s2), M, omega);
 float m01 = ExpHash(h(x0, y0+1, s1),
   h(x0, y0+1, s2), M, omega);
 float m10 = ExpHash(h(x0+1, y0, s1),
   h(x0+1, y0, s2), M, omega);
 float m11 = ExpHash(h(x0+1, y0+1, s1),
   h(x0+1, y0+1, s2), M, omega);
```

Finally, the last block of code in function `initEdgeTables` above, which fills the edge tables, is replaced by the following:

```
 //fill gradient tables from corners
 FillUp(uax, m00 * cosf((float)b00), n);
```

```
FillDn(vax, m01 * cosf((float)b01), n);
FillUp(ubx, m10 * cosf((float)b10), n);
FillDn(vbx, m11 * cosf((float)b11), n);
FillUp(uay, m00 * sinf((float)b00), n);
FillUp(vay, m01 * sinf((float)b01), n);
FillDn(uby, m10 * sinf((float)b10), n);
FillDn(vby, m11 * sinf((float)b11), n);
```

Figures 19, 20, 21, and 22 show terrain generated using 8 octaves of exponentially distributed amortized noise and, respectively, $\omega = 0.5$, 0.4, 0.3, and 0.2. These examples were generated using the same random number seed and were rendered from the same location with the same camera orientation, and therefore they clearly illustrate the effect of changing the parameter ω. Notice that although the terrain gets generally flatter as ω increases, not all gradients are decreased at the same rate.

V. Conclusion and Further Work

We have shown that the average gradient in a large area of southern Utah when plotted as a function of increasing distance describes an s-shaped curve, and that the gradients in each octave are roughly exponentially distributed. We have also shown how to make terrain generated from Perlin noise more realistic by using these observations to tune the output of the 2D Perlin noise generation algorithm, and shown that the same modifications can be applied to amortized noise [7] to obtain infinite nonrepeating terrain.

Some interesting open questions remain. We conjecture that gradients are exponentially distributed in all areas of the world. However, a close examination of Figure 10 reveals some interesting lumps and bumps (at gradients 0.28–0.34, for example). These may not be significant, but on the other hand they may be unique to the area under study. It is interesting to ask whether any of the local variations in gradient distribution can be used to generate particular kinds of terrain, in the same way that local variations in height distribution can be

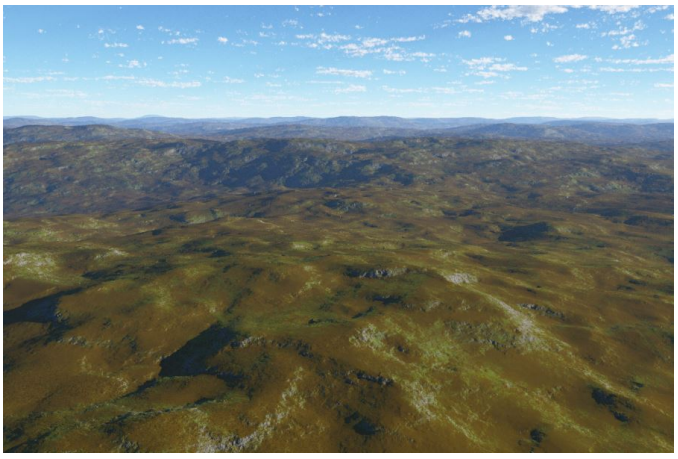

Fig. 19: Terrain generated using exponentially distributed amortized noise with $\omega = 0.5$.

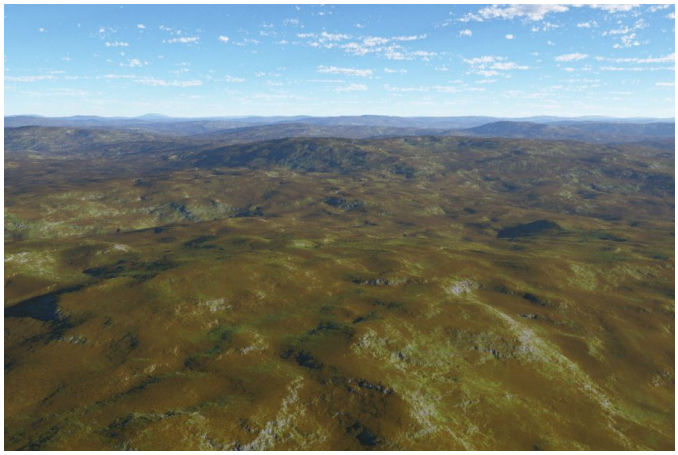

Fig. 20: Terrain generated using exponentially distributed amortized noise with $\omega = 0.4$.

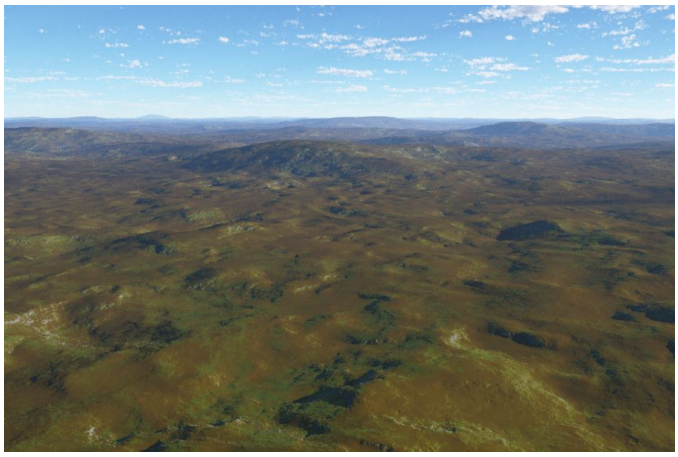

Fig. 21: Terrain generated using exponentially distributed amortized noise with $\omega = 0.3$.

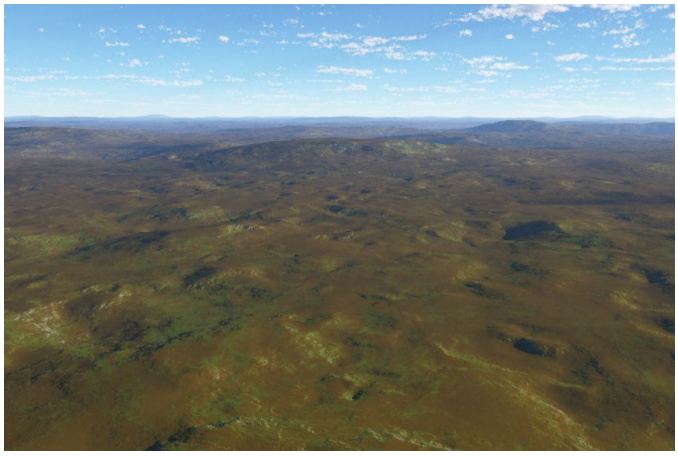

Fig. 22: Terrain generated using exponentially distributed amortized noise with $\omega = 0.2$.

used by a designer to generate terrain suggestive of particular geographical areas [4].

REFERENCES

[1] R. M. Smelik, K. J. De Kraker, T. Tutenel, R. Bidarra, and S. A. Groenewegen, "A survey of procedural methods for terrain modelling," in *Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS)*, 2009.

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

[3] W. Tobler, "A computer movie simulating urban growth in the Detroit region," *Economic Geography*, vol. 46, no. 2, pp. 234–240, 1970.

[4] I. Parberry, "Designer worlds: Procedural generation of infinite terrain from real-world elevation data," *Journal of Computer Graphics Techniques*, vol. 3, no. 1, pp. 74–85, 2014.

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

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

[7] I. Parberry, "Amortized noise," *Journal of Computer Graphics Techniques*, p. To Appear, 2014.

INDEX OF SUPPLEMENTAL MATERIALS

Source code and data is available under the GNU All-Permissive License at https://github.com/Ian-Parberry/Tobler. There you will find five folders, each of which contains a program that will help you to reproduce the results of this paper. You will find full C++ source code that compiles under both Visual Studio 2012 and gcc. Each folder contains a Microsoft Visual Studio 2012 project and a Unix makefile.

1) `Generate with Perlin Noise`. If you wish to generate terrain using Perlin noise with an exponential gradient distribution from Section III, then compile and run the program in this folder which will generate random terrain in a DEM file `output.asc`. You will also find a Terragen project file `output.asc` that can be used to render the terrain from `output.asc`. A subfolder called `Terrain Images` contains copies of Figures 11, 12, 13, 14, 15, and some supplementary images.
2) `Generate with Amortized Noise`. This folder contains a second version of the Generate program using amortized noise [7] with the modifications from Section IV. A subfolder called `Terrain Images` contains copies of Figures 19, 20, 21, 22, and some supplementary images.
3) `Exponential Distribution`. If you wish to verify the code for generating exponentially distributed random numbers from Section IV, then compile and run the program in this folder. An Excel spreadsheet called `exponential.xlsx` contains a copy of the data generated by this program used to generate Figure 17 and Figure 18.
4) `Pack`. If you wish to go further and verify the gradient analysis in Section II, then begin by downloading the DEM files listed in `filelist20x20.txt` from the Utah Automated Geographic Reference Center at http://gis.utah.gov/data/. You will need approximately 51GB of disk space to store these files. Compile and run the Pack program, which will read the DEM data and pack it into a binary file `UtahDEMData.bin` for faster processing. You will need an additional 11GB of disk space to store this file, but the 51GB of DEM files that you downloaded may be deleted after this step.
5) `Analyze`. Continuing verification of the gradient analysis in Section II, after running the Pack program above, move the resulting packed binary data file `UtahDEMData.bin` from the `Pack` folder to the `Analyze` folder, then compile and run the Analyze program. The results will be placed in `output.txt`. An Excel spreadsheet called `utah20x20.xlsx` contains the data from `output.txt` and the graphs from Figures 2, 8, 9, and 10.

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/tobler/.