Procedural textures using tilings with Perlin Noise

Conference Paper · July 2012

DOI: 10.1109/CGames.2012.6314553

CITATIONS
4

READS
946

3 authors:

David Maung
The Ohio State University
10 PUBLICATIONS **39** CITATIONS
SEE PROFILE

Yinxuan Shi
The Ohio State University
6 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Roger Crawfis
The Ohio State University
113 PUBLICATIONS **2,123** CITATIONS
SEE PROFILE

Procedural Textures Using Tilings With Perlin Noise

David Maung, Yinxuan Shi, and Roger Crawfis
Department of Computer Science
The Ohio State University
Ohio, USA

***Abstract*—In this paper, we demonstrate the use of tiling with noise to generate rich procedural textures. We introduce the idea of storing tiles which consist of only the gradients stored at the integer lattice points and constructing a texture on the GPU from these tiles. We also introduce the idea of using mipmapped tiles to store gradients for turbulence. Finally we demonstrate a novel use of mipmaps to generate infinite aperiodic textures with varying frequency patterns.**

Keywords; noise, tiling, procedural textures

I. INTRODUCTION

Procedural content generation is used to create increasingly realistic and interesting content for games and film, while CPU and GPU performance continue to increase and reduce the overall cost of such techniques. Tiling can be used to reduce the memory and development cost of textures used in video games and film. Here we study the use of procedural textures and tiling together. Specifically we examine tiles of noise. One difficulty encountered when generating procedural textures is to avoid repetition. Repetition in textures reduces believability. This can be avoided using aperiodic tiling. We present a method of constructing a tile set which can produce an aperiodic tiling of Perlin Noise.

II. RELATED WORK

Ken Perlin introduced the concept of noise and its use in computer graphics in [1] and [5]. Later, in [2] Perlin improved the properties of his noise algorithm and provided a reference implementation which is ubiquitous today. Since his introduction, other noise algorithms have been considered. In [8], Cook and DeRose use wavelets to generate noise with improved frequency qualities. Wavelet noise is less prone to problems with aliasing and detail loss. Goldberg introduces the concept of anisotropic noise which is more suitable for anisotropic filtering [9]. Lagae proposed Gabor noise which offers more accurate spectral control with intuitive parameters such as orientation, principal frequency, and bandwidth [11]. A good survey of procedural noise functions can be found in [6].

The concept of aperiodic tiling has been an interesting mathematical problem for years. Stam introduces the idea of aperiodic texture mapping of homogeneous textures [3]. In [7], Cohen further popularized the use of Wang Tiles in computer graphics. We introduce a fast, low memory footprint GPU implementation of tiling of Perlin Noise. Secondly, we introduce a technique of providing tileable turbulence on the GPU. Finally, we show how to implement varying frequency noise with tiles.

III. NOISE TILES

Current state of the art with regard to seamless tiling still uses periodic tiling [4]. Consider a noise function $F : \mathbb{R}^2 \rightarrow \mathbb{R}$. A seamless tile can be generated by a function $G : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined as follows:

```
G(x, y) = (
       F(x, y) * (w - x) * (h - y) +
       F(x - w, y) * (x) * (h - y) +
       F(x - w, y - h) * (x) * (y) +
       F(x, y - h) * (w - x) * (y)
       ) / (w*h)
```

Here w is the width of the resulting tile, and h is the height. A tile generated like this is seamless along the edges; however, it still generates noticeable artifacts when tiled. Figure 1 shows a plane tiled with a seamless tile of Perlin Noise. This figure shows a discernible pattern even though the tile is seamless. Furthermore, the noise is also blurred in the middle of the tiles due to averaging from the equation above.

To improve on this we wish to construct an aperiodic tiling without blurring or repetition. This involves the generation of a tile set containing multiple tiles of noise. To construct such a tile set, we must first consider the properties of the tiles themselves and then consider the properties of the noise with which we wish to fill the tiles.

Cohen describes how a set of Wang Tiles can be used to tile the plain. The edges of the tile are considered colored, where color is abstracted to mean a combination of color, pattern, and appearance along the edge. In order to seamlessly tile a plane, the pattern on the right edge of one tile must match the pattern on the left edge of the tile immediately to its right. Similarly, the pattern on the bottom edge of a tile must match the pattern on the top edge of the tile immediately below. Stam noted that the transition must be smooth between the tiles; however, the colors need not match exactly. He proposed a set of $N \, x \, N$ Tiles with each tile containing an edge width of K where $K < N/2$.

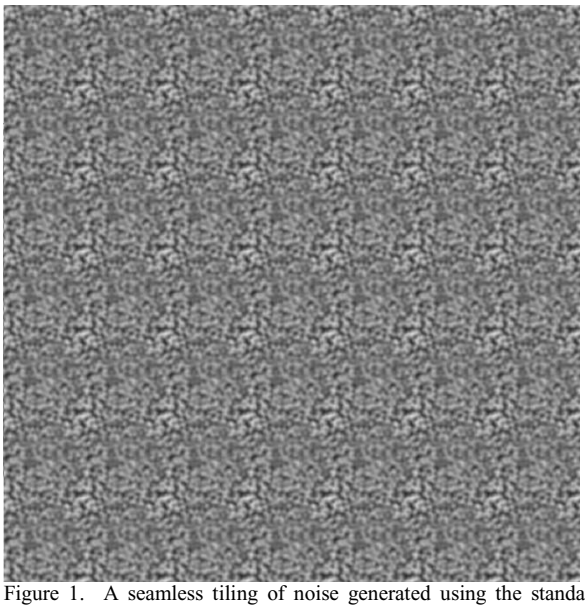


Figure 1. A seamless tiling of noise generated using the standard method described in Equation 1.

Most papers develop a complicated graph cutting algorithm to generate matching colors across the tile boundary. Our technique takes advantage of the Perlin Noise algorithm to generate seamless boundaries. Perlin Noise is generated by using a set of random gradient values specified on the integer lattice. Each gradient value is either chosen using a pseudo random algorithm, or set using a permutation of the special coordinates.

Each gradient value has a local influence on the overall noise. For any given point(x, y) in $\mathbb{R}^2$, Perlin Noise is a bi-quintic interpolation between the gradients at the four surrounding integer lattice points. The range of influence of a given gradient is a window ± 1 unit in the x and y directions. Now let us consider two tiles constructed of Perlin Noise. These two tiles can be seamlessly tiled horizontally by constructing the gradients along the left edge of one tile to match the gradients along the right edge of another as shown in Figure 2. Note that this involves making sure an $N\ x\ 1$ vector of gradients on the left edge of one tile matches the $N\ x\ 1$ vector of gradients on the right edge of the other. Similarly these tiles can also be seamlessly tiled vertically by replacing the gradients on the top of one tile with the gradients on the bottom of the other. Since the $N\ x\ 1$ column is arbitrary and does not affect the rest of the tile, the width K of the Wang Tile needed for Perlin Noise is $K = 1$.

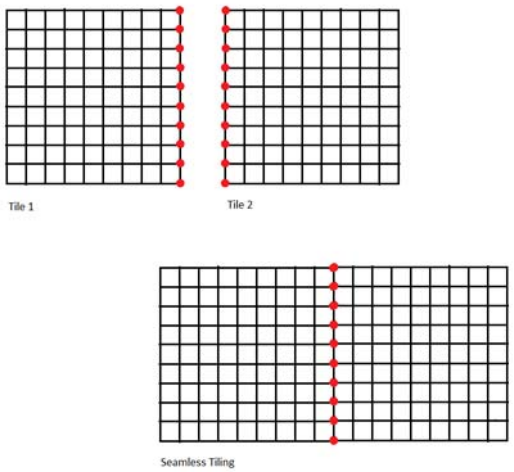


Figure 2. Above, two tiles are constructed such that the gradients at the locations represented by the red dots are the same. When these two tiles are joined (lower), they will be seamless.

IV. Constructing Noise Tiles

Using the above, we construct a set of Wang Tiles of Perlin Noise to aperiodically tile a plane. In order to do this, let us consider tiles of Perlin noise which cover an $N\ x\ N$ integer lattice. We define a color C as an $N\ x\ 1$ vector of gradients. Given a set of n colors, $\{C_0, C_1, \ldots, C_{n-1}\}$, we construct a Wang tile out of Perlin noise by constructing the integer lattice as shown in Figure 3. We usually choose the tiles such that N is a power of 2.

The corner is shared among 2 edges. Furthermore each corner is adjacent to 3 other tiles in a tiling. Gradients that do not match at the corner would create seams in the image. We resolve this by using a single common gradient for all corners. Our testing shows that if $N > 4$ the choice of a single corner color does not develop discernible patterns in the noise. We have also used the corner tiling method shown in [10] to add variability to the corners. In this case, an edge color is a unique combination of the corner colors on its left and right side, giving n^2 edge colors for n corner colors.

V. Turbulence Tiles

So far, we have discussed a single frequency Perlin Noise; however, Perlin Noise is often used in multiple frequencies or bands. Perlin first described this in [1] as 1/f noise which we will call turbulence. Turbulence can be simulated as:

$$Turbulence(point) = \sum_{i=1}^{n} \frac{Noise(point * 2^i)}{2^i}$$

73	52	231	127	2	18	38	144	37	210	34	233	21	63	183	73
151	109	174	183	253	94	84	151	220	224	18	38	22	138	203	29
227	15	149	60	139	237	67	215	32	120	102	24	238	128	164	206
72	116	27	62	157	56	18	112	33	170	86	90	89	125	250	195
52	19	208	58	27	62	109	127	97	127	60	44	228	226	126	219
120	136	164	128	9	160	39	233	237	91	187	84	120	177	111	241
184	202	69	53	174	107	248	173	122	128	210	156	213	3	124	28
238	18	219	57	157	121	191	173	244	106	183	10	86	9	255	250
214	199	95	18	180	95	14	30	26	21	234	198	127	98	101	148
15	100	110	204	109	68	32	158	129	66	72	191	41	118	149	138
17	66	207	245	30	127	162	194	211	226	68	92	226	44	148	4
46	43	121	234	145	39	26	110	66	27	2	77	253	249	60	16
58	190	110	74	78	140	48	88	252	245	145	101	218	54	217	48
10	115	181	247	182	110	92	179	6	150	231	123	232	2	112	15
207	60	42	51	95	88	122	38	187	121	242	231	185	193	160	193
73	18	134	176	9	8	198	28	1	98	141	194	188	143	252	73

Figure 3. A representation of the square integer lattice from which a single tile is constructed. Each byte at coordinate (x, y) represents an index to a gradient. All the corners are the same value. Each edge is constrained to a specific edge color.

Constructing noise in this fashion requires more gradient values than is provided by a single tile and would require us to sample across a tiling to provide turbulence for each tile. To keep the coordinates within a single tile, we instead we use mipmaps to generate Turbulence Tiles. We modify the standard turbulence equation above to be

$$Turbulence(point) = \sum_{i-1}^{n} \frac{Noise_{\lambda_i}(point)}{2^i}$$

Where λ_i represents a new set of gradients for each band of noise. Given a tile of Perlin noise based on an $N \times N$ integer lattice, to generate noise of half the frequency we construct an integer lattice of $\frac{N}{2} \times \frac{N}{2}$. From the turbulence equation, we construct Turbulence as a weighted sum of i octaves.

We can generate the noise for each octave from a different mipmap level. For these to tile correctly, we construct each mipmap level using a modified version of the Noise Tile construction algorithm above. Consider the corner colors to be a set of gradients with one gradient for each octave. Extend the edge colors to be a set of vectors of sizes $N \times 1, \frac{N}{2} \times 1, \frac{N}{4}$, ..., where we use one vector for each mipmap level. The number of octaves available for turbulence is determined by the number of mipmap levels available and the maximum tile size. We use a minimum tile size of 4 x 4. Therefore, if we have 128 x 128 pixel tiles we can have 6 mipmap levels or octaves.

VI. Varying Frequencies

Another use of mipmap tiles is to generate noise of varying frequencies. First, note that each mipmap level represents a doubling in frequency of noise. If we smoothly interpolate from one mipmap level to the next, we can generate a noise pattern that appears to vary smoothly in frequency. A simple linear interpolation between mipmap levels does not work because the frequency increases exponentially as the mipmap level increases. Instead we use $log2(\ freq\ )$ to determine the interpolation between two mipmap levels. We can then generate a frequency function over the 2D plane $f(x, y)$ with which to modify the noise. We adjust our noise function to be $Noise(x, y, f)$ to allow us to specify the isotropic frequency at each position on the tile. We use the frequency to determine an index into the mipmap or the interpolation between two mipmap levels.

VII. Implementation

A. Noise Tiles

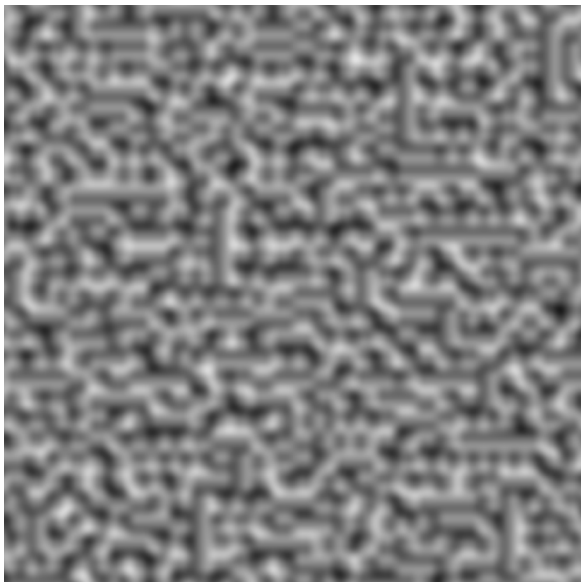
To construct the tiling on the CPU would require a large amount of memory. Consider a 128 x 128 tiling of Perlin Noise with each tile using a 16 x 16 array of gradients. We would have to pass a 2048 x 2048 texture to the shader to implement this tiling, using 4Mb. Instead, we pass the gradients for the tiles as a texture array to a GPU shader. We also pass a texture which includes the tiling. In the above example, for a complete set of tiles with 2 colors per edge ($2^4 = 16$ possible tiles), we would need one 16x16x16 texture array and one 128 x 128 texture for the tiling, using a total of 20Kb.

The gradients for each tile are generated using the following algorithm:

1) Choose a random corner gradient G.
2) Construct an array for edge colors C[M,N] where M is the number of edge colors and N is the size of the gradient tile.
   a. For m = 0 to M -1, C[m,0] = G, C[m,N-1] = G
   b. Set the rest of the elements of the array to random gradients.
3) Each tile in the tile set has a top, bottom, left, and right edge color. Set the edges of the T[N,N] array of gradients for each tile.

```
For n = 0 to N - 1
        T[n,0] = C[top,n]
        T[n,N-1] = C[bottom,n]
        T[0,n] = C[left,n]
        T[N-1,n] = C[right,n]
Next
```

4) Fill the middle of the tile with random gradients.


a

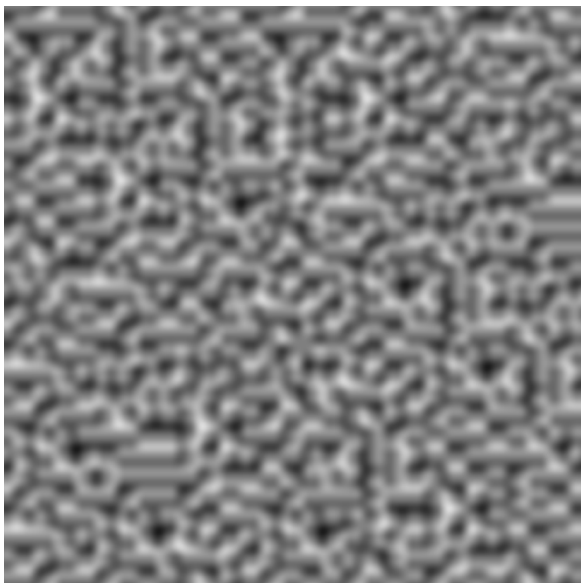

b

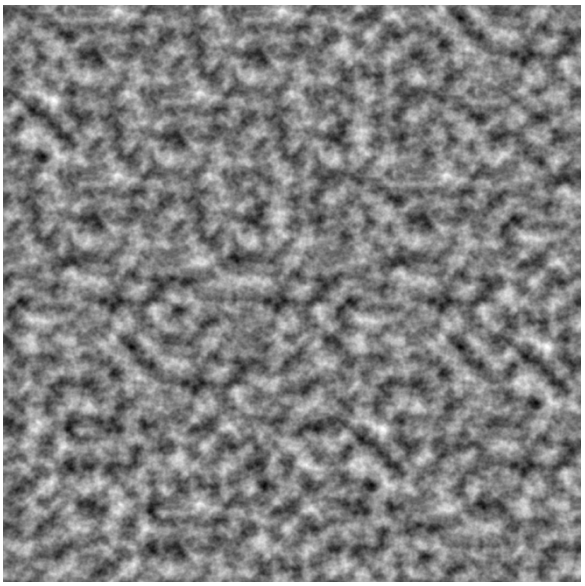
Figure 4. Original Perlin vs. Tiled Perlin. The Original Perlin is above and Tiled Perlin is below.

These gradient values are stored in a texture array with one $N \; x \; N$ texture for each tile. The final texture is computed on the GPU.

Figure 4 shows a plane tiled with Wang tiles constructed in this manner on the bottom. The top is standard Perlin noise shown for comparison.

VIII. TURBULENCE TILES

We implement turbulence on the GPU similarly passing the gradients as a texture; however, for turbulence, each frequency is stored as a separate mipmap level as described in Turbulence Tiles above. Figure 5 shows a comparison between turbulence generated from tiles and the Perlin implementation.


a

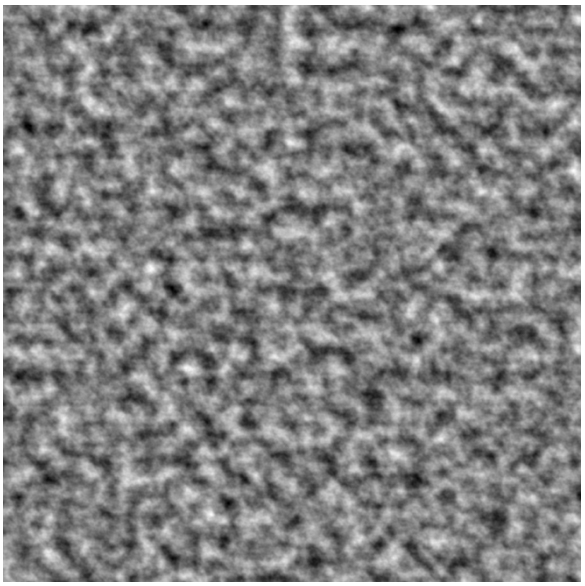

b

Figure 5. Original Turbulence vs. Tiled Turbulence. Original Turbulence is above and Tiled Turbulence is below.

IX. VARYING FREQUENCY TILES

We implement the varying frequencies described in section VI by implementing a frequency function applied on top of our noise function over the whole tiling. We provide two sample tilings of varying frequency in Figure 8. First we show a linear ramp function as the frequency increases linearly in the x direction across the tiling in Figure 8a. Second, we show an interesting pattern where the frequency itself is noise in Figure 8b.

X. MARBLE

Marble is also implemented on the GPU. To generate marble, we subtract different layers which are composed of turbulence themselves. Because the gradients for each tile are fixed, the

turbulence for a specific frequency for each tile is unique. Since we want to subtract differing layers of turbulence, we modify the algorithm so that each layer is generated from a different tile in the tiling. The first layer of turbulence is generated as described above. For each subsequent layer, we increment our x or y index into the tiling to generate an offset layer of turbulence. These layers of turbulence can be subtracted to generate the marble pattern. Figure 8d shows marble generated with this method.

XI. WOOD

Our wood pattern is generated using three layers of Perlin Noise of differing frequency. The first layer gives coarse grained detail, the second layer concentrates on color variation, and the third layer provides a rough sanded effect. Figure 8c shows a tiling of wood generated using three layers of noise.

XII. CONCLUSION

The use of noise in textures is ubiquitous and important in the gaming and film industries to add realism. We have shown a method of producing aperiodic tilings of noise. Generating the tiles on the GPU realizes a savings in memory usage and increases flexibility. We have extended our GPU implementation to include a novel use of mipmaps to generate turbulence. In addition, we can use these mipmaps to vary the frequency of the noise across the tile. We also show that you can still produce complex patterns such as wood and marble using tilings of noise. These techniques allow a rich set of interesting aperiodic tilings of noise to be used for texturing in gaming and films.

XIII. REFERENCES

[1] K. Perlin, "An Image Synthesizer," in *Proceedings of the 12th annual conference on computer graphics and interactive techniques (SIGGRAPH '85)*, New York, NY, USA, 1985.

[2] K. Perlin, "Improving Noise," *ACM Trans. Graph.,* vol. 21, no. 3, pp. 681-682, July 2002.

[3] J. Stam, "Aperiodic Texture Mapping," European Research Consortium for Informatics and Mathematics, 1997.

[4] M. Zucker, "Perlin Noise Math FAQ," 2001. [Online]. Available: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. [Accessed 22 05 2012].

[5] K. Perlin and E. Hoffert, "Hypertexture," in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1989.

[6] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. Ebert, J. Lewis, K. Perlin and M. Zwicker, "A Survey of Procedural Noise Functions," *Computer Graphics Forum,* vol. 29, no. 8, pp. 2579-2600, 2010.

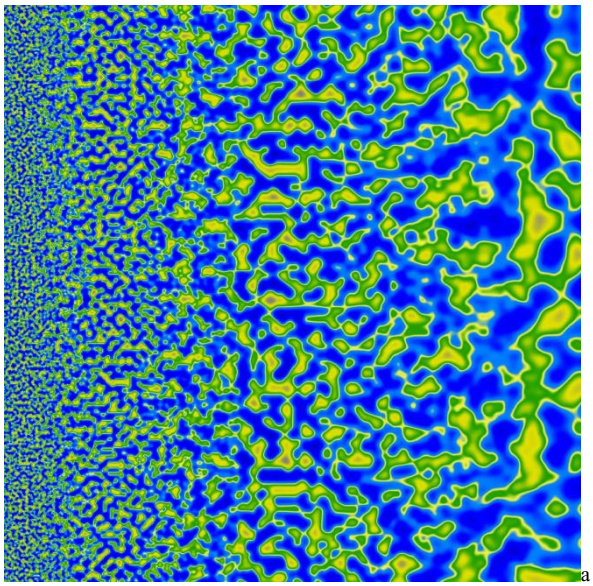
[7] M. F. Cohen, J. Shade, S. Hiller and D. Oliver, "Wang Tiles for image and texture generation," New York, NY, USA, 2003.

[8] R. L. Cook and T. DeRose, "Wavelet Noise," in *ACM SIGGRAPH 2005 Papers*, New York, NY, USA, 2005.

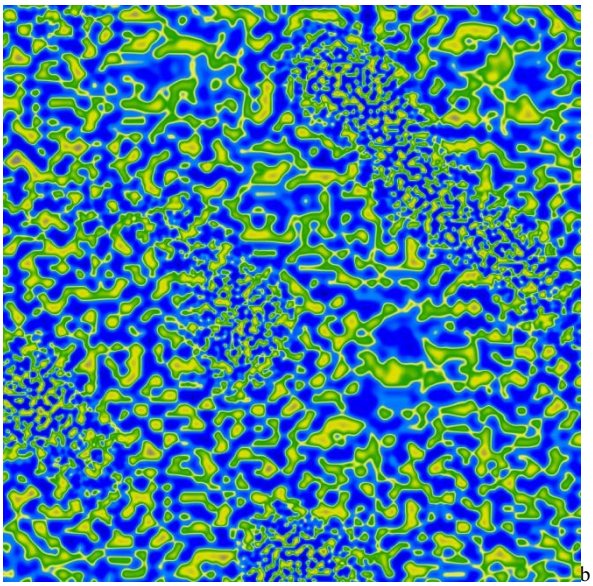
[9] A. Goldberg, M. Zwicker and F. Durand, "Anisotropic Noise," in *ACM SIGGRAPH 2008 papers*, New York, NY, USA, 2008.

[10] A. Lagae and P. Dutré, "An alternative for Wang tiles: colored edges versus colored corners," *ACM Trans. Graph.,* vol. 25, no. 4, pp. 1442-1459, October 2006.

[11] A. Lagae, S. Lefebvre, G. Drettakis and P. Dutré, "Procedural noise using sparse Gabor convolution," *ACM Trans. Graph.,* vol. 28, no. 3, pp. 54:1-54:10, July 2009.


a


c


b

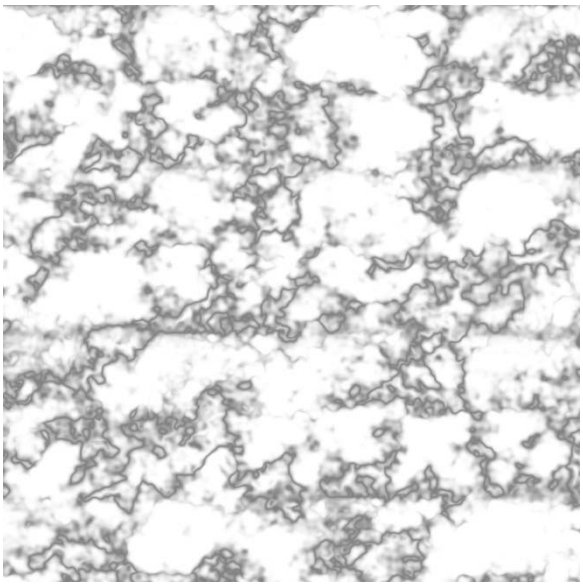

d

Figure 5. The results of applying various functions on tiles. Images (a) and (b) show tilings with smooth varying frequency. Image (c) shows a wood pattern, and Image (d) shows a marble pattern.