Parallel White Noise Generation on a GPU via Cryptographic Hash

Stanley Tzeng　　Li-Yi Wei

Microsoft Research Asia

Abstract

A good random number generator is essential for many graphics applications. As more such applications move onto parallel processing, it is vital that a good parallel random number generator be used. Unfortunately, most random number generators today are still sequential, exposing performance bottlenecks and denying random accessibility for parallel computations. Furthermore, popular parallel random number generators are still based off sequential methods and can exhibit statistical bias.

In this paper, we propose a random number generator that maps well onto a parallel processor while possessing white noise distribution. Our generator is based on cryptographic hash functions whose statistical robustness has been examined under heavy scrutiny by cryptologists. We implement our generator as a GPU pixel program, allowing us to compute random numbers in parallel just like ordinary texture fetches: given a texture coordinate per pixel, instead of returning a texel as in ordinary texture fetches, our pixel program computes a random noise value based on this given texture coordinate. We demonstrate that our approach features the best quality, speed, and random accessibility for graphics applications.

Keywords: noise, random number generation, texturing, parallel computation, GPU techniques

1 Introduction

Producing high quality random numbers is important for a variety of disciplines such as graphics, simulation, signal processing, and cryptography. An ideal source of random numbers is a random variable with uniform distribution. The uniform distribution is often called the "white noise" since signals drawn from such a distribution contain a roughly equal amount of energy across all frequency bands.

One way to acquire a white noise is to measure random physical quantities such as the thermal noise of a resistor. However, physical noises are not very suitable for computer simulation since they are hard to control and non-repeatable. An alternative approach is to produce pseudo-random numbers on a computer via mathematical methods such as linear congruential regression [Knuth 1998]. This is perhaps the most popular method in generating uniform random numbers and is employed in most compiler libraries and numerical simulation software. Unfortunately, linear regression and similar methods have several major drawbacks:

Statistical Randomness Classical random generators, including linear congruential regression, have been shown to seriously bias the outcome of certain simulations [Brands and Gill 1996]. As a corollary, the usual approach taken by many graphics researchers that a random generator "is good enough if the results look random enough" may not be sufficient for certain applications (e.g. Monte Carlo sampling or numerical simulation).

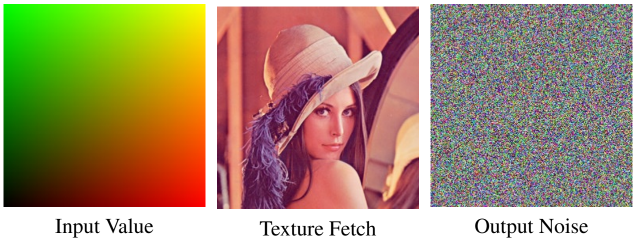

Figure 1: Parallel order-independent white noise generation. Our system functions just like an ordinary texture fetch (middle), but instead of returning a texture we compute a random noise (right) based on the input texture coordinates (left).

Parallel Computation Classical random generators are not parallel in the sense that the statistical randomness is guaranteed only if a long sequence of numbers is generated sequentially. This implies that if a parallel algorithm needs to access such a random number source, we have to either serialize the computation, which reduces performance, or pre-compute the random numbers, which consumes storage. Although there are efforts to parallelize such sequential random number generators on GPUs (e.g. [Sussman et al. 2006]), such an approach remains fundamentally difficult.

Order Independence On a notion related to but different from parallel computation, for GPU applications we would like to have a parallel white noise generator that allows random-accessible/order-independent evaluation [Peachey 1990; Wei and Levoy 2002; Lefebvre and Hoppe 2005], where any subset of the total samples can be evaluated in any order while guaranteeing invariant results with a fixed input. This order-independence property is important for graphics applications because, similar to texture mapping, we do not know in advance which samples would be required in which order at run time. Ideally, we would like to be able to compute any subset of noise samples at any order at a constant cost per sample. This cannot be achieved by a sequential random number generator as ensuring order-independent computation of the last sample would almost certainly require the computation of all preceding samples.

We have conducted a comprehensive survey on graphics and signal processing literature and, to our surprise, we have found very little emphasis on white noise generators that simultaneously satisfy the requirements listed above. Instead, we have found this very same problem studied extensively in cryptography, despite under a very different context. In this paper, we point out the equivalence of a good parallel order-independent white noise generator, which we need for GPU graphics applications, and a good cryptographic hash algorithm, which has been studied primarily for the purpose of encryption, security, and authentication.

We propose an implementation of such a cryptographic hash-based random number generator on a GPU pixel program. This allows us

I3D 2008, Redwood City, California, February 15–17, 2008.
© 2008 ACM 978-1-59593-983-8/08/0002 $5.00

to compute random numbers in parallel just like ordinary texture fetches: given a texture coordinate per pixel, instead of returning a texel as in ordinary texture fetch, our pixel program computes a random noise value based on this given texture coordinate. (See Figure 1.) Our method ensures order-independence since the underlying cryptographic hash ensures invariant results with respect to a given input texture coordinate value.

Our method allows parallel and completely independent random noise generation per pixel with constant time complexity, and results produced by our approach exhibit excellent statistical randomness. Our method is also very easy to implement and runs fast on modern commodity GPUs.

Given these desirable properties, we believe our method can be implemented as a universal uniform random number generator on GPU shading languages. As many current noise functions under prevailing shading languages (such as Cg and GLSL) are often left unimplemented, our random number generator can be used to implement these noise functions.

2 Previous Work

Random number generation contains a huge literature. A good survey on sequential random number generators (PRNGs) can be found in the seminal work by Knuth [Knuth 1998]. Here, we only cover prior art most relevant to our technique, including parallel random number generators and applications of random numbers in graphics.

Parallel PRNGs Most existing parallel PRNGs use a "baking" methodology where a set of random numbers is batch generated by multiple sequential generators and the resulted is stored in memory (hence the term "baking" as from [Olano 2005]). These methods are parallel but not random accessible. Some early examples include [Mascagni and Srinivasan 2000; L'Ecuyer et al. 2001; Entacher et al. 1998]. [Sussman et al. 2006; Pang et al. 2006] explored shader implementation of PRNGs on commodity GPU as well as potential new hardware features. These methods are not designed to be random accessible and they require a texture to advance the state of the PRNG. Several recent techniques also explore the NVIDIA CUDA architecture [NVIDIA 2007b] such as [Podlozhnyuk 2007; Howes and Thomas 2007]. These CUDA-based methods deviate away from the realm of graphics and use the GPU solely as a parallel processor. Our goal in keeping the graphical abstraction is to show that many graphical related concepts can still be used with our algorithm and is not just designed for simulations. Furthermore, CUDA is able to transfer its computed data into the rendering pipeline but only through the use of texture memory. We strive for a PRNG that can be implemented entirely inside the pixel shader without relying on texture memory.

Several techniques have adopted a hash-based heuristic to PRNG generation to create memory-less and random accessible PRNGs. The Combined Explicit Inverse Congruential Generator (CEICG) [Entacher et al. 1998] was the only successful ported PRNGs in [Sussman et al. 2006]. This generator has the properties of a hash as it uses no memory and is random accessible, which was more suitible for graphics hardware than the other two PRNGs attempted in the paper. Olano [Olano 2005] uses the Blum Blum Shub (BBS) [Blum et al. 1986] hash on the GPU for generating noise on graphics hardware. Alas, despite their efficiency and suitability for GPU implementation, these heuristics usually fall short of producing random numbers with sufficient statistical quality. We provide a detailed analysis in Section 4.

Applications There are numerous applications of random numbers in graphics. Perhaps the most famous example is Perlin noise [Perlin 1985; Perlin 2002], where all noise samples are derived from a small pre-computed permutation table that emulates a random number source. Random numbers are also widely used in other kinds of procedural textures as detailed in [Ebert et al. 1998]. Olano proposed a more GPU-friendly implementation of Perlin noise [Olano 2005] as opposed to common implementations which pre-compute the whole noise image on the CPU and transfer the image onto video memory as a texture [Rost 2004]. As pointed out in [Olano 2005], this pre-computation method can be expensive for storage and access since traditional 2D Perlin noise textures requires a resolution of 768^2 or 1024^2 for accurate sampling. Furthermore, this method does not work well for other texturing methods, such as texture synthesis [Lefebvre and Hoppe 2005], where the size of the generated texture may well exceed the available texture memory. Lefebvre [Lefebvre and Neyret 2003] and Wei [Wei 2004b] also use the 1D hash table method as a permutation table for tile-based texturing. Even though the tile location hashing can be computed on the fly as shown in [Wei 2004b], the author reported significant performance degradation compared to a pre-baked tiling. These applications suffer from the heavy texture accesses per fragment, which becomes a bottleneck in the application. Our approach would provide a more suitable random number source for these applications.

3 Our Approach

To meet the goal of a parallel order-independent white noise generator with excellent statistical randomness, our basic approach is to replace a traditional sequential random number generator with a cryptographic hash. In the following, we first describe some general properties of cryptographic hashes, highlighting why they are ideal candidates for our goal. We then describe our design decisions in settling down on MD5 as the specific cryptographic hash function for our implementation. We discuss the best practices to utilize such hash functions on a GPU, as well as detailed GPU implementation and optimization issues.

3.1 Cryptographic Hash

We will now give a brief overview of the properties of hashes that make hashes suitable for our goals and end with a description of MD5. Hashes, unlike sequential PRNGs, do not use an internal state to help generate numbers. Popular PRNGs, such as linear congruential generators, have an internal state that must be updated with each iteration of the generators. Thus, these PRNGs become order dependent as the time to retrieve the $(i+n)^{th}$ output given the i^{th} output is linear with n. Hashes can generate both the i^{th} and $(i+n)^{th}$ output in constant time.

This feature is crucial for GPU implementation, as it guarantees order invariance in the output. Since the rasterization order of fragments is unknown at run time, it is essential that the PRNG is random accessible. The data a fragment generates and the times it takes to generate the data should be invariant to the order that fragment is processed. Hashes fall into this category as they generate an output given only an input and have no internal state. Cryptographic hashes are a subset of hashes. Unlike traditional hashes concentrating on efficiency and minimizing conflicts (e.g. see [Lefebvre and Hoppe 2006]), cryptographic hashes possess additional properties that make then suitable as a white noise generator [Wayner 2002].

Cryptographic hashes take in a message of arbitrary bit length and outputs a message digest of a fixed length. The hash should be one way (i.e. given a message digest, it should be highly improbable if not impossible, to recover the original message), and it should be very hard to find two different inputs that produce the same digest. Given any message, the probability of the n^{th} bit of the digest being a 1 should be 50%. Finally, two digests should be completely uncorrelated to one another, regardless of how similar their inputs are. A cryptographic hash can be seen as a PRNG where the outputs from consecutive inputs are uncorrelated. The probability of any sequence of bits being chosen is constant for all possible combinations of bits. A cryptographic hash is designed to produce white noise distributions in output, even for input as simple as a linear ramp.

MD5 is one of the cryptographic hash functions proposed by Rivest [Rivest 1992]. The algorithm takes in as input any arbitrary bit length, and produces a message digest of 128 bits. MD5 begins by padding the number of bits in the input to the next multiple of 512. It then takes each 128-bit chunk and scrambles the bits with various bitwise operations. These operations, which include bitwise and, or, xor, negation, and circular shifts, are called MD5's compression function. Finally these chunks are summed together as the final 128-bit message digest.

3.2 Design Decisions

Here we justify our decisions on why we chose MD5 in our implementation. Our target hash function should satisfy the following requirements:

1. The algorithm should have a white noise distribution.

2. The algorithm's digest should be power-of-two aligned. Specifically, its message digest length should be less than or equal to the maximum number of bits that can be outputted per fragment in one color buffer. On modern GPUs, this value is 128-bits (four 32-bit integers/floats per pixel).

3. The algorithm should not use up any texture memory, whether it is for states or initialization. All constants needed for the algorithm should fit in shader code.

4. The algorithm should translate well onto pixel shader program on modern graphics hardware and can be utilized similar to a texture map. This is desired because most modern GPU computations (whether rendering or general purpose computations) happen mainly on pixel shaders. And since GPU programmers are already familiar with the texture metaphor, we would like to present our random number generator as a special variation of texturing.

5. The algorithm should be fast to execute.

Our initial candidates include popular cryptographic hash functions used today: SHA-1 [(NSA) 2001], Tiger [Anderson and Biham 1996], MD5 [Rivest 1992], and RIPEMD-128 [Dobbertin et al. 1996]. These functions all satisfy 1, as they are used for security purposes. SHA-1, while very popular, does not satisfy 2. The output of SHA-1 is a 160-bit digest, which is not power of two aligned. While Tiger's output is a 128-bit digest, it's algorithm is implemented on 64-bit architecture. This does not satisfy 4, as modern GPUs have no 64-bit data types and converting between 32-bit and 64-bit values would take too much time. RIPEMD-128 and MD5 share both very similar qualities, but in terms of speed, MD5 has an advantage over RIPEMD-128. RIPEMD-128 is hindered by the extra rounds of scrambling in its design which MD5 does not have. While RIPEMD-128 may seem to be more cryptographically secure, for our purposes of uniform distribution, MD5 suits our needs just as well.

Other cryptographic primitives that we have considered are block ciphers such as the Advanced Encryption Standard (AES) Rijndael algorithm [AES 1997]. Rijndael used under the counter mode [AES 1997] can be random accessible and parallel computable just as hashes. However, since AES is a reversible process (encryption rather than hash), we have found that it is much less random than MD5 (as detailed in Section 4). Furthermore, even though AES has simpler arithmetic operations than MD5, it uses a large lookup table (known as the S-box) that incurs significantly more memory access than MD5. Consequently, the actual performance of AES is not much faster than MD5 on the latest GPUs, and is actually slower than our optimized implementation (as detailed in Section 4).

Although collisions have been found with the MD5 [Wang and Yu 2005], we feel that this fact will not affect our choice. We chose MD5 not because of its cryptographical strength, but because of its uniform distribution in output. Plus, for most graphics applications we can think of, perfect crytographical security is rarely necessary and experimentally we have been happy with the statistical randomness of MD5. (See Section 4 for a detailed analysis.)

3.3 Usage

We now describe how to use MD5 in GPU pixel shaders. As a GPU noise generator, our aim is to construct a noise generator that can be used similar to a hardware texturing unit: given a query (texture coordinate), the algorithm would return a random number (texel). Instead of a level of detail argument used in texture queries, the user would specify a key. We have implemented MD5 on the GPU (henceforth MD5GPU) as the function `whiteNoise` and it is defined as follows:

$$\mathbf{uvec4}\ digest = \texttt{whiteNoise}(texCoord, key)$$

where $texCoord$ are the current texture coordinates and **uvec4** is the new 4-wide unsigned interger data type available on the G80.

One aspect of MD5GPU that differs from the CPU version is the choice of input. The original MD5 allowed a variable input length, but MD5GPU will always have a fixed input length for each fragment. There is no way to alter the input length between fragment processing; this makes the choice of input even more important. If the input for MD5GPU is not properly chosen, the input will not vary enough and repetitions will occur in the output. This may lead to artifacts in the generated noise.

For our input, we chose two sources: a varying 128-bit base and a 32-bit key. There are many choices for the base, and we require that the value must differ with each fragment. The simplest choice is to use texture coordinates as a base. In our implementation, we use one set of texture coordinates (s, t, p, q). We chose a 128-bit base so that we could adhere to the metaphor of texture lookups since texture coordinates on the shader are given also as 4 component vectors. This base choice allows us to support both 2D and 3D noise textures. Although the input into MD5 is specified as 512-bits, we do not wish for users to specify all 512-bits of the input. We find that this is cumbersome for the user and our goal is to keep the usage as simple as possible while producing the highest quality noise. In our tests, we have tried using MD5GPU with only 2D texture coordinates, specified from $(1 \ldots width)$ and $(1 \ldots height)$ and have found the quality to be acceptable (see Section 4 for the results). For 3D texturing the third coordinate (p) can be utilized as well.

The key is used as an extra scrambler to the base and is useful as a quick way to vary the noise when the base stays the same. We combine the two by performing a bitwise xor on each 32-bit segment of the base. Note that bitwise-and and bitwise-or are not viable options, for they may give improper input if a key is chosen poorly (consider 0 for bitwise-and and $2^{32}-1$ for bitwise-or). Many applications use a texture atlas [Ebert et al. 1998] so primitives will map to disjoint locations in a texture map. In this case, the key could be chosen as an object ID. This guarantees that each object will have an unique noise pattern associated with it.

integer to float We have described how we generate random numbers in 32-bit integer format. For applications desiring floating point numbers, the conversion can be done as follows:

$$(2b_0 - 1) \times 1.b_{1\to 23} \times 2^{(b24\to N)+M} \qquad (1)$$

where $b_{p\to q}$ is the integer represented by bits p through q of the input 32-bit integer. In particular, b_0 is the sign bit, $b_{1\to 23}$ the mantissa bits (with leading 1), and $b_{24\to N}$ the exponent bits. For users who only want to use non-negative floats, the term $(2b_0 - 1)$ can be dropped. For the exponent term, N allows the user to choose the exponent range of the random float and M is the proper bias. For example, floats in the range $[1\ 2)$ can be achieved by $N = 23$ (i.e. no bit used) and $M = 0$, floats in the range $[0.5\ 1)$ can be achieved by $N = 23$ (i.e. no bit used) and $M = -1$, and floats in the range $[1\ 4)$ can be achieved by $N = 24$ and $M = 0$. The only restriction is that $(b_{24\to N}) + M$ must be in the range $[-126\ 127]$ to avoid special numbers (denorm, INF, and NAN). Equation 1 can be efficiently implemented by a integer-to-float bit-packing operation under the G80 pixel shader.

3.4 Implementation

MD5 uses extensive integer and bit-wise operations that are unavailable in earlier generations of graphics hardware that supports only floating point arithmetic inside pixel shaders. With the G80, fortunately, integer data types and bitwise operations are now fully supported [NVIDIA 2007a]. In addition, the new integer buffer object extension allows each output channel to be a full 32-bit integer [NVIDIA 2007a]. With a 4-wide 32-bit unsigned integer fragment output, it is now possible to have each pixel contain a message digest.

We have implemented the MD5 algorithm in OpenGL using GLSL for the shading language. Once the input has been set up, it is padded to a 512-bit input with extra zeroes as described in the MD5 specification. This input is fed into the compression function and the resulting digest is stored as an unsigned integer vector. One strong point about MD5 is that it can generate a large amount of random bits per function call as compared to other PRNGs which generate 32 or 64 random bits per function call. We keep the digest in its original form to provide applications with a generous amount random bits so that they do not need to call MD5GPU many times per fragment.

The original MD5 requires a sine function multiplied with 2^{32} to be added to each round of the scrambling. Unforunately, the largest integer representation on GPUs is limited to $2^{32} - 1$. This means that our sine table will be different from the one used in the original MD5. However, we have found that even with this variation, MD5GPU produces acceptable results.

3.5 Optimization

The reference implementation of MD5GPU uses a heavy amount of memory per fragment. First, the algorithm calls for a 64-element array to hold the sine function and many implementations have another 64-element array to hold the order of rotations. Such a heavy memory requirement per fragment becomes a bottleneck on the GPU.

```
**function** digest ← whiteNoise($texCoord$, $key$)
**uint** data [16]
setupInput($texCoord$,$key$,data)
**uvec4** rotate = {7,12,17,22}
**uvec4** digest = initDigest()
**uvec4** tD = digest

// simple one round example
    Ft = F(tD.yzw)
    idx = i
    r = rotate.x
    rotate = rotate.yzwx
    trig = floor(abs(sin(i+1)) * $2^{32}$)
    tD.x = tD.y + leftRotate( (tD.x + Ft + data[idx] + trig), r)
    tD = tD.yzwx;
**return** digest+tD;
```

Table 1: Pseudocode of MD5 optimized. This is an example of how one round on the GPU would work. Here F is the auxiliary function as described in the specification. In the actual algorithm, 64 of these rounds are performed.

Our solution is to compute the sine function on the fly and to reduce the rotation storage to 16 elements. The sine table does not need to be stored and can be computed with a built-in `sin` function. For the rotation, since the rotation only consists of 16 unique numbers rotated around in groups of four times, we can store these 16 numbers as four 4-element vectors. We permute the order of rotations with a swizzle operation. Another optimization that we have done is to unroll the 64 compression function. Many implementations of MD5 use a loop to execute the compression function, but this does not translate very well onto graphics hardware, as there is still a minor penalty for looping. Unrolling the loop has produced yet another significant speedup of MD5GPU.

Table 1 shows a pseudo-code example of one optimized round. In the actual MD5 code, 64 of these rounds are executed in this fashion. Table 3 in Section 4 shows the running time of our optimized algorithm in comparison to the reference GPU implementation. The speedup is averaged around a factor of 20 with respect to the reference implementation and both GPU implementations produce equivalent results, as we do not alter the compression function in any way.

4 Results

In this section, we demonstrate that our choice of MD5 as a GPU random number generator strikes the best balance between quality, speed, and random-accessibility. We achieve this by providing experimental results analyzing the quality and speed of various contending algorithms and implementations. In particular, we will evaluate our MD5GPU against current GPU PRNGs as well as popular CPU PRNG choices used by graphics programmers today.

Our candidate GPU PRNG set consists of Blum Blum Shub shader for noise [Olano 2005], CEICG [Sussman et al. 2006] and AES [Yamanouchi 2007]. For CPU generators, we will compare against Goulburn [Patel 2006], which is a hash-based PRNG specifically designed for graphics applications. In addition, we have examined many source codes for shader based noise applications and have found that the majority uses the `rand` function included with compilers. Therefore we will also compare our results to the GCC implentation of `rand` and the improved `drand48` routine on UNIX machines [The IEEE and The Open Group 2004]. Each one of these PRNGs will be tested in their quality and speed.

Note on CUDA: CUDA based PRNGs, while it can compute random numbers on the GPU as discussed in Section 2, are not considered for our GPU PRNG candidates. CUDA-based PRNGs can send their computed results into shader programs, but only through texture memory. From our point of view, this is no different than previous methods of baking the random numbers into a texture. We consider GPU PRNGs that can be generate numbers solely from shader code without relying on any texture memory.

4.1 Quality

We measure quality of a PRNG based on several criteria: the number of the DIEHARD [Marsaglia 1995] tests passed, how well it passes the DIEHARD test suite, and a view of its power spectrum distribution. We chose DIEHARD as it is the de-facto standard in random number generator testing, and we interpret the results as follows. The number of DIEHARD test passed is based on the output p-value [D'Agostino and Stephens 1986] for each test. This value should fall between 0.01 and 0.99 for a successful completion of each test [Marsaglia 1995].

How well an algorithm passes a DIEHARD test suite is determined by how uniform are the p-values generated. This is assessed by running a Kolmogorov-Smirnov test (KS-test) [D'Agostino and Stephens 1986] on the p-values generated from each test. If the stream of random numbers is truly random, then the DIEHARD p-values should be indistinguishable from a set of uniform values in the range [0,1). As a quality control, we compare with the CPU based MT19937 [Matsumoto and Nishimura 1998], which is a variation of the Mersenne Twister algorithm widely in use today and possesses excellent statistical qualities.

In our KS-tests, we look at the Kolmogorov-Smirnov statistic which we refer to as the variable D. D is defined as the maximum vertical deviation between two curves plotted on a cumultive distribution function [D'Agostino and Stephens 1986]. The two curves will be our test data, and a set of uniformly distributed numbers. The smaller the D, the closer the test data is to resembling a uniform distribution.

Each PRNG will also be measured in its Fourier space with three measurements: the power spectrum distribution(PSD) of the PRNG, the radially averaged PSD, and the anisotropy. These measurements are commonly used as measurements in evaluating Poisson Disk Distributions [Lagae and Dutré 2006b]. For a uniform distribution, we expect to see a smooth PSD image, and flat lines in both the radially averaged PSD and anisotropy. The PSD is found by averaging 10 periodogram samples of each PRNG. As mentioned in [Lagae and Dutré 2006b], it is necessary to take an average of peridograms to estimate the PSD because a periodogram is associated with one sample of noise. A PSD estimate is associated with a specific PRNG.

Note on drand48: `drand48` only outputs its value in floating point. It may seem reasonable to directly convert this value into binary for use with the DIEHARD test suite, but this is not the correct method. Floating points always store the most significant bit as a sign bit, and since `drand48` will never output negative values, that bit will ways be 0. This means that after every 31 bits the test program will always encounter a 0 and the results will be biased. Our solution is to extract the random bits before they are converted into a floating point number. These 48 random bits are fed directly into DIEHARD.

4.2 Speed

Our speed test consists of two tests: a batch rendering test and a texture subset test. The batch rendering test measures the time it takes to generate n^2 128-bit MD5 digests where n is a target resolution size. Our tests consists of $n = 512, 1024, 2048,$ and 4096. In addition to the PRNGs mentioned above, we will compare three versions of MD5. The CPU MD5 will be used as a control, and we have the reference implementation of MD5GPU and the optimized version. Note that PRNGs like Goulburn generates its output as 64-bit unsigned intergers. To make sure that it generates an equal amount of bits, the time for these specific PRNGs are marked after $n^2 \times 2$ iterations. Likewise, the time for 32-bit PRNGs will be marked after $n^2 \times 4$ iterations.

The texture subset test is a measure of the access time for a specific region of a texture. We visualize the noise generated as a large texture that is not stored on graphics memory, so each query for a texel would require the PRNG to generate the noise texels at run time. In our test, we query for a specific region of this large texture and see how long it takes for the PRNGs to generate it. Each output of a PRNG will be directed at one RGBA channel; for one texel, the PRNG may need to run four times. Note that since MD5 generates 4 32-bit values at once, it only needs to run once per texel. Our texture has resolution of $2^{20} \times 2^{20}$ and the region we will be accessing is the bottom right (worst case for sequential computation) and upper left (best case for sequential computation) 512×512 region. Our key requirement is that there is invariance in the values generated. This test will see how random accessible the PRNGs we have tested are, for we expect random accessible PRNGs to generate both regions in constant time.

Our speed tests ran on a Pentium 4 running at 3.2 GHz with 1 GB of RAM. Our graphics card used for testing was a GeForce 8800 GTX with 768MB VRAM. For each test, we report the average of ten trials. The GPU PRNGs of CEICG and Blum Blum Shub were left in its original form but ran on the G80 chip as opposed to their original tests on the G70 chip. In quality tests, hashes were fed 2D texture coordinates and PRNGs used the current time for its seed. There were no modifications to PRNG code from the original discription in their respective papers. All values chosen for their constants were kept as is.

4.3 Analysis

Algorithm	Test Results		
	DH Tests Passed	KS-test D	Failed tests
MD5GPU	15 / 15	0.2029	None
GPU CEICG	10 / 15	0.3202	1,5,6,7,8
GPU BBS	2 / 15	0.9280	All but 11, 14
GPU AES	7 / 15	0.6288	1,4,5,6,7,9,10,12
Goulburn	9 / 15	0.4402	4,6,7,9,10,12
`rand`	6 / 15	0.6212	3,5,7,9,10 11,12,13,15
`drand48`	12 / 15	0.3320	4,6,7
M. Twister	15 / 15	0.1660	None

Table 2: A comparison of PRNGs used in graphics and how they fare with statistical tests. The lower D in the middle column implies a higher quality of the PRNG. In the final column, the number refers to the specific test index as described in the DIEHARD specification.

Table 2 shows our test results on the quality. While many previous papers claim that their have passed the DIEHARD test suite, unfortunately we were not able to replicate their claims. This might be due to different criteria in judging test results. Alas, many previous papers did not document their criteria for passing the DIEHARD test suite.

Our tests show that our MD5GPU is only slightly worse than the quality control Mersenne Twister (MT), but we beat all other potential graphics random number generators. Besides MT, MD5GPU was the only other PRNG to pass all tests of the DIEHARD test suite and we have the most uniform distribution of p-values, as seen in the middle column, from all other graphics PRNGs used.

Our spectral analysis in Figure 2 shows that most PRNGs appear to have a white noise distribution. Specifically they have a flat power spectrum distribution (PSD) and their power is radially symmetric, as shown by the radially averaged PSD and the anisotropy. The fact that many PRNGs have good power spectrum but fail DIEHARD tests indicates that power spectrum is a much looser criteria. This resonates with our observation in the introduction that a random number generator whose results look good enough may not be statistically good enough. The major exception to this rule is GPU BBS. The power spectrum of GPU BBS shows unevenness and peaks of power at several frequencies. In addition, the power is not radially spread out as there are heavy variations in its radially averaged PSD and anisotropy. In its current form, GPU BBS is not suitable for a noise generator, as seen in both its DIEHARD tests and its PSD.

Algorithm	Rendering resolution			
	512^2	1024^2	2048^2	4096^2
MD5CPU	964.1	3812.5	14704.7	58257.8
MD5GPURef	59.3	229.7	904.6	1146.8
MD5GPUOpt	4.7	6.3	39.1	134.3
GPU BBS	< 1	< 1	1.6	4.7
GPU CEICG	20.3	56.2	198.4	692.2
GPU AES	45.3	170.4	679.5	862.9
Goulburn	173.4	676.5	2673.5	106781
rand	23.4	90.6	362.5	1457.9
drand48	428.2	1668.7	6659.4	26673.5
M.Twister	87.5	368.8	1318.8	5260.9
GPU BBS2	15.9	31.2	63.0	172.2

Table 3: Comparison of the running times of the algorithms in the batch rendering test. All times given in the table are in milliseconds. For GPU BBS, the first two values are < 1 because it was faster than the precision of the timer we used. The last row is a modified version of GPU BBS which uses a larger M and extracts only 4 bits per iteration.

Algorithm	Texture Subset Location		
	Upper Left	Lower Right	Difference
MD5CPU	937.5	934.4	3.1
MD5GPURef	60.9	60.9	0.0
MD5GPUOpt	4.7	4.7	0.0
GPU BBS	< 1	< 1	< 1
GPU CEICG	20.4	15.6	4.8
GPU AES	45.1	45.1	0.0
Goulburn	318.7	317.2	1.5
rand	59506.5	481789.0	422282.5
drand48	339218.5	2606250.0	2267031.5
M.Twister	41093.4	339188.7	298095.3
GPU BBS2	15.6	15.7	0.1

Table 4: A comparison of texture subset retrieval times. All times are given in milliseconds. Notice that PRNGs that order dependent take much longer to access the subregion and the times to retrieve the information is not constant.

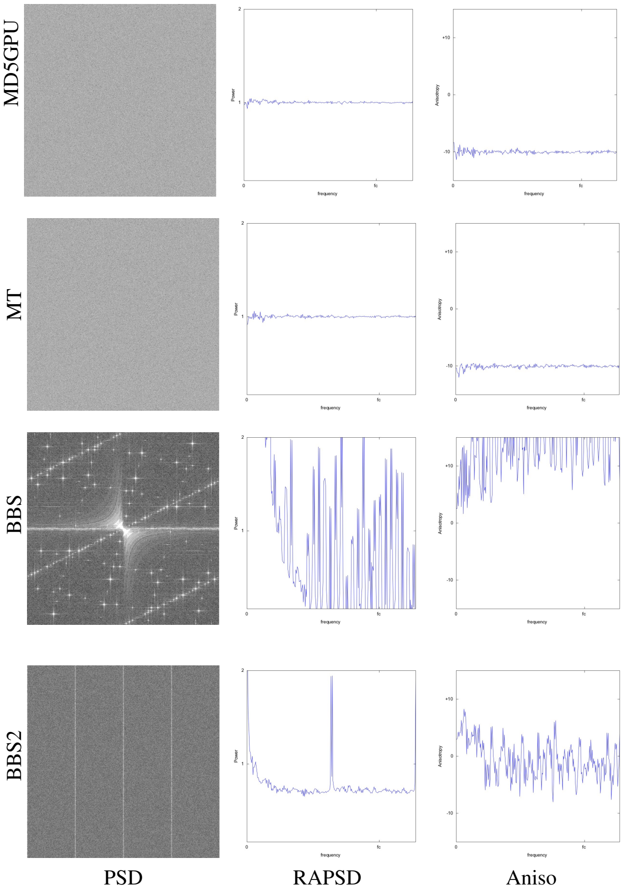

Figure 2: Power Spectrum Images. Images of the Power Spectral Density (PSD), radially averaged PSD (RAPSD), and anisotropy of each PRNG used in quality tests. As many of the images are similar, we have only shown the most significant ones: MD5GPU, MT control, and GPU BBS. The bottom row is GPU BBS with $M = 64769$. Even with a larger M, the PSD images are still rough.

Table 3 and Table 4 shows the results of our speed tests. In terms of speed, all but one of the GPU based PRNGs are faster than CPU based ones. MD5GPURef lags behind CPU based PRNG rand at lower resolutions in the batch rendering test, but its random accessibility outperforms rand in the texture subset test. Non-random-accessible PRNGs suffer greatly from the texture subset test and their retrieval times for both regions are not constant. This is most evident in rand and drand48. The hashes and GPU PRNGs do not suffer from this problem and retrieval of the subset is near constant.

From the other three GPU based PRNGs, GPU BBS is the fastest with instant evaluation at lower resolutions, but the quality of GPU BBS is poor. While it may seem that using a larger M may solve the problem (compared to $M = 61$ used in [Olano 2005]), unfortunately this is not the case. We have implemented GPU BBS on the G80 with integer arithmetic and data types and have found that the resulting PSD is still rough. The last row of Figure 3, Figure 4 and Figure 2 shows the results of a variation of GPU BBS with $M = 251 \times 271 = 64769$. The two primes 251 and 271 satisfy the BBS requirement for primes such that $251 \equiv 271 \equiv 3(mod4)$, and M is less than $\sqrt{2^{32}}$ (to prevent overflow). We adhered to Olano's method of using only the 4 least significant bits per iteration of the hash. For each fragment, the hash is called 32 times. The PSD is smoother than the original GPU BBS, but still not as smooth as MD5GPU. The running time for GPU BBS2 is however slower than our optimized MD5GPU. Using 1 bit per iteration may improve the quality of GPU BBS2, but will slow down the output even more.

CEICG is a PRNG similar to hashes as it does not have an internal state [Entacher et al. 1998]. This made it applicable as a GPU based PRNG in [Sussman et al. 2006], for it can be used just like a hash. Unfortunately, compared to MD5GPU, GPU CEICG has one major drawback: the time it takes to find the inverse will vary. Specifically, finding the modular inverse is an expensive operation and the number of iterations needed is not constant with every fragment. Some fragments will find the inverse faster than others, and thus the pixel workload is not balanced. This is seen in Figure 4, where the two sub-regions on average do not take the same time to access when compared to all the other GPU based PRNGs.

Our MD5GPU always returns a pseudorandom number in constant time and the number it generates is more uniformly distributed than the majority of PRNGs that we have tested. Our optimized implementation MD5GPUOpt is also faster than the majority of previous PRNGS, except for a few that lack in statistical quality (e.g. BBS as implemented in [Olano 2005]). We believe that we have found a PRNG that achieves the maximal balance between quality, speed, and random accessibility. It is possible to modify GPUMD5opt by reducing the number of rounds in the compression to speed up the algorithms. However, we warn readers that this sacrifices the quality of the PRNG for an improvement in speed. Figure 5 shows the result of running GPUMD5opt with a lower number of rounds.

Iterations	Time	DH Tests Passed	KS D-val
64	6.3	15 / 15	0.2029
48	4.7	14 / 15	0.2042
32	3.1	13 / 15	0.2295
16	1.6	11 / 15	0.2530

Table 5: Reducing the compression function iterations in MD5. All iteration reductions were done on GPUMD5opt and times are reported in milliseconds. The rendering resolution for the time test is 1024^2.

5 Applications

We now exemplify two applications enabled by our approach: one for terrain generation (vertex data) and another texture tiling (pixel data).

5.1 Fractal Terrain Generation

As an application of our MD5GPU, we have constructed a terrain generator. This terrain is created by using fractal noise generated with MD5GPU as a height map. We modeled fractal noise via a gradient noise approach as described in [Ebert et al. 1998] where gradient noises at different frequencies are summed together to produce the final fractal noise image. (Please refer to our accompanying video for a fly through of this terrain.)

Our inputs into MD5GPU are the lattice point coordinates and we treat the resulting digest as the gradient for that lattice point. Given a vertex with texture coordinates (s, t), we find the vertex's 4 closest lattice points, compute the noise, and use the noise value as the height of the vertex. To scroll around the terrain, we simply translate the texture coordinates and compute the noise again. In each frame a fixed number of vertices are rendered. Our terrain can scroll almost endlessly in any direction and does not use up any texture memory, as the height map is computed on the fly each time. Due to the random accessibility of our noise generator, we can query any location on the terrain in constant time and the height will always be invariant.

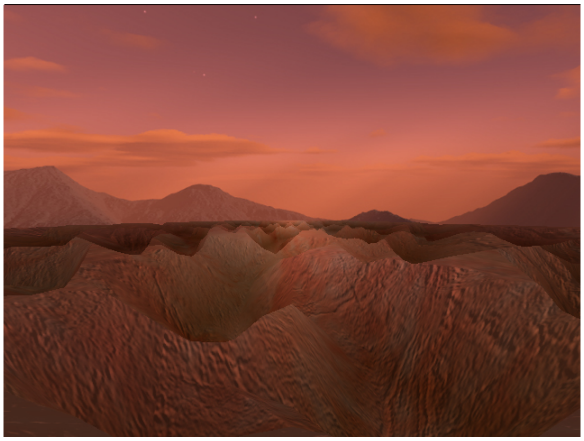

Figure 3: Fractal terrain with MD5GPU. We create fractal noise with MD5GPU and use the noise values as a height map for terrain generation. This allows the generated terrain to scroll infinitely while consuming no memory to store the height map. Our demo runs at 60 frames per second.

This is suitable for a flight simulator where the terrain needs to vary without repetition and the terrain data should take minimal or better no memory. Due to the excellent statistical properties of MD5GPU, repetitions in the terrain will be extremely rare and as mentioned above there is no storage of the terrain data. The size of our terrain is no longer limited by video memory, but by floating point precision. Texture coordinates are given as float point values and can represent integer values accurately up to 2^{24}. Although it is possible to utilize different exponent values of FP32 as well (barring special numbers such as NAN, denorms, and INF), for simplicity in this demo we use only the 1-bit sign and 23-bit mantissa. This means that our implementation can represent a virtual height map of size $2^{24} \times 2^{24}$, which is simply too large for any pre-baked texture height maps. Figure 3 shows an image of our fractal terrain demo and this demo renders a terrain using 70×70 vertices runs at 60 frames per second.

5.2 Texture Tiling

Texture tiling applications (e.g. [Lefebvre and Neyret 2003; Wei 2004b; Lagae and Dutré 2006a]) also use hash functions and random variables to procedurally generate a texture that has no visible pattern repetition. Wei [Wei 2004b] uses a hash function stored as a 2D texture to find the color of Wang Tiles [Cohen et al. 2003] needed for a tiling texture synthesis. The color of an edge is found by iteratively hashing the cell's coordinate several times. However, one major drawback to this method is that it requires many texture accesses per fragment due to hashing and this slowed down the algorithm. By using MD5GPU, we can reduce the number of hashes needed to find the colors for a Wang Tile and remove the bandwidth bottleneck since our hashes do not rely on texture lookups.

In our implementation of texture tiling, we use one digest per edge to preserve edge color consistency. Using MD5GPU, the new formula to find the edge colors would be:

$$\begin{aligned}
C_S &= \texttt{whiteNoise}(vec4(O_h, O_v), key).x\%K_h \\
C_E &= \texttt{whiteNoise}(vec4(O_h + 1, O_v \times 2), key).x\%K_v \\
C_N &= \texttt{whiteNoise}(vec4(O_h, O_v + 1), key).x\%K_h \\
C_W &= \texttt{whiteNoise}(vec4(O_h, O_v \times 2), key).x\%K_v
\end{aligned}$$

where (O_h, O_v) is the output tile index, and (K_h, K_v) is the number of horizontal and vertical colors respectively.

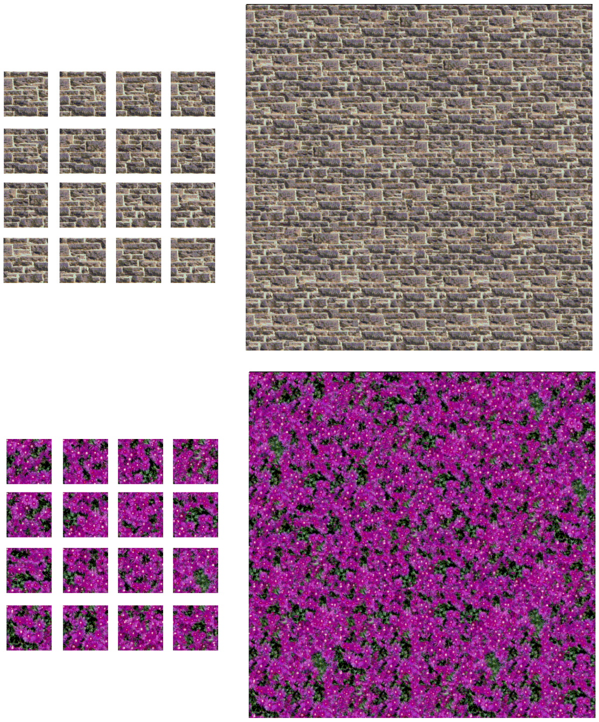

Figure 4: Tiling with MD5GPU. MD5GPU can be used to generate edge colorings for Wang Tiles. Left: the input tile set (4×4 tiles). Right: the tiled output (8×8 tiles). The output is rendered with resolution 512^2 and run at 60 fps.

(C_S, C_E, C_W, C_N) are the edge colors of (O_h, O_v). A similar formula can be used to find the corner colors as well if corner packing is desired. The key can be used as a quick way to vary the resulting texture while keeping the rest of the variables constant.

Figure 4 shows the result of such an application of texture synthesis rendered at 512^2 resolution. The original implementation requires $8 + 1$ texture accesses per fragment (for accessing both input tiles and the hash table) resulting in $512^2 \times 9$ total texture accesses for a frame. Our method only uses up 512^2 texture accesses per frame (access only the input tiles but not the hash table). This reduction allows our implementation to run at 60 fps as opposed to the original method which runs at 30 fps on a GeForce 8800 GTX.

6 Conclusions and Future Work

We have demonstrated that the MD5 hashing algorithm is well suited for parallel random number generation on a GPU, for both quality and performance. Quality-wise, we have performed a battery of tests and have demonstrated that this method passes all tests while some contending methods do not. Performance-wise, our method is fastest among all methods which pass all statistical quality tests.

Our method is particularly beneficial for applications that cannot afford to use low-quality random numbers or pre-compute high-quality random numbers and store the result in texture memory. We have demonstrated one potential application of procedural terrain generation, and there are other possibilities ranging from texturing, rendering, to general purpose computations (GPGPU).

Our current implementation of MD5 as a pixel program might be too slow for certain applications. This can be fixed by implementing MD5 as a hardware unit with usage scenario very similar to a texture unit [Wei 2004a]. As pointed out in [Wei 2004a], MD5 is reasonably cheap to implement in hardware; in addition it can be utilized for on-chip cryptography and security functionalities as well. Another potential future work is to investigate alternative cryptographic hash functions that provide better quality/performance tradeoffs. Although our study selects of MD5, the general methodology certainly carries to other hash functions.

Acknowledgement

We would like to thank Ke Deng for reading an early draft of our paper, George Danezis for recommending block/stream ciphers as a more efficient replacement for hashes, Hsiao-Wuen Hon for help arranging Stanley's internship in MSRA, and anonymous reviewers for their comments.

References

1997. Advanced encryption standard. In *FSE '97: Proceedings of the 4th International Workshop on Fast Software Encryption*, Springer-Verlag, London, UK, 83–87.

ANDERSON, AND BIHAM. 1996. Tiger: A fast new hash function. In *IWFSE: International Workshop on Fast Software Encryption, LNCS*.

BLUM, L., BLUM, M., AND SHUB, M. 1986. A simple unpredictable pseudo random number generator. *SIAM J. Comput. 15*, 2, 364–383.

BRANDS, S., AND GILL, R. 1996. Cryptography, statistics and pseudo-randomness, part ii. *Probability and Mathematical Statistics 16*, 1–17.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM Press, New York, NY, USA, 287–294.

D'AGOSTINO, R. B., AND STEPHENS, M. A., Eds. 1986. *Goodness-of-fit techniques*. Marcel Dekker, Inc., New York, NY, USA.

DOBBERTIN, H., BOSSELAERS, A., AND PRENEEL, B. 1996. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*, 71–82.

EBERT, D. S., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing & Modeling, A Procedural Approach*. AP Professional.

ENTACHER, K., UHL, A., AND WEGENKITTL, S. 1998. Linear and inversive pseudorandom numbers for parallel and distributed simulation. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, IEEE Computer Society, Washington, DC, USA, 90–97.

HOWES, L., AND THOMAS, D. 2007. Efficient random number generation and application using CUDA. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, July, ch. 37, 805–830.

KNUTH, D. 1998. *The Art of Computer Programming*. Addison-Wesley Publishing Company.

LAGAE, A., AND DUTRÉ, P. 2006. An alternative for wang tiles: colored edges versus colored corners. *ACM Trans. Graph. 25*, 4, 1442–1459.

Lagae, A., and Dutré, P. 2006. A comparison of methods for generating Poisson disk distributions. Report CW 459, Department of Computer Science, K.U.Leuven, Leuven, Belgium, August.

L'Ecuyer, P., Simard, R., Chen, E., and Kelton, W., 2001. An object-oriented randomnumber package with many long streams and substreams.

Lefebvre, S., and Hoppe, H. 2005. Parallel controllable texture synthesis. *ACM Trans. Graph. 24*, 3, 777–786.

Lefebvre, S., and Hoppe, H. 2006. Perfect spatial hashing. *ACM Trans. Graph. 25*, 3, 579–588.

Lefebvre, S., and Neyret, F. 2003. Pattern based procedural textures. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, 203–212.

Marsaglia, G., 1995. The marsaglia random number cdrom including the diehard battery of tests of randomness. http://www.stat.fsu.edu/pub/diehard/.

Mascagni, and Srinivasan. 2000. SPRNG: A scalable library for pseudorandom number generation. *ACMTMS: ACM Transactions on Mathematical Software 26*.

Matsumoto, M., and Nishimura, T. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. Model. Comput. Simul. 8*, 1, 3–30.

(NSA), N. S. A., 2001. Us secure hash algorithm 1 (sha1). http://tools.ietf.org/html/rfc3174.

NVIDIA, 2007. Ext_gpu_shader4 opengl extension. `http://developer.download.nvidia.com/opengl/specs/GL_EXT_gpu_shader4.txt`.

NVIDIA, 2007. Nvidia cuda compute unified device architecture. `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`.

Olano, M. 2005. Modified noise for evaluation on graphics hardware. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 105–110.

Pang, W.-M., Wong, T.-T., and Heng, P.-A. 2006. Implementing high-quality prng on gpus. In *Shader X5: Advanced Rendering Techniques*, Charles River Media, Inc., Rockland, MA, USA, 579 – 590.

Patel, M., 2006. The goulburn hashing function. http://www.geocities.com/drone115b/Goulburn06.pdf.

Peachey, D. 1990. Texture on demand. Tech. Rep. technical memo 217, Pixar.

Perlin, K. 1985. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, 287–296.

Perlin, K. 2002. Improving noise. *ACM Transactions on Graphics 21*, 3 (July), 681–682.

Podlozhnyuk, V., 2007. Parallel mersenne twister. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf.

Rivest, R., 1992. Rfc 1321: The md5 message-digest algorithm. Reference implementation: http://www.faqs.org/rfcs/rfc1321.html.

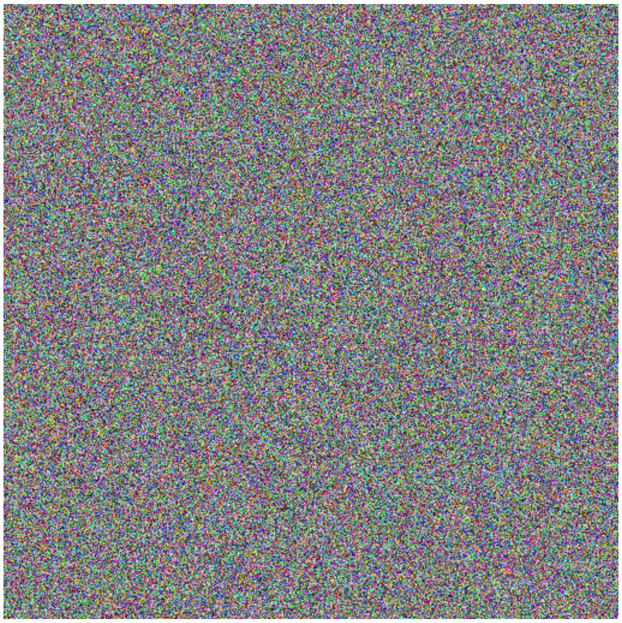

Figure 5: A white noise generated by our approach.

Rost, R. J. 2004. *OpenGL(R) Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Sussman, M., Crutchfield, W., and Papakipos, M. 2006. Pseudorandom number generation on the gpu. In *Graphics Hardware 2006*, 87–94.

The IEEE, and The Open Group. 2004. *The Open Group Base Specifications Issue 6*.

Wang, X., and Yu, H. 2005. How to break md5 and other hash functions. In *EUROCRYPT*, 19–35.

Wayner, P. 2002. *Disappearing Cryptography*. Morgan Kaufmann.

Wei, L.-Y., and Levoy, M. 2002. Order-independent texture synthesis. Tech. Rep. TR-2002-01, Computer Science Department, Stanford University.

Wei, L.-Y., 2004. A system and method of generating white noise for use in graphics and image processing. U.S. Patent Pending No. 10/956,954, filed September 30, 2004 with NVIDIA Corp.

Wei, L.-Y. 2004. Tile-based texture mapping on graphics hardware. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM Press, New York, NY, USA, 55–63.

Yamanouchi, T. 2007. Aes encryption and decryption on the gpu. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, July, ch. 36, 785–803.