High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

GPU Random Numbers via the Tiny Encryption Algorithm

Fahad Zafar, Marc Olano, Aaron Curtis

University of Maryland, Baltimore County

Abstract

Random numbers are extensively used on the GPU. As more computation is ported to the GPU, it can no longer be treated as rendering hardware alone. Random number generators (RNG) are expected to cater general purpose and graphics applications alike. Such diversity adds to expected requirements of a RNG. A good GPU RNG should be able to provide repeatability, random access, multiple independent streams, speed, and random numbers free from detectable statistical bias. A specific application may require some if not all of the above characteristics at one time. In particular, we hypothesize that not all algorithms need the highest-quality random numbers, so a good GPU RNG should provide a speed quality tradeoff that can be tuned for fast low quality or slower high quality random numbers.

We propose that the Tiny Encryption Algorithm satisfies all of the requirements of a good GPU Pseudo Random Number Generator. We compare our technique against previous approaches, and present an evaluation using standard randomness test suites as well as Perlin noise and a Monte-Carlo shadow algorithm. We show that the quality of random number generation directly affects the quality of the noise produced, however, good quality noise can still be produced with a lower quality random number generator.

Categories and Subject Descriptors (according to ACM CCS): G.3 [Probability and Statistics]: Random number generation— I.3.7 [Computer Graphics]: Three-dimensional Graphics and Realism—Color, shading, shadowing, and texture E.2 [Data Encryption]: —

Keywords: cryptographic hash, noise, shadows

1. Introduction

Random numbers have many uses in computer graphics, from Monte-Carlo sampling for realistic image synthesis to noise generation for artistic shader construction. The modern Graphics Processing Unit (GPU) is a high performance programmable parallel processor. While many randomized shading effects can be accomplished through textures, the increasing bias of GPUs towards computation over texture lookups, especially potentially uncorrelated texture lookups, leads to a desire for more computational solutions.

In particular, generation of random numbers on the GPU should not require valuable texture storage and texture bandwidth. Since changes in random sampling can introduce dancing or shimmering artifacts, we need the same random numbers from the RNG given the same seed position in space or on a surface. Additionally we require independent parallel streams of random numbers. For some applications, these streams are per-GPU thread, while for others they are based on either position in space or on the surface, with a single GPU thread potentially pulling numbers from several streams. For spatial streams, several threads may access numbers from the same stream, and should get the same results. The numbers should be statistically independent within each stream and between streams. Multiple calls should also not halt for some shared memory object that maintains the state of the RNG.

This leads to a set of desired properties for GPU random number generation: repeatability to avoid frame-to-frame coherence artifacts, independent random streams and random access within the stream for parallel consistency, random numbers free from noticeable statistical bias, and generator speed. Like other authors, we find a cryptographic function satisfies all of these requirements [TW08]. By running a seed and possible sequence number through the cryptographic function, we get repeatable, randomly accessible parallel streams of random numbers. One of the primary

contributions of this paper is identifying the Tiny Encryption Algorithm (TEA) [WN94] as a particularly good fit for GPU random number generation. TEA uses only simple operations that are fast on modern GPUs and has minimal state and internal data tables. Additionally, We can vary the number of TEA rounds to provide a speed/quality tradeoff.

We evaluate the statistical quality of TEA as a random number generator for varying number of rounds using the DIEHARD and NIST randomness test suites [Mar95, RSN*08]. We show that TEA is faster for equivalent quality than previous high-quality alternatives such as MD5 [TW08, Riv92a] as well as previous fast but low-quality alternatives like modified Blum-Blum-Shub(BBS) [Ola05, BBS86].In addition, we observe that many visual applications of random numbers do not exhibit visible artifacts, even with a generator that does not pass all of the NIST or DIEHARD tests.

A second contribution of this paper is the idea that fewer rounds of TEA are sufficient in many visual applications, supported by two examples. The first example is an improved Perlin noise [Per02] with modifications by Kensler et al. [KKS08]. We compare both the spatial appearance of the noise and its frequency spectrum for differing rounds of TEA, and against several reference Perlin-like noise functions. The second example is a Monte-Carlo soft shadow algorithm that blurs the sample results across a neighborhood of several pixels [Cur09]. We compare shadowing results with this algorithm for differing numbers of TEA rounds.

2. Related Work

2.1. Random Number Generation

The most common CPU pseudo-random number generators are based on a recurrence with a small amount of state and a relation that transforms the state and produces a new random number each iteration [Knu97]. These include the fast Linear Congruential Generator (LCG) [Knu97] and the high-quality Mersenne Twister [MN98]. Many algorithms avoid pseudo-random number generation on the GPU by precomputing a set of numbers on the CPU and storing them in a GPU buffer or texture. This approach allows random access, but uses valuable texture resources and limits the length of the random number sequence and number of streams due to storage limitations. Pang et al. [PWH06] compare classic PRNG implementations in shaders, though these methods do not provide random access. There are implementations available for PRNGs on the NVIDIA [NVI07] CUDA architecture for General Purpose GPU techniques, but they focus more on generating random numbers in bulk, so do not allow random access to the stream.

Lagae et al. [LLDD09] use an LCG, with separate seeds to provide independent spatial streams. This was surprisingly effective, despite their use of sequential seeds and the fact that the first several numbers from an LCG have almost a linear relationship with seed value. The success was likely due to their discarding hundreds of initial numbers, moving well past the less random startup region. Also, a variable number of pseudo-random numbers were consumed from the beginning of the stream to generate a single Poison distributed random number, which has less visual impact on the result than later numbers and allows nearby streams to get out of step.

A couple of recent methods have used a cryptographic hash as the basis of a random number generator [Ola05, TW08]. Olano [Ola05] used a variation of BBS PRNG in order to create purely computational noise. His variation changed the modulus of the generator from the product of two large primes to a single small prime, to be computable in a 16-bit half float. This results in a fast generator, but as Tzeng and Wei [TW08] note, not a particularly high quality one. Tzeng and Wei used the MD5 hash as a PRNG, confirming its randomness through the DIEHARD test suite [Mar95].

2.2. Applications

Monte-Carlo methods are a class of computational algorithms that use random sampling to estimate a function that would otherwise be infeasible or too computationally expensive to solve. We refer the reader to Keller et al. [KHN06] for more in depth coverage. Monte-Carlo methods are popular for solving light transport problems [CPC84, VG97, Jen96]. Many Monte-Carlo algorithms are particularly sensitive to correlation amongst generated random numbers. To avoid artifacts, no correlations or clumps should exist in the pair sampled graph of those values. Using a series of numbers that possess a correlation or are not statistically random will negatively affect results.

One of the most important uses of random numbers in computer graphics is noise. The original gradient noise proposed by Perlin required lookups for a spatial hash and permutation table values [Per85, EMP*98]. The many alternatives to Perlin's original noise function can be roughly divided into those that compute large textures in an offline process [CD05, GZD08], those that use a different algorithm to generate a random function with a similar donut-shaped band-limited frequency spectrum [Lew89, Per04, LLDD09], and those that improve Perlin's original algorithm [Per02, Gre05, Ola05, TW08, KKS08]. The first class uses valuable texture resources, presents inherent limits on the size before the noise repeats, and has natural limitations when extending to 3D and 4D noise variations. The second class allows computational run-time evaluation and usually less frequency bleeding than Perlin's original noise, but without the speed and simplicity of the original. We focus on the last class.

Perlin's original noise defines an integer lattice over the N-D input space (commonly 1D, 2D, 3D and 4D). Each in-

teger point is hashed to a random gradient vector using nesting lookups into a 1D hash table followed by a lookup into a table of random vectors. Each cell is uniquely determined by its integer corners to give a smooth piecewise polynomial function with the given gradients and a value of zero at the cell corners [Per85]. Perlin's Improved Noise used a higher order polynomial interpolant and avoided the final lookup by using bits of the hash to define a limited set of gradients [Per04]. Green created a straightforward GPU implementation, collapsing each pair of 1D lookups into a 2D texture lookup [Gre05]. Olano's Modified Noise removed all GPU texture lookups by using a PNRG seeded with each integer coordinate as the hash [Ola05]. Tzeng and Wei [TW08] used MD5 as their hash, providing better randomness and avoiding axis-aligned artifacts caused by chaining 1D hash functions to create an N-D hash. We are inspired by their work to explore cheaper cryptographic hash functions as alternatives to MD5. Finally, Kenseler et al. [KKS08] use a larger neighborhood around each cell to improve the frequency discrimination.

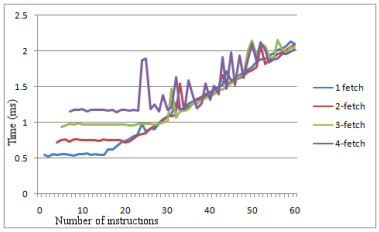
3. Pseudo-Random Number Generator Selection

Our goal is to find a pseudo-random number generator that is sufficiently random and free of statistical artifacts, repeatable given the same seed, and allows parallel streams of numbers with good randomness between streams as well as within each stream. In addition, for applications like Perlin noise, we would like to be able to get unique random numbers based on an N-D position, either through random access to numbers within a single stream or easy reseeding without start-up artifacts that affect some generators like LCG.

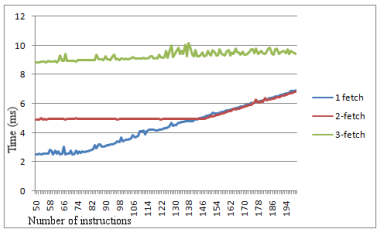
One of the standard analyses in GPUbench [HJK*04] shows how many MAD (Fused Multiply and Add) instructions are required before a shader with specific number of texture fetches becomes compute bound (Figure 1). Of course, results differ on different GPUs, but we can easily see that a computational low-cost PRNG is preferable to using texture accesses, especially since those accesses are often likely to be incoherent and unable to take full advantage of hardware texture caching.

3.1. Cryptographic Functions as a Hash

Cryptographic functions are designed to turn a coherent plaintext signal into an incoherent encrypted form. Ideally, the encrypted form should look as much as possible like random noise, without any identifiable statistical characteristic that might be used to tell something about the original message. This is the core feature that allows cryptographic functions to be used for random number generation. Yet cryptographic functions serve the purpose of encryption while our goal is randomness with as little computational cost possible. For example, MD5 uses four different types of rounds each of which are executed 16 times [Riv92a]. The *S-Boxes*



(a) Sequential texture access pattern



(b) Random texture access pattern

Figure 1: *GPUbench shader analysis on an NVIDIA 8600M GT to identify how many instructions before a shader becomes compute bound.*

continually mix the input bits to remove any correlation between them to make it hard for someone to break its cipher. These measures add to the security of the algorithm, making it more difficult for a malicious attacker to generate a different plain text to produce a predetermined MD5 result, but also add to the complexity of the algorithm.

Other candidates incuded RC4 [Riv92b], which requires a lookup table; CAST-128 [Ada97], which requires a large amount of data for initialization; Blowfish [Sch94], which is fast but each new key requires pre-processing equivalent to encrypting nearly 4 kilobytes of data. A short survey of some other cryptographic hash functions is presented by Tzeng and Wei [TW08]. The authors justify their decision for selecting MD5 on the basis of their requirements which are similar to ours.

We found the Tiny Encryption Algorithm (TEA) and its extension XTEA to be ideal for our purposes [WN94, Red03]. TEA was designed for fast execution and minimal memory footprint. Its use for encryption is limited due to its security holes and weaknesses in the output cipher, but it is a simple algorithm that provide randomness comparable to any good RNG. This property of TEA has been discussed by Reddy [Red03] finding that the *Avalanche Effect* exists at just 6 rounds, causing a drastic change to the output when any one bit of the input is changed.

3.2. TEA, XTEA and XXTEA

TEA is a Feistel type cipher. A Feistel cipher is a symmetric iterative structure used in block ciphers. A dual shift in a sin-

```
UInt32[2] encrypt( UInt32 v[2] ) {
  UInt32 k[4], sum=0, delta=0x9e3779b9
  k={A341316C, C8013EA4, AD90777D, 7E95761E}

  for $N$ rounds
    sum += delta
    v0 += ((v1 << 4)+k0) ^ (v1+sum) ^ ((v1 >> 5)+k1)
    v1 += ((v0 << 4)+k2) ^ (v0+sum) ^ ((v0 >> 5)+k3)

  return v
}
```

Figure 2: *Pseudo-code for N rounds of TEA. Typically the loop would be unrolled for the desired N. Input a seed and sequence number to use as a random number generator.*

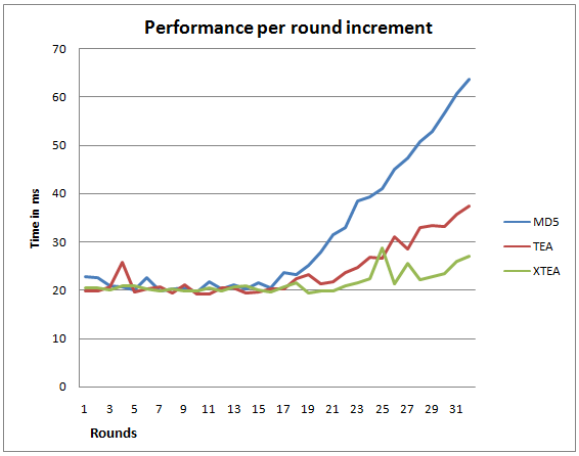


Figure 3: *Frame time (in ms) for a shader with one RNG call, with increasing number of rounds. Resolution: 1440 × 900, Triangles: 220,000, GPU: NVIDIA 9800GT*

gle round allows the data and key to be mixed continuously per round. After weaknesses were found in the original TEA, it was extended to XTEA and XXTEA [Red03]. In XTEA, the key scheduling is modified to slowly introduce the key while the shifts, XORs, and additions are also rearranged.

XXTEA is more efficient for encrypting longer messages. It operates on variable-length blocks that are some arbitrary multiple of 32 bits in size (minimum 64 bits). The total number of cycles depends on the block size. XXTEA uses a more involved round function which makes use of both immediate neighbors in encrypting each word in the block. Due to the increased complexity and conditionals in the XXTEA function, we do not consider it further here. For more detailed analysis, refer to Zafar [Zaf10].

We implemented our TEA random number generator in CUDA [NVI07], and in shaders using GLSL and HLSL [Ros04]. We propose TEA for graphics as well as GPGPU applications. The CUDA implementation helps gauge performance for a threaded algorithm regardless of rendering. The shader implementations are used to test the PRNG for graphical use. Performance analysis is conducted for the shaders invoking the noise function or making direct calls to the PRNG from within the shaders. The TEA port to either languages is simple and straightforward. No specialized API functions are required that might need implementing before use. Any language that supports bitwise operations will execute the TEA algorithm with ease.

We restrict the input for TEA and XTEA to a fixed 64-bit block, consisting of two 32-bit integers. Pseudo-code for the TEA generator is shown in Figure 2. The constant delta is a binary representation of the golden ratio, as specified by the original TEA algorithm. The key is one suggested by Reddy [Red03]. Other choice of key could work, though the analyses would need to be re-run. In the text, we will indicate the number of rounds as TEA_N for N rounds.

4. TEA as a Random Number Generator

Random numbers used in numerical analysis need to be unbiased for good results. This factor determines the type of PRNG that should be used for specific applications. We judge quality of numbers using the NIST randomness tests and the DIEHARD tests.

Random numbers were generated for the DIEHARD tests by passing the range of integers 1 - 67108889 for the first 32-bit input and fixing the remaining bits to 0. The encrypted numbers resulted in a 512 MB dataset for TEA functions. The NIST dataset was created by passing the whole range of 1-10485670. The resulting input for the NIST tests is around 80 MB. The DIEHARD dataset is substantially larger in order to cater to the minimum size requirement for some tests in the new DIEHARD test suite.

We compare against MD5 and the modified BBS generator. The MD5 in our experiments is the optimized *MD5GPU* presented by Tzeng and Wei [TW08]. Unlike the typical CPU implementation, this version does not store the array of sine values, but instead calculates them using the GPU *sin*() function, thus the rotational storage is reduced to 16 unique numbers. All rounds for all hash functions used were unrolled and inlined for optimal performance, and to present a fair comparison of results.

Both the NIST and DIEHARD test suites include a combination of statistical tests on the number stream and random processes with known expected results. Since these are statistical tests of a random stream, success or failure is not a binary result, but is provided in the form of a p-value for each test or subtest, indicating the probability that the result is due to random chance. We use a significance level of 0.01. As discussed by Marsaglia [Mar95], some tests produce mul-

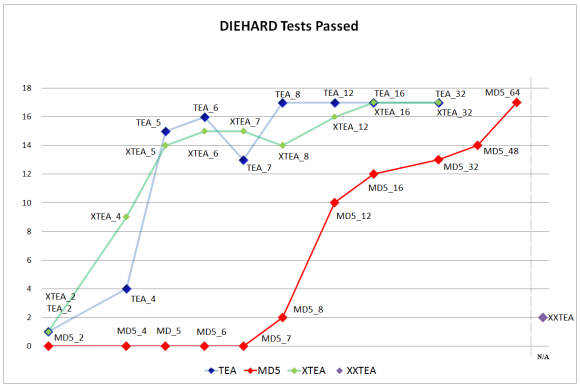


Figure 4: *DIEHARD randomness test results*

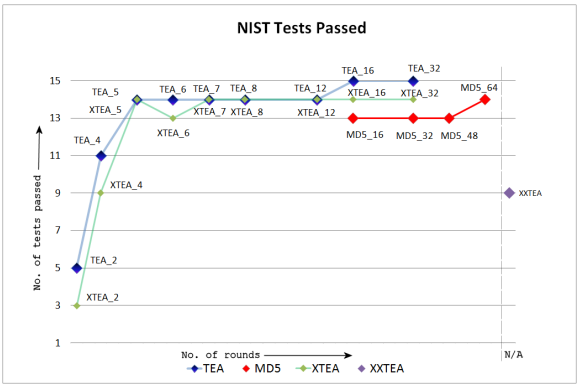
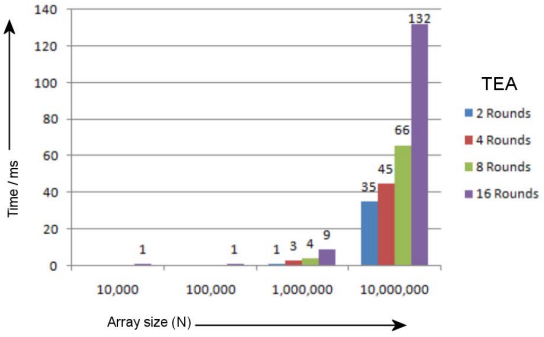


Figure 5: *NIST randomness test results*

tiple results, and may still generate one or two failing p-values due to random chance. For example, the DIEHARD Bit Stream Test produces a p-value for each individual range of bits. Marsaglia suggests considering such a test to have passed if fewer than six p-values fail. We adopt a stricter criteria: if a test outputs ten or fewer p-values, we consider a test failed if any single p-value is out of range; if the test outputs more than ten p-values, we accept one out-of-range p-value, but consider any second out-of-range p-value a failure. Tzeng and Wei [TW08] use the Kolmogorov-Smirnov test (KS-test) [DS86] in their analysis to resolve multiple conflicting p-values. This criteria may pass some tests that we consider to fail, but if sufficient for an application, may allow even fewer rounds of TEA than we recommend.

The NIST randomness tests contain a set of 15 tests while the new version of DIEHARD tests contains 17 tests. One of our goals was to show that TEA and its extensions can be used in place of MD5 and use less GPU time. Therefore, we need to find the minimum number of rounds required for TEA and XTEA to produce the best results so we are not wasting GPU clock cycles on extra work. The results (Figure 4 and Figure 5) show that TEA_8 is one of the best options as a cryptographic hash function for generating fast and extremely high quality random numbers. The results Figure 3 show how the workload increases when MD5 is used as a general purpose PRNG when rendering. TEA however performs much better even with a higher number of rounds and multiple calls. With fewer than eight rounds, TEA produces numbers less random than $MD5_{64}$ (Figure 4). Since the avalanche effect for TEA starts at six rounds [Red03], fewer than six rounds are not expected to produce numbers that are random in all bits.




```
Threads per Block : 256
Number of Blocks : (N + ThreadsperBlock - 1) / ThreadsperBlock
Input size : 64 bit
Output size : 64 bit
```

Figure 6: *Time (in milliseconds) to complete all threads on NVIDIA 8600M. Some intervals were too short for the counter*

A simple and tunable PRNG can find many uses in GPGPU applications. We benchmarked our CUDA implementation of TEA to show how many random numbers can be produced per second. In our tests, an output array element consists of 64 random bits. One CUDA kernel executes for an entire array and generates random numbers using the TEA algorithm with a 64-bit input. We provide timing results for an NVIDIA 8600M in Figure 6. These results show that we can generate roughly 400 million TEA_2 random numbers per second, or 151 million TEA_8 random numbers per second, even on this mobile GPU. We observe that sixteen calls to TEA_1 is the same computation as one call to TEA_{16} (not counting the code between TEA calls), so a more predictive statistic is that this mobile GPU can run approximately 1.2 billion TEA rounds per second. A similar test is conducted from within a shader when rendering at a resolution of 1440 $\times$ 900 (Figure 7).

5. TEA for Perlin Noise

The Perlin noise function is a common primitive in procedural shaders. Figure 8 shows several examples of typical use. The noise function is designed to be deterministic, fre-

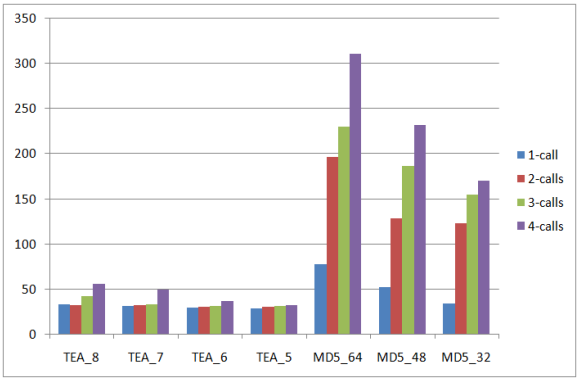


Figure 7: *Frame time (in ms) to render one frame with different number of calls to PRNGs from within a shader. Resolution: 1024 × 800, Triangles: 220,000, GPU: ATI 3870 X2*

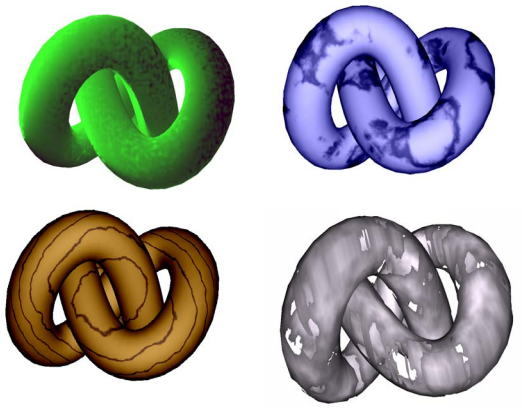


Figure 8: *Four GPU shaders using TEA_2-based noise (bump, marble, wood, erosion)*

quency band limited, and isotropic. It is deterministic to ensure the same input always produces the same result and noise-generated surface features will not change from frame to frame. It is band limited to one octave (power of two in frequency) to allow tailoring a desired frequency spectrum by weighted sum of noise functions at different scales. Finally, it is isotropic to avoid directional artifacts, though recent noise functions have allowed controlled anisotropy that can be summed to an isotropic noise [GZD08, LLDD09].

Several recent GPU variations of Perlin noise use a cryptographic hash to select the random gradient vectors at the corners of each integer cell without the lookup tables present in the original algorithm [Ola05, TW08]. Our test code is based on the online reference implementation by Olano. We use Olano's code for the modified BBS hash, and Tzeng and Wei's code for the MD5 hash.

Given the success of even poor random number generators at making reasonable noise [Ola05, LLDD09], we suggest that the random number quality can be tailored to the application. In particular, that the number of rounds might be reduced without hurting the quality of the Perlin noise.

	2-D Noise		*4 × 4 Kernel*	
RNG	**HD 4870**	**8600M**	**HD 4870**	**8600M**
BBS	3077	1628		
$MD5_{64}$	173	61	27	10
TEA_8	1000	322	281	81
TEA_2	3243	1305	1000	364
$XTEA_{16}$	635	184	187	40
$XTEA_2$	3529	1551	1143	419

Table 1: *2D Perlin Noise speed in MPixels/second for AMD Radeon HD 4870 and NVIDIA 8600M GT.*

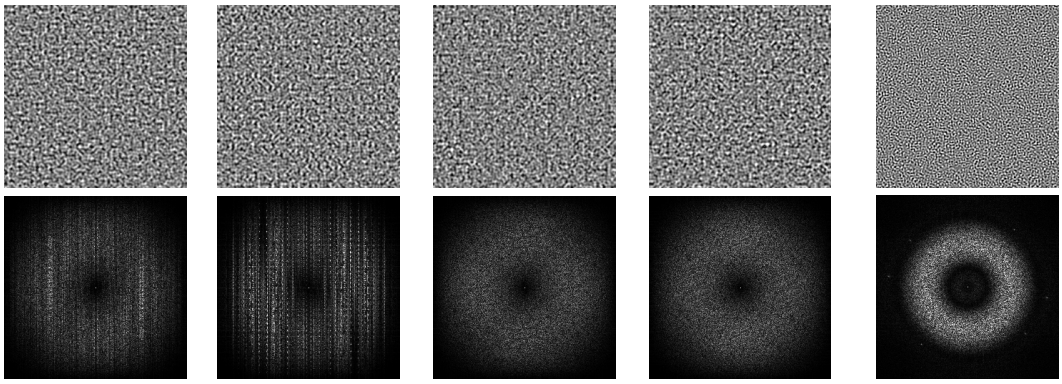
5.1. 2D Noise

Table 1 gives timing data for 2D noise with a range of hash-based random number generators. We ran our speed tests using the AMD GPU Shader Analyzer on an AMD Radeon HD 4870 and the NVIDIA Shader Perf tool on an NVIDIA 8600M GT .The results show that all TEA and XTEA (2D and 3D) noise implementations are faster than equivalent $MD5_{64}$, with TEA_2 and $XTEA_2$ producing the fastest results, faster even than Olano's BBS-based modified noise [Ola05]. Figure 3 shows how TEA, XTEA and MD5 perform with increasing number of rounds. MD5 is most expensive per round. election of either TEA or its extension (XTEA) depends on the application. XTEA is the fastest function to use for graphics noise. With this implementation a 4 kernel noise implementation can be used with maximum performance benefits. TEA can be used as a fast all purpose RNG and can be used for noise as well as random number generation with just 8 rounds.

Figure 9 shows the spatial appearance and frequency spectrum for a representative subset of tested noise algorithms. The spatial appearance should consist of random blobs of uniform size, free from directional and clumping artifacts. The frequency plot should be a well defined donut shape, with a ring of uniform white noise and little energy inside or outside the ring. Figure 10 shows the effect of varying the number of rounds in frequency space for MD5 and XTEA. Though six rounds are needed for the avalanche effect, the nearby integer grid coordinates differ in the bottom bits, and we only use the bottom two of the generated bits, a couple of rounds are sufficient when using TEA and XTEA. MD5, however, exhibits serious artifacts and only becomes usable for noise starting at about 6 rounds.

Both Perlin noise [Per02] and the modified noise [Ola05] use Perlin's hash nesting to construct a 2D hash from a 1D one:

$$hash2D(P_i) = hash1D(hash1D(x) + y). \quad (1)$$



(a) Perlin [Per02] (b) BBS [Ola05] (c) $MD5_{64}$ [TW08] (d) TEA_2 (e) 4×4 Kernel TEA_2

Figure 9: *Perlin-style noise and its Fourier transform for selected PRNG choices*

RNG	HD 4870	8600M
$MD5_{64}$	55	30
TEA_8	359	102
TEA_2	1250	481
$XTEA_{16}$	235	40
$XTEA_2$	1412	604

Table 2: *3D Perlin Noise Speed in MPixels/second for AMD Radeon HD 4870 and NVIDIA 8600M GT*

This causes the vertical stripe artifacts in the frequency spectra for these two cases and directional artifacts in the noise images. The other hash functions can take two (or more) coordinates directly. We use the x and y integer coordinates for two of the hash inputs, avoiding these directional problems. We also show results using a 4×4 neighborhood around each cell, as described by Kensler et al. [KKS08]. This requires four times as many calls to TEA than the original 2×2 neighborhood, but gives a better defined frequency profile.

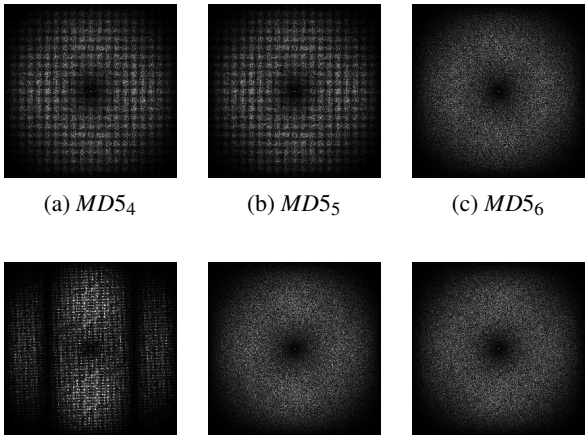
5.2. 3D Noise

3D noise uses a 2×2×2 neighborhood around each integer cell. Since our single 64-bit block TEA takes only two inputs, we use a nested hash as in Equation (1). The MD5 hash takes a four-component vector as input. Thus, we need only 8 calls to the MD5 hash as compared to 12 for TEA. Even so, Table 2 shows that TEA significantly outperforms MD5.

6. TEA for Monte-Carlo Shadowing

Curtis [Cur09] describes a method for rendering high-quality soft shadows in real time using Monte Carlo ray tracing on the GPU. As in methods derived from backprojection [GBP06], a shadow map is used as a discretized representation of all the occluding geometry in a scene. When projected into world space, each texel in the shadow map represents a patch of micro-geometry that a shadow ray might intersect. In this method, the Z-buffer, plus front and back surfaces from the light are combined to give a more accurate approximation to the occluding geometry. For each point to be shaded, an occlusion test is performed by tracing a ray through these depth maps to a randomly sampled point on the surface of an area light source. Figures 11a-c shows an intermediate result in the algorithm, directly after the occlusion test.

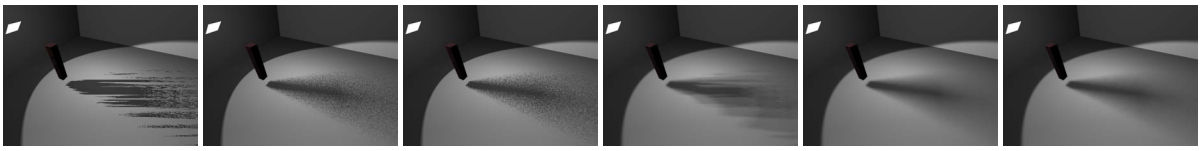
To achieve realistic shadowing, the algorithm adds a variance reduction stage, in which a depth-sensitive Gaussian filter is applied to the occlusion test results in screen space. This effectively hides the randomization, though in scenes



(a) $MD5_4$ (b) $MD5_5$ (c) $MD5_6$

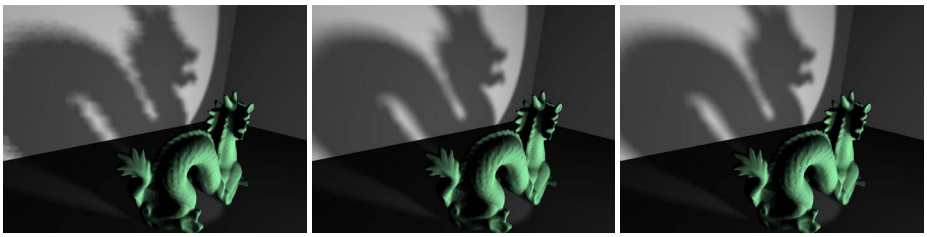
(d) $XTEA_1$ (e) $XTEA_2$ (f) $XTEA_3$

Figure 10: *Fourier Transforms for MD5 and XTEA Noise with increasing number of rounds. Notice that MD5 is usable for noise with at least 6 rounds. XTEA however can be used for noise with just 2 rounds. TEA results are similar to XTEA.*



(a) TEA_1 1 ray, no filtering | (b) TEA_2 1 ray, no filtering | (c) TEA_4 1 ray, no filtering | (d) TEA_1 9 rays, filtered | (e) TEA_2 9 rays, filtered | (f) TEA_4 9 rays, filtered

Figure 11: *Monte-Carlo shadow algorithm, (a-c) one ray per pixel; (d-f) nine rays per pixel with final screen-space filtering*



(a) TEA_1 (b) TEA_2 (c) TEA_4

Figure 12: *Monte-Carlo shadow algorithm on the dragon model*

with wide penumbrae, it is also necessary to trace multiple rays per pixel (4 or 9 are generally sufficient). The shadow algorithms is a perfect example to demonstrate the visual quality and speed tradeoff advantage offered by TEA. Reducing the number of rounds for the PRNG has a direct impact on the visual output of the shadow algorithm. Figures 11d-f shows the final result.For this algorithm, we once again observe that fewer rounds are sufficient for good visual result. TEA-2 produces a noticeable pattern in the results prior to filtering, though it is largely hidden in the final image. With more rounds TEA yields results that are nearly indistinguishable. Figure 12 shows results on a more complex model. The algorithm can be tuned for performance by simply adjusting the number of TEA rounds employed.

7. Conclusion

We have shown that TEA provides an effective GPU random number solution with easy tradeoff between speed and random number quality. Further, we show that lower quality random numbers can be used to generate a noise function, while still allowing higher quality random number generation for GPGPU applications. Based on DIEHARD and NIST test results, as well as noise Fourier plots, we conclude that TEA with eight rounds is faster than MD5 for producing equivalent high-quality random numbers, and that XTEA with two rounds can be used in a fast and artifact-free gradient noise.

ACKNOWLEDGEMENTS

We would like to thank the reviewers for their helpful comments. We would also like to thank the Stanford Computer Graphics Laboratory for the dragon model, and Tzeng and Wei for their GPU MD5 code. Finally, we would like to thank AMD and NVIDIA for GPU donations.

References

[Ada97] ADAMS C.: *The CAST-128 Encryption Algorithm*. Tech. rep., RFC Editor, United States, 1997. 3

[BBS86] BLUM L., BLUM M., SHUB M.: A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing 15*, 2 (May 1986), 364–383. 2

[CD05] COOK R. L., DEROSE T.: Wavelet noise. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM SIGGRAPH, ACM, pp. 803–811. 2

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM SIGGRAPH, ACM, pp. 137–145. 2

[Cur09] CURTIS A.: *Real-time Soft Shadows on the GPU via Monte Carlo Sampling*. Master's thesis, University of Maryland, Baltimore County, 2009. 2, 7

[DS86] D'AGOSTINO R. B., STEPHENS M. A.: *Goodness of fit techniques*. CRC Press, 1986. 5

[EMP*98] EBERT D. S., MUSGRAVE K., PEACEY D., PERLIN K., ANDWORLEY S.: *Texturing and Modeling, A Procedural Approach*. Morgan Kaufmann, 1998. 2

[GBP06] GUENNEBAUD G., BARTHE L., PAULIN M.: Real-time soft shadow mapping by backprojection. In *Eurographics Symposium on Rendering (EGSR), Nicosia, Cyprus, 26/06/2006-28/06/2006* (http://www.eg.org/, 2006), Eurographics, pp. 227–234. 7

[Gre05] GREEN S.: Implementing improved Perlin noise. In *GPU Gems 2*, Pharr M., (Ed.). Addison-Wesley, 2005, ch. 26. 2, 3

[GZD08] GOLDBERG A., ZWICKER M., DURAND F.: Anisotropic noise. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–8. 2, 6

[HJK*04] HANRAHAN P., JEREMY S., KAYVON F., HOUSTON M., FOLEY T., HORN D.: Gpu bench, 2004. ACM Workshop on General Purpose Computing on Graphics Processors. `graphics.stanford.edu/projects/gpubench`. 3

[Jen96] JENSEN H. W.: Realistic image synthesis using photon mapping. In *Proceedings of the Eurographics workshop on Rendering Techniques* (1996), pp. 21–30. 2

[KHN06] KELLER A., HEINRICH S., NIEDERREITER H.: *Monte Carlo and Quasi-Monte Carlo Methods*. Springer-Verlag, 2006. 2

[KKS08] KENSLER A., KNOLL A., SHIRLEY P.: *Better Gradient Noise*. Tech. Rep. UUSCI-2008-001, SCI Institute, 2008. 2, 3, 7

[Knu97] KNUTH D. E.: *The Art of Computer Programming*, third ed., vol. 2. Addison-Wesley, 1997. 2

[Lew89] LEWIS J. P.: Algorithms for solid noise synthesis. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), ACM Press, pp. 263–270. 2

[LLDD09] LAGAE A., LEFEBVRE S., DRETTAKIS G., DUTRÉ P.: Procedural noise using sparse gabor convolution. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers* (New York, NY, USA, 2009), ACM, pp. 1–10. 2, 6

[Mar95] MARSAGLIA G.: The MARSAGLIA random number cdrom including the DIEHARD battery of tests of randomness v0.2 beta, 1995. `http://i.cs.hku.hk/ diehard/`. 2, 4

[MN98] MATSUMOTO M., NISHIMURA T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8*, 1 (1998), 3–30. 2

[NVI07] NVIDIA: NVIDIA CUDA compute unified device architecture, 2007. `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`. 2, 4

[Ola05] OLANO M.: Modified noise for evaluation on graphics hardware. In *GH '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (New York, NY, USA, 2005), ACM, pp. 105–110. 2, 3, 6, 7

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), ACM, pp. 287–296. 2, 3

[Per02] PERLIN K.: Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM, pp. 681–682. 2, 6, 7

[Per04] PERLIN K.: Implementing improved perlin noise. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, ch. 5. 2, 3

[PWH06] PANG W. M., WONG T. T., HENG P. A.: *ShaderX5: Advanced Rendering Techniques*. Charles River Media, 2006, ch. 9.8: Implementing High-Quality PRNG on GPU. 2

[Red03] REDDY V.: *A cryptanalysis of the Tiny Encryption Algorithm*. Master's thesis, University of Alabama, 2003. 3, 4, 5

[Riv92a] RIVEST R.: *The MD5 Message-Digest Algorithm*. Tech. rep., RFC Editor, 1992. 2, 3

[Riv92b] RIVEST R. L.: *The RC4 Encryption Algorithm*. Tech. rep., RSA Data Security, Inc., March 1992. 3

[Ros04] ROST R. J.: *The OpenGL Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. 4

[RSN*08] RUKHIN A., SOTO J., NECHVATAL J., SMID M., BARKER E., LEIGH S., LEVENSON M., VANGEL M., BANKS D., HECKERT A., DRAY J., VO S.: *A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications*. Tech. rep., NIST Special Publication, August 2008. 2

[Sch94] SCHNEIER B.: Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption, Cambridge Security Workshop* (London, UK, 1994), Springer-Verlag, pp. 191–204. 3

[TW08] TZENG S., WEI L.-Y.: Parallel white noise generation on a GPU via cryptographic hash. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2008), ACM, pp. 79–87. 1, 2, 3, 4, 5, 6, 7

[VG97] VEACH E., GUIBAS L. J.: Metropolis light transport. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 65–76. 2

[WN94] WHEELER D. J., NEEDHAM R. M.: TEA, a tiny encryption algorithm. In *Lecture Notes in Computer Science. Fast Software Encryption: Second International Workshop* (1994), pp. 363–366. 2, 3

[Zaf10] ZAFAR F.: *Tiny Encryption Algorithm for Cryptographic Gradient Noise*. Master's thesis, University of Maryland, Baltimore County, 2010. 4