Tiny Encryption Algorithm for Parallel Random Numbers on the GPU

Fahad Zafar* Marc Olano†

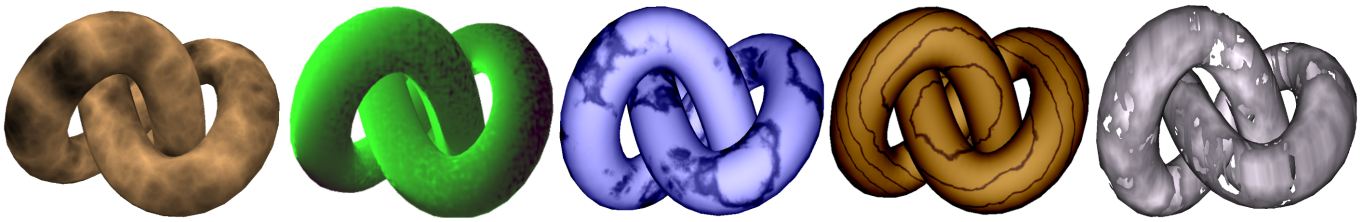

Figure 1: Using Tiny Encryption Algorithm Noise to generate procedural granite, bumps, marble, wood and erosion (*Left to Right*).

Abstract

Random numbers have many uses in computer graphics, from Monte Carlo sampling for realistic image synthesis to noise generation for artistic shader construction. The random number generators used in GPUs today mostly produce poor quality random numbers and run slowly due to their sequential nature, hence they find only limited use for numerical simulations and synthetic noise generation. The ones that do produce acceptable results do so at high execution cost.

In this paper we present some efficient cryptographic hash functions that produce high quality random numbers, map well to the GPU architectures and are easy to implement. We compare results for a variety of hash functions based on speed and quality. We also show how these hash functions can be used with gradient noise. This noise is entirely computational and does not require any texture lookups. Our hash function allows a speed/quality trade off, and we show that faster versions can be used for noise generation, resulting in a noise that is efficient, does not suffer from discernible periodicity and produces excellent results.

Keywords: noise, random number generators

1 Introduction

Modern graphics cards are high performance programmable parallel processors. A shader is a set of flexible software instructions used to program the rendering pipeline. Texture mapping is a widely used technique used in shaders to apply detail or color to a surface using images generated procedurally or stored in memory. When using stored images for texture mapping, a *texture lookup* takes place in order to find out where on the texture each pixel's center falls. A texture fetch halts the shader execution streams and causes delays in the output. Some of these delays can be overcome by allowing independent instructions to execute while texture data is being fetched. The best solution is to remove unnecessary texture fetches from a shader if at all possible. Procedural texturing is the process of generating a texture on the fly, rather than getting it from a memory location.

Random numbers are used in procedural texturing, simulation and numerical analysis, being an important part of GPU applications. In particular, we believe the generation of random numbers on the GPU should not require valuable texture storage and texture bandwidth. Multiple calls should not halt for some shared memory object that maintains the state of the random number generator (RNG). Hence, the best GPU random number generator would be one that is totally computational and would be able to produce good quality random numbers in parallel independently to successive calls.

Our context of parallel random number generation does not imply that we generate numbers in bulk. We do not require a large set of parallel numbers for one pixel, we just require that the algorithm does not have a state, so that multiple pixels can call the function simultaneously.

In our analysis of the Tiny Encryption Algorithm (TEA), we show how it compares to other algorithms in terms of speed and quality. Analysis results are provided on NVIDIA and AMD hardware. Quality comparisons are done using DIEHARD and NIST randomness test suites. Fast Fourier Transforms are used to analyse the band limitation and random distribution of the noise.

2 Related Work

Previous methods for accessing random numbers on the GPU normally involve using a series of random numbers stored in memory. The numbers are generated offline using sequential pseudo random number generators (PRNG). Pang et al. [2006] talk about PRNG implementations in shaders. These methods do not provide random access to the random numbers and use memory locations for their state management. There are implementations available for PRNGs on the NVIDIA [2007] CUDA architecture for General Purpose GPU techniques, but they cannot be used in shaders for computer graphics. Olano [2005] used the Blum Blum Shub PRNG in order to create purely computational noise. The modified noise was fast though the quality of the random numbers was not sufficient for any numerical analysis input as reported by Tzeng and Wei [2008]. They talk about many PRNGs that fail to port onto the GPU architecture with sufficient performance benefits and use the MD5 hash as their PRNG.

Random numbers are used in numerical analysis to help generate unbiased output. Monte Carlo methods are a class of computational algorithms that apply repeated random sampling to generate their results. These methods are used to simulate physical and mathematical systems which have many degrees of freedom, examples might include: fluids, cellular structures, strongly coupled solids and solving some classes of mathematical equations. These methods are used when deterministic algorithms are infeasible or are unable to compute exact results. Monte Carlo methods rely on repeated computation of random or pseudo-random numbers for their execution. Keller et al. [2006] talk about many such problems that can be addressed using this method in their book. Cook et al.

*e-mail: fahad3@umbc.edu
†e-mail: olano@umbc.edu

[1984] used the Monte Carlo method for image rendering in distributed ray tracing. Veach and Guibas [1997] show how the Monte Carlo method can be used to solve the Light Transport problem.

All applications of random sampling require statistically random numbers. These random numbers should not have any correlation amongst generated values and the pair sampled graph of the random numbers should not have any clumps. Using a series of numbers that posses a correlation or are not statistically random will negatively affect results when used with in numerical analysis.

One of the most important uses of random numbers in computer graphics is noise. Perlin [1985] introduced the image synthesizer that could better mimic different real world textures using the gradient method for noise generation. The original gradient noise proposed by Perlin [1985] required a texture lookup for the permutation table values. Anther noise algorithm includes Wavelet Noise by Cook and Derose [2005] which uses the wavelet theory to create noise with less aliasing. They precompute a tile of noise coefficients by filling the tile with random noise, then downsampling, upsampling, and subtracting. Anisotropic Noise by Alexander et al. [2008] uses a blend of directional noise stored in different channels of a texture to reduce detail loss. The noise textures used for anisotropic noise are precomputed and accessed from a memory location. Memory accesses tend to slow down the graphics hardware since all dependant instructions are put on hold till the texture value is fetched. Sparse Convolution Noise and Spot Noise generate noise differently as they do not use a regular lattice of PRN [Ebert et al. 1998]. Modified noise by Olano [2005] generates random numbers on the fly, rather than using a *baking* mechanism. Baking implies generating random numbers off line and accessing them through texture memory. MD5 does half the job as it provides good quality random numbers, but consumes sufficient time for its execution. Recently, Tzeng and Wei [2008] have used MD5 for their white noise. MD5 was primarily a good match since it utilised the new generation of GPU hardware, but it lacked simplicity and had a substantial amount of running time.

After surveying many cryptographic hash functions, we found that all of them lacked some part of our requirements. Some of these candidates included RC4; which requires a lookup table, CAST-128; which requires a large amount of data for initialization, Blowfish; which was fast but each new key required pre-processing equivalent to encrypting nearly 4 kilobytes of data. These hash functions serve the purpose of encryption while our goal here is randomness with as little computational cost as possible. MD5 is purely computational but we propose being a recent cryptographic hash function, contains most elements to confuse and puzzle potential code breakers. This being the property of any good encryption algorithm to protect itself and make it hard for someone to break its cipher. However, we are not interested in wasting any clock cycles on such instructions or structural mapping since our use here is not for encryption. We just require good quality random numbers with the least computational expense as possible. TEA being designed for fast execution and minimal memory imprint served well to all our requirements. It can be used in a parallel environment with sufficiently less load than MD5 and the speed and quality trade off can be tuned accordingly for different applications.

3 Our Approach

3.1 Requirement Goals

The first stage of our design decisions involved choosing the right method for creating the random numbers. Our primary focus was on quality, simplicity and fast computation of the algorithm; quality meaning a very random output, simplicity with respect to implementation, fast implies that a minimal set of instructions and dependencies of those instructions on each other.

Furthermore, no texture memory should be required. This meant that the random number generator being used will not be able to save its state and hence all such algorithms that require large arrays to initialize or have a dependant structure will not work. The algorithm should be able to use the GPU to its maximum strength. Looking for a cryptographic hash served just about all of those purposes. Another added advantage was that these functions use bitwise operations which can easily be executed quickly on the new generation of GPUs using the *unsigned integers* data type.

In our research we found that TEA and its extensions were best suited to our needs. Simple algorithms that provides sufficient randomness comparable to any good RNG. This property of TEA has been discussed by Reddy [2003] stating that even when reducing the number of rounds, it provides a very random output. The **Avalanche Effect** exists in the output result when 6 rounds are used. The Avalanche Effect implies that changing 1 bit of input causes a drastic change to the output. Changing the number of rounds allows the user to tweak the randomness function according to requirements, rather than allowing it to become a bottleneck by using the same algorithms in every application.

3.2 TEA, XTEA and XXTEA

TEA is a Feistel type cipher. A Feistel cipher is a symmetric structure of iterative nature used in the construction of block ciphers. A dual shift in a single round allows the output to be mixed continuously per round. Its performance on the desktop and the GPU is very impressive. It is simple and produces a very random output. It was used extensively on legacy computers considering their executional strength and minimal RAM (Random Access Memory) capabilities. After analysis of the original TEA algorithm there were some weaknesses found [Reddy 2003] and it was thus extended to XTEA and XXTEA. The Key was introduced into XTEA slowly as compared to the original form. The key scheduling including a rearrangement of the shifts, XORs, and additions is also present. XXTEA is more efficient than XTEA 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 the immediate neighbours in encrypting each word in the block. However, our input is fixed to 64-bit for TEA, XTEA and XXTEA algorithms used in our research. From this point onwards we will refer to TEA, XTEA and XXTEA collectively as ALL_TEA in our discussion. Furthermore, an algorithms with a specific number of rounds will be refereed to as TEA_X, where X signifies the number of rounds employed.

4 Implementation

We used GLSL for our primary implementation and tested MD5, TEA, XTEA and XXTEA on the GPU. Most of these functions can be varied by changing the number of rounds that are executed. Our MD5 implementation that is compared to TEA and its invariants in all our experiments is the optimized *MD5GPU* presented by Tzeng and Wei [2008]. This specific implementation does not store an array of sine values like the original MD5 encryption algorithm. The sine values are calculated at runtime and the rotational storage is reduced to 16 unique numbers. All rounds for all hash functions used were inlined for optimal performance and to present a fair contrast of results.

Following is a single round implementation of TEA.

```
UInt32[2] encrypt ( UInt32 v[2] ){

UInt32 k[4], sum = 0, delta = 0x9e3779b9
k = { A341316C, C8013EA4, AD90777D, 7E95761E }

// OneRound
sum+ = delta
v0+ = ((v1 ≪ 4) + k0) ⊕ (v1 + sum) ⊕ ((v1 ≫ 5) + k1)
v1+ = ((v0 ≪ 4) + k2) ⊕ (v0 + sum) ⊕ ((v0 ≫ 5) + k3)

return v
}
```

Algorithm 1 : Single round TEA Implementation

4.1 Random Number Datasets

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 rest to 0. The encrypted numbers resulted in a 512 MB dataset for ALL_TEA functions. The same input was used for MD5 and TEA_X 3-D noise implementation. The NIST dataset was created by passing the whole range of 1-10485670. The resulting output for NIST 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 found that having this large a dataset affected some of the results. XXTEA with a smaller dataset when used with the older version of DIEHARD which contains 15 test passed 9 in total. When the algorithm was tested with a larger dataset with the new DIEHARD suite v0.2b which has 17 tests, it only managed to pass two.

4.2 2-D Noise

Noise generation using cryptographic hash is done using the gradient noise generation technique in Ebert et al. [1998].

```
float noise ( vec2 p )
{

vec2 i = floor(p), f = fract(p),
sf = 6 ∗ pow(f, 5) − 15 ∗ pow(f, 4) + 10 ∗ pow(f, 3)

//4 components encode corners of square
//(x, y), (x, y + 1), (x + 1, y), (x + 1, y + 1)
vec4 h;
h.x = hash(uvec2(i.y, i.x))
h.y = hash(uvec2(i.y + 1, i.x))
h.z = hash(uvec2(i.y, i.x + 1))
h.w = hash(uvec2(i.y + 1, i.x + 1))
//h = h/constant (depending on X in TEA_X)

//gradients at four corners
vec4 g = (f.x − vec4(0, 0, 1, 1)) ∗ (fract(h ∗ .5) ∗ 4 − 1)+
(f.y − vec4(0, 1, 0, 1)) ∗ (fract(floor(h ∗ .5) ∗ .5) ∗ 4 − 1)

//combine to generate noise
g.xy = mix(g.xy, g.zw, sf.x)
return mix(g.x, g.y, sf.y)
}
```

Algorithm 2 : Pseudo-code for 2D Noise Implementation

Using 2D noise requires 4 function calls no matter which cryptographic hash is chosen. Since ALL_TEA require a 64-bit input and MD5 requires a 128-bit input, the two corner values are input to the hash functions and the encrypted output is used as gradients at the edges. All unused bits in the input are set to 0. The dot product is computed between the vectors and the gradients. We use the interpolation function provided by Perlin [2002] for his improved noise.

We also provide results using a 4×4 grid of gradients for each lattice point. The 16 weights are blended together using a bicubic interpolation. These gradient values require 16 calls to ALL_TEA and provide a tighter frequency bound for the output noise. Even with the added computations, the work load is much less and it falls below that of the MD5_64 implementation of 2D noise (Table 1 and 2) with only 4 function calls using a 2×2 grid. Still, number of rounds can be reduced for TEA and the grid can be sampled at even points to reduce computational overhead.

4.3 3-D Noise

When encrypting lattice points of a cube, MD5 can be used as it is since it takes a 4-component vector as input. MD5 in this case requires 8 function calls to the PRNG for a single pixel to generate noise at its location. For ALL_TEA, we require 12 function calls. Even with the added number of calls TEA stands out as being the optimal choice to be used for 3-D noise considering quality and speed.

$$hash(P_i) = ALL\_TEA(ALL\_TEA(P_i[x], P_i[y]) + P_i[z], 0) \quad (1)$$

5 Analysis and Results

We compare ALL_TEA against the MD5 cryptographic hash function. Fourier transforms are provided for white noise and gradient noise comparisons with existing techniques. We analyse execution times on AMD and NVIDIA GPU platforms.

5.1 Quality

Random numbers used in numerical analysis need to be unbiased for good results. This factor draws a line between the type of PRNG that should be used. We judge quality of numbers using the NIST randomness tests and the DIEHARD tests. Furthermore, we provide the Fourier transforms for encryption algorithms we tested to visually show how the distribution of noise is spread out across the frequency spectrum.

5.1.1 Randomness Tests

Statistical tests tend to provide a p-value when a set of numbers is run through it in order to test their statistical quality. The passing criteria for any set of numbers is that the p-value should lie with in the range .01< p <.99. For tests that produce just 1 p-value like the Overlapping 5-Permutation Test, it is easy to assign a passed or a failed status. For tests that produce 1 p-value for an individual range of bits like the Bit Stream Test, it is sometimes hard to determine if the test was passed or failed. More than 1 out of all the ranges might fail or in the case of the 3D Sphere Test, 1 sample might fail even though all others passed, at these occurrences interpretation gets a little doubtful. This phenomena is also discussed in the test documentation where about 6 p-values must fail for a set of numbers to actually consider that PRNG to have failed that test. In our experiments, if a test outputs less than 10 p-values, we consider a test failed if any single p-values is out of range. If they are greater than 10, one p-value out of the range is considered as being a random anomaly and can be ignored. If however there is

any other p-value in any part of that statistical test that does not fall in the acceptable range, the test is considered a failure.

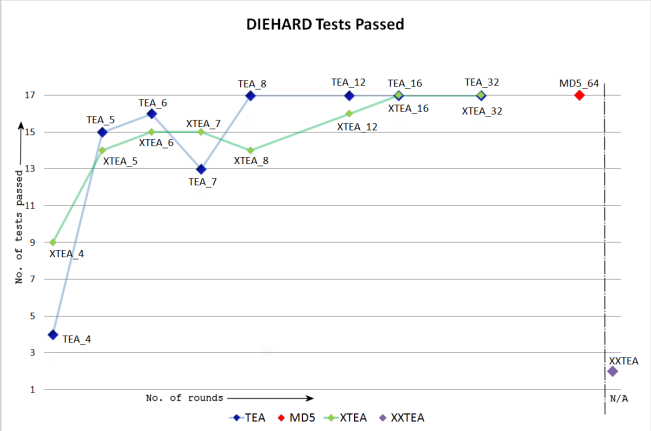

Figure 2: DIEHARD Test Results

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. DIEHARD and NIST tests help in identifying the least number of rounds required for the cryptographic hash functions to output truly random numbers. Using fewer rounds might produce a speed up, but will likely compromise the quality of the random numbers. The results (Figure 2 and Figure 4) show that TEA_8 is one the best option as a cryptographic hash function for generating numbers with the least correlated numbers. Its quality is remarkable as it passes 14 NIST and all 17 DIEHARD randomness tests with 8 number of rounds. It provides a perfect balance between number of rounds executed and quality of output.

TEA implementation with less than 8 number of rounds is less random than MD5_64 (Figure 2). TEA_7 (Figure 2) turns out to be the breaking point, and from which point on TEA output becomes progressively less random as the number of rounds are decreased. Figure 4 shows the same trend starting at TEA_5. XTEA shows signs of degradation in quality when the number of rounds is less than 16 (Figure 2). The general trend leading to a fall in quality after that. XXTEA only manages to pass 2 out of 17 DIEHARD and 9 out of 15 NIST randomness tests.

5.1.2 Visual Comparisons of Fourier Transforms

White noise distribution in frequency space is presented in (Figure 3) for MD5 and ALL_TEA hash functions. Fourier Transforms for Perlin Noise [Perlin 2002] and Modified Noise [Olano 2005] are also presented to match quality contrast with our technique. The images show a flat power spectrum and that it is radially symmetric for the cryptographic functions. The uneven power distribution for Blum Blum Shub can be seen in the image for modified noise inside the band limits. Perlin noise also lacks the even distribution inside the frequency doughnut present for MD5 and ALL_TEA.

5.2 Speed

We ran our speed tests using the AMD GPU Shader Analyzer and the NVIDIA Shader Perf tool. We used an AMD Radeon HD 4870 and the NVIDIA G80. Only the computational algorithms were

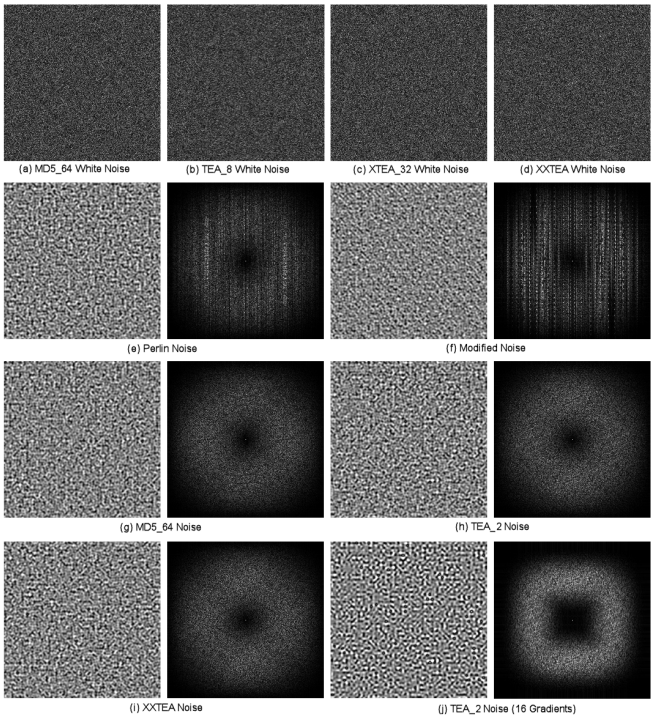

Figure 3: Fourier Transforms

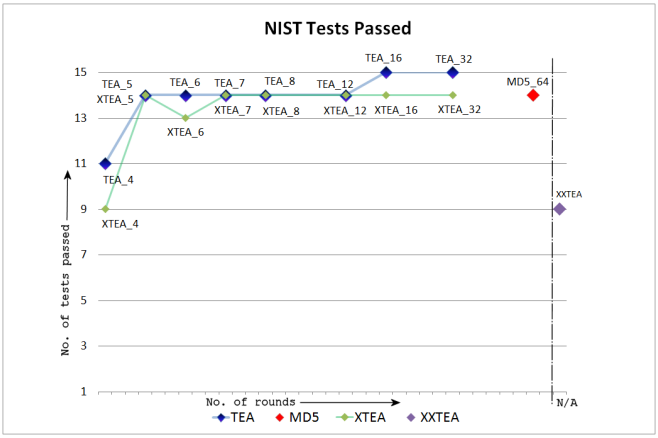

Figure 4: NIST Test Results

tested on the two GPU architectures. The results show that all the TEA and XTEA (2D and 3D) noise implementations are faster than equivalent MD5_64 noise on both the Radeon HD 4870 and the G80 (Table 1).

Conducting the tests on different platforms shows that results may vary for an algorithm from architecture to architecture. XXTEA in the case for 3-D noise is the slowest algorithm to run on the AMD Radeon HD 4870, but it does pretty well on the NVIDIA G80. The differences can be accounted for by noticing that the algorithm uses conditional statements which we assume have different execution performance on different architectures. For a random number requirement that is solely for noise generation to be used graphically, it is recommended that XTEA_2 or TEA_2 be used, since they have very random frequency plot distributions with no visual artefacts

	AMD		NVIDIA	
Function	**Cycles**	**MPixel/sec**	**Cycles**	**MPixel/sec**
2-D Noise				
Modified	3.9	3077	147	1628
MD5_64	69.2	173	2629	61
TEA_32	45.6	263	2066	79
TEA_16	23.2	517	1040	174
TEA_8	12.0	1000	528	322
TEA_2	3.7	3243	144	1305
XTEA_32	36.5	329	1566	92
XTEA_16	18.9	635	796	184
XTEA_2	3.4	3529	124	1551
XXTEA	63.1	190	212	1031
2-D 16 Gradient Noise				
MD5_64	443.9	27	10474	10
TEA_8	42.7	281	2064	81
TEA_2	12.0	1000	527	364
XTEA_16	64.3	187	3135	40
XTEA_2	10.5	1143	445	419
XXTEA	4047.6	13	823	276
3-D Noise				
MD5_64	218.9	55	5452	30
TEA_8	33.4	359	1546	102
TEA_2	9.6	1250	394	481
XTEA_16	51.0	235	2361	40
XTEA_2	8.5	1412	342	604
XXTEA	2399.8	20	632	358

Table 1: Noise Speed Statistics for AMD Radeon HD 4870 and the NVIDIA G80

and run the fastest.

Speed results with different numbers of rounds for different cryptographic functions are provided to show how they drain GPU resources and to help choose the right one for the required task. If any operation requires quality pseudo random numbers other than generating visual objects in a scene (a learning algorithm perhaps), then TEA_8 will be best choice as it strikes a perfect balance between speed and quality.

5.3 Comparisons with best performance and best quality producing previous work

The performance of TEA algorithms can be easily compared with the fastest performing noise algorithm (Modified Noise) using Table 1. Table 1 shows that XTEA_2 actually out performs modified noise in execution time. Fourier transforms in Figure 5 show that the random distribution in frequency space of XTEA_2 is as good as the best quality previous work (MD5_64). For gaming platforms, the cryptographic hash function XTEA_2 is the optimal choice for generating noise. The video file provided with this paper compares its visual quality with the fastest performing previous work (Modified Noise) and execution performance (in frames per second) against the best quality producing previous work (MD5_64). The video shows that modified noise exhibits visual artefacts which are not present when using XTEA_2 in the same texture space for the same effect (Figure 6). It also shows how the frame rate is substantially less when using MD5_64 than that of an XTEA_2 implementation for an effect.

6 Application

Random numbers are used for numerical analysis methods, learning algorithms and Monte Carlo methods. Amongst other applications,

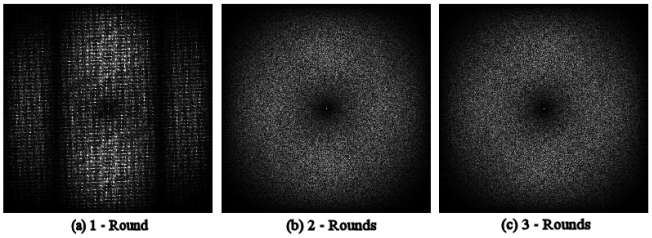

Figure 5: Fourier Transforms for XTEA Noise with increasing number of rounds

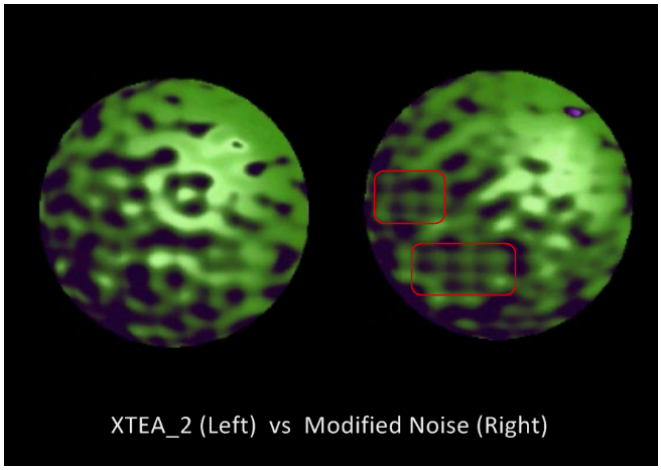

Figure 6: Visual artefacts exhibited by Modified Noise, not present in XTEA_2 Noise

Soft Shadowing and Metropolis Light Transport, all need good random numbers for realistic results. Random number use in Noise applications range from creating procedural textures (wood, marble, stone etc.) to imitating effects like smoke, translucency, fire, etc.

6.1 Fast Controlled Noise using Sparse Gabor Convolution

Recently, Lagae et al.[2009] used gabor filters to control noise in the frequency domain. They use a linear congruential generator as their PRNG. In their algorithm they use a 3×3 grid of cells per lattice point and require $1+4x$ calls to their PRNG per cell; where x is the number of impulses for a cell. Since they use the convolution method for noise generation, the Poisson function also uses calls to the PRNG. In the GPU implementation for using the linear congruential generator, the previous value has to be stored in a small texture location and multiple read/write accesses are required for calculating a noise value for a single cell. A chaining of instructions will occur as parallel shader unit executions will block due to accessing of a shared memory space, thus increasing the running time of the algorithm. We propose that using XTEA_2 instead of the linear congruential generator will be faster and provide pure procedural noise using sparse gabor convolution with no texture access for the PRNG. XTEA_2 and TEA_2 are the only algorithms that rival the speed of Blum Blum Shub on the NVIDIA and AMD hardware (Table 1).

This application is an excellent example where the user might requires good random numbers to be used in the Poisson function, but mildly random numbers for hashing, just enough to have a random

distribution in the frequency space with no visual artefacts. Our testing showed that the linear congruential generator only passes 3 out of the 17 DIEHARD Tests. With our proposed cryptographic functions, the best fits with least time consumption considering quality can easily be found, rather than using one rigid implementation for a PRNG.

7 Conclusion

The choice of an algorithm for random numbers depends on a number of factors. A better understanding of how the random numbers are being used and the choice of the algorithm can make a huge difference in the performance of an application. Using TEA and its extensions as a PRNG has many advantages for certain algorithms with respect to the application of random numbers and the time of execution. We have provided a detailed analysis for how these cryptographic hash functions perform and which one to use depending on a number of factors involved.

References

COOK, R. L., AND DEROSE, T. 2005. Wavelet noise. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 803–811.

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

EBERT, D. S., MUSGRAVE, K., PEACEY, D., PERLIN, K., AND ANDWORLEY, S. 1998. *Texturing Modeling, A Procedural Approach*.

GOLDBERG, A., ZWICKER, M., AND DURAND, F. 2008. Anisotropic noise. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, ACM, New York, NY, USA, 1–8.

HART, J. C. 2001. Perlin noise pixel shaders. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM, New York, NY, USA, 87–94.

KELLER, A., HEINRICH, S., AND NIEDERREITER, H. 2006. *Monte Carlo and Quasi-Monte Carlo Methods*.

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

LAGAE, A., LEFEBVRE, S., DRETTAKIS, G., AND DUTRÉ, P. 2009. Procedural noise using sparse gabor convolution. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, ACM, New York, NY, USA, 1–10.

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

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

NIST, 2008. Nist statistical test suite. `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`.

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., HART, J. C., HEIDRICH, W., MARK, B., AND PERLIN, K. Real-time shading languages. SIGGRAPH 2002 Course 36.

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

PANG, W. M., WONG, T. T., AND HENG, P. A. 2006. *Shader X5: Advanced Rendering Techniques*.

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

PERLIN, K. 2002. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 681–682.

REDDY, V. 2003. *A cryptanalysis of the Tiny Encryption Algorithm*. Master's thesis, University of Alabama.

RUKHIN, A., SOTO, J., NECHVATAL, J., SMID, M., BARKER, E., LEIGH, S., LEVENSON, M., VANGEL, M., BANKS, D., HECKERT, A., DRAY, J., AND VO, S. 2008. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. In *NIST Special Publication*.

SPJUT, J. B., KENSLER, A. E., AND BRUNVAND, E. L. 2009. Hardware-accelerated gradient noise for graphics. In *GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI*, ACM, New York, NY, USA, 457–462.

TZENG, S., AND WEI, L.-Y. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 79–87.

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