APPROVAL SHEET

Title of Thesis: Tiny Encryption Algorithm for Cryptographic Gradient Noise

Name of Candidate: Fahad Zafar
Master of Science, 2009

Thesis and Abstract Approved: ________________________
Dr. Marc Olano
Associate Professor
Department of Computer Science and
Electrical Engineering

Date Approved: ______________________________

Curriculum Vitae

Name: Fahad Zafar.

Permanent Address: 8 Comill Court Apt 2D. Owings Mills, MD 21117.

Degree and date to be conferred: Master of Science, December 2009

Date of Birth: July 13, 1985.

Place of Birth: Karachi, Pakistan.

Secondary Education: BeaconHouse School System, Lahore, Pakistan.

Collegiate institutions attended:

University of Maryland Baltimore County, M.S. Computer Science, 2009
Punjab University College of Information Technology, B.S. Computer Science, 2007

Major: Computer Science.

Professional publications:

Zafar, Fahad and Olano, Marc. 2009. Tiny Encryption Algorithm for Cryptographic Gradient Noise. Interactive 3D Graphics and Games 2010. (submitted)

Professional positions held:

Research Assistant. Multicore Computational Center UMBC (8/2009 - Present).

Associate Software Developer. Generation Internet Teams (9/2007 12/2007).

ABSTRACT

Title of Thesis: Tiny Encryption Algorithm for Cryptographic Gradient Noise

Fahad Zafar, Master of Science, December 2009

Thesis directed by: Marc Olano, Associate Professor
Department of Computer Science and
Electrical Engineering

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 (Graphics Processing Units) 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.

This thesis explores use of some cryptographic algorithms as hash functions that produce random numbers with minimum correlation, 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 the band-limited gradient noise rendering primitive. 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 better results, while slower versions can be used for numeric simulation, providing random numbers that pass the NIST and DIEHARD randomness test suites.

Tiny Encryption Algorithm for Cryptographic Gradient Noise

by

Fahad Zafar

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2009

Dedicated to my family.

ACKNOWLEDGMENTS

I thank my advisor Dr. Marc Olano for his support and guidance throughout this thesis work. Thanks to my parents for their never-ending support and for all their motivational speeches.

I am pleased to thank all VANGOGH Lab members for their time and support. I am also grateful to my roommates Nathan Buesgens and Patrick Gray for adjusting to my schedule.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

THESIS STATEMENT

Certain cryptographic algorithms can provide a basis for sufficiently fast random number generators on the GPU. MD5 is the current existing solution, but it is shown to run slower and a much better alternative to this algorithms is offered. The thesis brings forward TEA as a much better approach using quantitative results based on visual and randomness quality tests.

Chapter 2

INTRODUCTION

Modern graphics cards are high performance programmable parallel processors. One of the most computationally expensive aspects of image generation performed by a graphics processing unit (GPU) is determining the color of each pixel for each rendered surface. This process is described by a flexible set of instructions called a *shader*, run on the parallel processors of the GPU. One widely used technique uses images called *texture maps* to provide color and detail for these procedures. When using stored images for texture mapping, a *texture lookup* takes place to find out where on the texture each pixel falls. Texturing hardware uses compression and block caching to avoid stalling for memory accesses on image-texture fetches, but when textures hold non-image data, incoherent access patterns can defeat the caching. An incoherent texture fetch can halt the shader execution stream and cause delay 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 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, and are an important part of GPU applications. In particular, we believe the generation of random numbers on the GPU should not require the use of texture lookups as they slow down computational instructions which do not have control statements. Hence, the most suitable GPU random number generator would be one that is entirely computational and would be able to produce sufficiently good quality random numbers in parallel and independent to successive calls.

Parallel random number generation does not necessarily imply that the numbers must be generated in bulk. We do not require a large set of parallel numbers for one pixel, we just require that the algorithm be stateless, so that multiple pixels can call the function simultaneously.

We show that the Tiny Encryption Algorithm (TEA) meets these needs. We also show how it compares to other algorithms in terms of speed and quality. Analysis results are provided on NVIDIA and AMD hardware. DIEHARD and NIST randomness test suites are used for quality comparisons. Fast Fourier Transforms help analyze the frequency characteristics and random distribution of the noise.

2.1 Gradient Noise

Noise is a stochastic function that produces a random value for every input. You input the point x where you want to calculate the noise value and a function returns the answer. Gradient noise is a particular type of noise used in graphics where you use the neighboring points in order to calculate noise at some point x. For instance, if you are generating gradient noise for a point (x,y) on a 2-dimentional grid, first you calculate the four neighboring points of point (x,y). The four neighboring points for an input (x,y) on a 2-D grid for gradient noise are calculated as follows.

$F_x = Floor(x) \quad F_y = Floor(y) \quad C_x = F_x + 1 \quad C_y = F_y + 1$

Points = (C_x, C_y) , (C_x, F_y), (F_x, C_y), (F_x, F_y)

Input these four points into any pseudo random number generator and get 4 random values; one at each corner point of (x,y). Blend these values using an interpolation function. Similarly, when generating gradient noise in higher dimensions, the number of neighboring points increases. For 3D we have eight points and so on and so forth.

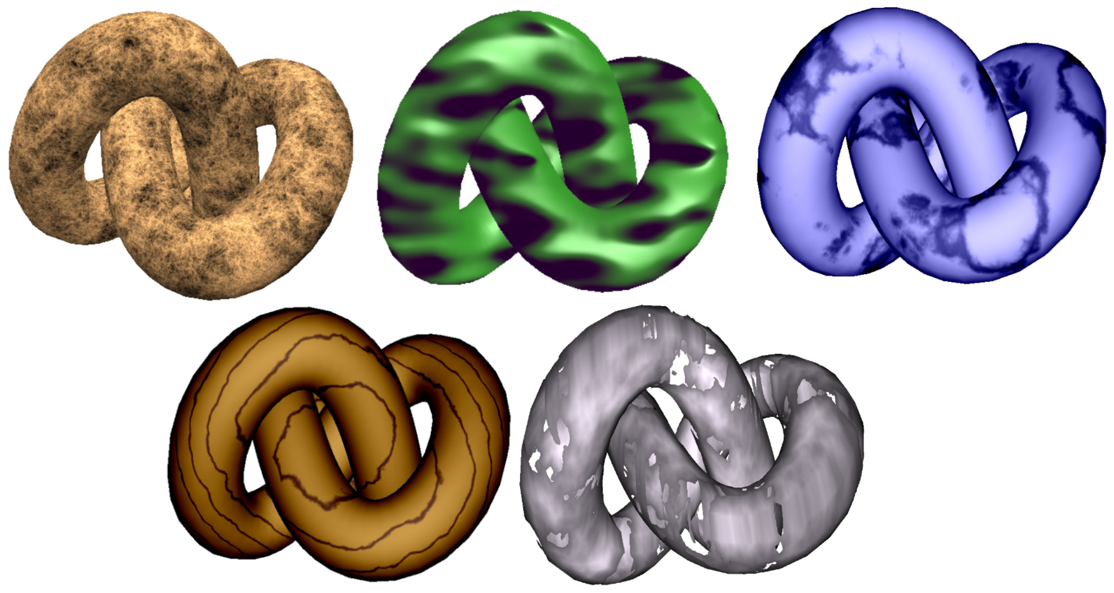


FIG. 2.1. Using TEA Noise to generate procedural granite, bumps, marble, wood and erosion.

Chapter 3

RELATED WORK AND BACKGROUND MATERIAL

3.1 Overview

Gradient noise is used as a tool for generating realistic visual effects. Much work has been done on different ways of generating noise for the GPU. Random numbers also have gone under focused analysis in the past few decades. They find extensive use in multiple theories and applications.

During the course of our discussion we use three terms which need clarification. A Pseudo Random Number Generator is one that generates a stream of random numbers seeded by some arbitrary number. Encryption is the process of converting a plain text message into cipher text which can be decoded back into the original message. Plain text may be streamed or manipulated in blocks depending on the choice of the selected algorithm. A hash function is any well-defined procedure or mathematical function that converts a large, possibly variable-sized amount of data into a usually a single integer that may serve as an index to an array. The values returned by a hash function are called hash values. We tend to use the Tiny Encryption Algorithm as a PRNG for GPU applications. When generating noise, it also serves as a hash function when observed strictly with in the lights of the definition of gradient noise. The mixing of these definitions of a hash function, a PRNG and the process of encryption in our case is due to the nature of our application. We port an encryption algorithm to computer graphics and use it as a random number generator and a hash function. This blurring of definitions applies only to our application in computer graphics and is not strictly in compliance with general distinct characteristics of the above mentioned terminologies (Encryption, PRNG and Hash functions) in their respective domains (pure mathematics and cryptography). The use of these terms in inter changeable in our discussion with loose affiliation with their explicit distinctions.

3.2 Uses of Random Numbers

All applications of random sampling require statistically random numbers. While some specialized applications use a physical process to create true randomness, almost all random number applications including Monte Carlo simulation and shader programs use a Pseudo-random Number Generator (PRNG) derived from some mathematical process that mimics true randomness. These random numbers should not have any correlation amongst generated values. Using a series of numbers that posses a correlation or are not statistically random will negatively affect results when used in numerical analysis. Random numbers used in numerical analysis 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 solutions to 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 pseudorandom numbers for their execution. Keller et al. [2006] discuss many such problems that can be addressed using this method. Cook et al. [1984] used Monte Carlo 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.

One of the most important uses of random numbers in computer graphics is noise. Noise provides realism to effects incorporated in visual applications. Common effects that require noise are fire, smoke, bumps, etc. 4-D noise normally refers to time variant random effects,for instance ripples on a water surface. Some effects require *turbulence* in order to create the desired results. The turbulence at a point is created by summing the noise at that point with scaled down noise values at other points. In artificial intelligence, most learning algorithms use noise in order to create a realistic learning effort.

3.3 Random Number Generation

Many algorithms exist for random number generation on the CPU [Menezes 1996]. PRNGs generate long sequences of number that eventually repeat after a selected period. Some common algorithms include the Linear Congruential Generator (LCG) and the Mersenne Twister [Knuth 1997]. LCG is defined by a recurrence relation and executes quickly while the Mersenne Twister algorithm, based on a matrix linear recurrence on a finite field, produces very high quality random numbers.

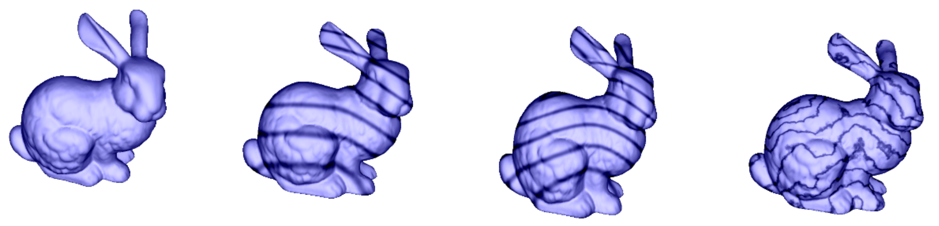
Previous methods for accessing random numbers on the GPU normally involve using a series of random numbers stored in memory. The numbers are generated offline using sequential Random Number Generators (RNGs). Pang et al. [2006] discuss Pseudo-Random Number Generator (PRNG) implementations in shaders. 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 a modification of the Blum Blum Shub PRNG in order to create purely computational noise [Blum et al. 1986]. The modified noise was fast, though the quality of the random numbers were not of sufficient quality for any numerical analysis input as reported by Tzeng and Wei [2008]. Tzeng and Wei analyze many PRNGs that fail to port onto the GPU architecture with sufficient performance benefits and use the MD5 hash as their PRNG.

3.3.1 Noise

In computer graphics, noise and other images (normally referred to as textures) are splatted on meshes using a technique called texturing. A texture co-ordinate (u,v) is input with every mesh vertex that helps place a texture on the mesh. Creating texture effects that use noise can be implemented using a noise texture or computational noise. When a noise texture is used, the GPU looks up a noise value for every u,v co-ordinate from the noise texture. When noise is generated on the fly, the vertex points are passed in to generate a noise value using a functional approach.

Perlin [1985] introduced the image synthesizer that could better mimic different real world textures using the gradient method for noise generation. Perlin's noise, and the other variants that followed, are repeatable and producing the same result if given the same argument, so noise-derived textures will remain stable on the object. A complex frequency spectrum can be sculpted by scaled additions of noise at different scales. Since they are band limited, they will be locally smooth, but should remain uncorrelated for sufficiently different argument values. There are versions of the noise function for 1D, 2D, 3D, or 4D arguments. The original gradient noise proposed by Perlin [1985] required a memory lookup for a permutation table-based hash and a table of random gradients for each point on an integer grid surrounding the noise argument. For 2D, this is four points in a square surrounding the argument position, for 3D it is eight, and for 4D it is 16.

Perlin noise is repeatable, band limited and stochastic. The same input gives same output, so it can be used for surface variation that does not change from frame to frame.



(a) Model (b) Add rings using the cos function (c) 1 octave of noise (d) 9 octaves of noise added together

FIG. 3.1. Procedural Marble created using the turbulence function

This means the local neighborhood of any point x will be similar to x, but far away may be very different. This feature allows scaling the input to the noise to control the look and scale of the output, and combining noise at multiple scales to produce more interesting prodecural effects. Each successive noise function you add is known as an octave. The reason for this is that each noise function is twice the frequency of the previous one. The marble shader uses Ken Perlin's turbulence function to distort parallel lines of color (Fig 3.1).

When adding together noise functions, you may wonder exactly what amplitude and frequency to use for each one. You can create Perlin Noise functions with different characteristics by using other frequencies and amplitudes at each step. For example, to create smooth rolling hills, you could use Perlin noise function with large amplitudes for the low frequencies , and very small amplitudes for the higher frequencies. Or you could make a flat, but very rocky plane choosing low amplitudes for low frequencies. Real marble is layers of rock with different impurities and colors that are mixed and folded over long periods of time. The noise version uses several scales of noise to replicate that look, with low frequency noise providing the large scale layer mixing and finer scales reproducing the thin, almost turbulent boundary between the layers. Figure 3.1 shows how the marble texture can be generated procedurally using scaled additions of noise. Bump mapping is implemented using distortion of normals on the surface using the noise function. No additions of noise bands is required for this effect. Procedural wood is normally generated using 2 colors of wood. In Figure 2.1 we use similar characteristics of turbulence as for the marble shader, but with less varying amplitude. Mostly artists have to tweek the frequency and the number of noise bands to get visually acceptable results. It is also wise to stop adding noise functions when their amplitude becomes too small to reproduce. Exactly when that happens depends on the level of persistence, the overall amplitude of the Perlin function and the bit resolution of your screen.

Band limited allows smooth changing of values across the noise pattern. Populate an integer grid with random values. Interpolate in between to get values at non integer points. Using a higher order interpolation function adds smoothing characteristic to the noise.

The noise function is stochastic meaning the neighborhood of a point will be similar to the point, but more than a unit or two away will be effectively random (since it is repeatable and band limited).

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 dependent instructions are put on hold while 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]. Gabor noise is a recent example of spot noise [Lagae et al. 2009] .

Modified noise by Olano [2005] generates random numbers on the fly, rather than

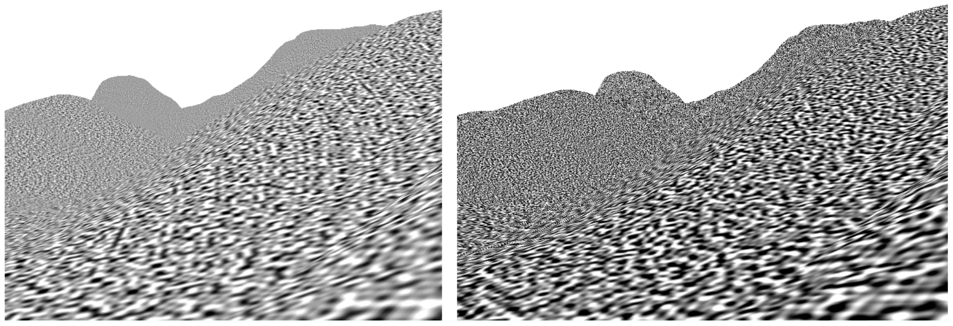


FIG. 3.2. Texture noise (*left*) suffers from loss of detail due to resolution dependence while procedural noise (*right*) does not.

using a *baking* mechanism. Baking implies generating random numbers off-line and accessing them through texture memory. Also, procedural noise does not suffer from loss of detail (Figure 3.2). When using noise stored in a texture, the noise becomes resolution dependant. The models textured with this noise suffer from loss of detail and stretching artefacts.

Recently, Tzeng and Wei [2008] used MD5 for random number generation for noise, by applying the MD5 hash function to the integer lattice coordinates. MD5 does half the job as it provides quality random numbers, but at the expense of higher execution time. It utilized integer operations in the (then) new generation of GPU hardware, but it lacked simplicity and had performance losses.

3.3.2 Cryptographic Random Numbers

After surveying many cryptographic algorithms, we found that most of them when used as hash functions lacked some part of our requirements to be fast, avoid lookup tables or large internal state, and produce high quality random numbers. Some of these candidates included RC4; which requires a lookup table [Golic 1997], CAST-128; which requires a large amount of data for initialization, Blowfish; which was fast but each new key required

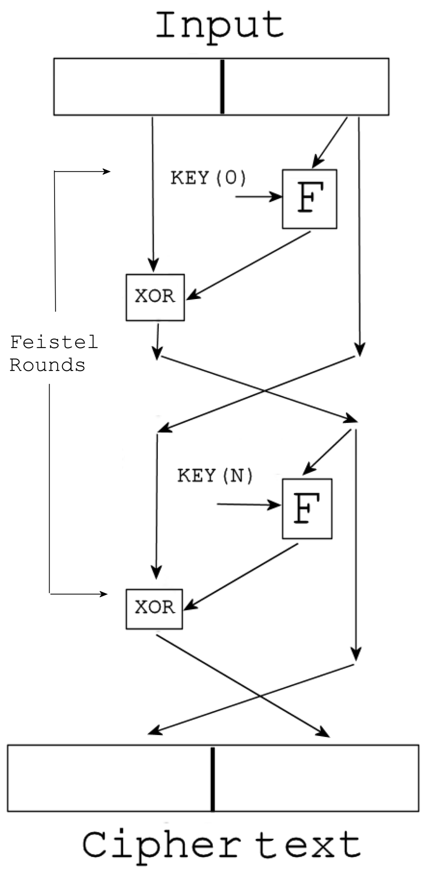


FIG. 3.3. Feistel Cipher flow diagram. F is the Feistel round that is repeated while a new key is mixed at every iteration

pre-processing equivalent to encrypting nearly 4 kilobytes of data [Schenier 1996]. These algorithms serve the purpose of encryption while our goal here is to use them as hash functions generating randomness with as little computational cost as possible. MD5 [Wang et al.] is purely computational. MD5 has 4 different type of mixing functions which run 16 times each to encrypt a single input. It has proven cryptographic qualities but must be tested to see if it can be replaced by another algorithm for computer graphics. We just require random numbers with the least computational expense possible and with minimum correlation amongst them. TEA [Reddy 2003] being designed for fast execution and minimal memory imprint fulfils 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.

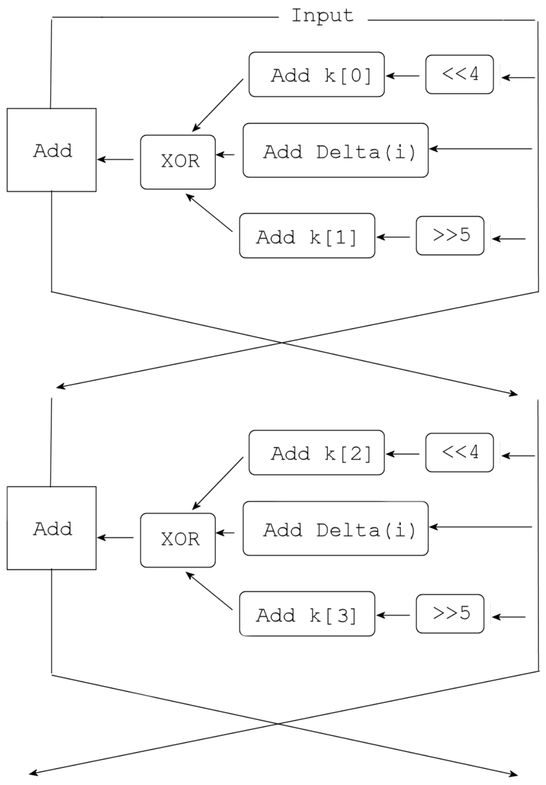


FIG. 3.4. One round for Tiny Encryption Algorithm on the GPU. The algorithm involves shifting, addition and XOR for every iteration. The number of iterations can be repeated as required.

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 if better when compared to MD5. It is simple and produces a sufficiently random output for use in graphics. It was used extensively on legacy computers considering their executional strength and minimal RAM capabilities.

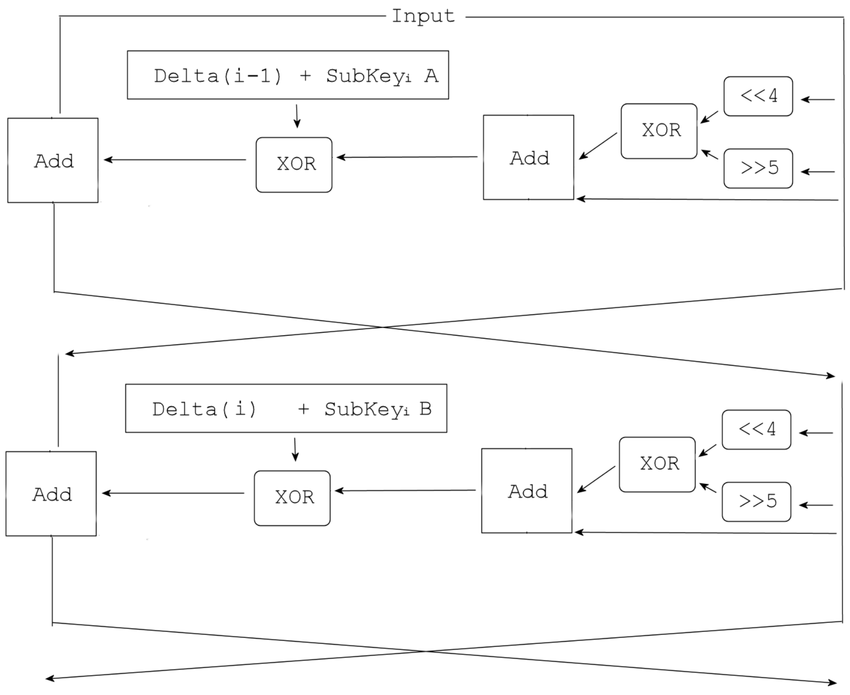


FIG. 3.5. One round for XTEA on the GPU. A variation of the original TEA algorithm. Addition of the constant Delta and the subKey occur at a later stage in the iteration

After analysis of the original TEA algorithm, there were some weaknesses found [Reddy 2003] and it was thus extended to XTEA and XXTEA [Needham et al. 1997]. The Key was introduced into XTEA slowly as compared to the original form. The key scheduling included a rearrangement of the shifts, XORs, and additions. 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 both of the immediate neighbours in encrypting each word in the block. However, our input is fixed to 64-bit for the 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 algorithm with a specific number of rounds will be refereed to as TEA_X, where X signifies the number of rounds employed.

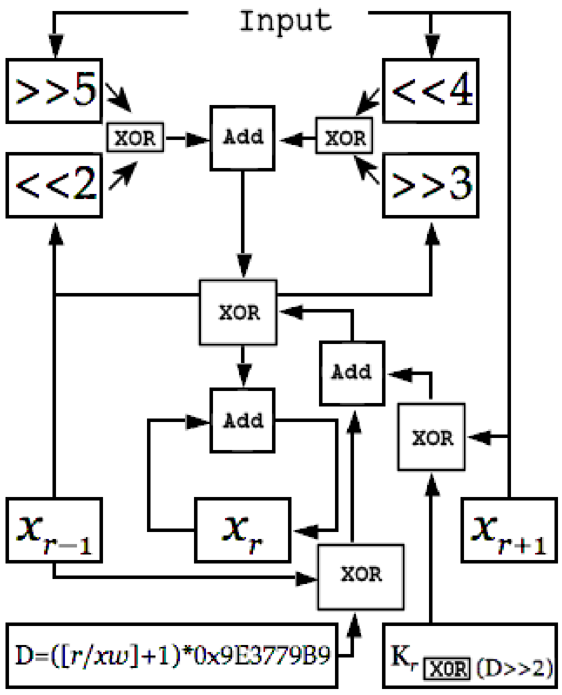


FIG. 3.6. XXTEA flow diagram. A different approach to implementing TEA rounds, using conditional statements. Atomic operations still include XOR, addition and shifting.

3.4 Graphics Hardware

The GPU is an extremely fast computational system. Our approach is accelerated by using hardware that supports bitwise operations. The new line of GPU hardware supports unsigned integer operations on a hardware level, as compared to GPUs of a few years ago that supported floating point computation. GPUs contain programmable stream processors in a massively parallel structure. Vertex data sent from the CPU is converted to pixels in

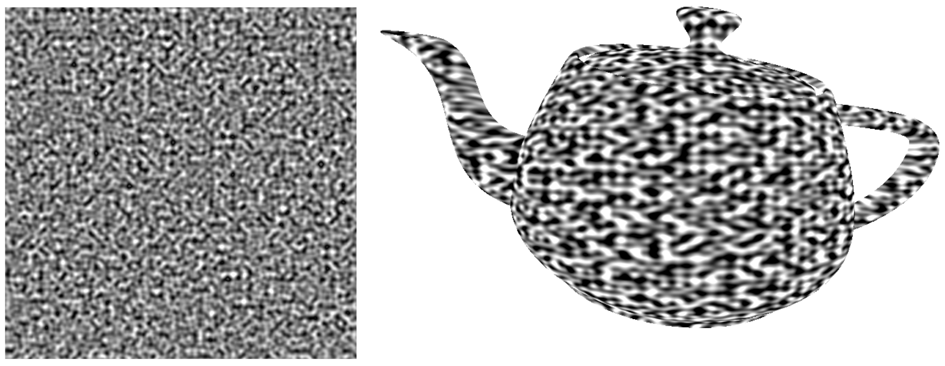


FIG. 3.7. 2-Dimensional TEA Noise generated using 2 rounds. The image is textured on to a teapot mesh.

screen space through a series of stages. The vertex, geometry, and fragment stages of the rendering pipeline can be programmed with shader programs at each pipeline stage performing functionality as required. In the vertex stage, all the vertex (compulsory), normal, color data is streamed from the main memory. Input vertices are multiplied by the world, view and projection matrices. Clipping is applied for triangles outside the screen space to reduce execution cost. The rasterization brings the remaining triangles into screen space. At the end they are colored and/or textured as needed and anti-aliased(if required).

The architecture works because all the pixels and vertices at their respective stages execute independently of each other. The unified shader design consisting of individual stream processors is capable of being dynamically allocated to vertex, pixel, geometry operations for the utmost efficiency in GPU resource allocation. Figure 2.7 shows the overall architecture of the NVIDIA GeForce 8800, representative of GPU processors. The multi-stream processors (large grey blocks), which contain 16 stream processors (small green blocks), do not hold up the pipeline if there is a demanding executional task. The texture filtering units (blue blocks) get the required texture data ready for the stream processors to be used. They also store frequently used memory. The frame buffer bandwidth is very high

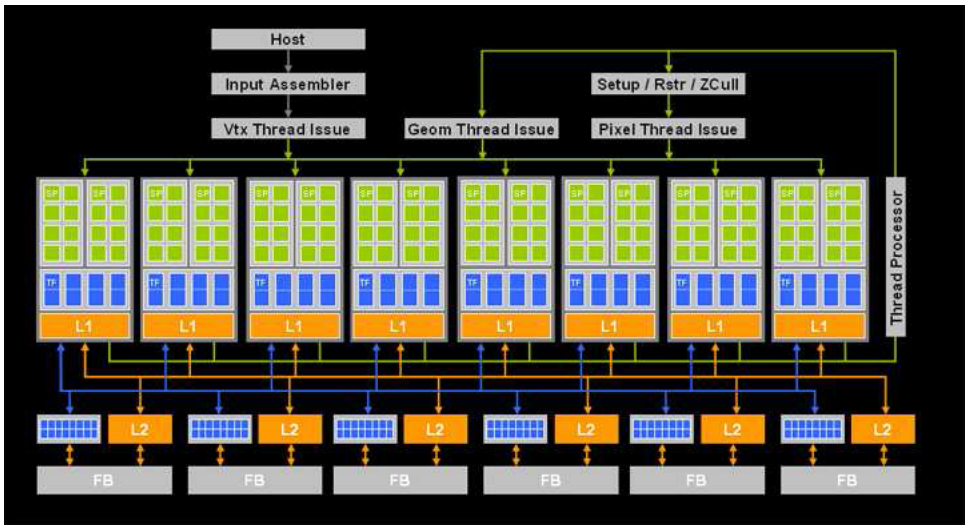


FIG. 3.8. The GeForce 8800 architecture. Source: [NVIDIA] GeForce 8800 GPU Architecture Overview, no. TB-02787-001_v01

and far more complex models and textures can be supported at high resolutions and image quality settings.

There is no data passing (yet) present from a shader executing for one pixel to any other, and very limited ability to store data from one execution to another (tens of bytes), though constant memory access which is common to all shaders can be allocated before rendering. The Tiny Encryption Algorithm relies heavily on bitwise operators for manipulating its unsigned integer variables. These operations map directly to the hardware and allow a speed up compared to a texture fetch which might halts the programmable stage while the data is being fetched.

Chapter 4

METHOD

4.1 Overview

After selecting some initial cryptographic functions we generated datasets to feed into the randomness test suites. Independent datasets were created for a function when used with a different number of rounds. First the randomness tests were conducted and then the algorithms were run through different speed tests. Results from two different randomness test suites help establish a concrete claim. Similarly, the speed tests were run on two different GPU architectures.

4.2 Requirement Goals

When choosing a method for creating the random numbers, our primary focus was on quality, simplicity and fast computation of the algorithm. Quality meaning a random output where we see no clumping artefacts in the frequency plots; 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 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. TEA is a simple algorithm that provide sufficient randomness comparable to any RNG. This property of TEA has been discussed by Reddy [2003] stating that even when reducing the number of rounds, it provides a sufficiently random output for graphics. The **Avalanche Effect** exists in the output result when six rounds are used. The Avalanche Effect implies that changing one 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.

4.3 Approach

We used The OpenGL Shading Language (GLSL) [Kessenich 1996] for our primary implementation and tested MD5, TEA, XTEA and XXTEA on the GPU. We will refer to all three TEA versions collectively as ALL_TEA, versions with X rounds as TEA_X. Most of these functions can be varied by changing the number of rounds that are executed. Our MD5 implementation 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 cryptographic functions when used as hashes were inlined for performance and to present a fair contrast of results. In all cases, the hash input is a sequence number (or optionally processor number), and the hash output is used directly as a pseudo-random number.

```
$UInt32[2]\ \ encrypt\ (\ UInt32\ v[2]\ )\{$
$UInt32\ k[4],\ sum = 0,\ delta = 0x9e3779b9$
$k = \{\ A341316C,\ C8013EA4,\ AD90777D,\ 7E95761E\ \}$
// One Round
$sum+ = delta$
$v_0+ = ((v_1 \ll 4) + k_0) \oplus (v_1 + sum) \oplus ((v_1 \gg 5) + k_1)$
$v_1+ = ((v_0 \ll 4) + k_2) \oplus (v_0 + sum) \oplus ((v_0 \gg 5) + k_3)$
$return\ \ v$
$\}$
```

Algorithm 1 : Single round TEA Implementation

4.4 Random Number Datasets

We tested randomness with the DIEHARD and NIST test suites [Marsaglia 1995 and NIST 2008]. Each consists of a series of statistical tests on a stream of random numbers, and produces a set of *p* values giving the chance that any patterns in the number stream could be due to random chance. 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. We did run an older version of DIEHARD with a smaller dataset. Of those fifteen tests, XXTEA passed nine. 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. One of the reasons for such an occurrence shows that XXTEA might have a repetition or cycling of random numbers which was large enough to escape detection in a smaller dataset. When increasing the amount of the dataset, this correlation of numbers was easily identifiable by the statistical tests.

4.5 2-D Noise

Noise is generated using cryptographic algorithms as hashed using the gradient noise generation technique in Ebert et al. [1998]. Algorithm 2 represents the noise implementation using GLSL; the OpenGL Shading Language. The language uses some uncommon data types such as *vec2, vec3 and vec4* , which represent vectors containing two, three and four floats, respectively. Similarly a *uvec* represents an unsigned integer vector. The sub-vectors are referenced using the *x*,*y*,*z* and *w* components.

Our shader implementation uses a vertex shader; that runs for every vertex of the mesh and a pixel shader; that runs for every pixel on the screen. The algorithm stated in Algorithm 2 is an implementation for noise in the pixel shader. Another fairly new operation not present in any common programming language such as java or C++ is the swizzle operation for accessing selected or repeated members of a vector data type (e.g *vector.xxyy*). The less familiar functions used include the **fract** function which returns only the fractional part of the input variable and the **mix** function which is a linear interpolation.

```
$float\ noise\ (\ vec2\ p\ )$
$\{$
$vec2\ i = floor(p),\ f = fract(p),$
$sf = 6 * pow(f, 5) - 15 * pow(f, 4) + 10 * pow(f, 3)$
// 4 components encode corners of square
// (x,y), (x,y+1), (x+1,y), (x+1,y+1)
$vec4\ h;$
$h.x =\ hash(uvec2(i.y, i.x))$
$h.y =\ hash(uvec2(i.y + 1, i.x))$
$h.z =\ hash(uvec2(i.y, i.x + 1))$
$h.w =\ hash(uvec2(i.y + 1, i.x + 1))$

// 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 in GLSL

Calculating noise at a point in 2D requires four random vectors generated at the nearest corner points. These vectors are interpolated using a blending function to output a vector value for the selected point (Figure 4.1). Using 2D noise requires four function calls no matter which cryptographic function 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

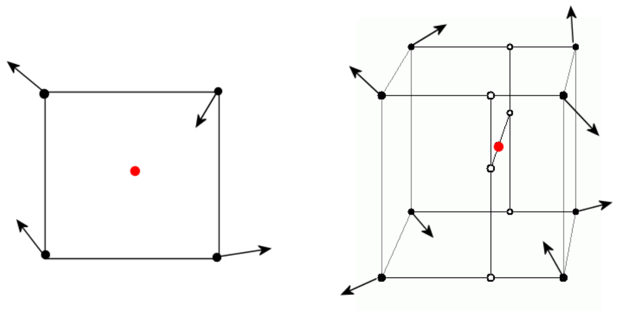


FIG. 4.1. Visualizing the neighbouring vectors for a point in 2D (*Left*) and 3D (*Right*) space when calculating gradient noise.

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. The input *x* and *y* values are used as incoming text for TEA algorithms. The key is fixed, but can be changed when required. The returned value is used as a hash value related to those *x* and *y* coordinate values.

$$hash(P_i) = ALL\_TEA(P_i[x], P_i[y]) \tag{4.1}$$

We also provide results using a 4×4 grid of gradients for each lattice point. The sixteen 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 workload is much less and it falls below that of the MD5GPU_64 implementation of 2D noise (Table 1) with only four function calls using a 2×2 grid. Still, the number of rounds can be reduced for TEA and the grid can be sampled at even points to reduce computational overhead.

4.6 3-D Noise

When finding hash values for 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 eight 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. Note that additional calls are required since every lattice point is now represented as a 3-component vector, rather than a 2-component one (as in the case for 2-D noise). If the lattice points are restricted to lie within an 8-bit value, even still there is no reduction in the number of calls required.

$$hash(P_i) = ALL\_TEA(ALL\_TEA(P_i[x], P_i[y]), P_i[z]) \tag{4.2}$$

$$hash(P_i) = ALL\_TEA(ALL\_TEA(P_i[x], P_i[y]) + P_i[z], 0) \tag{4.3}$$

Intuitively, extending Equation 3.1 to fit the 3-D noise implementation would result in Equation 3.2. However, the resulting output when using Equation 3.2 exhibits visual artifacts. Instead, we adopt the nested combination of functions, as used by Perlin [1985]. Perlin used addition with a nested 1D function to build functions for 2D and 3D inputs when hashing. When we do the same by adding the second argument to the result of the inner encryption function and using 0 as the second argument for the outer encryption function, we avoid the visual artifacts. Other implementations [Spjut 2009] use the XOR operation rather than an addition to add dimensions. Other options for building higher-dimensional hash functions and their relationship to the number of TEA iterations would be an interesting area for future work.

Chapter 5

ANALYSIS AND RESULTS

We compare the TEA family of functions against the MD5GPU cryptographic hash function. Fourier transforms are provided for white noise and gradient noise comparisons with existing techniques. We analyse execution times on the AMD and NVIDIA GPU platforms.

5.1 Quality

Random numbers used in numerical analysis need to be unbiased for accurate results. PRNGs that generate output with minimal correlation between output values are the most widely used. We judge quality of numbers using the NIST randomness tests and the DIEHARD tests. Furthermore, we provide the Fourier transforms for the encryption algorithms we tested to show how the distribution of noise is spread out across the frequency spectrum.

5.1.1 Randomness Tests

The NIST randomness tests contain a set of 15 tests while the new version of the 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 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 speedup, but will likely compromise the quality of the random numbers. The results (Figure 5.1 and Figure 5.2) show that TEA_8 is the best potential option as a hash function for random noise generation. Its quality is remarkable as it passes 14 NIST and all 17 DIEHARD randomness tests with just eight rounds, giving the same quality as MD5. It provides a balance between number of rounds executed and quality of output.

The graph starts to dip when reaching TEA_7 in (Figure 5.1). The recommended number of rounds when using TEA is sixteen. Reducing number of rounds for fiestel algorithms lowers the randomness of the resulting output. This is the point where the TEA algorithms starts to decrease in quality. Results for TEA_6 may tend to imply that the previous results were a statistical anomaly, but it can be concluded using (Figure 5.2). that TEA is loosing its output quality between four to six rounds. Similarly, XTEA shows signs of degradation with fewer than 12 rounds of execution, the general trend leading to a fall in quality after that.

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 one *p* value like the Overlapping 5-Permutation Test, it is easy to assign a passed or a failed status. For tests that produce one p-value for an individual range of bits like The Bit Stream Test, sometimes it is hard to determine if the test was passed or failed. More than one out of all the ranges might fail or in the case of the 3D Sphere Test, one 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 six p-values must fail for a set of numbers

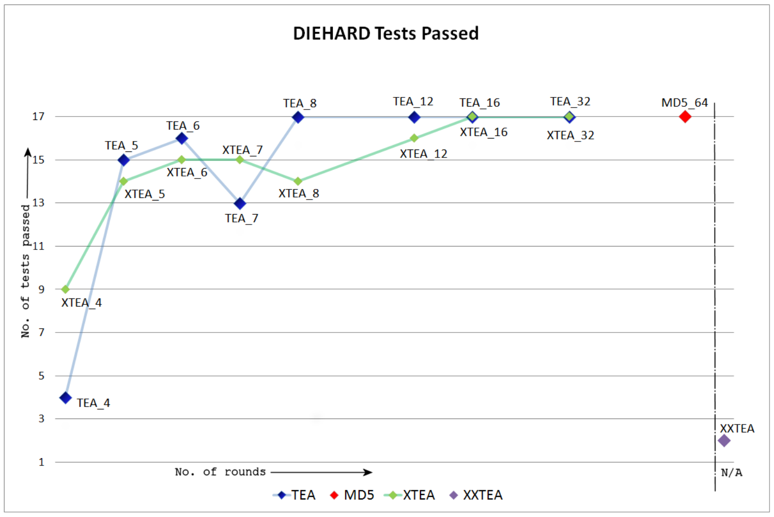


FIG. 5.1. DIEHARD Test Results for different algorithms with increasing number of encryption rounds.

to actually fail a single randomness test. In our experiments, if a test outputs less than ten p-values, we consider a test failed if any single p-values is out of range. If they are greater than ten, 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.

5.2 Visual Comparisons of Fourier Transforms

White noise distribution in frequency space is presented in Figure 5.4 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 in Figure 5.3. An algorithm that does not present any visual artifacts should not have high frequency content in the center of the circle or outside the circle. The plotted points inside the circle should be radially symmetric and uniformly distributed. The bounds for the plotted doughnut should be as tight with the least bleeding possible. A good comparison

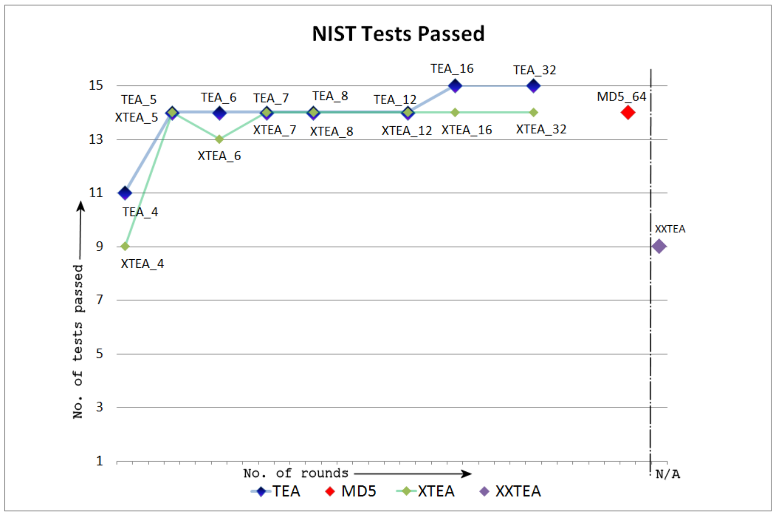


FIG. 5.2. NIST Test Results for different algorithms with increasing number of encryption round

can be made with reference to Figure 5.5 as originally presented in the Anisotropic paper by Goldberg et al. 2008. The figure shows noise obtained with custom tailored frequency spectra created by adding subband images to yield perfectly isotropic noise. Any gridding artifacts or directionality in the noise reflect poor realism the implementing visual effect. These patterns (most obvious in modified noise) point out the weakness of the hashing function that is not able to produce sufficiently random results.

The images in Figure 5.4 show a flat power spectrum for the cryptographic functions and that their noise frequency spectrums (Figure 5.3) are radially symmetric. 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. From these results, we can see that even TEA_2, which the DIEHARD and NIST tests show to be poor quality for computational random numbers, is still sufficiently random to produce a high-quality artifact-free noise.

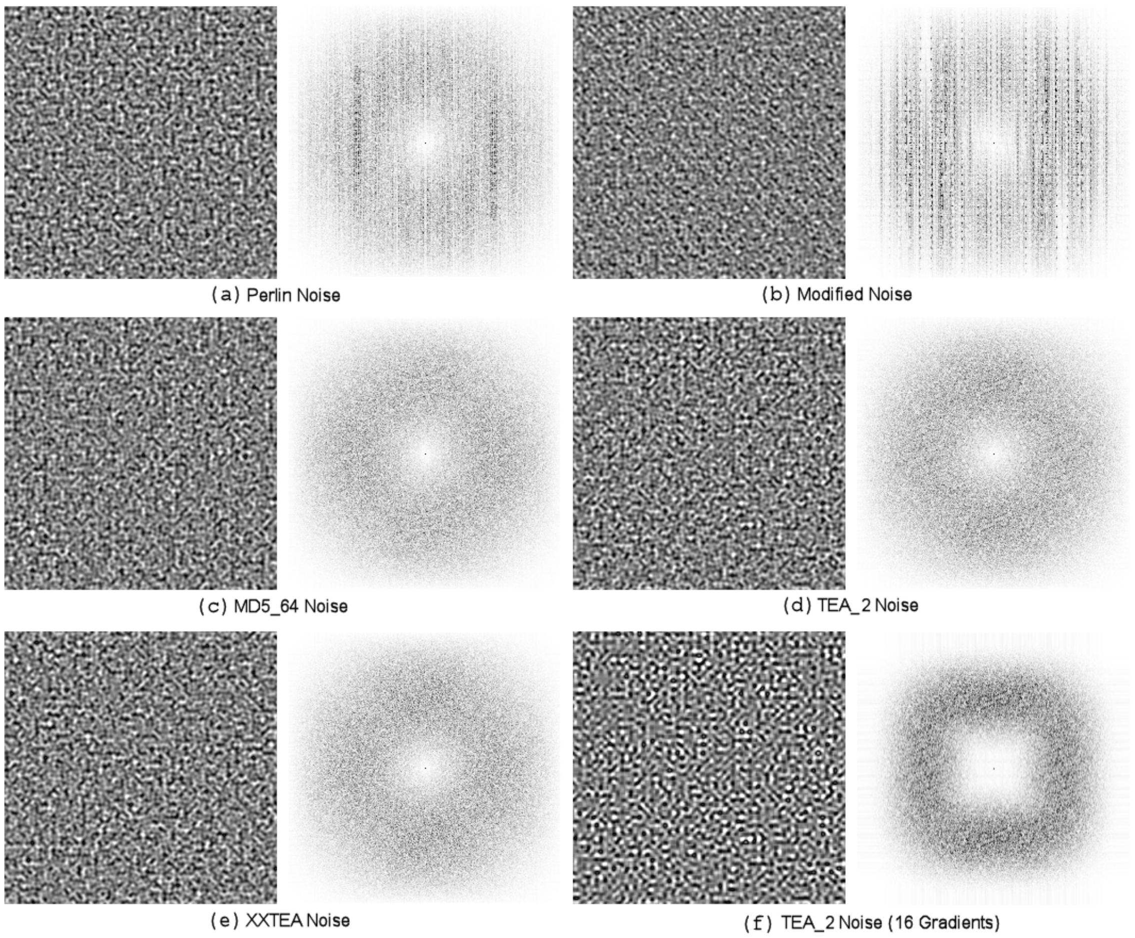


FIG. 5.3. Noise tile (*left*) with respective Fourier Transform (*right*) for different noise algorithms

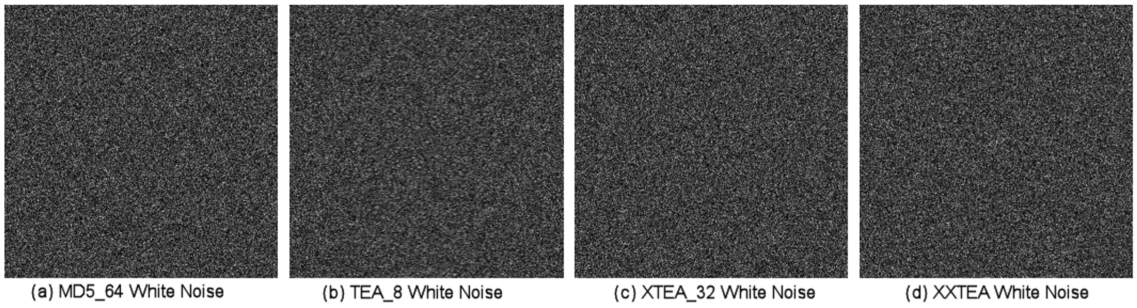


FIG. 5.4. White noise frequency distribution for different cryptographic functions

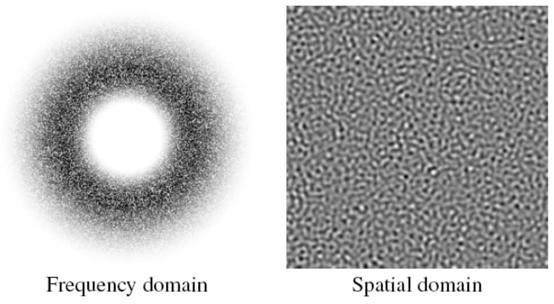


FIG. 5.5. Custom created frequency spectra to obtain perfectly isotropic noise (Source : Goldberg et al. 2008)

5.3 Speed

We ran 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 tested on the two GPU architectures. The results show that TEA_2 2D and 3D noise is faster than equivalent MD5_64 noise on both the Radeon HD 4870 and the G80.

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 comparatively batter on the NVIDIA G80. The differences can be accounted for by noticing that the algorithm uses conditional statements that 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 uniform distribution in the frequency plot with no visual artifacts 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 best fit for the required task. If any operation requires quality pseudo random numbers other than

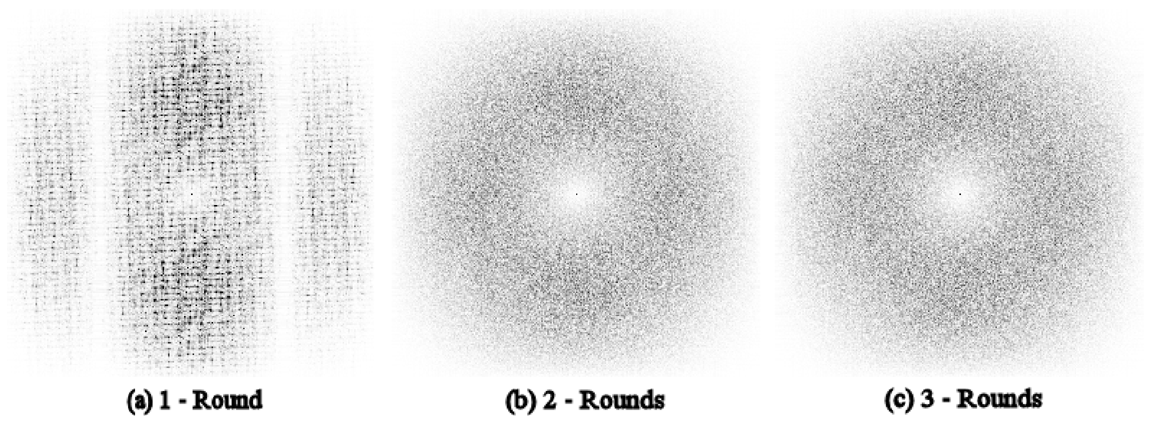


FIG. 5.6. Fourier Transforms for XTEA Noise with increasing number of rounds

generating visual objects in a scene, then TEA_8 will be best choice as it strikes a balance between speed and quality.

5.4 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 4.1. Table 4.1 shows that XTEA_2 actually out performs Modified Noise in execution time. Fourier transforms in Figure 5.6 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 comparatively better choice for generating noise.

Additionally, we show it is both faster and produces sufficiently random results than the fastest performing previous work (Modified Noise) and faster with equivalent quality to the best quality producing previous work (MD5_64). Figure 5.10 shows that modified noise exhibits visual artifacts which are not present when using XTEA_2 in the same texture space for the same effect. Figure 5.9 shows how the frame rate is substantially less when using XTEA_2 than that of an MD5_64 implementation for an effect.

	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 5.1. Noise Speed Statistics for AMD Radeon HD 4870 and the NVIDIA G80

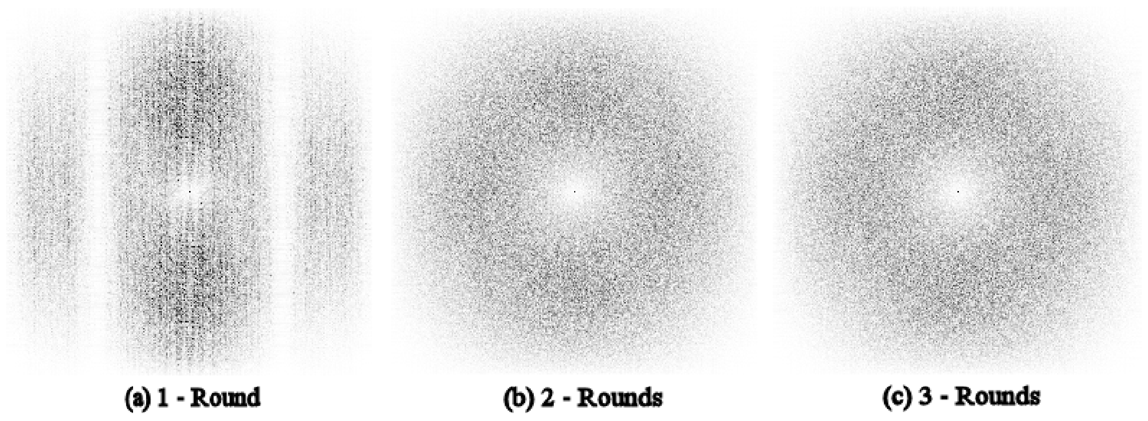


FIG. 5.7. Fourier Transforms for TEA Noise with increasing number of rounds

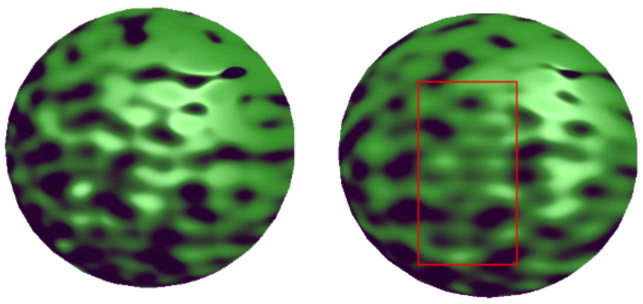


FIG. 5.8. Visual artifacts exhibited by Modified Noise (*Right*), not present in XTEA_2 Noise (*Left*)

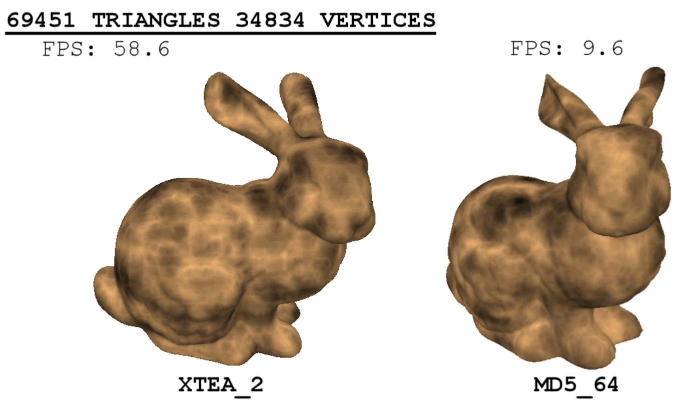


FIG. 5.9. Frame rate when using XTEA_2 Noise on an NVIDIA 8600M GT

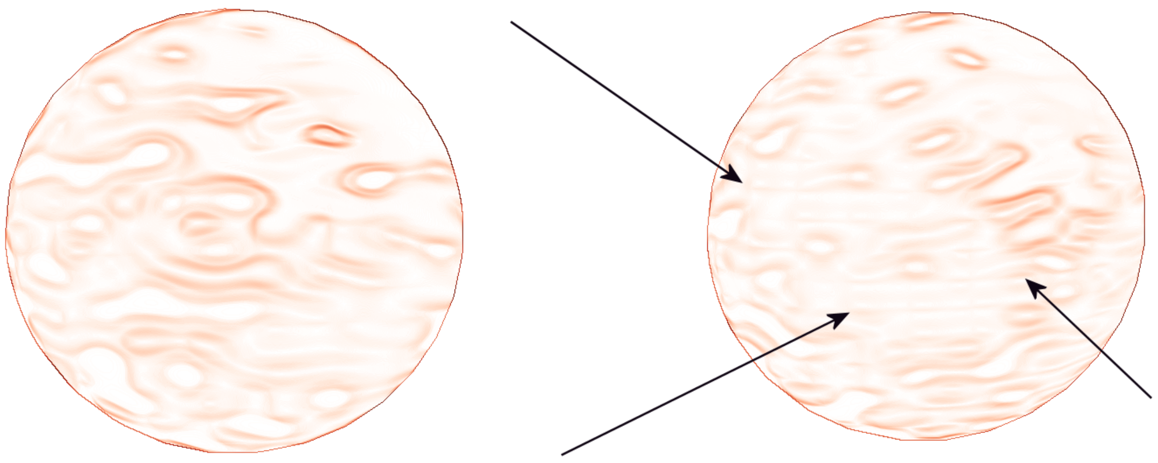


FIG. 5.10. Pointing out grid aligned artefacts using an edge detection filter on bump mapped spheres. (XTEA_2 Noise (*Left*), Modified Noise (*Right*))

5.5 Speedups for Applications using Noise

Random numbers are used for numerical analysis methods, learning algorithms and Monte Carlo methods. Among other applications, Soft Shadowing and Metropolis Light Transport, all need good random numbers for realistic results. Random numbers used in Noise applications includes creating procedural textures (wood, marble, stone) and imitating effects like smoke, translucency, fire.

5.5.1 Fast Controlled Noise using Sparse Gabor Convolution

Recently, Lagae et al. [2009] use 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 noise evaluation and require an average of $5x$ calls to their PRNG per cell, where x is the number of impulses for a cell. Typically x is between 20 and 50, for an average of 100-250 PRNG calls per cell.

This application is an excellent example where the user might require comparatively less random set of 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 artifacts. Our testing showed that the linear congruential generator passes three 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 of a PRNG.

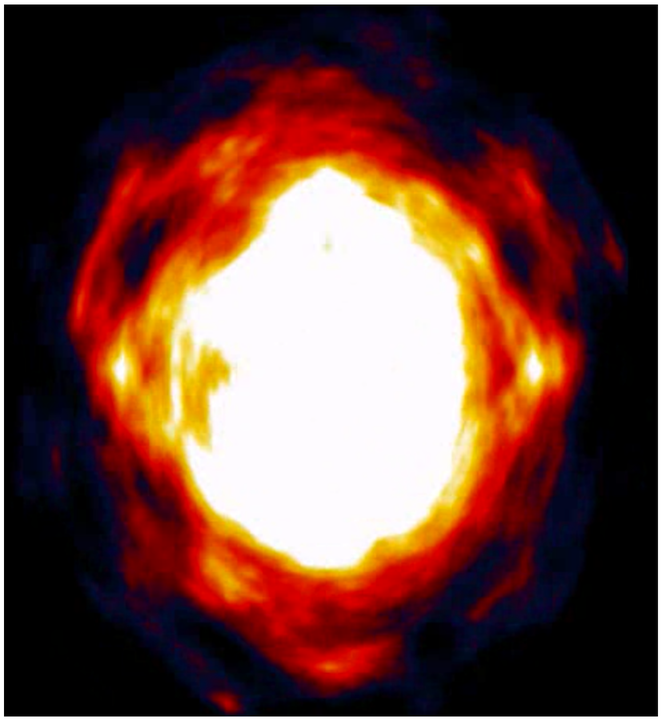


FIG. 5.11. Fire effect created using time variables and TEA noise octaves

Chapter 6

CONCLUSION

The choice of an algorithm for generating pseudo 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 hash has many advantages for certain algorithms with respect to the application of random numbers and the time of execution. We analysed how the TEA family of functions can be used as hashes in computer graphics as a better alternative than MD5. TEA has been shown to work best for generating GPU random numbers and visual effects alike. As a cryptographic function, it is stateless, allowing independent streams of random numbers to be generated without communication or storage. The ability to modify it by changing the number of rounds provides an effective tradeoff between execution speed and quality. Our results and analysis show that it is faster at equivalent or higher quality than both the fastest and the highest quality current alternatives. We have shown how TEA and its extensions perform and which one to use depending on a number of factors involved.

6.0.2 Future Work

It might be worthwhile exploring even more hash functions that work best for GPU random number generations. We have shown the previously existing TEA functions are comparatively better than any previous work, but it would be interesting to try to design a custom-tailored hash designed specifically for GPU random number generation. Instruction by instruction analysis of the algorithms on random number generators used in CUDA might be a direction to take. This process will help analyse how cryptographic instructions perform on the GPU and then take up an algorithm rich in that approach. Some hardware changes that might speed up such a process can also be proposed with such in depth analysis.

Appendix A

PSEUDO CODE FOR TINY ENCRYPTION ALGORITHMS AND ITS EXTENSIONS

A.1 TEA

```
$UInt32[2]\ \ encrypt\ (\ UInt32\ v[2]\ )\{$
$UInt32\ k[4],\ sum = 0,\ delta = 0x9e3779b9$
$k = \{\ A341316C,\ C8013EA4,\ AD90777D,\ 7E95761E\ \}$
$sum+ = delta$
$v_0+ = ((v_1 \ll 4) + k_0) \oplus (v_1 + sum) \oplus ((v_1 \gg 5) + k_1)$
$v_1+ = ((v_0 \ll 4) + k_2) \oplus (v_0 + sum) \oplus ((v_0 \gg 5) + k_3)$
$return\ \ v$
$\}$
```

A.2 XTEA

```
$UInt32[2]\ \ encrypt\ (\ UInt32\, v[2]\ )\{$
$UInt32\, k[4],\ sum = 0,\ delta = 0x9e3779b9$
$k = \{\ A341316C,\ C8013EA4,\ AD90777D,\ 7E95761E\ \}$
$v0+ = (((v1 \ll 4) \oplus (v1 \gg 5)) + v1) \oplus (sum + k[sum\&3])$
$sum+ = delta$
$v1+ = (((v0 \ll 4) \oplus (v0 \gg 5)) + v0) \oplus (sum + k[(sum \gg 11)\&3])$

$return\ \ v$
$\}$
```

A.3 XXTEA

```
$UInt32[2]\ \ encrypt\ (\ UInt32\, v[2]\ )\{$
$UInt32n = 2$
$k = \{\ A341316C,\ C8013EA4,\ AD90777D,\ 7E95761E\ \}$
$UInt32\, z = v[1],\ y = v[0],\ e,$
$sum = 0, delta = 0x9E3779B9$
$UInt32\, p, q = 6 + 52/n$
$while(q - - > 0)\{$
$sum+ = delta$
$e = (sum \gg 2)3$
$for(p = 0; p < 1; p + +)\{$
$y = v[p + 1]$
$z = v[p]+ = ((z \gg 5 \oplus y \ll 2) + (y \gg 3 \oplus z \ll 4) \oplus (sum \oplus y) + (k[y] \oplus z))\}$
$y = v[0]$
$z = v[n - 1]+ = ((z \gg 5 \oplus y \ll 2) + (y \gg 3 \oplus z \ll 4) \oplus (sum \oplus y) + (k[x] \oplus z))\}$
$v[0] = y$
$v[1] = z$
$return\ \ v$
$\}$
```

REFERENCES

[1] Blum, Lenore, B. M., and Shub, M. 1986. A simple unpredictable pseudo-random number generator. In *SIAM Journal on Computing*, 364–383.

[2] Cook, R. L., and DeRose, T. 2005. Wavelet noise. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, 803–811. New York, NY, USA: ACM.

[3] 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*, 137–145. New York, NY, USA: ACM.

[4] Ebert, D. S.; Musgrave, K.; Peacey, D.; Perlin, K.; and Andworley, S. 1998. *Texturing Modeling, A Procedural Approach.*

[5] Goldberg, A.; Zwicker, M.; and Durand, F. 2008. Anisotropic noise. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, 1–8. New York, NY, USA: ACM.

[6] Golic, J. 1997. Linear statistical weakness of alleged RC4 key stream generator. In *EUROCRYPT*.

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

[8] Keller, A.; Heinrich, S.; and Niederreiter, H. 2006. *Monte Carlo and Quasi-Monte Carlo Methods.*

[9] Kessenich, J.; Baldwin, D.; and Rost, R. 1996. *The OpenGL Shading Language.*

[10] Knuth, D. E. 1997. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition.

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

[12] 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/`.

[13] Menezes, A.; Oorschot, P.; and Vanstone, S. 1996. *Handbook of Applied Cryptography*.

[14] Needham, R., and Wheeler, D. 1997. Correction to XTEA.

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

[16] NVIDIA. 2006. Geforce 8800 gpu architecture overview. `http://www.nvidia.com/attach/941771?type=supportprimitive=0`.

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

[18] Olano, M.; Hart, J. C.; Heidrich, W.; Mark, B.; and Perlin, K. Real-time shading languages. SIGGRAPH 2002 Course 36.

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

[20] Pang, W. M.; Wong, T. T.; and Heng, P. A. 2006. *Shader X5: Advanced Rendering Techniques*.

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

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

[23] Reddy, V. 2003. A cryptanalysis of the tiny encryption algorithm. Master's thesis, University of Alabama.

[24] 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*.

[25] Schenier, B. 1996. *Applied Cryptography*.

[26] 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*, 457–462. New York, NY, USA: ACM.

[27] 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*, 79–87. New York, NY, USA: ACM.

[28] 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*, 65–76. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

[29] Wang, X., and Yu, H. 2005. How to break MD5 and other hash functions. In *EUROCRYPT*.