Hardware-Accelerated Gradient Noise for Graphics

Josef B. Spjut
School of Computing
University of Utah
sjosef@cs.utah.edu

Andrew E. Kensler
SCI Institute
University of Utah
aek@cs.utah.edu

Erik L. Brunvand
School of Computing
University of Utah
elb@cs.utah.edu

ABSTRACT

A synthetic noise function is a key component of most computer graphics rendering systems. This pseudo-random noise function is used to create a wide variety of natural looking textures that are applied to objects in the scene. To be useful, the generated noise should be repeatable while exhibiting no discernible periodicity, anisotropy, or aliasing. However, noise with these qualities is computationally expensive and results in a significant fraction of the run time for scenes with rich visual complexity. We propose modifications to the standard algorithm for computing synthetic noise that improve the visual quality of the noise, and a parallel hardware implementation of this improved noise function that allows the use of reduced precision arithmetic during the noise computation. The result is a special-purpose function unit for producing synthetic noise that computes high-quality noise values approximately two orders of magnitude faster than software techniques. The circuit, using a commercial CMOS cell library in a 65nm process, would run at 1GHz and consume $325\mu m \times 325\mu m$ of chip area.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*; I.3.7 [**Three-Dimensional Graphics and Realism**]: Color, shading, shadowing, and texture

General Terms

Algorithms, Design

1. INTRODUCTION

Procedural methods have many advantages in computer graphics. By tweaking only a handful of parameters, a digital artist can quickly populate a scene with massive amounts of rich detail. Each object or texture generated this way may have a unique appearance without any obvious repetition (e.g., tiling a hand-drawn texture.) Moreover, procedural techniques trade computation for memory. This is important since as process technology scales, compute resources will increasingly outstrip memory speeds. For texturing surfaces, the memory reduction can be two-fold: first there is the simple reduction in texture memory itself. Second, 3D or "solid" procedural textures can eliminate the need for explicit texture coordinates to be stored with the models. However, in order to avoid uniformity and produce visual richness, a simple, repeatable, pseudo-random function is required. Noise functions meet this need.

Simply described, a noise function in computer graphics is an $\mathbb{R}^N \rightarrow \mathbb{R}$ mapping used to introduce irregularity into an otherwise regular pattern. With the introduction of his noise function, Perlin [11, 16] enumerated several ideal qualities for such a function. Ideally, a noise function should have (1) a narrow bandpass limit in the texture space, and (2) a statistical character that is both stationary (translation invariant) and isotropic (rotation invariant). Peachey [10], in his excellent overview of noise, added: (3) be a repeatable pseudo-random function for a given input, (4) have a known range of outputs, and (5) avoid exhibiting obvious periodicity.

Noise has been used to simulate an incredible variety of appearances. Published examples of noise-based procedural shaders include cumulus clouds, hurricanes, clouds with coriolis effects, fire, water ripples, wavy water, sedimentary rock, and moons with rayed craters [8], marble, oak wood, brick walls, ceramic tiles, volumetric smoke, and lens flares [1]. (Figure 1 shows a simple example demonstrating some of these.) Higher dimensional noise allows for time-varying animations. Noise has even been used to compute velocity fields to emulate the appearance of turbulent fluid flow [2]. High-end movie graphics also makes extensive use of noise: rendered effects for "The Perfect Storm" were said to have averaged approximately 200 noise evaluations per shading sample [12]. Noise is ubiquitous in movie imagery and as a result, Ken Perlin was awarded a Technical Academy Award for Perlin Noise in 1997.

The noise hardware proposed in this paper is being explored as part of a special-purpose hardware architecture called TRaX [18], a multi-threaded many-core processor designed for ray tracing [17, 19]. In this architecture many thread processors share larger special-purpose functional units (such as inv-sqrt, FP-mult, and noise) to increase performance and to amortize hardware costs. That architecture is specifically targeted at ray tracing, but our noise hardware could also be used in any graphics system where high-quality noise is used for shading calculations. Including noise

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.
Copyright 2009 ACM 978-1-60558-522-2/09/05 ...$5.00.

Figure 1: An example scene exhibiting noise-based procedural textures. Perlin noise was used to generate the wood grain pattern, the marble pattern, and the irregularities in the bricks and to control the density of the volumetric smoke. 1.3 billion noise evaluations were computed to render this image, averaging 552 per shading sample. 37.2% of the renderer's execution time was spent evaluating noise.

hardware on an existing commodity graphics chip (GPU), could greatly increase performance for procedural texturing on those systems.

1.1 Noise in Graphics

One of the simplest possible noise functions for computer graphics is value noise. Conceptually, this is produced by randomly sampling a white noise function and then using a reconstruction filter to interpolate between the samples. Lewis's sparse convolution noise [7] is one such example. For efficiency, most implementations use samples taken along a regular lattice and a simple interpolating reconstruction filter. Each lattice vertex within the range of the reconstruction filter's support around the input point is mapped to a sample value by hashing its coordinates, and then these values are interpolated at the input point to compute the noise function's value. A good hash function can provide a very large volume of noise without obvious periodicity, while reducing memory capacity requirements. Value noise is simple to understand and can be efficient for low-order interpolants. However, it tends to suffer from a blocky, anisotropic appearance (Figure 2a), even with a more expensive higher-order interpolant.

To overcome this, Perlin [11, 16] introduced gradient noise. Instead of using the product of the reconstruction filter with a random scalar value at each sample point, the product of the filter with a randomly oriented linear gradient is employed. The lattice coordinates are hashed to a unit vector, and the dot product of this vector with the vector from the lattice point to the input point is used to multiply the value

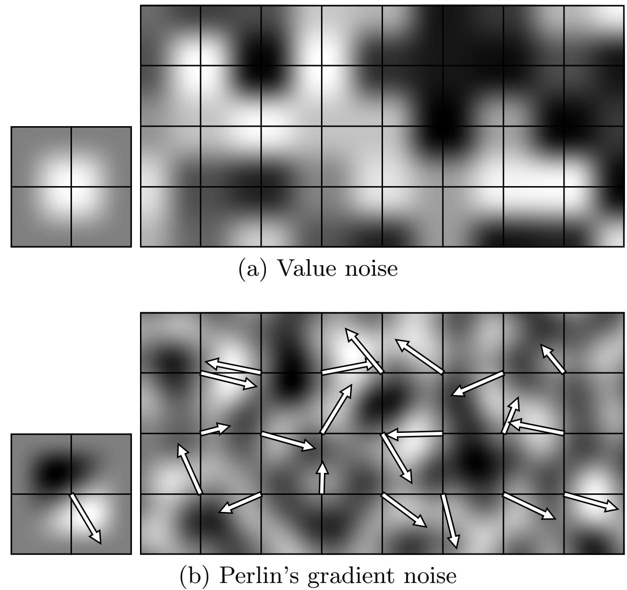


Figure 2: Comparison of (a) value noise, with (b) Perlin's gradient noise. To the left is the the reconstruction filter and an example of an oriented dipole, respectively. Gray represents zero, white indicates positive, and black is used for negative values. The grid overlay shows unit lengths in texture space. Arrows indicate gradient directions.

from the filter. This effectively creates a set of overlapping, randomly oriented dipole functions ("surflets" in Perlin's terminology). Though the vector operations increase the computational complexity, the gradients eliminate much of the blockiness and a narrower filter over fewer samples can be used. (Figure 2b)

In 2001, a more hardware-amenable noise function was developed by Perlin [13] that introduced a variant of gradient noise known as simplex noise. Traditional implementations of Perlin noise interleaved the sampling and reconstruction steps via interpolation. With simplex noise, Perlin made the noise more isotropic by substituting a radial reconstruction filter for the previous separable one. The second change was to switch from a cubic lattice to a simplex lattice, thereby reducing the number of sample points evaluated during the reconstruction. The simplicial grid is also far more efficient for higher-dimensional noise. Finally, he introduced the idea of only using -1, 0 and 1 as components of the gradients in order to eliminate the multiplications in the dot products. However, this reduction in samples and simplification of the gradient calculations gives simplex noise a noticeably different visual quality.

In 2002, Perlin [14] returned to a more traditional noise function (with cubic grid and separable filter) with small adjustments to the interpolant and a clarification of simplified gradients from simplex noise. First, he switched to a higher-order polynomial for the interpolant in order to improve the appearance when Perlin noise is used for displacement mapping. Second, by changing from a table of random unit vectors to a set of vectors based on the midpoints of the edges of a cube, he further reduced the effective number of gradients to twelve. The regular distribution also eliminates the problem of clumping. An alternate solution would have been

to apply a relaxation algorithm to the randomly generated unit vectors as a preprocess [15].

Cook and DeRose [3] noted that Perlin's noise still had several flaws and introduced an alternative, wavelet noise, to overcome these issues. The essence of their algorithm is to initially create an image of random noise, down-sample to half size, up-sample back to full size, and then subtract this result from the original. To evaluate the noise at a point, they filter the image with a uniform quadratic B-spline, in a process similar to evaluating value noise. The initial construction of wavelet noise produces tighter band limits in its frequency distribution, both at the lower and at the upper limits. The result is orthogonal bands that allow for better spectral control. They also note the Fourier slice theorem is responsible for low frequencies "leaking" in when evaluating 2D slices embedded in a higher 3D noise volume. They solve this problem with a modification to the filtering step based on projection along the surface normal.

1.2 Noise in Hardware

There has been some work published on hardware noise implementations [5, 9, 15]. This work does not actually propose special hardware for computing noise, instead it describes details to implement Perlin style noise using GPUs by mapping the lookup tables to texture memory. They are software adaptations of a noise algorithm to run on GPU hardware. While this approach is useful, it is quite different from our approach to hardware noise.

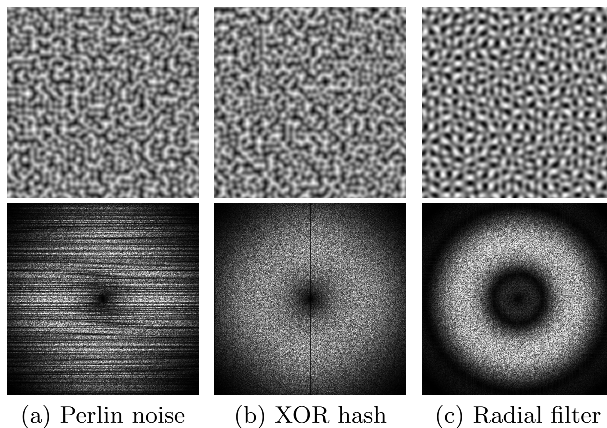
Our noise implementation is an actual parallel hardware implementation of noise as a custom circuit for use as a co-processor or as a functional unit to be included in future designs. While this approach does not leverage existing high performance architectures like GPUs, it does have the potential to be used in GPUs to further increase the performance and quality of these kinds of computations.

As described in Section 1.1 Perlin designed his simplex noise to be more amenable to hardware acceleration by using a set of twelve fixed vectors with unit components to reduce the computation needed for the dot products. While this does reduce the required computation somewhat, it also creates some additional artifacts in the image that are quite apparent and which we wanted to avoid in our noise algorithm.

2. IMPROVED GRADIENT NOISE

In previous work on software noise algorithms [6], we noted that the wavelet noise algorithm, while improving on the spectral characteristics, displayed marked anisotropy. To be efficient, wavelet noise also requires significantly more memory to store the complete pre-processed noise volume, whereas Perlin noise relies on a simple hashing scheme to generate the volume. We introduced a set of modifications to the Perlin noise algorithm [14] to improve the spectral characteristics to match the benefits of wavelet noise.

The first change is to the hash function. Perlin noise normally uses a hash function, $H_{i,j_k} = P[P[P[i] + j] + k]$ to hash coordinate i, j, k on the integer lattice to an index into the gradients. Here, P is a table of integers $0 \dots (N-1)$ randomly permuted and the table lookups are modulo N. This produces striations in the Fourier transform of the noise due to periodicity (Figure 3a): the same sequence of gradients will be used along k, simply shifted depending on i and j. To solve this, we changed the hash function to use a separate



(a) Perlin noise (b) XOR hash (c) Radial filter

Figure 3: Detail of images and associated Fourier transforms of 2D versions of (a) standard Perlin noise, (b) Perlin noise using our XOR hash function, and (c) noise with our new broader radial filter and coarser grid sampling.

permutation table for each dimension and then exclusive-or the values from each: $H_{i,j,k} = P_x[i] \oplus P_y[j] \oplus P_z[k]$ (Figure 3b). This has the added benefit of eliminating dependent lookups.

The second change is to the filter kernel. By multiplying the gradients (dot products) with a broader radial filter, $s(x) = 4(1 - x^2/4)^5 - 3(1 - x^2/4)^4$, where x is the magnitude of the vector from the texture coordinate to the lattice sample point, we achieve tighter bandlimits on the noise. This radial filter, together with a larger table of gradients additionally makes the noise more isotropic. The wider domain, $x \in [-2, 2]$, does carry additional computational cost by requiring a $4 \times 4 \times 4$ stencil for the evaluations. Sampling the grid at only half the frequency (at the even numbered lattice coordinates) reduces this back to the $2 \times 2 \times 2$ evaluations of regular Perlin noise. While slightly detrimental to the direct visual appearance, this more efficient but coarser sampling retains most of the characteristic spectral benefits (Figure 3c). For actual use in most textures, they are largely indistinguishable.

A third improvement is to return to the use of a true table of gradients. Perlin's reduced set of gradients [13, 14] is fast and has the advantage of eliminating clumping in the distribution of the gradients, but can still lead to a strong statistical bias when taking axis-aligned planar slices of a 3D noise volume. Even without this bias, these gradient vectors are prone to producing artifacts in the form of runs at 45° angles. Instead, we use a larger table with Perlin's relaxation idea [15] to solve the clumping problem. In our implementation, each vector in the table is treated as a point charge confined to the surface of a unit sphere [4] which moves about until it reaches equilibrium. This preprocess produces a set of vectors which evenly cover the sphere.

We have used this modified gradient noise algorithm as the basis for our hardware noise implementation. We implemented the new hashing scheme, the new radial polynomial with the coarser grid evaluation, and generated the vectors in the gradient tables with the relaxation technique.

3. HARDWARE GRADIENT NOISE

As a data point, one straightforward implementation of

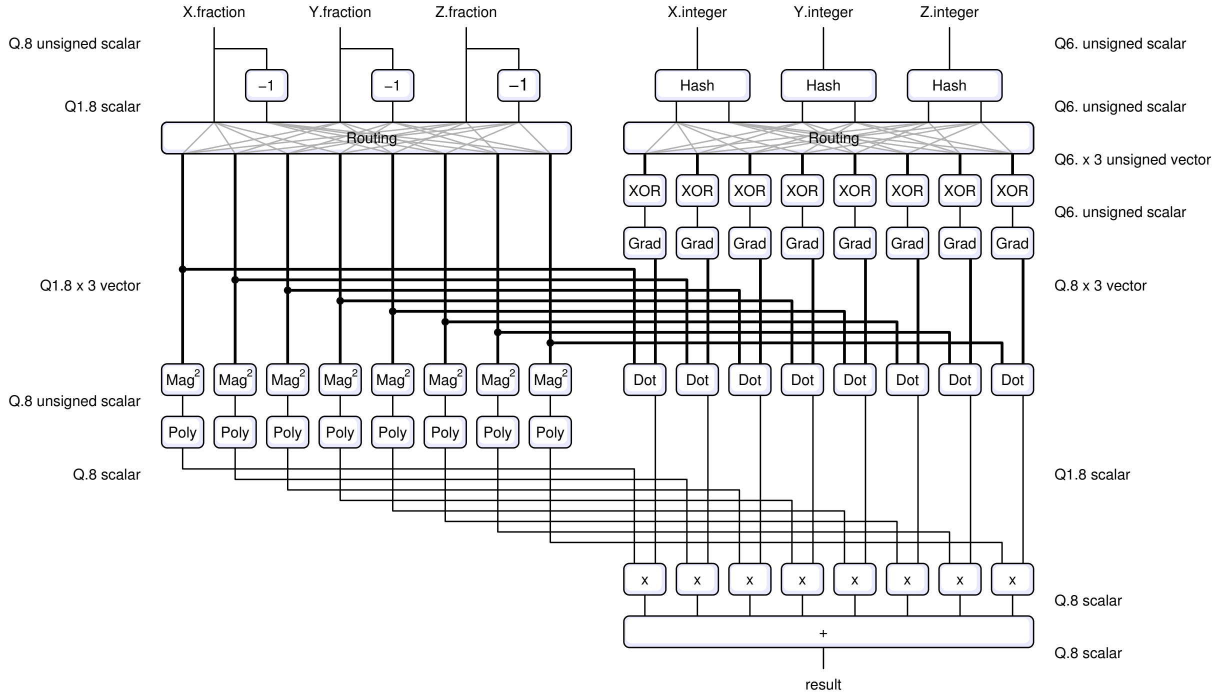


Figure 4: High-level diagram of our noise implementation. Thick lines indicate vector values. All values are signed (two's complement form with an implicit additional high bit) unless noted as unsigned. The left side illustrates the computation of the polynomials on the squared magnitudes of the vector offsets. The right side shows hashing the integer lattice coordinates to lookup the gradient vectors for the dot products before combining these with the polynomials to produce the final value.

Perlin noise that we have measured requires 120 floating point operations to compute one noise value. Our higher quality noise requires as many as 172 operations in software, though many of these operations are independent of each other. Clearly if we can parallelize this process we can achieve a much faster noise implementation. An overview of the parallel hardware implementation of our improved noise algorithm can be seen in Figure 4. As mentioned above, our improved noise uses a gradient table to provide better results than either simplex noise or value noise. This improvement in quality comes at a cost of increased circuit size (to hold the table and to perform the full dot products). We believe it is a good trade off considering the improvement in quality. Additionally, we use reduced precision fixed point arithmetic to save area, energy and delay. We also achieved additional savings by reducing the sizes of the gradient and hash tables. As can be seen from Figures 1 and 5, our fixed point implementation, while distinguishable as being a different image, does not have a noticeable difference in quality from the standard floating point implementation used in software.

For our circuits we used standard cells from Artisan targeted to a 65nm CMOS process. We used Synopsys Design Compiler as a synthesis front end and Cadence SOC Encounter for back-end place and route. For table comparisons between standard cells and ROMs we used Artisan via-ROM generators for the same 65nm CMOS process.

The typical application for noise is generating images with an 8-bit representation per color channel. This allows us to use much lower precision in our circuits than would be used in a traditional software implementation of noise. Even when the software version, or the GPU version, of noise does the full computation in 32-bit floating point most of the precision is thrown out in the end. We leverage this to use 8-bit fixed point computations in our noise circuits which results in tremendous savings in area, power, and latency with no discernible visual artifacts. While multiplication operations do lose some precision due to truncation of the lower bits, the deepest chain of arithmetic operations contains just five multiplies.

3.1 Lookup Tables

There are a number of lookup tables used in our improved noise algorithm (permutation tables and gradient tables) and we explored a few options for implementing them. We first used the ROM generator, but the results required a fairly large area. We found that by implementing the lookups as case statements in Verilog and synthesizing to standard cells we were able to decrease the area needed for the lookup tables by a factor of 3.3. The latency through the ROM was also worse and forced us to latch the value at the output instead of allowing us to perform register retiming through the lookup tables. We believe this is because the lookup tables we are using are at the smallest possible size for the ROM generator, and the amortized ROM overhead is relatively large.

We also analyzed the tradeoff between a full 256 entry gradient table described in Section 2 and a much smaller 64 entry gradient table which we believe is the smallest size that generates results that are visually indistinguishable from the larger table sizes. Because we replicate this table eight times to allow for parallel lookups, the area savings are considerable. We opted to shrink the table rather than pipeline the

Figure 5: Example scene from Figure 1 rendered with a software implementation of our modified noise algorithm. All noise evaluations were performed with 8 fractional bits and a 64-entry gradient table. While not identical, this shows that the modified algorithm with reduced precision can be a viable alternative to floating point Perlin noise.

lookups in the table for simplicity and parallelism. In addition, this reduction in the size of the gradient tables allowed us to shrink the bit width of the 256 entry hash tables as we only need six bits to find the gradient. In each case, the gradient values used in our tables were generated using the point repulsion technique described in Section 2.

3.2 Pipelining

Our initial design was entirely combinational where it was assumed that a full computation of noise would be performed in a single cycle. We compared that design to a pipelined design with up to four stages and found that we could meet our goal of 1GHz frequency with four pipeline stages where the non-pipelined version would only reach 344MHz. The pipelined version was generated by first synthesizing the entire combinational circuit to meet the combinational timing requirement. Design Compiler was then run on the synthesized circuit to perform register retiming and distribute the registers throughout the circuit resulting in a pipelined implementation. Register retiming can create circuits of very different sizes depending on where the registers are placed in the final retimed version.

To explore the design space we synthesized a few different pipeline depths as part of our design process. The results of these synthesis runs are detailed in Table 1. While we were also able to achieve a 1GHz clock frequency with a three stage pipeline, the size of the three stage design was larger than the four stage design because a larger combinational circuit is needed to meet the timing requirements. We therefore chose the smaller design since throughput is more important than latency for this application.

Table 1: Cell area and performance numbers are from synthesis before place and route.

Pipeline stages	Clock	Cell Area (μm^2)		
		Comb	Seq	Total
0 (combinational)	344MHz	60,350	0	60,350
1 (not retimed)	337MHz	57,052	2,470	59,522
3 (retimed)	1GHz	76,256	11,980	88,236
4 (retimed)	1GHz	58,090	11,470	69,560

3.3 Physical Implementation

Figure 4 shows the computational flow of information for a single noise calculation. A three-space point (vector) is input and the result is a single noise value which is used in the shading computations. From this diagram the parallelism in the algorithm is apparent, and would be difficult to exploit in software. The thick lines in the diagram are vectors (typically 3 elements of 8 bits each) and the thin lines are single 8-bit values. The boxes labeled **Hash** and **Grad** are the hash lookups and the gradient tables described in Section 3.1. Section 2 also describes the polynomial operation (radial filter) performed by the **Poly** boxes.

The arithmetic implemented in the magnitude, dot product, and polynomial computations, as well as the multipliers and adders, is all fixed point. The range of values for the fractional inputs is in the range $[0, 1]$ and only the most significant bits really matter, which is why fixed point is sufficient and beneficial to our design. Our design retains all the bits needed to encode the exponent in a floating point number and also results in smaller circuits because of the fixed point representation. The arithmetic circuits were described with behavioral verilog and synthesized with Synopsis.

While this design was not fabricated, the final circuit after synthesis and place and route can be seen in Figure 6. The size of this final layout is 105kμm^2 (325μm × 325μm). For comparison, and also in the context of the TRaX processor, we produced other more well-known circuits using the same standard cells and the same 65nm technology. A single-precision floating-point 3-element dot product takes an area of 111kμm^2. A double-precision floating-point multiplier consumes 73kμm^2, while a single-precision floating-point ALU (performing add, subtract or multiply) uses 34kμm^2 in this technology.

Our circuit is smaller than a single-precision dot product, despite containing eight dot product operations because we use lower precision arithmetic in our circuit. While this reduced precision is not appropriate for every potential application of noise, we believe it is sufficient for our intended application of shading in graphics. It would also map well to any other application that ends up truncating the precision when using the results of the noise.

4. CONCLUSION

Our improved noise algorithm results in high quality noise that avoids the downfalls of periodicity, anisotropy, and aliasing. This functional unit performs this operation quickly and requires only a relatively small die area. We reached our target of a 1 GHz clock frequency with a four stage pipelined design which produces one noise value per clock once the pipeline is full. This can be compared to a straightforward software implementation of Perlin noise which requires 120 floating point operations and peaks at 16.7M evaluations

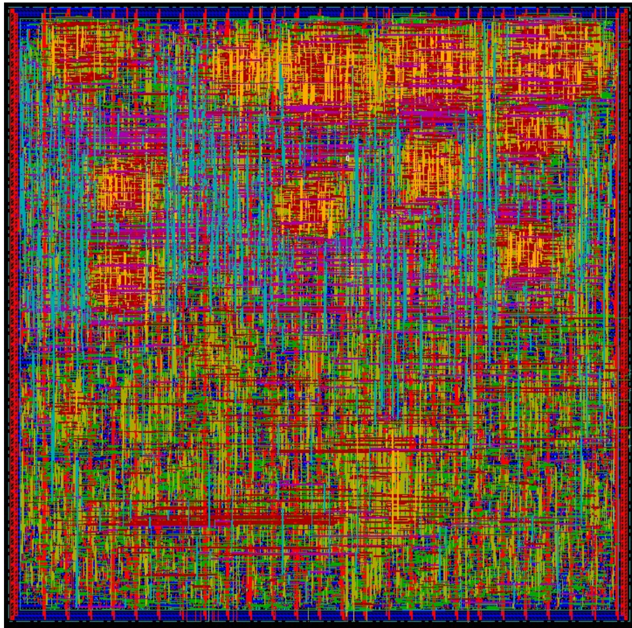


Figure 6: Placed and routed circuit implementing our improved noise function as a four-stage pipeline (105kμm^2). This image is a screen capture from Cadence SOC Encounter and shows only metal routing layers.

per second on a single core of 2.8GHz Core 2 Duo. Our final design uses three 256 entry hash tables where, to avoid additional adders, each table entry encodes the hash value for the input, and for the input + 1 (see Figure 4). We also use eight copies of a 64 entry gradient table, where each gradient is a three element vector of fixed point values.

As graphics pipelines demand more and more memory bandwidth we believe that providing a method for high quality textures through a hardware accelerated noise function provides a good trade-off. Much of the bandwidth of high-performance graphics chips is devoted to image-based (look-up) texturing. Procedural textures using noise offer an alternative that trades memory bandwidth for computation. The scene in Figure 1 is an example that uses an average of 552 calls to the noise function per shading sample. 37.2% of the total execution time for rendering the image was spent in the evaluation of noise for various aspects of the image. The textures on all of the surfaces and the smoke use noise to improve visual quality. The use of image-based textures would require far more memory bandwidth than our approach.

Admittedly, many applications would see more modest improvements in performance than the specific scene used here which is designed to demonstrate heavy use of noise-based textures. However, any time noise is used there would be a speedup using our hardware over a software implementation. At least one place where this could encourage visually complex images at a reduced memory bandwidth requirement would be video games. Games typically use very large image textures to avoid the appearance of repetition. While we do not have specific projections of memory bandwidth savings, it is well known that the large image textures are a significant fraction of the memory bandwidth in video games. Our design could increase the performance of applications that use noise by as much as 50% and would be a good step toward high quality procedural texture generation and could become a viable real-time alternative to image-based texturing.

5. REFERENCES

[1] A. A. Apodoca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers, 2000.

[2] R. Bridson, J. Hourihan, and M. Nordenstam. Curl-Noise for Procedural Fluid Flow. *ACM Transactions on Graphics (SIGGRAPH '07)*, 26(3), 2007.

[3] R. L. Cook and T. DeRose. Wavelet Noise. *ACM Transactions on Graphics (SIGGRAPH '05)*, 24(3):803–811, 2005.

[4] T. Erber and G. M. Hockney. Equilibrium Configurations of N Equal Charges On a Sphere. *Journal of Physics A: Mathematical and General*, 24(23):L1369–L1377, 1991.

[5] S. Gustavson. Simplex Noise Demystified. In *http://webstaff.itn.liu.se/∼stegu/simplexnoise/*, 2005.

[6] A. Kensler, A. Knoll, and P. Shirley. Better Gradient Noise. Technical Report UUSCI-2008-001, SCI Institute, University of Utah, 2008.

[7] J. P. Lewis. Algorithms for Solid Noise Synthesis. *ACM SIGGRAPH Computer Graphics*, 23(3):263–270, 1989.

[8] F. K. Musgrave. Fractal Solid Textures: Some Examples. In *Texturing and Modeling: A Procedural Approach*, chapter 15, pages 447–487. Morgan Kaufmann Publishers, third edition, 2003.

[9] M. Olano. Modified Noise for Evaluation on Graphics Hardware. In *Proceedings of Graphics Hardware 2005*, pages 105–110, 2005.

[10] D. Peachey. Building Procedural Textures. In *Texturing and Modeling: A Procedural Approach*, chapter 2, pages 7–94. Morgan Kaufmann Publishers, third edition, 2003.

[11] K. Perlin. An Image Synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, 1985.

[12] K. Perlin. In the beginning: The Pixel Stream Editor. In M. Olano, editor, *Real-Time Shading SIGGRAPH Course Notes*, chapter 2. 2001.

[13] K. Perlin. Noise Hardware. In M. Olano, editor, *Real-Time Shading SIGGRAPH Course Notes*, chapter 9. 2001.

[14] K. Perlin. Improving Noise. *ACM Transactions on Graphics (SIGGRAPH '02)*, 21(3):681–682, 2002.

[15] K. Perlin. Implementing Improved Perlin Noise. In *GPU Gems*, chapter 5, pages 73–85. Addison-Wesley, 2004.

[16] K. Perlin and E. Hoffert. Hypertexture. *ACM SIGGRAPH Computer Graphics*, 23(3):253–262, 1989.

[17] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Natick, MA, 2003.

[18] J. Spjut, D. Kopta, S. Boulos, S. Kellis, and E. Brunvand. TRaX: A Multi-Threaded Architecture for Real-Time Ray Tracing. In *6th IEEE Symposium on Application Specific Processors (SASP)*, 2008.

[19] T. Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.