Hash Functions for GPU Rendering

Mark Jarzynski
UMBC

Marc Olano
UMBC

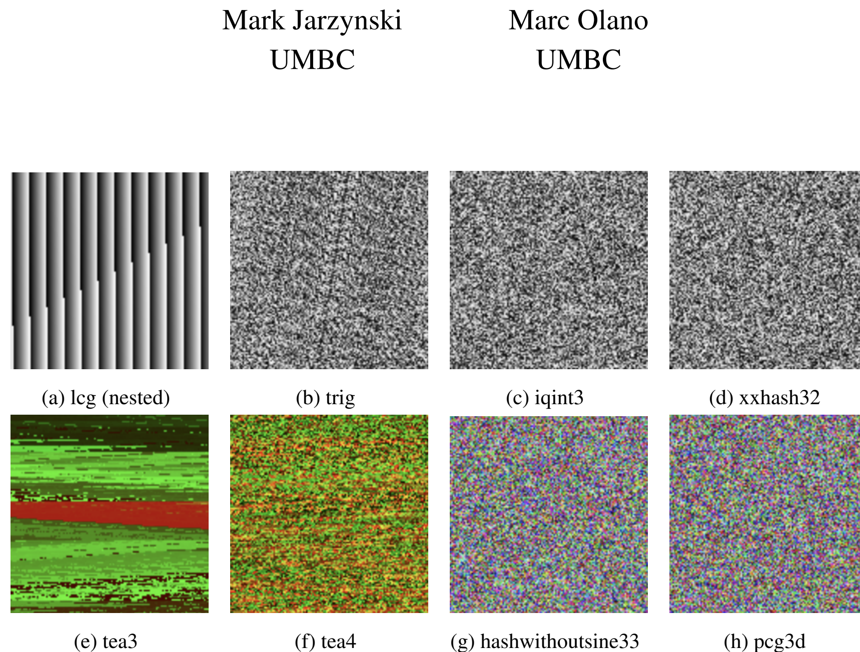


Figure 1. 2D plot of hash results, providing visual evidence of certain types of errors. (a) `LCG` and (b) `trig` are obviously bad, with visible banding, linear artifacts, and repeated patterns. (c) `iqint3` and (d) `xxhash32` are better, though this plot does not show that `xxhash32` is significantly better quality. (e) `tea3` and (f) `tea4` have 2D output, and (g) `hashwithoutsine33` and (h) `pcg3d` have 3D output, all four shown as color.

Abstract

In many graphics applications, a deterministic random hash provides the best source of random numbers. We evaluate a range of existing hash functions for random number quality using the TestU01 test suite, and GPU execution speed through benchmarking. We analyze the hash functions on the Pareto frontier to make recommendations on which hash functions offer the best quality/speed trade-off for the range of needs, from high-performance/low-quality to high-quality/low-performance. We also present a new class of hash tuned for multidimensional input and output that performs well at the high-quality end of this spectrum. We provide a supplemental document with test results and code for all hashes.

 ISSN 2331-7418

1. Introduction

Random number generation comes up in a variety of contexts in 3D rendering. Research into pseudo-random number generators (PRNGs) is almost as old as computer science [Knuth 1978]. The goal of PRNGs is to provide a deterministic sequence of numbers that are indistinguishable from a true uniform random process. It is important that the generation of sequential random numbers be as fast as possible and that it be possible to seed the generator to produce the same deterministic sequence on subsequent runs of a program. It generally is not important that traditional PRNG results appear random given adjacent seeds and most do not. It is also not particularly important to be able to access a number from the sequence out of its sequential order; some PRNGs allow this while others do not.

Unfortunately, both of these last two conditions are requirements for many graphics applications, due to their need for both spatial locality and parallel execution. For example, producing a random value per pixel in a GPU shader requires both that nearby pixels be random relative to each other, and that each pixel should be able to determine its value without first evaluating the PRNG for every other pixel with a lower pixel index. A sequential program might be able to loop over all the lower-indexed pixels to use a PRNG, but a GPU needs to be able to evaluate all the pixels independently, with each pixel remaining sufficiently random with respect to the others.

For many graphics uses, a random hash is a better match than a PRNG. Perlin noise uses a hash of several grid points close to the query point [Perlin 1985]. Unlike PRNGs, random numbers based on location or other characteristics of the geometric data are independent of the number and distribution of threads or order of evaluation. Other applications use a hash of position, particle or object ID, or UV coordinate [Wyman and McGuire 2017; Phillips et al. 2011; Wei 2004]. Random hashes need to execute quickly on the GPU, need to be able to execute in parallel, need to be sufficiently random when invoked with adjacent seeds, and often need to handle multidimensional inputs and multidimensional outputs. Also, some graphics applications need high-quality random numbers, while others such as Perlin noise achieve acceptable visual results even with lower-quality random values. As such, it is important to understand the performance and quality needs of the application, and the performance and quality characteristics of available hash functions.

Cryptographic hashes have many of the desired characteristics. If a cryptographic hash of sequential values deviates in any detectable way from independent, uniformly distributed random data, the correlation between values could be used as the basis for an attack. Several researchers have used cryptographic hashes as a source of random numbers for GPU and graphics code [Tzeng and Wei 2008; Zafar et al. 2010]. Unfortunately, though their quality is excellent, as a protection against brute-force attacks cryptographic hashes are designed to be slow.

Hashes for data structures are designed to be fast, since the insertion time is directly affected by the evaluation speed. However, even distribution in a hash table is not the same goal as having a uniform random distribution. As our results show, hashes designed for data-structure use tend not to produce high-quality random numbers when using adjacent seeds.

Many commonly used hash functions only accept one integer as input and output another integer. We refer to these as one-dimensional or $(1 \to 1)$ hashing algorithms. Many graphics applications require a hash function with vector input and/or vector output, for example a 2D pixel-coordinate input with a 3D random-vector output. A hashing algorithm with N-dimensions of input and M-dimensions of output would be $(N \to M)$.

To know if a hash is a good fit for real-time GPU random-number generation, it is necessary to measure both its quality and evaluation speed. Others have performed these tests using the Diehard, SMHasher, or TestU01 test suites for quality comparison [Zafar et al. 2010; Collet 2012; O'Neill 2014a]. However, many graphics uses require multidimensional inputs to the hash (typically 1D–4D), and multidimensional outputs (1D–4D). The best hash for use with 1D input and 1D output may not be the best for 3D input and 3D output. In addition, many of the GPU hashes in use today have only appeared in blog posts or shadertoy code, and have never been formally evaluated or compared to other alternatives. This leaves anyone looking for a good GPU hash with little guidance to decide which to use. We evaluate over 100 hash variants, in terms of quality and speed for random number generation, and propose a class of new hashes with better quality for 3D and 4D use cases. Based on our data, we make recommendations for hash use across the speed/quality spectrum.

2. Background

Tests to compare the quality of random number generators typically perform experiments using the generated values and evaluate whether differences from the known expected result could be due to random chance. A first set of four tests was proposed by Kendall and Smith [1938]. Since then, several more tests have been developed and released in test suites such as Diehard, Statistical Test Suite (STS), Dieharder, TestU01, and PracRand [Marsaglia 1995; Bassham et al. 2010; Brown 2017; L'Ecuyer and Simard 2007; Appleby 2008; Doty-Humphrey 2019]. These test suites are used to compare the quality of sets of random numbers, and thus can be used to compare the randomness quality of hashing functions. For example, O'Neill [2014a] used the TestU01 suite to compare random number generators and hash functions.

SMHasher [Appleby 2008] performs similar tests for data-structure hashes, evaluating the distribution of the hash across a hash table and its probability of collisions.

A less rigorous test often used for graphics purposes draws the random values as grayscale (or extracted single bits) into a 2D image [Zafar et al. 2010; Reed 2013]. This can show visual repetition and patterns, though not deeper statistical anomalies detected by the more sophisticated tests (See Figure 1).

When performing a multidimensional optimization, the *Pareto frontier* contains solutions where improving one dimension will make the other(s) worse. For hashes where we care about both performance and quality, if a hash is on the Pareto frontier, the only faster hashes will have worse quality, and the only higher-quality hashes will be slower. For hashes off of the frontier, there is another hash in the set that will improve at least one of the criteria without making the other criteria worse. Therefore, we are primarily interested in identifying hashes that define the speed/quality Pareto frontier.

The list of base hash functions we compare is given in Table 1. We also compare several derived hashes created by applying the transformations in Sections 4.1–5.

3. Methodology

3.1. Testing Quality

Following O'Neill [2014a], we use the total number of TestU01 [L'Ecuyer and Simard 2007] BigCrush tests that failed as a measure of hash quality. BigCrush consists of 160 individual tests: a result of 0 means that the hash passed every test, and a result of 160 means that the hash failed every test. The PractRand [Doty-Humphrey 2019] provides more thorough testing for high-quality random number generators, but since it runs with incrementally increasing test sizes and stops at the first failure, it appears to provide less useful discrimination at lower quality levels.

To accelerate testing, we parallelized each BigCrush test on our high performance computing facility cluster. To ensure consistency between CPU code used for randomness testing, and GPU code used for timing, we created C++ classes and functions to allow the HLSL hash code to be compiled as valid C++ code without change. The input for each hashing algorithm was initialized to be an unsigned integer with a value of 0, incremented each time that the hashing algorithm was called.

TestU01 expects a sequential stream of integers as the output of any random number generator to be tested. For hashes that had a vector as the output, each value was serialized. For example, a hash with three-dimensional output would produce X, then Y, then Z for one index before incrementing the index. Since the randomness tests include testing for translational repetition, this method will catch problems in any single channel as well as between channels. We did try testing each component independently, but the results did not significantly differ from rotating through the output dimensions.

Name	Shortname	Reference
Blum Blum Shub	bbs	[Blum et al. 1986]
CityHash	city	[Pike and Alakuijala 2011]
hash	esgtsa	[Schechter and Bridson 2008]
RandFast	fast	[Epic Games 2016]
Hash without Sine	hashwithoutsine	[Hoskins 2014]
Hybrid Taus	hybridtaus	[Howes and Thomas 2007]
Interleaved Gradient Noise	ign	[Jimenez 2014]
Integer Hash - I	iqint1	[Quilez 2017a]
Integer Hash - II	iqint2	[Quilez 2017b]
Integer Hash - III	iqint3	[Quilez 2017c]
JKISS32	jkiss32	[Jones 2010]
Linear Congruential Generator	lcg	[Lehmer 1949; Press et al. 2007]
MD5GPU	md5	[Tzeng and Wei 2008]
MurmurHash3	murmur3	[Appleby 2008]
PCG	pcg	[O'Neill 2014a]
PCG2D	pcg2d	[This paper]
PCG3D	pcg3d	[Epic Games 2016]
PCG4D	pcg4d	[This paper]
PseudoRandom	pseudo	[Epic Games 2014]
Ranlim32	ranlim32	[Press et al. 2007]
SuperFastHash	superfast	[Hsieh 2004]
Tiny Encryption Algorithm	tea	[Wheeler and Needham 1995]
Trig	trig	[Rey 1998]
Integer Hash Function	wang	[Wang 2007]
Xorshift	xorshift32	[Marsaglia 2003]
Xorshift	xorshift128	[Marsaglia 2003]
xxHash	xxhash32	[Collet 2012]

Table 1. Hashes tested, short names used in plots, and the reference for each.

For multidimensional input, a Morton (or Z) ordering [Morton 1966] was used to translate the single integer-sequence order of TestU01 into a vector of integers for hash input. Since Morton ordering interleaves the bits of the vector dimensions, the TestU01 translational repetition testing will catch repetitions in any of the dimensions.

3.2. Testing Performance

To test run-time performance, we measured execution time as reported by the GPU profiler in Unreal Engine 4.19 (UE4) for a single screen-filling quad at 1280×720 pixel resolution. For every hash tested, a material was created and marked as translucent, using an opacity of 1, such that no translucency was actually used. During each test, one material is applied to the quad. Since the quad was the only polygon in the scene with a material that was marked translucent, its sub-frame render-

ing time appears alone in the timestamp-query-based UE4 GPU Profile performance-measurement tool without needing to make any modifications to the core engine code.

In order to get measurable results, we executed each hash 10,000 times, feeding the output of each iteration as the input to the next, so the GPU and compiler would not optimize out any instructions. All timings reported here are the median of five runs, divided by 10,000. Some additional overhead beyond the actual hash call could be counted in the total per call, but only 10^{-5} of that overhead per call, so timings should still be ordered the same as a single call to the hash function.

All timing tests were done on a Windows 10 computer with an NVIDIA GeForce GTX 1080 graphics card with no other applications running. Given the simple, purely computational nature of the code generated for any of these hash functions, the relative performance of our results should still be valid for other current and future GPUs, though we expect there might be some change in the relative performance between

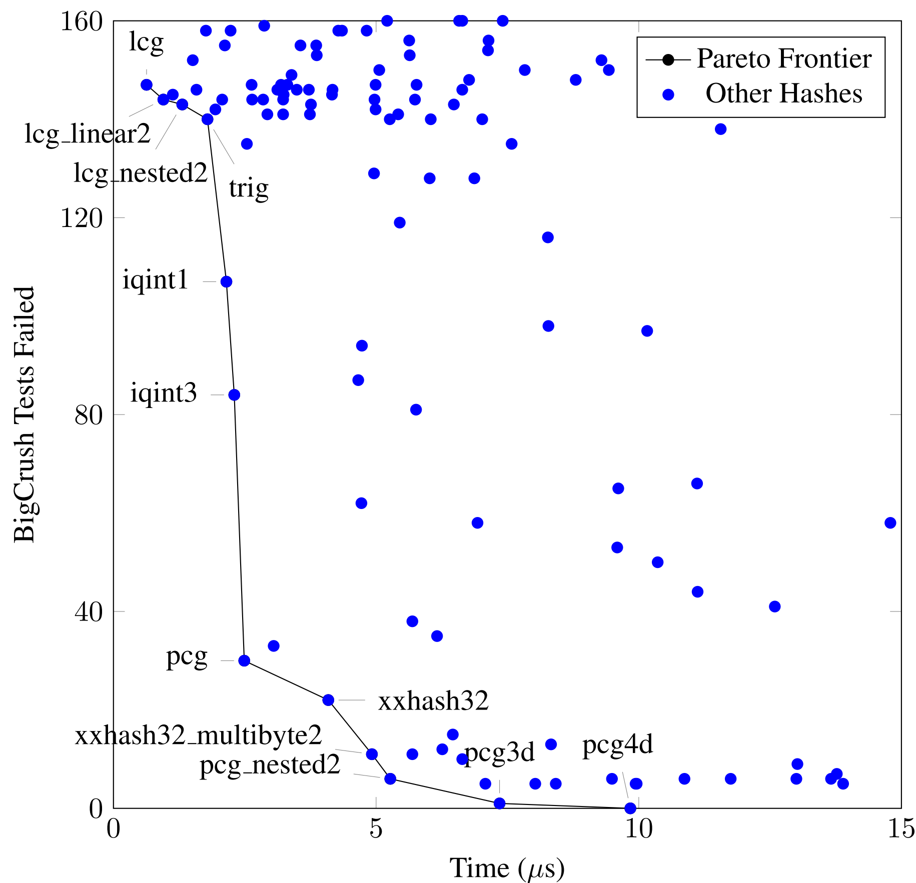


Figure 2. Quality vs. performance for all hashes. Labeled hashes on the black line are in the Pareto frontier, meaning they are the fastest hashes for their quality. Please see the supplementary document for full data and hash code.

integer and floating-point hashes on newer GPUs with faster 32-bit integer operations. As our best performing hashes were already integer, this would not significantly change our results.

Hashing algorithms not originally implemented in HLSL were converted to HLSL. In those cases, though C++ code could have been used for the TestU01 testing, the HLSL code was used for both UE4 and TestU01 for consistency.

Figure 2 shows an overall plot of quality vs. performance of the hashes we tested. Only hashes on the Pareto frontier are labeled, though detailed results as well as code for all hashes are available in the supplementary materials. For a visual comparison of a low-quality and high-quality hash from this set, see Figure 1(a) as compared to (c), and (d).

4. Converting a $(1 \rightarrow)$ Hash to $(N \rightarrow)$

Since many hash functions accept just one dimension of input, several methods have been used in the literature to convert such hash functions to higher-dimensional input.

4.1. Linear Combination

A common way to convert a $(1 \rightarrow)$ hash to an $(N \rightarrow)$ hash is to use a linear combination of the input dimensions, usually with a prime multiplier between dimensions

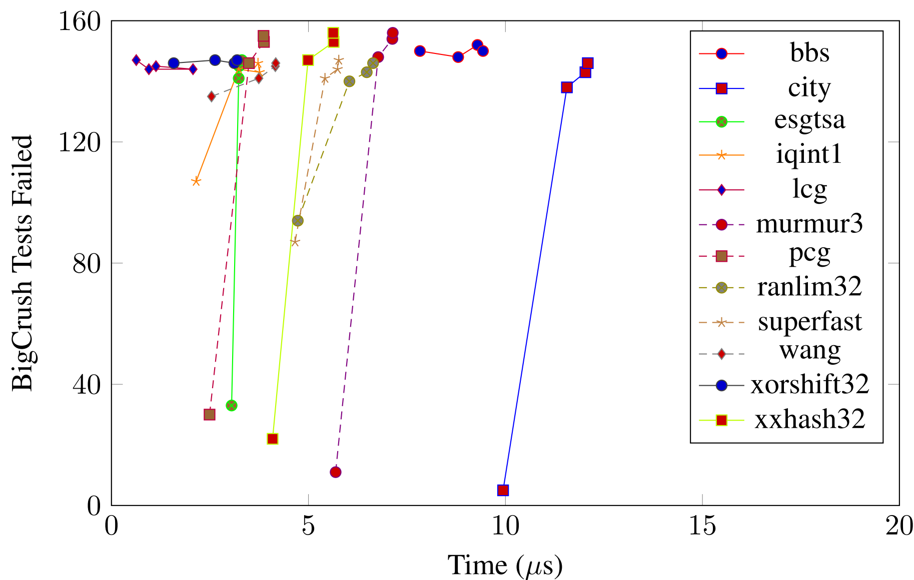


Figure 3. Using linear combination to increase the input dimensions of $(1 \rightarrow)$ hashes. The first point of each line is the normal 1D hash. Each subsequent point increases the number of dimensions of the input combined linearly. This graph shows that linear combination destroys the hash quality for all hashes.

[Hoskins 2014; Quilez 2017a]:

$$\text{hash}(aX + bY + cZ + d).$$

While this method is fast, Figure 3 shows that it greatly diminishes the quality of the hash. This method generates repeating patterns in the hash, though how close they are to each other is dependent on the choice of constants (see Figure 6(a)).

We only recommend this method when speed is the overriding concern and hash quality is not important. The fastest multidimensional hash in our testing set is a linear combination applied to `lcg`, though it fails 90% of the BigCrush tests.

Harnett [2018] used a variant of linear combination with xor instead of addition:

$$\text{hash}(aX \wedge bY \wedge cZ).$$

We tested this method with several $(1 \rightarrow 1)$ hashes and found this method improved low-quality hashes for $N = 2$, but for higher-quality hashes and with a higher number of dimensions, the results only got worse (see Figure 4).

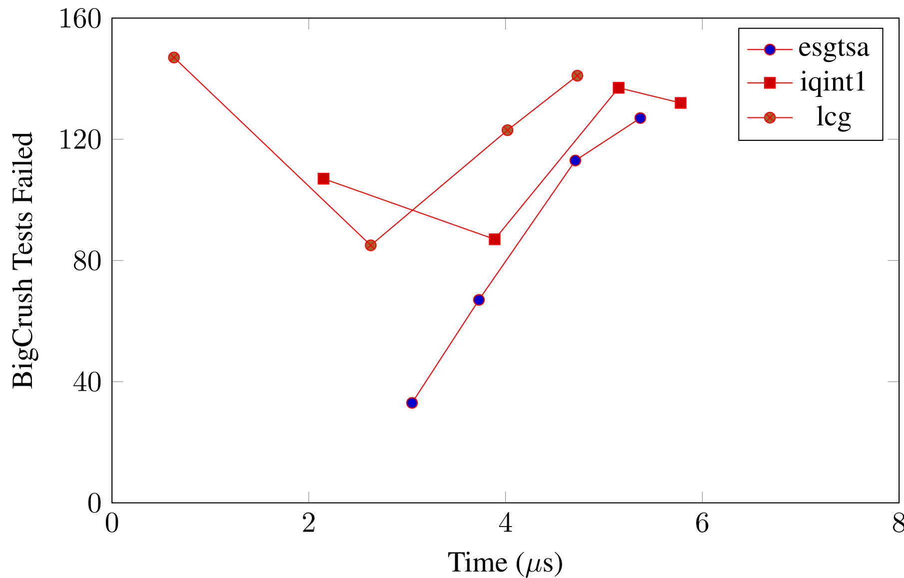


Figure 4. Combining $(1 \rightarrow 1)$ hashes with XOR. The first point of each line is the normal $(1 \rightarrow 1)$ hash. Each subsequent point increases the number of input dimensions.

4.2. Nested

Perlin [1985] converted a $(1 \rightarrow 1)$ hash into $(N \rightarrow 1)$ for Perlin noise by nesting the hash:

$$\text{hash}(X + \text{hash}(Y + \text{hash}(Z))).$$

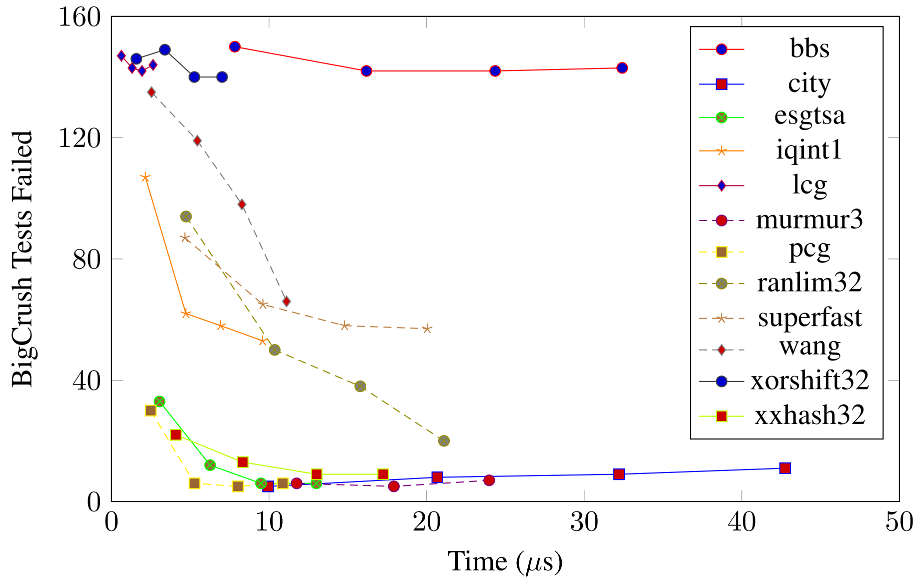
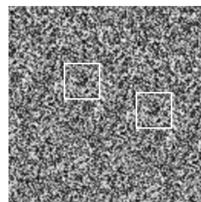
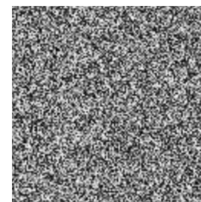


Figure 5. Using nesting to increase the number of input dimensions of $(1 \rightarrow 1)$ hashes. The first point of each line is the normal $(1 \rightarrow 1)$ hash. Each subsequent point increases the number of dimensions of the input by nesting the hash with itself for each dimension. In most cases, the quality of the hash improves when increasing the dimensions of input. The time increases linearly with each additional input. When considering multiple inputs by nesting $(1 \rightarrow 1)$ hashes `pcg` is a good default choice in terms of both quality and performance.

Since this method uses the hash multiple times, it avoids the repetition and quality problems of the linear-combination method. Unfortunately, with the increased quality comes increased computation time, since an N-dimensional input will use the hash N times (see Figure 5). When considering multiple inputs, `pcg` is a good default choice for an $(N \rightarrow 1)$ hash in terms of quality per speed. Using `iqint1` is only slightly faster than `pcg` for middle-range quality. While `lcg` is the fastest hash with this method, one should consider using the linear-combination method instead of the nesting method if quality is not a concern (and consider a different hash than `lcg` if quality is a concern).



(a) 2D linear combination



(b) 2D nested

Figure 6. Comparison of the Wang hash using (a) linear combination, and (b) 2D nested methods. Notice there are repeating patterns in (a), highlighted in the white squares.

Figure 6 shows a visual comparison of linear combination and nesting on the same base hash. The linear-combination method introduces repeating patterns in the hash, while nesting does not.

4.3. Multi-byte

Some hash functions are designed to operate on variable-length input. This includes hash functions designed to operate on strings or for file checksum operations. Since these hashes can handle multi-byte input, they can be used as a $(N \rightarrow)$ hash by feeding all N input coordinates as input. In order to handle the variable length input, these hashes typically loop over the input in multi-byte chunks performing a set of mixing operations at each iteration, with the output of one iteration as one input for the next iteration. After the iterator completes, these hashes then typically execute a final scrambling on output. In our test set, the following hashes are multi-byte, `city`, `murmur3`, `superfast`, `xxhash32`. Figure 7 shows a comparison of these multi-byte hash functions.

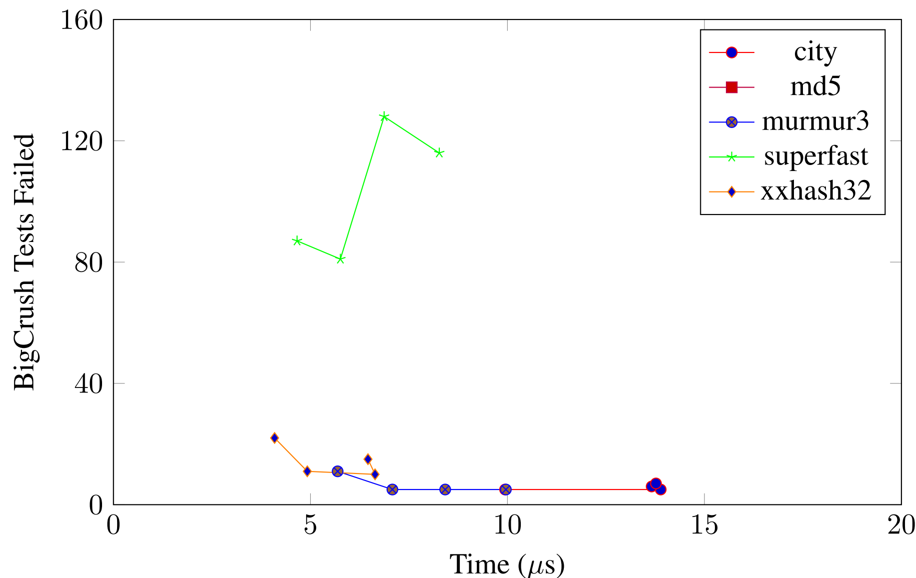


Figure 7. Multi-byte hashes. The first point of each line is the normal $(1 \rightarrow 1)$ hash. Each subsequent point increases the number of input dimensions; `md5` appears off the right edge of the chart.

5. Converting a $(\rightarrow 1)$ Hash to $(\rightarrow N)$

There are many methods to convert a 1D-output hash into an ND-output hash. The most popular is to use a single hash evaluation and apply a single additional `lcg` step, $ah + b$, with different constants to generate each subsequent output. The most

common variants of these use just addition (with $a = 1$), or just multiplication (with $b = 0$).

A common method is one where the hash input is offset by an arbitrary number, such as a prime, for each desired output beyond the first [Ebert et al. 2002]:

$$(\text{hash}(X),\ \text{hash}(X+a),\ \text{hash}(X+b)).$$

We call this method *translated* because, when used in a Perlin noise function, this idiom appears as a translation of the input point.

6. Native Multidimensional Hashes

Multidimensional hashes use vectors for input and output. Typically, the number of inputs and outputs is the same, but this is not always the case. Good quality multidimensional hash functions combine and intertwine their inputs in such a manner

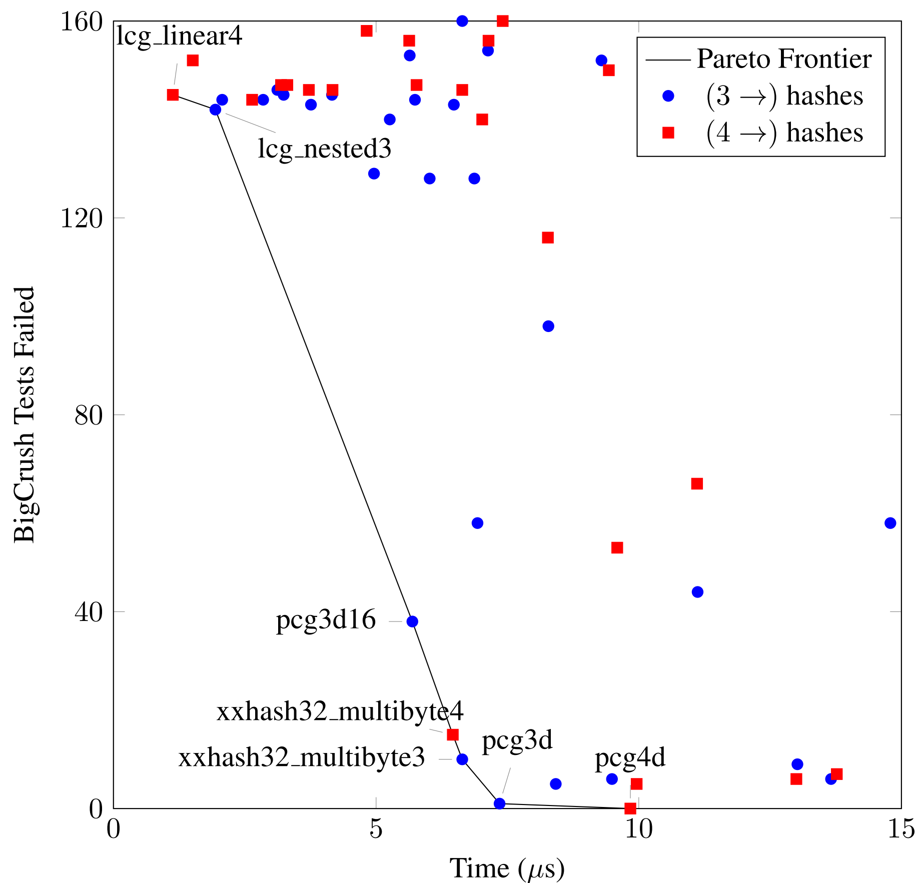


Figure 8. Results for $(3\ \rightarrow)$ and $(4\ \rightarrow)$ hashes. The hashes that fall along the black line (labeled) are on the Pareto frontier, meaning they are the fastest hashes for their quality.

that all components of the output change even if only one component of the input was changed. This means the hash can also be used with fewer dimensions, holding the unneeded input components constant. So, we can use a $(3 \to 3)$ hash function as $(2 \to 3)$ or $(1 \to 3)$, without sacrificing quality, though possibly at a higher execution cost than a lower-dimensional hash.

Figure 8 compares $(3 \to)$ and $(4 \to)$ hashes with any size output. Figure 9 compares only $(3 \to 3)$ and $(4 \to 4)$ hashes with 3D or 4D output. In both cases, `pcg3d` and `pcg4d` fall on the Pareto Frontier and are a good default choice for multidimensional high-quality hash functions; `xxhash32` would be a good default choice for $(3 \to 1)$.

Multidimensional hashes also have the advantage of being able to use fewer dimensions than the hash outputs. Due to dead-expression elimination by the compiler, this may actually decrease the computation time of the hash.

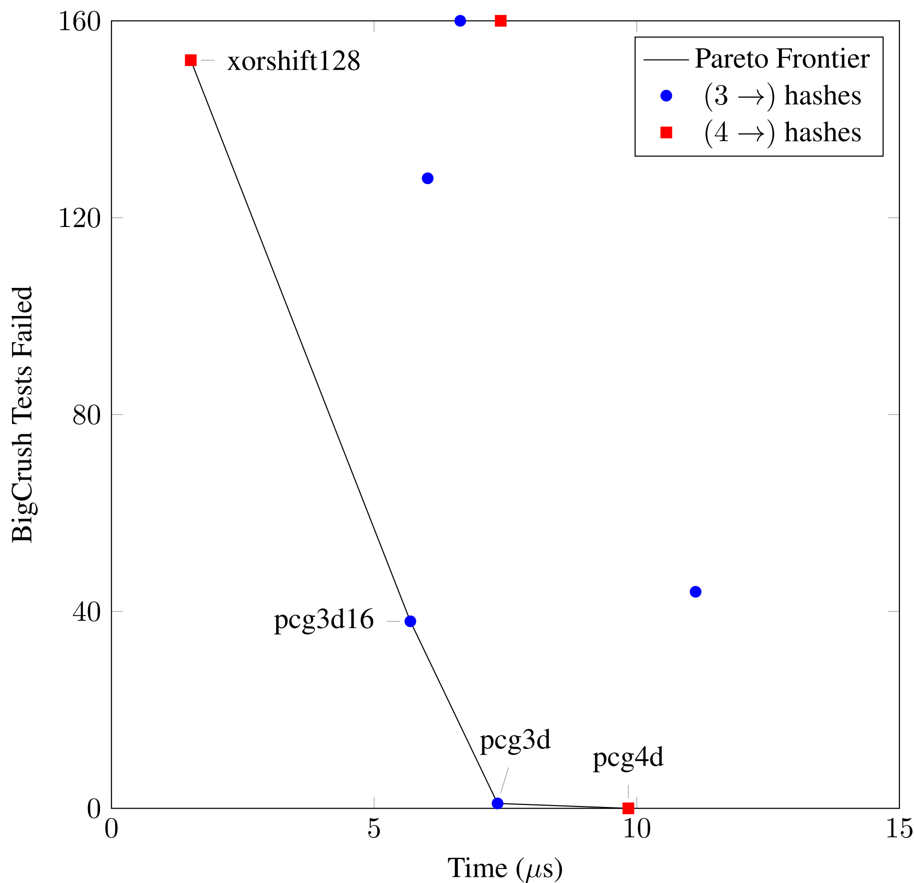


Figure 9. Results for $(3 \to 3)$ and $(4 \to 4)$ hashes. The hashes that fall along the black line (labeled) are among the Pareto frontier, meaning they are the fastest hashes for their quality.

6.1. PCG3D and PCG4D

Given the poor results for existing 3- and 4-input hashes, we present the `pcg3d` and `pcg4d` hashes, which are natively $(3 \rightarrow 3)$ and $(4 \rightarrow 4)$, respectively. These hashes have not been previously published, although an earlier version of `pcg3d` and a faster variant producing 16-bit output (`pcg3d16`) were both originally developed by one of the authors for use in the Unreal Engine [Epic Games 2016].

The design of these hashes is inspired by O'Neill's permuted congruential generator [O'Neill 2014a]. This generator runs a low-quality `lcg` through a permutation function to improve the quality. O'Neill provides significant guidance on what makes a good permutation function. The sample implementation [O'Neill 2014b] uses a permutation combining shifts, xors, and a bit rotation. Instead, we follow the guidelines to create a simple permutation function inspired by generalized Feistel ciphers [Hoang and Rogaway 2010; Sastry and Kumar 2012]. Generalized Feistel ciphers break the input into independent parts and update each part in turn with a reversible operation (typically xor or addition) with an arbitrary (not necessarily invertible) function of the remaining parts. This process is reversible and proceeds for several rounds.

The `pcg3d` and `pcg4d` use a series of multiply-and-add operations as the basic permutation round with a high-to-low bit mixing between rounds. They are effectively unbalanced generalized Feistel networks, where 1/3 or 1/4 of the state is updated, using addition as the blending function and a simple multiplication of all or part of the remaining state as the round function. The result follows O'Neill's guideline that it should be a reversible permutation to avoid loss of state, while using only a limited number of multiply/add operations:

```
uint3 pcg3d(uint3 v)
{
    v = v * 1664525u + 1013904223u;
    v.x += v.y*v.z; v.y += v.z*v.x; v.z += v.x*v.y;
    v ^= v >> 16u;
    v.x += v.y*v.z; v.y += v.z*v.x; v.z += v.x*v.y;
    return v;
}
```

```
uint4 pcg4d(uint4 v)
{
    v = v * 1664525u + 1013904223u;
    v.x += v.y*v.w; v.y += v.z*v.x; v.z += v.x*v.y; v.w += v.y*v.z;
    v ^= v >> 16u;
    v.x += v.y*v.w; v.y += v.z*v.x; v.z += v.x*v.y; v.w += v.y*v.z;
    return v;
}
```

7. Conclusion

As shown in the results, no single hashing algorithm can fit every use case. Rather, each algorithm on the Pareto frontier lies on a spectrum from the fastest algorithms to the algorithms that yield the best results. Many of the hashes that fall along the Pareto frontier were designed to be executed on the GPU with neighboring seeds. Chained PRNGs with a state, such as `city`, `murmur3`, and `ranlim32`, may be great PRNGs [Pike and Alakuijala 2011; Appleby 2008; Press et al. 2007] when used sequentially, but when used in parallel with neighboring seeds, they fall short.

We conclude by providing a spanning set of exemplar algorithms for each usage category, ordered from fastest to highest quality.

- $(1 \to 1)$: `iqint1`, `pcg`, `xxhash32`, `pcg3d`, `pcg4d`
- $(2 \to 1)$: `iqint3`, `xxhash32_multibyte2`, `pcg3d`, `pcg4d`
- $(3 \to 1)$: `pcg3d16`, `xxhash32_multibyte3`, `pcg3d`, `pcg4d`
- $(4 \to 1)$: `xxhash32_multibyte4`, `pcg4d`
- $(N \to N)$: `pcg3d`, `pcg4d`

In all the test cases, the fastest hashes are `lcg`, nested `lcg`, and `xorshift128`. However, these hashes have such poor quality they are not practical without additional scrambling operations. Because of this, we did not recommend `lcg` and `xorshift128` above.

The methods and results presented here will hopefully serve as a benchmark for hashing algorithms in terms of quality and run-time performance.

8. Future Work

PractRand [Doty-Humphrey 2019] is a newer test suite that is more thorough than TestU01 BigCrush [L'Ecuyer and Simard 2007]. We found better differentiation of (fast) low-quality hashes using TestU01, but if several significantly higher-quality and higher-performance hashes are discovered, PractRand would most likely be a better differentiator.

It would be worthwhile to create a multidimensional hash test suite rather than trying to borrow existing test suites that were designed for linear sequences. Such a test suite could be more likely to notice problems in the domain than running a set of one-dimensional experiments on the Morton order.

Given that most hashes tested in this work were not designed to simultaneously optimize for both randomness and speed, there is certainly room for further development of new fast and high-quality random hashes. One approach that has been successfully used for hash development is random genetic-algorithm mutation of code.

This could be successful for GPU or multidimensional hash development with an appropriate choice of objective function with a balance of both factors. In addition, there are very few hashes natively designed for multidimensional input and/or output. Future work that provided options in the mid-range of Figure 8 could be particularly useful.

9. Acknowledgments

Work on this project was funded in part by a gift from Epic Games. The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258, CNS-1228778, and OAC-1726023) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

APPLEBY, A., 2008. SMHasher. [Online; accessed June-2018]. URL: `https://github.com/aappleby/smhasher/`. 22, 24, 33

BASSHAM, L., 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., 2010. A statistical test suite for random and pseudorandom number generators for cryptographic applications. URL: `https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final`. 22

BLUM, L., BLUM, M., AND SHUB, M. 1986. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing 15*, 2, 364–383. URL: `https://doi.org/10.1137/0215025`. 24

BROWN, R. G., 2017. Dieharder: A random number test suite. URL: `http://webhome.phy.duke.edu/~rgb/General/dieharder.php`. 22

COLLET, Y., 2012. xxHash: Extremely fast hash algorithm. [Online; accessed June-2018]. URL: `https://github.com/Cyan4973/xxHash`. 22, 24

DOTY-HUMPHREY, C., 2019. PractRand pre0.95, October. [Online; accessed March 2020]. URL: `https://sourceforge.net/projects/pracrand`. 22, 23, 33

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing and Modeling: A Procedural Approach*, 3rd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ch. 2, 91–93. 30

EPIC GAMES, 2014. Unreal Engine 4.1, Nov. URL: `https://github.com/EpicGames/UnrealEngine`. 24

EPIC GAMES, 2016. Unreal Engine 4.13, Sept. URL: `https://github.com/EpicGames/UnrealEngine`. 24, 32

HARNETT, J., 2018. Toolbox of noisey goodness. [Online; accessed April-2019]. URL: https://www.shadertoy.com/view/4dVBzz. 27

HOANG, V. T., AND ROGAWAY, P. 2010. On generalized Feistel networks. In *CRYPTO 2010: Advances in Cryptology*, Springer, 613–630. URL: https://link.springer.com/chapter/10.1007/978-3-642-14623-7_33. 32

HOSKINS, D., 2014. Hash without sine. [Online; accessed June-2018]. URL: https://www.shadertoy.com/view/4djSRW. 24, 27

HOWES, L., AND THOMAS, D. 2007. Efficient random number generation and application using CUDA. In *GPU Gems 3*, H. Nguyen, Ed., first ed. Addison-Wesley Professional, Boston, MA, USA, ch. 37. URL: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch37.html. 24

HSIEH, P., 2004. Hash functions. [Online; accessed June-2018]. URL: http://www.azillionmonkeys.com/qed/hash.html. 24

JIMENEZ, J. 2014. Next generation post processing in Call of Duty: Advanced Warfare. In *SIGGRAPH Courses: Advances in Real-Time Rendering*. ACM, New York, NY, USA. URL: http://advances.realtimerendering.com/s2014/index.html. 24

JONES, D., 2010. Good practice in (pseudo) random number generation for bioinformatics applications. [Online; accessed June-2018]. URL: http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf. 24

KENDALL, M. G., AND SMITH, B. B. 1938. Randomness and random sampling numbers. *Journal of the Royal Statistical Society 101*, 1, 147–166. URL: http://www.jstor.org/stable/2980655. 22

KNUTH, D. E. 1978. *The Art of Computer Programming*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 21

L'ECUYER, P., AND SIMARD, R. 2007. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw. 33*, 4 (Aug.), 22:1–22:40. URL: https://doi.org/10.1145/1268776.1268777. 22, 23, 33

LEHMER, D. H. 1949. Mathematical methods in large-scale computing units. In *Proceedings of a Second Symposium on Large Scale Digital Calculating Machinery*. Harvard University, Cambridge, MA, USA, 141–146. URL: https://archive.org/details/proceedings_of_a_second_symposium_on_large-scale_. 24

MARSAGLIA, G., 1995. The Marsaglia random number CDROM including the diehard battery of tests of randomness. URL: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/. 22

MARSAGLIA, G. 2003. Xorshift RNGs. *Journal of Statistical Software 8*, 14, 1–6. URL: https://www.jstatsoft.org/v008/i14. 24

MORTON, G. M. 1966. A computer oriented geodetic data base; and a new technique in file sequencing. Tech. rep., IBM Ltd., Ottawa, Canada. URL: https://domino.research.ibm.com/library/cyberdig.nsf/papers/0DABF9473B9C86D48525779800566A39/$File/Morton1966.pdf. 24

O'NEILL, M. E. 2014. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. URL: https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf. 22, 23, 24, 32

O'NEILL, M. E., 2014. PCG random number generation, C++ edition, version 0.98, December. [Online; accessed July-2019]. URL: https://github.com/imneme/pcg-cpp/releases/tag/v0.98. 32

PERLIN, K. 1985. An image synthesizer. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85. ACM, New York, NY, USA, 287–296. URL: https://doi.org/10.1145/325165.325247. 21, 27

PHILLIPS, C. L., ANDERSON, J. A., AND GLOTZER, S. C. 2011. Pseudo-random number generation for Brownian dynamics and dissipative particle dynamics simulations on GPU devices. *Journal of Computational Physics 230*, 19, 7191–7201. 21

PIKE, G., AND ALAKUIJALA, J., 2011. CityHash. [Online; accessed June-2018]. URL: https://github.com/google/cityhash. 24, 33

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2007. *Numerical Recipes: The Art of Scientific Computing*, 3rd ed. Cambridge University Press, New York, NY, USA. 24, 33

QUILEZ, I., 2017. Integer hash - I. [Online; accessed June-2018]. URL: https://www.shadertoy.com/view/llGSzw. 24, 27

QUILEZ, I., 2017. Integer hash - II. [Online; accessed June-2018]. URL: https://www.shadertoy.com/view/XlXcW4. 24

QUILEZ, I., 2017. Integer hash - III. [Online; accessed June-2018]. URL: https://www.shadertoy.com/view/4tXyWN. 24

REED, N., 2013. Quick and easy GPU random numbers in D3D11. [Online; accessed April-2019]. URL: http://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/. 23

REY, W. 1998. On generating random numbers, with help of y= [(a+x)sin(bx)] mod 1. In *22nd European Meeting of Statisticians and the 7th Vilnius Conference on Probability Theory and Mathematical Statistics*. VSP, Zeist, The Netherlands. 24

SASTRY, V., AND KUMAR, K. 2012. A modified Feistel cipher involving modular arithmetic addition and modular arithmetic inverse of a key matrix. *International Journal of Advanced Computer Science and Applications 3* (07). URL: https://doi.org/10.14569/IJACSA.2012.030705. 32

SCHECHTER, H., AND BRIDSON, R. 2008. Evolving sub-grid turbulence for smoke animation. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08. Eurographics, Aire-la-Ville, Switzerland, 1–7. URL: http://dl.acm.org/citation.cfm?id=1632592.1632594. 24

TZENG, S., AND WEI, L.-Y. 2008. Parallel white noise generation on a GPU via cryptographic hash. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, I3D '08. ACM, New York, NY, USA, 79–87. URL: https://doi.org/10.1145/1342250.1342263. 21, 24

WANG, T., 2007. Integer hash function. [Online; accessed June-2018]. URL: https://web.archive.org/web/20070108113114/http://www.cris.com:80/~Ttwang/tech/inthash.htm. 24

WEI, L.-Y. 2004. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, GH 2004. ACM, New York, NY, USA, 55–63. 21

WHEELER, D. J., AND NEEDHAM, R. M. 1995. TEA, a tiny encryption algorithm. In *Fast Software Encryption*, Springer Berlin Heidelberg, Berlin, Heidelberg, B. Preneel, Ed., International Association for Cryptologic Research, 363–366. 24

WYMAN, C., AND MCGUIRE, M. 2017. Hashed alpha testing. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17. ACM, New York, NY, USA, 7:1–7:9. URL: https://doi.org/10.1145/3023368.3023370. 21

ZAFAR, F., OLANO, M., AND CURTIS, A. 2010. GPU random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*, HPG '10. Eurographics, Aire-la-Ville, Switzerland, Switzerland, 133–141. URL: http://dl.acm.org/citation.cfm?id=1921479.1921500. 21, 22, 23

Index of Supplemental Materials

The supplementary materials available online include

- `supplementary.pdf`: shader code, timing, number of tests failed, and sample images for every hash tested (http://jcgt.org/published/0009/03/02/supplementary.pdf)
- `bigcrush.zip`: Full results of all the TestU01 BigCrush runs (http://jcgt.org/published/0009/03/02/bigcrush-results.zip)

Author Contact Information

Mark Jarzynski
UMBC
1000 Hilltop Circle
Baltimore, MD 21250
jarzynski@umbc.edu

Marc Olano
UMBC
1000 Hilltop Circle
Baltimore, MD 21250
olano@umbc.edu

Mark Jarzynski and Marc Olano, Hash Functions for GPU Rendering, *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 3, 20–38, 2020
http://jcgt.org/published/0008/02/??/

Received:	2019-12-13		
Recommended:	2020-01-31	Corresponding Editor:	Eric Haines
Published:	2020-10-17	Editor-in-Chief:	Marc Olano

© 2020 Mark Jarzynski and Marc Olano (the Authors).
The Authors provide this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at http://creativecommons.org/licenses/by-nd/3.0/. The Authors further grant permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.