The Goulburn Hashing Function

Mayur Patel*

Abstract

We discuss the use of random numbers in computer graphics. We analyze a new hashing function, which we call Goulburn, and argue that it is well-suited for computer graphics because of its very high quality and speed. We discuss how to apply Goulburn to applications requiring pseudo-random number generation.

1. Introduction

Random numbers are critical in computer graphics, from sampling to aesthetics. We believe that there are three families of random sources of interest to computer graphics developers:

- Quasi-random number generators (QRNGs)
- Pseudo-random number generators (PRNGs)
- Hashing functions

Ideally a developer's toolbox would include high-quality algorithms for each of the three random sources.

In practice, however, developers will often substitute tools. In computer graphics sampling applications, it is usually sufficient to use PRNGs to produce jittered sample positions rather than to use QRNGs to produce low discrepancy sequences of sample points.

In a similar spirit, a PRNG can be implemented using a hashing function in the following way: the seed for the generator would constitute the most significant bits of a very large integer counter. The integer counter can be hashed to produce a random value. Between calculations, the counter can be incremented by one. This is a known technique in cryptography, where secure PRNGs are implemented using secure hashing functions.

Producing random values this way allows the developer to control the state size of the PRNG very explicitly. This is particularly important in agent simulations which may require an instance of a PRNG for each agent. Modern high-quality PRNGs often have large fixed-size states, making it prohibitive for very many instances to coexist in memory. A good hashing function will be able to produce random values with sufficient speed and quality.

Because PRNGs can be substituted for QRNGs in many cases, and good hashing functions can be substituted for PRNGs, we focus on good hashing functions as a random source for computer graphics.

We found many popular hashing functions to be inadequate for computer graphics. Some very fast hashing functions are sufficiently random to avoid collisions in a hash table, but still produce artifacts when used for lattice noises [Uzgalis and Tong 1994; Noll]. Jenkins' hash provides a high quality of randomness, but its speed is not exceptional [1997]. Inspired by the table-based techniques used in Buzhash and Pearson's algorithm [1990], we developed a hashing function suitable for computer graphics, which we call Goulburn.

In the next section, we discuss Goulburn's relationships to existing hashing functions. In section 3, we discuss the design of Goulburn. In section 4, we discuss the implementation of a PRNG using Goulburn. In section 5, we present the results of empirical testing.

2. Prior Art

These are the most common application of hashing functions in computer science:

- Error detection in information exchange.
- Fast indexing of data / accelerating compare operations.
- Cryptography.

For most error detection and indexing applications, the avoidance of collisions is the foremost consideration, not the quality of randomness. We found readily recognizable patterns in the output of many hashing functions that perform well for indexing applications [Pearson 1990; Uzgalis and Tong 1994; Noll]. Some of these results are presented in section 5. Cryptographically secure hashing functions and strongly universal hashing functions do emphasis a good quality of randomness; however, we found these algorithms to be much less flexible in their application. They tended to require fixed-sized input or input in large fixed-size blocks.

For computer graphics, we require the hashing function to return high-quality random values from variable-length keys. We also require very fast execution, because the hashing function will likely be called a very large number of times.

In our investigations, table-based general hashing functions seemed to have the greatest potential to meet all requirements, so we focused our attention there. Zobrist's hash [1970] is an early form of table-based hashing. The table for this function needs to be tailored for each context, so we did not find it best as a general hashing function. Pearson's hash [1990] requires processing output in 8-bit chunks, so that a 32-bit result requires four iterations. Buzhash is very fast, but despite the author's strong assertions of quality [Uzgalis and Tong 1994], we were readily able to find patterns in the output which made its quality unacceptable for our graphics applications.

Goulburn can best be described as an evolution of Pearson's hash and Buzhash. In the next section, we describe the function in some detail.

3. The Goulburn Hashing Function

Goulburn is a general hashing function. Like many hashing schemes, Goulburn is based on executing a series of instructions that are one-to-one mappings, whose range and domain are the same set of values when executed on modern microprocessors. This helps to insure that collisions do not occur unnecessarily frequently. Here is a list of such instructions:

- $f(x) = x + i$
- $f(x) = x \wedge i$
- $f(x) = r(x, j)$
- $f(x) = x \wedge r(x, j)$

with the following definitions:

- i and j are integers; $j \neq 0$
- "^" is the bitwise XOR operation
- $r(x, j)$ is the circular left shift of x by j bits.

The inner loop of the 32-bit hashing function is listed, using C:

```
1.  for( u=0; u<len; ++u )
2.  {
3.    h += table0[ c[u] ];
4.    h ^= (h << 3) ^ (h >> 29);
5.    h += table1[ h >> 25 ];
6.    h ^= (h << 14) ^ (h >> 18);
7.    h += 1783936964UL;
8.  }
```

Clearly, we could produce 64-bit or 128-bit hash values on machines that support registers of those widths, without significant additional expense. A more thorough listing of the Goulburn function can be found in the appendices.

Lines 3 and 4 closely resemble a Buzhash function. Table zero is constructed in the same manner as the table used in Buzhash: the table is an unbiased, randomized set of binary words. Lines 5 and 6 apply additional randomization to avoid "echoing" patterns observed when using Buzhash in visual applications.

Table one contains binary words, where each word has an equal number of ones and zeroes, but the exact permutations are random. We do this to guarantee that the addition instruction in line 5 toggles at least half the bits of the word.

Buzhash and Pearson's hash use the XOR instruction, relying on the property that the XOR of two values has the maximum entropy of the two values. In other words, if you cannot predict the value of either X or Y, then you cannot predict the value of the XOR of X and Y. The quality of randomness is intended to come from the lookup table, and this quality is intended to be preserved through the XOR operations.

Goulburn uses ADD instructions on lines 3 and 5 instead of XOR. We do this to increase the complexity of the relationships between the random values in the tables and the bits of the hash value. Let binary words X and Y consist of binary digits, $x_n..x_0$, and $y_n..y_0$ respectively, and let Y be a uniformly distributed random value. The addition of the least significant bits, x_0 and y_0, creates an output value which has the maximum entropy of x_0 and y_0 because this is simply an XOR operation. The carry bit becomes an extra random signal, even if it is a weak one, to be applied to the next bit, where a three-way XOR is performed to produce that output bit. In this way, randomness in less significant bits is carried to more significant bits. In testing, we observed that this improved the quality of randomness in our hashing function. The use of an ADD instruction on line 7, and the rotations on lines 4 and 6, insures that entropy is distributed more evenly, instead of collecting in a narrow range of bits within the hash value.

Jenkins lists these as properties of a good general hashing function [1997]:

1. Keys can be variable length, of any kind of data.
2. Sets of independent hashing functions may be required.
3. The hash calculation should be fast.
4. A change in any bit of the input could possibly affect every bit of the output.
5. Common substrings and patterns between different inputs should not increase the probability that the hashes of the two keys are equal.

Goulburn loops byte-wise over a key, so it satisfies criterion one. The tables used in Goulburn can be regenerated dynamically to satisfy criterion two. In section five, we discuss performance issues relating to criterion three. It suffices to say here that we found Goulburn to be at least twice as fast as Jenkins' function. The application of the two random tables, followed with rotations, insures that criteria four and five are satisfied.

4. Goulburn as a PRNG

In the introduction, we described how a good hashing function could be used as a PRNG, by initializing the most significant bits of a large integer counter with a seed value, by hashing this large integer to produce random values, and by incrementing the large integer between invocations of the function.

Goulburn, like many hashing functions, has a simple initialization step, followed by a per-byte loop to produce the output hash value. When used as a PRNG, this structure facilitates the following optimization: if the large integer counter is implemented as a series of bytes $b_n..b_0$, where b_0 is the least significant byte, and if the traversal order of the hash loop is from b_n to b_0, then we can cache the hash of the partial word, $b_n..b_1$ so we usually only need to execute the final iteration of the hash loop to incorporate b_0 into the final hash value. This means that most invocations of the PRNG will execute in constant time regardless of the size of the integer counter. Of course, when b_0 overflows, then we need to refresh our cached calculation, which is linear with the size of the integer counter. If the output hash value is 32-bits, then the optimization costs 32-bits of storage. However, the dramatic increase in the performance of the PRNG can often justify the additional storage expense. The listing in the appendices demonstrates the technique.

When considering the periodicity of Goulburn as a PRNG, unfortunately we cannot make very strong guarantees. If A is a counter consisting of bytes $a_n..a_0$, and B is a counter consisting of bytes $b_m..b_0$, where a_0 and b_0 are the least significant bytes and are

unequal, then even if the hash value of A equals the hash value of B, the two words do not represent the same state of the PRNG and will not produce equivalent sequences. For proof of this, consider incrementing A and B. The least significant bytes will now produce very different values during the hash calculations (especially during table lookup) and the hash output value will be different.

However, imagine that A and B are the most significant bytes of other larger counters, A' and B'. Also let A' and B' have equal least significant bytes: A' has the bytes $a_n..a_0c_i..c_0$ and B' has the bytes $b_m..b_0c_i..c_0$. In this case, A' and B' will have the same hash value. In fact, they will produce equal hash values for a sequence length of 256^{i+1}. We know that some pair of A and B must exist, because the hash value is of fixed size, so collisions must occur for some pair of arbitrarily big keys. Because of this, we cannot guarantee that the PRNG based on Goulburn will not demonstrate some properties of short periodicity; however, in practice and empirical testing we have not observed particularly bad results in this regard.

In many computer graphics applications, the need for huge periodicities is not as acute as it might be in some scientific applications. Furthermore, since the *quality* of randomness is good, it may not be harmful to observe short sequences of redundant values at irregular intervals in computer graphics applications. This is obviously not true for all fields of computing.

L'Ecuyer gives these as criteria for good random number generators [2001]:

- Guaranteed long periodicity
- Efficient in memory and execution
- Repeatable
- Portable
- Allows jumping forward and backward in the sequence by large steps
- Emperically tested, both with statistical tests and in the context of target applications.

Goulburn, as a PRNG, scores poorly in the first criterion, but does well in the others. Results of empirical testing are the subject of the next section.

5. Emperical Testing

In this section, we present tests for quality and speed for four hashing functions: Buzhash, FNV1a, Jenkins' hash and Goulburn.

In the following images, we simply produced an image of hash values, where the color of each pixel is the hash of the 2D pixel coordinate. It is easy to observe patterns in Buzhash and FNV1a. Jenkins' hash and Goulburn produced images that visually appear to be random.

Figure 1: Image of a Buzhash output

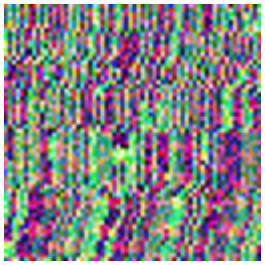


Figure 2: Image of an FNV1a hash output

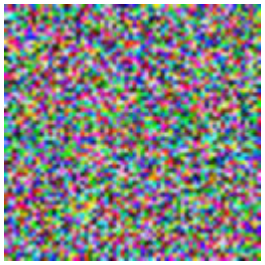


Figure 3: Image of a Jenkins' hash output

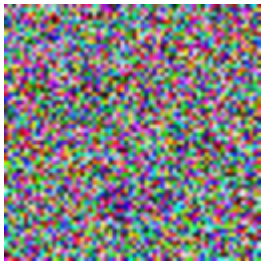
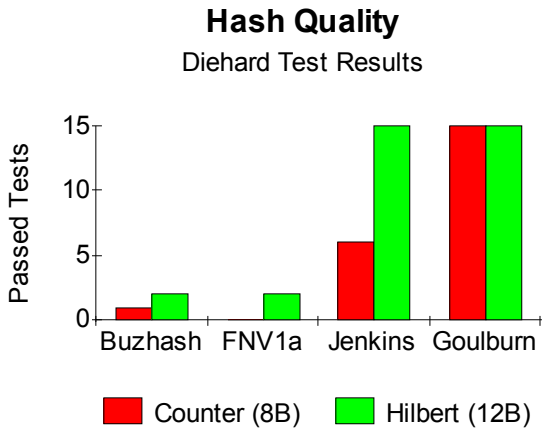


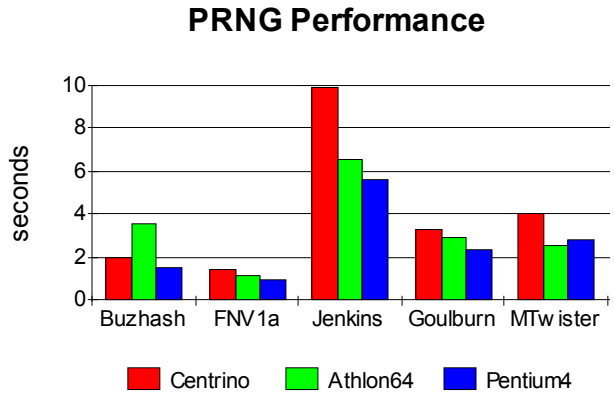
Figure 4: Image of a Goulburn hash output

The visual representation is obviously not a rigorous test of randomness. We tested the four functions using the Diehard statistical tests. We analyzed hash values generated from two different access patterns, an incrementing large integer key and 3D Hilbert curve coordinates. We believe these are representative of the types of access patterns that developers observe in computer graphics.



Goulburn passed all the Diehard statistical tests for both types of keys. Jenkins' hash matched Goulburn's quality on the Hilbert curve coordinates.

We also implemented PRNGs using the four hashing functions, with a 64-bit state and with the optimization described in the previous section, costing an extra 4 bytes of storage. We compared the performance of these to Wagner's implementation of Mersenne Twister [Wagner 2003; Matsumoto and Nishimura 1998], which is generally considered to be the fastest PRNG of very high quality.



Comparing the Goulburn PRNG to Mersenne Twister, it ran 15% slower on an Athlon64 processor, 21% faster on a Pentium 4, and 24% faster on a Centrino processor. Buzhash and FNV1a tended to outperform Goulburn, but Jenkins' hash was found to be slow.

While Goulburn was not clearly the fastest function, its tradeoff between speed and quality makes it attractive for computer graphics applications.

Conclusions

Each random source has its strengths and weaknesses, but Goulburn has been developed to be well-suited to the demands of computer graphics. It provides flexibility and good randomness at a fair computational cost.

References

JENKINS, R. September 1997. Algorithm Alley: Hashing Functions. *Dr. Dobb's Journal*.

L'ECUYER, P. 2001. Software for Uniform Random Number Generation: Distinguishing the Good from the Bad, *Proceedings on the 2001 Winter Simulation Conference*.

MATSUMOTO, M. AND NISHIMURA, T. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator, *ACM Transactions on Modeling and Computer Simulation 8,* 1. 3-30.

NOLL, L. C. *Fowler / Noll / Vo (FNV) Hash*. http://www.isthe.com/chongo/tech/comp/fnv/

PEARSON, P. June 1990. Fast Hashing of Variable-Length Text Strings. *Communications of the ACM, 33,* 6. 677-680.

UZGALIS, R. AND TONG, M. 1994. *Hashing Myths.* Technical Report 97, Department of Computer Science University of Auckland.

WAGNER, R. May 2003. *Mersenne Twister Random Number Generator.* http://www-personal.engin.umich.edu/~wagnerr/MersenneTwister.html.

ZOBRIST, A. L. April 1970. A New Hashing Method with Application for Game Playing. Technical Report 88, Computer Sciences Department, University of Wisconsin.

Appendix 1: goulburn.h

```
// ********************************************
//
// The Goulburn Hashing Function and
// Pseudo-Random Number Generator
//
// Random Numbers for Computer Graphics, Aug 06
// ACM SIGGRAPH 2006, Sketches and Applications
//
// Copyright 2006, Mayur Patel
// This software listing for illustrative
// purposes only:  USE AT YOUR OWN RISK
// ********************************************

#ifndef GOULBURN_H
#define GOULBURN_H

namespace goulburn {

typedef unsigned long value_type;
static const value_type max_value;

//
// The Goulburn Hashing Function
// The interface allows long hashes to be broken
// into piece-wise calculations, by passing the
// last value of the hash into the function.
//
value_type hash( const unsigned char *cp, unsigned len,
value_type last_value );

//
// A Pseudo-Random Number Generator based on the
// Goulburn hashing function.  Developer specifies
// how many bytes to use for the state; of course
// this affects periodicity.  Note that there's
// no magic here: any really good hashing function
// could be used as a driver for this kind of
// pseudo-random number generator.
//
```

```
// This is implemented with an optimization which
// costs 4 bytes of storage, so that the full hash
// over the entire state size is only required once
// every 256 calls.  The rest of the time, we're only
// hashing one byte.
//
// The API is a subset of the ForwardIterator
// interface.
//
template< unsigned BYTES >
class prng {
protected:
  unsigned char _cp[BYTES];
  value_type _cache;

public:
  prng( void ){ seed( 0, 0 ); }
  prng( const unsigned char *cpSeed, unsigned len )
  { seed( cpSeed, len ); }

  void seed( const unsigned char *cpSeed, unsigned len )
  {
    unsigned u=0;

    if( cpSeed )
    {
      for( ; (u<len) && (u<BYTES); ++u )
        _cp[u] = cpSeed[u];
    }

    for( ; u<BYTES; ++u )
      _cp[u] = 0;

    _cache = hash( _cp, BYTES - 1, 0 );
  }


  value_type operator *(void) const
  {
    return hash( _cp + BYTES - 1, 1, _cache );
  }

  prng& operator++(void)
  {
    unsigned u = BYTES;
    do {
      --u;
      ++_cp[u];
    } while( u && (!_cp[u]) );

    if( u != (BYTES-1) )
      _cache = hash( _cp, BYTES - 1, 0 );

    return *this;
  }

  prng operator++(int)
  {
    prng ret = *this;
    ++(*this);
    return ret;
  }
};

} // end namespace goulburn

#endif
```

Appendix 2: goulburn.cpp

```
// ********************************************
//
// The Goulburn Hashing Function and
// Pseudo-Random Number Generator
//
// Random Numbers for Computer Graphics, Aug 06
// ACM SIGGRAPH 2006, Sketches and Applications
//
// Copyright 2006, Mayur Patel
// This software listing for illustrative
// purposes only:  USE AT YOUR OWN RISK
// ********************************************

#include <goulburn.h>

const goulburn::value_type g_table0[256] =  {
4143812366,2806512183,4212398656,393834663,
3943187971,847901099,3746904015,2990585247,
4243977488,4075301976,2737181671,2429701352,
4196558752,3152011060,1432515895,204108242,
1180540305,922583281,1734842702,1453807349,
507756934,1553886700,2005976083,3346025117,
97642817,2510760451,4103916440,3222467334,
1312447049,522841194,3955607179,3028936967,
2763655970,3033075496,1935362065,512912210,
2660383701,1652921526,260485165,141882627,
2895806269,804034013,1356707616,3942447612,
2875374199,81028672,1055595160,2755907176,
2880512448,1232977841,3719796487,2940441976,
3739585976,168332576,1318372270,3173546601,
3992298512,3785690335,3667530757,3101895251,
2789438017,3213463724,3067100319,2554433152,
794184286,2599814956,1251486151,4214997752,
690900134,323888098,1537487787,1155362310,
1826165850,2358083425,2957662097,2514517438,
1828367703,3847031274,2308450901,955547506,
1037823031,2922505570,2544914051,2572931499,
442837508,1873354958,2004376537,25413657,
3560636876,1768043132,2870782748,1031556958,
715180405,201079975,4116730284,2748714587,
1091411202,33354499,1931487277,1039106939,
3327011403,396608379,3447523131,301432924,
3180185526,1780290520,3909968679,2398211959,
3704875308,66082280,601805180,3226323057,
3284786200,2282257088,700775591,3528928994,
1601645543,120115228,568698020,178214456,
41846783,897656032,3309570546,2624714322,
2542948622,1168171675,2460933760,93808223,
2384991231,4268721795,4001720080,1516739672,
4111847489,810915309,1238071781,935043360,
2020231594,37717498,3603218947,1534593867,
2819275526,1965883441,674162751,128087286,
4138356188,543626850,1355906380,3565721429,
1142978716,1614752605,1624389156,3363454971,
2029311310,2249603714,3448236784,1764058505,
2198836711,3481576182,3168665556,3834682664,
1979945243,3456525349,2721891322,1099639387,
1528675965,3069012165,1807951214,1901014398,
2805656341,3321210152,2317543573,1015607418,
178584554,4020226276,492648819,97778844,
4134244261,1389599433,331211243,3769684011,
2036127367,3174548433,3241354897,2570869934,
3071842004,1972073698,48467379,1015444026,
3126762609,1104264591,3096375666,1380392409,
684368280,1493310388,2109527660,3034364089,
3168522906,3042350939,3696929834,3410250713,
3726870750,3357455860,1816295563,2678332086,
26178399,614899533,2248041911,1431155883,
1184971826,3711847923,2744489682,168580352,
694400736,2659092308,811197288,1093111228,
824677015,2041709752,1650020171,2344240270,
3773698958,3393428365,3498636527,556541408,
1883820721,3249806350,3635420446,1661145756,
3087642385,1620143845,3852949019,1054565053,
3574021829,2466085457,2078148836,460565767,
4097474724,1381665351,1652238922,2200252397,
3726797486,4001080204,259576503,567653141,
325219513,1227314237,3191441965,1433728871,
4198425173,2908977223,3757065246,294312130,
4136006097,3409363054,2112383431,1177366649
};

const goulburn::value_type g_table1[128] = {
826524031,360568984,3001046685,1511935255,
1287825396,3167385669,1488463483,4077470910,
1360843071,986771770,2307292828,3845679814,
1429883439,1990257475,4087625806,1700033651,
1388994450,935547107,3237786789,644530675,
2274037095,888755779,3020158166,2136355264,
2558959443,1751931693,2325730565,3029134627,
668542860,2140243729,2384660990,666440934,
842610975,1563602260,1429103271,899918690,
3441536151,4078621296,1527765522,4191433361,
222526771,309447417,2035245353,3730203536,
```

```
3330019758,876252573,2545027471,453932528,
282738293,1826993794,1569532013,543681326,
3097574376,2336551794,1563241416,1127019882,
3088670038,2766122176,3706267663,1110947226,
2608363541,3166834418,1310161541,755904436,
2922000163,3815555181,1578365408,3137960721,
3254556244,4287631844,750375141,1481489491,
1903967768,3684774106,765971482,3225162750,
2946561128,1920278401,1803486497,4166913456,
1855615192,1934651772,1736560291,2101779280,
3560837687,3004438879,804667617,2969326308,
3118017313,3090405800,566615197,2451279063,
4029572038,2612593078,3831703462,914594646,
2873305199,2860901605,3296630085,1273702937,
2852911938,1003268745,1387783190,159227777,
2211994285,28095103,3659848176,3976935977,
3301276082,2641346573,651238838,2264520966,
1484747269,3016251036,3857206301,91952846,
1662449304,2028491746,1613452911,2409055848,
1453868667,4146146473,1646176015,3769580099,
3171524988,2980516679,828895558,3384493282
};

const goulburn::value_type max_value = 0xFFFFFFFF;

goulburn::value_type
goulburn::hash( const unsigned char *cp, unsigned len,
goulburn::value_type last_value )
{
  register goulburn::value_type h = last_value;
  unsigned u;

  for( u=0; u<len; ++u )
  {
    h += g_table0[ cp[u] ];
    h ^= (h << 3) ^ (h >> 29);
    h += g_table1[ h >> 25 ];
    h ^= (h << 14) ^ (h >> 18);
    h += 1783936964UL;
  }

  return h;
}
```

* email: patelm@acm.org

This is an unpublished, extended version of the sketch:

PATEL, M. August 2006. Random Numbers for Computer Graphics. *ACM SIGGRAPH 2006, Sketches and Applications.*