Pixar Technical Memo 13-01

Correlated Multi-Jittered Sampling

Andrew Kensler

March 5, 2013

We present a new technique for generating sets of stratified samples on the unit square. Though based on jittering, this method is competitive with low-discrepancy quasi-Monte Carlo sequences while avoiding some of the structured artifacts to which they are prone. An efficient implementation is provided that allows repeatable, random access to the samples from any set without the need for precomputation or storage. Further, these samples can be either ordered (for tracing coherent ray bundles) or shuffled (for combining without correlation).

Introduction

Image synthesis techniques requiring Monte Carlo integration frequently need to generate uniformly distributed samples within the unit hypercube. The ideal sample generation method will maximize the variance reduction as the number of samples increases. Stratification via jittering[1] is one simple, tractable, well known and well understood technique for generating samples, though there are other methods.

[1]R. L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

Of these, low-discrepancy quasi-Monte Carlo (QMC) sequences such as Sobol's (0,2) sequence[2] and the Larcher-Pillichshammer sequence[3] have become quite popular for their deterministic generation and improved variance reduction versus jitter. See Kollig and Keller[4] for an overview. Many QMC sampling methods also offer easy random access to shuffled sets of samples. This makes it easy to "pad" or combine together samples from several decorrelated lower-dimensional distributions into a single higher-dimensional sample (e.g., combining image samples, lens samples, material samples, and light samples) whereas the equivalent shuffling of jittered samples typically requires enumerating the full set first. Unfortunately, QMC methods can also be prone to structured artifacts.

[2]I. M. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[3]G. Larcher and F. Pillichshammer. Walsh series analysis of the L_2-discrepancy of symmetrisized point sets. *Monatshefte für Mathematik*, 132:1–18, April 2001.

[4]T. Kollig and A. Keller. Efficient multidimensional sampling. *Computer Graphics Forum*, 21(3):557–563, September 2002.

Therefore we seek an alternative that combines the robust foundation of jittered sampling with the benefits of QMC. Our new approach is based on three main contributions: first, a modification to Chiu et al's multi-jittered sampling[5] that greatly improves the convergence rate. Second, a way to generate patterns with arbitrary sample counts (including prime numbers) without noticeable gaps or clumps. Finally, we provide an implementation that builds a pseudorandom permutation function out of a reversable hash. This allows us to compute any arbitrary sample directly from its index and a pattern index.

[5]K. Chiu, P. Shirley, and C. Wang. Multi-jittered sampling. In *Graphics Gems IV*, chapter V.4, pages 370–374. Academic Press, May 1994.

Multi-Jittered Sampling

In 2D, jittered sampling stratifies a set of N samples by dividing the unit square into equal area cells using an $m \times n$ grid (where $N = mn$ and $m \approx n$) and randomly positioning a single sample within each cell. This reduces clumping since samples can only clump near the cell boundaries; no more than four samples can ever clump near a given location.

However, jittering suffers when these samples are projected onto the X- or Y-axis. In this case, we effectively have only m or n strata rather than N. This can greatly increase the variance at edges, especially when they are nearly axis-aligned. The N-rooks sampling pattern[6] fixes this by jittering independently in each dimension, shuffling the samples from one of the dimensions, and then pairing the samples from each dimension. This gives the full N strata when projecting onto an axis, but may result in more clumping in 2D.

[6] P. Shirley. Discrepancy as a quality measure for sample distributions. In *Eurographics '91*, pages 183–193, September 1991.

Chiu et al's multi-jittered sample pattern achieves both of these properties simultaneously. Each cell and each horizontal or vertical substratum is occupied by a single jittered sample. Their method for producing these samples begins by placing them in an ordered, "canonical" arrangement as shown in Figure 1:

Listing 1: Producing the canonical arrangement.

```
for (int j = 0; j < n; ++j) {
    for (int i = 0; i < m; ++i) {
        p[j * m + i].x = (i + (j + drand48()) / n) / m;
        p[j * m + i].y = (j + (i + drand48()) / m) / n;
    }
}
```

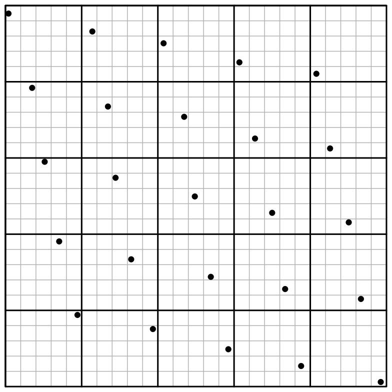
Figure 1: The canonical arrangement. Heavy lines show the boundaries of the 2D jitter cells. Light lines show the horizontal and vertical substrata of N-rooks sampling. Samples are jittered within the subcells.

The next step shuffles the arrangement. First, the X coordinates in each column of 2D cells are shuffled. Then, the Y coordinates in each row are shuffled:

Listing 2: Shuffling the canonical arrangement.

```
1   for (int j = 0; j < n; ++j) {
2       for (int i = 0; i < m; ++i) {
3           int k = j + drand48() * (n - j);
4           std::swap(p[j * m + i].x,
5                     p[k * m + i].x);
6       }
7   }
8   for (int i = 0; i < m; ++i) {
9       for (int j = 0; j < n; ++j) {
10          int k = i + drand48() * (m - i);
11          std::swap(p[j * m + i].y,
12                    p[j * m + k].y);
13      }
14  }
```

Figure 2 shows an example of the result. The shuffle preserves the 2D jitter and the N-rooks properties of the original canonical arrangement. The result is slightly better than plain jitter, but still shows a degree of clumpiness and unevenness in the distribution of samples.

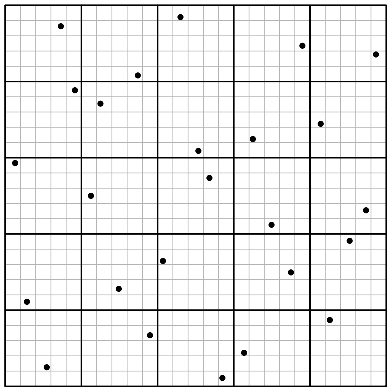
Figure 2: A multi-jittered sampling pattern.

Correlated Multi-Jittered Sampling

Our key observation is that a small tweak to the shuffling strategy can make a dramatic improvement to the sample distribution: apply the *same* shuffle to the X coordinates in each column, and apply the *same* shuffle to the Y coordinates in each row. Listing 2 can be trivially changed to accomplish this simply by exchanging line 2 with 3, and line 9 with 10.

This change greatly reduces the clumpiness as shown by the example pattern in Figure 3. There is a still a randomness to the position of the samples but they are much more evenly distributed throughout the unit square. The correlation in the shuffles means that each sample is now roughly $1/m$ apart

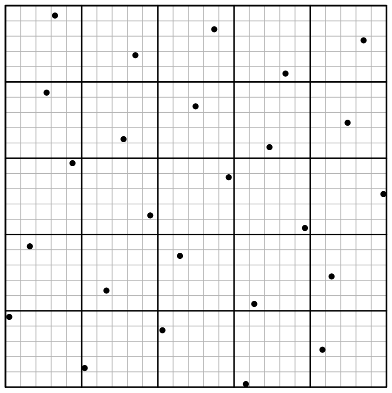
Figure 3: With correlated shuffling.

horizontally from its neighbors to the left and right in the 2D jitter cells and $1/n$ apart vertically from its top and bottom neighbors (though they have greater lateral freedom in the relative placement). The only chance for clumping is from samples in diagonally connected jitter cells and because of the nature of the initial pattern and the special shuffling, no more than two pairs of samples within a pattern may ever be in directly diagonally neighboring subcells. Note that these properties hold even under toroidal shifts.

If N is prime or otherwise not easily factored into m and n, there is a slight variation on the above algorithm that works very well. Choose m and n by rounding up such that $N \leq mn$ but they are otherwise as close to equal as possible. Then stretch the pattern along one axis by mn/N and clip the last $mn - N$ samples (which should now be outside the unit square) from the end. This is illustrated in Figure 4. The remaining samples are evenly spaced along the stretched axis. Though this produces gaps in the substrata along the other axis, they will tend to be well distributed. It can also also result in minor clumping of the samples when wrapping the stretched axis toroidally.

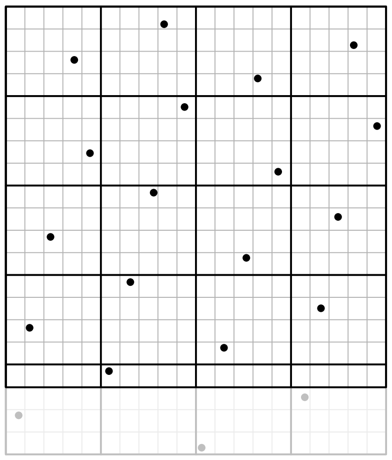
Figure 4: Irregular sample counts ($N = 17$ in this example) by stretching and clipping.

Results of Sampling on the Square

Figure 5 shows an example of correlated multi-jittered sampling in use in a direct lighting test. Here, it compares favorably to jittered and QMC techniques for sampling an area light source; it is less noisy than the standard jittered sampling and avoids the structured artifacts seen with the Larcher-Pillichshammer sampling.

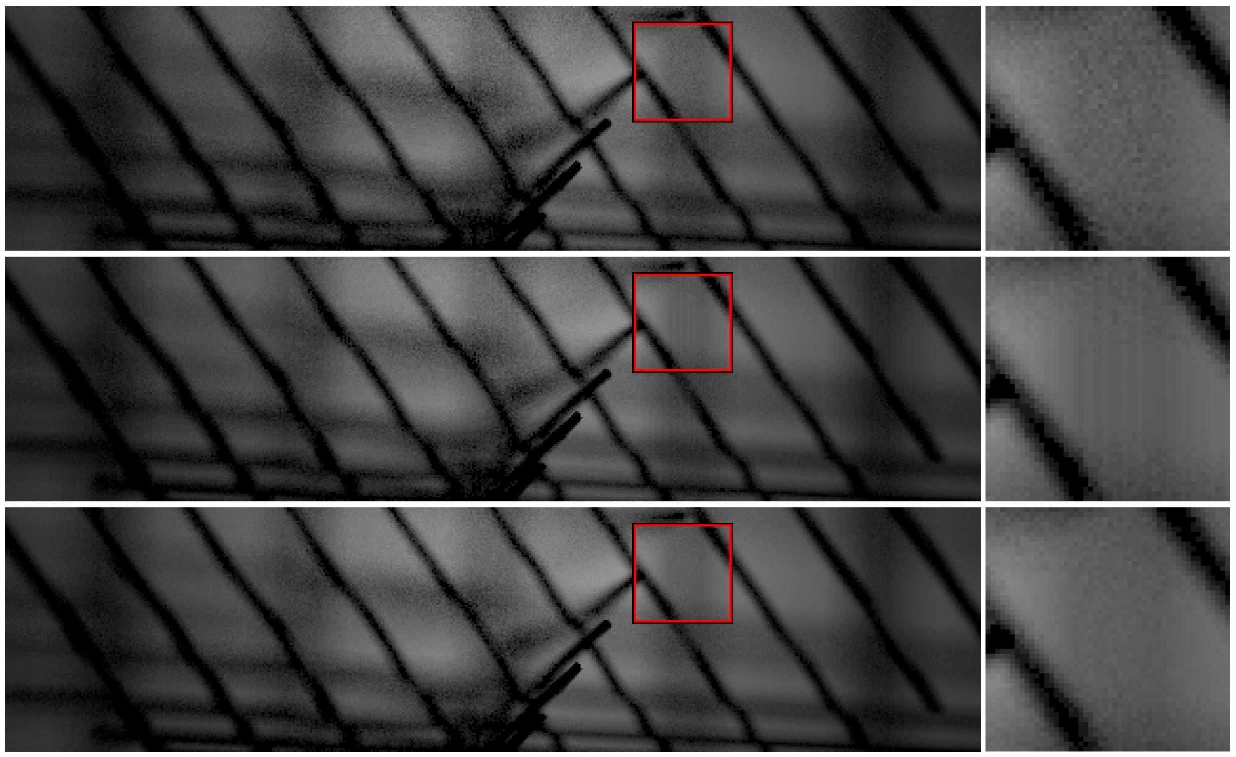
Figure 5: Sampling a square area light. Rendered using 1 image sample and 25 light samples per pixel with per-pixel scrambling. The light-blockers are hidden to the camera. From top to bottom: standard jitter, Larcher-Pillichshammer, and correlated multi-jittered sampling.

The cause of the artifacts for Larcher-Pillichshammer is shown in Figure 6; each sample falls at a regular $1/N$ interval along one axis (X in this case). This property is shared with Hammersley sampling and it is part of what gives them such low discrepancy. Presumably, adding jitter could help with this, albeit at the cost of increasing noise. Alternatively, instead of using per-pixel scrambling to produce a unique pattern at each pixel, some sampling patterns may be stretched over the entire image and used to produce all of the samples. This strategy can still alias, however (e.g., using the Halton sequence for image plane samples with an infinite checkerboard plain scene[7]). In general, low-discrepancy QMC sequences trade a higher potential for structured artifacts against faster convergence.

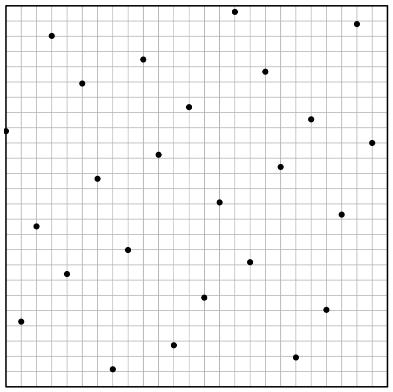

Figure 6: Larcher-Pillichshammer points. Light lines are at $1/N$ intervals.

[7]M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory To Implementation.* Morgan Kaufmann, 2nd edition, 2010.

The convergence rate of these and other sampling patterns in the above scene is shown in Figure 7. Here, variance is measured as the mean squared error relative to a reference image computed using 16384 jittered light samples per pixel. Above 25 samples per pixel, correlated multi-jittering generally outperforms all other methods except for the Larcher-Pillichshammer sequence.

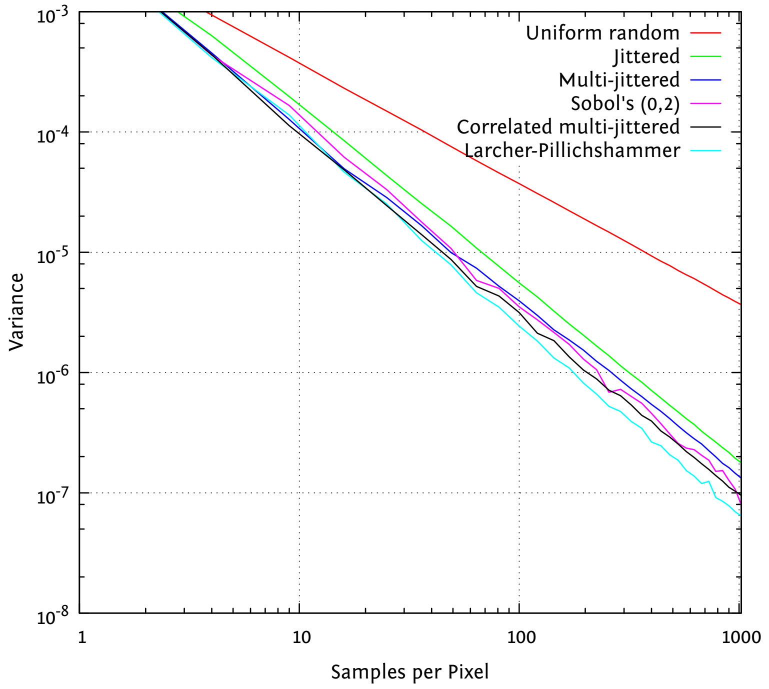

Figure 7: Samples per pixel versus variance in the above rectangular light source test scene.

Similar measurements were done on scenes that tested other applications including the sampling of disk area lights, uniform- and cosine-weighted hemispheres for BRDFs, image plane positions for antialiasing, and lens positions for depth of field. All showed the variance with correlated multi-jittering to be competitive with Sobol's (0,2) sequence and occasionally competitive with the Larcher-Pillichshammer sequence. However, Sobol's sequence has an advantage in that it does not require a bound on the number of samples to be known ahead of time and that it is hierarchical (i.e., the initial points in the sequence are well distributed) which can simplify progressive rendering.

Pattern	Discrepancy
Uniform random	0.0275
N-rooks	0.0159
Poisson disk	0.0150
Jittered	0.0110
Multi-jittered	0.0059
Halton	0.0053
Sobol's (0,2)	0.0047
Correlated multi-jittered	0.0042
Hammersley	0.0030
Larcher-Pillichshammer	0.0021

Table 1: Patterns ordered by decreasing star discrepancy.

Table 1 shows the results of our final evaluation method: for a number of different pattern generation algorithms, an example point set containing

1600 points on the unit square was generated and the star discrepancy[8] was estimated to within 0.0001. The relative ranking here corresponds well to the ranking seen in Figure 7. In this test, correlated multi-jittered sampling had lower discrepancy than Halton and Sobol, the two hierarchical low-discrepancy sequences. Note that low discrepancy does not guarantee a good image however; the two lowest discrepancy patterns here are also the two most prone to the artifact seen in Figure 5.

[8] E. Thiémard. An algorithm to compute bounds for the star discrepancy. *Journal of Complexity*, 17(4):850–880, September 2001.

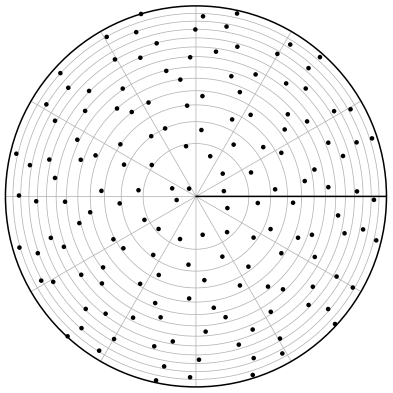

Figure 8: Polar warp with $m = 12, n = 13$. Heavy lines show the edges from the original unit square. Light lines show the boundaries of the 2D jitter cells.

Results of Sampling on the Disk

Because sampling disks and hemispheres are so crucial to BRDFs, we repeated the variance measurements using several methods for uniformly sampling a disk. The simplest of these approaches would be to assume that $m \approx n \approx \sqrt{N}$ as for sampling a square and then warp the samples from (x, y) in the square to polar coordinates on the disk with $(\theta, r) = (2\pi x, \sqrt{y})$. Figure 8 shows an example of this. Note that the "seam" in the mapping is well hidden because the sample pattern wraps toroidally.

A variation from Ward and Heckbert[9] uses standard jittering with the same square-to-disk polar mapping, but chooses the number of 2D jitter cells so that $m \approx \pi n$ in order to stratify more strongly along the axis used for the angular component. This is perfectly doable with our correlated multi-jittering as well and an example of the result is shown in Figure 9.

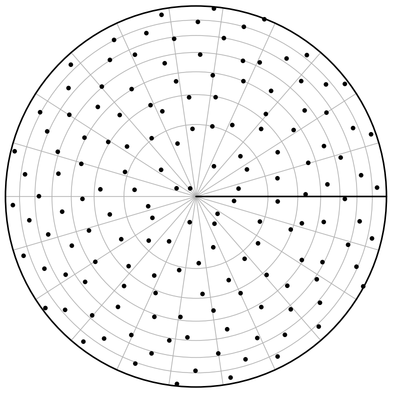

Figure 9: Polar warp with $m = 22, n = 7$.

[9] G. J. Ward and P. S. Heckbert. Irradiance gradients. In *Third Eurographics Rendering Workshop*, pages 85–98, May 1992.

The last approach that we tested, illustrated in Figure 10, was to use the original $m \approx n$ ratio, but instead of the simple polar mapping we used Shirley and Chiu's[10] low distortion "concentric" disk mapping.

Surprisingly, the simpler polar mapping consistently outperformed Shirley and Chiu's mapping in terms of variance when used with our sampling algorithm. Of the two jitter stratification ratios, $m \approx n$ appeared to give slightly better results for sampling hemispheres (both uniformly and cosine weighted) while $m \approx \pi n$ was the slightly better ratio for soft shadows from disk shaped area light sources. Note that the exact ratio is somewhat less relevant to correlated multi-jittering when compared to standard jittering due to the N-rooks property of the former; regardless of the actual ratio, the samples will be distributed into N strata when projected onto the X- or Y-axis. It is still important to maintaining the average 2D distance between the points, however.

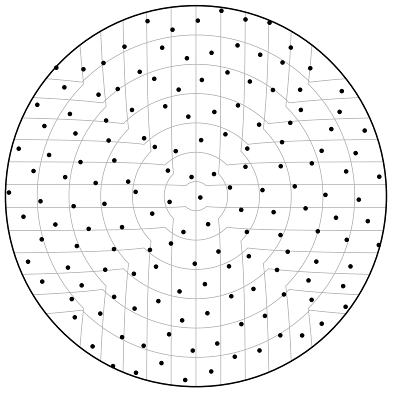

Figure 10: "Concentric" disk mapping with $m = 12, n = 13$.

[10] P. Shirley and K. Chiu. A low distortion map between disk and square. *Journal of Graphics Tools*, 2(3):45–52, 1997.

Pseudorandom Permutations

The previously described method for generating correlated multi-jittered samples suffices if we wish to generate all the samples in a given pattern at once. For practical reasons, however, we also want to be able to deterministically and repeatably compute an arbitrary sample out of the set of N in a given pattern. These samples may be needed in any order and some samples may never actually be needed.

The difficulty, then, arises mainly from the random shuffles on m or n elements at a time. As a shuffle is equivalent to indexing through a random permutation vector, we can reduce this to needing to efficiently compute the element at position i in a random permutation vector of length l where the value p is used to select one of the $l!$ possible permutations.

Interestingly, it turns out that this is a problem well studied in cryptography. For example, a 128-bit block cipher such as the Advanced Encryption

Standard[11] divides the plain text into 128-bit blocks and maps these through a permutation selected by bits from the key. It must be a permutation to be decryptable, and it must be statistically indistinguishable from a random permutation in order to be strong against an adversary.

[11]National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication (FIPS 197), 2001.

Unfortunately, the domain of most block ciphers is much larger than we need. "Format preserving", "data preserving", or "small block" encryption attempts to address the need for smaller domains but can be quite expensive in time or space due to the need to withstand an adversary. Nonetheless, some of the concepts will prove useful.

For a simpler, faster method we turn to hash functions. Several of the elementary functions used to create hash functions are reversible:[12]

[12]B. Mulvey. Hash functions. `http://home.comcast.net/~bretm/hash/`, 2007.

```
hash ^= constant;
hash *= constant; // if constant is odd
hash += constant;
hash -= constant;
hash ^= hash >> constant;
hash ^= hash << constant;
hash += hash << constant;
hash -= hash << constant;
hash = (hash << constant) | (hash >> (wordsize-constant));
```

These bitwise operations are reversible for any power-of-two sized domain. Given this, a hash function composed entirely of such operations must be a permutation. Of these, the second and fifth operations in the list are less trivial to reverse. The former is also good for mixing the lower bits into the higher bits of the hash while the later is good for mixing the higher bits back into the lower bits.

We can evaluate the quality of a hash function by the "avalanche" property:[13] for any input, flipping any single bit of the input should, on average, cause half of the output bits to flip. Using a hill-climbing optimization program to generate candidate hashes and evaluate them by the avalanche property, we have found a short hash function composed only of reversible operations that achieves good results at word sizes between 33 and 59 bits. Allotting 32 bits to choose a permutation, this allows us random permutations on power-of-two-sized domains up to 2^{27}.

[13]H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973.

If the size of the permutation vector, l, is not a power of two we can use a technique called cycle walking:[14] we map a pseudorandom permutation with a larger domain to a smaller one by iteratively applying the larger permutation until we get a value within the range of the smaller. So long as the initial value is within the smaller range, this is guaranteed to terminate with an expected number of iterations proportional to the ratio of their sizes. In this case, we can round l up to the next power of two and so the expected number of iterations will always be fewer than two.

[14]J. Black and P. Rogaway. Ciphers with arbitrary finite domains. *Topics in Cryptology – CT-RSA 2002*, pages 114–130, February 2002.

In short, our pseudorandom permutation function looks like this:

Listing 3: A pseudorandom permutation function.

```
unsigned permute(unsigned i, unsigned l, unsigned p) {
    unsigned w = l - 1;
    w |= w >> 1;
    w |= w >> 2;
    w |= w >> 4;
    w |= w >> 8;
    w |= w >> 16;
    do {
        i ^= p;             i *= 0xe170893d;
        i ^= p       >> 16;
        i ^= (i & w) >>  4;
        i ^= p       >>  8;  i *= 0x0929eb3f;
        i ^= p       >> 23;
        i ^= (i & w) >>  1;  i *= 1 | p >> 27;
                             i *= 0x6935fa69;
        i ^= (i & w) >> 11;  i *= 0x74dcb303;
        i ^= (i & w) >>  2;  i *= 0x9e501cc3;
        i ^= (i & w) >>  2;  i *= 0xc860a3df;
        i &= w;
        i ^= i       >>  5;
    } while (i >= l);
    return (i + p) % l;
}
```

The other piece that we will need (for repeatable jittering within the substrata) is a way to map an integer value to a pseudorandom floating point number in the [0,1) interval where the sequence is determined by a second integer. Again using hill-climbing to evaluate candidate hash functions for the avalanche property we found the following to work well:

Listing 4: A pseudorandom floating point number generator.

```
float randfloat(unsigned i, unsigned p) {
    i ^= p;
    i ^= i >> 17;
    i ^= i >> 10;     i *= 0xb36534e5;
    i ^= i >> 12;
    i ^= i >> 21;     i *= 0x93fc4795;
    i ^= 0xdf6e307f;
    i ^= i >> 17;     i *= 1 | p >> 18;
    return i * (1.0f / 4294967808.0f);
}
```

Implementation

Given these pieces, we can now construct a function that efficiently computes a repeatable arbitrary sample s from a set of $N = mn$ samples in the pattern p without requiring significant precomputation or storage:

Listing 5: Computing an arbitrary sample from a correlated multi-jittered pattern.

```
xy cmj(int s, int m, int n, int p) {
    int sx = permute(s % m, m, p * 0xa511e9b3);
    int sy = permute(s / m, n, p * 0x63d83595);
    float jx = randfloat(s, p * 0xa399d265);
    float jy = randfloat(s, p * 0x711ad6a5);
    xy r = {(s % m + (sy + jx) / n) / m,
            (s / m + (sx + jy) / m) / n};
    return r;
}
```

Essentially, this maps s to its 2D jitter cell and then offsets it within that cell to the appropriate substratum in X by the permutation of its cell coordinate in Y and vice-versa. A small bit of additional random jitter is also added to its position in the substratum. With higher sample counts (e.g., $N \geq 64$), this may be removed to slightly reduce the variance. For lower sample counts, it is necessary for avoiding structured artifacts. If strict repeatability is not needed, the `randfloat()` function may be omitted and calls to it replaced with calls to `drand48()` or similar.

Note that this function produces samples in roughly scan-line order by jitter cell as s increases. This may actually be desirable for ray coherence. Furthermore, this makes it straightforward to enumerate the samples from within a subregion of the full pattern. If the ordering is not wanted, an additional permutation step on *s* can be added to decorrelate corresponding samples from different patterns.

With a small variation, we can also easily implement the stretching and clipping method previously described for handling irregular *N*. The following modifies the previous function to compute a suitable *m* and *n* with aspect ratio *a* and apply the stretch and clip. It also shuffles the samples' order:

Listing 6: The final sampling code.

```
xy cmj(int s, int N, int p, float a = 1.0f) {
    int m = static_cast<int>(sqrtf(N * a));
    int n = (N + m - 1) / m;
    s = permute(s, N, p * 0x51633e2d);
    int sx = permute(s % m, m, p * 0x68bc21eb);
    int sy = permute(s / m, n, p * 0x02e5be93);
    float jx = randfloat(s, p * 0x967a889b);
    float jy = randfloat(s, p * 0x368cc8b7);
    xy r = {(sx + (sy + jx) / n) / m,
            (s + jy) / N};
    return r;
}
```

Finally, note that though the code shown here is aimed at conciseness rather than speed, most C++ compilers do a good job of optimizing it by moving invariants out of the loop when computing a pattern's worth of samples at a time; we can get 16.9M samples/s on a single 2.8GHz Xeon X5660 core with ICC 12.1. Computing a single sample at a time (as for path tracing), however, tends to defeat this and force recomputation of the invariants. Manually creating a small thread-local single-item cache that recomputes the permutation masks and multiplicative inverses only when *N* truly does change can provide a significant speed improvement.

Conclusion

We have presented a modification to jittering and multi-jittering that greatly reduces the noise in rendered images at higher sample counts while gracefully degrading to unstructured noise at low sample counts. The provided implementation is well suited for parallel environments: it produces repeatable samples, is flexible with respect to sample ordering, and allows for efficient sampling from subregions of a full pattern. It will be the foundation for stratified sampling in Pixar's RenderMan Pro Server begining with version 18.0.

While the focus of this work has been on 2D, the functions `permute()` and `randfloat()` can trivially be used for shuffled 1D jittered sampling as well. The ease of shuffling makes generating higher dimensional samples from combinations of 1D and 2D samples quite straightforward. Nonetheless, we are currently exploring generalizations of the algorithm to 3D and above.

Other areas for future work include modifying the shuffling of the samples' order to visit well-spaced samples first. This would make the sample sequence hierarchical and improve the initial feedback in progressive rendering. We would also like to consider spatially-varying sample densities, arbitrary dimensional samples (for paths), and mapping sample patterns over entire images rather than individual pixels.

The author wishes to thank the members of the RenderMan team for their help on this project, especially Per Christensen who commented on early drafts of this paper. Leonhard Grünschloß of Weta also provided helpful comments.