Blue-Noise Dithered QMC Hierarchical Russian Roulette
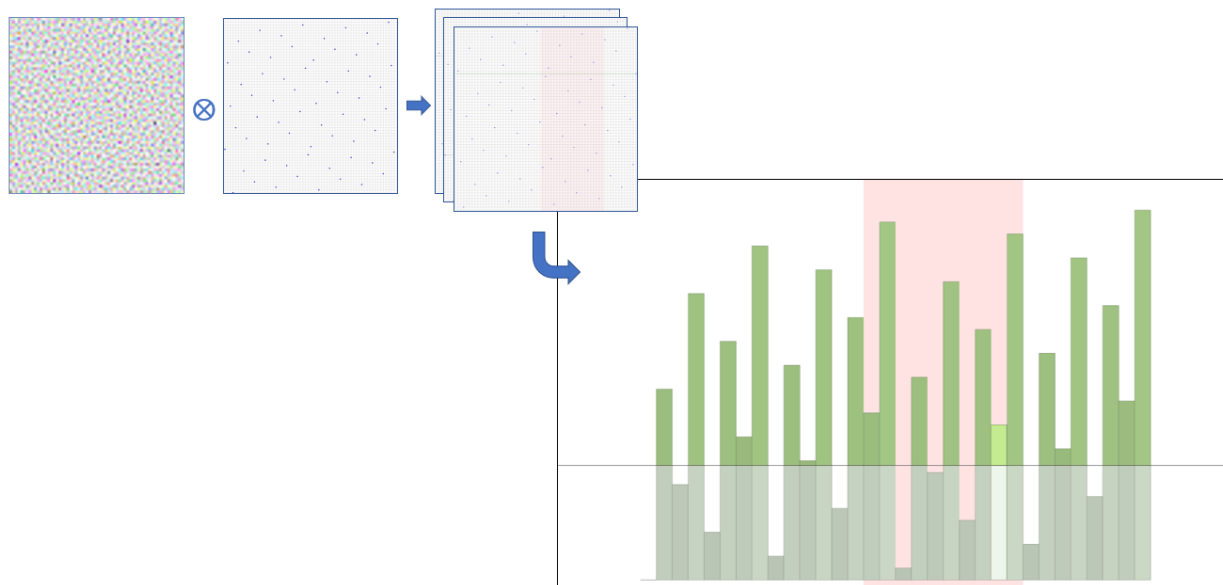
Jacopo Pantaleoni*
NVIDIA

Figure 1: This work replaces the pseudo-random sequences used in the original hierarchical Russian roulette many-light algorithm with blue-noise dithered QMC sequences, making use of a novel algorithm to compute the minimum of a shifted Halton sequence over arbitrary intervals. A tile of blue-noise distributed values (top row, left) is associated with the 2d pixel distribution of the eye vertices; each pixel in the tile is used to shift a common base-2 Halton sequence (top row, middle), producing a multitude of shifted sequences, one for each eye vertex e_j (top-row, right). The green horizontal lines in the shifted sequences represent the shift values. The figure on the bottom shows a representation of the key operation needed to perform hierarchical Russian roulette, finding the minimum over a range of indices, highlighted in pink: notice that since the shift is performed modulo 1, if the shift is r, in the unshifted sequence the minimum value has to be found in the vertical range $[1-r, 1)$.

Abstract

In order to efficiently sample specular-diffuse-glossy and glossy-diffuse-glossy transport phenomena, [Tokuyoshi and Harada 2019] introduced *hierarchical Russian roulette*, a smart algorithm that allows to compute the minimum of the random numbers associated to the leaves of a tree at each internal node. The algorithm is used to efficiently cull the connections between the product set of eye and light vertices belonging to large caches of eye and light subpaths produced through bidirectional path tracing [Veach 1997].

The original version of the algorithm is entirely based on

*e-mail: jpantaleoni@nvidia.com

the generation of semi-stratified pseudo-random numbers. Our paper proposes a novel variant based on deterministic blue-noise dithered Quasi Monte Carlo samples [Georgiev and Fajardo 2016].

1 Introduction

Throughout this work, we assume that we have two large sets of eye vertices, $\{e_j : j \in [0, N_e)\}$ and light vertices $\{l_i : i \in [0, N_l)\}$. We further assume that the set of light vertices is sorted by a hierarchy, so that each leaf is given an index i in $[0, N_l)$, and each internal node n of the hierarchy can be represented by a sub-range $[a_n, b_n) \subseteq [0, N_l)$.

Tokuyoshi and Harada [2019] showed that if one associates a randomly sized culling shape to each pair of eye and light vertices (e_j, l_i), where the size of the culling shape is inversely proportional to a pseudo-random number ξ_{ij}, it is possible to use the hierarchy over the lights to efficiently cull all lights in sub-linear time for each eye vertex, performing so called *hierarchical Russian roulette*. The key operation required to perform hierarchical Russian roulette is a top-down tree traversal where the minimum of all pseudo-random numbers within each node's range is efficiently computed.

In this work, we replace the pseudo-random sequences with deterministic QMC sequences. Specifically, we assign a

unique Cranley-Patterson rotated Halton sequence to each eye vertex j:

$$\phi_j(i) = (\Phi_2(i) + r_j) \mod 1 \qquad (1)$$

where Φ_2 is the base-2 radical inverse (also known as van der Corput sequence, see [Niederreiter 1992]) and r_j is the Cranley-Patterson rotation, or *shift*, assigned to the given eye vertex. In practice, if the eye vertex indices are sorted by pixel and path depth, the generation of the shifts can be easily made such that its image-plane projection has a desirable blue-noise distribution [Georgiev and Fajardo 2016]. Notice that the sequence index i here corresponds to a light vertex index in the range $[0, N_l)$: hence, if the light vertices are spatially sorted, as for example done in Matrix Bidirectional Path Tracing [2018], the low-discrepancy and stratification properties of the sequence will be inherited by the distribution of the final samples.

Now, given the above setting, the key operation needed to perform hierarchical Russian roulette consists in finding, for each internal node, the minimum value that the sequence assumes inside its range $[a, b)$:

$$v(a, b) = \min_{i \in [a, b)} \phi_j(i) \qquad (2)$$

While for the final application the use of this novel sampling scheme has interesting aspects of its own, the focus of this work will be the solution of this central problem.

2 Computing minima of the shifted Halton sequence over arbitrary intervals

We first recap the definition of the radical inverse:

$$\begin{aligned}
\Phi_b(i) : \mathbb{N} &\rightarrow \mathbb{Q} \cap [0, 1) \\
i = \sum_k a_k(i) b^k &\mapsto \sum_k a_k(i) b^{-k-1}
\end{aligned} \qquad (3)$$

In practice, for any index i, the radical inverse is obtained by first reversing the binary expansion of i, and then using that as the binary expansion of the base-2 digits after the comma of the result.

We will also change the problem setting by trying to find the index $k \in [a, b)$ that minimizes ϕ_j:

$$k = \arg\min_{i \in [a, b)} \phi_j(i) \qquad (4)$$

In the absence of a shift, $\phi_j = \Phi_2$, and since the most important digits of i become the least important in the expansion of $\Phi_2(i)$, finding k would mean finding the number in $[a, b)$ with the most rightmost zeros. This can be done with an $O(B)$ algorithm in the number of bits B, which we report in Algorithm 1.

The presence of the shift makes the problem slightly more complex. Pseudo-code for the solution we are about to detail is given in Algorithm 2. In the following, to simplify notation we'll drop the explicit dependence on the eye vertex index, and simply use r instead of r_j. Now, we can see that for a given sequence index i there are essentially three cases:

$$\begin{aligned}
\Phi_2(i) &= 1 - r \Rightarrow \phi_j(i) = 0 \qquad (5) \\
\Phi_2(i) &> 1 - r \Rightarrow \phi_j(i) \in (0, r) \qquad (6) \\
\Phi_2(i) &< 1 - r \Rightarrow \phi_j(i) \in (r, 1) \qquad (7)
\end{aligned}$$

The first case can only be obtained if $k_r := \Phi_2^{\leftarrow}(1 - r) \in [a, b)$. If that is the case, $k = k_r$ and the minimum value assumed by ϕ_j inside the node range is zero.

Algorithm 1: find the number in $[a, b)$ with most rightmost zeros

```
if a == 0 or a == b-1:
    return a;

// find the most significant bit set of b
b_h = fls(b);

// find the least significant bit set of a
b_l = ffs(a);

// sweep right-to-left, each time clearing
// the current bit and setting the next
k = m = a;
for i = b_l in b_h - 1:
    m = clear(m, i); // clear the i-th bit

    m = set(m, i + 1); // set bit i+1

    if m ∈ [a, b):
        k = m;
return k;
```

If $k_r \notin [a, b)$, the next option is to look for an index i such that its radical inverse $\Phi_2(i)$ is the smallest number larger than 1 - r. That is to say, we'd have to look for solutions of:

$$k = \arg\min_i \left\{ \Phi_2(i) \right\} : \quad i \in [a, b) \cap \left\{ \Phi_2(i) > 1 - r \right\} \qquad (8)$$

In order to have a value of $\Phi_2(k)$ that is larger than $1 - r$, but at the same time as small as possible, we'll start looking at the bit structure of a, b and k_r. We'll indicate the position of the last (i.e. highest) bit set of b with $b_h = fls(b)$, and the first (i.e. lowest) bit set of k_r with $k_p = ffs(k_r)$. If k_p is part of a contiguous range of bits set in k_r, we'll indicate the entire range with $\{k_p, ..., k_q\}$, with $k_p < ... < k_q$. (e.g. $b = 0\mathbf{1}0010$, we'll have $b_h = 4$ and if $k_r = 100\mathbf{11}0$, we'll have $p = 1$, and $q = 2$). Moreover, we'll indicate the last bit set of a with $a_h = fls(a)$. We're now ready to proceed with the algorithm descriptions.

2.1 Left-to-right sweep

We'll start by considering a mask m obtained by clearing all the leading bits of k_r higher than b_h (e.g. with k_r and b as in the example above, we'd consider $m = 000110$). The main idea is that we'll try to craft k by flipping only the highest possible zero bit of m to the left of k_q that leads to a value contained in $[a, b)$. We do this, because the highest bits of m are the least important in its image $\Phi_2(m)$, due to the bit reversal in the radical inverse, and by modifying them we are trying to obtain a value that is just slightly larger than $1 - r$.

Hence, we'll proceed with a single left-to-right sweep through its bits, b_i, starting from b_h and ending at k_p, performing the following sequence of operations:

- if the bit b_i is already set in m, we clear it and increment i, continuing to the next bit;

- else, we set the corresponding bit in m, obtaining:

$$m_i = m \,|\, 2^{b_i}; \qquad (9)$$

by construction, $\Phi_2(m_i) > 1 - r$; hence, if m_i is contained in the range $[a, b)$, and if $a_h \le i < b_h$ we can conclude that $k = m_i$;

If during the loop we moved beyond a_h, it might still be possible to find a valid index satisfying equation (6) by

altering a single bit between a_h and k_q; however, we need to make sure we keep and merge into m all the set bits of a we encounter in our left-to-right walk, as otherwise we'd produce an index smaller than a, and hence outside the target range; in fact, occasionally, if the i-digit prefix of m (and hence k_r) is smaller than the corresponding prefix of a, even after merging all the bits of a into m, m_i might be too small to fall into $[a, b)$: in these cases, we set an extra 1 bit at index $i + 1$; it is important to note, however, that in this case m_i is no longer necessarily the smallest such value: we are only guaranteed that the bit b_i is the highest possible bit in k_r (i.e. the lowest possible bit in the binary expansion of $1 - r$) at which a flip from zero to one produces a value inside $[a, b)$ satisfying equation (6), or - in other words - that the minimum value k exists and must have bit b_i set.

2.1.1 Solution refinement

In order to find k given m_i we'll have to make several observations. The first is that the *prefix* of k, starting at bit 0 and ending at bit i, is now fixed as $k^* = \{\mathbf{1}k_{i-1}...k_0\}$, where k_j is the j-th bit of k_r. In the following, we'll indicate the (i+1) digit prefix of the binary expansion of a number n with $n[0, i]$. We'll also call I the subset of $[a, b)$ with prefix k^*, i.e. $I = \{j \in [a, b) : j[0, i] = k^*\}$, and indicate with I_0 the subset of I which has bit (i+1) set to 0, and I_1 the subset of I with bit (i+1) set to 1, and similarly call I_{00} the subset of I_0 with bit (i+2) set to 0, and so on. We'll then have the following relations:

$$\Phi_2(k_1) < \Phi_2(k_2) \quad \forall \quad k_1 \in I_0, \ k_2 \in I_1$$
$$\Phi_2(k_1) < \Phi_2(k_2) \quad \forall \quad k_1 \in I_{00}, \ k_2 \in I_{01}$$
$$\Phi_2(k_1) < \Phi_2(k_2) \quad \forall \quad k_1 \in I_{000}, \ k_2 \in I_{001} \quad (10)$$

In other words, we can establish an ordering relation among the sets $I_{j_1 j_2 ... j_n}$:

$$I_{s_1 s_2 ... s_n} < I_{t_1 t_2 ... t_n} \Leftrightarrow s_1 s_2 ... s_n < t_1 t_2 ... t_n. \quad (11)$$

Now, if the prefix $b[0, i]$ is larger than k^*, we can form k as a number that has 1 in position b_h and zeros for each index $j : b_i < j < b_h$, i.e. $k = b_h 0...0 k^*$ and conclude our search, as k belongs to the *smallest* set $I_{0...01}$. If that is not the case, we'll have to proceed by induction, and start by looking for a solution in I_0. If that exists, we'll proceed to I_{00}. If not, we'll look for one in I_{10}. And so on.

By construction, the number m_i we obtained might already belong to a given set $I_{0...01...}$, with a given number n_0 of zeros before the first bit set is encountered. In that case, we'll directly skip those n_0 zero bits, and simply assume, without lack of generality, they are part of k^*. With this assumption, we know that the next bit in position $i + 1$ must be 1 in m_i. Let's assume we have a whole sequence of 1's, $m_i[i + 1, i + j] = 1...1$. In this case, our only chance to find a number in I_0 is to flip all those 1's to 0's, and change the next bit to 1, which can be achieved by means of a single addition of 2^{i+1}. This is because this procedure will produce the smallest number bigger than our current value of m_i having the bit in position $i + 1$ set to 0. If the value thus obtained belongs to $[a, b)$, we are done, as we effectively found a number in $I_{s_1...s_{j+1}}$, with $s_1 = ... = s_j = 0$ and $s_{j+1} = 1$. Now we can repeatedly apply the same algorithm to the next sequence of 1's. Pseudo-code for this procedure is given in Algorithm 3.

2.2 Right-to-left sweep

If we conclude the left-to-right loop without finding k, it means it is only possible to satisfy equation (6) by flipping
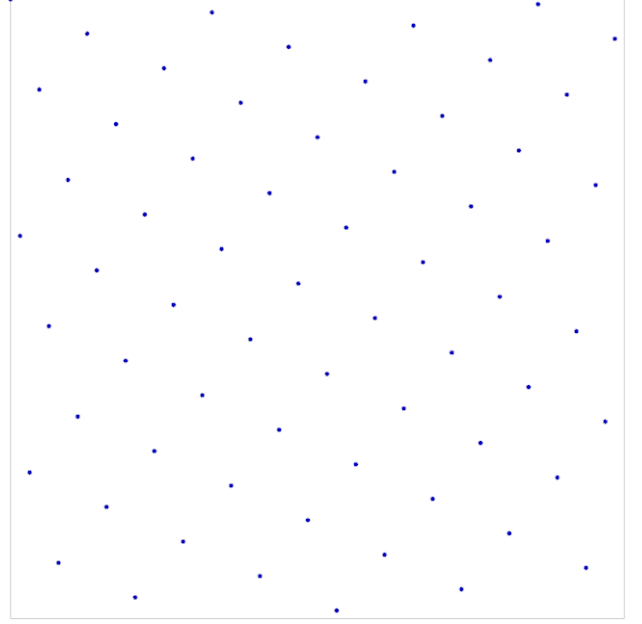

Figure 2: The golden-ratio rank-1 lattice can be used to further shift the base sequence to obtain multiple samples for each eye vertex.

bits of m to the right of k_p. For example, this would happen with $a = 000101$, $b = 001000$, and $k_r = 010110$: by necessity, k would have to have the form *****$\mathbf{1}$. In this case, we'll proceed with the same right-to-left sweep used in Algorithm 1, except we modify its bounds and we further track the case $\Phi_2(k) > 1 - r$, which is only fulfilled if the last bit set to 1 preceeds k_p. If no such k is found, it simply means that there is no index i in $[a, b)$ such that $\Phi_2(i) > 1 - r$, and the algorithm proceeds till the maximum word size B is reached, allowing it to automatically catch corner cases like $a = 0$.

3 Adaptive sampling

The sequence definition we proposed uses the light hierarchy leaf index to directly address into a sequence, resulting in a single sample per light. With this setup, the resulting samples are stratified across all lights, but it is not directly possible to extend the process and its stratification to multiple samples per light.

In order to achieve this, we here propose a simple variant. The idea is to use another sequence $\psi(n)$ to rotate the first, while stratifying across n. That is to say, we redefine ϕ_j as:

$$\phi_j(i, n) = \phi_j(i) + \psi(n) \mod 1 \quad (12)$$

Note that we're free to choose any sequence for ψ, possibly even depending on j. From the purpose of the application of our algorithm, for a fixed n, nothing is changed, as the new term can be absorbed into the shift:

$$\phi_j(i, n) = \Phi_2(i) + (r_j + \psi(n)) \mod 1 \quad (13)$$

In practice, we suggest using a simple low dimensional sequence with desirable blue-noise and low-discrepancy distribution, such as the rank-1 lattice:

$$\psi_(n) = s + n \cdot \alpha \mod 1 \quad (14)$$

where s is an optional shift (potentially set to zero) and α is any irrational number. The choice of α that achieves

the lowest possible discrepancy is $\alpha = 1/\phi$, where ϕ is the well known golden ratio $(\sqrt{5} + 1)/2 \approx 1.61803398875$ (see Figure 2).

Another option is to get rid of the explicit shifts r_j and use a single higher-dimensional sequence $\psi(j, n)$ that is stratified and has blue-noise properties both in space (i.e. across pixels) and in time:

$$\phi_j(i, n) = \Phi_2(i) + \psi(j, n) \quad \mod \ 1 \qquad (15)$$

For small n, this could for example be achieved with a 3d tile of precomputed shifts. Alternatively, ψ could be crafted with the techniques introduced by Heitz and Belcour [2019].

4 Alternative sampling schemes

Another possibility to stratify across multiple dimensions would be to adopt the framework introduced by Grünschloss et al [2012]. This would be achieved considering a single global sequence ϕ_j and, for each leaf i, defining the n-th sample associated with that leaf by enumerating the n-th sequence sample falling into the interval $[i/N_l, (i+1)/N_l)$. In this case, we would need to generalize the sample enumeration technique to finding the minimum across a range $[a, b)$ for a given sample index n. In the following we will be assuming without lack of generality that that $N_l = 2^d$, as the other cases can be handled by taking a domain sized by the smallest power of 2 greater than N_l, and only focusing on the first N_l entries.

Unfortunately, for this sampling scheme using a single 1D Halton sequence $\phi_j = \Phi_2$ would not be very useful. In fact, the radical inverse is such that the d least significant digits l of an integer select an interval, while the higher bits specify the relative sample coordinates inside the interval. In other words, with such a scheme a given leaf i is addressed by $l = \Phi_2^{\leftarrow}(i)$, while all the sequence points inside that leaf have index $l + n \cdot 2^d$, and the actual coordinates of the samples within the leaf depend on n only, as we can see from the following equation:

$$\Phi_2(l + n \cdot 2^d) = b^{-d} \cdot \Phi_2(n) + \Phi_2(l) \qquad (16)$$

This, however, means that for a given value of n, all the samples inside all leaves would be exactly the same: they would have relative coordinates $\Phi_2(n)$. At that point returning the minimum over a given range $[a, b)$ would certainly be easy, as it would be a constant, but it wouldn't be of much use.

A more interesting alternative could be obtained by using a 2D Halton sequence, where the light vertex index i is the first sequence dimension, and the eye vertex index j is the second. The sample enumeration framework would then allow us to ask for the sequence index of the n-th sample corresponding to any eye-light pair (j, i). In practice, the equation providing this index has the form:

$$\phi^{\leftarrow}(j, i, n) = (\Phi_2^{\leftarrow}(i) \cdot x + \Phi_3^{\leftarrow}(j) \cdot y \quad \mod \ N_e N_l) + N_e N_l \cdot n \qquad (17)$$

where x and y are constants. From the perspective of a fixed j, this can be further simplified to:

$$\phi_j^{\leftarrow}(i, n) = (\Phi_2^{\leftarrow}(i) \cdot x + c_j \quad \mod \ N_e N_l) + N_e N_l \cdot n \qquad (18)$$

Finding the minimum value assumed by ϕ_j over a range $[a, b)$ would hence require the evaluation of:

$$\min_{i \in [a,b)} (\Phi_2((\Phi_2^{\leftarrow}(i) \cdot x + c_j \quad \mod \ N_e N_l) + N_e N_l \cdot n)) \qquad (19)$$

Algorithm 2: pseudo-code for our algorithm

```
// first case
if k_r ∈ [a, b):
    return k_r;

// find the most significant bit set of b
b_h = fls(b);

// find the most significant bit set of a
a_h = fls(a);

// find least significant range of bits set of k_r
k_p = ffs(k_r);
k_q = k_p;
while is_set(k_r, k_q + 1):
    k_q = k_q + 1;

// start building m
m = k_r;

// clear all its bits more important than b_h
for i > b_h:
    m = clear(m, i);

// start our left-to-right sweep
i = b_h;
while i ≥ k_p:
    if is_set( m, i):
        m = clear(m, i); // clear the i-th bit
    else:
        m_i = set(m, i); // set the i-th bit

        if m_i < a:
            // try to make m_i a little bigger
            m_i = set(m_i, i + 1);

        m_i = refine( m_i, i + 1, [a, b) );

        if m_i ∈ [a, b):
            return m_i;

    if is_set( a, i):
        m = set( m, i); // keep the i-th bit from a
    i = i - 1;

// sweep right-to-left, each time clearing
// the current bit and setting the next
m = a; // start from m = a
k = b; // mark k as invalid
for i = 0 in B − 1:
    // if i == k_p and we found a valid k ∈ [a, b)
    // we can stop here: by looking further we will not
    // find any other with Φ_2(k) > 1 − r
    if k ∈ [a, b) and i == k_p:
        return k;

    if i > 0:
        m = clear(m, i − 1); // clear the previous bit

    m = set(m, i); // set the i-th bit

    if m < a:
        // try to make m a little bigger
        m = set(m, i + 1);

    m_i = refine( m, i + 1, [a, b) );

    if m_i ∈ [a, b):
        k = m_i;
return k;
```

```
Algorithm 3: refine( m, i, [a, b) )

  Input: a range [a, b), the number m to refine, and the
         length i of the prefix m[0, i) to keep

  // jump to the next non-zero bit of m
  i = ffs( clear_prefix(m, i) );

  // check whether we are done
  while  i < B − 1:
      // jump to the next integer with a 0-bit at index i
      k = m + (1 << i);

      // check whether it's a valid solution
      if  k ∈ [a, b):
          // refine(k, i, [a, b));
          m = k;

      // jump to the next non-zero bit of m
      i = ffs( clear_prefix(m, i + 1) );
  return m;
```

Again, due to the bit reversal nature of Φ_2, this means finding the index i that maximizes the number of rightmost zeros in the argument $(\Phi_2^{\leftarrow}(i) \cdot x + c_j \mod N_e N_l)$, or more formally the index that makes the expression belong to the smallest possible set $I_{s_1 \ldots s_B}$, where the initial set I is now defined as $I = \Phi_2^{\leftarrow}([a, b))$. Unfortunately, this seems rather difficult due to the presence of the modulo arithmetic. We believe, however, that the approach we suggested in the previous section might have more desirable blue-noise distribution and discrepancy properties.

References

CHAITANYA, C. R. A., BELCOUR, L., HACHISUKA, T., PREMOZE, S., PANTALEONI, J., AND NOWROUZEZAHRAI, D. 2018. Matrix Bidirectional Path Tracing. In *Eurographics Symposium on Rendering - Experimental Ideas and Implementations*, The Eurographics Association, W. Jakob and T. Hachisuka, Eds.

GEORGIEV, I., AND FAJARDO, M. 2016. Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*, ACM, New York, NY, USA, SIGGRAPH '16, 35:1–35:1.

GRÜNSCHLOSS, L., RAAB, M., AND KELLER, A. 2012. Enumerating Quasi-Monte Carlo Point Sequences in Elementary Intervals. *Springer Proceedings in Mathematics and Statistics 23* (01).

HEITZ, E., AND BELCOUR, L. 2019. Distributing Monte Carlo Errors as a Blue Noise in Screen Space by Permuting Pixel Seeds Between Frames. In *Eurographics Symposium on Rendering*, The Eurographics Association.

NIEDERREITER, H. 1992. *Random Number Generation and quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

TOKUYOSHI, Y., AND HARADA, T. 2019. Hierarchical Russian Roulette for Vertex Connections. *ACM Trans. Graph. 38*, 4 (July), 36:1–36:12.

VEACH, E. 1997. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University.