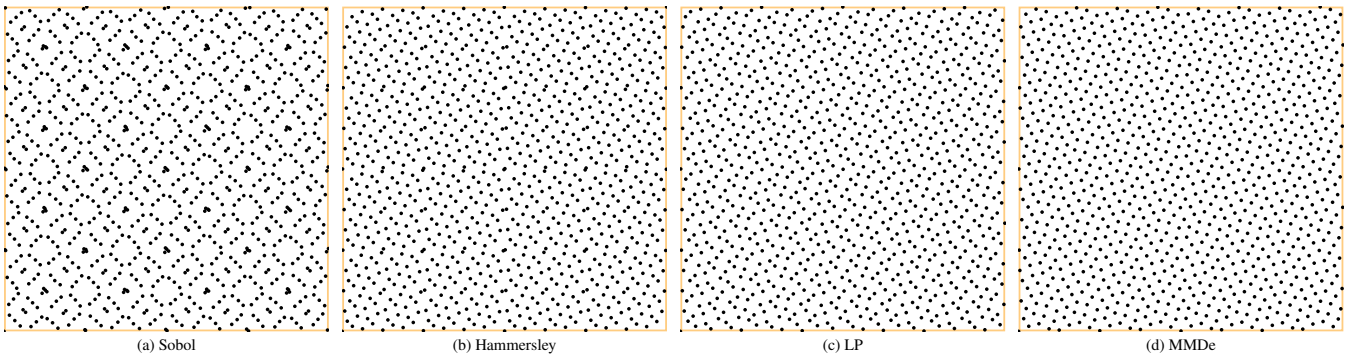


# An Implementation Algorithm of 2D Sobol Sequence Fast, Elegant, and Compact

Abdalla G. M. Ahmed <sup>†</sup> 

Khartoum, Sudan



**Figure 1:** Using only a few lines of code, we can generate (a) vanilla 2D Sobol sample points at  $5\times$  the speed of state of the art implementations. At a negligible additional cost, we can also rearrange the points into (b) an improved distribution, including ones with (c) least discrepancy and (d) maximized minimum distance between points.

## Abstract

We present a novel algorithm to evaluate 2D Sobol samples, bringing the time complexity for  $m$ -bit resolution to  $\mathcal{O}(\log(m))$  instead of  $\mathcal{O}(m)$ , thus gaining tangible performance boost. We take advantage of the geometric structure of the underlying Pascal matrix to factor it into diagonally-running matrices that are efficient to implement using bit-wise operations. We extend the method to inversion in global Sobol sampling. The algorithms form a flexible framework, able to generate several well-known sample sequences as special cases. We compare the speed performance and memory footprint of our algorithms to state of the art implementations.

## 1. Introduction

The impressive renderings (for their time) by Cook et al. [CPC84] arguably marked the beginning of the era of (quasi-)Monte-Carlo-based rendering. Heavily based on sampling, this initiated a long history of research on providing low-cost high-quality sampling patterns. Among the many proposed alternatives come Sobol sequences [Sob67], combining excellent numeric integration performance thanks to their *low-discrepancy* properties [Nie92] (cf. random sampling), very high generation rates thanks to their binary nature (cf. Halton sequence), relatively low memory footprint (cf. lookup-based methods), and excellent scaling with dimension (cf.

blue noise). These features, among others, arguably make Sobol sequences lead the competition by far, providing low-cost high-performance sampling patterns for building reliable and versatile rendering samplers. Indeed, over its successive editions, the rendering reference book PBRT [PJH23] used three different Sobol-based samplers as its default.

A Sobol sequence is obtained by linearly mapping sample indices to coordinates using an ordered set (tuple)

$$\left( I = \begin{matrix} \text{[diagonal matrix]} \\ \dots \end{matrix}, P = \begin{matrix} \text{[Pascal matrix]} \\ \dots \end{matrix}, \dots \right) \quad (1)$$

of binary matrices, one matrix per dimension, displayed here using (1) black and (0) gray dots for better visibility. These matri-

<sup>†</sup> abdalla\_gafar@hotmail.com

ces are computed using special algebraic recipes that derive subsequent columns from initial ones. Understanding these recipes is not needed for this paper, and the interested reader is referred to Sobol [Sob67] or Bratley and Fox [BF88]. Once the matrices are there, each coordinate of the  $v$ th sample, binary encoded as

$$v = \sum_{i=1}^{\infty} v_i 2^{i-1} = \dots v_3 v_2 v_1, \quad (2)$$

is obtained by multiplying the respective matrix by a bit-reversed column vector representation of  $v$ . For example,

$$\left( \left( \begin{pmatrix} x_1 \\ x_2 \\ \vdots \end{pmatrix}, \begin{pmatrix} y_1 \\ y_2 \\ \vdots \end{pmatrix} \right) \right) = (I, P) \begin{pmatrix} v_1 \\ v_2 \\ \vdots \end{pmatrix} \quad (3)$$

gives the 2D projection of the  $v$ th sample. We emphasize that bits of each coordinate in Eq. (3) are generated by the respective matrix in Eq. (1), and the bits of the indexing vector are in a reverse ordering, so  $v_1$  is the least significant bit of the sample index. These computations use Galois Field 2, that is, modulo-2 addition, which translates into fast primitive bit operations that account for the exceptional speed performance of Sobol sequences.

In this paper, we are mostly interested in the first two matrices shown in Eq. (1), known as ( $I$ ) the identity and ( $P$ ) Pascal matrices, which are invariant, unlike the subsequent matrices that may vary by user initialization. Indeed, of the three Sobol-based samplers of PBRT, two, the (0, 2)-Sequence Sampler and ZSampler, are based only on this 2D pair of Sobol dimensions, while the third, Global Sobol sampler, uses these two dimensions to sample the image plane, posing a requirement to invert them for sample indices.

The use of Sobol sequences in computer graphics was primarily advocated and mostly curated by Keller and colleagues, providing algorithms, lookup tables, and efficient code for generation [KK02], scrambling [FK02], and inversion [GRK12], as well as many articles explaining how to use and manipulate these sequences [Kel06; Kel13; GHSK08].

With its unrivaled speed performance, comparable to quasi-random number generation, Sobol reigned for long as the fastest imaginable low-discrepancy sequence, and these algorithms seemed optimal. Quite recently, however, Ahmed et al. [ASHW23] introduced  $\xi$ -sequences: a family of low-discrepancy sequences that—for the same numerical performance—offer a significant reduction of computational cost (speed and memory footprint) compared to Sobol. In the same paper, they demonstrate a linear mapping between these sequences and Sobol, suggesting the possibility of having better algorithms for the latter.

In this paper, we present a novel algorithm for evaluating the second ( $P$ ) dimension of the Sobol sequence, offering a significant performance boost. We extend the idea to inversion, offering an even better reduction of computational cost, and clearing the main bottleneck of global Sobol sampling. In contrast to inversion, sample generation is usually far from being the bottleneck in rendering of very complex scenes. Improving the sampling rate, though, would not only still count, but also makes it possible to implement some improvements to the distribution quality of the samples, as we will discuss later.

## 2. Related Work

Sobol sequences belong to a larger class of low-discrepancy nets and sequences, a concept due to Neiderreiter [Nie87] that encompasses earlier works by Hammersley [Ham60], Sobol [Sob67], and Faure [Fau82], as well as new constructions that appeared thereafter. The first two-dimensions of Sobol sequences constituted a “(0, 2)-sequence in base 2” in Neiderreiter’s notation, which constitutes a hierarchical binary tree of  $(0, m, 2)$ -nets in base 2. The interested reader is referred to Niederreiter [Nie92] for more details. Before the recent introduction of  $\xi$ -sequences [ASHW23], the  $(I, P)$  Sobol sequence, also known as the 2D Faure sequence, was the one and only known explicit matrix-based construction of a (0, 2)-sequence, and all alternatives were derived from it either by Tezuka-scrambling the generator matrices [Tez94] or Owen-scrambling the resulting points [Owe95; KK02].

The works most related to ours are already mentioned in the introduction, as this area may be considered stable since 2012. In the following subsections we discuss technical details in more depth, and cite some other related works.

### 2.1. Common Implementations

A straightforward computation of a constituent dimension of the  $v$ th Sobol samples initializes an empty accumulator, iterates through bits of  $v$ , from least significant position to the most significant set bit, and xor-add the respective column of the matrix to the accumulator if the bit is set [PJH23, 8.7 Sobol’ Samplers]. The columns of the matrices are stored in a reverse order to imply bit-reversal. We trust this is optimal for an arbitrary matrix, as may be considered the case with the high-dimension constituent matrices of the Sobol sequence. Some matrices, however, may embody a structure that admits optimization.

The current state of the art in generating 2D Sobol samples, due to Kollig and Keller [KK02], eliminates the storage of the two  $(I, P)$  matrices. For the identity matrix, they use this common loop-less C-language sequence

```
inline uint32_t J(uint32_t i) {
    i = ( i << 16) | ( i >> 16);
    i = ((i & 0x00FF00FF) << 8) | ((i & 0xFF00FF00) >> 8);
    i = ((i & 0x0F0F0F0F) << 4) | ((i & 0xF0F0F0F0) >> 4);
    i = ((i & 0x33333333) << 2) | ((i & 0xCCCCCCCC) >> 2);
    i = ((i & 0x55555555) << 1) | ((i & 0xA0000000) >> 1);
    return i;
}
```

of bit operations for implementing the implicit bit reversal. The function is so named  $J$  here because it effectively implements multiplication by the anti-diagonal matrix

$$J = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \quad (5)$$

For the Pascal matrix, they compute the columns on the fly using the original recurrence formula

$$c_j = c_{j-1} \oplus 2^{-1} c_{j-1} \quad (6)$$

that defines the matrix in Sobol construction, where  $\oplus$  is bit-wise vector xor-addition, and  $2^{-1}$  means shifting the column down by 1 in the semantics of this context.

The Kollig and Keller implementation above is not the only

common implementation algorithm of the 2D Sobol sequence. Indeed, Antonov and Saleev [AS79] early presented an exceptionally fast implementation by using a Gray-code scanning order of sample indices. Recently, Helmer et al. [HCK21] presented an Owen-scrambling implementation algorithm that embeds another exceptionally fast generation algorithm of the 2D Sobol sequence. The problem with these algorithms is that they use previous samples to generate subsequent ones, hence the whole set of samples has to be generated at once, and in a specific order, which is not suitable for some samplers, e.g., Global Sobol and Z, that require random access to samples in the sequence. These techniques, however, provide a good reference for performance benchmarking.

## 2.2. Global Sobol Sampling

In contrast to 2D-based samplers, the Global Sobol Sampler [PJH23] uses a high-dimensional Sobol sequence to sample the whole scene domain, including the image plane, as a single hypercube. To our knowledge, this idea was first proposed by Grünschloß et al. [GRK12], and it exhibits superior convergence rates [PJH23] and decent error diffusion [AW20], which lead to its adoption as the default sampler in PBRT's 2nd edition [PJH16], before being replaced by the ZSampler [AW20] in the third edition [PJH23]. Yet, the global Sobol sampler maintains the best convergence rate we are aware of.

Global Sobol sampling marginalizes the cost of sampling the first two dimensions, especially in complex scenes. There is, however, a new cost of inverting these two dimensions that comes into play. It works as follows: The image plane is scaled to the unit square. For each pixel area, we need to retrieve the indices of the samples falling in that area, and use them back to compute the exact sample locations in the pixel, as well as their high-dimensional constituents. This operation needs to be done for every single sample, hence contributes substantially to the cost of global Sobol sampling.

The current state of the art in inversion, due to Grünschloß et al. [GRK12], uses a complex piece-wise process, and involves a lot of lookup tables: two per image resolution. The inversion rate in this process is relatively low, less than 30% of the generation rate, making a substantial bottleneck in the current PBRT implementation of global Sobol sampling.

In contrast, the family of  $\xi$ -sequences recently introduced by Ahmed et al. [ASHW23] exhibits a very high inversion rate, exceeding its own (superior) generation rate. This prompts that there might exist better algorithms for Sobol, given the linear mapping between the two.

## 2.3. Net-to-Sequence Mapping

The Tezuka scrambling [Tez94] mentioned earlier multiplies the  $(I, P)$  generator pair of the Sobol sequence from the left by a pair  $(L_x, L_y)$  of arbitrary lower-triangular matrices to obtain a new pair

$$(L_x, L_y P) \quad (7)$$

of generator matrices that retains the  $(0, 2)$ -sequence properties when used to replace  $(I, P)$  in Eq. (3). Thus, Tezuka scrambling

can randomly sample a large set of Sobol-like variants. Quite recently, Ahmed et al. [ASHW23] introduced a digital net-to-sequence translation algorithm that can compute a specific scrambling pair  $(L_x, L_y)$  to reproduce a given matrix-generated  $(0, m, 2)$ -net, but reordering the points into a hierarchical Sobol-like  $(0, 2)$ -sequence. For a given  $(C_x, C_y)$  pair of generator matrices that is known to produce a  $(0, m, 2)$ -net, the algorithm may be summarized in the following algebraic refactoring

$$(C_x, C_y)V = (J, C_y C_x^{-1} J) J C_x V \quad (8)$$

$$= (J, LU) J C_x V \quad (9)$$

$$= (JU^{-1}, L) U J C_x V \quad (10)$$

$$= (JU^{-1} P J, L P J) J P^{-1} U J C_x V \quad (11)$$

$$= \underbrace{(JU^{-1} P J)}_{L_x} \underbrace{(L P J)}_{L_y} \underbrace{J P^{-1} U J C_x V}_{V'} \quad (12)$$

which takes advantages of many properties, including (i) the LU-decomposability of any generator matrix that pairs with  $J$  into a  $(0, m, 2)$ -net, (ii) the self-inversion of  $P$ , (iii) the

$$P J P J P J = I \quad (13)$$

identity, and (v) the invertibility (full-rank) of  $J P U J C_x$  that maintains the full set of indices. The interested reader is referred to Ahmed et al. [ASHW23] for detailed analysis and proofs. Using this refactoring, it is possible to reorder some interesting  $(0, m, 2)$ -nets into a Sobol-like sequence. We will integrate some of these later into our algorithm.

## 3. Fast 2D Sobol

In this section we present our suggested algorithms for boosting the performance of Sobol-based samplers, addressing different aspects through successive subsections.

### 3.1. Generation


The key observation that prompted our work is that the Pascal matrix  $P$ , as visualized in Eq. (1), exhibits a geometrical structure, namely a Sierpiński triangle, that embodies many partial translation symmetries. For example, shifting the  $32 \times 32$  Pascal matrix 16 steps to the right gives a neat overlap


(14)

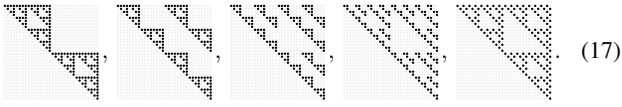
that may be eliminated if we xor-add the original and shifted matrices. Similar overlaps,


(15)

occur at other power-of-two shifts, but are not as perfect as the 16-steps one, leading to setting some clear entries in the matrix. However, perfect overlaps,

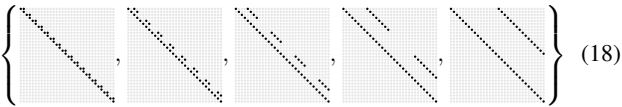

(16)

may be obtained if we first mask every other band of columns of a width equal to the shift. We obtain the following result for xor-addition of the original and shifted matrices:


(17)

Finally, we note that concatenating these mask-shift-xor-add operations, in any order, would completely clear the off-diagonal elements of the matrix.

Through this sequence of visual thoughts we were able to conceive our algorithm, as we present next. We first translate these geometric descriptions into algebraic manipulations, which is straightforward: (i) shifting to the right is equivalent to multiplying from the right by an equally shifted identity matrix, (ii) masking is equivalent to clearing the respective diagonal bits, (iii) xor-addition to the original matrix is equivalent to including the main diagonal in the multiplied matrix, and (iv) concatenating the operations is equivalent to consecutively multiplying from the right by the respective matrices. Since this would clear the matrix, the product of so-constructed matrices constitutes the inverse Pascal matrix. Notably, the Pascal matrix inverts itself, hence it may be factored into the set


(18)

of factors. The essence of this factoring is that multiplying such diagonally running matrices by a column vector directly translates into simple masking/shifting of the vector by the same mask/shift of the diagonals, and the result is xor-summed over the diagonals of a multi-diagonal matrix. This leads to the following compact algorithm

```
inline uint32_t P(uint32_t v) {
    v ^= v << 16;
    v ^= (v & 0x00FF00FF) << 8;
    v ^= (v & 0x0F0F0F0F) << 4;
    v ^= (v & 0x33333333) << 2;
    v ^= (v & 0x55555555) << 1;
    return v;
} (19)
```

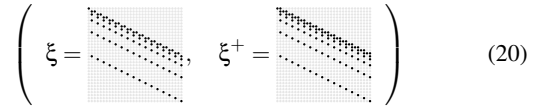
for computing the second Sobol dimension, where  $v$  is a bit-reversed representation of the sample index. This is especially useful if the  $(x, y)$  pair of coordinates is evaluated together, which is quite common, in which case  $x$  would be computed first using fast bit reversal, then plugged as  $v$  in Listing (19). If the second ( $y$ ) component has to be evaluated separately then we may use Listing (4) first for bit reversal.

Computing 2D Sobol samples accounts for almost all the computational cost of PBRT's (0, 2)-sequence sampler, and contributes significantly to the cost of the ZSampler [AW20], hence this makes a tangible difference to the sampling cost in these samplers.

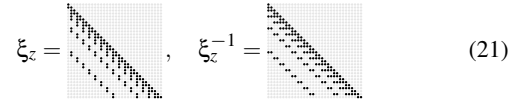
### 3.2. Inversion

Towards fast inversion of 2D Sobol sequence, we start by analysing the canonical  $\xi$ -sequence to reveal the reasons behind its superior

performance. It replaces  $(I, P)$  by the

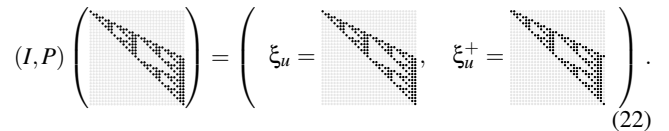
$$\left( \begin{array}{c} \xi \\ \xi^+ \end{array} \right) \quad (20)$$


pair of matrices. A fast inversion advantage of this sequence is already reflected in its generator matrices. To reveal it, we interleave  $x$  and  $y$  bits for Morton ordering, making  $x$  more significant for best visualization, then the rows of the two matrices have to be interleaved accordingly to get

$$\xi_z = \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array}, \quad \xi_z^{-1} = \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array} \quad (21)$$


for transforming a sequence number to a Morton index and back. The essence of this lower-triangular pair of matrices is that sample points sharing leading bits in their Morton index would share leading bits in their (bit-reversed) sequence numbers, and the reverse. This means that all samples in the pixel would share the same suffix in their sequence numbers, and are linearly ordered in their appended higher-ordered bits. Beyond that, the two matrices look ripe for diagonal-matrix factoring, though Ahmed et al. took advantage instead of another property that the matrices are made up of a stair-like pair of columns.

This property of lower-triangular Morton matrix in  $\xi$ -sequences is applicable to the 2D Sobol sequence by multiplying an appropriate reordering matrix,

$$(I, P) \left( \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array} \right) = \left( \begin{array}{c} \xi_u \\ \xi_u^+ \end{array} \right) \quad (22)$$


The interested reader is referred to Helmer et al. [HCK21, Supplemental Materials] or Hofer and Pirsic [HP11] for an explicit construction of the reordering matrix, but as it turns out, the resulting pair of matrices coincide with the upper-triangular factors of the  $\xi$  pair of matrices [ASHW23], hence the names in Eq. (22). This pair of matrices generates the same 2D Sobol sequence, but permutes the order of points over power-of-two octaves, which should not affect the rendering outcomes in global Sobol sampling.

Our initial vision for fast Sobol inversion was to completely replace  $(I, P)$  by their  $(\xi_u, \xi_u^+)$  counterparts, which requires multiplying all high-dimensional matrices by the same reordering matrix  $\xi_u$ , as described in [ASHW23]. Bearing a resembling geometric structure to  $P$ , the generator matrices can easily be factored similarly to diagonally-running matrices, e.g.,

$$\xi_u = \begin{array}{c} \dots \\ \dots \\ \dots \\ \dots \\ \dots \end{array}, \quad (23)$$


with a slight difference that the factors are no longer commuting like  $P$ 's, and have to be applied in a fixed order.

Not to our favor, however, the absence of the lower-triangular factor  $\xi_l$  played a detrimental role, as the resulting inverse Morton-

ordering matrix

$$\xi_{uz} = \begin{matrix} \text{[Matrix with 5 diagonal bands]} \end{matrix} \implies \xi_{uz}^{-1} = \begin{matrix} \text{[Matrix with 5 diagonal bands]} \end{matrix} \quad (24)$$

is skewed; that is, similar blocks are not aligned along a single axis as needed in our method. Despite the evident self-similar structure, we failed to factor this inversion matrix into a reasonable number of diagonally running factors, ideally 5. While we made many attempts, we do not claim exhaustion, and a feasible factoring may still be available. Alternatively, there is still an option of falling back to conventional matrix-vector multiplication for inversion. While we trust this would still be faster than the current state of the art, and saves on lookup tables, we were not satisfied with this option. Indeed, after taking the effort of reordering the higher-dimensional tables, it is arguably more feasible to migrate altogether to a  $\xi$ -based sampler.

After a lot of experimentation we arrived at a satisfactory solution; notably, while attempting to prove the opposite. We proceed to present our proposed inversion algorithm while discussing the steps. We start by assembling a conversion/inversion pair of matrices akin to the Morton ordering ones above. However, instead of interleaving the  $x$  and  $y$  bits, we choose a more convenient combination by keeping the bits of each coordinate together:

$$IP = \begin{matrix} \text{[Matrix with 2 diagonal bands]} \end{matrix}, \quad IP^{-1} = \begin{matrix} \text{[Matrix with 2 diagonal bands]} \end{matrix} \quad (25)$$

which is encoded as

$$\text{uint32\_t } v = ((p.x \ll 16) | p.y) \ll (16 - m); \quad (26)$$

in practice, which indexes the bottom-left corner of pixel  $p$  in a  $2^{-16} \times 2^{-16}$  grid resolution of the unit-size image, in a row-major ordering. Not only easier to implement, but this  $I : P$  concatenated matrix makes it much easier to see the way forward. For example, we were easily able to find the inverse by hand using the block matrices.

Factoring the inverse matrix into diagonally running factors is straightforward: we first split the lower-triangular diagonal factor, then factor the upper part akin to  $P$ :

$$\begin{matrix} \text{[Matrix with 5 diagonal bands]} \end{matrix}, \quad (27)$$

which translates to

$$\begin{aligned} \text{inline uint32\_t } IP\_inv(\text{uint32\_t } v) \{ \\ v \wedge= (v \& 0x3333) \ll 2; \\ v \wedge= (v \& 0x5555) \ll 1; \\ v \wedge= (v \& 0x0F0F) \ll 4; \\ v \wedge= (v \& 0x00FF) \ll 8; \\ v \wedge= v \gg 16; \\ \text{return } v; \\ \} \end{aligned} \quad (28)$$

in code. The key trick now is to translate this index in  $(I, P)$  ordering into a corresponding index in  $(\xi_u, \xi_u^+)$  ordering, taking advantage of the linear mapping between the two. This requires multiply-

ing the index by

$$\xi_u^{-1} = \begin{matrix} \text{[Matrix with 5 diagonal bands]} \end{matrix}, \quad (29)$$

which we may obtain directly from Eq. (23) by inverting the factors and inverting their order. The corresponding code

$$\begin{aligned} \text{inline uint32\_t } xi\_u\_inv(\text{uint32\_t } v) \{ \\ v \wedge= ((v \& 0x000FFFF) \ll 8) \wedge ((v \& 0x00000FF) \ll 16); \\ v \wedge= ((v \& 0x00FF00FF) \ll 4) \wedge ((v \& 0x00F000F) \ll 8); \\ v \wedge= ((v \& 0x0F0F0F0F) \ll 2) \wedge ((v \& 0x03030303) \ll 4); \\ v \wedge= ((v \& 0x33333333) \ll 1) \wedge ((v \& 0x11111111) \ll 2); \\ \text{return } v; \\ \} \end{aligned} \quad (30)$$

achieves the required index translation. Now we have the index of a sample inside pixel  $p$ , not necessarily the first one, in  $(\xi_u, \xi_u^+)$  ordering. The essence of this ordering, as mentioned, is that the global sample index is partitioned into a pixel index and the index of the sample inside the pixel, hence we can easily retrieve the pixel index

$$v = J(v) \& ((1 \ll m2) - 1); \quad (31)$$

in a normal ordering, still using  $(\xi_u, \xi_u^+)$  ordering, where

$$m2 = 2 \cdot m \quad (32)$$

is the overall image bit resolution, split equally between the two axes.

Now we proceed to retrieve an  $i$ th sample inside the pixel, as would be queried by the rendering integrator. The index of the queried sample inside the pixel is given in  $(I, P)$  order, so we have to translate it in  $(\xi_u, \xi_u^+)$  ordering before appending it to the pixel index retrieved above. This step is optional: if we just append the sample number then we would get exactly the same power-of-two-sized block of samples in a different order, which should not make a difference. However, we find it a good practice in presenting a drop-in replacement to ensure identical outcomes. The conversion of sample number to  $(\xi_u, \xi_u^+)$  ordering involves the same  $\xi_u^{-1}$  mapping in Listing (30), with two differences. First, there is a chance that the global sample index may exceed  $2^{32}$ , so it is more appropriate to perform this and the following step in 64-bit resolution. Secondly, the sample number is provided in normal ordering, requiring bit reversal. Since the final outcome is also in normal bit ordering, it makes more sense to use  $J\xi_u^{-1}J$ , a half-cycle rotation of the matrix, hence

$$\begin{aligned} \text{inline uint64\_t } Jxi\_u\_invJ(\text{uint64\_t } v) \{ \\ v \wedge= ((v \& 0xFFFFFFFF00000000ll) \gg 16) \wedge ((v \& 0xFFFF000000000000ll) \gg 32); \\ v \wedge= ((v \& 0xFFFFFFFF00000000ll) \gg 8) \wedge ((v \& 0xFF000000FF000000ll) \gg 16); \\ v \wedge= ((v \& 0xFF00FF00FF00FF00ll) \gg 4) \wedge ((v \& 0xF000F000F000F000ll) \gg 8); \\ v \wedge= ((v \& 0xF0F0F0F0F0F0F0F0ll) \gg 2) \wedge ((v \& 0xC0C0C0C0C0C0C0C0ll) \gg 4); \\ v \wedge= ((v \& 0xCCCCCCCCCCCCll) \gg 1) \wedge ((v \& 0x8888888888888888ll) \gg 2); \\ \text{return } v; \\ \} \end{aligned} \quad (33)$$

does the appropriate transformation, while

$$\text{sampleNo} = Jxi\_u\_invJ(\text{sampleNo} \ll m2) \& (0xFFFFFFFFFFFFFFFFllu \ll m2); \quad (34)$$

applies it to a pixel sample number. The pixel index and pixel sample number in  $(\xi_u, \xi_u^+)$  ordering are then combined, and the final step is to transform back to the  $(I, P)$  ordering using the  $\xi_u$  in

Eq. (23), encoded as

```
inline uint64_t Jxi_uJ (uint64_t v) {
    v ^= (v & 0xCCCCCCCCCCCCllu) >> 1;
    v ^= (v & 0xF0F0F0F0F0F0F0llu) >> 2;
    v ^= (v & 0xFF00FF00FF00FFllu) >> 4;
    v ^= (v & 0xFFFF0000FFFF00llu) >> 8;
    v ^= (v & 0xFFFFFFFF000000llu) >> 16;
    return v;
}
```

(35)

using normal bit ordering and 64-bit resolution. Finally, the function

```
inline uint64_t invIP(int m, const Point &p, uint64_t sampleNo) {
    if (m == 0) return 0;
    int m2 = m << 1;
    uint32_t v = ((p.x << 16) | p.y) << (16 - m);
    v = IP_inv(v);
    v = xi_u_inv(v);
    v = bitReverse(v);
    v &= (1 << m2) - 1;
    sampleNo = Jxi_u_invJ(sampleNo << m2);
    sampleNo &= 0xFFFFFFFFFFFFFFFFllu << m2;
    return JxiJ(v | sampleNo);
}
```

(36)

sums up the preceding steps. This concludes our implementation of Sobol inversion to retrieve sample indices. While much faster and compact than state of the art, we still think there might be a small room for further optimization.

### 3.3. 2D Sample Distribution

Up until now, we are still generating the 2D Sobol sequence as is, without altering the sample distribution. Nets derived from 2D Sobol sequence are not considered the best: there are constructions of standalone nets that are believed to be better, including the Hammersley [Ham60]

$$(X, Y) = (J, I)V \quad (37)$$

Larcher-Pillichshammer [LP03]

$$(X, Y) = \left( J, C_{LP} = \begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix} \right) V, \quad (38)$$

and Gray [AW21] family of nets are widely believed to be superior by different quality measures. Grünschoß et al. [GHSK08] presented a net construction generator pair

$$\left( J, C_{MMDe} = \begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix} \right) \quad (39)$$

that is believed to generate  $(0, m, 2)$ -nets with the largest maximized minimum distance (MMD) between points among all matrix generated nets with even  $m$ , and a similar construction for odd  $m$ . Pharr et al. [PJH16] attempted a sampler for PBRT employing these nets, but the model was far from optimal, and retired in the following edition [PJH23]. The problem is two fold: the same net is replicated over all pixels, raising well known aliasing flags, and the high dimensions are sampled using 2D samples like the  $(0, 2)$ -Sampler, losing the advantages of global sampling.

It would be great if it is possible to integrate these good 2D sampling distribution into the Global Sobol Sampler, which is possible using the net-to-sequence mapping of Ahmed et al. [ASHW23] mentioned in Section 2.3. That is, the LU factors of a generator matrix paired with  $J$  are embedded into lower-triangular scrambling matrices of the Sobol sequence. This makes it possible to integrate redistributing the pixel samples favorably as a post-processing step

after generating the samples. Doing so, however, would undo all our preceding optimization effort, so we aim at a better treatment.

We start by noting that

$$L_x = JU^{-1}PJ = JU^{-1}JJPJ, \quad (40)$$

hence  $JPJ$  is a common factor to all the digital re-distributions that we may want to handle first. As pointed out earlier for  $J\xi_u J$ , enclosing between two  $J$  matrices is just a  $180^\circ$  rotation of the matrix, leading to an algorithm

```
inline uint32_t JPJ(uint32_t v) {
    v ^= v >> 16;
    v ^= (v & 0xFF00FF00) >> 8;
    v ^= (v & 0xF0F0F0F0) >> 4;
    v ^= (v & 0xCCCCCCC) >> 2;
    v ^= (v & 0xAAAAAAAA) >> 1;
    return v;
}
```

(41)

quite similar to the  $P$  one in Listing (19). This is then multiplied from left by the  $(JU^{-1}J, L)$  pair of lower-triangular scrambling matrices of the desired distribution. There is a small caveat here, though, that needs to be handled appropriately. The Pascal matrix  $P$  in Eq. (12) is clipped to the bit resolution  $m$  of the target net. For example, for a  $2^6$  Hammersley net, we use

$$J \begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix} = \begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}, \quad (42)$$

rather than the

$$\begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix} \quad (43)$$

we find at the top of the 32-bit  $JPJ$ . This is a bit confusing, because the scrambling applies to the most significant bits. It took us a while to identify that the appropriate scrambling bits are actually the ones in the bottom rows of the  $JPJ$  matrix, and the vector needs to be brought down. Specifically,

```
inline uint32_t JPJ(uint32_t v, int m) {
    return (JPJ(v >> (32 - m)) << (32 - m)) | (v & (0xFFFFFFFF >> m));
}
```

(44)

achieves the task, leading to

```
inline void hammersley(Points &p) {
    int m = builtin_ctz(p.size());
    for (uint32_t i = 0; i < p.size(); i++) {
        uint32_t x = J(i);
        uint32_t y = P(x);
        p[i] = {JPJ(x, m), JPJ(y, m)};
    }
}
```

(45)

for the Hammersley net optimization. That is, the points of a Sobol sequence are automatically rearranged in a Hammersley net of the desired size  $2^m$ . Ideally, this would be the overall number of samples for the whole image; i.e., (number of pixels)  $\times$  (samples per pixel). If more or fewer samples than the designated  $m$  are taken, they would lose the optimal net distribution, but still retain the net properties of the underlying Sobol sequence.

The Hammersley net distribution is the easiest, but by no means the only attainable one. Fortunately, though not quite surprisingly, all the interesting net classes enumerated above admit simple diagonal factoring, making them easy to integrate in our code. For the LP nets, for example, we only have an all-ones  $U$  factor. The inverse of this, notably, is a Gray code matrix

$$G = \begin{matrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}, \quad (46)$$

which is already in the desired form, encoded as

```
inline uint32_t G(uint32_t x, int m) {
    uint32_t v = JPJ(x >> (32 - m));
    v ^= v >> 1;
    return (v << (32 - m)) | (x & (0xFFFFFFFF >> m));
}
```

(47)

to give the following code for LP-net rearrangement of the Sobol sequence:

```
inline void LP(Points &p) {
    int m = builtin_ctz(p.size());
    for (uint32_t i = 0; i < p.size(); i++) {
        uint32_t x = J(i);
        uint32_t y = P(x);
        p[i] = {G(x, m), JPY(y, m)};
    }
}
```

(48)

For the MMD net rearrangement, we LU-decompose the middlemost  $6 \times 6$  matrix, then invert and rotate the upper factor to get

$$(L_x, L_y) = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \quad (49)$$

already in a usable form. However, the bit-masks need to be shifted by  $(m - 6)/2$  from the bottom to center the matrix at the  $m \times m$  window. Here is the complete code

```
inline uint32_t mmdX(uint32_t x, int m) {
    uint32_t v = JPJ(x >> (32 - m));
    int padding = (m - 6) >> 1;
    v ^= (v & (0x10 << padding)) >> 1;
    return (v << (32 - m)) | (x & (0xFFFFFFFF >> m));
}

inline uint32_t mmdY(uint32_t y, int m) {
    uint32_t v = JPY(y >> (32 - m));
    int padding = (m - 6) >> 1;
    v ^= (
        ((v & (0x30 << padding)) >> 1) ^
        ((v & (0x08 << padding)) >> 2)
    );
    return (v << (32 - m)) | (y & (0xFFFFFFFF >> m));
}

inline void mmd(Points &p) {
    int m = builtin_ctz(p.size());
    for (uint32_t i = 0; i < p.size(); i++) {
        uint32_t x = J(i);
        uint32_t y = P(x);
        p[i] = {mmdX(x, m), mmdY(y, m)};
    }
}
```

(50)

split into parts to accommodate the case of separate computation of  $x$  and  $y$ .

Figure 1 shows samples of the three distributions generated by our code listings above, which is also available in the supplementary materials.

### 3.4. Higher Dimensions

While our focus in this paper is on the first two dimensions, we briefly mention a “free” related opportunity for improving the distribution in high dimensions as well. The current state-of-the-art tables of Sobol matrices, due to Joe and Kuo [JK08], use upper-triangular matrices, as defined in Sobol construction. These may be scrambled with random lower-triangular matrices to gain better coverage. The process is offline, so may be considered free.

## 4. Evaluation

We now benchmark our implementation against state of the art to evaluate the gains and costs.

**Table 1:** Generation rates, in million 2D sample per second (MSPS), of different sequence generation codes, including non-Sobol ones. The benchmarks were conducted on a contemporary laptop with an Intel Core i7-10510U CPU.

Algorithm	Rate	$\times$ Ref
PBRT [PJH16]	59	1.00
Ours (vanilla)	340	5.75
Ours (+ MMD rearrangement)	114	1.94
Helmer [HCK21]	466	7.87
Gray Code Scan Order [AS79]	1119	18.91
$\xi$ [ASHW23]	84	1.42
Random (std::mt19937) [MN98]	185	3.13

### 4.1. Speed

This is possibly the most important measure for a low-level code element like Sobol samples. In Table 1 we benchmark the performance of our generation algorithm against various state-of-the-art alternatives. For Helmer [HCK21] we removed the randomization code, leaving only the Sobol generation part. While it and Gray-code scanning are not well-suited for real-time rendering in samplers requiring random access, it serves as a good reference.

The advantage of our method is evident, even with the rearrangement of the samples. This gain is close to the ratio of time complexity:  $\mathcal{O}(m)$  columns vs  $\mathcal{O}(\log(m))$  diagonals, hence may increase for 64 bits. Another factor contributing to our advantage is the elimination of loops.

Our algorithm truly surpasses previous algorithms when it comes to inversion, attaining a rate of 400 MSPS, compared to less than 20 MSPS for PBRT code. This is not only more than 20 times faster, but the 20 MSPS rate is really slow, and may be considered a bottleneck. In the supplementary materials we provide the code used for this benchmarking, which includes more variations.

### 4.2. Memory footprint

The state of the art generation of 2D Sobol is already table-less, as we discussed earlier. For the inversion, however, our table-less implementation, compared to state of the art, saves 1352 64-bit integers in two 2D-structured lookup tables. This is a considerable saving, especially for GPU processing.

### 4.3. Quality

The LP distribution in Figure 1(c) is known to bear the least discrepancy among digital nets [LP03], while the MMD one in Figure 1(d) is established to bear the largest minimum distance between points among digital nets. Thus, our algorithm attains the maximum quality in these measures, and avails this quality to different Sobol-based samplers at a negative cost compared to the available implementations. There is a long standing belief that these two quality measure are yielding. Validating these beliefs, however, is far beyond the scope of our work, and we can not make a claim of improving the actual rendering quality. What we can claim, however, is that our distribution rearrangements are effectively costless,

making it possible to empirically evaluate these quality measures extensively.

#### 4.4. Coding Complexity

Almost all the actual code is already in the paper, and may directly be copied from here. The code in the supplementary materials also includes more variations.

#### 5. Conclusion

In this paper we presented a novel implementation of the 2D Sobol sequence, covering both generation and inversion. Our algorithm offers significant gains in both speed and memory footprint, and avails further improvements, all at no cost to mention. The actual cost we had to pay for these improvements is a deeper understanding of the matrix structure, which emphasises the importance of in-depth study of the tools we use.

It is arguable that the current 60 MSPS is already enough for practical purposes. However, for a process that has to be repeated billions of times per single image, every performance improvement is welcome. We actually demonstrated the value of our speed performance improvements by integrating quality improvements that would have been considered infeasible before.

We learned a lot through this narrow scope, and could see multiple opportunities for future follow up. For example, we learned that diagonal factoring pays off, so it would be great to see automatic algorithms for that, instead of our manual inspection. The way they are computed suggests that other Sobol matrices may have simple factoring like  $P$ .

Finally, we want to emphasize the important role of “competition” in research development. A part of our motivation in this work was to defend the classic Sobol against the new  $\xi$ -sequence. Now we are already considering the opposite, and actually started searching for further optimization of the latter. This spirit may especially be exploited among students, and many such real-life low-level coding problems may be pushed via student competitions.

#### References

- [AS79] ANTONOV, I.A. and SALEEV, V.M. “An economic method of computing LP-sequences”. *USSR Computational Mathematics and Mathematical Physics* 19.1 (1979), 252–256. ISSN: 0041-5553. DOI: [https://doi.org/10.1016/0041-5553\(79\)90085-5](https://doi.org/10.1016/0041-5553(79)90085-5), 3, 7.
- [ASHW23] AHMED, ABDALLA G. M., SKOPENKOV, MIKHAIL, HADWIGER, MARKUS, and WONKA, PETER. “Analysis and Synthesis of Digital Dyadic Sequences”. *ACM Trans. Graph.* 42.6 (Dec. 2023). DOI: [10.1145/3618308](https://doi.org/10.1145/3618308) 2–4, 6, 7.
- [AW20] AHMED, ABDALLA G. M. and WONKA, PETER. “Screen-Space Blue-Noise Diffusion of Monte Carlo Sampling Error via Hierarchical Ordering of Pixels”. *ACM Trans. Graph.* 39.6 (Nov. 2020). ISSN: 0730-0301. DOI: [10.1145/3414685](https://doi.org/10.1145/3414685). [10.1145/3414685](https://doi.org/10.1145/3414685). [3417881](https://doi.org/10.1145/3414685) 3, 4.
- [AW21] AHMED, ABDALLA G. M. and WONKA, PETER. “Optimizing Dyadic Nets”. *ACM Trans. Graph.* 40.4 (July 2021). ISSN: 0730-0301. DOI: [10.1145/3450626](https://doi.org/10.1145/3450626). [3459880](https://doi.org/10.1145/3450626). URL: <https://doi.org/10.1145/3450626>. [3459880](https://doi.org/10.1145/3450626) 6.
- [BF88] BRATLEY, PAUL and FOX, BENNETT L. “Algorithm 659: Implementing Sobol’s Quasirandom Sequence Generator”. *ACM Trans. Math. Softw.* 14.1 (Mar. 1988), 88–100. ISSN: 0098-3500. DOI: [10.1145/42288](https://doi.org/10.1145/42288). [214372](https://doi.org/10.1145/42288). URL: <https://doi.org/10.1145/42288>. [214372](https://doi.org/10.1145/42288) 2.
- [CPC84] COOK, ROBERT L., PORTER, THOMAS, and CARPENTER, LOREN. “Distributed Ray Tracing”. *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), 137–145. ISSN: 0097-8930. DOI: [10.1145/964965](https://doi.org/10.1145/964965). [808590](https://doi.org/10.1145/964965). URL: <https://doi.org/10.1145/964965>. [808590](https://doi.org/10.1145/964965) 1.
- [Fau82] FAURE, HENRI. “Discrépance de Suites Associées à un Système de Numération (en Dimension  $s$ )”. *fr. Acta Arithmetica* 41.4 (1982), 337–351. URL: <http://eudml.org/doc/205851> 2.
- [FK02] FRIEDEL, ILJA and KELLER, ALEXANDER. “Fast Generation of Randomized Low-Discrepancy Point Sets”. *Monte Carlo and Quasi-Monte Carlo Methods 2000*. Springer, 2002, 257–273 2.
- [GHSK08] GRÜNSCHLOSS, LEONHARD, HANIKA, JOHANNES, SCHWEDE, RONNIE, and KELLER, ALEXANDER. “(t, m, s)-Nets and Maximized Minimum Distance”. *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Ed. by KELLER, ALEXANDER, HEINRICH, STEFAN, and NIEDERREITER, HARALD. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, 397–412. ISBN: 978-3-540-74496-2 2, 6.
- [GRK12] GRÜNSCHLOSS, LEONHARD, RAAB, MATTHIAS, and KELLER, ALEXANDER. “Enumerating Quasi-Monte Carlo Point Sequences in Elementary Intervals”. *Monte Carlo and Quasi-Monte Carlo Methods 2010*. Ed. by PLASKOTA, LESZEK and WOŹNIAKOWSKI, HENRYK. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, 399–408. ISBN: 978-3-642-27440-4 2, 3.
- [Ham60] HAMMERSLEY, J. M. “Monte Carlo Methods for Solving Multivariable Problems”. *Annals of the New York Academy of Sciences* 86.3 (1960), 844–874. DOI: <https://doi.org/10.1111/j.1749-6632.1960.tb42846.x> 2, 6.
- [HCK21] HELMER, ANDREW, CHRISTENSEN, PER, and KENSLER, ANDREW. “Stochastic Generation of (t, s) Sample Sequences”. *Eurographics Symposium on Rendering - DL-only Track*. Ed. by BOUSSEAU, ADRIEN and MCGUIRE, MORGAN. The Eurographics Association, 2021. ISBN: 978-3-03868-157-1. DOI: [10.2312/sr.20211287](https://doi.org/10.2312/sr.20211287) 3, 4, 7.
- [HP11] HOFER, ROSWITHA and PIRSIC, GOTTLIEB. “An Explicit Construction of Finite-Row Digital (0, s)-Sequences”. *Unif. Distrib. Theory* 6.2 (2011), 13–30 4.
- [JK08] JOE, STEPHEN and KUO, FRANCES Y. “Notes on Generating Sobol Sequences”. *ACM Transactions on Mathematical Software (TOMS)* 29.1 (2008), 49–57 7.
- [Kel06] KELLER, ALEXANDER. “Myths of Computer Graphics”. *Monte Carlo and Quasi-Monte Carlo Methods 2004*. Springer, 2006, 217–243 2.
- [Kel13] KELLER, ALEXANDER. “Quasi-Monte Carlo Image Synthesis in a Nutshell”. *Monte Carlo and Quasi-Monte Carlo Methods 2012*. Ed. by DICK, JOSEF, KUO, FRANCES Y., PETERS, GARETH W., and SLOAN, IAN H. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, 213–249. ISBN: 978-3-642-41095-6 2.
- [KK02] KOLLIG, THOMAS and KELLER, ALEXANDER. “Efficient Multidimensional Sampling”. *Computer Graphics Forum*. Vol. 21. 3. 2002, 557–563 2.
- [LP03] LARCHER, G and PILLICHSHAMMER, F. “Sums of Distances to the Nearest Integer and the Discrepancy of Digital Nets”. *Acta Arithmetica* 106 (2003), 379–408 6, 7.
- [MN98] MATSUMOTO, MAKOTO and NISHIMURA, TAKUJI. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), 3–30. ISSN: 1049-3301. DOI: [10.1145/272991](https://doi.org/10.1145/272991). [272995](https://doi.org/10.1145/272991). URL: <https://doi.org/10.1145/272991>. [272995](https://doi.org/10.1145/272991) 7.
- [Nie87] NIEDERREITER, HARALD. “Point Sets and Sequences with Small Discrepancy”. *Monatshefte für Mathematik* 104.4 (1987), 273–337 2.



- [Nie92] NIEDERREITER, HARALD. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, 1992 **1**, **2**.
- [Owe95] OWEN, ART B. “Randomly Permuted (t,m,s)-Nets and (t, s)-Sequences”. *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*. Ed. by NIEDERREITER, HARALD and SHIUE, PETER JAU-SHYONG. New York, NY: Springer New York, 1995, 299–317. ISBN: 978-1-4612-2552-2 **2**.
- [PJH16] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128006455 **3**, **6**, **7**.
- [PJH23] PHARR, MATT, JAKOB, WENZEL, and HUMPHREYS, GREG. *Physically Based Rendering: From Theory to Implementation*. 4th. MIT Press, 2023. ISBN: 0262048027 **1–3**, **6**.
- [Sob67] SOBOL', IL'YA MEEROVICH. “On the Distribution of Points in a Cube and the Approximate Evaluation of Integrals”. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki* 7.4 (1967), 784–802 **1**, **2**.
- [Tez94] TEZUKA, SHU. “A Generalization of Faure Sequences and its Efficient Implementation”. *Technical Report, IBM Research, Tokyo Research Laboratory* (1994). DOI: [10.13140/RG.2.2.16748.16003](https://doi.org/10.13140/RG.2.2.16748.16003) **2**, **3**.