2020_07_13__采样_02序列采样.md

目录

02序列采样

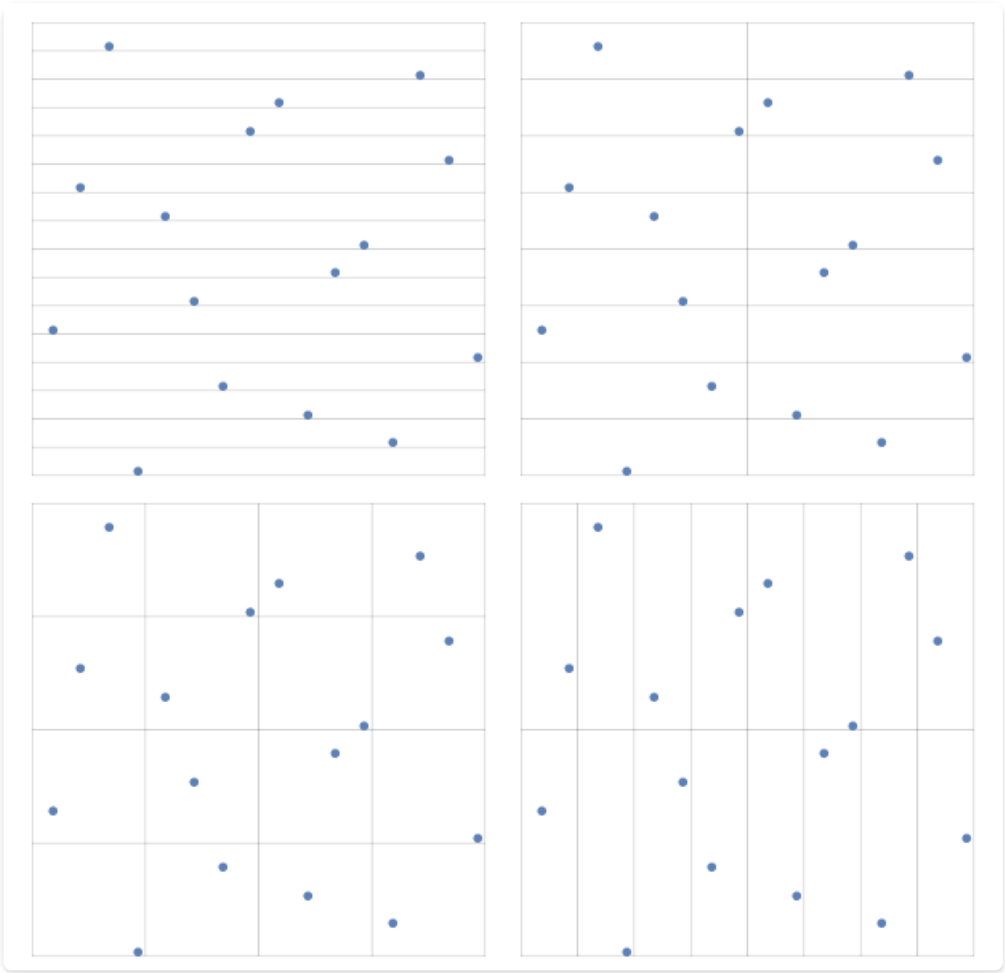
主要参考文章

文刀大神的文章

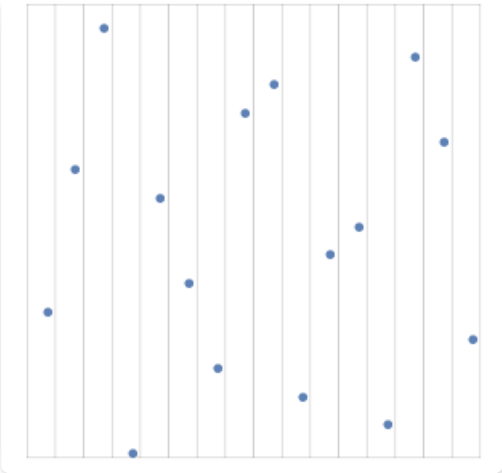
原理

0-2 采样器是基于 0-2 序列来生成的样本采样器

0-2 序列既能满足分层采样的限制, 也能满足拉丁超拉丁采样的要求, 对于一个 4*4 采样的情况, 甚至满足 (1/2,1/8) 的每一块出现恰好一个采样的情况



1



2

我们对其做 2 次方系数的分割, 都能在每一块都能恰好找到一个样本

0-2 序列怎么生成 2D 的图像采样?对于一个总共需要 2的整次幂 样本数图像(如果不够, 则填充), 我们可以把它拆分为 $2^{l_1+l_2}$, 其中 l_i 是非负的整数, 可以假设 l_1 是横坐标的幂次, l_2 是纵坐标的幂次, 他们组成 2维的元素点, 每个元素点和前一个元素点的距离在范围:

$$E = \left\{ \left[\frac{a_1}{2^{l_1}}, \frac{a_1+1}{2^{l_1}} \right) \times \left[\frac{a_2}{2^{l_2}}, \frac{a_2+1}{2^{l_2}} \right) \right\},$$

3

其中, a是

$$\text{where the integer } a_i = 0, 1, 2, 3, \dots, 2^{l_i} - 1.$$

4

这个式子, 对于任意的 $l_1 + l_2$ 组合都成立, 对于这个序列的子序列, 也成立。(比如 16个其中的前 4 个, 也是成立的)

个人理解

首先理解这个间隔, 这里的间隔是指, 对前一个数的间隔, 第一个数是和 0 做对比, 所以每个维度的第一个数都是在范围

$$\left[\frac{0}{2^{l_1+l_2}}, \frac{1}{2^{l_1+l_2}} \right)$$

- 为什么是 0, 1
- - 因为在这里, 第一个数, 所以必取 0, 1, 计算的是和起点 0,0 的距离

- 为什么是 $l_1 + l_2$
- ◦ 因为这里是对任意的 $l_1 + l_2$ 的组合都满足, 所以对于第一个数, 最严格的组合, 就是另一边的幂次是 0 的时候

那么, 理解了第一个点的范围后, 后续的点, 就根据这个范围限制来随机

后续的以此类推, 直到 $2^{l_i - 1}$ 个数, 恰好排满 2^{l_i} 的需求

这里进一步解释了, 如果面对, 单个样本, 也有多重数据的情况(2D 单个样本需求)应该怎么处理

这里假设, 一个 pixel 需要 (2*2) 个样本, (4*4)的 2D 单个样本需求, 我们应该怎么用 02序列 完成采样

- 首先, 我们使用的依旧是 02 序列, 但用的是前 $(2 * 2) * (4 * 4) = 2^6$ 个样本
- 对于前 $(4 * 4)$ 个样本, 一样满足 02序列, 那么这 16 个样本值, 就作为第一个采样样本的 2D 需求
- 依次类推

采样和生成矩阵

个人理解，这里主要解释了一个原因，为什么 02序列 能够在计算机上极速运行，根本原因是其 02 特性跟计算机的 2进制 十分吻合

02序列的生成

Halton 采样, 对于成千上万的像素, 非 2进制 的 Radical Inverse 计算开销还是太大了, 所以如果都使用 2 进制, 将提升很多。

在这里提到一个概念, 生成器矩阵(generator matrices), 在 Halton 采样中, 因为base 会变化, 所以对应的生成器矩阵, 会每一种 base 对应一个。

但是 02序列 采样, 我们打算都用 base 2 做基底, 所以可以通用一个矩阵。

对于 Radical Inverse 函数, $\Phi(x)$ 其实可以用矩阵来表示,

To see how generator matrices are used, consider an n-digit number a in base b, where the ith digit of a is $d_i(a)$ and where we have an $n \times n$ generator matrix $\mathbf{C}$. Then the corresponding sample point $x_a \in [0, 1)$ is defined by

$$x_a = [b^{-1}\, b^{-2} \cdots b^{-n}] \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & \ddots & & c_{2,n} \\ \vdots & & \ddots & \vdots \\ c_{n,1} & \cdots & \cdots & c_{n,n} \end{bmatrix} \begin{bmatrix} d_1(a) \\ d_2(a) \\ \vdots \\ d_n(a) \end{bmatrix},$$

这里做一下说明:

- 输入是 a , 它有 n 位
- 输出是 x_a, 它的 base 是 b, 它的取值范围一定在 $[0, 1)$
- 如果把中间的矩阵, 看成是单元矩阵, 那么这条式子, 就是 $a = \sum_{i=1}^{m} d_i(a) b^{i-1},$ 的矩阵拆除行列相乘的写法
- C 这个矩阵, 是一个已经存在的矩阵, 能够满足第一部分要求的矩阵, 直接拿来用即可

```
where all arithmetic is performed in the ring Zb (in other words, when all operations are performed modulo b)
```

在这里面, 所有的运算结果, 都会模一次 b, 这里主要指的是 C矩阵 和 d列 的运算部分, 因为整个逻辑中, 出现的应该都是, $[0, b-1]$ 的单个的数字

在这一部分, 我们使用的条件是 $b = 2$ and $n = 32.$, 并且利用各种 2进制 的特性, 来简化运算。

注意到, 因为是 2进制, 所以 C矩阵, 里面要么是 0, 要么是 1, 所以我们用一个 uint32_t 来表示它的列向量, 这种表示方法, 既省内存又高效

考虑到右边的矩阵乘法 $\mathbf{C}[d_i(a)]^T$, 我们将其展开, 有:

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & \ddots & & c_{2,n} \\ \vdots & & \ddots & \vdots \\ c_{n,1} & \cdots & \cdots & c_{n,n} \end{bmatrix} \begin{bmatrix} d_1(a) \\ d_2(a) \\ \vdots \\ d_n(a) \end{bmatrix} = d_1 \begin{bmatrix} c_{1,1} \\ c_{2,1} \\ \vdots \\ c_{n,1} \end{bmatrix} + \cdots + d_n \begin{bmatrix} c_{1,n} \\ c_{2,n} \\ \vdots \\ c_{n,n} \end{bmatrix}.$$

11

在这里, 数字 d_i 只有 1和0, 代表是否在总和上, 加上 $C[i]$, 而在这里面, 加法再模2的结果, 是和 **或** 运算是同一个结果

```
94
95  // 02序列的算法函数
96  // C矩阵的列向量，在参数 a 的每一位是否位 1 的情况下，判断是否做 相加再模2(这个操作，等同于 或)
97  inline uint32_t MultiplyGenerator(const uint32_t *C, uint32_t a) {
98      uint32_t v = 0;
99      for (int i = 0; a != 0; ++i, a >>= 1)
100         if (a & 1) v ^= C[i];
101     return v;
102 }
103
```

12

对于左边的运算, 也可以化简, 我们假设, 上一步计算出来的结果 $v = \mathbf{C}[d_i(a)]^T$, 那么下一步就是:

$$x_a = \begin{bmatrix} 2^{-1} & 2^{-2} & \cdots & 2^{-n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^{32} 2^{-i} v_i.$$

14

注意到, 我们的 v, 也是用 uint32_t 的格式进行保存的, 即, 这个 v 的值是:

$$v = v_1 + 2v_2 + \cdots = \sum_{i=1}^{32} 2^{i-1} v_i.$$

15

那对于这个数, 我们进行, 头尾调换操作, 也就是做 Radical Inverse:

首先变成:

$$v' = \sum_{i=1}^{32} 2^{32-i} v_i.$$

16

用哪个函数?上一节用的

```
65
66  // Low Discrepancy Inline Functions
67  inline uint32_t ReverseBits32(uint32_t n) {
68      n = (n << 16) | (n >> 16); // 因为 n 是32位的，所以 n << 16 是会自动拓展成 64 位，能够保留数据，64 位不能这么做的原因是，不能再往前拓展了
69      n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
70      n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
71      n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2); // 这里是把 abcd 改成了 cdab
72      n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1); // 这里要把 cdab 改成 dcba，所以用的是 5(0101) 和 a(1010)
73      return n;
74  }
75
```

17

然后, 再乘上 2^{-32} 即可

乱序

为了让 02序列 更具有实用性, 我们需要在对每个像素做采样时, 加入一定的随机性, 除了使用不同的 02 序列, 另一种方法就是在某一步, 我们加入随机的因子。

个人理解，在上一整步中，C矩阵是固定的，整个算法流程也是固定的，所以同样的输入，就有同样的输出，这显然对于采样的需求上，是不可接受的，因此，就需要加入随机的因子

这里提供的随机方法是, 不停的改动区域, 并尝试置换

对于每个维度

- 首先分为 $[0, 0.5)$ $[0.5, 1)$ 来确定他们是否进行交换
- 然后对分成的 2半, 做上一步相同的处理
- 直到处理完, 所有的 2进制位数为止

这种方法有一个好处:

- 保留了原本数据的 低差异性(不知道怎么得出的结论, pbrt说是就是了

下面是, 原来的 02序列, 和打乱了的视图

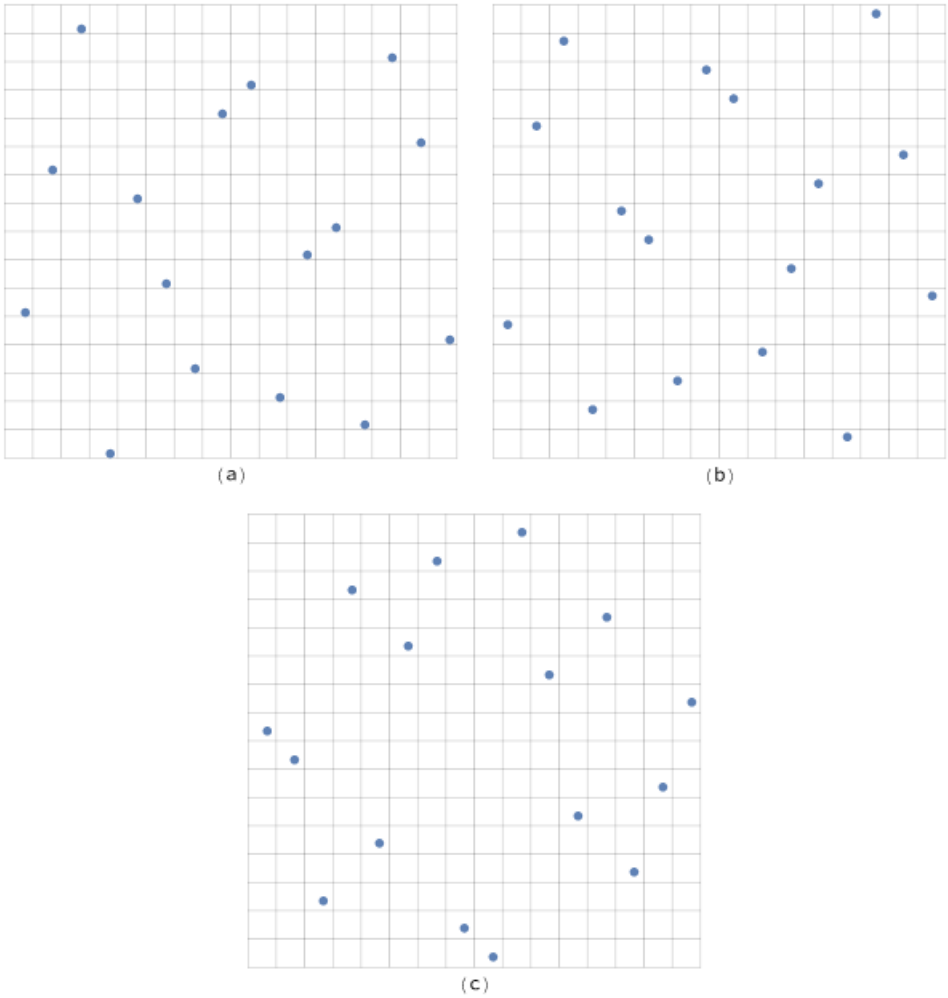


Figure 7.29: (a) A low-discrepancy $(0, 2)$-sequence-based sampling pattern and (b, c) two randomly scrambled instances of it. Random scrambling of low-discrepancy patterns is an effective way to eliminate the artifacts that would be present in images if we used the same sampling pattern in every pixel, while still preserving the low-discrepancy properties of the point set being used.

这个方法, 同样可以用异或来实现, 因为都是 01 表示的 uint32_t , 且异或中, 异或 0 等于本身

我们暂时可以理解成, 通过异或, 来表示是否交换(**TODO**, 第一次看, 是看不懂为什么可以, 但确实有这个流程)

在看完后面的内容，可以理解为，我们通过一个随机数，就直接代表了整个的随机过程，然后再这个随机数的基础上，做我们的 Radical Inverse 算法

在这里, 先提出一个概念, 格雷码

格雷码的特点:

- 对 2^i 都支持 一一映射
- 相邻的数字, 他们只有一位的变化
- 一位的变化是哪一位?比如下图中的 $g(n)$ 对比 $g(n-1)$, 变化的位数就是 n 本身, 后面尾巴有多少个 0(

100 到 000 不算

n (base 10)	n (binary)	$g(n)$	Changed Bit Index
0	000	000	n/a
1	001	001	0
2	010	011	1
3	011	010	0
4	100	110	2
5	101	111	0
6	110	101	1
7	111	100	0

19

```
117 // 一个数，获得它对应的 格雷码
118 // 这个坑爹的函数，全局引用就这里有个定义，就是拿给你看的
119 inline uint32_t GrayCode(uint32_t v) { return (v >> 1) ^ v; }
```

20

格雷码的获取方程

```
121 // 非常关键的一点，这个算法，不是 02序列，就是简单的 2进制 Radical Inverse,也叫做 VanDerCorput
122 // 在这里， 同样有 C 矩阵的概念，不过是被大大优化了的，优化的方式主要是 异或计算 和 格雷码 这两点
123 // C: 2进制 逆转矩阵
124 // n: 传入的计算数量，这里的 n 是被补齐到 2^i 次方的
125 // scramble: 随机数
126 // p: 存放结果的地方
127 inline void GrayCodeSample(const uint32_t *C, uint32_t n, uint32_t scramble,
128                            Float *p) {
129     uint32_t v = scramble; // 这是一个随机数，随机初值
130     for (uint32_t i = 0; i < n; ++i) {
131 #ifndef PBRT_HAVE_HEX_FP_CONSTANTS
132         p[i] = std::min(v * Float(2.3283064365386963e-10) /* 1/2^32 */,
133                         OneMinusEpsilon);
134 #else
135         // 第一个数，就直接存随机数
136         p[i] = std::min(v * Float(0x1p-32) /* 1/2^32 */,
137                         OneMinusEpsilon);
138 #endif
139         // 算尾巴有几个 0 ，从 1 开始，到 n 本身(也就是 2^xxx) 不从 0 开始的原因是，0 的 CountTrailingZeros 是搞不了的
140         // 1. 因为用的是格雷码的理念，所以当前这个数v(这里把它当做格雷码来用)，只在 CountTrailingZeros(i + 1) 位上，和前一个数不同
141         // 2. 因为我们用的是 2进制的 Radical Inverse 也就是 VanDerCorput，所以，要用 C 矩阵做一个映射，映射到对应相对称的位置
142         // 3. 因为不一样，所以做一次 异或
143         v ^= C[CountTrailingZeros(i + 1)];
144     }
145 }
```

21

上面是我对于整个函数的理解

吐槽一下, 在pbrt中, 作者还给出了对应的 指令代码, 非常自恋的夸赞代码的简洁(代价就是太难看懂了

The x86 assembly code for the heart of the inner loop (with the loop control logic elided) is wonderfully brief:

```
xorps      %xmm1, %xmm1
cvtsi2ssq  %rax, %xmm1
mulss      %xmm0, %xmm1
movss      %xmm1, (%rcx,%rdx,4)
incq       %rdx
bsfl       %edx, %eax
xorl       $31, %eax
xorl       (%rdi,%rax,4), %esi
```

22

非常关键的一点

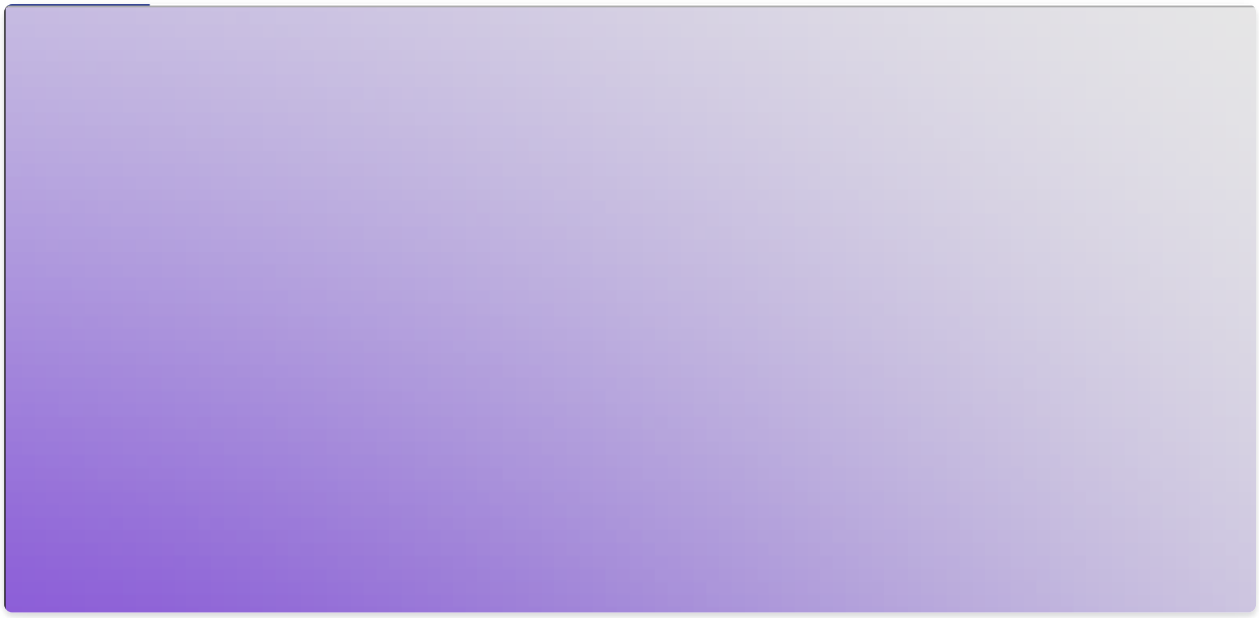
- 虽然前面一直在谈 02序列, 但这里举的例子, 是 VanDerCorput 来说明格雷码的好用之处!一开始看会非常非常懵逼

采样器的实现

02 采样器, 2D 的数据使用的是 02序列, 而 1D 的数据是 VanDerCorput

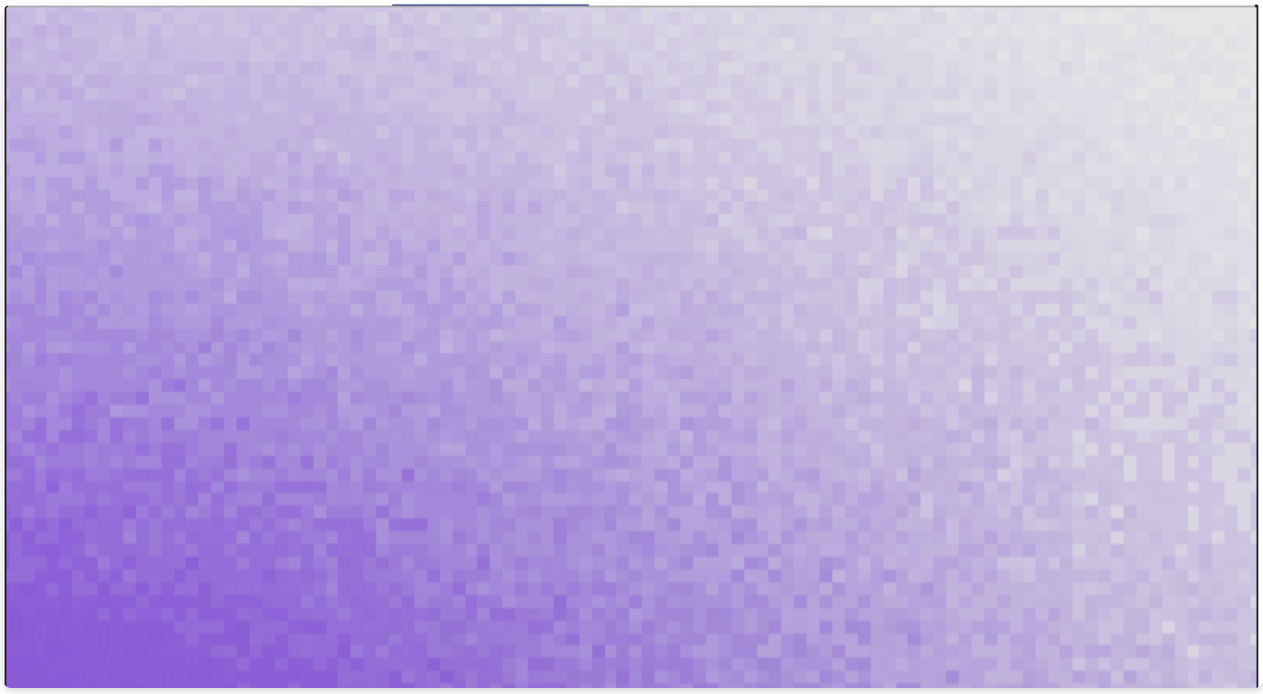
pbrt 中给出了 02序列采样 和 分层采样 的对比, 相比来说, 02序列采样, 放大后有横竖纹路(但是从效率, 和整体效果, 肯定 02 更优

如下图



23

24



25

在 02序列采样中, 初始化:

```
41
42  // ZeroTwoSequenceSampler Method Definitions
43  ZeroTwoSequenceSampler::ZeroTwoSequenceSampler(int64_t samplesPerPixel,
44                                                 int nSampledDimensions)
45      : PixelSampler(RoundUpPow2(samplesPerPixel), nSampledDimensions) {
46      if (!IsPowerOf2(samplesPerPixel))
47          Warning(
48              "Pixel samples being rounded up to power of 2 "
49              "(from %" PRId64 " to %" PRId64 ").",
50              samplesPerPixel, RoundUpPow2(samplesPerPixel));
51  }
52
```

26

这个函数, 把采样数, 补到了 2 的整数次幂, 为的就是方便我们进行后续的 02序列 运算(当然, 因为我们的 VanDerCorput 采用了格雷码映射, 所以也需要补齐

```
53  void ZeroTwoSequenceSampler::StartPixel(const Point2i &p) {
54      ProfilePhase _(Prof::StartPixel);
55      // 这里针对 1D 使用 VanDerCorput,  2D 使用 02 序列
56      // Generate 1D and 2D pixel sample components using $(0,2)$-sequence
57      for (size_t i = 0; i < samples1D.size(); ++i)
58          VanDerCorput(1, samplesPerPixel, &samples1D[i][0], rng);
59      for (size_t i = 0; i < samples2D.size(); ++i)
60          Sobol2D(1, samplesPerPixel, &samples2D[i][0], rng);
61
62      // Generate 1D and 2D array samples using $(0,2)$-sequence
63      for (size_t i = 0; i < samples1DArraySizes.size(); ++i)
64          VanDerCorput(samples1DArraySizes[i], samplesPerPixel,
65                       &sampleArray1D[i][0], rng);
66      for (size_t i = 0; i < samples2DArraySizes.size(); ++i)
67          Sobol2D(samples2DArraySizes[i], samplesPerPixel, &sampleArray2D[i][0],
68                  rng);
69      PixelSampler::StartPixel(p);
70  }
```

27

这个是在某个像素开始时, 数据的初始化(因为这个是 Pixel Sampler, 所以针对每一个像素, 都有需要的内容初始化, 参见 Pixel Sampler 那一章

我们先对 1D 的情况做采样, 这里加上了, 采样后的洗牌流程

```
162
163  inline void VanDerCorput(int nSamplesPerPixelSample, int nPixelSamples,
164                           Float *samples, RNG &rng) {
165      uint32_t scramble = rng.UniformUInt32();
166      // Define _CVanDerCorput_ Generator Matrix
167      const uint32_t CVanDerCorput[32] = {
168  #ifdef PBRT_HAVE_BINARY_CONSTANTS 活动预处理器块
204  #else 非活动预处理器块
211  #endif  // PBRT_HAVE_BINARY_CONSTANTS
212      };
213      int totalSamples = nSamplesPerPixelSample * nPixelSamples;
214      GrayCodeSample(CVanDerCorput, totalSamples, scramble, samples);
215      // Randomly shuffle 1D sample points
216      // 第一次洗牌，是针对 SPP，每一个样本的采样数据结构，不是数组就是 1，是数组就是 数组内乱序
217      for (int i = 0; i < nPixelSamples; ++i)
218          Shuffle(samples + i * nSamplesPerPixelSample, nSamplesPerPixelSample, 1,
219                  rng);
220      // 第二次洗牌，是针对 当前 Pixel 的所有样本，是数组就是 1，在 nPixelSamples 下乱序
221      // 是数组就保存 nSamplesPerPixelSample 也就是数组的长度的记录
222      Shuffle(samples, nPixelSamples, nSamplesPerPixelSample, rng);
223  }
224
```

28

如果硬是要理解 VanDerCorput 中的 C矩阵, 那就是单位矩阵, 不过 VanDerCorput 的流程中的矩阵, 其实是做了一个逆转的

```
167      const uint32_t CVanDerCorput[32] = {
168  #ifdef PBRT_HAVE_BINARY_CONSTANTS
169          // clang-format off
170          // 这里是一个逆转 0 对应 最高位的 1,1 对应 次高位的 1，以此类推
171          0b10000000000000000000000000000000,
172          0b01000000000000000000000000000000,
173          0b00100000000000000000000000000000,
174          0b00010000000000000000000000000000,
175          // Remainder of Van Der Corput generator matrix entries
176          0b00001000000000000000000000000000,
177          0b00000100000000000000000000000000,
178          0b00000010000000000000000000000000,
179          0b00000001000000000000000000000000,
180          0b00000000100000000000000000000000,
181          0b00000000010000000000000000000000,
182          0b00000000001000000000000000000000,
183          0b00000000000100000000000000000000,
184          0b00000000000010000000000000000000,
185          0b00000000000001000000000000000000,
186          0b00000000000000100000000000000000,
187          0b00000000000000010000000000000000
```

29

那么 02序列要怎么使用呢?没错, C 矩阵已经给好了, 我们只需要把 1D 的 VanDerCorput C矩阵, 换成 2D 的 C 矩阵即可

```
inline void Sobol2D(int nSamplesPerPixelSample, int nPixelSamples,
                    Point2f *samples, RNG &rng) {
    Point2i scramble;
    scramble[0] = rng.UniformUInt32();
    scramble[1] = rng.UniformUInt32();

    // Define 2D Sobol$'$ generator matrices _CSobol[2]_
    // 2D 的使用这里给出的矩阵（为什么是这个矩阵，不清楚 TODO????
    const uint32_t CSobol[2][32] = {
        {0x80000000, 0x40000000, 0x20000000, 0x10000000, 0x8000000, 0x4000000,
         0x2000000, 0x1000000, 0x800000, 0x400000, 0x200000, 0x100000, 0x80000,
         0x40000, 0x20000, 0x10000, 0x8000, 0x4000, 0x2000, 0x1000, 0x800,
         0x400, 0x200, 0x100, 0x80, 0x40, 0x20, 0x10, 0x8, 0x4, 0x2, 0x1},
        {0x80000000, 0xc0000000, 0xa0000000, 0xf0000000, 0x88000000, 0xcc000000,
         0xaa000000, 0xff000000, 0x80800000, 0xc0c00000, 0xa0a00000, 0xf0f00000,
         0x88880000, 0xcccc0000, 0xaaaa0000, 0xffff0000, 0x80008000, 0xc000c000,
         0xa000a000, 0xf000f000, 0x88008800, 0xcc00cc00, 0xaa00aa00, 0xff00ff00,
         0x80808080, 0xc0c0c0c0, 0xa0a0a0a0, 0xf0f0f0f0, 0x88888888, 0xcccccccc,
         0xaaaaaaaa, 0xffffffff}};
    GrayCodeSample(CSobol[0], CSobol[1], nSamplesPerPixelSample * nPixelSamples,
                   scramble, samples);
    for (int i = 0; i < nPixelSamples; ++i)
        Shuffle(samples + i * nSamplesPerPixelSample, nSamplesPerPixelSample, 1,
                rng);
    Shuffle(samples, nPixelSamples, nSamplesPerPixelSample, rng);
}
```

30

额外谈的一点

在pbrt中, 他额外提到了一点

这里做洗牌的原因, 是因为 1D 的数据, 会和 2D 的数据显示出相关性, 这个很好理解

因为都是同一个算法出来的, 虽然用的随机值不同, 但整个流程是一样的

但是提了一点, 因为相关性

for each one, some correlation can still remain between elements of these patterns, such that the ith element of the 1D pattern and the ith element of the 2D pattern are related. As such, in the earlier area lighting example, the distribution of sample points on each light source would not in general cover the entire light due to this correlation, leading to unusual rendering errors.

31

面光源无法覆盖在面光源上采样的点, 这一点比较迷惑, 但是这里有相关性, 做随机洗牌确实是有必要的

TODO 这个相关性, 导致面光源覆盖性不高的点