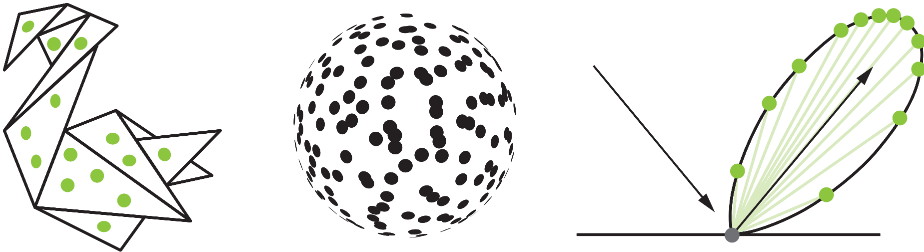
CHAPTER 16

Sampling Transformations Zoo

Peter Shirley, Samuli Laine, David Hart, Matt Pharr, Petrik Clarberg,
Eric Haines, Matthias Raab, and David Cline
NVIDIA



ABSTRACT

We present several formulas and methods for generating samples distributed according to a desired probability density function on a specific domain. Sampling is a fundamental operation in modern rendering, both at runtime and in preprocessing. It is becoming ever more prevalent with the introduction of ray tracing in standard APIs, as many ray tracing algorithms are based on sampling by nature. This chapter provides a concise list of some useful tricks and methods.

16.1 THE MECHANICS OF SAMPLING

A common task in ray tracing programs is to choose a set of samples on some domain with an underlying probability density function (PDF): for example, a set of points on the unit hemisphere whose probability density is proportional to the cosine of the polar angle. This is often accomplished by taking a set of samples that are uniform on the unit hypercube and transforming them to the desired domain. For readers unfamiliar with this general sample-generation pipeline, please refer to Chapter 13 of Pharr et al. [8].

This chapter catalogs a variety of methods to generate specific distributions that the authors have found useful in ray tracing programs. These are all either previously published or are part of the "conventional wisdom."

© NVIDIA 2019

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_16

16.2 INTRODUCTION TO DISTRIBUTIONS

In one dimension, there is a fairly standard way to create a transform that will generate samples with the desired PDF p. The key observation behind this method uses a construct called the *cumulative distribution function* (CDF), usually denoted with a capital $P(x)$:

$$P(x) = \text{probability that a uniformly distributed sample } u < x = \int_{-\infty}^{x} p(y)\,dy. \tag{1}$$

To see how this function can become useful, suppose that we want to determine where a particular uniformly distributed value $u = 0.5$ will go when passed through our desired warping function $g : x = g(0.5)$. If we assume that g is nondecreasing (so its derivative is never negative), then half of the points will map to values of x less than $g(0.5)$ and the other half to values of x greater than $g(0.5)$. Because of the intrinsic property of CDFs, when $P(x) = 0.5$ we also know that half of the area under the PDF is to the left of that x, so we can deduce that

$$P(g(0.5)) = 0.5. \tag{2}$$

This basic observation actually works for any x in addition to $x = 0.5$. Thus, we have

$$g(x) = P^{-1}(x), \tag{3}$$

where P^{-1} is the inverse function of P. The notation of inverse functions can be confusing. In practice what it means algebraically is that given a PDF p we integrate it using the integral in Equation 1, and then we solve for x in the resulting equation (which is inverting P):

$$u = P(x).$$

Given a sequence of uniformly distributed samples u, we compute the inverse of P to find a p-distributed sequence of samples x.

For two-dimensional domains, two uniformly distributed samples are needed, $u[0]$ and $u[1]$. These together give a point on the two-dimensional unit square: $(u[0], u[1]) \in [0, 1)^2$. They can also be transformed to a desired domain.

For example, to pick a uniformly distributed sample on a unit disk, we would write down an integral in polar coordinates with a measure $dA = r dr d\varphi$, where r represents a distance (radius) from the origin along the angle φ from the positive x-axis. When possible in 2D domains, the two dimensions are separated into two independent 1D PDFs, and terms such as the r in the measure need to be handled carefully. Although a uniform $p(r, \varphi) = \frac{1}{\pi}$ for uniform density on the unit disk, when separated into two 1D independent densities, the r is attached to the density of the radius. The resulting two 1D PDFs are

$$p_1(\varphi) = \frac{1}{2\pi}, \quad p_2(r) = 2r. \tag{4}$$

The constant terms $\frac{1}{2\pi}$ and 2 make each of the PDFs integrate to 1 (a required property of PDFs as discussed earlier). If we find the CDFs for those two PDFs, we get

$$P_1(\varphi) = \frac{\varphi}{2\pi}, \quad P_2(r) = r^2. \tag{5}$$

If we want to transform uniform samples $u[0]$ and $u[1]$ to respect those CDFs, we can apply Equation 2 to each 1D CDF:

$$(u[0], u[1]) = \left(\frac{\varphi}{2\pi}, r^2\right), \tag{6}$$

and then solve each for φ and r, yielding

$$(\varphi, r) = \left(2\pi u[0], \sqrt{u[1]}\right). \tag{7}$$

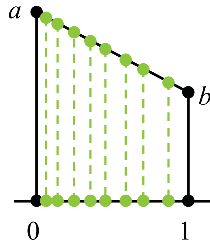
This basic "playbook" is used for most of the transforms found in the literature.

An important note is that, while our treatment assumes that $(u[0], u[1])$ are uniform on a unit square, in a higher-dimension d the points are uniformly distributed in the unit hypercube $[0, 1)^d$. Such samples can be generated by (pseudo-)random or quasi-random methods [6].

The rest of this chapter gives several transforms, usually without derivation and most of them in two dimensions, that we have found to be useful in ray tracing programs.

16.3 ONE-DIMENSIONAL DISTRIBUTIONS

16.3.1 LINEAR



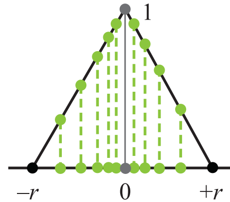
Given the *linear function* over [0, 1] with $f(0) = a$ and $f(1) = b$ and given a uniformly distributed sample u, the following generates a value $x \in [0, 1]$ distributed according to f:

```
1 float SampleLinear( float a, float b ) {
2     if (a == b) return u;
3     return clamp((a - sqrt(lerp(u, a * a, b * b))) / (a - b), 0, 1);
4 }
```

The value of the PDF of a sample x can be found by the following:

```
1 if (x < 0 || x > 1) return 0;
2 return lerp(x, a, b) / ((a + b) / 2);
```

16.3.2 TENT



A non-normalized *tent function* is specified by a width r and is defined by a pair of linear functions: it goes linearly from 0 at $-r$ to a value of 1 at the origin, and then back down to 0 at r. The `SampleLinear()` function in the following code implements the technique described in Section 16.3.1:

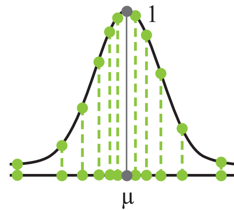
```
1 if (u < 0.5) {
2     u /= 0.5;
3     return -r * SampleLinear(u, 1, 0);
4 } else {
5     u = (u - .5) / .5;
6     return r * SampleLinear(u, 1, 0);
7 }
```

Note that we use the uniformly distributed sample u to choose one half of the tent function and then remap the sample back to [0, 1) to sample the appropriate linear function.

The PDF for a value sampled at x can be computed as follows:

```
1 if (abs(x) >= r) return 0;
2 return 1 / r - abs(x) / (r * r);
```

16.3.3 NORMAL DISTRIBUTION



The *normal distribution* is defined as

$$f(x) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{8}$$

It has infinite support but falls off quickly once $\|x - \mu\|$ is a few multiples of σ. It is not possible to analytically generate a single sample from this distribution, since doing so requires inverting the error function,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2}\, dx, \tag{9}$$

which is not feasible in closed form. One option is to use a polynomial approximation of the inverse, which we take to be implemented by `ErfInv()`. Given that, a sample can be generated as

```
1 return mu + sqrt(2) * sigma * ErfInv(2 * u - 1);
```

The PDF for a sample x is then given by

```
1 return 1 / sqrt(2 * M_PI * sigma * sigma) *
2           exp(-(x - mu) * (x - mu) / (2 * sigma * sigma));
```

If more than one sample is needed, the *Box-Müller transform* generates two samples from the normal distribution, given two uniformly distributed samples:

```
1 return { mu + sigma * sqrt(-2 * log(1-u[0])) * cos(2*M_PI*u[1]),
2           mu + sigma * sqrt(-2 * log(1-u[0])) * sin(2*M_PI*u[1])) };
```

16.3.4 SAMPLING FROM A ONE-DIMENSIONAL DISCRETE DISTRIBUTION

Given an array of floating-point values, there are a few ways to choose one of them with a probability proportional to its relative magnitude. We present two methods here: one is better if only a single sample is needed, and the other is better if multiple samples are needed.

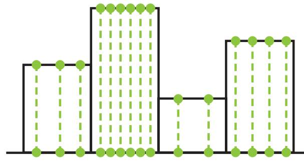
16.3.4.1 JUST ONCE

If only a single sample is needed, then the function in the following code can be used. It computes the sum of the values (expecting that all are nonnegative) and then scales the provided uniformly distributed sample u, remapping it from the $[0, 1)$ domain to $[0; \texttt{sum})$. It then walks through the array, subtracting each array element's value from the remapped sample. Once it gets to the point that subtracting the next value would make the scaled value negative, it has found the right place to stop.

This function also returns the PDF for choosing an element as well as a remapped sample value in $[0, 1)$ based on the original sample value. Intuitively, there is still a uniform distribution left in the sample, because we used it to make only a discrete sampling decision. However, the number of uniformly distributed bits left may be too small for the sample to be reused, especially if the selected event has a tiny probability.

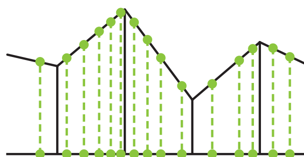
```
 1 int SampleDiscrete(std::vector<float> weights, float u,
 2             float *pdf, float *uRemapped) {
 3     float sum = std::accumulate(weights.begin(), weights.end(), 0.f);
 4     float uScaled = u * sum;
 5     int offset = 0;
 6     while (uScaled > weights[offset] && offset < weights.size()) {
 7         uScaled -= weights[offset];
 8         ++offset;
 9     }
10     if (offset == weights.size()) offset = weights.size() - 1;
11
12     *pdf = weights[offset] / sum;
13     *uRemapped = uScaled / weights[offset];
14     return offset;
15 }
```

16.3.4.2 MULTIPLE TIMES



If an array needs to be sampled more than once, it is much more efficient to precompute the array's CDF and perform a binary search for each sample. Care must be taken to distinguish between piecewise constant and piecewise linear data, as the CDF computation and sampling are different for each. For example, to sample from a piecewise constant distribution, we would use the following:

```
 1 vector<float> makePiecewiseConstantCDF(vector<float> pdf) {
 2     float total = 0.0;
 3     // CDF is one greater than PDF.
 4     vector<float> cdf { 0.0 };
 5     // Compute the cumulative sum.
 6     for (auto value : pdf) cdf.push_back(total += value);
 7     // Normalize.
 8     for (auto& value : cdf) value /= total;
 9     return cdf;
10 }
11
12 int samplePiecewiseConstantArray(float u, vector<float> cdf,
13             float *uRemapped)
14 {
15     // Use our (sorted) CDF to find the data point to the
16     // left of our sample u.
17     int offset = upper_bound(cdf.begin(), cdf.end(), u) -
18     cdf.begin() - 1;
19     *uRemapped = (u - cdf[offset]) / (cdf[offset+1] - cdf[offset]);
20     return offset;
21 }
```

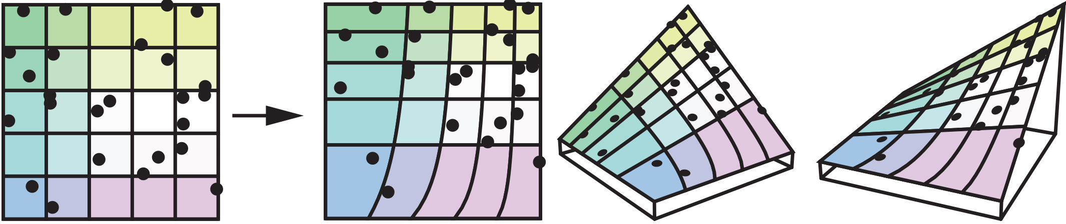


For sampling a piecewise linear distribution, the CDF can be constructed by computing the area of the trapezoid between each pair of samples. Sampling the distribution involves sampling from the linear segment using the `SampleLinear()`

function from Section 16.3.1, after the binary search. If using C++, the Standard Template Library's `random` module introduced `piecewise_constant_distribution` and `piecewise_linear_distribution` in C++11.

16.4 TWO-DIMENSIONAL DISTRIBUTIONS

16.4.1 BILINEAR



It can be useful to sample from the *bilinear interpolation function*, which we define as taking four values $v[4]$ that define a function over $[0, 1]^2$ by

$$f(x,y) = ((1-x)(1-y))v[0] + x(1-y)v[1] + (1-x)yv[2] + xyv[3]. \quad (10)$$

Then, given two uniformly distributed samples $u[0]$ and $u[1]$, a sample can be taken from the distribution $f(x,y)$ by first sampling one dimension and then sampling the second. Here, we use the one-dimensional linear sampling function, `SampleLinear()`, defined in Section 16.3.1:

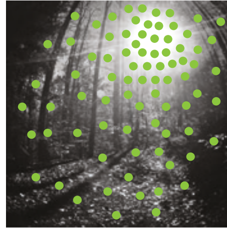
```
1 // First, sample in the v dimension. Compute the endpoints of
2 // the line that is the average of the two lines at the edges
3 // at u = 0 and u = 1.
4 float v0 = v[0] + v[1], v1 = v[2] + v[3];
5 // Sample along that line.
6 p[1] = SampleLinear(u[1], v0, v1);
7 // Now, sample in the u direction from the two line endpoints
8 // at the sampled v position.
9 p[0] = SampleLinear(u[0],
10                     lerp(p[1], v[0], v[2]),
11                     lerp(p[1], v[1], v[3]));
12 return p;
```

The PDF of a sampled value p is the following:

```
1 return (4 / (v[0] + v[1] + v[2] + v[3])) * Bilerp(p, v);
```

16.4.2 A DISTRIBUTION GIVEN A TWO-DIMENSIONAL TEXTURE

16.4.2.1 REJECTION SAMPLING



To choose a texel in a texture with probability proportional to the texel's brightness, one simple technique is to use *rejection sampling*, where texels are uniformly chosen and a sample is accepted only if the texel's brightness is greater than another uniformly distributed value:

```
1 do {
2     X = u();
3     Y = u();
4 } while (u() > brightness(texture(X,Y))); // Brightness is [0,1].
```

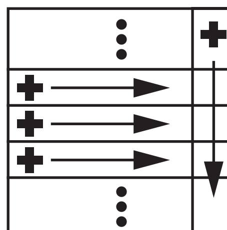
Note that the efficiency of rejection sampling a texture is proportional to the texture's average brightness, so if performance is a concern, avoid this method for sparse (mostly dark) textures.

16.4.2.2 MULTI-DIMENSIONAL INVERSION METHOD

To sample a texture in two dimensions, we can build on Section 16.3.4.2 (sampling from a one-dimensional array) by sampling from two distributions, vertical and horizontal:

> Build CDF tables (cumulative distribution) of brightness, one for each row of pixels, and normalize.

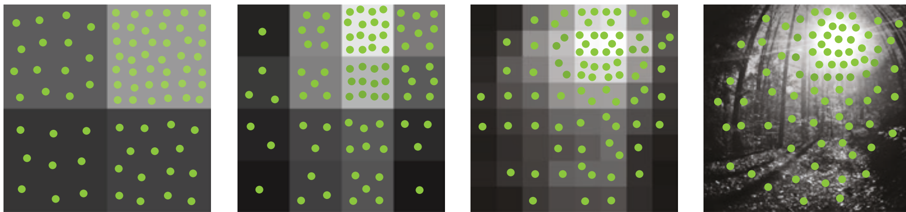
> Build a CDF for the last column (the sum of brightness across each row) and normalize.



- To sample from the texture's distribution, take a uniform two-dimensional sample $(u[0], u[1])$. Use $u[1]$ to binary-search the column CDF. This determines which row to use. Now, use $u[0]$ to binary-search the row to find the sample's column. The resulting coordinates (column, row) are distributed according to the texture.

The drawback of this *inversion method* is that it does not preserve stratification properties of the sample points (e.g., blue noise or low-discrepancy points) well. If this is an issue, it is preferable to sample hierarchically in two dimensions, as described next.

16.4.2.3 HIERARCHICAL TRANSFORMATION



Hierarchical warping is a way to improve on the shortcomings of the inverse transform sampling described in the previous section, namely that the row- and column-based inverse transform mapping may cause samples to be clustered. We note in advance that hierarchical warping does not completely solve the problems of continuity and stratification, especially when using correlated samples, e.g., blue noise or low-discrepancy sequences, but it is a practical way to have some spatial coherence while sampling a texture. Example applications of hierarchical warping include importance sampling methods for complex light sources [3, 7].

The principle is to build a tree of conditional probabilities, where at each node we store the relative importance of the node's children. Sampling is performed by starting from the root and at each node probabilistically deciding which child node to select based on a uniformly distributed sample. Rather than drawing a new uniform sample at each level, the algorithm both gets more efficient and generates better distributions if the uniform sample is remapped at each step. See illustrations of generated sampling probabilities in the article by Clarberg et al. [2].

This method is not limited to sampling discrete distributions in two dimensions. For example, the tree can be a binary tree, quad tree, or octree, depending on the domain. The following pseudocode illustrates the method for a binary tree:

```
1 node = root;
2 while (!node.isLeaf) {
3     if (u < node.probLeft) {
4         u /= node.probLeft;
5         node = node.left;
6     } else {
7         u /= (1.0 - node.probLeft);
8         node = node.right;
9 }
10 // Ok. We have found a leaf with the correct probability!
```

For two-dimensional textures, the implementation becomes particularly simple, as we can sample based on the texture's mipmap hierarchy directly. Starting at the 2 × 2 texel mipmap, the conditional probabilities are computed based on the texel values, first horizontally and then vertically. The best final distribution is achieved with a two-dimensional uniformly distributed sample:

```
1 int2 SampleMipMap(Texture& T, float u[2], float *pdf)
2 {
3     // Iterate over mipmaps of size 2x2 ... NxN.
4     // load(x,y,mip) loads a texel (mip 0 is the largest power of two)
5     int x = 0, y = 0;
6     for (int mip = T.maxMip()-1; mip >= 0; --mip) {
7         x <<= 1; y <<= 1;
8         float left = T.load(x, y, mip) + T.load(x, y+1, mip);
9         float right = T.load(x+1, y, mip) + T.load(x+1, y+1, mip);
10        float probLeft = left / (left + right);
11        if (u[0] < probLeft) {
12            u[0] /= probLeft;
13            float probLower = T.load(x, y, mip) / left;
14            if (u[1] < probLower) {
15                u[1] /= probLower;
16            }
17            else {
18                y++;
19                u[1] = (u[1] - probLower) / (1.0f - probLower);
20            }
21        }
22        else {
23            x++;
24            u[0] = (u[0] - probLeft) / (1.0f - probLeft);
25            float probLower = T.load(x, y, mip) / right;
```

```
26              if (u[1] < probLower) {
27                  u[1] /= probLower;
28              }
29              else {
30                  y++;
31                  u[1] = (u[1] - probLower) / (1.0f - probLower);
32              }
33          }
34      }
35      // We have found a texel (x,y) with probability proportional to
36      // its normalized value. Compute the PDF and return the
37      // coordinates.
38      *pdf = T.load(x, y, 0) / T.load(0, 0, T.maxMip());
39      return int2(x, y);
40 }
```

It should be noted that some numerical precision can be lost for all these methods that remap one or more uniformly distributed sample along the way. The input values are generally in 32-bit floating-point format, which means that once we get a leaf to sample, there may be only a few bits of precision left. This is not usually a problem in practice for common texture sizes, but it is important to know about. For higher precision, we always have the option of drawing new uniformly distributed samples at each step, but then stratification properties may be lost.

Another useful tip is that it is not necessary for the probabilities at each level in the tree to be the sums of the underlying nodes. If this is not the case, we can simply compute the sampling probability density function along the way by multiplicatively accumulating the selecting probabilities at each step. This leads to algorithms that allow sampling of functions where the full probability density function is not known beforehand but is created on the fly.

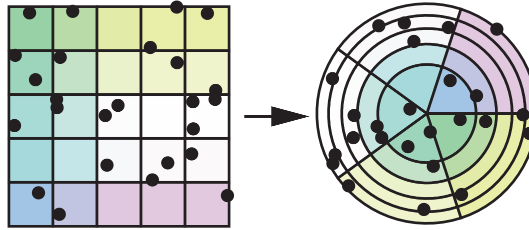
16.5 UNIFORMLY SAMPLING SURFACES

When sampling a two-dimensional surface uniformly, i.e., every point on the surface is equally likely to be sampled, the PDF of all points equals one over the area of the surface. For example, for a unit sphere $p = \frac{1}{4\pi}$.

16.5.1 DISK

A disk is centered at the origin $(x, y) = (0, 0)$ and has radius r.

16.5.1.1 POLAR MAPPING

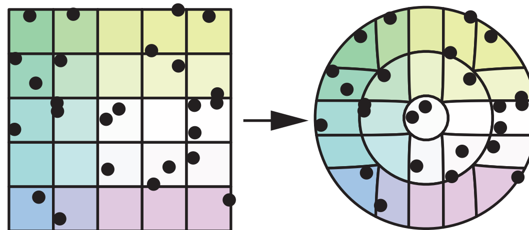


A *polar mapping* transforms the uniform $u[0]$ to favor larger radii, which in turn ensures a uniform distribution of samples. The area of the disk increases as the radius increases, with only a fourth of the total being within the half-radius.

```
1 r = R * sqrt(u[0]);
2 phi = 2*M_PI*u[1];
3 x = r*cos(phi);
4 y = r*sin(phi);
```

This polar mapping is usually not used because of the "seam" (discontinuity in the inverse transform) and the concentric mapping discussed next is preferred unless branching is being avoided.

16.5.1.2 CONCENTRIC MAPPING



A *concentric mapping* maps concentric squares with $[0, 1)^2$ to concentric circles so that there is no seam and adjacency is preserved [11].

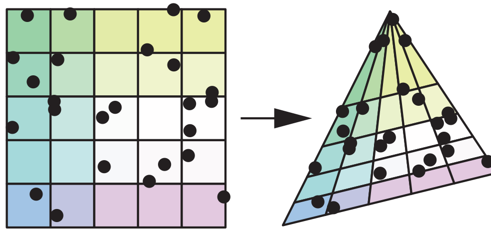
```
1 a = 2*u[0] - 1;
2 b = 2*u[1] - 1;
3 if (a*a > b*b) {
4     r = R*a;
5     phi = (M_PI/4)*(b/a);
6 } else {
7     r = R*b;
8     phi = (M_PI/2) - (M_PI/4)*(a/b);
9 }
```

```
10 X = r*cos(phi);
11 Y = r*sin(phi);
```

16.5.2 TRIANGLE

To uniformly sample a triangle with vertices P_0, P_1, and P_2, barycentric coordinates are used to transform the coordinates to be in range, or to flip the seed point if it is not in the lower half of the square.

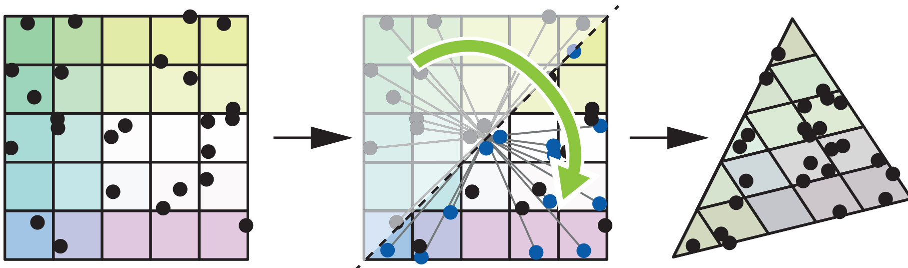
16.5.2.1 WARPING



We can sample directly in the valid barycentric range to warp a quadrilateral into a triangle:

```
1 beta = 1-sqrt(u[0]);
2 gamma = (1-beta)*u[1];
3 alpha = 1-beta-gamma;
4 P = alpha*P0 + beta*P1 + gamma*P2;
```

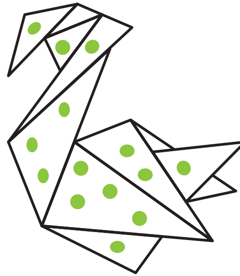
16.5.2.2 FLIPPING



To avoid the square root, you can also sample from a quadrilateral and flip the sample if you are on wrong side of the diagonal. However, flipping over the diagonal can reduce the effectiveness of blue noise or low-discrepancy sampling within the triangle, as there is usually no guarantee that well-distributed points in two dimensions remain well distributed when folded.

```
1 alpha = u[0];
2 beta = u[1];
3 if (alpha + beta > 1) {
4     alpha = 1-alpha;
5     beta = 1-beta;
6 }
7 gamma = 1-beta-alpha;
8 P = alpha*P0 + beta*P1 + gamma*P2;
```

16.5.3 TRIANGLE MESH



To sample points on a triangle mesh, Turk [13] suggests using binary search on the one-dimensional discrete distribution of triangle areas.

We can improve mesh sampling and create a mapping from samples in the unit square to points on the mesh by combining texture sampling from Section 16.4.2.2, triangle sampling from Section 16.5.2, and the remapped uniformly distributed samples from our array sampling function in Section 16.3.4.2. The steps are:

- Store the area of each triangle in a square-ish two-dimensional table. Order does not matter. Use 0 as the area for cells not associated with any triangles.
- Build a CDF of area for each row in the table and normalize.
- Build a CDF for the last column (the sum of area across each row) and normalize.

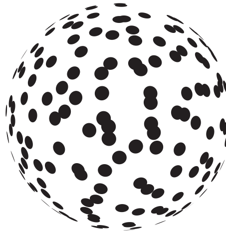
To sample the mesh:

- Take a uniformly distributed two-dimensional sample $(u[0], u[1])$.
- Use $u[1]$ to binary-search the column CDF. This determines which row r to use.
- Use $u[0]$ to binary search the row to find the sample's column c.
- Save the remapped samples from $(u[0], u[1])$ as $(v[0], v[1])$.

- Using our remapped two-dimensional variable $(v[0], v[1])$, sample the triangle corresponding to row r and column c, using the triangle sampling method from Section 16.5.2.
- The resulting three-dimensional coordinates are uniformly distributed on the triangle mesh.

Note that this method is discontinuous, which may affect the quality of the samples after transformation.

16.5.4 SPHERE



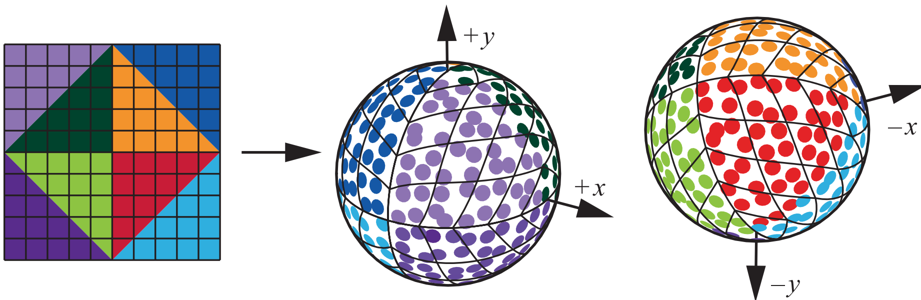
The sphere is centered at the origin and has radius r.

16.5.4.1 LATITUDE-LONGITUDE MAPPING

The following code shows how points can be generated using a uniform *latitude-longitude mapping*. Note the z value is uniformly distributed on $[-1, 1]$.

```
1 a = 1 - 2*u[0];
2 b = sqrt(1 - a*a);
3 phi = 2*M_PI*u[1];
4 x = R*b*cos(phi);
5 y = R*b*sin(phi);
6 z = R*a;
```

16.5.4.2 OCTAHEDRAL CONCENTRIC (UNIFORM) MAP



The previous method (the latitude-longitude map) is intuitive, but a drawback is that it "stretches" the sampling domain quite significantly at the top and bottom. Building on the concentric map in Section 16.5.1.2 and combining it with an octahedral map (cf., Figure 2 in Praun and Hoppe [9]), it is possible to define an *octahedral concentric mapping* of the sphere with good properties; its stretch is at worst a factor of 2 : 1 [1]. With a uniform two-dimensional point as input, the optimized transform to the unit sphere is as follows:

```
 1 // Compute radius r (branchless).
 2 u = 2*u - 1;
 3 d = 1 - (abs(u[0]) + abs(u[1]));
 4 r = 1 - abs(d);
 5
 6 // Compute phi in the first quadrant (branchless, except for the
 7 // division-by-zero test), using sign(u) to map the result to the
 8 // correct quadrant below.
 9 phi = (r == 0) ? 0 : (M_PI/4) * ((abs(u[1]) - abs(u[0])) / r + 1);
10 f = r * sqrt(2 - r*r);
11 x = f * sign(u[0]) * cos(phi);
12 y = f * sign(u[1]) * sin(phi);
13 z = sign(d) * (1 - r*r);
14 pdf = 1 / (4*M_PI);
```

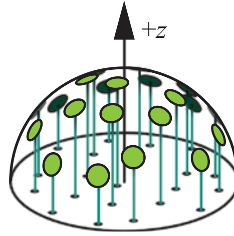
Note that in many applications these transforms from the unit square to the unit sphere are useful not only for generating samples, but also for representing spherical functions in a convenient square two-dimensional domain. The inverse operation, to map points on the unit sphere (e.g., ray directions) back to two dimensions, is equally useful.

16.6 SAMPLING DIRECTIONS

Sampling PDFs defined over directions on the sphere or hemisphere is a central part of many ray tracers. Often this sampling is for integrating incoming light to compute an outgoing intensity at a point. These PDFs are commonly defined in spherical coordinates where the polar angle (sometimes called the *zenith angle*) is usually denoted θ and the azimuthal angle is denoted φ. Unfortunately, different fields vary in whether they use this or the opposite notation convention. So, this notation may be the reverse of what the reader is used to, depending on their background, but it is relatively standard in computer graphics.

When choosing a direction, a common convention is to choose a point on the unit sphere (or hemisphere) and define the direction as the unit vector from the sphere center to that point.

16.6.1 COSINE-WEIGHTED HEMISPHERE ORIENTED TO THE *Z*-AXIS



A common way to generate diffuse rays in rendering methods for matte surfaces is to sample uniformly from a disk (as in Section 16.5.1) and then project the sample point up to the hemisphere. Doing so produces samples with a *cosine-weighted* distribution, where the density is high at the apex of the hemisphere and falls off toward the base. Generated samples will need to be transformed into the local tangent space of the surface being rendered.

```
1 x = sqrt(u[0])*cos(2*M_PI*u[1]);
2 y = sqrt(u[0])*sin(2*M_PI*u[1]);
3 z = sqrt(1-u[0]);
4 pdf = z / M_PI;
```

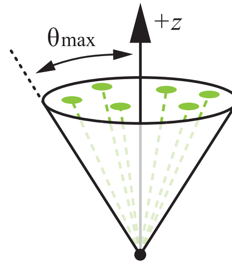
16.6.2 COSINE-WEIGHTED HEMISPHERE ORIENTED TO A VECTOR

As an alternative to transforming the *z*-axis to n (e.g., the normal of the tangent space), we can use a uniformly distributed sample on a tangent sphere. This method avoids constructing tangent vectors, but it comes at the expense of numerical precision for the grazing case. We can pick a uniformly distributed direction through a sphere by connecting two uniformly distributed samples on the surface of a sphere [10]. Doing so implies that the directions to the second point have a cosine density relative to the first point. If the vector n = (n_x, n_y, n_z) is a unit-length vector, this implies the following:

```
1 a = 1 - 2*u[0];
2 b = sqrt(1 - a*a);
3 phi = 2*M_PI*u[1];
4 x = n_x + b*cos(phi);
5 y = n_y + b*sin(phi);
6 z = n_z + a;
7 pdf = a / M_PI;
```

Note that (x, y, z) is not a unit vector. The precision problem arises when the uniformly distributed sample on the tangent sphere is nearly opposite to **n**, resulting in an output vector that is close to zero. Such points correspond to grazing rays (perpendicular to the normal). To avoid these cases, we can shrink the tangent sphere a bit by multiplying both a and b by a number slightly less than one.

16.6.3 DIRECTIONS IN A CONE

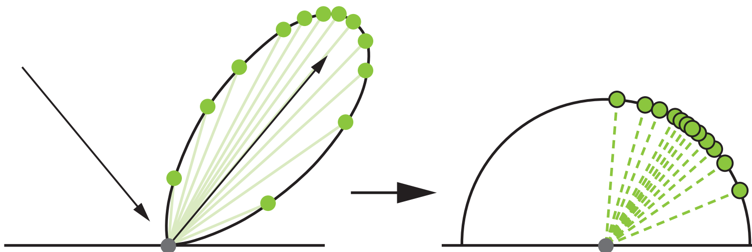


Given a cone with axis along the $+z$-axis and a spread angle θ_{max}, uniform directions in the cone can be sampled as follows:

```
1 float cosTheta = (1 - u[0]) + u[0] * cosThetaMax;
2 float sinTheta = sqrt(1 - cosTheta * cosTheta);
3 float phi = u[1] * 2 * M_PI;
4 x = cos(phi) * sinTheta
5 y = sin(phi) * sinTheta
6 z = cosTheta
```

The PDF of all samples is $1/(2\pi(1 - \cos\theta_{max}))$.

16.6.4 PHONG DISTRIBUTION



Given a Phong-like PDF with exponent s,

$$p(\theta, \varphi) = \frac{s+1}{2\pi} \cos^s \theta, \tag{11}$$

we can sample a direction relative to the z-axis as follows:

```
1 cosTheta = pow(1-u[0],1/(1+s));
2 sinTheta = sqrt(1-cosTheta*cosTheta);
3 phi = 2*M_PI*u[1];
4 x = cos(phi)*sinTheta;
5 y = sin(phi)*sinTheta;
6 z = cosTheta;
```

Note that the generated direction may be below the surface indicated in the diagram. Most programs use a test to set the contribution of such directions to zero.

16.6.5 GGX DISTRIBUTION

The Trowbridge-Reitz GGX normal distribution function [12, 15]:

$$D(\theta_h) = \frac{\alpha^2}{\pi\left(1+\left(\alpha^2-1\right)\cos^2\theta_h\right)^2}, \tag{12}$$

is commonly used for the specular lobe in microfacet reflectance models. Its width or *roughness* parameter α defines the appearance of the surface, with lower values indicating shinier surfaces.

The GGX distribution can be sampled by transforming two-dimensional uniformly distributed samples into spherical coordinates for the half-vector as follows:

$$\theta_h = \arctan\left(\frac{\alpha\sqrt{u[0]}}{\sqrt{1-u[0]}}\right), \tag{13}$$

$$\varphi_h = 2\pi u[1], \tag{14}$$

where α is the GGX roughness parameter. It is often convenient to rewrite the expression using trigonometric identities to directly compute $\cos\theta_h$ as

$$\cos\theta_h = \sqrt{\frac{1-u[0]}{\left(\alpha^2-1\right)u[0]+1}} \tag{15}$$

and to use the Pythagorean identity to compute $\sin\theta_h = \sqrt{1-\cos^2\theta_h}$ as before. The PDF of the sampled half-vector is $p(\theta_h, \varphi_h) = D(\theta_h)\cos\theta_h$.

For rendering, we are usually interested in sampling incident directions based on a given outgoing direction and local tangent frame. To do so, the outgoing direction $\hat{\mathbf{v}}$ is reflected around the sampled half-vector $\hat{\mathbf{h}}$ to find the incident direction as $\hat{\mathbf{l}} = 2(\hat{\mathbf{v}} \cdot \hat{\mathbf{h}})\hat{\mathbf{h}} - \hat{\mathbf{v}}$. This operation changes the PDF above, which must be multiplied by the Jacobian of the transform that is $1/\left(4(\hat{\mathbf{v}} \cdot \hat{\mathbf{h}})\right)$ in this case [14].

As with the Phong sampling in Section 16.6.4, the generated direction can be below the surface. Typically these are areas where the integrand is zero, but programmers should make sure to handle these cases carefully.

16.7 VOLUME SCATTERING

For volumes, also often called *participating media*, rays will "collide" with the volume in a probabilistic fashion. Some programs do this with incremental *ray integration*, but an alternative is to compute discrete collisions. For more information on volumes for graphics, see Chapter 11 of Pharr et al. [8].

Also see Chapter 28 for more information on this topic.

16.7.1 DISTANCES IN A VOLUME

Tracing photons through scattering and absorbing media requires importance sampling of distances proportional to the volume transmittance

$$T(s) = \exp\left(-\int_0^s \kappa(t)\,dt\right) \tag{16}$$

for the volume extinction coefficient $\kappa(t)$. The PDF for this distribution is

$$p(s) = \kappa(s)\exp\left(-\int_0^s \kappa(t)\,dt\right). \tag{17}$$

16.7.1.1 HOMOGENEOUS MEDIA

In the case that κ is a constant, we have $p(s) = \kappa \exp(-s\kappa)$ and the inversion method can be used to obtain the following:

```
1 s = -log(1 - u) / kappa;
```

Note that $1 - u$ is important: remember that we assumed $u \in [0, 1)$, so $1 - u \in (0, 1]$, which avoids invoking the logarithm for zero!

16.7.1.2 INHOMOGENEOUS MEDIA

For spatially varying $\kappa(t)$, a procedure often referred to as *Woodcock tracking* gives the desired distribution [16]. Given the maximum extinction coefficient $\kappa_{\max}$ along the ray and a generator u for samples uniform in [0, 1), the procedure is as follows:

```
1 s = 0;
2 do {
3     s -= log(1 - u()) / kappa_max;
4 } while (kappa(s) < u() * kappa_max);
```

16.7.2 HENYEY-GREENSTEIN PHASE FUNCTION

The *Henyey-Greenstein phase function* is a useful tool to model the directional scattering characteristics inside a volume. It is a PDF on the sphere of all directions that depends only on the angle θ between the incoming and outgoing directions and that is controlled with a single parameter g (the average cosine):

$$p(\theta) = \frac{1-g^2}{4\pi\left(1+g^2-2g\cos\theta\right)^{3/2}}. \tag{18}$$

For $g = 0$ the scattering is isotropic, for g approaching −1 the scattering becomes highly focused forward scattering, and for g approaching 1 the scattering turns into highly focused backward scattering.

```
1 phi = 2.0 * M_PI * u[0];
2 if (g != 0) {
3     tmp = (1 - g * g) / (1 + g * (1 - 2 * u[1]));
4     cos_theta = (1 + g * g - tmp * tmp) / (2 * g);
5 } else {
6     cos_theta = 1 - 2 * u[1];
7 }
```

16.8 ADDING TO THE ZOO COLLECTION

We have presented a variety of transforms we have found useful for ray tracing programs. We have not delved deeply into the theory needed to add to this collection. Readers that want to learn more about that theory so they can add their own “animals” can find thorough treatments in the books by Pharr et al. [8], Glassner [5], and Dutré et al. [4].

REFERENCES

[1] Clarberg, P. Fast Equal-Area Mapping of the (Hemi)Sphere Using SIMD. *Journal of Graphics Tools 13*, 3 (2008), 53–68.

[2] Clarberg, P., Jarosz, W., Akenine-Möller, T., and Jensen, H. W. Wavelet Importance Sampling: Efficiently Evaluating Products of Complex Functions. *ACM Transactions on Graphics 24*, 3 (2005), 1166–1175.

[3] Conty Estévez, A., and Kulla, C. Importance Sampling of Many Lights with Adaptive Tree Splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 25:1–25:17.

[4] Dutré, P., Bekaert, P., and Bala, K. *Advanced Global Illumination*. A K Peters, 2006.

[5] Glassner, A. S. *Principles of Digital Image Synthesis*. Elsevier, 1995.

[6] Keller, A. Quasi-Monte Carlo Image Synthesis in a Nutshell. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*. Springer, 2013, pp. 213–249.

[7] Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, `http://arxiv.org/abs/1705.01263`, 2017.

[8] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, third ed. Morgan Kaufmann, 2016.

[9] Praun, E., and Hoppe, H. Spherical Parametrization and Remeshing. *ACM Transactions on Graphics 22*, 3 (2003), 340–349.

[10] Sbert, M. An Integral Geometry Based Method for Fast Form-Factor Computation. *Computer Graphics Forum 12*, 3 (1993), 409–420.

[11] Shirley, P., and Chiu, K. A Low Distortion Map Between Disk and Square. *Journal of Graphics Tools 2*, 3 (1997), 45–52.

[12] Trowbridge, T. S., and Reitz, K. P. Average Irregularity Representation of a Rough Surface for Ray Reflection. *Journal of the Optical Society of America 65*, 5 (1975), 531–536.

[13] Turk, G. Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion. *Computer Graphics (SIGGRAPH) 25*, 4 (July 1991), 289–298.

[14] Walter, B. Notes on the Ward BRDF. Tech. Rep. PCG-05-06, Cornell Program of Computer Graphics, April 2005.

[15] Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. Microfacet Models for Refraction Through Rough Surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (2007), pp. 195–206.

[16] Woodcock, E. R., Murphy, T., Hemmings, P. J., and Longworth, T. C. Techniques Used in the GEM Code for Monte Carlo Neutronics Calculations in Reactors and Other Systems of Complex Geometry. In *Applications of Computing Methods to Reactor Problems* (1965), p. 557.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.