# INFINITE AREA LIGHT SOURCE WITH IMPORTANCE SAMPLING

Matt Pharr and Greg Humphreys

*This document describes the solution to Exercise 15.4 on page 716.*

## 0.1 Introduction

The basic implementation of the Monte Carlo sampling methods in the implementation of the `InfiniteAreaLight` light source uses a cosine-weighted distribution of directions over the hemisphere when sampling incident illumination directions at the point being shaded. While this sampling distribution is guaranteed to lead to the correct result in the limit (due to having a non-zero probability of selecting any particular direction), it may lead to high variance in direct lighting estimates if the environment map used for illumination is much brighter in some parts than others. Almost all environment maps of realistic scenes have this property; Figure 1 shows two examples.

This document shows how to use the environment map to define a probability density function for importance sampling over the sphere of directions. This approach is easy to implement and substantially reduces the variance of images rendered using environment map illumination because it does a good job of matching the distribution of one of the terms in the direct lighting integrand. In conjunction with Multiple Importance Sampling to weight the samples taken, high quality images can be generated at relatively low sampling rates.

Figure 2 shows two images of the Audi TT car model illuminated by the morning skylight environment map from Figure 1. The top image was rendered using the simple cosine-weighted sampling distribution, while the bottom image was rendered using the improved sampling method implemented here. Both images used just 32 shadow samples per pixel. For the same number of samples taken and at negligable computational cost, the new sampling method computes a substantially better result with much lower variance.

With an environment map with smaller regions of bright focused light like the St. Peter's environment map is used for illumination, using importance sampling in this manner is even more effective. With the cosine sampling method, sometimes none of the samples will be in the important bright regions and other times many of them will be. The end result is excessive variance. The killeroo images in Figure 3 compare the two sampling approaches with the St. Peter's environment map.

In contrast to methods for environment map sampling like those developed by Kollig and Keller (Kollig and Keller 2003) and Agarwal et al. (Agarwal, Ramamoorthi, Belongie, and Jensen 2003), this approach requires almost no precomputation time (as opposed to minutes of preprocessing time). As with those approaches, there is negligible computational cost at render-time. In our experience, it gives results with equivalent quality to those from those approaches with substantially less implementation complexity.

## 0.2 Implementation

The `InfiniteAreaLightIS` light source implements the sampling method described above. Most of its implementation is the same as `InfiniteAreaLight`; we will just discuss the differences here.

There are three main steps to the sampling approach implemented here:

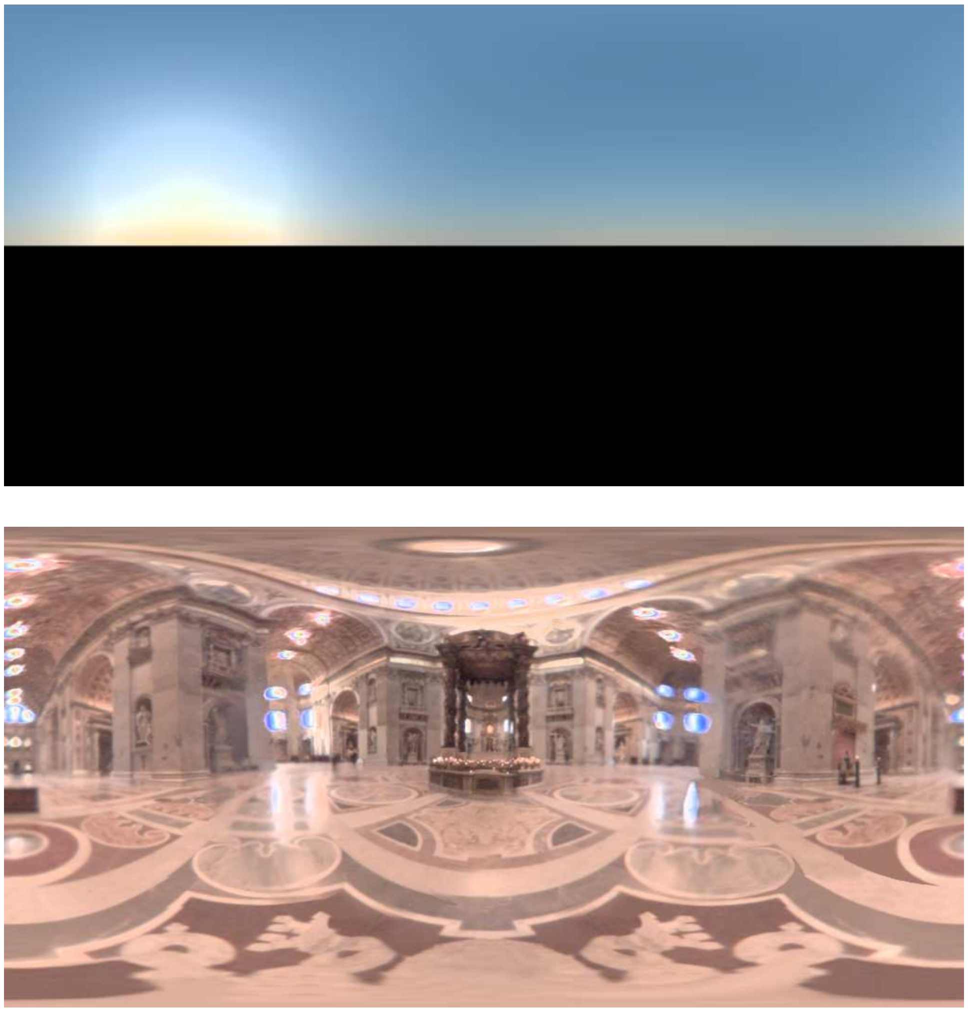
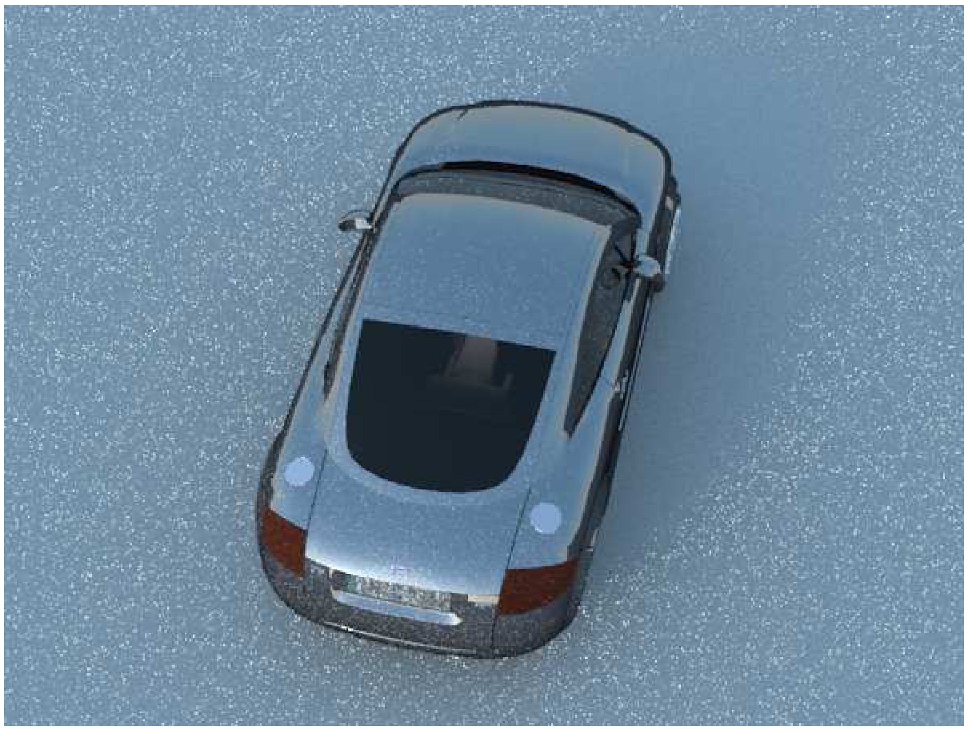


Figure 1: A latitude-longitude environment map from a simulation of the early morning sky (top) and an environment of St. Peter's Cathedral (bottom). These images are used for illuminating the TT model in Figure 2 and the Killeroo in Figure 3. These environment maps cover multiple orders of magnitude in variation of the radiance function's value. (Images courtesy Nolan Goodnight and Paul Debevec, respectively.).



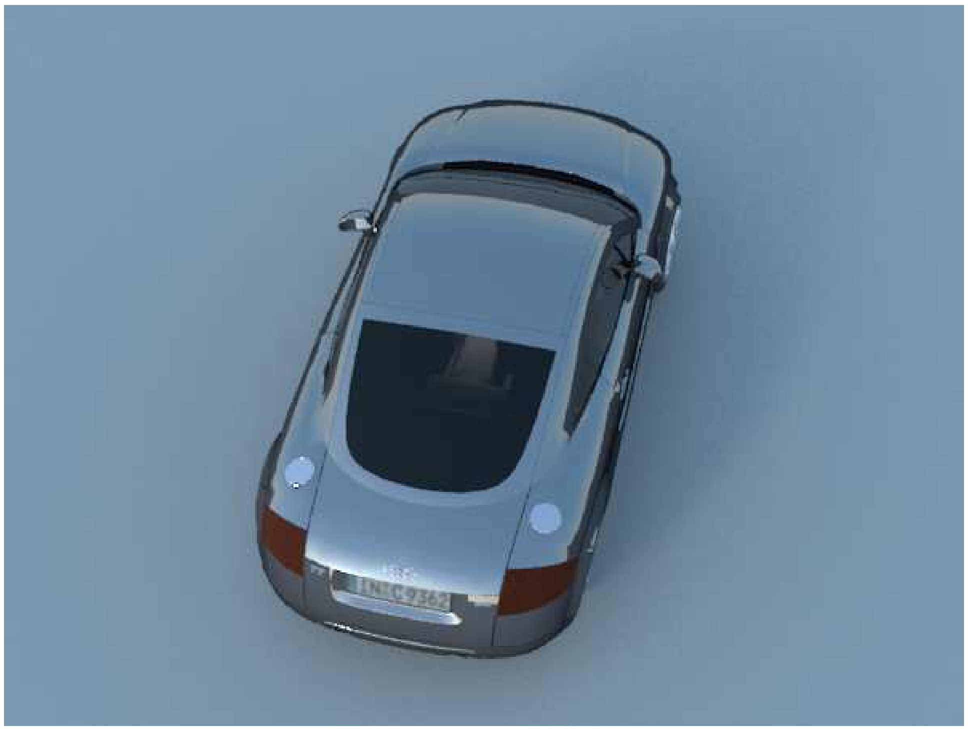


Figure 2: TT car model illuminated by the morning skylight environment map, rendered with four image samples per pixel and eight light source samples per image sample. The top image shows the result from using a uniform sampling distribution, whlie the bottom image shows the improvement from the method implemented here. Note that a total of just 32 light samples per pixel gives an excellent result with this approach.

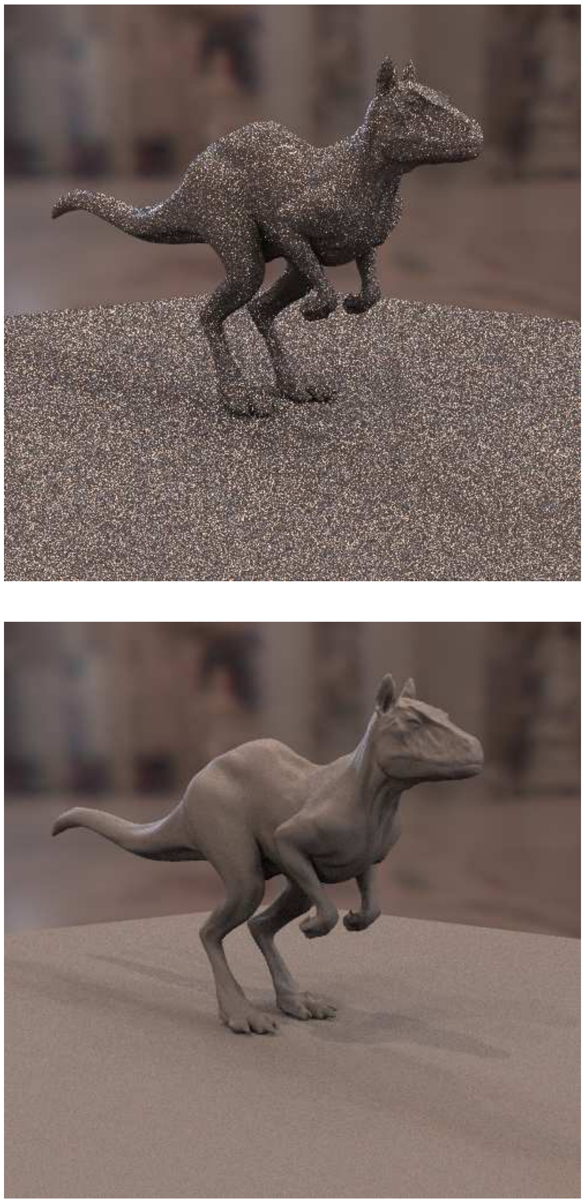


Figure 3: Killeroo illuminated by the St. Peter's cathedral environment map, rendered with four image samples per pixel and sixteen light source samples per image sample. The top image was rendered using a uniform sampling distribution, giving a result with very high variance. The bottom image shows the substantial improvement from the method implemented here.

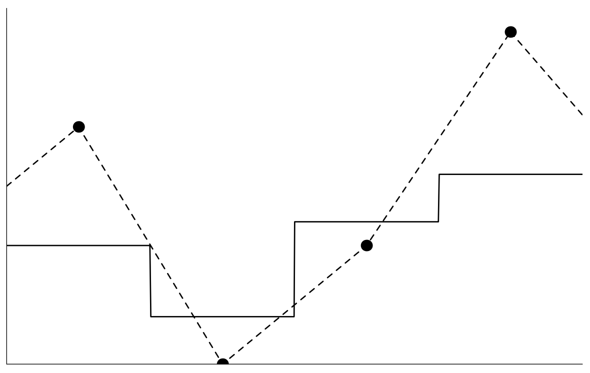


Figure 4: 1D example of finding a piecewise-constant function (solid lines) that approximates a piecewise linear function (dashed lines) for use as a sampling distribution for importance sampling. Even though some of the sample points that define the piecewise linear function (solid dots) may be zero-valued, the piecewise-constant function must not be zero over a finite range. A reasonable approach to avoid this case, shown here and implemented in the `InfiniteAreaLightIS` integrator, is to find the average value of the function over some range and use that to define the piecewise-constant function.

- Define a piecewise-constant 2D function in image coordinates $(u,v)$, $p(u,v)$ that is based on the distribution of the radiance function represented by the environment map.
- Develop a sampling method to transform uniformly distributed 2D random numbers to samples drawn from the piecewise-constant $p(u,v)$ distribution.
- Define a probability density function over directions on the unit sphere based on the probability density over $(u,v)$.

The combination of these three steps makes it possible to generate samples on the sphere of directions according to a distribution that matches the radiance function very closely, leading to substantial variance reduction.

The `InfiniteAreaLightIS` constructor precomputes the piecewise-constant function and its CDF for sampling. The first step in this process is to transform the continuously-defined spectral radiance function defined by the environment map's texels to a piecewise-constant scalar function by computing its luminance at a set of sample points using the `Spectrum::y()` method. There are three things to note in the code below that does this computation.

First, it computes values of the radiance function at the same number of points as there are texels in the original image map. It could use either more or fewer points, leading to a corresponding increase or decrease in memory use while still generating a valid sampling distribution, however. These values work well, though, as fewer points would lead to a sampling distribution that didn't match the function as well while more would mostly waste memory.

The second thing of note in this code is that the piecewise constant function values being stored here in `img` are found by slightly blurring the radiance function with the `MIPMap::Lookup()` method (rather than just copying the corresponding texel values). The motivation for this is subtle; Figure 4 illustrates the idea in

1D. Because the continuous radiance function used for rendering is reconstructed by bilinearly interpolating between texels in the image, just because some texel is completely black, for example, the radiance function may be non-zero a tiny distance away from it due to a neighboring texel's contribution. Because we are using a piecewise-constant function for sampling rather than a piecewise-linear one, it must account for this issue in order to ensure greater-than-zero probability of sampling any point where the radiance function is non-zero. (Alternatively, we could use a piecewise-linear function for importance sampling and thus match the radiance function exactly. However, it's easier to draw samples from a piecewise-constant function's distribution and because environment maps generally have a large number of texel samples, a piecewise-constant function suffices to match its distribution well.)

Finally, note that the loops over `nu` and `nv` and the indexing scheme used for the `img` array are interchanged versus typical C++ usage (where the outler loop would be over `nv` rather than `nu` and where `img` would be indexed as `img[u+v*nu]`. We made these slightly unusual choices here so that later code can more closely match the mathematics of the sampling method's derivation.

⟨*Compute scalar-valued image from environment map*⟩≡ used: none

```
float filter = 1.f / max(width, height);
int nu = width, nv = height;
float *img = new float[width*height];
for (int u = 0; u < nu; ++u) {
    float up = (float)u / (float)nu;
    for (int v = 0; v < nv; ++v) {
        float vp = (float)v / (float)nv;
        img[v+u*nv] = radianceMap->Lookup(up, vp, filter).y();
    }
}
```

Like the `InfiniteAreaLight`, `InfiniteAreaLightIS` stores the image using a `MIPMap`.

⟨*InfiniteAreaLightIS Private Data*⟩≡ used: none

```
Spectrum Lbase;
MIPMap<Spectrum> *radianceMap;
```

Sampling from the 2D piecewise-constant function now stored in the `img` array can be done as a two-step process. Intuitively, first we choose sample along one column of the image, based on a 1D probability density defined by the integral of the function along the columns. (Thus, columns with relatively bright environment map function values are more likely to be selected.) Next, given such a column we need to sample from the distribution of the function along the column, Each of these steps is just a 1D sampling problem. Figure 5 shows this idea with a low-resolution image.

More formally, to understand how to draw a sample from a a 2D distribution $p(u,v)$, recall from Section 14.5 that for general multidimensional joint probability distributions, each dimension must be sampled in turn in a manner based on the sample values chosen for previous dimensions.

For the case here, consider a function $f(u,v)$ defined over $\mathbb{R}^2$ by a set of $n_u n_v$ values $f_{\tilde{u},\tilde{v}}$ where $f_{\tilde{u},\tilde{v}}$ gives the value of $f$ over the range $[u,u+1) \times [v,v+1)$.

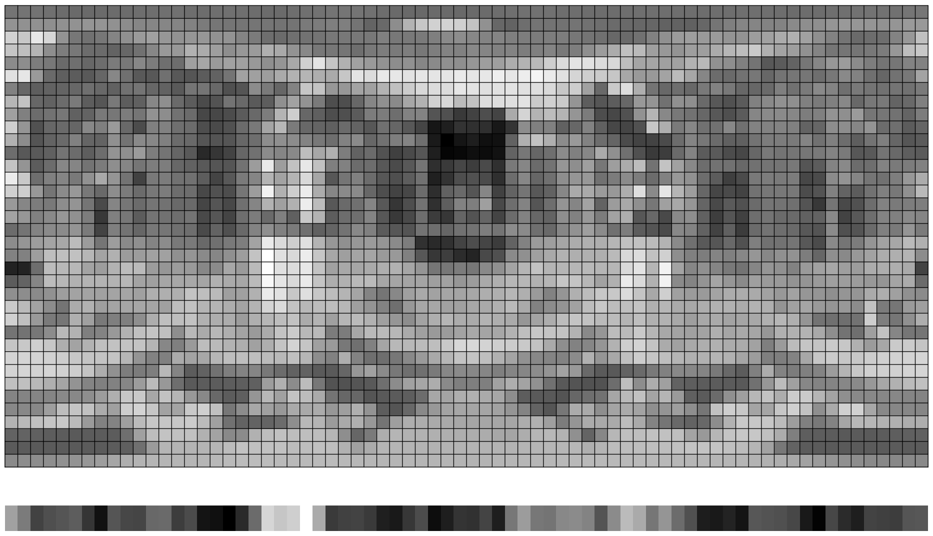


Figure 5: Plot of the piecewise-constant sampling distribution for the St. Peter's environment map (top) and the marginal density function $p_u(u)$ (bottom). First the 1D distribution at the bottom is used to select a $u$ value, giving a column of the image to sample. Columns with bright pixels are more likely to be sampled. Then, given a column, a value $v$ is sampled from that column's 1D distribution.

Recall that the joint distribution of a 2D function is defined as

$$p(u,v) = \frac{f(u,v)}{\int\int f(u,v)\,\mathrm{d}u\,\mathrm{d}v}.$$

Thanks to $f$'s definition, integrals of its value are simple sums of $f_{\tilde{u},\tilde{v}}$ values, so that

$$p(u,v) = \frac{f(u,v)}{\sum_u \sum_v f_{\tilde{u},\tilde{v}}}.$$

As usual, we will define

$$I_f = \int\int f(u,v)\,\mathrm{d}u\,\mathrm{d}v = \sum_u \sum_v f_{\tilde{u},\tilde{v}}.$$

The marginal density $p_u(u)$ is easily found as a sum of $f_{\tilde{u},\tilde{v}}$ values

$$p_u(u) = \int p(u,v)\,\mathrm{d}v = \frac{\sum_v f_{\tilde{u},\tilde{v}}}{I_f},$$

for $\tilde{u} = \lfloor u \rfloor$. Note that $p_u(u)$ is itself a piecewise constant function with $n_u$ values. These values be easily computed in a preprocessing step, and thus $u$ samples can be taken from its distribution using the approach for sampling piecewise constant function described in Section 14.3.4.

Given such a $u$ sample, the conditional density $p_v(v|u)$ is

$$p_v(v|u) = \frac{p(u,v)}{p_u(u)} = \frac{\frac{f_{\tilde{u},\tilde{v}}}{I_f}}{p_u(u)}.$$

If the piecewise constant $p_u(u)$ function is represented as a set of values $g_{\tilde{u}}$ with the conventions above, we have

$$p_v(v|u) = \frac{(f_{\tilde{u},\tilde{v}}/I_f)}{g_{\tilde{u}}},$$

itself a piecewise-constant function that can be sampled with the same one-dimensional approach.

Given this context, the following fragment from the `InfiniteAreaLightIS` constructor computes the piecewise constant distribution $p_u(u)$ as well as the $n_u$ distinct piecewise constant distributions $p(v|u)$. First some working memory is allocated and the value of $\sin\theta$ for each row of the latitude-longitude image is computed. The use of these $\sin\theta$ values will be explained shortly.

⟨*Initialize sampling PDFs for infinite area light*⟩≡ used: none

```
float *func = (float *)alloca(max(nu, nv) * sizeof(float));
float *sinVals = (float *)alloca(nv * sizeof(float));
for (int i = 0; i < nv; ++i)
    sinVals[i] = sin(M_PI * float(i+.5)/float(nv));
vDistribs = new Distribution1D *[nu];
for (int u = 0; u < nu; ++u) {
    ⟨Compute sampling distribution for column u 8⟩
}
⟨Compute sampling distribution for columns of image 9⟩
```

⟨*InfiniteAreaLightIS Private Data*⟩+≡ used: none

```
Distribution1D *uDistrib, **vDistribs;
```

First the $p(v|u)$ distributions are found. The function values are copied from the luminance image into the temporary `func` buffer and are multiplied by the value of $\sin\theta$ corresponding to the $\theta$ value each row has when the latitude-longitude image is mapped to the sphere. Note that this multiplication has no effect on the sampling method's correctness: because its value is always greater than zero, we are just reshaping the sampling PDF. The motivation for adjusting the PDF is to eliminate the affect of the distortion from mapping the 2D image to the unit sphere in the sampling method here. It will be fully explained later.

Note also that this loop linearly steps through the `img` array in memory. If `img` had been initialized previously with the usual indexing scheme–`img[u+v*nu]`– then this loop would have a stride of `nu` floats in memory, leading to many more cache misses. Indeed, this initialization step is nearly an order of magnitude slower if the usual stepping is used.

⟨*Compute sampling distribution for column* `u`⟩≡ used: 8

```
for (int v = 0; v < nv; ++v)
    func[v] = img[u*nv+v] * sinVals[v];
vDistribs[u] = new Distribution1D(func, nv);
```

`Distribution1D` is a small utility class that represents a piecewise-constant 1D function's distribution.

⟨*Utility Classes and Functions*⟩≡ used: none
```
struct Distribution1D {
    ⟨Distribution1D Methods 9⟩
    ⟨Distribution1D Data 9⟩
};
```

The `Distribution1D` constructor takes the values of a piecewise-constant function `f` with `n` values. It makes its own copy of the function values, computes the function's CDF, and stores some auxiliary data, including the integral of the function, `funcInt` and its reciprocal, `invFuncInt`.

⟨*Distribution1D Methods*⟩≡ used: 9
```
Distribution1D(float *f, int n) {
    func = new float[n];
    cdf = new float[n+1];
    count = n;
    memcpy(func, f, n*sizeof(float));
    ComputeStep1dCDF(func, n, &funcInt, cdf);
    invFuncInt = 1.f / funcInt;
    invCount = 1.f / count;
}
```

⟨*Distribution1D Data*⟩≡ used: 9
```
float *func, *cdf;
float funcInt, invFuncInt, invCount;
int count;
```

Given the conditional densities for each column of the image, we an find the 1D density for sampling a particular column, $p_u(u)$. The `Distribution1D` class stores the integral of its piecewise-constant function in its `funcInt` member variable, so it's just necessary to copy these values to the `func` buffer and construct the `Distribution1D` for $p_u(u)$.

⟨*Compute sampling distribution for columns of image*⟩≡ used: 8
```
for (int u = 0; u < nu; ++u)
    func[u] = vDistribs[u]->funcInt;
uDistrib = new Distribution1D(func, nu);
```

Given this precomputed data, the task of the sampling method is relatively straightforward. Given a pair of uniformly distributed random variables $(\xi_1, \xi_2)$ over $[0,1]^2$, it draws a sample from the function's distribution using the sampling algorithm described previously, giving a $(u,v)$ value and the value of the probability density function for taking this sample, $p(u,v)$. The $(u,v)$ sample is mapped to spherical coordinates by

$$(\theta, \phi) = \left(\frac{\pi v}{n_v}, \frac{2\pi u}{n_u}\right)$$

and then using the spherical coordinates formula to give the direction $\omega = (x,y,z)$.

⟨*InfiniteAreaLightIS Definitions*⟩+≡ used: none

```
Spectrum InfiniteAreaLightIS::Sample_L(const Point &p, float u1,
        float u2, Vector *wi, float *pdf,
        VisibilityTester *visibility) const {
    ⟨Find floating-point (u,v) sample coordinates 10⟩
    ⟨Convert sample point to direction on the unit sphere 10⟩
    ⟨Compute PDF for sampled direction 11⟩
    ⟨Return radiance value for direction 11⟩
}
```

As described previously, first a sample is drawn from the $p_u(u)$ distribution in order to find the $u$ coordinate of the sample. Rounding this floating point value down gives which column to use for sampling the $v$ value.

⟨*Find floating-point* $(u,v)$ *sample coordinates*⟩≡ used: 10

```
float pdfs[2];
float fu = uDistrib->Sample(u1, &pdfs[0]);
int u = Clamp(Float2Int(fu), 0, uDistrib->count-1);
float fv = vDistribs[u]->Sample(u2, &pdfs[1]);
```

`Distribution1D`'s `Sample()` method is very similar to `SampleStep1d()`. The only differences are that it returns a value over $[0,\texttt{count})$, not $[0,1)$, it takes fewer parameters, since the `Distribution1D` class variables store many of the values needed, and it is `inline`, thus making calls to it more efficient.

⟨*Distribution1D Methods*⟩+≡ used: 9

```
float Sample(float u, float *pdf) {
    // Find surrounding cdf segments
    float *ptr = std::lower_bound(cdf, cdf+count+1, u);
    int offset = (int) (ptr-cdf-1);
    // Return offset along current cdf segment
    u = (u - cdf[offset]) / (cdf[offset+1] - cdf[offset]);
    *pdf = func[offset] * invFuncInt;
    return offset + u;
}
```

Given the $(u,v)$ sample position, it's easy to first convert this to a $(\theta,\phi)$ sample and thence to a direction on the sphere.

⟨*Convert sample point to direction on the unit sphere*⟩≡ used: 10

```
float theta = fv * vDistribs[u]->invCount * M_PI;
float phi = fu * uDistrib->invCount * 2.f * M_PI;
float costheta = cos(theta), sintheta = sin(theta);
float sinphi = sin(phi), cosphi = cos(phi);
*wi = LightToWorld(Vector(sintheta * cosphi, sintheta * sinphi,
                          costheta));
```

Recall that the probability density values returned by the light source sampling routines must be defined in terms of the solid angle measure on the unit sphere.

Therefore, this routine must now compute the transformation between the sampling density used, which was the image function over $(n_u, n_v)$ and the corresponding density after the image has been mapped to the unit sphere with the latitude-longitude mapping. (Recall that the latitude-longitude parameterization of an image $(\theta, \phi)$ is $x = r \sin\theta \cos\phi$, $y = r \sin\theta \sin\phi$, and $z = r \cos\theta$.)

First, consider the function $g$ that maps from $(u, v)$ to $(\theta, \phi)$,

$$g(u,v) = \left( \frac{\pi v}{n_v}, \frac{2\pi u}{n_u} \right).$$

The absolute value of the determinant of the Jacobian $|J_g|$ is $2\pi^2/(n_u n_v)$. Applying the multidimensional change of variables equation from Section 14.4.1 on page 648, we can find the density in terms of spherical coordinates $(\theta, \phi)$.

$$p(\theta,\phi) = p(u,v) \frac{n_u n_v}{2\pi^2}.$$

From the definition of spherical coordinates, it is easy to determine that the absolute value of the Jacobian for the mapping from $(r, \theta, \phi)$ to $(x, y, z)$ is $r^2 \sin\theta$. Since we are interested in the unit sphere, $r = 1$, and again applying the multidimensional change of variables equation to find the final relationship between probability densities in terms of the probability density for the sample from $(\theta, \phi)$ constant function to the direction on the sphere,

$$p(\omega) = \frac{p(\theta,\phi)}{\sin\theta} = p(u,v) \frac{n_u n_v}{2\pi^2 \sin\theta}.$$

This is the key relationship for applying this technique: it lets us sample from the piecewise-constant distribution defined by the image map and transform the sample and its probability density to be in terms of directions on the unit sphere.

Here we can now see why the initialization routines multiplied the values of the piecewise-constant sampling function by a $\sin\theta$ term. Consider for example a constant-valued environment map: with the $p(u, v)$ sampling technique, all $(\theta, \phi)$ values are equally likely to be chosen. Due to the mapping to directions on the sphere, however, this would lead to more directions being sampled near the poles of the sphere, *not* a uniform sampling of directions on the sphere, which is the desired result. The $1/\sin\theta$ term in the $p(\omega)$ PDF corrects for this non-uniform sampling of directions so that correct results are computed in Monte Carlo estimates. Given this state of affairs, however, it's better to have modified the $p(u, v)$ sampling distribution so that it's less likely to select directions near the poles in the first place.

⟨*Compute PDF for sampled direction*⟩≡ used: 10

```
  *pdf = (pdfs[0] * pdfs[1]) / (2.f * M_PI * M_PI * sintheta);
```

⟨*Return radiance value for direction*⟩≡ used: 10

```
  visibility->SetRay(p, *wi);
  return Lbase * radianceMap->Lookup(fu * uDistrib->invCount,
                                     fv * vDistribs[u]->invCount);
```

Computing the PDF given a direction is also pretty straightforward. This method just needs to convert the direction $\omega$ to the corresponding $(u, v)$ coordinates in the sampling distribution. Given these, the PDF $p(u, v)$ is easily computed as the product of the two 1D PDFs, adjusted for the mapping to the sphere as done previously.

⟨*InfiniteAreaLightIS Definitions*⟩+≡ used: none

```
float InfiniteAreaLightIS::Pdf(const Point &,
        const Vector &w) const {
    Vector wi = WorldToLight(w);
    float theta = SphericalTheta(wi), phi = SphericalPhi(wi);
    int u = Clamp(Float2Int(phi * INV_TWOPI * uDistrib->count),
                  0, uDistrib->count-1);
    int v = Clamp(Float2Int(theta * INV_PI * vDistribs[u]->count),
                  0, vDistribs[u]->count-1);
    return (uDistrib->func[u] * vDistribs[u]->func[v]) /
           (uDistrib->funcInt * vDistribs[u]->funcInt) *
           1.f / (2.f * M_PI * M_PI * sin(theta));
}
```

## Exercises

0.1 The implementation here still doesn't use any form of importance sampling for sampling rays leaving the light source for use by integrators like the `PhotonIntegrator` that follow paths starting from the light source. Figure out how to apply this sampling technique to that case as well and render images showing the improvement from doing so.

0.2 One potential shortcoming of this sampling method is the case of an environment map with extremely small extremely bright regions. In that case, all of the samples taken may end up in the very bright area, with none in the dimmer areas, leading to a poor stratification of samples. If the bright part happens to be occluded at a particular point being shaded such that the dimmer areas of the environment map are the only ones that illuminate the point, the approach here will not be effective. Construct a scene where this problem occurs. Is this problem evident with real-world environment maps? How much does the BSDF sampling done for multiple importance sampling in the direct lighting calculation ameliorate this problem?

# Bibliography

Agarwal, S., R. Ramamoorthi, S. Belongie, and H. W. Jensen (2003, July). Structured importance sampling of environment maps. *ACM Transactions on Graphics 22*(3), 605–612.

Kollig, T. and A. Keller (2003, June). Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pp. 45–51.