**Specular IBL****镜面反射 IBL**

In the [previous](https://learnopengl.com/#!PBR/IBL/Diffuse-irradiance) tutorial we've set up PBR in combination with image based lighting by pre-computing an irradiance map as the lighting's indirect diffuse portion. 在上一节教程中，我们预计算了辐照度图作为照明的间接漫射部分，将 PBR 与基于图像的照明相结合。In this tutorial we'll focus on the specular part of the reflectance equation:在本教程中，我们将重点关注反射方程的镜面反射部分：

Lo(p,ωo)=∫Ω(kdcπ+ksDFG4(ωo⋅n)(ωi⋅n))Li(p,ωi)n⋅ωidωiLo(p,ωo)=∫Ω(kdcπ+ksDFG4(ωo⋅n)(ωi⋅n))Li(p,ωi)n⋅ωidωi$$

L\_o(p,\omega\_o) = \int\limits\_{\Omega}

(k\_d\frac{c}{\pi} + k\_s\frac{DFG}{4(\omega\_o \cdot n)(\omega\_i \cdot n)})

L\_i(p,\omega\_i) n \cdot \omega\_i d\omega\_i

$$

You'll notice that the Cook-Torrance specular portion (multiplied by kSkS) isn't constant over the integral and is dependent on the incoming light direction, but **also** the incoming view direction. 你会注意到 Cook-Torrance 镜面反射部分（乘以\(k\_s\)在整个积分上不是常数，不仅受入射光方向影响，还受视角影响。Trying to solve the integral for all incoming light directions including all possible view directions is a combinatorial overload and way too expensive to calculate on a real-time basis. 试图解决所有入射光方向加所有可能的视角方向的积分，二者组合数会极其庞大，实时计算太昂贵。Epic Games proposed a solution where they were able to pre-convolute the specular part for real time purposes, given a few compromises, known as the split sum approximation.Epic Games 提出了一个解决方案，他们预计算镜面部分的卷积，为了实时计算作了一些妥协，这种方案被称为分割求和近似法（split sum approximation）。

The split sum approximation splits the specular part of the reflectance equation into two separate parts that we can individually convolute and later combine in the PBR shader for specular indirect image based lighting. 分割求和近似将方程的镜面反射部分分裂成两个独立的部分，我们可以单独求卷积，然后在 PBR 着色器中求和，以用于间接镜面反射部分 IBL。Similar to how we pre-convoluted the irradiance map, the split sum approximation requires an HDR environment map as its convolution input. 类似于我们如何预先对辐照图进行卷积的方法，分割求和近似法需要 HDR 环境贴图作为其卷积输入。To understand the split sum approximation we'll again look at the reflectance equation, but this time only focus on the specular part (we've extracted the diffuse part in the [previous](https://learnopengl.com/#!PBR/IBL/Diffuse-irradiance) tutorial):为了理解分割求和近似法，我们回顾一下反射方程，但这次只关注镜面反射部分（我们在前一个教程中剥离了漫反射部分）：

Lo(p,ωo)=∫Ω(ksDFG4(ωo⋅n)(ωi⋅n)Li(p,ωi)n⋅ωidωi=∫Ωfr(p,ωi,ωo)Li(p,ωi)n⋅ωidωiLo(p,ωo)=∫Ω(ksDFG4(ωo⋅n)(ωi⋅n)Li(p,ωi)n⋅ωidωi=∫Ωfr(p,ωi,ωo)Li(p,ωi)n⋅ωidωi$$

L\_o(p,\omega\_o) = \int\limits\_{\Omega}

(k\_d\frac{c}{\pi} + k\_s\frac{DFG}{4(\omega\_o \cdot n)(\omega\_i \cdot n)})

L\_i(p,\omega\_i) n \cdot \omega\_i d\omega\_i

$$

For the same (performance) reasons as the irradiance convolution, we can't solve the specular part of the integral in real time and expect a reasonable performance. 由于与辐照度卷积相同的（性能）原因，我们无法以合理的性能实时求解积分的镜面反射部分。So preferably we'd pre-compute this integral to get something like a specular IBL map, sample this map with the fragment's normal and be done with it. 因此，我们最好预先计算这个积分，以得到像镜面 IBL 图这样的东西，用片段的法线对这张图采样并计算。However, this is where it gets a bit tricky. 但是，有一个有点棘手的地方。We were able to pre-compute the irradiance map as the integral only depended on ωiωi and we could move the constant diffuse albedo terms out of the integral. 我们能够预计算辐照度图，因为其积分仅依赖于\(\Omega\_i\)，并且我们可以将常数的漫反射反射率项移出积分。This time, the integral depends on more than just ωiωi as evident from the BRDF:这一次，积分不仅仅取决于\(\Omega\_i\)，从 BRDF 可以看出：

fr(p,wi,wo)=DFG4(ωo⋅n)(ωi⋅n)fr(p,wi,wo)=DFG4(ωo⋅n)(ωi⋅n)fR（P，WI，WO）= DFG4（ωo⋅n）（ωi⋅n）FR（P，WI，WO）= DFG4（ωo⋅n）（ωi⋅n）

This time the integral also depends on wowo and we can't really sample a pre-computed cubemap with two direction vectors. 这次积分也取决于\(\Omega\_o\)，我们无法以两个方向向量采样预计算的立方体图。The position pp is irrelevant here as described in the previous tutorial. 如前一个教程中所述，位置\(p\)与此处无关。Pre-computing this integral for every possible combination of ωiωi and ωoωo isn't practical in a real-time setting.在实时状态下，对每种可能的\(\Omega\_i\)和\(\Omega\_o\)的组合预计算该积分是不可行的。

Epic Games' split sum approximation solves the issue by splitting the pre-computation into 2 individual parts that we can later combine to get the resulting pre-computed result we're after. Epic Games 的分割求和近似法将预计算分成两个单独的部分求解，再将两部分组合起来得到后文给出的预计算结果。The split sum approximation splits the specular integral into two separate integrals:分割求和近似法将镜面反射积分拆成两个独立的积分：

Lo(p,ωo)=∫ΩLi(p,ωi)dωi∗∫Ωfr(p,ωi,ωo)n⋅ωidωiLo(p,ωo)=∫ΩLi(p,ωi)dωi∗∫Ωfr(p,ωi,ωo)n⋅ωidωi$$

L\_o(p,\omega\_o) =

k\_d\frac{c}{\pi} \int\limits\_{\Omega} L\_i(p,\omega\_i) n \cdot \omega\_i d\omega\_i

$$

The first part (when convoluted) is known as the pre-filtered environment map which is (similar to the irradiance map) a pre-computed environment convolution map, but this time taking roughness into account. 卷积的第一部分被称为预滤波环境贴图，它类似于辐照度图，是预先计算的环境卷积图，但这次考虑了粗糙度。For increasing roughness levels, the environment map is convoluted with more scattered sample vectors, creating more blurry reflections. 随着粗糙度增加，参与环境贴图卷积的采样向量会更分散，导致更模糊的反射。For each roughness level we convolute, we store the sequentially blurrier results in the pre-filtered map's mipmap levels. 对于卷积的每个粗糙度级别，我们将按顺序把模糊后的结果存储在预滤波贴图的 mipmap 中。For instance, a pre-filtered environment map storing the pre-convoluted result of 5 different roughness values in its 5 mipmap levels looks as follows:例如，预过滤的环境贴图在其 5 个 mipmap 级别中存储 5 个不同粗糙度值的预卷积结果，如下所示：



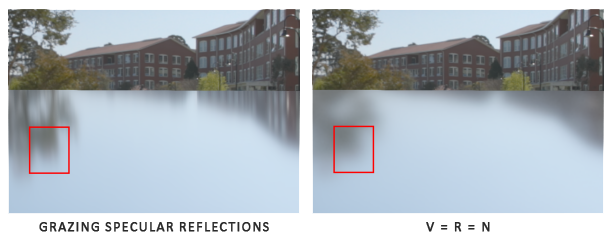
We generate the sample vectors and their scattering strength using the normal distribution function (NDF) of the Cook-Torrance BRDF that takes as input both a normal and view direction. 我们使用 Cook-Torrance BRDF 的正态分布函数 (NDF) 生成采样向量及其散射强度，该函数将法线和视角方向作为输入。As we don't know beforehand the view direction when convoluting the environment map, Epic Games makes a further approximation by assuming the view direction (and thus the specular reflection direction) is always equal to the output sample direction ωoωo. 由于我们在卷积环境贴图时事先不知道视角方向，因此 Epic Games 假设视角方向——也就是镜面反射方向——总是等于输出采样方向\(\Omega\_o\)，以进一步近似。This translates itself to the following code:翻译成代码如下：

vec3 N = normalize(w\_o);

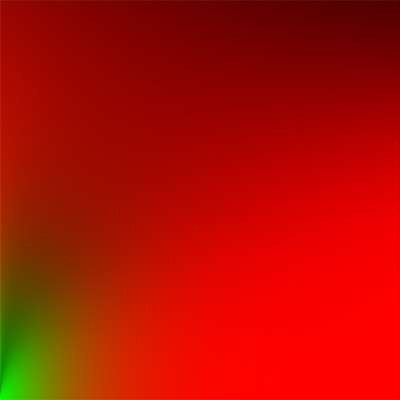
vec3 R = N;

vec3 V = R;

This way the pre-filtered environment convolution doesn't need to be aware of the view direction. 这样，预过滤的环境卷积就不需要关心视角方向了。This does mean we don't get nice grazing specular reflections when looking at specular surface reflections from an angle as seen in the image below (courtesy of the *Moving Frostbite to PBR* article); this is however generally considered a decent compromise:这意味着当从如下图的角度观察表面的镜面反射时，得到的掠角镜面反射效果不是很好（图片来自文章《Moving Frostbite to PBR》）。然而，通常认为这是一个体面的妥协：



The second part of the equation equals the BRDF part of the specular integral. 等式的第二部分等于镜面反射积分的 BRDF 部分。If we pretend the incoming radiance is completely white for every direction (thus L(p,x)=1.0L(p,x)=1.0) we can pre-calculate the BRDF's response given an input roughness and an input angle between the normal nn and light direction ωiωi, or n⋅ωin⋅ωi. 如果我们假设每个方向的入射辐射度都是白色的（因此 L(p,x)= 1.0 ），就可以在给定粗糙度、光线\(\Omega\_i\)和法线\(n\)夹角 \(n⋅\Omega\_i\) 的情况下，预计算 BRDF 的响应结果。Epic Games stores the pre-computed BRDF's response to each normal and light direction combination on varying roughness values in a 2D lookup texture (LUT) known as the BRDF integration map. Epic Games 将预计算好的 BRDF 对每个粗糙度和入射角的组合的响应结果存储在一张 2D 查找纹理(LUT)上，称为BRDF积分图。The 2D lookup texture outputs a scale (red) and a bias value (green) to the surface's Fresnel response giving us the second part of the split specular integral:2D 查找纹理的输出是菲涅耳响应的系数（红通道）和偏差值（绿通道），为我们提供了分割版镜面反射积分的第二个部分：



We generate the lookup texture by treating the horizontal texture coordinate (ranged between 0.0 and 1.0) of a plane as the BRDF's input n⋅ωin⋅ωi and its vertical texture coordinate as the input roughness value. 生成查找纹理的时候，我们以BRDF的输入\(n⋅\Omega\_i\)（范围在0.0和1.0之间）作为横坐标，以粗糙度作为纵坐标。With this BRDF integration map and the pre-filtered environment map we can combine both to get the result of the specular integral:有了此BRDF积分贴图和预过滤的环境贴图，我们就可以将两者结合起来，以获得镜面反射积分的结果：

**float** lod = getMipLevelFromRoughness(roughness);

vec3 prefilteredColor = textureCubeLod(PrefilteredEnvMap, refVec, lod);

vec2 envBRDF = texture2D(BRDFIntegrationMap, vec2(NdotV, roughness)).xy;

vec3 indirectSpecular = prefilteredColor \* (F \* envBRDF.x + envBRDF.y)

This should give you a bit of an overview on how Epic Games' split sum approximation roughly approaches the indirect specular part of the reflectance equation. 至此，你应该对Epic Games的分割求和近似法的原理，以及它如何近似求解反射方程的间接镜面反射部分有了一些基本印象。Let's now try and build the pre-convoluted parts ourselves.让我们现在尝试一下自己构建预卷积部分。

**Pre-filtering an HDR environment map****预过滤HDR环境贴图**

Pre-filtering an environment map is quite similar to how we convoluted an irradiance map. 预滤波环境贴图的方法与我们对辐射度贴图求卷积的方法非常相似。The difference being that we now account for roughness and store sequentially rougher reflections in the pre-filtered map's mip levels.对于卷积的每个粗糙度级别，我们将按顺序把模糊后的结果存储在预滤波贴图的 mipmap 中。

First, we need to generate a new cubemap to hold the pre-filtered environment map data. 首先，我们需要生成一个新的立方体贴图来保存预过滤的环境贴图数据。To make sure we allocate enough memory for its mip levels we call glGenerateMipmap as an easy way to allocate the required amount of memory.为了确保为其mip级别分配足够的内存，一个简单方法是调用glGenerateMipmap。

**unsigned** **int** prefilterMap;

glGenTextures(1, &prefilterMap);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, prefilterMap);

**for** (**unsigned** **int** i = 0; i < 6; ++i)

{

glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_RGB16F, 128, 128, 0, GL\_RGB, GL\_FLOAT, **nullptr**);

}

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE);

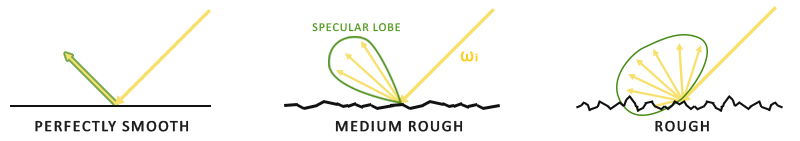
glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

glGenerateMipmap(GL\_TEXTURE\_CUBE\_MAP);

Note that because we plan to sample the prefilterMap its mipmaps you'll need to make sure its minification filter is set to GL\_LINEAR\_MIPMAP\_LINEAR to enable trilinear filtering. 注意，因为我们计划对prefilterMap采样mipmap，所以您需要确保将其缩小过滤器设置为GL\_LINEAR\_MIPMAP\_LINEAR以启用三线性过滤。We store the pre-filtered specular reflections in a per-face resolution of 128 by 128 at its base mip level. 我们以每面128×128的分辨率，在其基础mip级别存储预滤波的镜面反射。This is likely to be enough for most reflections, but if you have a large number of smooth materials (think of car reflections) you may want to increase the resolution.对于大多数反射来说，这可能已经足够了，但如果您有大量光滑材料（想想汽车反射），您可能需要提高分辨率。

In the previous tutorial we convoluted the environment map by generating sample vectors uniformly spread over the hemisphere ΩΩ using spherical coordinates. 在上一个教程中，我们使用球面坐标生成均匀分布在半球\(\Omega\)上的采样向量，以对环境贴图进行卷积。While this works just fine for irradiance, for specular reflections it's less efficient. 虽然这个方法非常适用于辐照度，但对于镜面反射效果较差。When it comes to specular reflections, based on the roughness of a surface, the light reflects closely or roughly around a reflection vector rr over a normal nn, but (unless the surface is extremely rough) around the reflection vector nonetheless:镜面反射依赖于表面的粗糙度，反射光线可能比较松散，也可能比较紧密，但是一定会围绕着反射向量\(r\)进行反射，除非表面极度粗糙：



The general shape of possible outgoing light reflections is known as the specular lobe. 所有可能出射的反射光构成的形状称为高光波瓣。As roughness increases, the specular lobe's size increases; and the shape of the specular lobe changes on varying incoming light directions. 随着粗糙度的增加，高光波瓣的大小增加；随着入射光方向不同，形状会发生变化。The shape of the specular lobe is thus highly dependent on the material.因此，高光波瓣的形状高度依赖于材质。

When it comes to the microsurface model, we can imagine the specular lobe as the reflection orientation about the microfacet halfway vectors given some incoming light direction. 当谈及微表面模型时，给定入射光方向之后，镜面波瓣朝向微平面的半向量的反射方向。Seeing as most light rays end up in a specular lobe reflected around the microfacet halfway vectors it makes sense to generate the sample vectors in a similar fashion as most would otherwise be wasted. 考虑到大多数光线最终反射到一个基于半向量的高光波瓣内，采样时根据相近的方式选取采样向量是有意义的，因为大部分其余的向量都被浪费掉了。This process is known as importance sampling.这个过程称为重要性采样。

**Monte Carlo integration and importance sampling****蒙特卡洛积分和重要性采样**

To fully get a grasp of importance sampling it's relevant we first delve into the mathematical construct known as Monte Carlo integration. 为了充分理解重要性抽样，我们首先要了解一种数学结构，称为蒙特卡洛积分。Monte Carlo integration revolves mostly around a combination of statistics and probability theory. 蒙特卡洛积分主要是统计和概率理论的组合。Monte Carlo helps us in discretely solving the problem of figuring out some statistic or value of a population without having to take **all** of the population into consideration.蒙特卡洛可以帮助我们离散地解决人口统计问题，而不必考虑所有人。

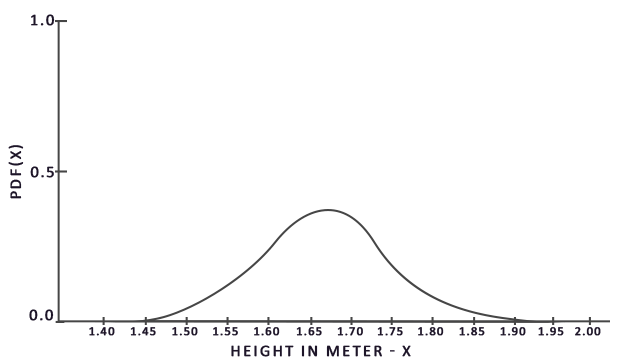
For instance, let's say you want to count the average height of all citizens of a country. 例如，假设您想要计算一个国家所有公民的平均身高。To get your result, you could measure **every** citizen and average their height which will give you the **exact** answer you're looking for. 为了得到你的结果，你可以测量每个公民并对他们的身高求平均，这样会得到你需要的确切答案。However, since most countries have a considerable population this isn't a realistic approach: it would take too much effort and time.但是，由于大多数国家人口众多，这不是一个现实的方法：需要花费太多精力和时间。

A different approach is to pick a much smaller **completely random** (unbiased) subset of this population, measure their height and average the result. 另一种方法是选择一个小得多的完全随机（无偏）的人口子集，测量他们的身高并对结果求平均。This population could be as small as a 100 people. 这部分人口数量可能只有100人。While not as accurate as the exact answer, you'll get an answer that is relatively close to the ground truth. 虽然答案并非非常精确，但会得到一个相对接近真相的答案。This is known as the law of large numbers. 这个方法被称作大数定律。The idea is that if you measure a smaller set of size NN of truly random samples from the total population, the result will be relatively close to the true answer and gets closer as the number of samples NN increases.我们的想法是，如果从总人口中测量一组较小的真正随机样本的\(N\)，结果将相对接近真实答案，并随着样本数\(N\)的增加而变得更接近。

Monte Carlo integration builds on this law of large numbers and takes the same approach in solving an integral. 蒙特卡罗积分建立在这个大数定律的基础上，并采用相同的方法来求解积分。Rather than solving an integral for all possible (theoretically infinite) sample values xx, simply generate NN sample values randomly picked from the total population and average. 不为所有可能的（理论上是无限的）样本值\(x\)求解积分，而是简单地从总体中随机挑选样本\(N\)生成采样值并求平均。As NN increases we're guaranteed to get a result closer to the exact answer of the integral:随着\(N\)的增加，我们的结果会越来越接近积分的精确结果：

O=∫abf(x)dx=1N∑i=0N−1f(x)pdf(x)O=∫abf(x)dx=1N∑i=0N−1f(x)pdf(x)

To solve the integral, we take NN random samples over the population aa to bb, add them together and divide by the total number of samples to average them. 为了求解这个积分，我们在\(a\)到\(b\)上采样\(N\)个随机样本，将它们加在一起并除以样本总数来取平均。The pdfpdf stands for the probability density function that tells us the probability a specific sample occurs over the total sample set. \(pdf\)代表概率密度函数 (probability density function)，它的含义是特定样本在整个样本集上发生的概率。For instance, the pdf of the height of a population would look a bit like this:例如，人口身高的pdf看起来应该像这样：



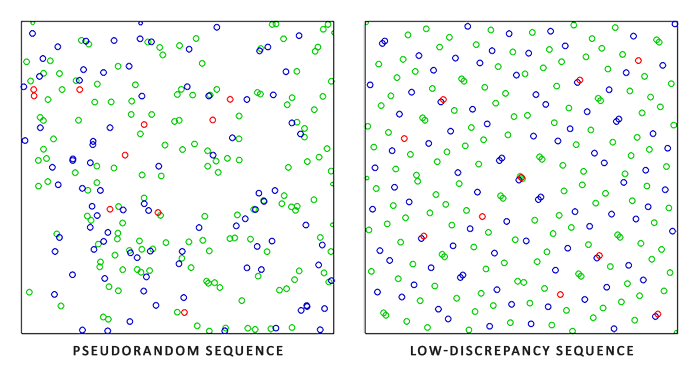
From this graph we can see that if we take any random sample of the population, there is a higher chance of picking a sample of someone of height 1.70, compared to the lower probability of the sample being of height 1.50.从该图中我们可以看出，如果我们对人口任意随机采样，那么挑选身高为1.70的人口样本的可能性更高，而样本身高为1.50的概率较低。

When it comes to Monte Carlo integration, some samples might have a higher probability of being generated than others. 当涉及蒙特卡洛积分时，某些样本可能比其他样本具有更高的生成概率。This is why for any general Monte Carlo estimation we divide or multiply the sampled value by the sample probability according to a pdf. 这就是为什么对于任何一般的蒙特卡洛估计，我们都会根据pdf将采样值除以或乘以采样概率。So far, in each of our cases of estimating an integral, the samples we've generated were uniform, having the exact same chance of being generated. 到目前为止，我们每次需要估算积分的时候，生成的样本都是均匀分布的，概率完全相等。Our estimations so far were unbiased, meaning that given an ever-increasing amount of samples we will eventually converge to the **exact** solution of the integral.到目前为止，我们的估计是无偏的，这意味着随着样本数量的不断增加，我们最终将收敛到积分的精确解。

However, some Monte Carlo estimators are biased, meaning that the generated samples aren't completely random, but focused towards a specific value or direction. 但是，某些蒙特卡洛估算是有偏的，这意味着生成的样本并不是完全随机的，而是集中于特定的值或方向。These biased Monte Carlo estimators have a faster rate of convergence meaning they can converge to the exact solution at a much faster rate, but due to their biased nature it's likely they won't ever converge to the exact solution. 这些有偏的蒙特卡洛估算具有更快的收敛速度，它们会以更快的速度收敛到精确解，但是由于其有偏性，可能永远不会收敛到精确解。This is generally an acceptable tradeoff, especially in computer graphics, as the exact solution isn't too important as long as the results are visually acceptable. 通常来说，这是一个可以接受的折衷方案，尤其是在计算机图形学中，因为只要结果在视觉上可以接受，解决方案的精确性就不太重要。As we'll soon see with importance sampling (which uses a biased estimator) the generated samples are biased towards specific directions in which case we account for this by multiplying or dividing each sample by its corresponding pdf.下文我们将会提到一种（有偏的）重要性采样，其生成的样本偏向特定的方向，在这种情况下，我们会将每个样本乘以或除以相应的 pdf 再求和。

Monte Carlo integration is quite prevalent in computer graphics as it's a fairly intuitive way to approximate continuous integrals in a discrete and efficient fashion: take any area/volume to sample over (like the hemisphere ΩΩ), generate NNamount of random samples within the area/volume and sum and weigh every sample contribution to the final result.蒙特卡洛积分在计算机图形学中非常普遍，因为它是一种以高效的离散方式对连续的积分求近似，而且非常直观的方法：对任何面积/体积进行采样——例如半球\(\Omega\)——在该面积/体积内生成数量\(N\)的随机采样，权衡每个样本对最终结果的贡献并求和。

Monte Carlo integration is an extensive mathematical topic and I won't delve much further into the specifics, but we'll mention that there are also multiple ways of generating the *random samples*. 蒙特卡洛积分是一个庞大的数学主题，在此不再赘述，但有一点需要提到：生成随机样本的方法也多种多样。By default, each sample is completely (pseudo)random as we're used to, but by utilizing certain properties of semi-random sequences we can generate sample vectors that are still random, but have interesting properties. 默认情况下，每次采样都是我们熟悉的完全（伪）随机，不过通过利用半随机序列的某些属性，我们可以生成虽然是随机样本但具有一些有趣性质的样本向量。For instance, we can do Monte Carlo integration on something called low-discrepancy sequences which still generate random samples, but each sample is more evenly distributed:例如，我们可以对一种名为低差异序列的东西进行蒙特卡洛积分，该序列生成的仍然是随机样本，但样本分布更均匀：



When using a low-discrepancy sequence for generating the Monte Carlo sample vectors, the process is known as Quasi-Monte Carlo integration. 当使用低差异序列生成蒙特卡洛样本向量时，该过程称为拟蒙特卡洛积分。Quasi-Monte Carlo methods have a faster rate of convergence which makes them interesting for performance heavy applications.拟蒙特卡洛方法具有更快的收敛速度，这使得它对于性能繁重的应用很有用。

Given our newly obtained knowledge of Monte Carlo and Quasi-Monte Carlo integration, there is an interesting property we can use for an even faster rate of convergence known as importance sampling. 鉴于我们新获得的有关蒙特卡洛（Monte Carlo）和拟蒙特卡洛（Quasi-Monte Carlo）积分的知识，我们可以使用一个有趣的属性来获得更快的收敛速度，这就是重要性采样。We've mentioned it before in this tutorial, but when it comes to specular reflections of light, the reflected light vectors are constrained in a specular lobe with its size determined by the roughness of the surface. 我们在前文已经提到过它，但是在高光反射的情况下，反射的光向量被限制在高光波瓣中，波瓣的大小取决于表面的粗糙度。Seeing as any (quasi-)randomly generated sample outside the specular lobe isn't relevant to the specular integral it makes sense to focus the sample generation to within the specular lobe, at the cost of making the Monte Carlo estimator biased.既然高光波瓣外的任何（拟）随机生成的样本与镜面积分无关，因此将样本集中在镜面波瓣内生成是有意义的，但代价是蒙特卡洛估算会产生偏差。

This is in essence what importance sampling is about: generate sample vectors in some region constrained by the roughness oriented around the microfacet's halfway vector. 本质上来说，这就是重要性采样的核心：只在某些区域生成采样向量，该区域围绕微表面半向量，受粗糙度限制。By combining Quasi-Monte Carlo sampling with a low-discrepancy sequence and biasing the sample vectors using importance sampling we get a high rate of convergence. 通过将拟蒙特卡洛采样与低差异序列相结合，并使用重要性采样偏置样本向量的方法，我们可以获得很高的收敛速度。Because we reach the solution at a faster rate, we'll need less samples to reach an approximation that is sufficient enough. 因为我们求解的速度更快，所以要达到足够的近似度，我们所需要的样本更少。Because of this, the combination even allows graphics applications to solve the specular integral in real-time, albeit it still significantly slower than pre-computing the results.因此，这套组合方法甚至可以允许图形应用程序实时求解高光积分，虽然比预计算结果还是要慢得多。

**A low-discrepancy sequence****低差异序列**

In this tutorial we'll pre-compute the specular portion of the indirect reflectance equation using importance sampling given a random low-discrepancy sequence based on the Quasi-Monte Carlo method. 在本教程中，我们将使用重要性采样来预计算间接反射方程的镜面反射部分，该采样基于拟蒙特卡洛方法给出了随机的低差异序列。The sequence we'll be using is known as the Hammersley Sequence as carefully described by [Holger Dammertz](http://holger.dammertz.org/stuff/notes_HammersleyOnHemisphere.html). 我们将使用的序列被称为 Hammersley 序列，Holger Dammertz 曾仔细描述过它。The Hammersley sequence is based on the Van Der Corpus sequence which mirrors a decimal binary representation around its decimal point.Hammersley 序列是基于 Van Der Corput 序列，该序列是把十进制数字的二进制表示镜像翻转到小数点右边而得。（译注：原文为Van Der Corpus疑似笔误，下文各处同）

Given some neat bit tricks we can quite efficiently generate the Van Der Corpus sequence in a shader program which we'll use to get a Hammersley sequence sample i over N total samples:给出一些巧妙的技巧，我们可以在着色器程序中非常有效地生成 Van Der Corput 序列，我们将用它来获得 Hammersley 序列，设总样本数为 N，样本索引为 i：

**float** RadicalInverse\_VdC(uint bits)

{

bits = (bits << 16u) | (bits >> 16u);

bits = ((bits & 0x55555555u) << 1u) | ((bits & 0xAAAAAAAAu) >> 1u);

bits = ((bits & 0x33333333u) << 2u) | ((bits & 0xCCCCCCCCu) >> 2u);

bits = ((bits & 0x0F0F0F0Fu) << 4u) | ((bits & 0xF0F0F0F0u) >> 4u);

bits = ((bits & 0x00FF00FFu) << 8u) | ((bits & 0xFF00FF00u) >> 8u);

**return** **float**(bits) \* 2.3283064365386963e-10; // / 0x100000000

}

// ----------------------------------------------------------------------------

vec2 Hammersley(uint i, uint N)

{

**return** vec2(**float**(i)/**float**(N), RadicalInverse\_VdC(i));

}

The GLSL Hammersley function gives us the low-discrepancy sample i of the total sample set of size N.GLSL 的 Hammersley 函数可以获取大小为 N 的样本集中的低差异样本 i。

**Hammersley sequence without bit operator support****无需位运算的 Hammersley 序列**

Not all OpenGL related drivers support bit operators (WebGL and OpenGL ES 2.0 for instance) in which case you might want to use an alternative version of the Van Der Corpus Sequence that doesn't rely on bit operators:并非所有 OpenGL 相关驱动程序都支持位运算符（例如WebGL和OpenGL ES 2.0），在这种情况下，您可能需要使用 Van Der Corput 序列不依赖位运算符的替代版本：

**float** VanDerCorpus(uint n, uint base)

{

**float** invBase = 1.0 / **float**(base);

**float** denom = 1.0;

**float** result = 0.0;

**for**(uint i = 0u; i < 32u; ++i)

{

**if**(n > 0u)

{

denom = mod(**float**(n), 2.0);

result += denom \* invBase;

invBase = invBase / 2.0;

n = uint(**float**(n) / 2.0);

}

}

**return** result;

}

// ----------------------------------------------------------------------------

vec2 HammersleyNoBitOps(uint i, uint N)

{

**return** vec2(**float**(i)/**float**(N), VanDerCorpus(i, 2u));

}

Note that due to GLSL loop restrictions in older hardware the sequence loops over all possible 32 bits. 请注意，由于旧硬件中的 GLSL 循环限制，该序列循环遍历了所有可能的32位。This version is less performant, but does work on all hardware if you ever find yourself without bit operators.该版本的性能略差，但是如果您没有位运算符可用，它可以在所有硬件上运行。

**GGX Importance sampling****GGX 重要性采样**

Instead of uniformly or randomly (Monte Carlo) generating sample vectors over the integral's hemisphere ΩΩ we'll generate sample vectors biased towards the general reflection orientation of the microsurface halfway vector based on the surface's roughness. 有别于均匀或纯随机地（比如蒙特卡洛）在积分半球\(\Omega\)产生采样向量，我们的采样会根据粗糙度，偏向微表面的半向量的宏观反射方向。The sampling process will be similar to what we've seen before: begin a large loop, generate a random (low-discrepancy) sequence value, take the sequence value to generate a sample vector in tangent space, transform to world space and sample the scene's radiance. 采样过程将与我们之前看到的过程相似：开始一个大循环，生成一个随机（低差异）序列值，用该序列值在切线空间中生成样本向量，将样本向量变换到世界空间并对场景的辐射度采样。What's different is that we now use a low-discrepancy sequence value as input to generate a sample vector:不同之处在于，我们现在使用低差异序列值作为输入来生成采样向量：

**const** uint SAMPLE\_COUNT = 4096u;

**for**(uint i = 0u; i < SAMPLE\_COUNT; ++i)

{

vec2 Xi = Hammersley(i, SAMPLE\_COUNT);

Additionally, to build a sample vector, we need some way of orienting and biasing the sample vector towards the specular lobe of some surface roughness. 此外，要构建采样向量，我们需要一些方法定向和偏移采样向量，以使其朝向特定粗糙度的镜面波瓣方向。We can take the NDF as described in the [Theory](https://learnopengl.com/#!PBR/Theory) tutorial and combine the GGX NDF in the spherical sample vector process as described by Epic Games:我们可以如理论教程中所述使用 NDF，并将 GGX NDF 结合到 Epic Games 所述的球体采样向量的处理中：

vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, **float** roughness)

{

**float** a = roughness\*roughness;

**float** phi = 2.0 \* PI \* Xi.x;

**float** cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a\*a - 1.0) \* Xi.y));

**float** sinTheta = sqrt(1.0 - cosTheta\*cosTheta);

// from spherical coordinates to cartesian coordinates

vec3 H;

H.x = cos(phi) \* sinTheta;

H.y = sin(phi) \* sinTheta;

H.z = cosTheta;

// from tangent-space vector to world-space sample vector

vec3 up = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);

vec3 tangent = normalize(cross(up, N));

vec3 bitangent = cross(N, tangent);

vec3 sampleVec = tangent \* H.x + bitangent \* H.y + N \* H.z;

**return** normalize(sampleVec);

}

This gives us a sample vector somewhat oriented around the expected microsurface's halfway vector based on some input roughness and the low-discrepancy sequence value Xi. 基于特定的粗糙度输入和低差异序列值 Xi，我们获得了一个采样向量，该向量大体围绕着预估的微表面的半向量。Note that Epic Games uses the squared roughness for better visual results as based on Disney's original PBR research.注意，根据迪士尼最开始对 PBR 的研究，Epic Games 使用了平方粗糙度以获得更好的视觉效果。

With the low-discrepancy Hammersley sequence and sample generation defined we can finalize the pre-filter convolution shader:使用低差异 Hammersley 序列和上述定义的样本生成方法，我们可以最终完成预滤波器卷积着色器：

#version 330 core

**out** vec4 FragColor;

**in** vec3 localPos;

**uniform** samplerCube environmentMap;

**uniform** **float** roughness;

**const** **float** PI = 3.14159265359;

**float** RadicalInverse\_VdC(uint bits);

vec2 Hammersley(uint i, uint N);

vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, **float** roughness);

**void** main()

{

vec3 N = normalize(localPos);

vec3 R = N;

vec3 V = R;

**const** uint SAMPLE\_COUNT = 1024u;

**float** totalWeight = 0.0;

vec3 prefilteredColor = vec3(0.0);

**for**(uint i = 0u; i < SAMPLE\_COUNT; ++i)

{

vec2 Xi = Hammersley(i, SAMPLE\_COUNT);

vec3 H = ImportanceSampleGGX(Xi, N, roughness);

vec3 L = normalize(2.0 \* dot(V, H) \* H - V);

**float** NdotL = max(dot(N, L), 0.0);

**if**(NdotL > 0.0)

{

prefilteredColor += texture(environmentMap, L).rgb \* NdotL;

totalWeight += NdotL;

}

}

prefilteredColor = prefilteredColor / totalWeight;

FragColor = vec4(prefilteredColor, 1.0);

}

We pre-filter the environment, based on some input roughness that varies over each mipmap level of the pre-filter cubemap (from 0.0 to 1.0) and store the result in prefilteredColor. 输入的粗糙度随着预过滤的立方体贴图的 mipmap 级别变化（从0.0到1.0），我们根据据粗糙度预过滤环境贴图，把结果存在 prefilteredColor 里。The resulting prefilteredColor is divided by the total sample weight, where samples with less influence on the final result (for small NdotL) contribute less to the final weight.所得的 prefilteredColor 除以采样权重总和，其中对最终结果影响较小的采样（NdotL 较小）对最终权重的贡献也较小。

**Capturing pre-filter mipmap levels****捕获预过滤 Mipmap 级别**

What's left to do is let OpenGL pre-filter the environment map with different roughness values over multiple mipmap levels. 剩下要做的就是让 OpenGL 在多个 mipmap 级别上以不同的粗糙度值预过滤环境贴图。This is actually fairly easy to do with the original setup of the [irradiance](https://learnopengl.com/#!PBR/IBL/Diffuse-irradiance) tutorial:有了最开始的辐照度教程作为基础，实际上很简单：

prefilterShader.use();

prefilterShader.setInt("environmentMap", 0);

prefilterShader.setMat4("projection", captureProjection);

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, envCubemap);

glBindFramebuffer(GL\_FRAMEBUFFER, captureFBO);

**unsigned** **int** maxMipLevels = 5;

**for** (**unsigned** **int** mip = 0; mip < maxMipLevels; ++mip)

{

// reisze framebuffer according to mip-level size.

**unsigned** **int** mipWidth = 128 \* std::pow(0.5, mip);

**unsigned** **int** mipHeight = 128 \* std::pow(0.5, mip);

glBindRenderbuffer(GL\_RENDERBUFFER, captureRBO);

glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH\_COMPONENT24, mipWidth, mipHeight);

glViewport(0, 0, mipWidth, mipHeight);

**float** roughness = (**float**)mip / (**float**)(maxMipLevels - 1);

prefilterShader.setFloat("roughness", roughness);

**for** (**unsigned** **int** i = 0; i < 6; ++i)

{

prefilterShader.setMat4("view", captureViews[i]);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0,

GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, prefilterMap, mip);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

renderCube();

}

}

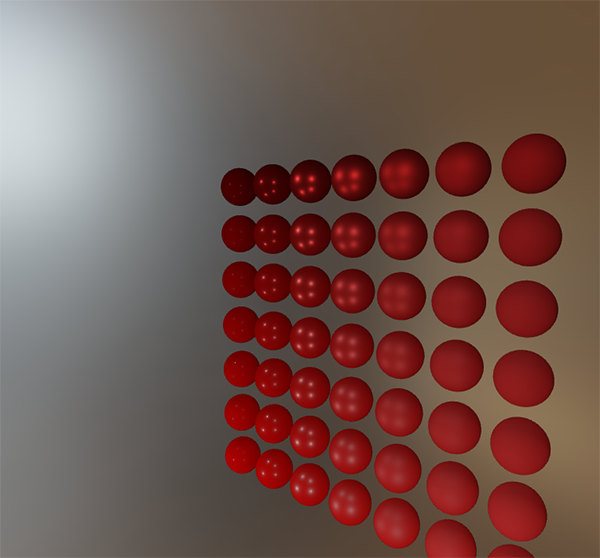
glBindFramebuffer(GL\_FRAMEBUFFER, 0);

The process is similar to the irradiance map convolution, but this time we scale the framebuffer's dimensions to the appropriate mipmap scale, each mip level reducing the dimensions by 2. 这个过程类似于辐照度贴图卷积，但是这次我们将帧缓冲区缩放到适当的 mipmap 尺寸， mip 级别每增加一级，尺寸缩小为一半。Additionally, we specify the mip level we're rendering into in glFramebufferTexture2D's last parameter and pass the roughness we're pre-filtering for to the pre-filter shader.此外，我们在 glFramebufferTexture2D 的最后一个参数中指定要渲染到的 mip 级别，然后将要预过滤的粗糙度传给预过滤着色器。

This should give us a properly pre-filtered environment map that returns blurrier reflections the higher mip level we access it from. 这样我们会得到一张经过适当预过滤的环境贴图，当我们访问该贴图时指定的 mip 等级越高，它返回的反射就越模糊。If we display the pre-filtered environment cubemap in the skybox shader and forecefully sample somewhat above its first mip level in its shader like so:如果我们在天空盒着色器中显示这张预过滤的环境立方体贴图，并在其着色器中强制在其第一个 mip 级别以上采样，如下所示：

vec3 envColor = textureLod(environmentMap, WorldPos, 1.2).rgb;

We get a result that indeed looks like a blurrier version of the original environment:我们得到的结果确实看起来像原始环境的模糊版本：



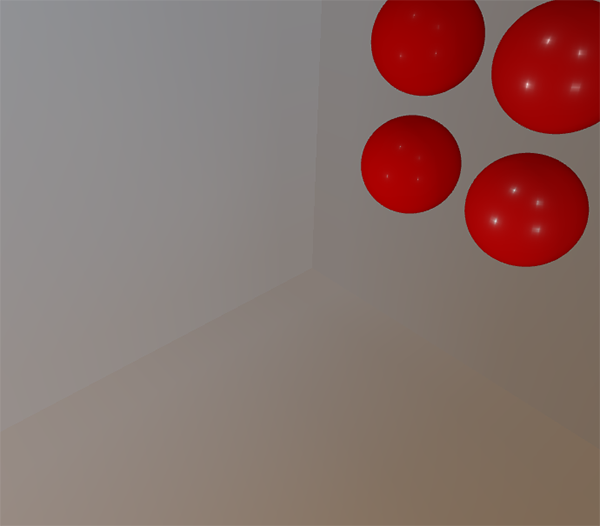
If it looks somewhat similar you've successfully pre-filtered the HDR environment map. 如果 HDR 环境贴图的预过滤看起来差不多成功了，Play around with different mipmap levels to see the pre-filter map gradually change from sharp to blurry reflections on increasing mip levels.尝试一下不同的 mipmap 级别，观察预过滤贴图随着 mip 级别增加，从锐利反射逐渐变为模糊反射的过程。

**Pre-filter convolution artifacts****预过滤卷积的伪像**

While the current pre-filter map works fine for most purposes, sooner or later you'll come across several render artifacts that are directly related to the pre-filter convolution. 当前的预过滤贴图可以在大多数情况下正常工作，不过你迟早会遇到几个与预过滤卷积直接相关的渲染问题。I'll list the most common here including how to fix them.我将在这里列出最常见的一些问题，以及如何修复它们。

**Cubemap seams at high roughness****高粗糙度的立方体贴图接缝**

Sampling the pre-filter map on surfaces with a rough surface means sampling the pre-filter map on some of its lower mip levels. 在具有粗糙表面的表面上对预过滤贴图采样，也就等同于在较低的 mip 级别上对预过滤贴图采样。When sampling cubemaps, OpenGL by default doesn't linearly interpolate **across** cubemap faces. 在对立方体贴图进行采样时，默认情况下，OpenGL不会对整个立方体贴图的面进行线性插值。Because the lower mip levels are both of a lower resolution and the pre-filter map is convoluted with a much larger sample lobe, the lack of *between-cube-face filtering* becomes quite apparent:由于较低的 mip 级别具有更低的分辨率，并且预过滤贴图代表了与更大的采样波瓣卷积，因此立方体的面和面之间缺乏滤波的问题就更明显：



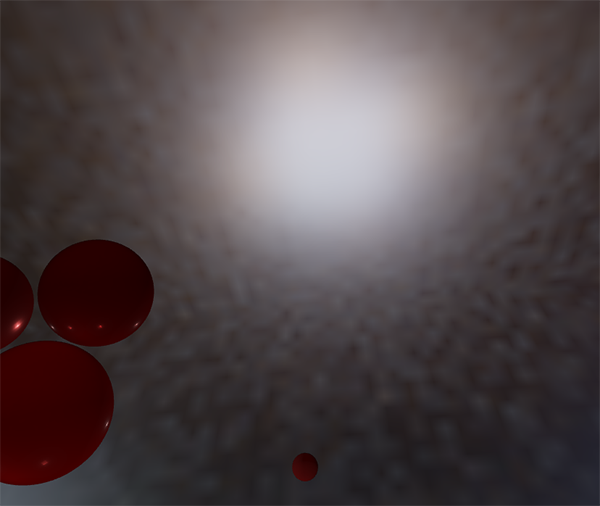
Luckily for us, OpenGL gives us the option to properly filter across cubemap faces by enabling GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS:幸运的是，OpenGL 可以启用 GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS，以为我们提供在立方体贴图的面之间进行正确过滤的选项：

glEnable(GL\_TEXTURE\_CUBE\_MAP\_SEAMLESS);

Simply enable this property somewhere at the start of your application and the seams will be gone.只需在应用程序开始的某个地方启用此属性，接缝就会消失。

**Bright dots in the pre-filter convolution****预过滤卷积的亮点**

Due to high frequency details and wildly varying light intensities in specular reflections, convoluting the specular reflections requires a large number of samples to properly account for the wildly varying nature of HDR environmental reflections. 由于镜面反射中光强度的变化大，高频细节多，所以对镜面反射进行卷积需要大量采样，才能正确反映 HDR 环境反射的混乱变化。We already take a very large number of samples, but on some environments it might still not be enough at some of the rougher mip levels in which case you'll start seeing dotted patterns emerge around bright areas:我们已经进行了大量的采样，但是在某些环境下，在某些较粗糙的 mip 级别上可能仍然不够，导致明亮区域周围出现点状图案：



One option is to further increase the sample count, but this won't be enough for all environments. 一种解决方案是进一步增加样本数量，但在某些情况下还是不够。As described by [Chetan Jags](https://chetanjags.wordpress.com/2015/08/26/image-based-lighting/) we can reduce this artifact by (during the pre-filter convolution) not directly sampling the environment map, but sampling a mip level of the environment map based on the integral's PDF and the roughness:另一种方案如 Chetan Jags 所述，我们可以在预过滤卷积时，不直接采样环境贴图，而是基于积分的 PDF 和粗糙度采样环境贴图的 mipmap ，以减少伪像：

**float** D = DistributionGGX(NdotH, roughness);

**float** pdf = (D \* NdotH / (4.0 \* HdotV)) + 0.0001;

**float** resolution = 512.0; // resolution of source cubemap (per face)

**float** saTexel = 4.0 \* PI / (6.0 \* resolution \* resolution);

**float** saSample = 1.0 / (**float**(SAMPLE\_COUNT) \* pdf + 0.0001);

**float** mipLevel = roughness == 0.0 ? 0.0 : 0.5 \* log2(saSample / saTexel);

Don't forget to enable trilinear filtering on the environment map you want to sample its mip levels from:既然要采样 mipmap ，不要忘记在环境贴图上开启三线性过滤：

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, envCubemap);

glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR);

And let OpenGL generate the mipmaps **after** the cubemap's base texture is set:设置立方体贴图的基本纹理后，让 OpenGL 生成 mipmap：

// convert HDR equirectangular environment map to cubemap equivalent

[...]

// then generate mipmaps// 然后生成 mipmap

glBindTexture(GL\_TEXTURE\_CUBE\_MAP, envCubemap);

glGenerateMipmap(GL\_TEXTURE\_CUBE\_MAP);

This works surprisingly well and should remove most, if not all, dots in your pre-filter map on rougher surfaces.这个方法效果非常好，可以去除预过滤贴图中较粗糙表面上的大多数甚至全部亮点。

**Pre-computing the BRDF****预计算 BRDF**

With the pre-filtered environment up and running, we can focus on the second part of the split-sum approximation: the BRDF. 预过滤的环境贴图已经可以设置并运行，我们可以集中精力于求和近似的第二部分：BRDF。Let's briefly review the specular split sum approximation again:让我们再次简要回顾一下高光的分割求和近似法：

Lo(p,ωo)=∫ΩLi(p,ωi)dωi∗∫Ωfr(p,ωi,ωo)n⋅ωidωiLo(p,ωo)=∫ΩLi(p,ωi)dωi∗∫Ωfr(p,ωi,ωo)n⋅ωidωi

We've pre-computed the left part of the split sum approximation in the pre-filter map over different roughness levels. 我们已经在预过滤贴图的各个粗糙度级别上预计算了分割求和近似的左半部分。The right side requires us to convolute the BRDF equation over the angle n⋅ωon⋅ωo, the surface roughness and Fresnel's F0F0. This is similar to integrating the specular BRDF with a solid-white environment or a constant radiance LiLi of 1.0. Convoluting the BRDF over 3 variables is a bit much, but we can move F0F0 out of the specular BRDF equation:右半部分要求我们在 \(n \dot \Omega\_o\) 、表面粗糙度、菲涅尔系数\(F0\)上计算 BRDF 方程的卷积。这等同于在纯白的环境光或者辐射度恒定为\(L\_i\)=1.0的设置下，对高光 BRDF 求积分。对3个变量做卷积有点复杂，不过我们可以把\(F\_0\)移出高光 BRDF 方程：

∫Ωfr(p,ωi,ωo)n⋅ωidωi=∫Ωfr(p,ωi,ωo)F(ωo,h)F(ωo,h)n⋅ωidωi∫Ωfr(p,ωi,ωo)n⋅ωidωi=∫Ωfr(p,ωi,ωo)F(ωo,h)F(ωo,h)n⋅ωidωi

With FF being the Fresnel equation. \(F\)为菲涅耳方程。Moving the Fresnel denominator to the BRDF gives us the following equivalent equation:将菲涅耳分母移到 BRDF 下面可以得到如下等式：

∫Ωfr(p,ωi,ωo)F(ωo,h)F(ωo,h)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)F(ωo,h)n⋅ωidωi

Substituting the right-most FF with the Fresnel-Schlick approximation gives us:用 Fresnel-Schlick 近似公式替换右边的 \(F\) 可以得到：

∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+(1−F0)(1−ωo⋅h)5)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+(1−F0)(1−ωo⋅h)5)n⋅ωidωi

Let's replace (1−ωo⋅h)5(1−ωo⋅h)5 by αα to make it easier to solve for F0F0:让我们用\(\alpha\)替换\({(1 - \omega\_o \cdot h)}^5\)以便更轻松地求解\(F\_0\)：

∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+(1−F0)α)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+(1−F0)α)n⋅ωidωi

∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+1∗α−F0∗α)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)(F0+1∗α−F0∗α)n⋅ωidωi

∫Ωfr(p,ωi,ωo)F(ωo,h)(F0∗(1−α)+α)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)(F0∗(1−α)+α)n⋅ωidωi

Then we split the Fresnel function FF over two integrals:然后我们将菲涅耳函数\(F\)分拆到两个积分里：

∫Ωfr(p,ωi,ωo)F(ωo,h)(F0∗(1−α))n⋅ωidωi+∫Ωfr(p,ωi,ωo)F(ωo,h)(α)n⋅ωidωi∫Ωfr(p,ωi,ωo)F(ωo,h)(F0∗(1−α))n⋅ωidωi+∫Ωfr(p,ωi,ωo)F(ωo,h)(α)n⋅ωidωi

This way, F0F0 is constant over the integral and we can take F0F0 out of the integral. 这样，\(F\_0\)在整个积分上是恒定的，我们可以从积分中提取出\(F\_0\)。Next, we substitute αα back to its original form giving us the final split sum BRDF equation:接下来，我们将\(\alpha\)替换回其原始形式，从而得到最终分割求和的 BRDF 方程：

F0∫Ωfr(p,ωi,ωo)(1−(1−ωo⋅h)5)n⋅ωidωi+∫Ωfr(p,ωi,ωo)(1−ωo⋅h)5n⋅ωidωiF0∫Ωfr(p,ωi,ωo)(1−(1−ωo⋅h)5)n⋅ωidωi+∫Ωfr(p,ωi,ωo)(1−ωo⋅h)5n⋅ωidωi

The two resulting integrals represent a scale and a bias to F0F0 respectively. 公式中的两个积分分别表示\(F\_0\)的比例和偏差。Note that as f(p,ωi,ωo)f(p,ωi,ωo) already contains a term for FF they both cancel out, removing FF from ff.注意，由于\(f(p, \omega\_i, \omega\_o)\)已经包含\(F\)项，它们被约分了，这里的\(f\)中不计算\(F\)项。

In a similar fashion to the earlier convoluted environment maps, we can convolute the BRDF equations on their inputs: the angle between nn and ωoωo and the roughness, and store the convoluted result in a texture. 和之前卷积环境贴图类似，我们可以对 BRDF 方程求卷积，其输入是\(n\)和\(\omega\_o\)的夹角，以及粗糙度，并将卷积的结果存储在纹理中。We store the convoluted results in a 2D lookup texture (LUT) known as a BRDF integration map that we later use in our PBR lighting shader to get the final convoluted indirect specular result.我们将卷积后的结果存储在 2D 查找纹理（Look Up Texture, LUT）中，这张纹理被称为 BRDF 积分贴图，稍后会将其用于 PBR 光照着色器中，以获得间接镜面反射的最终卷积结果。

The BRDF convolution shader operates on a 2D plane, using its 2D texture coordinates directly as inputs to the BRDF convolution (NdotV and roughness). BRDF 卷积着色器在 2D 平面上执行计算，直接使用其 2D 纹理坐标作为卷积输入（N·V 和 粗糙度）。The convolution code is largely similar to the pre-filter convolution, except that it now processes the sample vector according to our BRDF's geometry function and Fresnel-Schlick's approximation:卷积代码与预滤波器的卷积代码大体相似，不同之处在于，它现在根据 BRDF 的几何函数和 Fresnel-Schlick 近似来处理采样向量：

vec2 IntegrateBRDF(**float** NdotV, **float** roughness)

{

vec3 V;

V.x = sqrt(1.0 - NdotV\*NdotV);

V.y = 0.0;

V.z = NdotV;

**float** A = 0.0;

**float** B = 0.0;

vec3 N = vec3(0.0, 0.0, 1.0);

**const** uint SAMPLE\_COUNT = 1024u;

**for**(uint i = 0u; i < SAMPLE\_COUNT; ++i)

{

vec2 Xi = Hammersley(i, SAMPLE\_COUNT);

vec3 H = ImportanceSampleGGX(Xi, N, roughness);

vec3 L = normalize(2.0 \* dot(V, H) \* H - V);

**float** NdotL = max(L.z, 0.0);

**float** NdotH = max(H.z, 0.0);

**float** VdotH = max(dot(V, H), 0.0);

**if**(NdotL > 0.0)

{

**float** G = GeometrySmith(N, V, L, roughness);

**float** G\_Vis = (G \* VdotH) / (NdotH \* NdotV);

**float** Fc = pow(1.0 - VdotH, 5.0);

A += (1.0 - Fc) \* G\_Vis;

B += Fc \* G\_Vis;

}

}

A /= **float**(SAMPLE\_COUNT);

B /= **float**(SAMPLE\_COUNT);

**return** vec2(A, B);

}

// ----------------------------------------------------------------------------

**void** main()

{

vec2 integratedBRDF = IntegrateBRDF(TexCoords.x, TexCoords.y);

FragColor = integratedBRDF;

}

As you can see the BRDF convolution is a direct translation from the mathematics to code. 如你所见，BRDF 卷积部分是从数学到代码的直接转换。We take both the angle θθ and the roughness as input, generate a sample vector with importance sampling, process it over the geometry and the derived Fresnel term of the BRDF, and output both a scale and a bias to F0F0 for each sample, averaging them in the end.我们将角度\(\theta\)和粗糙度作为输入，以重要性采样产生采样向量，在整个几何体上结合 BRDF 的菲涅耳项对向量进行处理，然后输出每个样本上\(F\_0\)的比例和偏差，最后取平均值。

You might've recalled from the [theory](https://learnopengl.com/#!PBR/Theory) tutorial that the geometry term of the BRDF is slightly different when used alongside IBL as its kk variable has a slightly different interpretation:你可能回想起理论教程中的一个细节：与 IBL 一起使用时，BRDF 的几何项略有不同，因为对其\(k\)变量的解释稍有不同：

kdirect=(α+1)28kdirect=(α+1)28k\_{direct} = \frac{(\alpha + 1)^2}{8}

kIBL=α22kIBL=α22k\_{IBL} = \frac{\alpha^2}{2}

Since the BRDF convolution is part of the specular IBL integral we'll use kIBLkIBL for the Schlick-GGX geometry function:由于 BRDF 卷积是高光 IBL 积分的一部分，因此我们要在 Schlick-GGX 几何函数中使用 \(k\_{IBL}\)：

**float** GeometrySchlickGGX(**float** NdotV, **float** roughness)

{

**float** a = roughness;

**float** k = (a \* a) / 2.0;

**float** nom = NdotV;

**float** denom = NdotV \* (1.0 - k) + k;

**return** nom / denom;

}

// ----------------------------------------------------------------------------

**float** GeometrySmith(vec3 N, vec3 V, vec3 L, **float** roughness)

{

**float** NdotV = max(dot(N, V), 0.0);

**float** NdotL = max(dot(N, L), 0.0);

**float** ggx2 = GeometrySchlickGGX(NdotV, roughness);

**float** ggx1 = GeometrySchlickGGX(NdotL, roughness);

**return** ggx1 \* ggx2;

}

Note that while kk takes a as its parameter we didn't square roughness as a as we originally did for other interpretations of a; likely as a is squared here already. 请注意，虽然 \(k\) 还是从 a 计算出来的，但这里的 a 不是粗糙度的平方——如同最初对 a 的其他解释那样——在这里假装平方过了。I'm not sure whether this is an inconsistency on Epic Games' part or the original Disney paper, but directly translating roughness to a gives the BRDF integration map that is identical to Epic Games' version.我不确定这样处理是否与 Epic Games 或迪士尼原始论文不一致，但是直接将粗糙度赋给 a 得到的 BRDF 积分贴图与 Epic Games 的版本完全一致。

Finally, to store the BRDF convolution result we'll generate a 2D texture of a 512 by 512 resolution.最后，为了存储 BRDF 卷积结果，我们需要生成一张 512 × 512 分辨率的 2D 纹理。

**unsigned** **int** brdfLUTTexture;

glGenTextures(1, &brdfLUTTexture);

// pre-allocate enough memory for the LUT texture.

glBindTexture(GL\_TEXTURE\_2D, brdfLUTTexture);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RG16F, 512, 512, 0, GL\_RG, GL\_FLOAT, 0);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

Note that we use a 16-bit precision floating format as recommended by Epic Games. 请注意，我们使用的是 Epic Games 推荐的16位精度浮点格式。Be sure to set the wrapping mode to GL\_CLAMP\_TO\_EDGE to prevent edge sampling artifacts.确保将环绕模式设置为 GL\_CLAMP\_TO\_EDGE 以防止边缘采样的伪像。

Then, we re-use the same framebuffer object and run this shader over an NDC screen-space quad:然后，我们复用同一个帧缓冲区对象，并在 NDC [译注：Normalized Device Coordinates] 屏幕空间四边形上运行此着色器：

glBindFramebuffer(GL\_FRAMEBUFFER, captureFBO);

glBindRenderbuffer(GL\_RENDERBUFFER, captureRBO);

glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH\_COMPONENT24, 512, 512);

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, brdfLUTTexture, 0);

glViewport(0, 0, 512, 512);

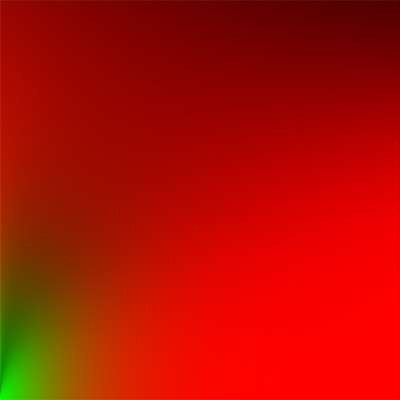
brdfShader.use();

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

RenderQuad();

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

The convoluted BRDF part of the split sum integral should give you the following result:分裂积分和的 BRDF 卷积部分应该得到以下结果：



With both the pre-filtered environment map and the BRDF 2D LUT we can re-construct the indirect specular integral according to the split sum approximation. 预过滤的环境贴图和 BRDF 的 2D LUT 都已经齐备，我们可以根据分割求和近似法重建间接高光部分积分了。The combined result then acts as the indirect or ambient specular light.最后合并的结果将被用作间接高光或环境高光。

**Completing the IBL reflectance****完成 IBL 反射**

To get the indirect specular part of the reflectance equation up and running we need to stitch both parts of the split sum approximation together. 为了使反射方程的间接高光部分正确运行，我们需要将分割求和近似法的两个部分缝合在一起。Let's start by adding the pre-computed lighting data to the top of our PBR shader:第一步是将预计算的光照数据声明到 PBR 着色器的最上面：

**uniform** samplerCube prefilterMap;

**uniform** sampler2D brdfLUT;

First, we get the indirect specular reflections of the surface by sampling the pre-filtered environment map using the reflection vector. 首先，使用反射向量采样预过滤的环境贴图，获取表面的间接高光反射。Note that we sample the appropriate mip level based on the surface roughness, giving rougher surfaces *blurrier*specular reflections.请注意，我们会根据表面粗糙度在合适的 mip 级别采样，以使更粗糙的表面产生更模糊的镜面反射。

**void** main()

{

[...]

vec3 R = reflect(-V, N);

**const** **float** MAX\_REFLECTION\_LOD = 4.0;

vec3 prefilteredColor = textureLod(prefilterMap, R, roughness \* MAX\_REFLECTION\_LOD).rgb;

[...]

}

In the pre-filter step we only convoluted the environment map up to a maximum of 5 mip levels (0 to 4), which we denote here as MAX\_REFLECTION\_LOD to ensure we don't sample a mip level where there's no (relevant) data.在预过滤步骤中，我们仅将环境贴图卷积最多 5 个 mip 级别（0到4），此处记为 MAX\_REFLECTION\_LOD，以确保不会对一个没有数据的 mip 级别采样。

Then we sample from the BRDF lookup texture given the material's roughness and the angle between the normal and view vector:然后我们用已知的材质粗糙度和视线-法线夹角作为输入，采样 BRDF LUT。

vec3 F = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

vec2 envBRDF = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;

vec3 specular = prefilteredColor \* (F \* envBRDF.x + envBRDF.y);

Given the scale and bias to F0F0 (here we're directly using the indirect Fresnel result F) from the BRDF lookup texture we combine this with the left pre-filter portion of the IBL reflectance equation and re-construct the approximated integral result as specular.这样我们就从 BRDF LUT 中获得了 \(F\_0\)的比例和偏移系数，这里我们就直接用间接光菲涅尔项 F 。把这个结果和 IBL 反射方程左边的预过滤部分结合起来，以重建整个近似积分作为高光。

This gives us the indirect specular part of the reflectance equation. 于是我们得到了反射方程的间接高光部分。Now, combine this with the diffuse part of the reflectance equation from the [last](https://learnopengl.com/#!PBR/IBL/Diffuse-irradiance) tutorial and we get the full PBR IBL result:现在，将其与上一节教程中的反射方程的漫反射部分结合起来，我们可以获得完整的 PBR IBL 结果：

vec3 F = FresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

vec3 kS = F;

vec3 kD = 1.0 - kS;

kD \*= 1.0 - metallic;

vec3 irradiance = texture(irradianceMap, N).rgb;

vec3 diffuse = irradiance \* albedo;

**const** **float** MAX\_REFLECTION\_LOD = 4.0;

vec3 prefilteredColor = textureLod(prefilterMap, R, roughness \* MAX\_REFLECTION\_LOD).rgb;

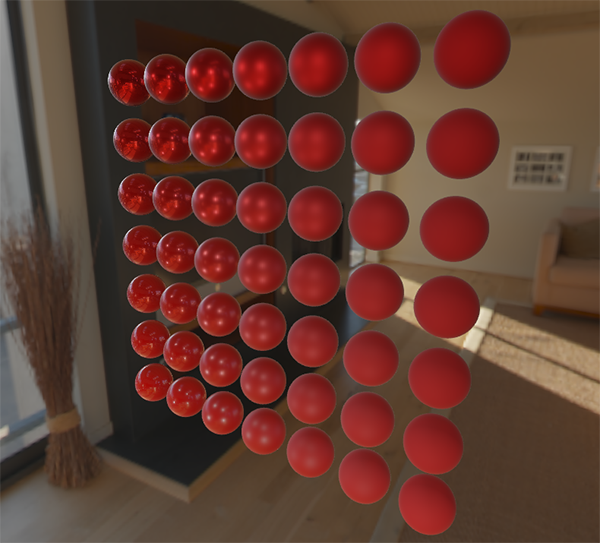
vec2 envBRDF = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;

vec3 specular = prefilteredColor \* (F \* envBRDF.x + envBRDF.y);

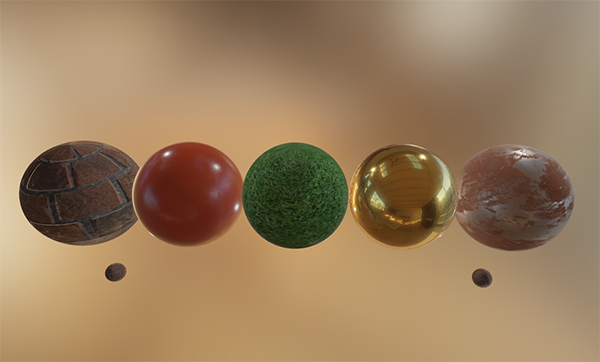
vec3 ambient = (kD \* diffuse + specular) \* ao;

Note that we don't multiply specular by kS as we already have a Fresnel multiplication in there.请注意，我们没有将高光反射乘以 kS，因为我们已经乘了菲涅耳系数。

Now, running this exact code on the series of spheres that differ by their roughness and metallic properties we finally get to see their true colors in the final PBR renderer:现在，在一系列粗糙度和金属度各异的球上运行此代码，我们终于可以在最终的 PBR 渲染器中看到其真实颜色：



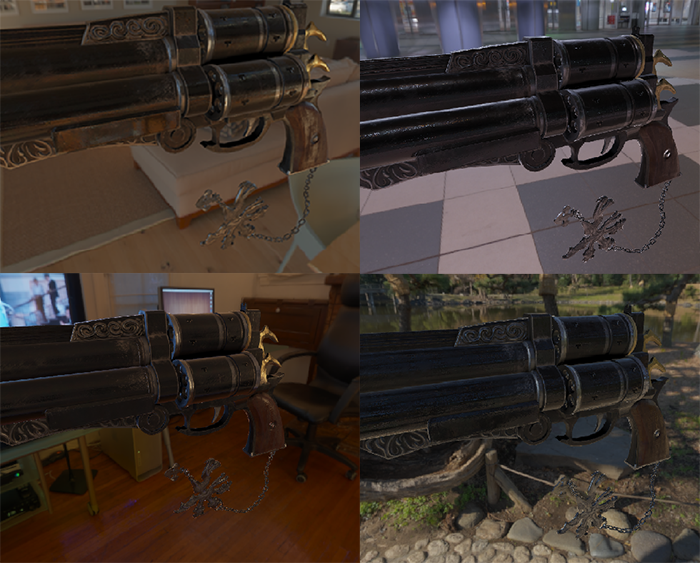
We could even go wild, and use some cool textured [PBR materials](http://freepbr.com/):我们甚至可以再疯狂一点，使用一些带酷炫纹理的 PBR 材质：



Or load [this awesome free PBR 3D model](http://artisaverb.info/PBT.html) by Andrew Maximov:或加载 Andrew Maximov 的这款出色的免费 PBR 3D 模型：



I'm sure we can all agree that our lighting now looks a lot more convincing. 我敢肯定我们都同意现在的光照看起来更具说服力。What's even better, is that our lighting looks physically correct, regardless of which environment map we use. 更妙的是，无论我们使用哪种环境贴图，我们的光照看起来都是物理正确的。Below you'll see several different pre-computed HDR maps, completely changing the lighting dynamics, but still looking physically correct without changing a single lighting variable!下面，您将看到几张不同的预计算 HDR 贴图，它们完全改变了光照动态，但是不需要调整任何光照变量，在外观上依然正确！



Well, this PBR adventure turned out to be quite a long journey. 好吧，这是一场长长的 PBR 冒险。There are a lot of steps and thus a lot that could go wrong so carefully work your way through the [sphere scene](https://learnopengl.com/code_viewer_gh.php?code=src/6.pbr/2.2.1.ibl_specular/ibl_specular.cpp) or [textured scene](https://learnopengl.com/code_viewer_gh.php?code=src/6.pbr/2.2.2.ibl_specular_textured/ibl_specular_textured.cpp) code samples (including all shaders) if you're stuck, or check and ask around in the comments.有很多步骤可能会出错，所以如果遇到问题卡住，请仔细研究球形场景或带纹理的场景代码示例——也包括所有着色器，或者检查之后在评论中提问。

**What's next?****下一步是？**

Hopefully, by the end of this tutorial you should have a pretty clear understanding of what PBR is about, and even have an actual PBR renderer up and running. 希望在本教程结束时，你会对 PBR 的相关内容有一个清晰的了解，甚至可以构造并运行一个实际的 PBR 渲染器。In these tutorials, we've pre-computed all the relevant PBR image-based lighting data at the start of our application, before the render loop. 在这几节教程中，我们已经在应用程序开始阶段，渲染循环之前，预计算了所有 PBR 相关的基于图像的光照数据。This was fine for educational purposes, but not too great for any practical use of PBR. 出于教育目的，这很好，但对于任何 PBR 的实践应用来说，都不是很漂亮。First, the pre-computation only really has to be done once, not at every startup. 首先，预计算实际上只需要执行一次，而不是每次启动时都要做。And second, the moment you use multiple environment maps you'll have to pre-compute each and every one of them at every startup which tends to build up.其次，当使用多个环境贴图时，你必须在每次程序启动时全部预计算一遍，这是个必须步骤。

For this reason you'd generally pre-compute an environment map into an irradiance and pre-filter map just once, and then store it on disk (note that the BRDF integration map isn't dependent on an environment map so you only need to calculate or load it once). 因此，通常只需要一次将环境贴图预计算为辐照度贴图和预过滤贴图，然后将其存储在磁盘上（注意，BRDF 积分贴图不依赖于环境贴图，因此只需要计算或加载一次）。This does mean you'll need to come up with a custom image format to store HDR cubemaps, including their mip levels. 这意味着您需要提出一种自定义图像格式来存储 HDR 立方体贴图，包括其 mip 级别。Or, you'll store (and load) it as one of the available formats (like .dds that supports storing mip levels).或者将图像存储为某种可用格式——例如支持存储 mip 级别的 .dds——并按其格式加载。

Furthermore, we've described the **total** process in these tutorials, including generating the pre-computed IBL images to help further our understanding of the PBR pipeline. 此外，我们也在教程中描述了整个过程，包括生成预计算的 IBL 图像，以帮助我们进一步了解 PBR 管线。But, you'll be just as fine by using several great tools like [cmftStudio](https://github.com/dariomanesku/cmftStudio" \t "_blank) or [IBLBaker](https://github.com/derkreature/IBLBaker" \t "_blank) to generate these pre-computed maps for you.此外还可以通过 cmftStudio 或 IBLBaker 等一些出色的工具为您生成这些预计算贴图，也很好用。

One point we've skipped over is pre-computed cubemaps as reflection probes: cubemap interpolation and parallax correction. 我们跳过的一点内容是如何将预计算的立方体贴图作为反射探针：立方体贴图插值和视差校正。This is the process of placing several reflection probes in your scene that take a cubemap snapshot of the scene at that specific location, which we can then convolute as IBL data for that part of the scene. 这是一个在场景中放置多个反射探针的过程，这些探针在特定位置拍摄场景的立方体贴图快照，然后我们可以将其卷积，作为相应部分场景的 IBL 数据。By interpolating between several of these probes based on the camera's vicinity we can achieve local high-detail image-based lighting that is simply limited by the amount of reflection probes we're willing to place. 基于相机的位置对附近的探针插值，我们可以实现局部的细节丰富的 IBL，受到的唯一限制就是探针放置的数量。This way, the image-based lighting could correctly update when moving from a bright outdoor section of a scene to a darker indoor section for instance. 这样一来，例如从一个明亮的室外部分移动到较暗的室内部分时，IBL 就能正确更新。I'll write a tutorial about reflection probes somewhere in the future, but for now I recommend the article by Chetan Jags below to give you a head start.我将来会在某个地方编写有关反射探针的教程，但现在，我建议使用下面的 Chetan Jags 撰写的文章来作为入门。

**Further reading**

* [Real Shading in Unreal Engine 4](http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf" \t "_blank): explains Epic Games' split sum approximation. Real Shading in Unreal Engine 4：讲解了 Epic Games 的分割求和近似法。This is the article the IBL PBR code is based of.IBL PBR 部分的代码就脱胎于此文。

* [Physically Based Shading and Image Based Lighting](http://www.trentreed.net/blog/physically-based-shading-and-image-based-lighting/" \t "_blank): great blog post by Trent Reed about integrating specular IBL into a PBR pipeline in real time.Physically Based Shading and Image Based Lighting：Trent Reed 的精彩博客文章，介绍了如何将高光反射 IBL 实时集成到 PBR 管道中。

* [Image Based Lighting](https://chetanjags.wordpress.com/2015/08/26/image-based-lighting/" \t "_blank): very extensive write-up by Chetan Jags about specular-based image-based lighting and several of its caveats, including light probe interpolation.Image Based Lighting：Chetan Jags 对基于高光的 IBL 及其一些注意事项（包括光探针插值）进行了广泛的讲解。

* [Moving Frostbite to PBR](https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf" \t "_blank): well written and in-depth overview of integrating PBR into a AAA game engine by Sébastien Lagarde and Charles de Rousiers.Moving Frostbite to PBR：Sébastien Lagarde 和 Charles de Rousiers 撰写的，对于如何将 PBR 集成到 AAA 游戏引擎进行了详尽而深入的概述。

* [Physically Based Rendering – Part Three](https://jmonkeyengine.github.io/wiki/jme3/advanced/pbr_part3.html" \t "_blank): high level overview of IBL lighting and PBR by the JMonkeyEngine team.Physically Based Rendering – Part Three：JMonkeyEngine 团队对 IBL 和 PBR 进行了较高层次的概述。

* [Implementation Notes: Runtime Environment Map Filtering for Image Based Lighting](https://placeholderart.wordpress.com/2015/07/28/implementation-notes-runtime-environment-map-filtering-for-image-based-lighting/" \t "_blank): extensive write-up by Padraic Hennessy about pre-filtering HDR environment maps and significanly optimizing the sample process.Implementation Notes: Runtime Environment Map Filtering for Image Based Lighting：Padraic Hennessy 撰写的大量有关预过滤 HDR 环境贴图并显著优化采样过程的文章。