

图形渲染管线的第一个部分是顶点着色器(Vertex Shader)，它把一个单独的顶点作为输入。顶点着色器主要的目的是把3D坐标转为另一种3D坐标（后面会解释），同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配(Primitive Assembly)阶段将顶点着色器输出的所有顶点作为输入（如果是GL\_POINTS，那么就是一个顶点），并所有的点装配成指定图元的形状；本节例子中是一个三角形。

图元装配阶段的输出会传递给几何着色器(Geometry Shader)。几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。例子中，它生成了另一个三角形。

几何着色器的输出会被传入光栅化阶段(Rasterization Stage)，这里它会把图元映射为最终屏幕上相应的像素，生成供片段着色器(Fragment Shader)使用的片段(Fragment)。在片段着色器运行之前会执行裁切(Clipping)。裁切会丢弃超出你的视图以外的所有像素，用来提升执行效率。

OpenGL中的一个片段是OpenGL渲染一个像素所需的所有数据。

片段着色器的主要目的是计算一个像素的最终颜色，这也是所有OpenGL高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，我们叫做Alpha测试和混合(Blending)阶段。这个阶段检测片段的对应的深度（和模板(Stencil)）值（后面会讲），用它们来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查alpha值（alpha值定义了一个物体的透明度）并对物体进行混合(Blend)。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。

Shader输入输出

输入与输出

虽然着色器是各自独立的小程序，但是它们都是一个整体的一部分，出于这样的原因，我们希望每个着色器都有输入和输出，这样才能进行数据交流和传递。GLSL定义了in和out关键字专门来实现这个目的。每个着色器使用这两个关键字设定输入和输出，只要一个输出变量与下一个着色器阶段的输入匹配，它就会传递下去。但在顶点和片段着色器中会有点不同。

顶点着色器应该接收的是一种特殊形式的输入，否则就会效率低下。顶点着色器的输入特殊在，它从顶点数据中直接接收输入。为了定义顶点数据该如何管理，我们使用location这一元数据指定输入变量，这样我们才可以在CPU上配置顶点属性。我们已经在前面的教程看过这个了，layout (location = 0)。顶点着色器需要为它的输入提供一个额外的layout标识，这样我们才能把它链接到顶点数据。

你也可以忽略layout (location = 0)标识符，通过在OpenGL代码中使用glGetAttribLocation查询属性位置值(Location)，但是我更喜欢在着色器中设置它们，这样会更容易理解而且节省你（和OpenGL）的工作量。

另一个例外是片段着色器，它需要一个vec4颜色输出变量，因为片段着色器需要生成一个最终输出的颜色。如果你在片段着色器没有定义输出颜色，OpenGL会把你的物体渲染为黑色（或白色）。

所以，如果我们打算从一个着色器向另一个着色器发送数据，我们必须在发送方着色器中声明一个输出，在接收方着色器中声明一个类似的输入。当类型和名字都一样的时候，OpenGL就会把两个变量链接到一起，它们之间就能发送数据了（这是在链接程序对象时完成的）。为了展示这是如何工作的，我们会稍微改动一下之前教程里的那个着色器，让顶点着色器为片段着色器决定颜色

Uniform

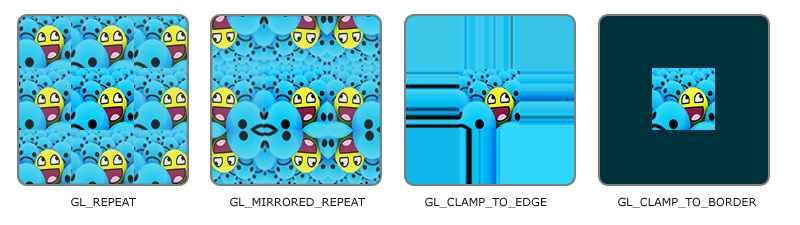
Uniform是一种从CPU中的应用向GPU中的着色器发送数据的方式，但uniform和顶点属性有些不同。首先，uniform是全局的(Global)。全局意味着uniform变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。第二，无论你把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。

纹理环绕方式

纹理坐标的范围通常是从(0, 0)到(1, 1)，那如果我们把纹理坐标设置在范围之外会发生什么？OpenGL默认的行为是重复这个纹理图像（我们基本上忽略浮点纹理坐标的整数部分），但OpenGL提供了更多的选择：

| **环绕方式** | **描述** |
| --- | --- |
| GL\_REPEAT | 对纹理的默认行为。重复纹理图像。 |
| GL\_MIRRORED\_REPEAT | 和GL\_REPEAT一样，但每次重复图片是镜像放置的。 |
| GL\_CLAMP\_TO\_EDGE | 纹理坐标会被约束在0到1之间，超出的部分会重复纹理坐标的边缘，产生一种边缘被拉伸的效果。 |
| GL\_CLAMP\_TO\_BORDER | 超出的坐标为用户指定的边缘颜色。 |

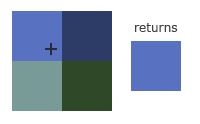
当纹理坐标超出默认范围时，每个选项都有不同的视觉效果输出。我们来看看这些纹理图像的例子：



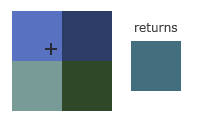
## 纹理过滤

纹理坐标不依赖于分辨率(Resolution)，它可以是任意浮点值，所以OpenGL需要知道怎样将纹理像素(Texture Pixel，也叫Texel，译注1)映射到纹理坐标。当你有一个很大的物体但是纹理的分辨率很低的时候这就变得很重要了。你可能已经猜到了，OpenGL也有对于纹理过滤(Texture Filtering)的选项。纹理过滤有很多个选项，但是现在我们只讨论最重要的两种：GL\_NEAREST和GL\_LINEAR。

GL\_NEAREST（也叫邻近过滤，Nearest Neighbor Filtering）是OpenGL默认的纹理过滤方式。当设置为GL\_NEAREST的时候，OpenGL会选择中心点最接近纹理坐标的那个像素。下图中你可以看到四个像素，加号代表纹理坐标。左上角那个纹理像素的中心距离纹理坐标最近，所以它会被选择为样本颜色：



GL\_LINEAR（也叫线性过滤，(Bi)linear Filtering）它会基于纹理坐标附近的纹理像素，计算出一个插值，近似出这些纹理像素之间的颜色。一个纹理像素的中心距离纹理坐标越近，那么这个纹理像素的颜色对最终的样本颜色的贡献越大。下图中你可以看到返回的颜色是邻近像素的混合色：



OpenGL使用一种叫做多级渐远纹理(Mipmap)的概念来解决这个问题，它简单来说就是一系列的纹理图像，后一个纹理图像是前一个的二分之一。多级渐远纹理背后的理念很简单：距观察者的距离超过一定的阈值，OpenGL会使用不同的多级渐远纹理，即最适合物体的距离的那个。由于距离远，解析度不高也不会被用户注意到。同时，多级渐远纹理另一加分之处是它的性能非常好。让我们看一下多级渐远纹理是什么样子的：

坐标系

为了将坐标从一个坐标系变换到另一个坐标系，我们需要用到几个变换矩阵，最重要的几个分别是模型(Model)、观察(View)、投影(Projection)三个矩阵。我们的顶点坐标起始于局部空间(Local Space)，在这里它称为局部坐标(Local Coordinate)，它在之后会变为世界坐标(World Coordinate)，观察坐标(View Coordinate)，裁剪坐标(Clip Coordinate)，并最后以屏幕坐标(Screen Coordinate)的形式结束。下面的这张图展示了整个流程以及各个变换过程做了什么：



1. 局部坐标是对象相对于局部原点的坐标，也是物体起始的坐标。
2. 下一步是将局部坐标变换为世界空间坐标，世界空间坐标是处于一个更大的空间范围的。这些坐标相对于世界的全局原点，它们会和其它物体一起相对于世界的原点进行摆放。
3. 接下来我们将世界坐标变换为观察空间坐标，使得每个坐标都是从摄像机或者说观察者的角度进行观察的。
4. 坐标到达观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标会被处理至-1.0到1.0的范围内，并判断哪些顶点将会出现在屏幕上。
5. 最后，我们将裁剪坐标变换为屏幕坐标，我们将使用一个叫做视口变换(Viewport Transform)的过程。视口变换将位于-1.0到1.0范围的坐标变换到由glViewport函数所定义的坐标范围内。最后变换出来的坐标将会送到光栅器，将其转化为片段。

词汇表

* **OpenGL**： 一个定义了函数布局和输出的图形API的正式规范。
* **GLAD**： 一个拓展加载库，用来为我们加载并设定所有OpenGL函数指针，从而让我们能够使用所有（现代）OpenGL函数。
* **视口(Viewport)**： 我们需要渲染的窗口。
* **图形管线(Graphics Pipeline)**： 一个顶点在呈现为像素之前经过的全部过程。
* **着色器(Shader)**： 一个运行在显卡上的小型程序。很多阶段的图形管道都可以使用自定义的着色器来代替原有的功能。
* **标准化设备坐标(Normalized Device Coordinates, NDC)**： 顶点在通过在剪裁坐标系中剪裁与透视除法后最终呈现在的坐标系。所有位置在NDC下-1.0到1.0的顶点将不会被丢弃并且可见。
* **顶点缓冲对象(Vertex Buffer Object)**： 一个调用显存并存储所有顶点数据供显卡使用的缓冲对象。
* **顶点数组对象(Vertex Array Object)**： 存储缓冲区和顶点属性状态。
* **索引缓冲对象(Element Buffer Object)**： 一个存储索引供索引化绘制使用的缓冲对象。
* **Uniform**： 一个特殊类型的GLSL变量。它是全局的（在一个着色器程序中每一个着色器都能够访问uniform变量），并且只需要被设定一次。
* **纹理(Texture)**： 一种包裹着物体的特殊类型图像，给物体精细的视觉效果。
* **纹理缠绕(Texture Wrapping)**： 定义了一种当纹理顶点超出范围(0, 1)时指定OpenGL如何采样纹理的模式。
* **纹理过滤(Texture Filtering)**： 定义了一种当有多种纹素选择时指定OpenGL如何采样纹理的模式。这通常在纹理被放大情况下发生。
* **多级渐远纹理(Mipmaps)**： 被存储的材质的一些缩小版本，根据距观察者的距离会使用材质的合适大小。
* **stb\_image.h**： 图像加载库。
* **纹理单元(Texture Units)**： 通过绑定纹理到不同纹理单元从而允许多个纹理在同一对象上渲染。
* **向量(Vector)**： 一个定义了在空间中方向和/或位置的数学实体。
* **矩阵(Matrix)**： 一个矩形阵列的数学表达式。
* **GLM**： 一个为OpenGL打造的数学库。
* **局部空间(Local Space)**： 一个物体的初始空间。所有的坐标都是相对于物体的原点的。
* **世界空间(World Space)**： 所有的坐标都相对于全局原点。
* **观察空间(View Space)**： 所有的坐标都是从摄像机的视角观察的。
* **裁剪空间(Clip Space)**： 所有的坐标都是从摄像机视角观察的，但是该空间应用了投影。这个空间应该是一个顶点坐标最终的空间，作为顶点着色器的输出。OpenGL负责处理剩下的事情（裁剪/透视除法）。
* **屏幕空间(Screen Space)**： 所有的坐标都由屏幕视角来观察。坐标的范围是从0到屏幕的宽/高。
* **LookAt矩阵**： 一种特殊类型的观察矩阵，它创建了一个坐标系，其中所有坐标都根据从一个位置正在观察目标的用户旋转或者平移。
* **欧拉角(Euler Angles)**： 被定义为偏航角(Yaw)，俯仰角(Pitch)，和滚转角(Roll)从而允许我们通过这三个值构造任何3D方向。

光照模型

现实世界的光照是极其复杂的，而且会受到诸多因素的影响，这是我们有限的计算能力所无法模拟的。因此OpenGL的光照使用的是简化的模型，对现实的情况进行近似，这样处理起来会更容易一些，而且看起来也差不多一样。这些光照模型都是基于我们对光的物理特性的理解。其中一个模型被称为冯氏光照模型(Phong Lighting Model)。冯氏光照模型的主要结构由3个分量组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。下面这张图展示了这些光照分量看起来的样子：



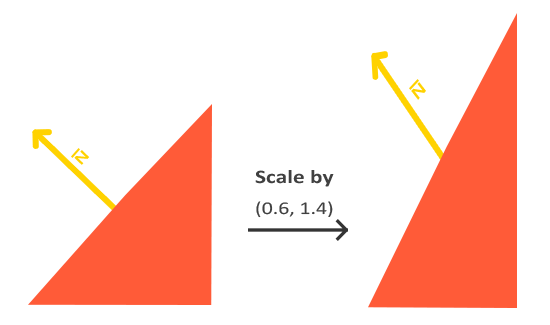
* 环境光照(Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色。
* 漫反射光照(Diffuse Lighting)：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮。
* 镜面光照(Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

**计算世界坐标系中的法向量时：**

现在我们已经把法向量从顶点着色器传到了片段着色器。可是，目前片段着色器里的计算都是在世界空间坐标中进行的。所以，我们是不是应该把法向量也转换为世界空间坐标？基本正确，但是这不是简单地把它乘以一个模型矩阵就能搞定的。

首先，法向量只是一个方向向量，不能表达空间中的特定位置。同时，法向量没有齐次坐标（顶点位置中的w分量）。这意味着，位移不应该影响到法向量。因此，如果我们打算把法向量乘以一个模型矩阵，我们就要从矩阵中移除位移部分，只选用模型矩阵左上角3×3的矩阵（注意，我们也可以把法向量的w分量设置为0，再乘以4×4矩阵；这同样可以移除位移）。对于法向量，我们只希望对它实施缩放和旋转变换。

其次，如果模型矩阵执行了不等比缩放，顶点的改变会导致法向量不再垂直于表面了。因此，我们不能用这样的模型矩阵来变换法向量。下面的图展示了应用了不等比缩放的模型矩阵对法向量的影响：



每当我们应用一个不等比缩放时（注意：等比缩放不会破坏法线，因为法线的方向没被改变，仅仅改变了法线的长度，而这很容易通过标准化来修复），法向量就不会再垂直于对应的表面了，这样光照就会被破坏。

修复这个行为的诀窍是使用一个为法向量专门定制的模型矩阵。这个矩阵称之为法线矩阵(Normal Matrix)，它使用了一些线性代数的操作来移除对法向量错误缩放的影响。如果你想知道这个矩阵是如何计算出来的，建议去阅读这个[文章](http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/)。

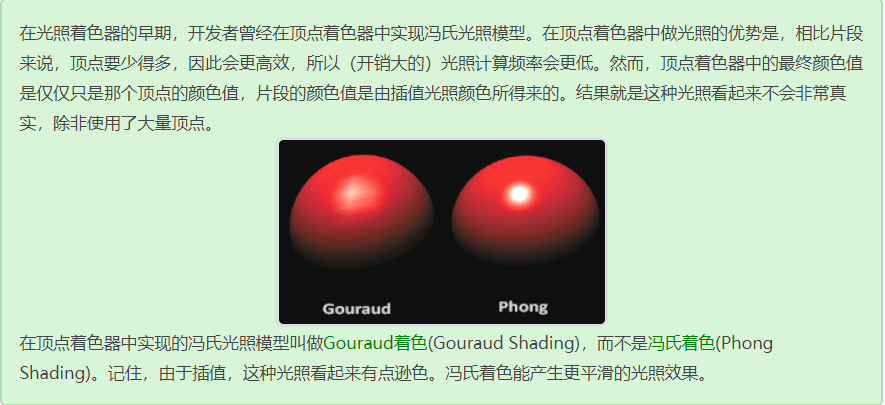
法线矩阵被定义为「模型矩阵左上角的逆矩阵的转置矩阵」。真是拗口，如果你不明白这是什么意思，别担心，我们还没有讨论逆矩阵(Inverse Matrix)和转置矩阵(Transpose Matrix)。注意，大部分的资源都会将法线矩阵定义为应用到模型-观察矩阵(Model-view Matrix)上的操作，但是由于我们只在世界空间中进行操作（不是在观察空间），我们只使用模型矩阵。

在顶点着色器中，我们可以使用inverse和transpose函数自己生成这个法线矩阵，这两个函数对所有类型矩阵都有效。注意我们还要把被处理过的矩阵强制转换为3×3矩阵，来保证它失去了位移属性以及能够乘以vec3的法向量。

Normal = mat3(transpose(inverse(model))) \* aNormal;

在漫反射光照部分，光照表现并没有问题，这是因为我们没有对物体本身执行任何缩放操作，所以并不是必须要使用一个法线矩阵，仅仅让模型矩阵乘以法线也可以。可是，如果你进行了不等比缩放，使用法线矩阵去乘以法向量就是必不可少的了。

即使是对于着色器来说，逆矩阵也是一个开销比较大的运算，因此，只要可能就应该避免在着色器中进行逆矩阵运算，它们必须为你场景中的每个顶点都进行这样的处理。用作学习目这样做是可以的，但是对于一个对效率有要求的应用来说，在绘制之前你最好用CPU计算出法线矩阵，然后通过uniform把值传递给着色器（像模型矩阵一样）。



词汇表

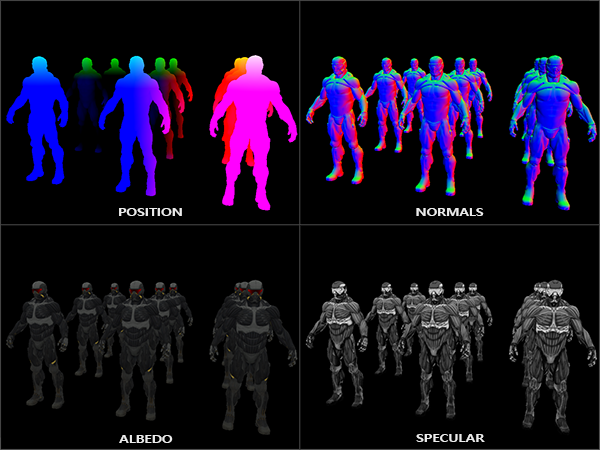
* **颜色向量(Color Vector)**：一个通过红绿蓝(RGB)分量的组合描绘大部分真实颜色的向量。一个物体的颜色实际上是该物体所不能吸收的反射颜色分量。
* **冯氏光照模型(Phong Lighting Model)**：一个通过计算环境光，漫反射，和镜面光分量的值来估计真实光照的模型。
* **环境光照(Ambient Lighting)**：通过给每个没有被光照的物体很小的亮度，使其不是完全黑暗的，从而对全局光照进行估计。
* **漫反射着色(Diffuse Shading)**：一个顶点/片段与光线方向越接近，光照会越强。使用了法向量来计算角度。
* **法向量(Normal Vector)**：一个垂直于平面的单位向量。
* **法线矩阵(Normal Matrix)**：一个3x3矩阵，或者说是没有平移的模型（或者模型-观察）矩阵。它也被以某种方式修改（逆转置），从而在应用非统一缩放时，保持法向量朝向正确的方向。否则法向量会在使用非统一缩放时被扭曲。
* **镜面光照(Specular Lighting)**：当观察者视线靠近光源在表面的反射线时会显示的镜面高光。镜面光照是由观察者的方向，光源的方向和设定高光分散量的反光度值三个量共同决定的。
* **冯氏着色(Phong Shading)**：冯氏光照模型应用在片段着色器。
* **Gouraud着色(Gouraud shading)**：冯氏光照模型应用在顶点着色器上。在使用很少数量的顶点时会产生明显的瑕疵。会得到效率提升但是损失了视觉质量。
* **GLSL结构体(GLSL struct)**：一个类似于C的结构体，用作着色器变量的容器。大部分时间用来管理输入/输出/uniform。
* **材质(Material)**：一个物体反射的环境光，漫反射，镜面光颜色。这些东西设定了物体所拥有的颜色。
* **光照属性(Light(properties))**：一个光的环境光，漫反射，镜面光的强度。可以使用任何颜色值，对每一个冯氏分量(Phong Component)定义光源发出的颜色/强度。
* **漫反射贴图(Diffuse Map)**：一个设定了每个片段中漫反射颜色的纹理图片。
* **镜面光贴图(Specular Map)**：一个设定了每一个片段的镜面光强度/颜色的纹理贴图。仅在物体的特定区域显示镜面高光。
* **定向光(Directional Light)**：只有一个方向的光源。它被建模为不管距离有多长所有光束都是平行而且其方向向量在整个场景中保持不变。
* **点光源(Point Light)**：一个在场景中有位置的，光线逐渐衰减的光源。
* **衰减(Attenuation)**：光随着距离减少强度的过程，通常使用在点光源和聚光下。
* **聚光(Spotlight)**：一个被定义为在某一个方向上的锥形的光源。
* **手电筒(Flashlight)**：一个摆放在观察者视角的聚光。
* **GLSL uniform数组(GLSL Uniform Array)**：一个uniform值数组。它的工作原理和C语言数组大致一样，只是不能动态分配内存。

法线贴图：主要注意切线空间的转换，使得法线贴图中的向量与世界坐标系中一致

延迟着色：

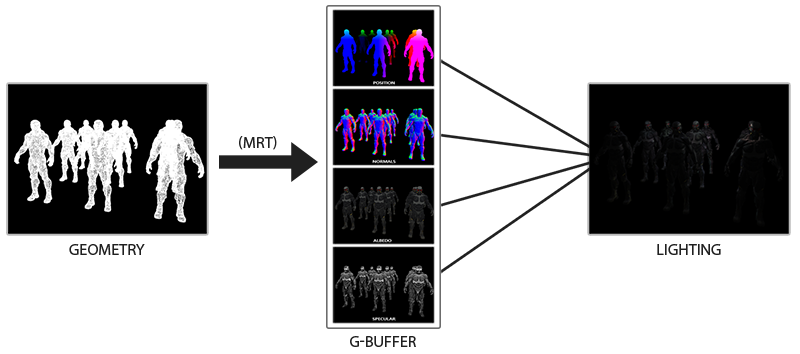
我们现在一直使用的光照方式叫做**正向渲染(Forward Rendering)**或者**正向着色法(Forward Shading)**，它是我们渲染物体的一种非常直接的方式，在场景中我们根据所有光源照亮一个物体，之后再渲染下一个物体，以此类推。它非常容易理解，也很容易实现，但是同时它对程序性能的影响也很大，因为对于每一个需要渲染的物体，程序都要对每一个光源每一个需要渲染的片段进行迭代，这是**非常**多的！因为大部分片段着色器的输出都会被之后的输出覆盖，正向渲染还会在场景中因为高深的复杂度(多个物体重合在一个像素上)浪费大量的片段着色器运行时间。

延迟着色法基于我们**延迟(Defer)**或**推迟(Postpone)**大部分计算量非常大的渲染(像是光照)到后期进行处理的想法。它包含两个处理阶段(Pass)：在第一个几何处理阶段(Geometry Pass)中，我们先渲染场景一次，之后获取对象的各种几何信息，并储存在一系列叫做G缓冲(G-buffer)的纹理中；想想位置向量(Position Vector)、颜色向量(Color Vector)、法向量(Normal Vector)和/或镜面值(Specular Value)。场景中这些储存在G缓冲中的几何信息将会在之后用来做(更复杂的)光照计算。下面是一帧中G缓冲的内容：



我们会在第二个光照处理阶段(Lighting Pass)中使用G缓冲内的纹理数据。在光照处理阶段中，我们渲染一个屏幕大小的方形，并使用G缓冲中的几何数据对每一个片段计算场景的光照；在每个像素中我们都会对G缓冲进行迭代。我们对于渲染过程进行解耦，将它高级的片段处理挪到后期进行，而不是直接将每个对象从顶点着色器带到片段着色器。光照计算过程还是和我们以前一样，但是现在我们需要从对应的G缓冲而不是顶点着色器(和一些uniform变量)那里获取输入变量了。

下面这幅图片很好地展示了延迟着色法的整个过程：



这种渲染方法一个很大的好处就是能保证在G缓冲中的片段和在屏幕上呈现的像素所包含的片段信息是一样的，因为深度测试已经最终将这里的片段信息作为最顶层的片段。这样保证了对于在光照处理阶段中处理的每一个像素都只处理一次，所以我们能够省下很多无用的渲染调用。除此之外，延迟渲染还允许我们做更多的优化，从而渲染更多的光源。

当然这种方法也带来几个缺陷， 由于G缓冲要求我们在纹理颜色缓冲中存储相对比较大的场景数据，这会消耗比较多的显存，尤其是类似位置向量之类的需要高精度的场景数据。 另外一个缺点就是他不支持混色(因为我们只有最前面的片段信息)， 因此也不能使用MSAA了。针对这几个问题我们可以做一些变通来克服这些缺点，这些我们留会在教程的最后讨论。

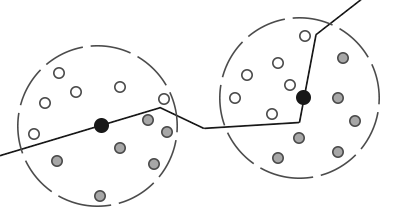
在几何处理阶段中填充G缓冲非常高效，因为我们直接储存像是位置，颜色或者是法线等对象信息到帧缓冲中，而这几乎不会消耗处理时间。在此基础上使用多渲染目标(Multiple Render Targets, MRT)技术，我们甚至可以在一个渲染处理之内完成这所有的工作。

<https://learnopengl-cn.github.io/05%20Advanced%20Lighting/08%20Deferred%20Shading/>

SSAO

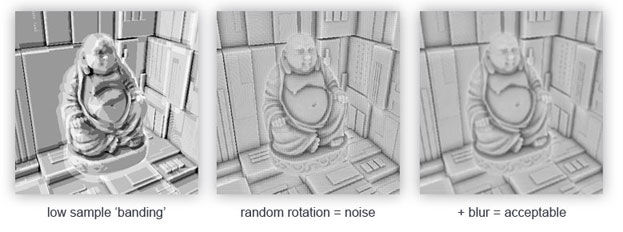
我们已经在前面的基础教程中简单介绍到了这部分内容：环境光照(Ambient Lighting)。环境光照是我们加入场景总体光照中的一个固定光照常量，它被用来模拟光的**散射(Scattering)**。在现实中，光线会以任意方向散射，它的强度是会一直改变的，所以间接被照到的那部分场景也应该有变化的强度，而不是一成不变的环境光。其中一种间接光照的模拟叫做**环境光遮蔽(Ambient Occlusion)**，它的原理是通过将褶皱、孔洞和非常靠近的墙面变暗的方法近似模拟出间接光照。这些区域很大程度上是被周围的几何体遮蔽的，光线会很难流失，所以这些地方看起来会更暗一些。站起来看一看你房间的拐角或者是褶皱，是不是这些地方会看起来有一点暗？

SSAO背后的原理很简单：对于铺屏四边形(Screen-filled Quad)上的每一个片段，我们都会根据周边深度值计算一个**遮蔽因子(Occlusion Factor)**。这个遮蔽因子之后会被用来减少或者抵消片段的环境光照分量。遮蔽因子是通过采集片段周围球型核心(Kernel)的多个深度样本，并和当前片段深度值对比而得到的。高于片段深度值样本的个数就是我们想要的遮蔽因子。



上图中在几何体内灰色的深度样本都是高于片段深度值的，他们会增加遮蔽因子；几何体内样本个数越多，片段获得的环境光照也就越少。

很明显，渲染效果的质量和精度与我们采样的样本数量有直接关系。如果样本数量太低，渲染的精度会急剧减少，我们会得到一种叫做**波纹(Banding)**的效果；如果它太高了，反而会影响性能。我们可以通过引入随机性到采样核心(Sample Kernel)的采样中从而减少样本的数目。通过随机旋转采样核心，我们能在有限样本数量中得到高质量的结果。然而这仍然会有一定的麻烦，因为随机性引入了一个很明显的噪声图案，我们将需要通过模糊结果来修复这一问题。下面这幅图片([John Chapman](http://john-chapman-graphics.blogspot.com/)的佛像)展示了波纹效果还有随机性造成的效果：



你可以看到，尽管我们在低样本数的情况下得到了很明显的波纹效果，引入随机性之后这些波纹效果就完全消失了。

通过在**法向半球体(Normal-oriented Hemisphere)**周围采样，我们将不会考虑到片段底部的几何体.它消除了环境光遮蔽灰蒙蒙的感觉，从而产生更真实的结果。这个SSAO教程将会基于法向半球法和John Chapman出色的[SSAO教程](http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html)。

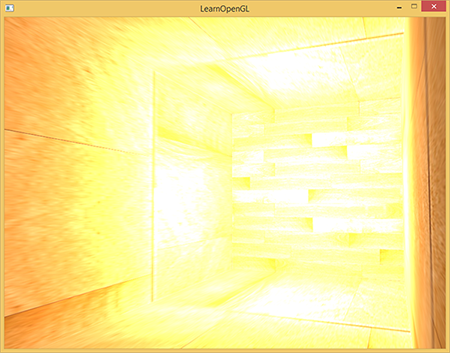
## 样本缓冲

SSAO需要获取几何体的信息，因为我们需要一些方式来确定一个片段的遮蔽因子。对于每一个片段，我们将需要这些数据：

* 逐片段**位置**向量
* 逐片段的**法线**向量
* 逐片段的**反射颜色**
* **采样核心**
* 用来旋转采样核心的随机旋转矢量（为了节约显存，不需要针对每个像素生成一个采样核心，只需要针对每个像素加个随机旋转的矢量即可）

HDR

一般来说，当存储在帧缓冲(Framebuffer)中时，亮度和颜色的值是默认被限制在0.0到1.0之间的。这个看起来无辜的语句使我们一直将亮度与颜色的值设置在这个范围内，尝试着与场景契合。这样是能够运行的，也能给出还不错的效果。但是如果我们遇上了一个特定的区域，其中有多个亮光源使这些数值总和超过了1.0，又会发生什么呢？答案是这些片段中超过1.0的亮度或者颜色值会被约束在1.0，从而导致场景混成一片，难以分辨：



这是由于大量片段的颜色值都非常接近1.0，在很大一个区域内每一个亮的片段都有相同的白色。这损失了很多的细节，使场景看起来非常假。

解决这个问题的一个方案是减小光源的强度从而保证场景内没有一个片段亮于1.0。然而这并不是一个好的方案，因为你需要使用不切实际的光照参数。一个更好的方案是让颜色暂时超过1.0，然后将其转换至0.0到1.0的区间内，从而防止损失细节。

显示器被限制为只能显示值为0.0到1.0间的颜色，但是在光照方程中却没有这个限制。通过使片段的颜色超过1.0，我们有了一个更大的颜色范围，这也被称作**HDR(High Dynamic Range, 高动态范围)**。有了HDR，亮的东西可以变得非常亮，暗的东西可以变得非常暗，而且充满细节。

HDR渲染和其很相似，我们允许用更大范围的颜色值渲染从而获取大范围的黑暗与明亮的场景细节，最后将所有HDR值转换成在[0.0, 1.0]范围的LDR(Low Dynamic Range,低动态范围)。转换HDR值到LDR值得过程叫做色调映射(Tone Mapping)，现在现存有很多的色调映射算法，这些算法致力于在转换过程中保留尽可能多的HDR细节。这些色调映射算法经常会包含一个选择性倾向黑暗或者明亮区域的参数。

实现只需借助一个浮点帧缓冲，

我们先渲染一个光照的场景到浮点帧缓冲中，之后再在一个铺屏四边形(Screen-filling Quad)上应用这个帧缓冲的颜色缓冲，代码会是这样子：

glBindFramebuffer(GL\_FRAMEBUFFER, hdrFBO);

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

// [...] 渲染(光照的)场景

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

// 现在使用一个不同的着色器将HDR颜色缓冲渲染至2D铺屏四边形上

hdrShader.Use();

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D, hdrColorBufferTexture);

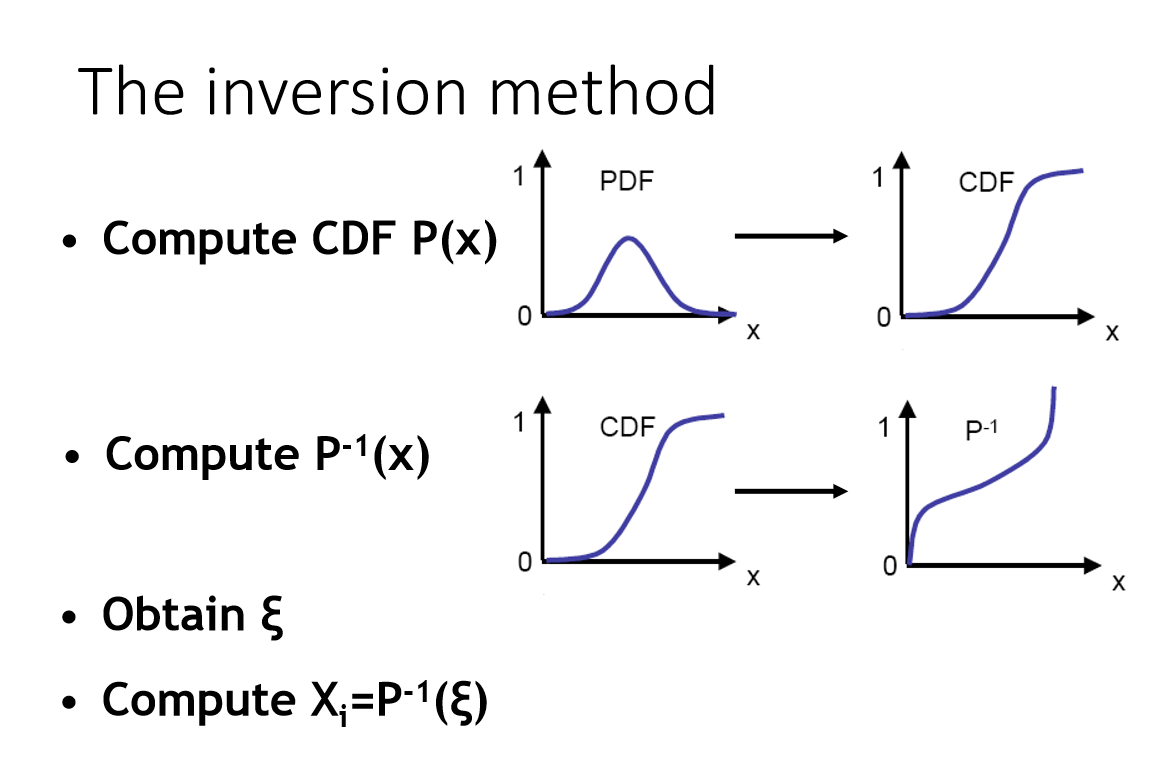
RenderQuad();

图形学中蒙特卡罗

蒙特卡洛方法（Monte Carlo method，也有翻译成“蒙特卡罗方法”）是以概率和统计的理论、方法为基础的一种数值计算方法，将所求解的问题同一定的概率模型相联系，用计算机实现统计模拟或抽样，以获得问题的近似解，故又称随机抽样法或统计试验法。

1． 用蒙特·卡罗方法模拟某一过程时，需要产生某一概率分布的随机变量。

2． 用统计方法把模型的数字特征估计出来，从而得到实际问题的数值解。



上图展示：如何利用0-1均匀分布得到所需要的分布，只需蒙特卡洛均匀分布取得的值经过一个函数的映射。