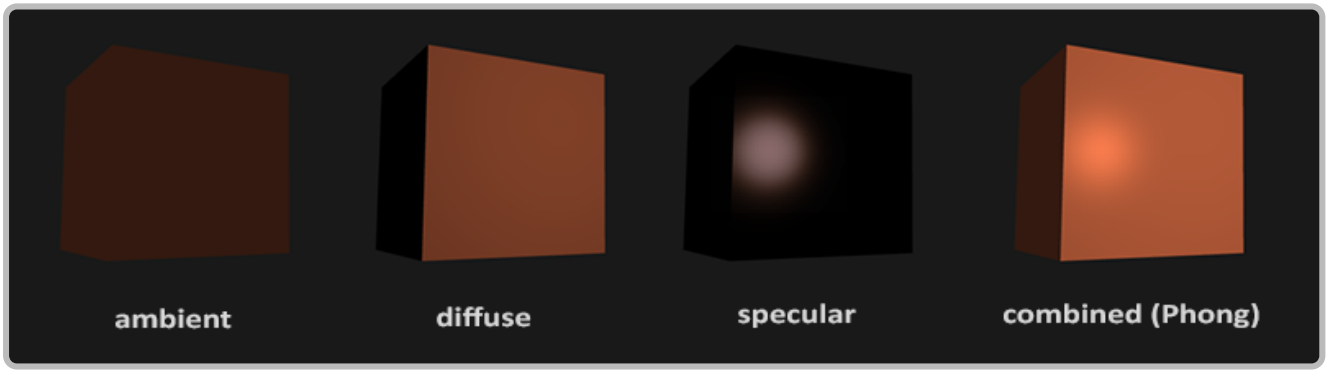
光照基础

原文	Basic Lighting
作者	JoeyDeVries
翻译	Django
校对	Geequlim, BLumia

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，这是以目前我们所拥有的处理能力无法模拟的。因此OpenGL的光照仅仅使用了简化的模型并基于对现实的估计来进行模拟，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个元素组成：环境(Ambient)、漫反射(Diffuse)和镜面(Specular)光照。这些光照元素看起来像下面这样：



- 环境光照(Ambient Lighting)：即使在黑暗的情况下，世界上也仍然有一些光亮(月亮、一个来自远处的光)，所以物体永远不会是完全黑暗的。我们使用环境光照来模拟这种情况，也就是无论如何永远都给物体一些颜色。
- 漫反射光照(Diffuse Lighting)：模拟一个发光物对物体的方向性影响(Directional Impact)。它是冯氏光照模型最显著的组成部分。面向光源的一面比其他面会更亮。
- 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，相比于物体的颜色更倾向于光的颜色。

为了创建有趣的视觉场景，我们希望模拟至少这三种光照元素。我们将以最简单的一个开始：**环境光照**。

环境光照

光通常都不是来自于同一光源，而是来自散落于我们周围的很多光源，即使它们可么显而易见。光的一个属性是，它可以向很多方向发散和反弹，所以光最后到达的不是它所临近的直射方向；光能够像这样**反射(Reflect)**到其他表面，一个物体的光照可能受到

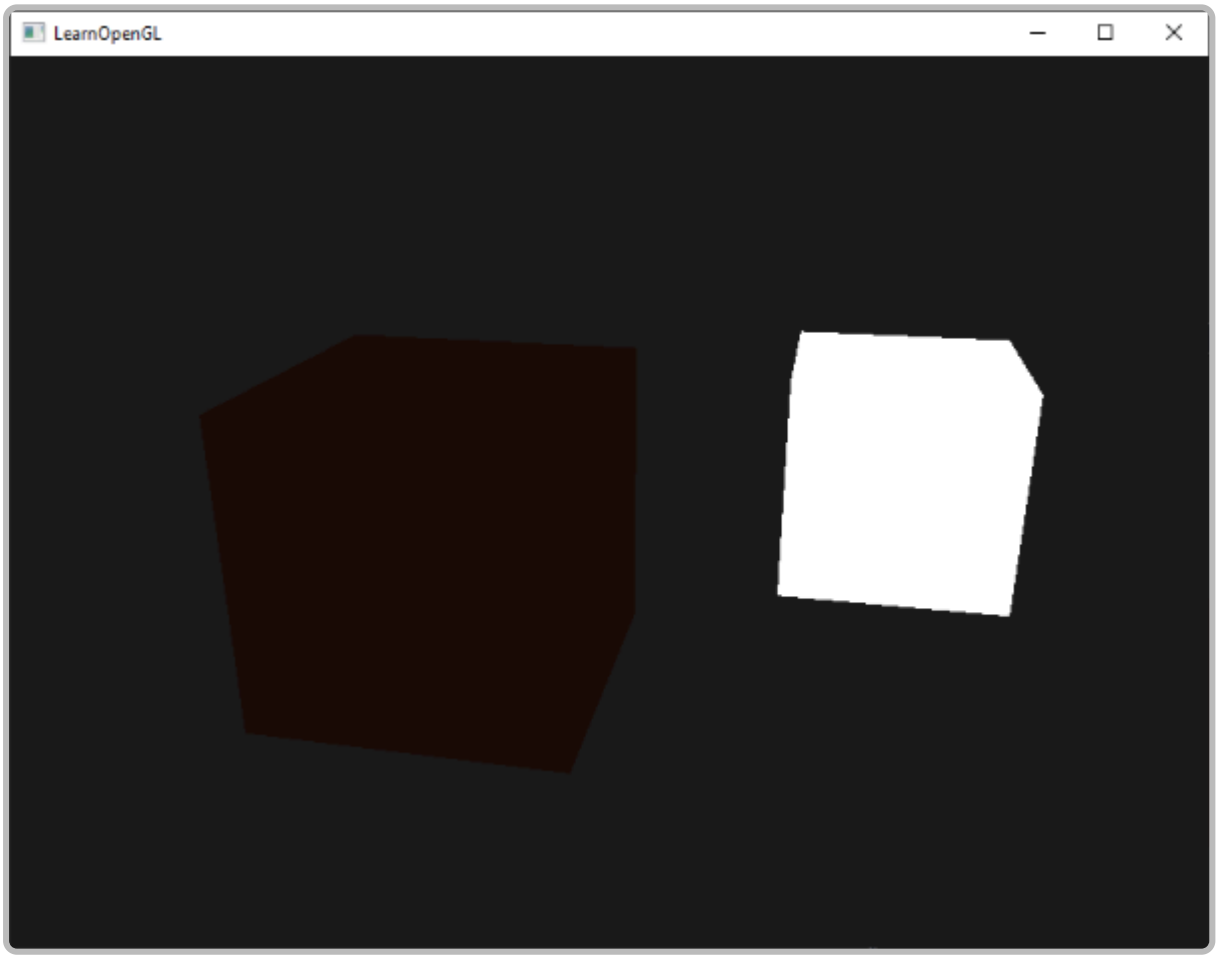
来自一个非直射的光源影响。考虑到这种情况的算法叫做**全局照明(Global Illumination)**算法，但是这种算法既开销高昂又极其复杂。

因为我们不是复杂和昂贵算法的死忠粉丝，所以我们将会使用一种简化的全局照明模型，叫做环境光照(Ambient Lighting)。如你在前面章节所见，我们使用一个(数值)很小的常量(光)颜色添加进物体**片段**(Fragment，指当前讨论的光线在物体上的照射点)的最终颜色里，这看起来就像即使没有直射光源也始终存在着一些发散的光。

把环境光照添加到场景里非常简单。我们用光的颜色乘以一个(数值)很小常量环境因子，再乘以物体的颜色，然后使用它作为片段的颜色：

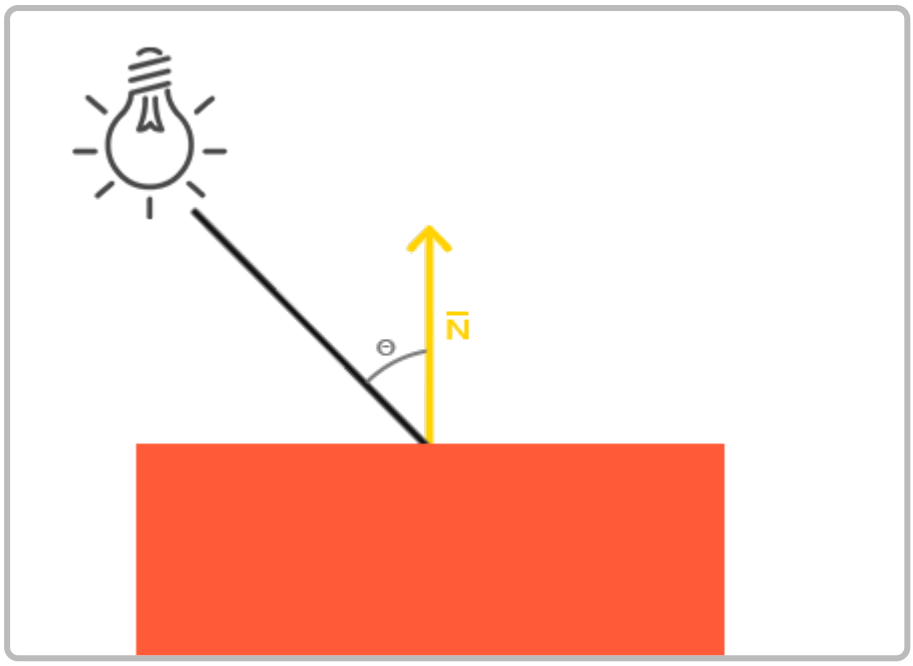
```
void main()
{
    float ambientStrength = 0.1f;
    vec3 ambient = ambientStrength * lightColor;
    vec3 result = ambient * objectColor;
    color = vec4(result, 1.0f);
}
```

如果你现在运行你的程序，你会注意到冯氏光照的第一个阶段已经应用到你的物体上了。这个物体非常暗，但不是完全的黑暗，因为我们应用了环境光照(注意发光立方体没被环境光照影响是因为我们对它使用了另一个着色器)。它看起来应该像这样：



漫反射光照

环境光本身不提供最明显的光照效果，但是漫反射光照(Diffuse Lighting)会对物体的视觉影响。漫反射光使物体上与光线排布越近的片段越能从光源处获得更多的亮度。为了更好的理解漫反射光照，请看下图：



图左上方有一个光源，它所发出的光线落在物体的一个片段上。我们需要测量这个光线与它所接触片段之间的角度。如果光线垂直于物体表面，这束光对物体的影响会最大化(译注：更亮)。为了测量光线和片段的角度，我们使用一个叫做法向量(Normal Vector)的东西，它是垂直于片段表面的一种向量(这里以黄色箭头表示)，我们在后面再讲这个东西。两个向量之间的角度就能够根据点乘计算出来。

你可能记得在变换那一节教程里，我们知道两个单位向量的角度越小，它们点乘的结果越倾向于1。当两个向量的角度是90度的时候，点乘会变为0。这同样适用于θ，θ越大，光对片段颜色的影响越小。

> **Important**
>
> 注意，我们使用的是单位向量(Unit Vector，长度是1的向量)取得两个向量夹角的余弦值，所以我们需要确保所有的向量都被标准化，否则点乘返回的值就不仅仅是余弦值了(如果你不明白，可以复习变换那一节的点乘部分)。

点乘返回一个标量，我们可以用它计算光线对片段颜色的影响，基于不同片段所朝向光源的方向的不同，这些片段被照亮的情况也不同。

所以，我们需要些什么来计算漫反射光照?

- 法向量：一个垂直于顶点表面的向量。
- 定向的光线：作为光的位置和片段的位置之间的向量差的方向向量。为了计算这个光线，我们需要光的位置向量和片段的位置向量。

法向量

法向量(Normal Vector)是垂直于顶点表面的(单位)向量。由于顶点自身并没有表面(它只是空间中一个独立的点)，我们利用顶点周围的顶点计算出这个顶点的表面。我们能够使用叉乘这个技巧为立方体所有的顶点计算出法线，但是由于3D立方体不是一个复杂的形状，所以我们可以简单的把法线数据手工添加到顶点数据中。更新的顶点数据数组可以在这里找到。试着去想象一下，这些法向量真的是垂直于立方体的各个面的表面的(一个立方体由6个面组成)。

因为我们向顶点数组添加了额外的数据，所以我们应该更新光照的顶点着色器：

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
...
```

现在我们已经向每个顶点添加了一个法向量，已经更新了顶点着色器，我们还要更新顶点属性指针(Vertex Attibute Pointer)。注意，发光物使用同样的顶点数组作为它的顶点数据，然而发光物的着色器没有使用新添加的法向量。我们不会更新发光物的着色器或者属性配置，但是我们必须至少修改一下顶点属性指针来适应新的顶点数组的大小：

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoi
glEnableVertexAttribArray(0);
```

我们只想使用每个顶点的前三个浮点数，并且我们忽略后三个浮点数，所以我们只需要把**步长**参数改成`GLfloat`尺寸的6倍就行了。

> **Important**
>
> 发光物着色器顶点数据的不完全使用看起来有点低效，但是这些顶点数据已经从立方体对象载入到GPU的内存里了，所以GPU内存不是必须再储存新数据。相对于重新给发光物分配VBO，实际上却是更高效了。

所有光照的计算需要在片段着色器里进行，所以我们需要把法向量由顶点着色器传递到片段着色器。我们这么做：

```
out vec3 Normal;
void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    Normal = normal;
}
```

剩下要做的事情是，在片段着色器中定义相应的输入值：

```
in vec3 Normal;
```

计算漫反射光照

每个顶点现在都有了法向量，但是我们仍然需要光的位置向量和片段的位置向量。由于光的位置是一个静态变量，我们可以简单的在片段着色器中把它声明为uniform：

```
uniform vec3 lightPos;
```

然后再游戏循环中(外面也可以，因为它不会变)更新uniform。我们使用在前面教`lightPos`向量作为光源位置：

```
GLint lightPosLoc = glGetUniformLocation(lightingShader.Program, "lightPos"
glUniform3f(lightPosLoc, lightPos.x, lightPos.y, lightPos.z);
```

最后，我们还需要片段的位置(Position)。我们会在世界空间中进行所有的光照计算，因此我们需要一个在世界空间中的顶点位置。我们可以通过把顶点位置属性乘以模型矩阵(Model Matrix,只用模型矩阵不需要用观察和投影矩阵)来把它变换到世界空间坐标。这个在顶点着色器中很容易完成，所以让我们就声明一个输出(out)变量，然后计算它的世界空间坐标：

```
out vec3 FragPos;
out vec3 Normal;

void main()
{
    gl_Position = projection * view * model * vec4(position, 1.0f);
    FragPos = vec3(model * vec4(position, 1.0f));
    Normal = normal;
}
```

最后，在片段着色器中添加相应的输入变量。

```
in vec3 FragPos;
```

现在，所有需要的变量都设置好了，我们可以在片段着色器中开始光照的计算了。

我们需要做的第一件事是计算光源和片段位置之间的方向向量。前面提到，光的方向向量是光的位置向量与片段的位置向量之间的向量差。你可能记得，在变换教程中，我们简单的通过两个向量相减的方式计算向量差。我们同样希望确保所有相关向量最后都转换为单位向量，所以我们把法线和方向向量这个结果都进行标准化：

```
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
```

> **Important**
>
> 当计算光照时我们通常不关心一个向量的“量”或它的位置，我们只关心它们的方向。所有的计算都使用单位向量完成，因为这会简化了大多数计算(比如点乘)。所以当进行光照计算时，确保你总是对相关向量进行标准化，这样它们才会保证自身为单位向量。忘记对向量进行标准化是一个十分常见的错误。

下一步，我们对`norm`和`lightDir`向量进行点乘，来计算光对当前片段的实际的散射影响。结果值再乘以光的颜色，得到散射因子。两个向量之间的角度越大，散射因子就会越小：

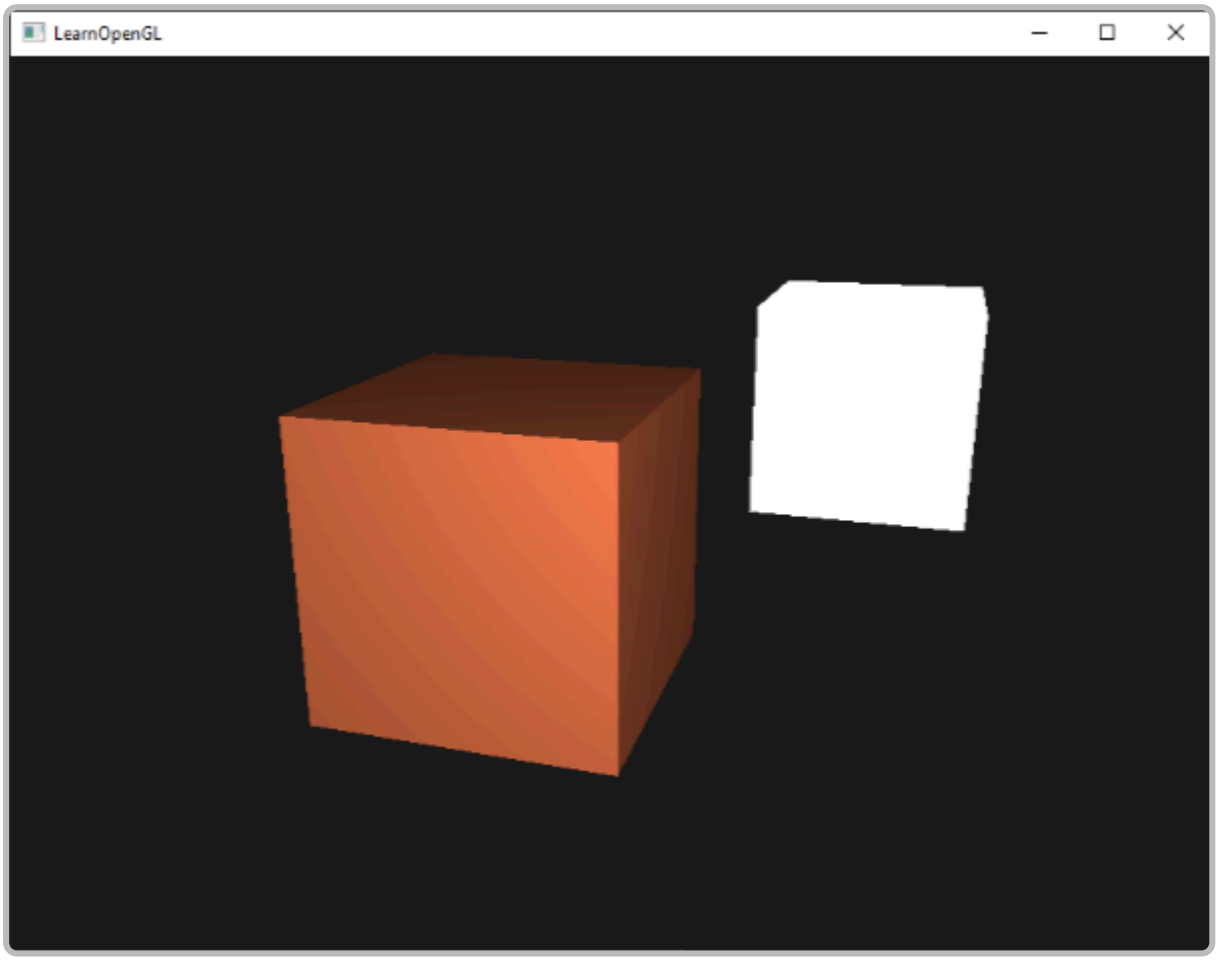
```
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;
```

如果两个向量之间的角度大于90度，点乘的结果就会变成负数，这样会导致散射因子变为负数。为此，我们使用`max`函数返回两个参数之间较大的参数，从而保证散射因子不会变成负数。负数的颜色是没有实际定义的，所以最好避免它，除非你是那种古怪的艺术家。

既然我们有了一个环境光照颜色和一个散射光颜色，我们把它们相加，然后把结果乘以物体的颜色，来获得片段最后的输出颜色。

```
vec3 result = (ambient + diffuse) * objectColor;
color = vec4(result, 1.0f);
```

如果你的应用(和着色器)编译成功了，你可能看到类似的输出：



你可以看到使用了散射光照，立方体看起来就真的像个立方体了。尝试在你的脑中想象，通过移动正方体，法向量和光的方向向量之间的夹角增大，片段变得更暗。

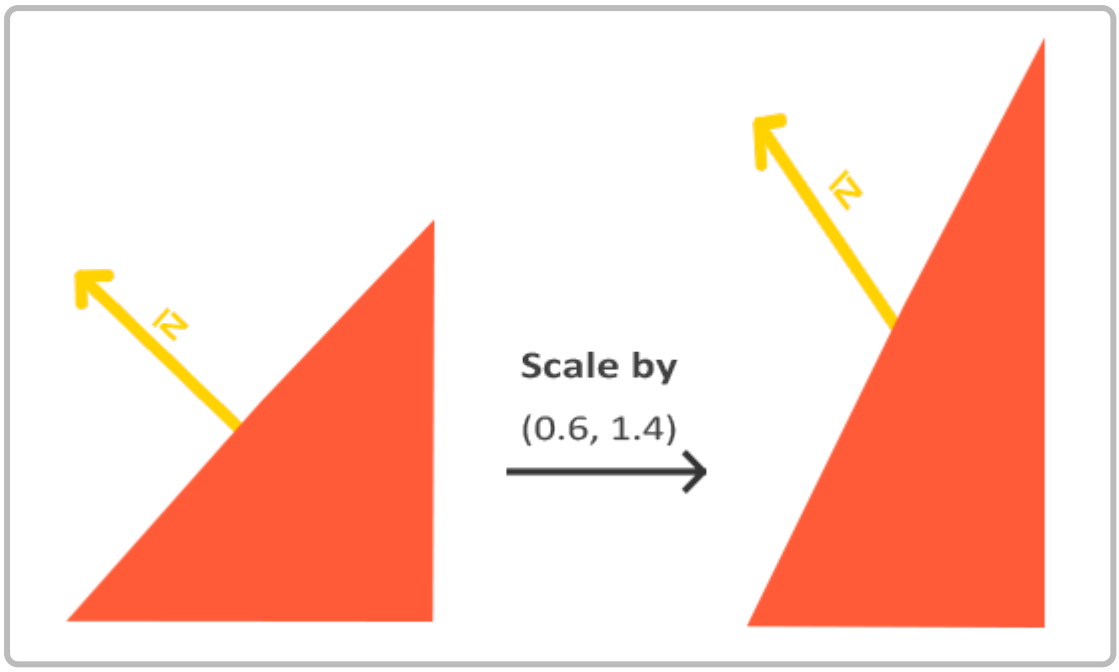
如果你遇到很多困难，可以对比完整的源代码以及片段着色器代码。

最后一件事

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，我们都是在世界空间坐标中进行计算的，所以，我们不是应该把法向量转换为世界空间坐标吗？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，平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，我们就要把模型矩阵左上角的3×3矩阵从模型矩阵中移除(译注：所谓移除就是设置为0)，它是模型矩阵的平移部分(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；同样可以移除平移)。对于法向量，我们只能对它应用缩放(Scale)和旋转(Rotation)变换。

其次，如果模型矩阵执行了不等比缩放，法向量就不再垂直于表面了，顶点就会以这种方式被改变了。因此，我们不能用这样的模型矩阵去乘以法向量。下面的图展示了应用了不等比缩放的矩阵对法向量的影响：



无论何时当我们提交一个不等比缩放(注意：等比缩放不会破坏法线，因为法线的方向没被改变，而法线的长度很容易通过标准化进行修复)，法向量就不会再垂直于它们的表面了，这样光照会被扭曲。

修复这个行为的诀窍是使用另一个为法向量专门定制的模型矩阵。这个矩阵称之为正规矩阵(Normal Matrix)，它是进行了一点线性代数操作移除了对法向量的错误缩放效果。如果你想知道这个矩阵是如何计算出来的，我建议看这个文章。

正规矩阵被定义为“模型矩阵左上角的逆矩阵的转置矩阵”。真拗口，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，定义正规矩阵的大多资源就像应用到模型观察矩阵(Model-view Matrix)上的操作一样，但是由于我们只在世界空间工作(而不是在观察空间)，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`函数自己生成正规矩阵，`inverse`和`transpose`函数对所有类型矩阵都有效。注意，我们也要把这个被处理过的矩阵强制转换为3×3矩阵，这是为了保证它失去了平移属性，之后它才能乘以法向量。

```
Normal = mat3(transpose(inverse(model))) * normal;
```

在环境光照部分，光照表现没问题，这是因为我们没有对物体本身执行任何缩放操作，因而不是非得使用正规矩阵不可，用模型矩阵乘以法线也没错。可是，如果你进行了不等比缩放，使用正规矩阵去乘以法向量就是必不可少的了。

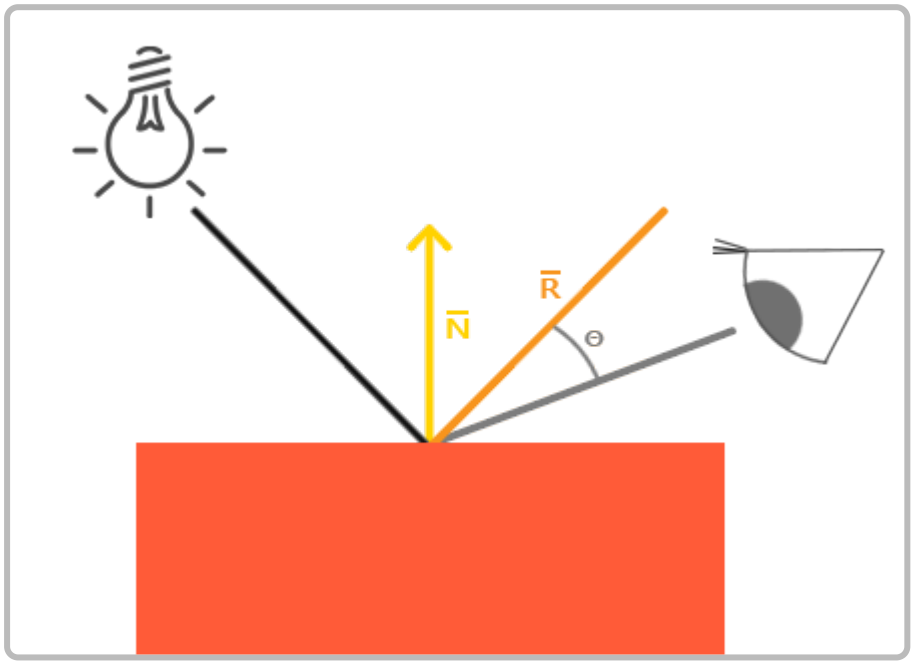
Attention

对于着色器来说，逆矩阵也是一种开销比较大的操作，因此，无论何时，在着色器中只要可能就应该尽量避免逆操作，因为它们必须为你场景中的每个顶点进行这样的处理。因为学习的目的这样做很好，但是对于一个对于效率有要求的应用来说，在绘制之前，CPU计算出正规矩阵，然后通过uniform把值传递给着色器(和模型矩阵一样)。

镜面光照

如果你还没被这些光照计算搞得精疲力尽，我们就再把镜面高光(Specular Highlight)加进来，这样冯氏光照才算完整。

和环境光照一样，镜面光照(Specular Lighting)同样依据光的方向向量和物体的法向量，但是这次它也会依据观察方向，例如玩家是从什么方向看着这个片段的。镜面光照根据光的反射特性。如果我们想象物体表面像一面镜子一样，那么，无论我们从哪里去看那个表面所反射的光，镜面光照都会达到最大化。你可以从下面的图片看到效果：



我们通过反射法向量周围光的方向计算反射向量。然后我们计算反射向量和视线方向的角度，如果之间的角度越小，那么镜面光的作用就会越大。它的作用效果就是，当我们去看光被物体所反射的那个方向的时候，我们会看到一个高光。

观察向量是镜面光照的一个附加变量，我们可以使用观察者世界空间位置(Viewer’s World Space Position)和片段的位置来计算。之后，我们计算镜面光亮度，用它乘以光的颜色，在用它加上作为之前计算的光照颜色。

> **Important**
>
> 我们选择在世界空间(World Space)进行光照计算，但是大多数人趋向于在观察空间(View Space)进行光照计算。在观察空间计算的好处是，观察者的位置总是(0, 0, 0)，所以这样你直接就获得了观察者位置。可是，我发现出于学习的目的，在世界空间计算光照更符合直觉。如果你仍然希望在视野空间计算光照的话，那就使用观察矩阵应用到所有相关的需要变换的向量(不要忘记，也要改变正规矩阵)。

为了得到观察者的世界空间坐标，我们简单地使用摄像机对象的位置坐标代替(它就是观察者)。所以我们把另一个uniform添加到片段着色器，把相应的摄像机位置坐标传给片段着色器：

```
uniform vec3 viewPos;

GLint viewPosLoc = glGetUniformLocation(lightingShader.Program, "viewPos");
glUniform3f(viewPosLoc, camera.Position.x, camera.Position.y, camera.Positi
```

现在我们已经获得所有需要的变量，可以计算高光亮度了。首先，我们定义一个镜面强度(Specular Intensity)变量`specularStrength`，给镜面高光一个中等亮度颜色，这样就不会产生过度的影响了。

```
float specularStrength = 0.5f;
```

如果我们把它设置为1.0f，我们会得到一个对于珊瑚色立方体来说过度明亮的镜面亮度因子。下一节教程，我们会讨论所有这些光照亮度的合理设置，以及它们是如何影响物体的。下一步，我们计算视线方向坐标，和沿法线轴的对应的反射坐标：

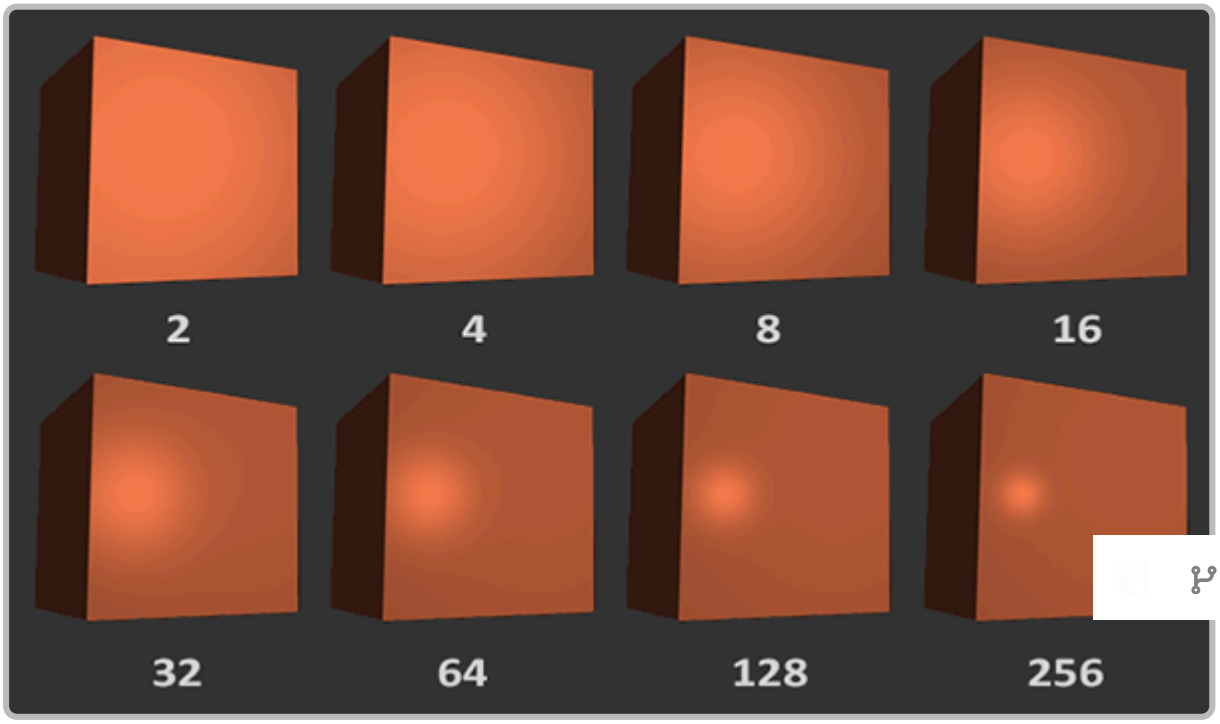
```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
```

需要注意的是我们使用了`lightDir`向量的相反数。`reflect`函数要求的第一个是从光源指向片段位置的向量，但是`lightDir`当前是从片段指向光源的向量(由先前我们计算`lightDir`向量时，(减数和被减数)减法的顺序决定)。为了保证我们得到正确的`reflect`坐标，我们通过`lightDir`向量的相反数获得它的方向的反向。第二个参数要求是一个法向量，所以我们提供的是已标准化的`norm`向量。

剩下要做的是计算镜面亮度分量。下面的代码完成了这件事：

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

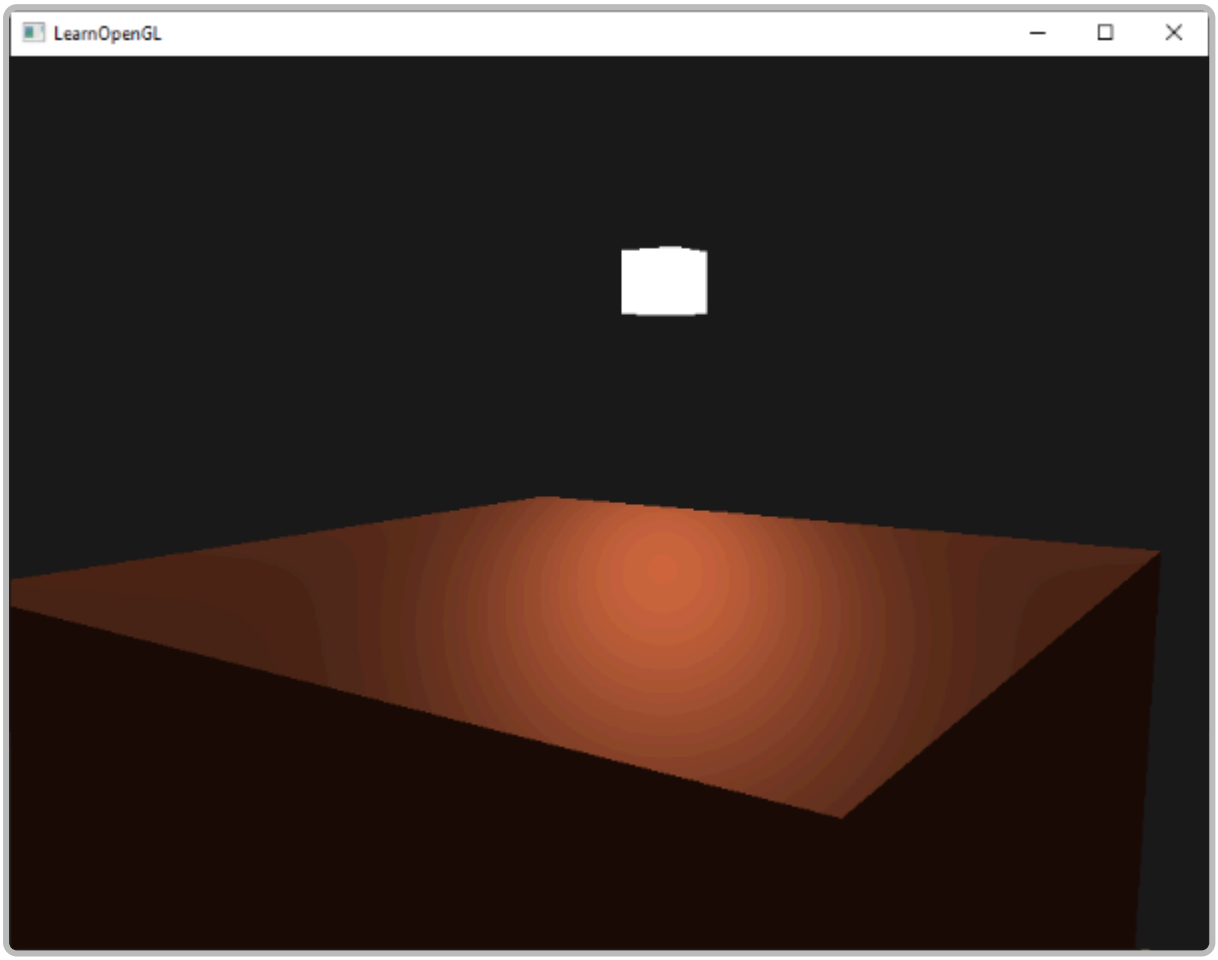
我们先计算视线方向与反射方向的点乘(确保它不是负值)，然后得到它的32次幂。这个32是高光的**发光值(Shininess)**。一个物体的发光值越高，反射光的能力越强，散射得越少，高光点越小。在下面的图片里，你会看到不同发光值对视觉(效果)的影响：



我们不希望镜面成分过于显眼，所以我们把指数设置为32。剩下的最后一件事情是把它添加到环境光颜色和散射光颜色里，然后再乘以物体颜色：

```
vec3 result = (ambient + diffuse + specular) * objectColor;
color = vec4(result, 1.0f);
```

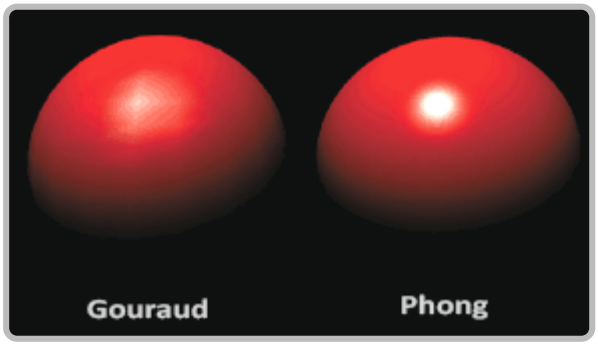
我们现在为冯氏光照计算了全部的光照元素。根据你的观察点，你可以看到类似下面的画面：



你可以在这里找到完整源码，在这里有顶点和片段着色器。

Important

早期的光照着色器，开发者在顶点着色器中实现冯氏光照。在顶点着色器中做这件事的优势是，相比片段来说，顶点要少得多，因此会更高效，所以(开销大的)光照计算频率会更低。然而，顶点着色器中的颜色值是只是顶点的颜色值，片段的颜色值是它与周围的颜色值的插值。结果就是这种光照看起来不会非常真实，除非使用了大量顶点。



在顶点着色器中实现的冯氏光照模型叫做Gouraud着色，而不是冯氏着色。记住这值，这种光照连起来有点逊色。冯氏着色能产生更平滑的光照效果。

现在你可以看到着色器的强大之处了。只用很少的信息，着色器就能计算出光照，影响到为我们所有物体的片段颜色。下一节中，我们会更深入的研究光照模型，看看我们还能做些什么。

练习

- 目前，我们的光源时静止的，你可以尝试使用`sin`和`cos`函数让光源在场景中来回移动，此时再观察光照效果能让你更容易理解冯氏光照模型。参考解答。
- 尝试使用不同的环境光、散射镜面强度，观察光照效果。改变镜面光照的`shininess`因子试试。
- 在观察空间(而不是世界空间)中计算冯氏光照：参考解答。
- 尝试实现一个Gouraud光照来模拟冯氏光照，参考解答。