着色器： Shader.h

纹理

坐标系统：model view projection

摄像机： Camera.h

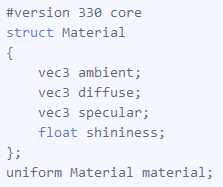
光照：光的类型：环境光、漫反射、镜面光照

**光照**：光的类型：环境光、漫反射、镜面光照

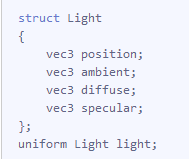
材质：

每个点：位置，纹理坐标，法向量着色器： Shader.h

**物体材质**：ambient材质向量定义了在环境光照下这个物体反射的是什么颜色；通常这是和物体颜色相同的颜色。diffuse材质向量定义了在漫反射光照下物体的颜色。漫反射颜色被设置为(和环境光照一样)我们需要的物体颜色。specular材质向量设置的是物体受到的镜面光照的影响的颜色(或者可能是反射一个物体特定的镜面高光颜色)。最后，shininess影响镜面高光的散射/半径。



**光的不同分量的属性**：



注意：镜面反射都是白光（lightSpecularLoc=1.0f,1.0f,1.0f）

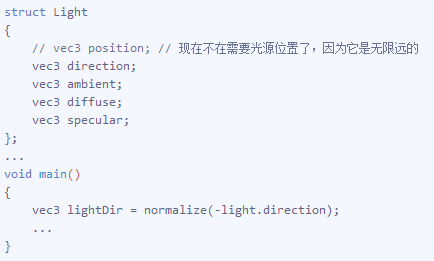


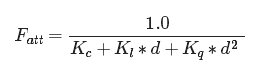
**漫反射贴图**

这次我们把纹理以sampler2D类型储存在Material结构体中。我们使用diffuse贴图替代早期定义的vec3类型的diffuse颜色。我们也要移除amibient材质颜色向量，因为ambient颜色绝大多数情况等于diffuse颜色，所以不需要分别去储存它：

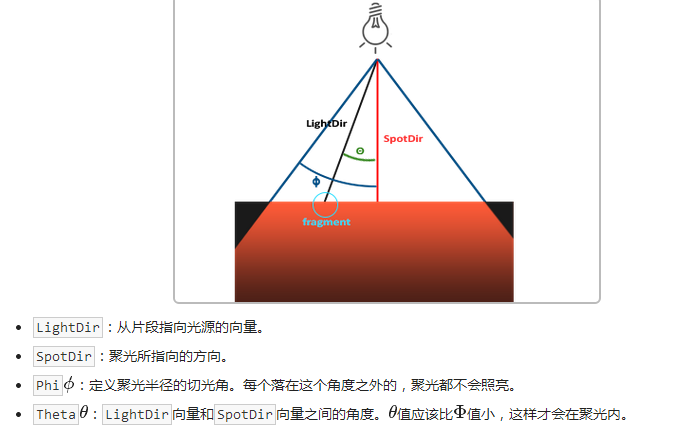
**透光物：**

绝我们之前讲的都是点光。接下来讨论定向光(directional light)，接着是作为之前学到知识的扩展的点光(point light)，最后我们讨论聚光(Spotlight)。

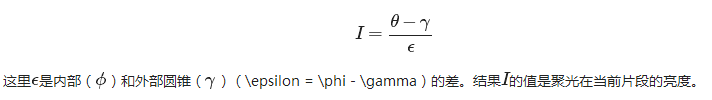
定向光：如太阳

点光：考虑衰减

手电筒：聚光



使突变不是太明显：平滑/软化边缘



每个点：位置，纹理坐标，法向量

Assimp载入模型

长时间探索均以失败告终，然而良久终于发现一人

<http://blog.csdn.net/huutu/article/details/50154875>

于此博客上提供下载可用链接Converter，其内附方便的assimp，

新建dependence文件夹，将assimp文件夹置入其内

添加链接（附加库目录、附加依赖项名称assimpd.dll，利用lib/debug中的assimpd.lib即可）

同时，将生成的assimpd.dll放入外部的debug（即exe文件目录下）

编译，即可运行！

载入模型步骤教学：

从模型网站上下载3dmax文件，打开。若纹理为tga可能需要转化（下载图片格式转化工具，bmp已知可用，然后修改名字（不超过8位！C++读取），转出为3ds文件（导出后不要再修改名字！））

导出为3ds文件，用assimp（我的文档）打开3ds文件（不要忘了纹理！放在和3ds文件统一目录下），导出为obj文件。然后将obj及obj.tml及纹理存入文件夹中（用于程序读取路径）。然后在程序中用Model类导入obj文件。

（同时，在3dsmax中修改导入贴图时记得把模式调成standard模式才能载入贴图到模型上）

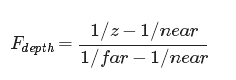
**高级OpenGL：**

**深度测试-----**

开启glEnable(GL\_DEPTH\_TEST);

深度测试函数：

用的是



这样，近的占长多，精度更高，更有深度感

两个平面或三角形如此紧密相互平行深度缓冲区不具有足够的精度以至于无法得到哪一个靠前。结果是，这两个形状不断似乎切换顺序导致怪异出问题。这被称为****深度冲突(Z-fighting)****，因为它看上去像形状争夺顶靠前的位置。

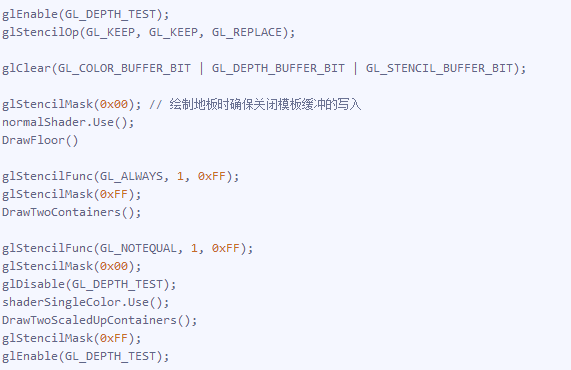
如何防止：1.不要让物体靠的太近，制造一点物体无法察觉的偏移，

1. 近平面设置的远，精度提高
2. 放弃一些性能来得到更高精度值，如现在显卡支持32位深度

**模板缓冲：**

模板测试发生在透明度测试（alpha test）之后，深度测试（depth test）之前。如果模板测试通过，则相应的像素点更新，否则不更新。

用处：给物体加边框，设置哪些显示哪些不显示（开关模板）

一共有两种函数可供我们使用去配置模板测试：glStencilFunc和glStencilOp。

混合：制作草，根据alpha值使用discard

透明----我们必须先绘制最远的物体，最后绘制最近的物体。普通的无混合物体仍然可以使用深度缓冲正常绘制，所以不必给它们排序。我们一定要保证它们在透明物体前绘制好。（镜子）

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

此时则需要使用png图片来实现镜面效果，会自动读取alpha通道的值（别忘了修改vs文件中对应的layout位置）

制作png：将下载的图片导入PS，将不用的部分裁剪掉，用透明度小的图层矩形填充，导出为png，然后用下面句式导入。

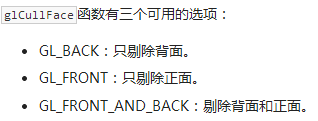
GLuint mirrorMap = loadTexture("mirror.png", true);

**面剔除：**

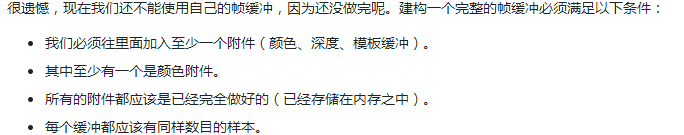
开启glEnable(GL\_CULL\_FACE);

opengl将绘制逆时针顺序的面，忽略顺时针顺序的面（因此提供顶点时要注意）

OpenGL默认关闭面剔除



帧缓存：

用于写入颜色值的颜色缓冲，用于写入深度信息的深度缓冲，以及允许我们基于一些条件丢弃指定片段的模板缓冲。把这几种缓冲结合起来叫做帧缓冲(Framebuffer)，它被储存于内存中。OpenGL给了我们自己定义帧缓冲的自由，我们可以选择性的定义自己的颜色缓冲、深度和模板缓冲。当你创建了你的窗口的时候默认帧缓冲就被创建和配置好了（GLFW为我们做了这件事）。通过创建我们自己的帧缓冲我们能够获得一种额外的渲染方式。

用处：存到一个四边形纹理上，对屏幕上的像素进行整体处理

顶点着色器有意思的地方在于我们把输入的位置向量作为输出给片段着色器的纹理坐标。片段着色器就会把它们作为输入去采样samplerCube：

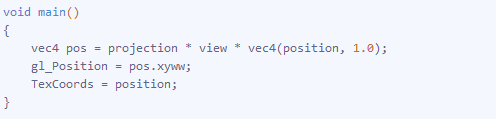
glm::mat4 view = glm::mat4(glm::mat3(camera.GetViewMatrix()));

SkyboxVAOVBO用提供的代码。。。

优化：

问题是天空盒是个1×1×1的立方体，极有可能会渲染失败，因为极有可能通不过深度测试。简单地不用深度测试渲染它也不是解决方案，这是因为天空盒会在之后覆盖所有的场景中其他物体。我们需要耍个花招让深度缓冲相信天空盒的深度缓冲有着最大深度值1.0，如此只要有个物体存在深度测试就会失败，看似物体就在它前面了。

同时，要设置为GL\_LEQUAL



**反射：**

将环境投影反射到物体上（可到复杂模型，利用normal运算）

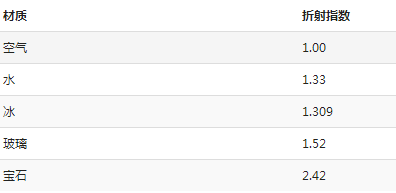
在渲染箱子前我们还得绑定立方体贴图纹理：

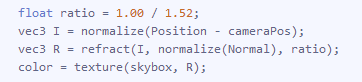
glBindTexture(GL\_TEXTURE\_CUBE\_MAP, skyboxTexture);

但是现实中大多数模型都不是完全反射的。我们可以引进反射贴图（reflection map）来使模型有另一层细节。和diffuse、specular贴图一样，我们可以从反射贴图上采样来决定fragment的反射率。使用反射贴图我们还可以决定模型的哪个部分有反射能力，以及强度是多少。



折射：





动态环境贴图：考虑物体相对环境的运动，思路------对每个面生成贴图，贴到包围物体的cubemap，然后计算反射和折射，然而这样的话每个物体都要算六次，计算开销加大

**高级数据：**

我们还可以调用glBufferSubData函数填充特定区域的缓冲

把数据直接映射到缓冲上使用glMapBuffer很有用，因为不用把它储存在临时内存里。你可以从文件读取数据然后直接复制到缓冲的内存里。

*分批处理顶点属性：*

使用glVertexAttribPointer函数可以指定缓冲内容的顶点数组的属性的布局(Layout)。我们已经知道，通过使用顶点属性指针我们可以交叉(Interleave)属性，也就是说我们可以把每个顶点的位置、法线、纹理坐标放在彼此挨着的地方。现在我们了解了更多的缓冲的内容，可以采取另一种方式了。

我们可以做的是把每种类型的属性的所有向量数据批量保存在一个布局，而不是交叉布局。与交叉布局123123123123不同，我们采取批量方式111122223333。

**GLSL的内建变量：**

### **gl\_FragCoord**：输入变量gl\_FragCoord让我们可以读得当前片段的窗口空间坐标和深度值，但是它是只读的

if(gl\_FragCoord.x < 400) color = vec4(1.0f, 0.0f, 0.0f, 1.0f);

**gl\_FrontFacing:**在面剔除教程中，我们提到过OpenGL可以根据顶点绘制顺序弄清楚一个面是正面还是背面。如果我们不适用面剔除，那么gl\_FrontFacing变量能告诉我们当前片段是某个正面的一部分还是背面的一部分。gl\_FrontFacing变量是一个布尔值，如果当前片段是正面的一部分那么就是true，否则就是false。这样我们可以创建一个立方体，里面和外面使用不同的纹理：

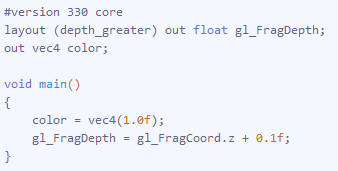
if(gl\_FrontFacing) color = texture(frontTexture, TexCoords);

如果你开启了面剔除，你就看不到箱子里面有任何东西了，所以此时使用gl\_FrontFacing毫无意义。

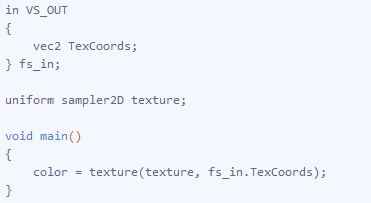
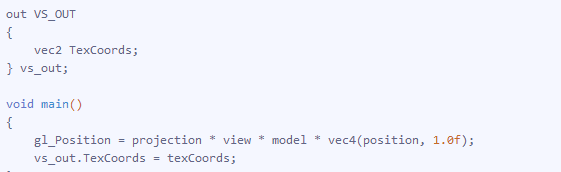
Gl\_FragDepth:GLSL给我们提供了一个叫做gl\_FragDepth的变量，我们可以用它在着色器中遂舍之像素的深度值。如果着色器中没有像gl\_FragDepth变量写入，它就会自动采用gl\_FragCoord.z的值。

这需要在片段着色器的顶部使用深度条件（depth condition）来重新声明gl\_FragDepth：

layout (depth\_<condition>) out float gl\_FragDepth;



**接口块：**



**Uniform缓冲对象：**

因为uniform缓冲对象是一个缓冲，因此我们可以使用glGenBuffers创建一个，然后绑定到GL\_UNIFORM\_BUFFER缓冲目标上，

#version 330 core

layout (location = 0) in vec3 position;

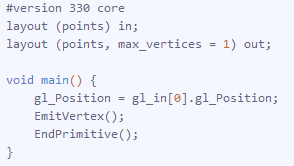
layout (std140) uniform Matrices { mat4 projection; mat4 view; };

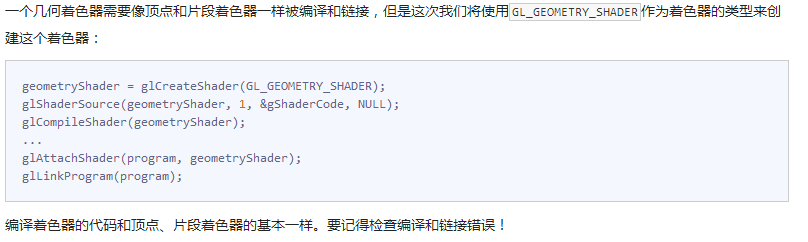
**几何着色器：**

在顶点和片段着色器之间有一个可选的着色器，叫做几何着色器(Geometry Shader)。几何着色器以一个或多个表示为一个单独基本图形（primitive）的顶点作为输入，比如可以是一个点或者三角形。几何着色器在将这些顶点发送到下一个着色阶段之前，可以将这些顶点转变为它认为合适的内容。几何着色器有意思的地方在于它可以把（一个或多个）顶点转变为完全不同的基本图形（primitive），从而生成比原来多得多的顶点。

Points, line\_strip, triangle\_strip

使用glDrawArrays(GL\_POINTS, 0, 4)调用





爆破物体：



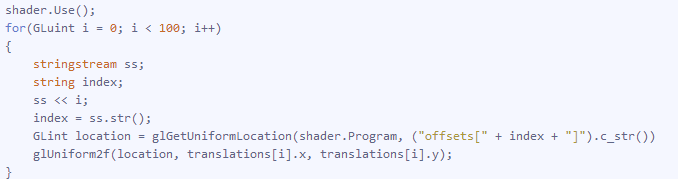
将读入三角形坐标沿法线方向放缩，实现爆炸效果

另一作用：检测法向量

**实例化：**

量如果有许多循环，只能重复调用glDrawArray等。glDrawArrays或glDrawElements这样的函数(Draw call)过多。会降低执行效率。

实例化是一种只调用一次渲染函数却能绘制出很多物体的技术，它节省渲染物体时从CPU到GPU的通信时间，而且只需做一次即可。要使用实例化渲染，我们必须将glDrawArrays和glDrawElements各自改为glDrawArraysInstanced和glDrawElementsInstanced。



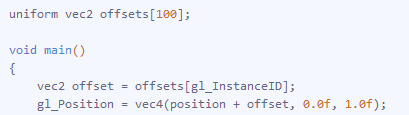
glBindVertexArray(quadVAO);

glDrawArraysInstanced(GL\_TRIANGLES, 0, 6, 100);

glBindVertexArray(0);

顶点着色器：

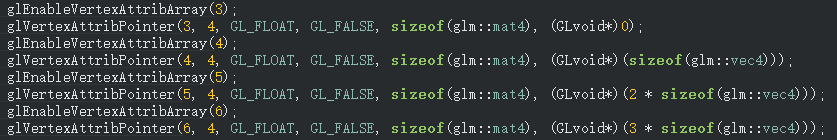
简单物体

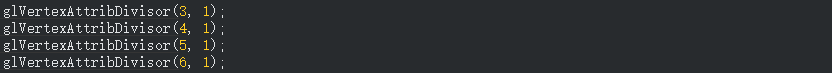


小行星带

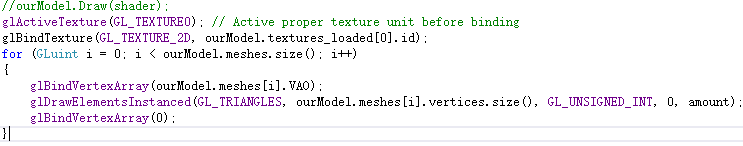
先生成位置数组modelMatrices[ ],

将会





别忘了

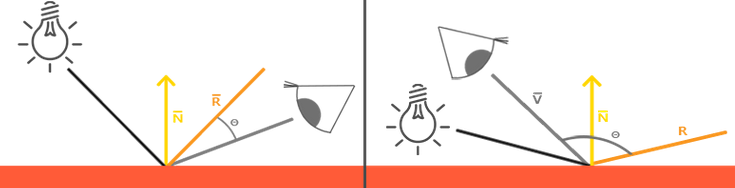


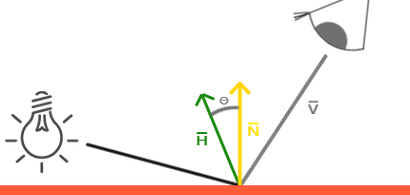
还可以利用内建变量：gl\_InstanceID

**高级光照：**

建Blinn-Phong光照模型：

按照以前计算镜面光照的方法，若夹角大于90°，则算出结果为0（如下）。为此进行修正









比较



**Gamma校正：**

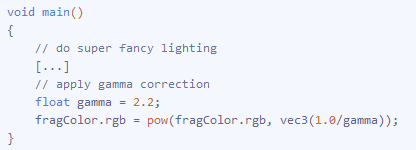
过去，大多数监视器是阴极射线管显示器（CRT）。这些监视器有一个物理特性就是两倍的输入电压产生的不是两倍的亮度。输入电压产生约为输入电压的2.2次幂的亮度，这叫做监视器Gamma。

比如一个东西的亮度0.2，让人感觉它比原来变亮一倍，那么现在这个亮度应该成为0.6，而不是0.4，也就是说人眼感知到的亮度的变化并非线性均匀分布的。

比如第二个灰阶里亮度0.1的光子数量是0.2的二分之一），但是由于这与我们的眼睛感知亮度不完全一致（对比较暗的颜色变化更敏感），所以它看起来很奇怪。

法一：glEnable(GL\_FRAMEBUFFER\_SRGB);

法二：在每个相关像素着色器运行的最后应用gamma校正



这实际意味着所有你创建或编辑的图片并不是在线性空间，而是在sRGB空间中（译注：sRGB空间定义的gamma接近于2.2），假如在你的屏幕上对暗红色翻一倍，便是根据你所感知到的亮度进行的，并不等于将红色元素加倍。

sRGB纹理：注意，如果已在监视器修改到了想要的纹理，就不要重新调sRGB纹理。

一个解决方案是重校，或把这些sRGB纹理在进行任何颜色值的计算前变回线性空间。我们可以这样做：

float gamma = 2.2;

vec3 diffuseColor = pow(texture(diffuse, texCoords).rgb, vec3(gamma));

为每个sRGB空间的纹理做这件事非常烦人。幸好，OpenGL给我们提供了另一个方案来解决我们的麻烦，这就是GL\_SRGB和GL\_SRGB\_ALPHA内部纹理格式。

如果我们在OpenGL中创建了一个纹理，把它指定为以上两种sRGB纹理格式其中之一，OpenGL将自动把颜色校正到线性空间中，这样我们所使用的所有颜色值都是在线性空间中的了。我们可以这样把一个纹理指定为一个sRGB纹理：

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_SRGB, width, height, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, image);

**光照衰减：**

真实的物理世界中，光照的衰减和光源的距离的平方成反比。

float attenuation = 1.0 / (distance \* distance);

然而，当我们使用这个衰减公式的时候，衰减效果总是过于强烈，光只能照亮一小圈，看起来并不真实。出于这个原因，我们使用在基本光照教程中所讨论的那种衰减方程，它给了我们更大的控制权，此外我们还可以使用双曲线函数：

float attenuation = 1.0 / distance;

使用gamma修正后需要注意光照衰减，因为

无gamma为1/d^2（二次导数符合物理），有gamma为1/d^2.2（双曲线符合物理）

**阴影映射：**

第一步我们需要生成一张深度贴图(Depth Map)。深度贴图是从光的透视图里渲染的深度纹理

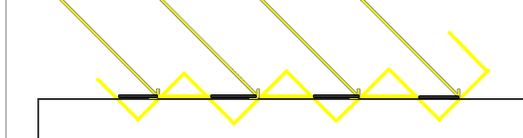
对应于一个片元的一个0到1之间的深度值。如果我们从光源的透视图来渲染场景，并把深度值的结果储存到纹理中会怎样？通过这种方式，我们就能对光源的透视图所见的最近的深度值进行采样。最终，深度值就会显示从光源的透视图下见到的第一个片元了。我们管储存在纹理中的所有这些深度值，叫做深度贴图（depth map）或阴影贴图。

### 渲染至深度贴图gl\_Position = lightSpaceMatrix \* model \* vec4(position, 1.0f);

渲染阴影：vec3 lighting = (ambient + (1.0 - shadow) \* (diffuse + specular)) \* color;

改进阴影贴图：

阴影失真：



由于计算机的离散性，可能相邻的有的被计算为最近点，有的却是最远点

校正方法：使用阴影偏移

float bias = max(0.05 \* (1.0 - dot(normal, lightDir)), 0.005);

float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;

选用正确的偏移数值，在不同的场景中需要一些像这样的轻微调校，但大多情况下，实际上就是增加偏移量直到所有失真都被移除的问题。

该调校仍存在问题：悬浮-----

为了修复peter游移，我们要进行正面剔除，先必须开启GL\_CULL\_FACE：

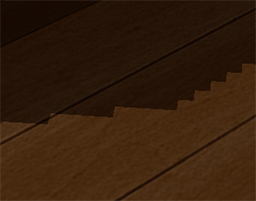
glCullFace(GL\_FRONT);

RenderSceneToDepthMap();

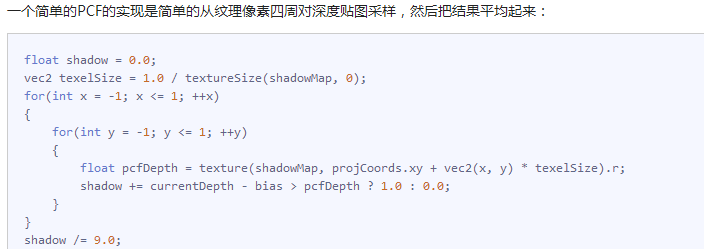
glCullFace(GL\_BACK); // 不要忘记设回原先的culling face

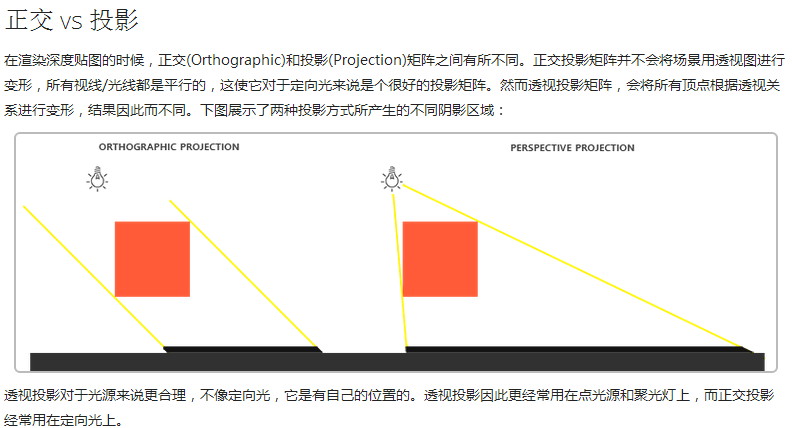
PCF阴影锯齿

阴影现在已经附着到场景中了，不过这仍不是我们想要的。如果你放大看阴影，阴影映射对解析度的依赖很快变得很明显。



1. 通过增加深度贴图解析度的方式来降低锯齿块
2. 从深度贴图中多次采样





**法线贴图：**

以光照算法的视角考虑的话，只有一件事决定物体的形状，这就是垂直于它的法线向量。

这种每个fragment使用各自的法线，替代一个面上所有fragment使用同一个法线的技术叫做法线贴图（normal mapping）或凹凸贴图（bump mapping）。



这会是一种偏蓝色调的纹理（你在网上找到的几乎所有法线贴图都是这样的）。这是因为所有法线的指向都偏向z轴（0, 0, 1）这是一种偏蓝的颜色。法线向量从z轴方向也向其他方向轻微偏移，颜色也就发生了轻微变化，这样看起来便有了一种深度。例如，你可以看到在每个砖块的顶部，颜色倾向于偏绿，这是因为砖块的顶部的法线偏向于指向正y轴方向（0, 1, 0），这样它就是绿色的了。

生成法线贴图方法：使用PS滤镜里的nvtools插件

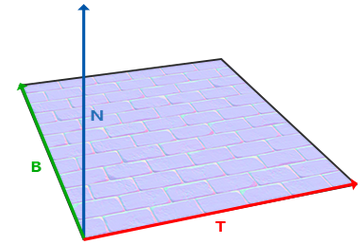
uniform sampler2D normalMap;

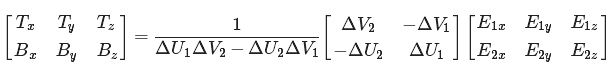
vec3 normal = texture(normalMap, fs\_in.TexCoords).rgb;

normal = normalize(normal \* 2.0 - 1.0);

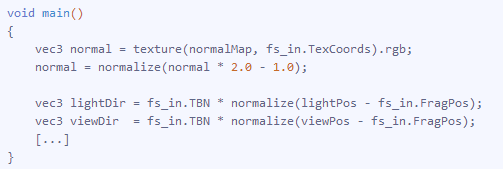
**切线空间：**

使用一个特定的矩阵我们就能将本地/切线空寂中的法线向量转成世界或视图坐标，使它们转向到最终的贴图表面的方向。

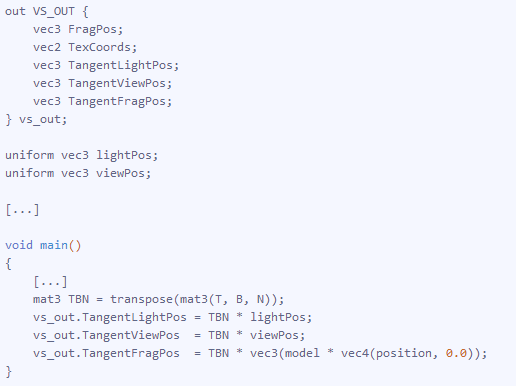
详细计算过程见网页



我们用TBN的逆矩阵把所有世界空间的向量转换到切线空间，使用这个矩阵将除法线以外的所有相关光照变量转换到切线空间中；这样法线也能和其他光照变量处于同一空间之中。

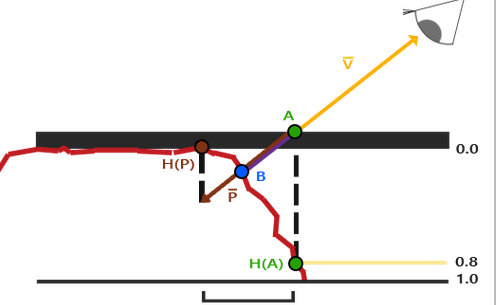
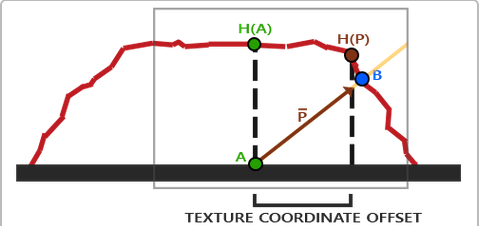


现在不是把TBN矩阵的逆矩阵发送给像素着色器，而是将切线空间的光源位置，观察位置以及顶点位置发送给像素着色器。这样我们就不用在像素着色器里进行矩阵乘法了。这是一个极佳的优化，因为顶点着色器通常比像素着色器运行的少。



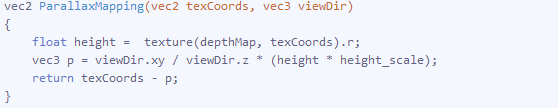
**视差贴图：**

这里粗糙的红线代表高度贴图中的数值的立体表达，向量V¯V¯代表观察方向。如果平面进行实际位移，观察者会在点BB看到表面。然而我们的平面没有实际上进行位移，观察方向将在点AA与平面接触。视差贴图的目的是，在AA位置上的fragment不再使用点AA的纹理坐标而是使用点BB的。随后我们用点BB的纹理坐标采样，观察者就像看到了点BB一样。



依旧使用切线空间进行转化。

同时引入ParallaxMapping函数



（帧缓冲重新解释：帧就是屏幕上显示的每一帧图像，帧缓冲就是实现前后图片渲染分离（离屏渲染），如果不设置的话就默认用原来的屏幕绘制帧，如果你创建了FrameBuffer并设置，就可以使用你自己的帧

然而，对每一帧而言，他都可以包含以下缓冲（至少包含颜色缓冲）：深度，模板，颜色，。。

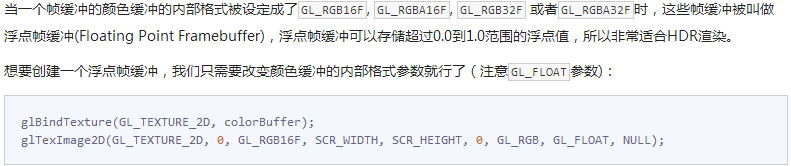
注意：帧不是一副2D图像，可以看成一个压缩成二维但仍然包含立体信息的3D图像）

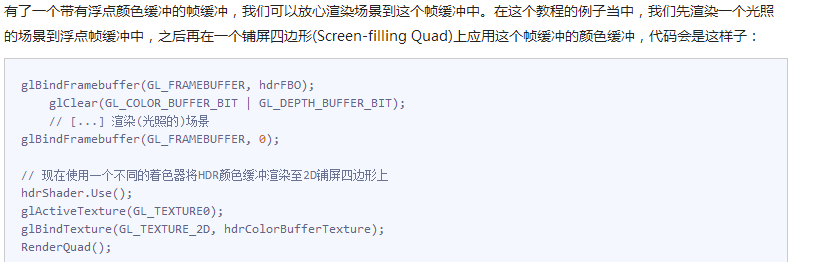
**HDR高动态范围渲染：**

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。

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。





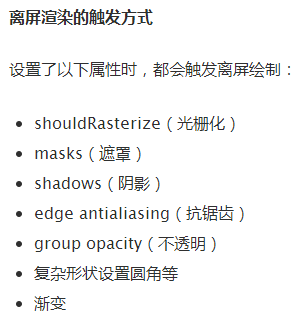


（以下解释离屏渲染）

**GPU屏幕渲染有以下两种方式：**

On-Screen Rendering  
意为当前屏幕渲染，指的是GPU的渲染操作是在当前用于显示的屏幕缓冲区中进行

Off-Screen Rendering  
意为离屏渲染，指的是GPU在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作。



**泛光：**

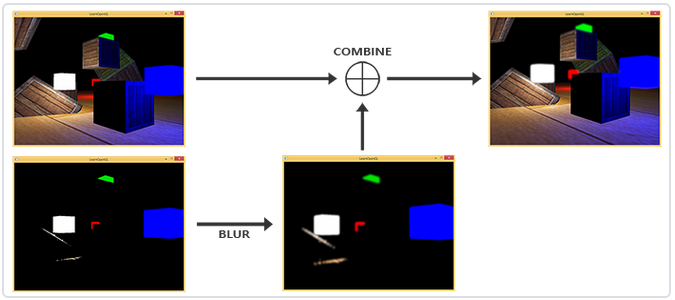


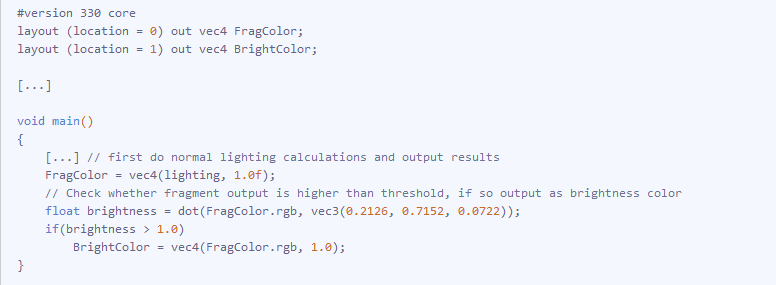
泛光可以极大提升场景中的光照效果，并提供了极大的效果提升，尽管做到这一切只需一点改变。

Bloom和HDR结合使用效果很好。常见的一个误解是HDR和泛光是一样的，很多人认为两种技术是可以互换的。但是它们是两种不同的技术，用于各自不同的目的上。可以使用默认的8位精确度的帧缓冲，也可以在不使用泛光效果的时候，使用HDR。只不过在有了HDR之后再实现泛光就更简单了。

实现方式：

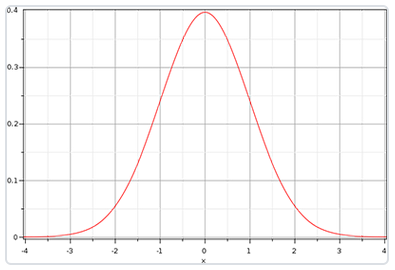
提取目标中的亮色，进行模糊处理





利用模糊滤镜：高斯模糊

高斯曲线在它的中间处的面积最大，使用它的值作为权重使得近处的样本拥有最大的优先权。



幸运的是，高斯方程有个非常巧妙的特性，它允许我们把二维方程分解为两个更小的方程：一个描述水平权重，另一个描述垂直权重。我们首先用水平权重在整个纹理上进行水平模糊，然后在经改变的纹理上进行垂直模糊。利用这个特性，结果是一样的，但是可以节省难以置信的性能，因为我们现在只需做32+32次采样，不再是1024了！这叫做两步高斯模糊。

我们为图像的模糊处理创建两个基本的帧缓冲，每个只有一个颜色缓冲纹理：

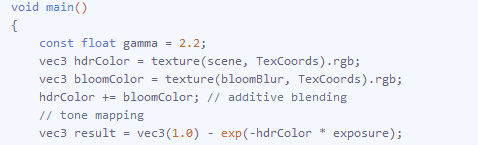
得到一个HDR纹理后，我们用提取出来的亮区纹理填充一个帧缓冲，然后对其模糊处理10次

（HDR纹理是为了让在泛光计算中高斯模糊计算出的亮度可以超过1.0，最后得出的效果更真实

）

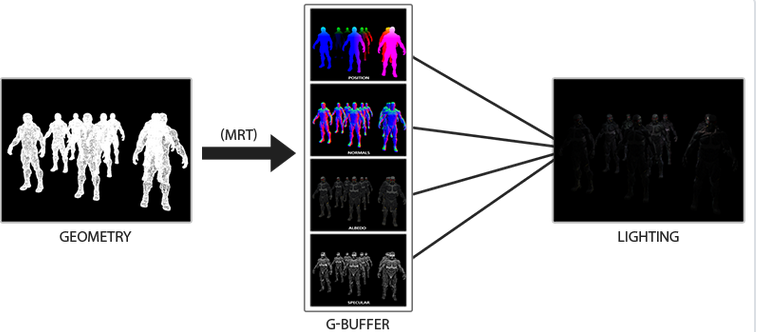


最后一步：混合纹理



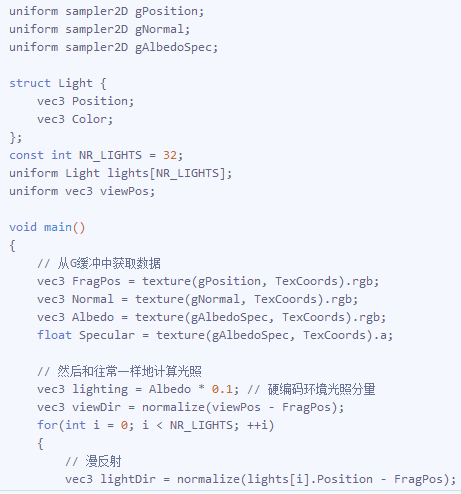
**延迟着色法：**

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。



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。

对于这个光照处理阶段，我们将会渲染一个2D全屏的方形(有一点像后期处理效果)并且在每个像素上运行一个昂贵的光照片段着色器。



延迟着色法的其中一个缺点就是它不能进行[混合](https://learnopengl-cn.github.io/04 Advanced OpenGL/03 Blending/)(Blending)，因为G缓冲中所有的数据都是从一个单独的片段中来的，而混合需要对多个片段的组合进行操作。延迟着色法另外一个缺点就是它迫使你对大部分场景的光照使用相同的光照算法，你可以通过包含更多关于材质的数据到G缓冲中来减轻这一缺点。

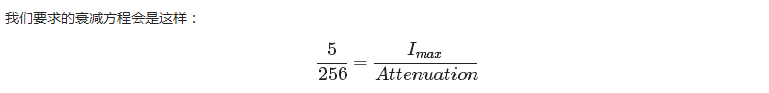
结合延迟渲染和正向渲染



我们需要做的就是首先复制出在几何渲染阶段中储存的深度信息，并输出到默认的帧缓冲的深度缓冲，然后我们才渲染光立方体。这样之后只有当它在之前渲染过的几何体上方的时候，光立方体的片段才会被渲染出来。我们可以使用glBlitFramebuffer复制一个帧缓冲的内容到另一个帧缓冲中，这个函数我们也在[抗锯齿](http://learnopengl-cn.readthedocs.org/zh/latest/04 Advanced OpenGL/11 Anti Aliasing/)的教程中使用过，用来还原多重采样的帧缓冲。glBlitFramebuffer这个函数允许我们复制一个用户定义的帧缓冲区域到另一个用户定义的帧缓冲区域。

更多的光源：

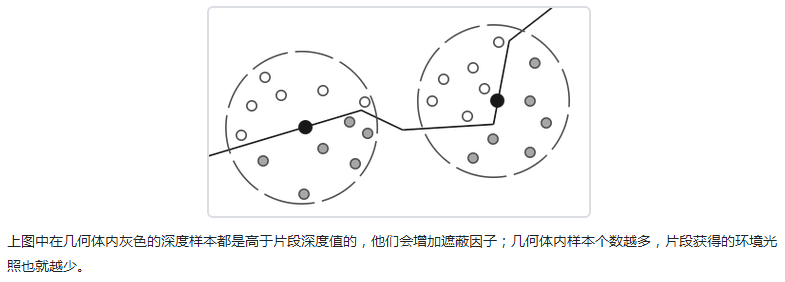
然而，延迟渲染它本身并不能支持非常大量的光源，因为我们仍然必须要对场景中每一个光源计算每一个片段的光照分量。真正让大量光源成为可能的是我们能够对延迟渲染管线引用的一个非常棒的优化：****光体积(Light Volumes)****



**SSAO：**

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。

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。



片段着色器通常是相当费时的所以我们应该尽量避免运行它们。

### 灰度

人眼趋向于对绿色更敏感，对蓝色感知比较弱，所以为了获得更精确的符合人体物理的结果，我们需要使用加权通道

