**一、开始**

进入到执行文件所在目录 执行命令 a5.exe ../../

将父目录作为第一个命令行参数 程序会自动从data/sponza\_low目录中加载场景数据

**二、需求总结**

1. 第一个任务，使用vertexRecorder类来绘制网格。

2. 纹理贴图：

创建OpenGL纹理对象

指定绘制时使用哪种纹理

修改着色器代码来从纹理坐标中采样（UV）

3. 阴影贴图

学习如何渲染一个纹理，而不是直接渲染到屏幕上

指示OpenGL在下一个draw调用中读取此纹理。

在片段着色器中实现阴影贴图

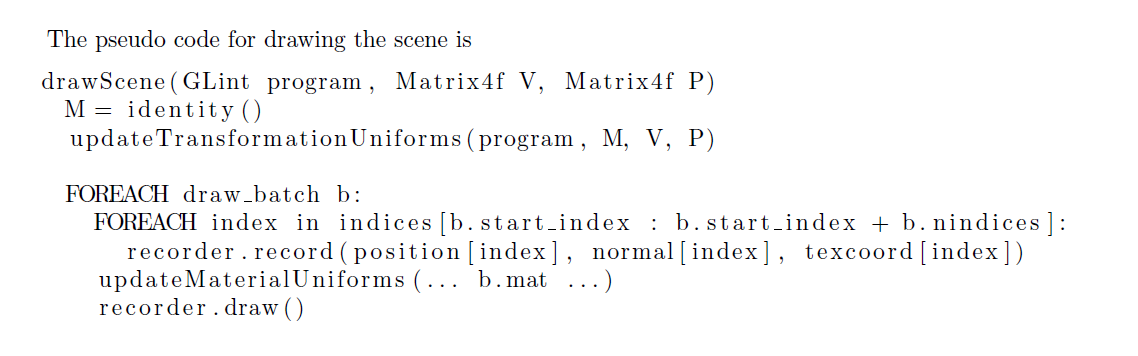
**三、绘制网格**

在**drawScene**方法中绘制网格，稍后我们将从点光源的视角来渲染一遍同样的场景，因此我们传入glsl程序以及灯光和摄影机矩阵（?）。

我们已经实现了一个简单的.obj解析器，**objparser.cpp**，它从场景中读取数据给顶点和索引数组。

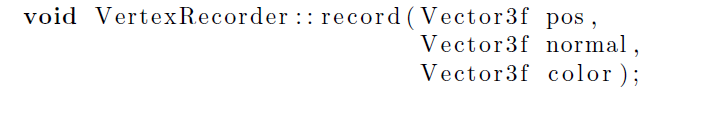
在这个文件里，我们存储了一些draws\_batches，每一个batch都应该在绘制调用前由单独的VertexRecoder::draw()方法来调用，确保更新材质的uniforms变量到由draw\_batch指定的材质属性中。你可以使用updateMaterialUniforms()方法来更新当前程序的材质。

场景绘制的伪代码：



注意：索引到索引：这个 draw\_batch结构存储的是batch里面的第一个顶点的start\_index，这不是位置数组的索引，这是一个指向索引数组的索引，要获得一个draw\_batch的第一个顶点位置，应该调用scene.position[scene.indices[batch.start\_index]]。

我们再次提供了VertexRecorder类，使用三个参数的记录方法：

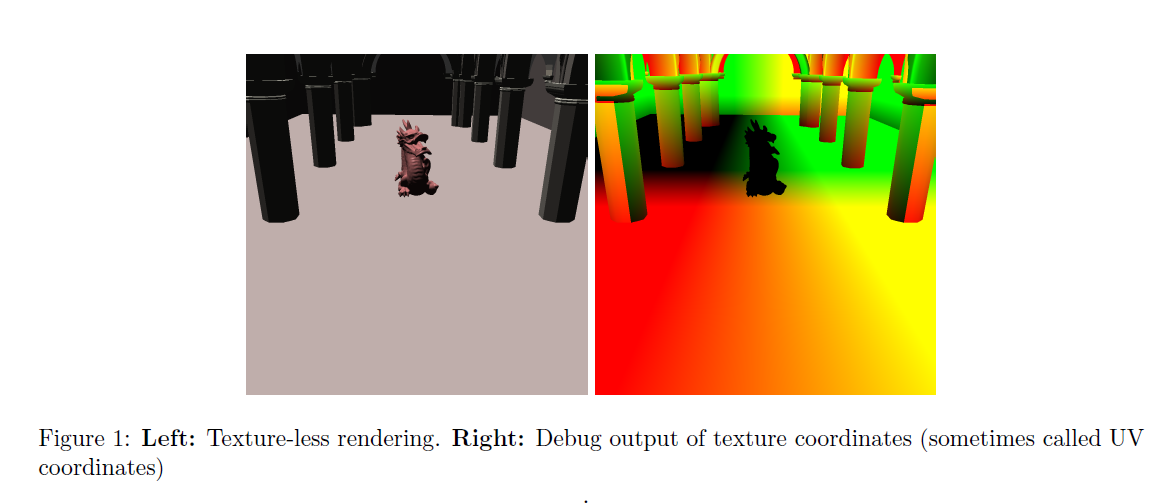


并且传递UV坐标作为颜色属性的前两个元素(color.z可以是空的)。

**可视化纹理坐标**

要确保纹理坐标被正确地传递到了片段着色器上，在CPU代码上，可以打印语句来看纹理坐标是否不等于0，但是在OpenGL着色器中没有打印语句，所以更常用的调试着色器的方法，就是绘制调试信息。

修改fragmentsshader\_dirlight.glsl文件（可以在程序运行的时候修改），这样它可以通过存储在var\_Color.xy上的纹理坐标给网格上色。

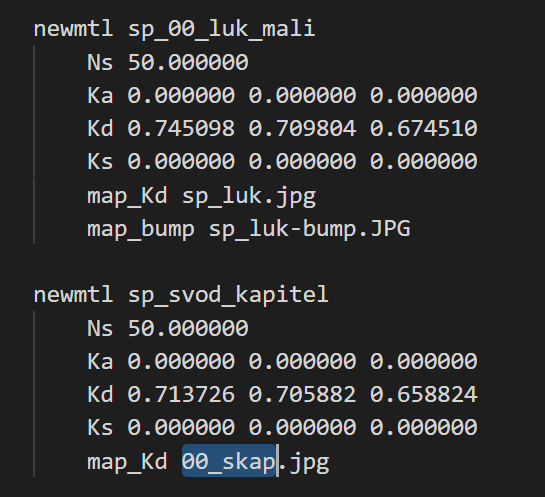


理解部分：

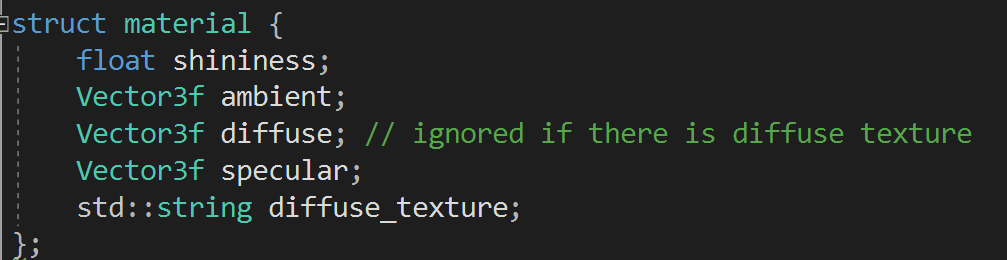
在objparser.cpp文件中：

1、材质及纹理的解析

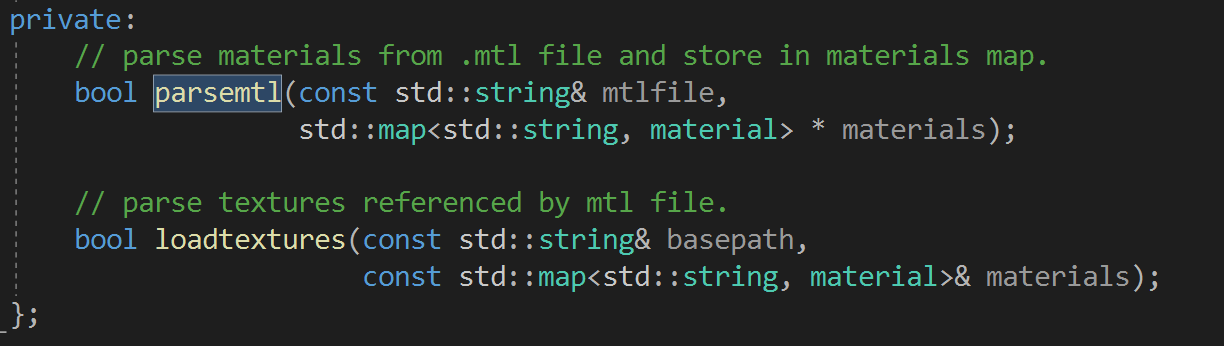
parsemtl方法，解析.mtl材质数据：



分别对应下面这些定义的数据字段 map\_bump暂时忽略

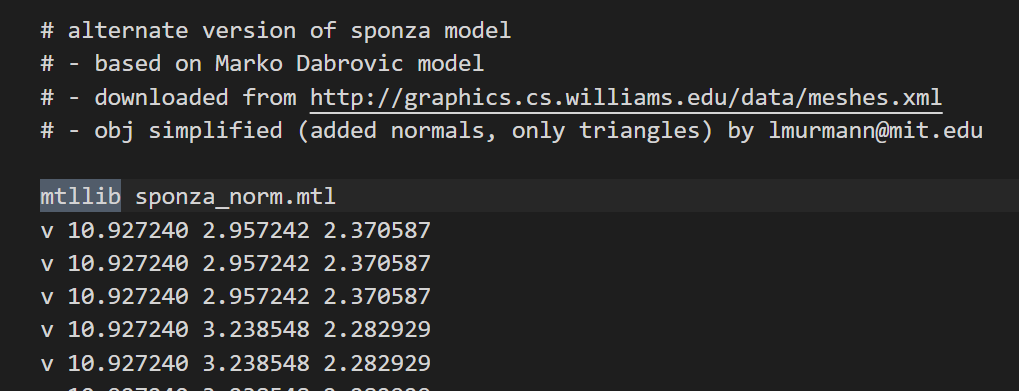


解析完之后，将这些material信息存入一个map里面；

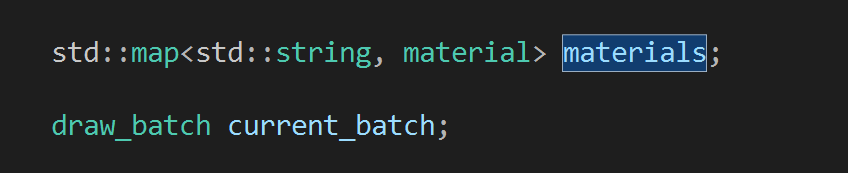


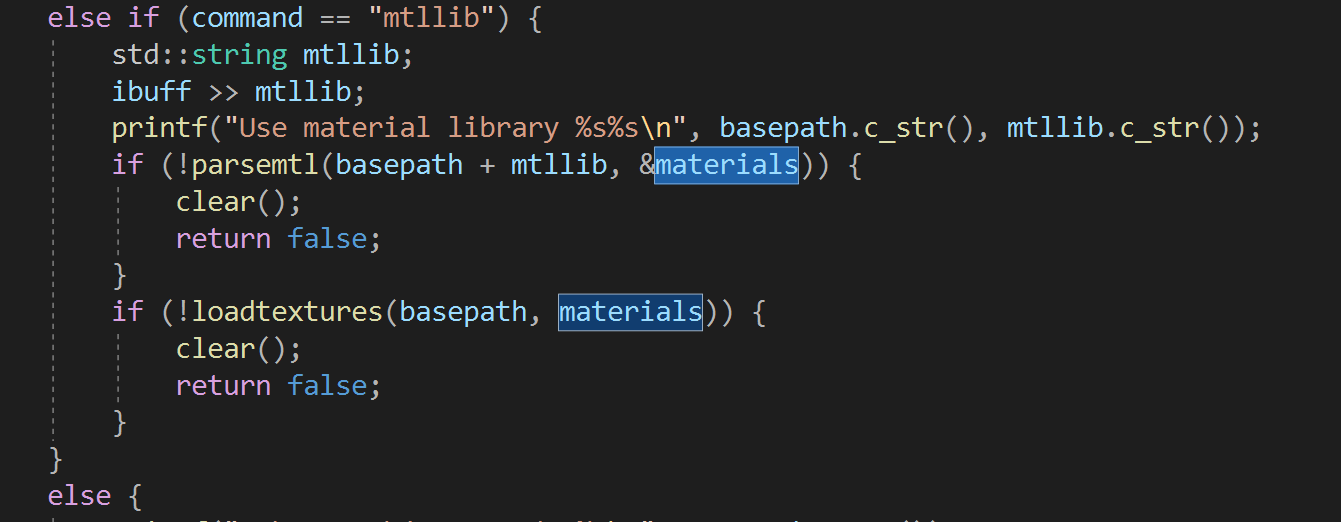
然后loadTextures方法，会遍历这个material的map，然后根据每一个material上面存储的texture名字找到纹理文件，解析处像素数据，存入rgbImage im的data中，再存入textures映射表里。

Obj文件解析以及Batches的理解：

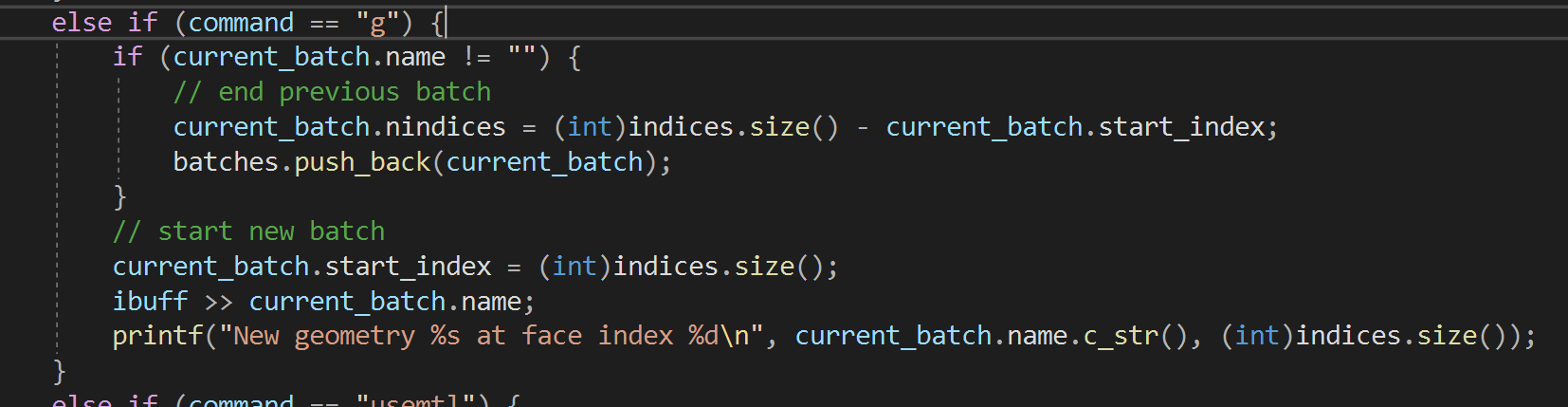


首先在文件首部，表明了材质文件在哪里读取，xxx.mtl（里面有很多种材质的数据），然后解析器会读取该mtl文件并解析到materials这个map里面：

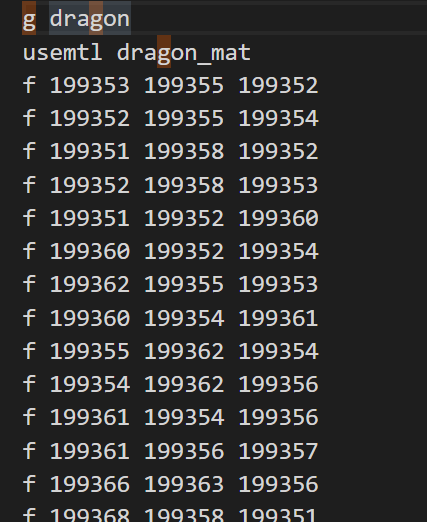


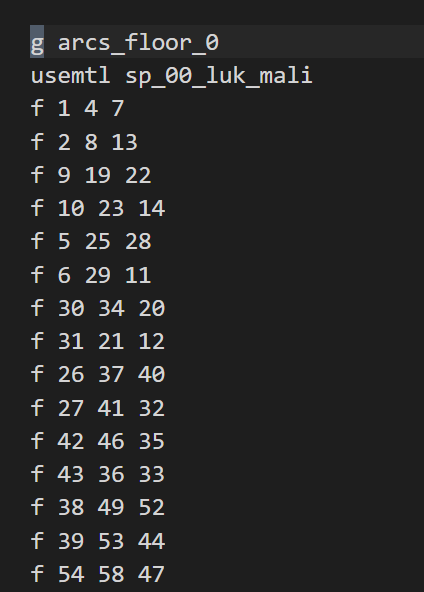


2. batches数据的解析：



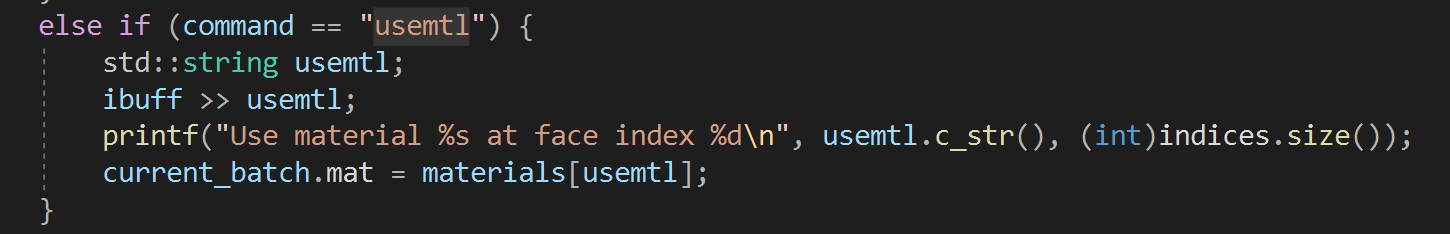
可以这么理解batches，（表示有相同材质的一系列顶点，作为一个批次进行渲染）：





如上图所示，当读取到第一批次，dragon的顶点索引时，由于此时index.size是0，所以第一批次的start\_index就是0，并保存当前batch的名字，当读取完了这一批次的顶点索引，走到下一批次g arcs\_floor\_0的时候，就会计算上一个批次读取了多少个顶点(索引)batch.nindices，并把这个batch存入到batches（vector)里面。

此外，每一个批次的声明，都会存储一个对应的材质，即usemtl xxx，然后就会在material这个map里面根据名字查找material，再存到batches.mat里面：

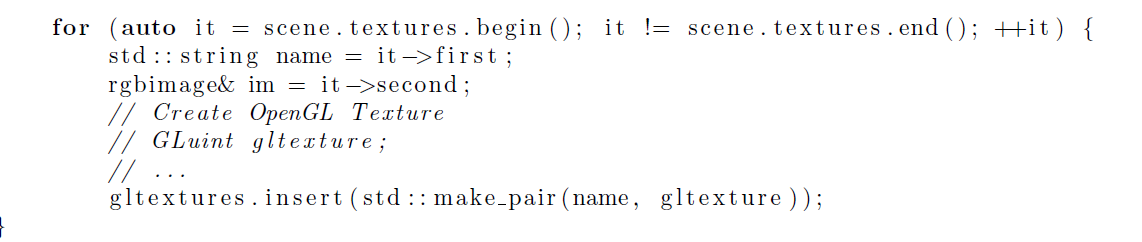


**四、纹理**

在上面的步骤中，你验证了纹理坐标是否被正确地传递到片段着色器中，现在你将要创建OpenGL的纹理对象，从纹理中读取漫反射材质的颜色，而不是每个对象的Uniforms。

你将要创建两个方法,loadTextures和freeTextures，在main主循环开始之前调用loadTextures()方法，在主循环结束之后调用freeTextures方法。

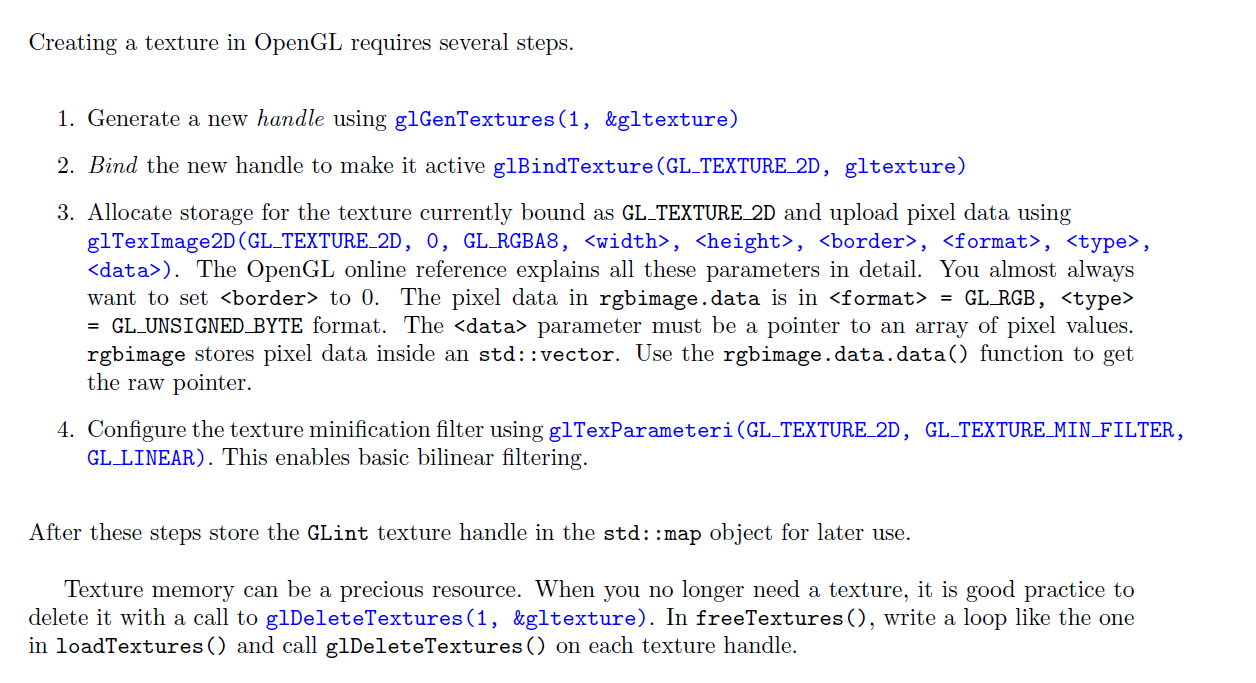
在objparser中，我们已经从磁盘中加载好了JEPG文件，他们被存储在一个map<string, rgbimage>里面，你可以这样遍历整个Map：



你需要为每个texture创建OpenGL对象，并且插入这个纹理对象插入到一个全局的map对象中。

**OpenGL纹理**

创建OpenGL纹理的步骤：



可以参考LearnOpenGL纹理那章的内容

在这些步骤之后，就可以将GLint纹理句柄存储在Map中以供后续使用。

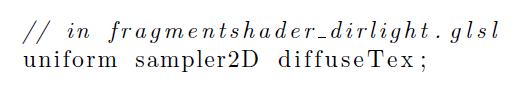
当不再需要一个纹理，就应该调用glDeleteTextures方法将纹理内存释放掉。

在绘制前绑定纹理：

在drawScene方法中，确保在调用recorder.draw方法前绑定正确的纹理，从map中寻找对应的纹理句柄，并且调用glBindTexture方法绑定纹理（GL\_TEXTURE\_2D）

在片段着色器中采样纹理

在片段着色器中我们使用一种特殊的uniform变量"sampler"来采样纹理。



在着色器的主函数中，使用GLSL方法texture()来从纹理中读取漫反射值，sampler uniform输入(?)第一个参数，2D纹理坐标输入第二个参数。

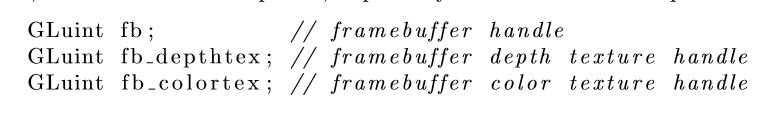
**五、阴影贴图**

阴影贴图是计算机图形学中的一个多过程(pass)算法的例子：在第一个pass中，我们从光源的位置渲染到一个深度纹理。在第二个pass中，我们使用存储在该纹理中的深度值来确定场景点的光源可见性。

默认情况下，OpenGL直接渲染到屏幕上。通过一些额外的配置，我们可以让OpenGL渲染到一个纹理。准确地说，我们将创建两种纹理，一种用于颜色，另一种用于深度，并将两者结合成一个帧缓冲对象。

阴影映射算法有多个步骤，如果出了问题，通常哪里引入了错误会不太明显。为了增加成功的机会，我们将实现算法的第一步（渲染到纹理），然后将中间结果可视化。一旦我们确定第一步是正确的，我们将继续进行第二步的实际阴影计算。

你将创建一个loadFraembuffer()和freeFraembuffer()方法，在主循环开始前和结束后分别调用。创建3个GLint OpenGL句柄作为全局变量。



在loadFramebuffer（）内，通过调用glGenTextures（）请求每个纹理的有效句柄。将颜色纹理绑定到GL texture 2D。使用glTexImage2D为其分配空间。这一次，不必传递数据指针，因为纹理将通过渲染填充到其中。

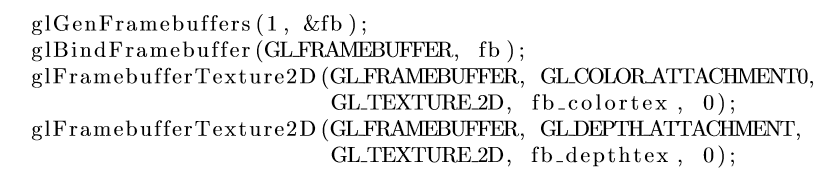
对于颜色纹理，使用GL RGBA8作为内部格式，width==height==4096作为大小，GL RGB作为格式，GL UNSIGNED BYTE作为数据类型。将nullptr作为数据指针传递。在glTexImage2D（）之后，需要为此纹理配置纹理插值，以便将其渲染进去。使用glTexParameter（）设置纹理插值设置，如下所示

clipboard.png

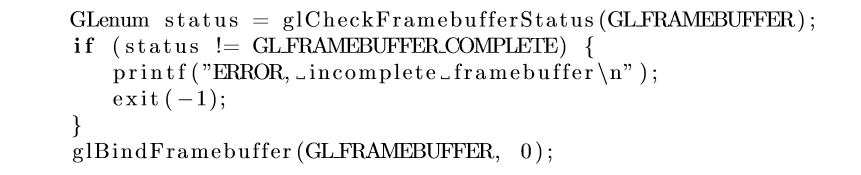
GL NEAREST表示“最近邻插值”

现在，将深度纹理绑定到GL texture 2D。深度纹理有一个特殊的内部格式：GL depth COMPONENT32F。作为<format>参数，使用GL depth COMPONENT（不带32F）和GL FLOAT作为数据类型。确保设置最接近深度的最小纹理模式。

接下来，使用glGenFramebuffers（）请求framebuffer的句柄。与纹理一样，必须先绑定当前帧缓冲对象，然后才能进一步配置它。绑定后，将指定这两种纹理分别作为颜色和深度缓冲的数据存储：



Since the code we wrote in this section is quite intricate, we double-check that our framebuﬀer is conﬁgured correctly:



在最后一行，我们设置激活的帧缓冲对象为默认帧缓冲0.

释放帧缓冲，使用glDeleteTextures（）和glDeleteFramebuffers释放在LoadFrameBuffer（）中分配的内存。

光的视图矩阵和投影矩阵

实现两个方法getLightView和getLightProjection方法，投影矩阵使用Matrix4f::orthographicProjection()来获得正交投影矩阵，参数要基于场景几何以及虚拟相机将要形成的视锥体来设置最近最远裁剪平面。

光的视图矩阵将要设置为沿着light\_dir看向场景，它到场景的距离是任意的，但必须要和投影矩阵设置的远近平面相符。

使用 Matrix4f::lookAt(eye, center, up)方法来获得视图矩阵，注意三点：

1. center应该是场景的中心，例如，龙模型的位置。

2. eye应该再场景之上（记住单位Y向量指向上）。从眼睛到中心的连线应该和光的方向平行。

3. up必须要和Light\_dir垂直

理解：

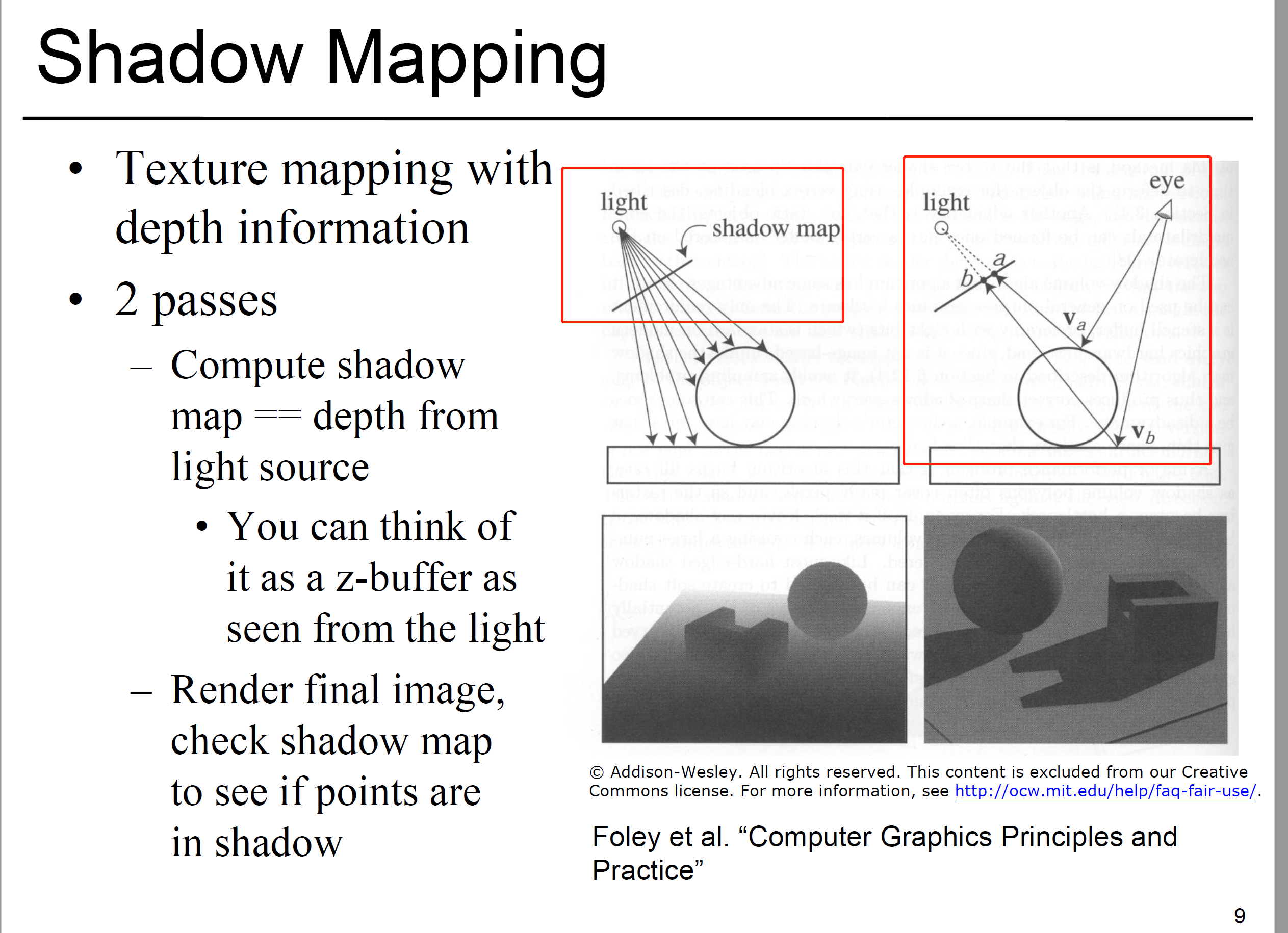
这部分要自己实现获取视图和投影矩阵的方法，已知参数是光源的方向。此时应该需要查看场景的数据(.obj)，判断场景的坐标范围，从而假定一个eye的位置(即光源位置)，和投影矩阵的宽、高以及远近裁剪平面的距离。

1. 这里根据场景的数据，暂定眼睛到中心的距离为20，宽、高定为50，近平面1，远平面100

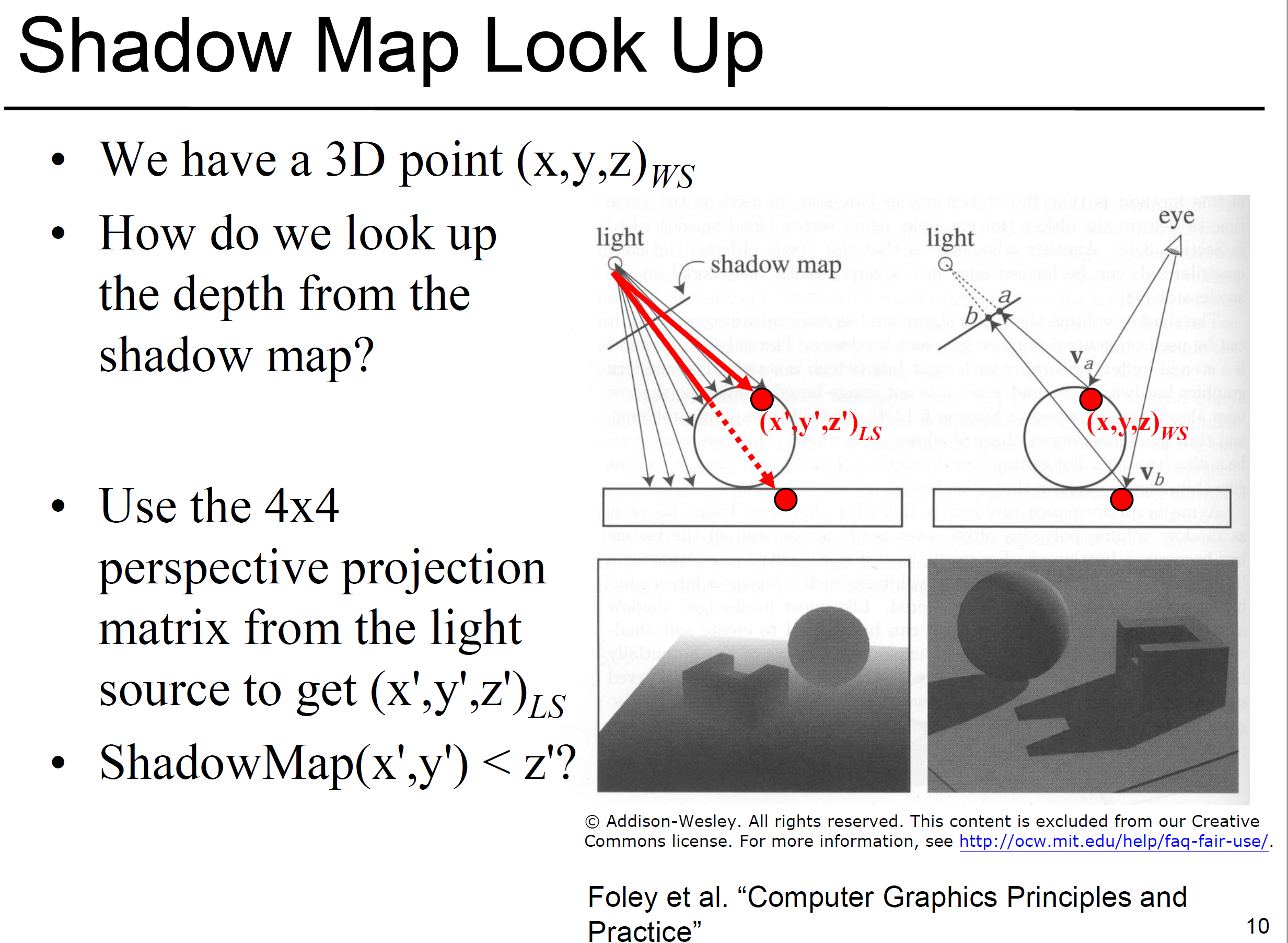
2. up向量需要进行两次叉乘计算获得，参考ray-tracing in one weekend。

\*这个Up向量不太理解，

最后一步，shadowMap的实现：



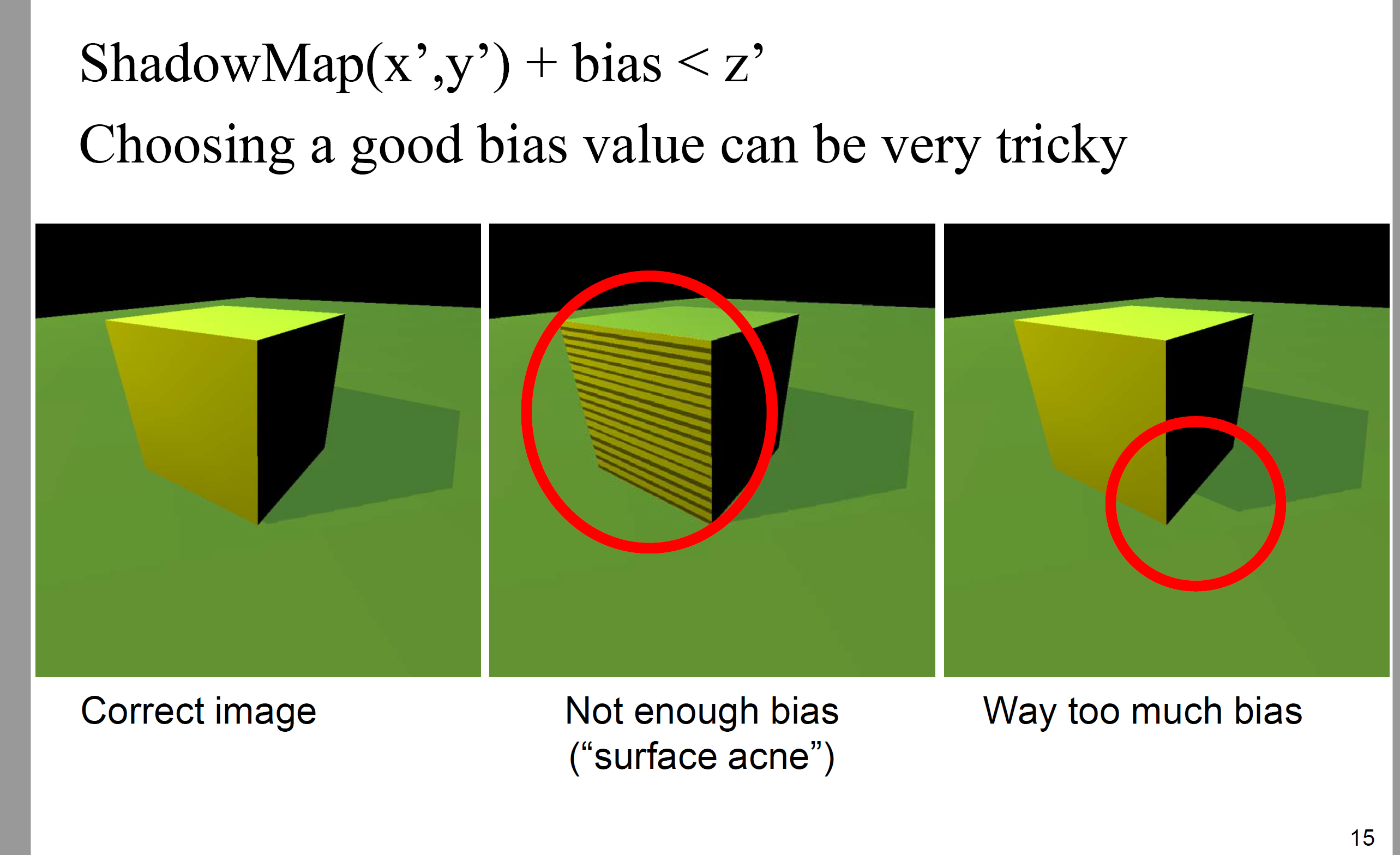
当前已经有一个深度(阴影)贴图了，那么在最终渲染图片的时候，对于视线看到的一个点，查找它在阴影贴图中的值，如果它自身的深度值(z)(相对于视线方向，到光源的距离），比阴影贴图中的z值大，说明它在阴影贴图中不是最近的点，会产生阴影。



目前我们已经有了一个世界坐标中的点p(x,y,z)，如何找到深度贴图中该点对应的深度值呢？

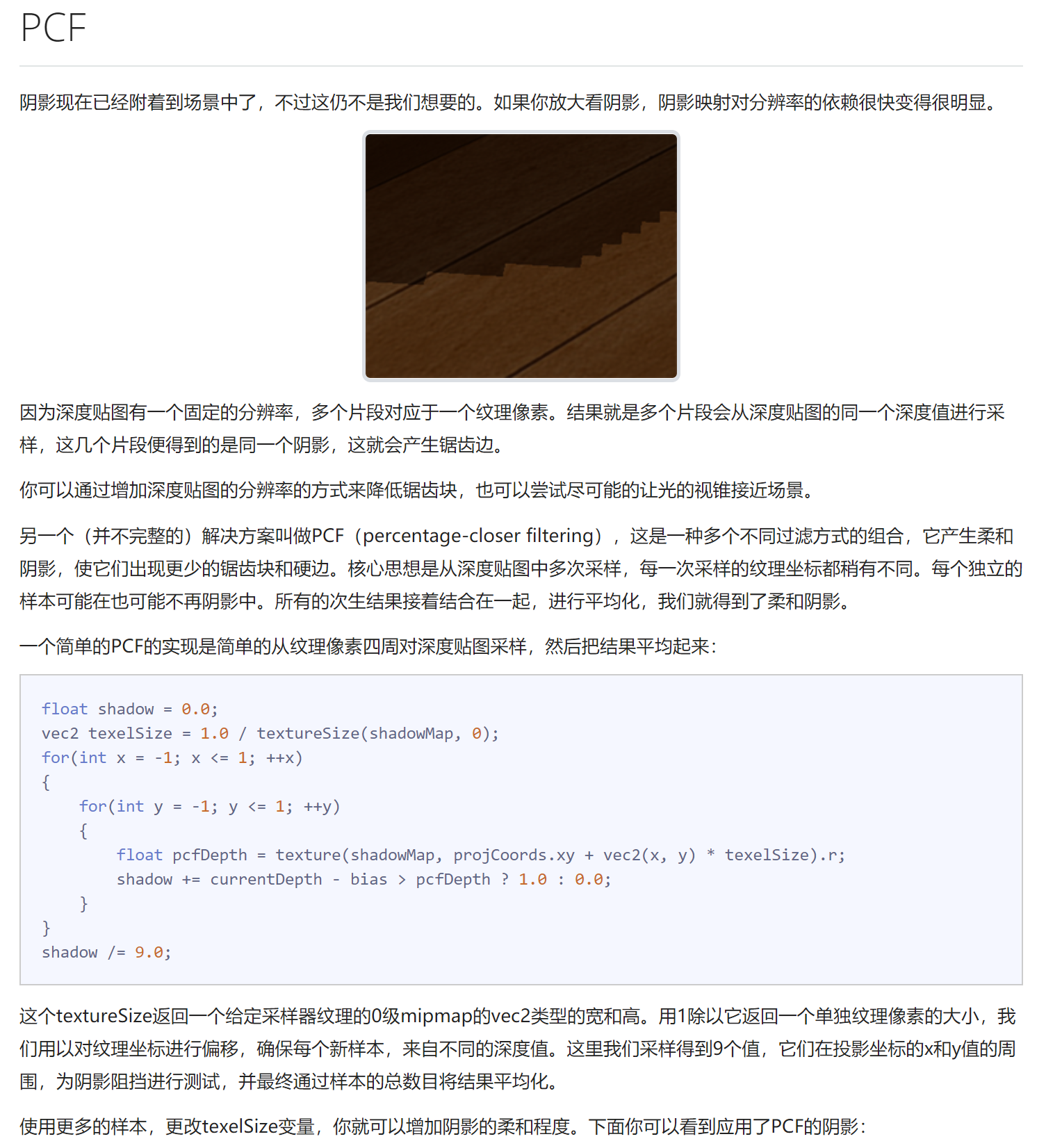
答案是使用4x4的投影裁剪矩阵(应该是观察投影矩阵吧），将该点转换到以光源作为视角的观察坐标，然后再根据这个x'和y'去查阴影贴图的值，再和z'比较即可。这部分要在着色器中实现。

并且需要一个bias作为误差修正。



\*记得最后在片段着色器中计算出来的pos\_in\_light，光源空间上的坐标，是[-1,1]区间的，需要映射到[0,1]区间才能从纹理中采样。

使用PCF技术进行抗锯齿处理：



ps: 遇到问题的时候，可以参考LearnOpenGL的相关内容。