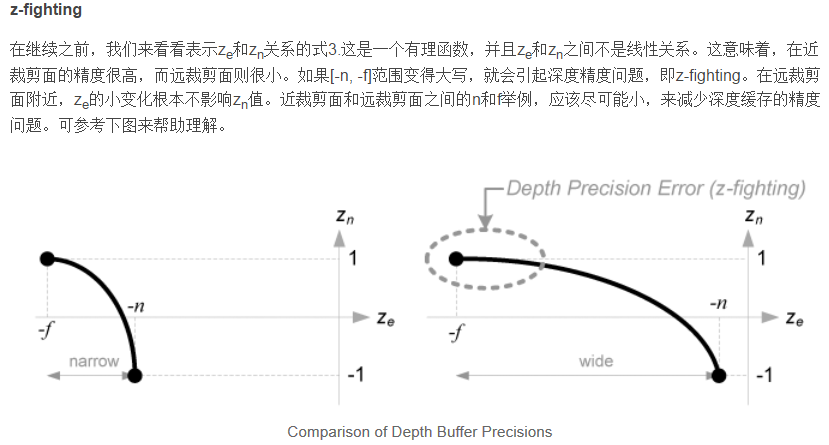
**一个问题:Z缓冲冲突**

由于深度缓冲的精度问题，当两个相聚特别近的顶点在投影映射后会出现两者渲染遮挡关系错误的情况，



解决该问题的方法：

<http://blog.csdn.net/iaccepted/article/details/44873517>

opengl中遮挡关系是通过z缓冲区+深度测试来实现的，而z缓冲区一般都有位数限制，通常为24位，也有32位。

这样在进行遮挡查询的时候，如果两个面距离非常近的话，那么这两个面的z值就可能会由于非常小而产生交替出现的现象，称为z-fighting，这就严重影响渲染质量。

通常的解决方法：

1.最常用的就是使用glPolygonOffset函数，对于两个非常近的面或者model我们可以将其分开渲染，先渲染一个然后通过glPolygonOffset函数设置一个偏移之后再渲染另一个，

   这样的结果就是在一个的深度值写入深度缓冲之后，另一个可以加一个偏移量来保证在深度测试的时候两者的深度值达到明显区别的程度，从而正确的绘制两个object。

2.通常我们在计算depth的时候并不使用线性计算，即计算出的depth并非是点的z坐标线性转换而来，而是使用非线性变换，Fdespth = （1/z - 1/near) / (1/far - 1/near),这样就能保证在离近切面近的部分depth值更精确，当然我们也很容易推出，当把near值适当增大时，可以更好的保证距离较远处的精度值，所以可以通过增大near值来解决，但是这种方法对于解决单模型渲染还是效果不错的，但是对于多模型渲染也会有问题。

3.提升硬件精度，主要是提升z缓冲区的位数来达到增加精度的目的，这对于要求较高的场合确实是很好的选择。

深度测试是在片元着色器运行之后（也是在模板测试之后），在屏幕空间中完成的。可以在片元着色器中通过glsl的内建变量gl\_FragCoord来获取。gl\_FragCoord的x和y元素代表fragment的屏幕空间坐标（0,0是左下角）。gl\_FragCoord也包含一个z元素。这个元素是fragment的实际深度值，z值就是用来和深度缓冲内容对比的那个值。

注意：如今，大多数gpu都支持一个硬件功能，这个功能叫做前置深度测试（early depth testing）。前置深度测试允许在片元着色器运行之前运行深度测试。显然，一个fragment如果运远不会被显示的话（它在其他物体后面），我们就可以提前丢弃这个fragment，像素说择期很奢侈，所以无论何处我们都要避免它们变得更复杂，像素着色器对前置深度测试的一个限制，是你不能对fragment的深度值进行写操作，如果一个像素着色器对深度值进行了写入操作，那么前置深度测试将不可用，opengl不可能提前知道深度值。

传统的深度测试是发生在片元着色器之后，因此仅仅依靠深度测试并不能加快多少渲染速度，而提前深度测试则发生在光栅化后，片元着色器之前，提前深度测试会对深度值进行比较，如果测试通过（Z-func），则执行片元着色器，否则跳过此片元，不过要注意的是，在片元着色器中不能修改深度值，否则提前深度测试会被禁用，这样，在整个流水线中，深度比较发生了2次，一次是提前深度测试，一次是传统的深度测试。

光栅化--片元着色器--传统深度测试

光栅化--提前深度测试--片元着色器--传统深度测试

**为什么需要进行两次深度测试？**

http://www.gamedev.net/topic/622047-early-z/

z-prepass: enable depth test, enable depth write, disable color write.  
second pass: enable depth test, disable depth write, enable color write.

深度缓冲的精度：

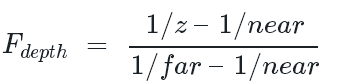
深度缓冲区的深度值范围在0.0-1.0之间，我们如何进行映射呢？

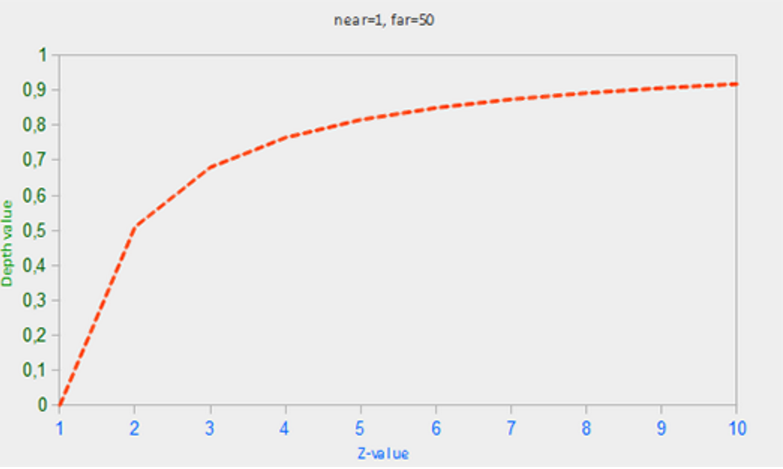
1）先行

depth = (z - near) / (far - near);

注意：这种方式，当物体接近的时候，所有方程的深度值接近0.0f，当物体接近远平面时深度值接近1.0f

然而，在实践中，这样的线性深度缓冲几乎是不会用的，为了获得正确的投影属性，使用的是一种非线性的深度方程，它与1/z成比例。当z越小越精确，当z越大越不精确。





重点记住：深度缓冲中的数值在屏幕空间中不是线性分布的（它们在应用投影矩阵之前的试图空间中是线性的）。一个深度缓冲中的数值0.5不代表物体的z值在中间；这个顶点的z值非常接近近平面。

将深度线性化：

float linearizeDepth(float depth)

{

float near = 0.1f;

float far = 500.0f;

float z = depth \* 2.0f - 1.0f; // 从0 - 1变为-1 - 1，即变为NDC坐标

return (2.0f \* near) / (far + near - z \* (far - near)); // 反向得到视觉空间的z值，并除以far进行归一化。注意：NDC是左手坐标系，视觉坐标是右手坐标系

}

gl\_FragColor = vec4(vec3(depth), 1.0f);

float depth = linearizeDepth(gl\_FragCoord.z);

