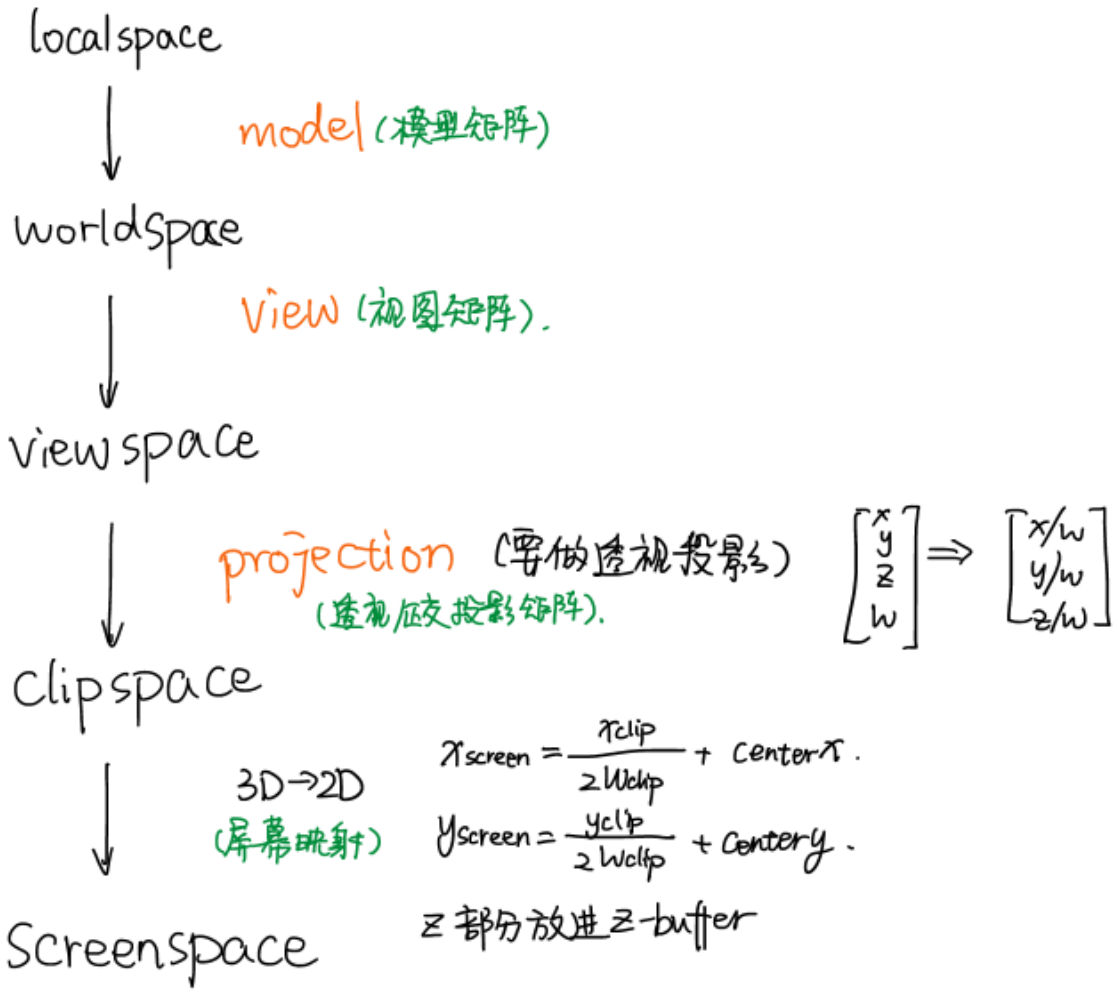
图形学背题大纲

1.坐标系转换图

模型空间->世界空间->摄像机空间->裁剪空间->屏幕空间



2.三个重要的空间变化矩阵

MVP矩阵。（model transform、view transform、projection transform）

3.渲染管线流程

广义上的图像渲染

1. **应用程序阶段**

   主要是CPU与内存打交道，例如碰撞检测，计算好的数据（顶点坐标、法向量、纹理坐标、纹理）就会通过数据总线传给图形硬件 。主要任务为在应用程序阶段的末端，将需要在屏幕上（具体形式取决于具体输入设备）显示出来绘制的几何体（也就是绘制图元，rendering primitives，如点、线、矩形等）输入到绘制管线的下一个阶段。

2. **几何阶段**

   几何阶段主要负责大部分多边形操作和顶点操作。可以将这个阶段进一步划分成如下几个功能阶段：

   - 模型视点变换 Model & View Transform
   - 顶点着色 Vertex Shading
   - 投影 Projection
   - 裁剪 Clipping
   - 屏幕映射 Screen Mapping

3. **光栅化阶段**

给定经过变换和投影之后的顶点，颜色以及纹理坐标（均来自于几何阶段），给每个像素（Pixel）正确配色，以便正确绘制整幅图像。即从二维顶点所处的屏幕空间（所有顶点都包含Z值即深度值，及各种与相关的着色信息）到屏幕上的像素的转换。

与几何阶段相似，该阶段细分为几个功能阶段：

- 三角形设定（Triangle Setup）阶段
- 三角形遍历（Triangle Traversal）阶段
- 像素着色（Pixel Shading）阶段

  主要任务：计算所有需逐像素操作的过程。例如**贴图纹理采样**
- 融合（Merging）阶段

  主要任务：合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色、可见性问题（alpha通道、模板缓冲器、帧缓冲器等）

GPU渲染管线：

1. **几何阶段：**
   - 顶点着色器

     顶点着色器可以对每个顶点进行诸如变换和变形在内的很多操作，提供了修改/创建/忽略顶点相关属性的功能，这些顶点属性包括颜色、法线、纹理坐标和位置。顶点着色器的必须完成的任务是将顶点从模型空间转换到齐次裁剪空间。
   - 几何着色器

     几何着色器是可选的，完全可编程的阶段，主要对图元（点、线、三角形）的顶点进行操作。几何着色器接收顶点着色器的输出作为输入，通过高效的几何运算，将数据输出，数据随后经过几何阶段和光栅化阶段的其他处理后，会发送给片段着色器。
   - 裁剪
   - 屏幕映射
2. **光栅化阶段：**
   - 三角形设定
   - 三角形遍历
   - 像素着色器

     常常又称为片断着色器，片元着色器(Fragment Shader，OpenGL中的叫法)，是完全可编程的阶段，主要作用是进行像素的处理，让复杂的着色方程在每一个像素上执行

     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
   - 合并阶段

     其除了进行合并操作，还分管颜色修改（Color Modifying），Z缓冲（Z-buffer），混合（Blend），模板（Stencil）和相关缓存的处理。

     各种测试的顺序：像素所有者测试（PixelOwnershipTest）、裁剪测试（ScissorTest）【前两者是OpenGL内部实现的】、模板测试（StencilTest）和深度测试（DepthTest）、混合（Blending）、抖动（Dithering）

     模板测试--》alpha测试--》深度测试--》合并图元

4.各种着色器的作用

见上

5.光栅化图形学

1. DDA算法

   dx>dy时，直线Y=kx+b，当 ▵x=1时，▵y=k。

   dx<dy时，直线x=y/k-b/k，当 ▵y=1时，▵x=1/k。

2. Bresenham算法

   见OneNote

6.深度/模板测试

深度测试

深度测试在片段着色器运行之后(并且模板测试运行之后)在屏幕空间中执行的。

一旦启用深度测试，如果片段通过深度测试，OpenGL自动在深度缓冲区存储片段的 z 值，如果深度测试失败，那么相应地丢弃该片段。在某些情况下我们需要进行深度测试并相应地丢弃片段，但我们不希望更新深度缓冲区，基本上，可以使用一个只读的深度缓冲区。

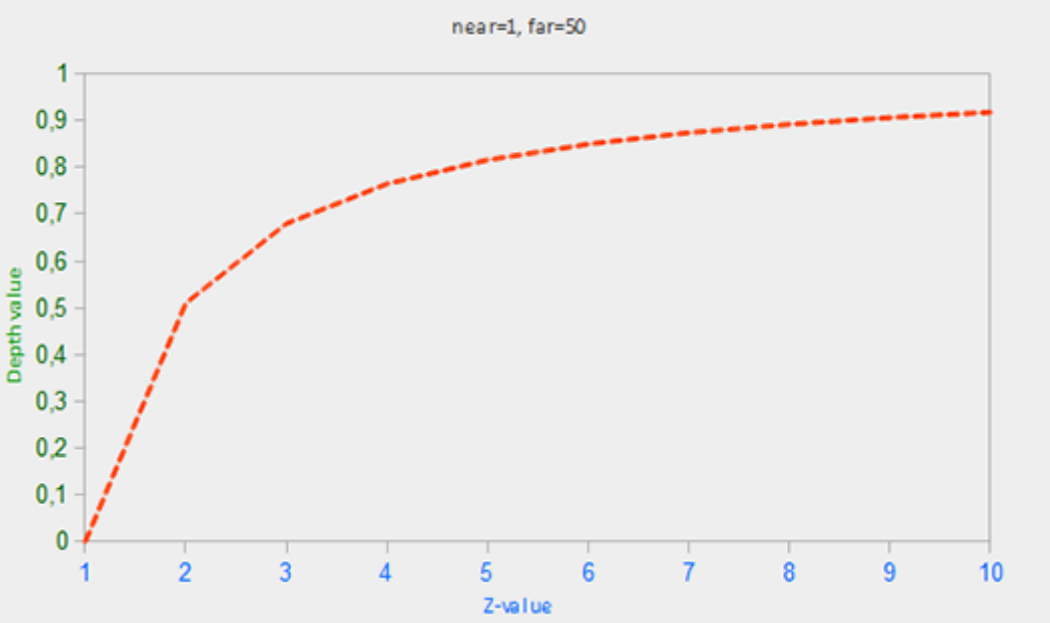
我们因此需要一些方法来转换这些视图空间 z 值到 [0，1] 的范围内，方法之一就是线性将它们转换为 [0，1] 范围内。下面的 (线性) 方程把 z 值转换为 0.0 和 1.0 之间的值 :

$$F_{depth} = \frac{z - near}{far - near}$$

然而，在实践中是几乎从来不使用这样的线性深度缓冲区。正确的投影特性的非线性深度方程是和1/z成正比的 。这样基本上做的是在z很近是的高精度和 z 很远的时候的低精度。用几秒钟想一想: 我们真的需要让1000单位远的物体和只有1单位远的物体的深度值有相同的精度吗?线性方程没有考虑这一点。

由于非线性函数是和 1/z 成正比，例如1.0 和 2.0 之间的 z 值，将变为 1.0 到 0.5之间， 这样在z非常小的时候给了我们很高的精度。50.0 和 100.0 之间的 Z 值将只占 2%的浮点数的精度，这正是我们想要的。这类方程，也需要近和远距离考虑，下面给出:

$$F_{depth} = \frac{1/z - 1/near}{1/far - 1/near}$$



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)**

防止深度冲突的方法：1.让物体之间不要离得太近；2.尽可能把近平面设置得远一些

模板测试

片段处理器->模板测试（模板缓冲）->深度测试（深度缓冲）

模板缓冲先清空模板缓冲设置所有片段的模板值为0，然后开启矩形片段用1填充。场景中的模板值为1的那些片段才会被渲染（其他的都被丢弃）。

Color buffer　　Stencil buffer　　After stencil test

无论我们在渲染哪里的片段，模板缓冲操作都允许我们把模板缓冲设置为一个特定值。改变模板缓冲的内容实际上就是对模板缓冲进行写入。在同一次（或接下来的）渲染迭代我们可以读取这些值来决定丢弃还是保留这些片段。当使用模板缓冲的时候，你可以随心所欲，但是需要遵守下面的原则：

- 开启模板缓冲写入。
- 渲染物体，更新模板缓冲。
- 关闭模板缓冲写入。
- 渲染（其他）物体，这次基于模板缓冲内容丢弃特定片段。

使用模板缓冲我们可以基于场景中已经绘制的片段，来决定是否丢弃特定的片段。

7.透明度测试/渲染半透明或透明物体

1. 对于全透明物体，我们可以简单的选择丢弃像素而不是混合
2. 渲染半透明物体时需要开启blending（），但是为了避免渲染错误，需为物体进行排序，先绘制最远的物体，最后绘制最近的物体。普通的无混合的物体仍然可以使用深度缓冲正常绘制，所以不必排序，但是必须保证在透明物体绘制完成前绘制完毕。

- 透明度测试（Alpha Test）全透明/不透明：不需关闭深度写入（ZWrite），根据透明度来舍弃片元。
- 透明度混合（Alpha Blending）可半透明：关闭深度写入（不关闭深度测试，还是会比较j，从而决定是否混合）用片元的透明度作为混合因子，与颜色缓冲中的颜色值进行混合。

对不透明（opaque）物体，由于**深度缓冲（depth buffer，z-buffer），**不考虑渲染顺序也能得到正确的排序效果。

关闭深度写入原因：否则测试成功就会写入颜色缓冲区，不进行颜色混合。

透明与不透明物体之间：**不透明物体渲染完之后再渲染半透明物体。**

例子：不透明物体开启了深度测试，此时深度缓冲中没有任何有效数据，因此B首先会写入颜色缓冲和深度缓冲。随后渲染A，透明物体深度测试通过，用A的透明度和颜色缓冲中的B的颜色混合。

半透明物体之间：从后往前

B正常写入颜色缓冲，A与颜色缓冲中的B颜色进行混合。

一些深度排序算法

- 深度缓存
- 画家算法
- 加权平均值算法
- 深度剥离算法(https://www.sardinefish.com/blog/?pid=348)

8.GLSL着色器程序创建的细节，uniform in out

1. 着色器的开头总是要声明版本，接着是输入和输出变量、uniform和main函数。每个着色器的入口点都是main函数，在这个函数中我们处理所有的输入变量，并将结果输出到输出变量中。
2. 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。
3. 顶点着色器：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。`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。

```
#version 330 core
layout (location = 0) in vec3 position; // position变量的属性位置值为0

out vec4 vertexColor; // 为片段着色器指定一个颜色输出

void main()
{
    gl_Position = vec4(position, 1.0); // 注意我们如何把一个vec3作为vec4的构造器的参数
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // 把输出变量设置为暗红色
}
```

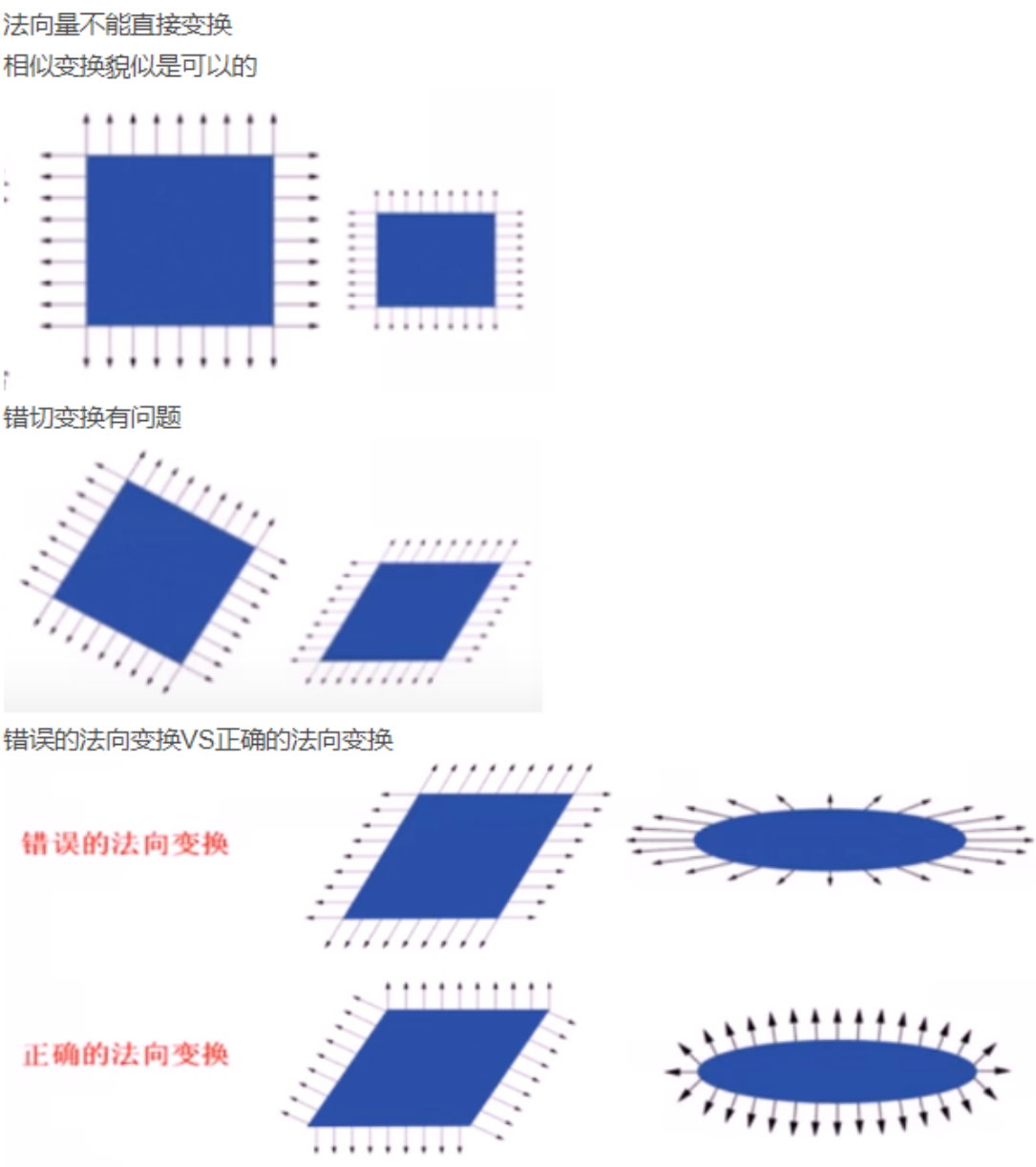
4. 片段着色器：需要一个`vec4`颜色输出变量，因为片段着色器需要生成一个最终输出的颜色。如果你在片段着色器没有定义输出颜色，OpenGL会把你的物体渲染为黑色（或白色）

```
#version 330 core
in vec4 vertexColor; // 从顶点着色器传来的输入变量（名称相同、类型相同）

out vec4 color; // 片段着色器输出的变量名可以任意命名，类型必须是vec4

void main()
{
    color = vertexColor;
}
```

5. 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。

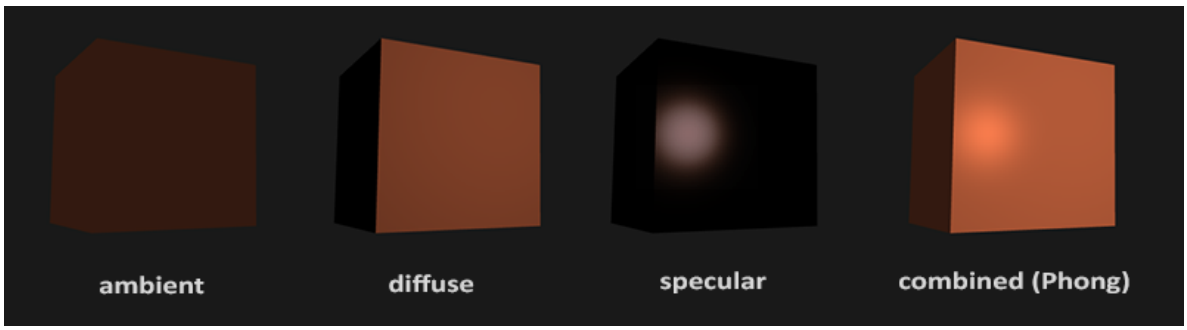
- uniform是全局的(Global)。全局意味着uniform变量必须在每个着色器程序对象中都是独一无二的，而且它可以被着色器程序的任意着色器在任意阶段访问。
- 无论你把uniform值设置成什么，uniform会一直保存它们的数据，直到它们被重置或更新。

9.透视投影 矩阵计算



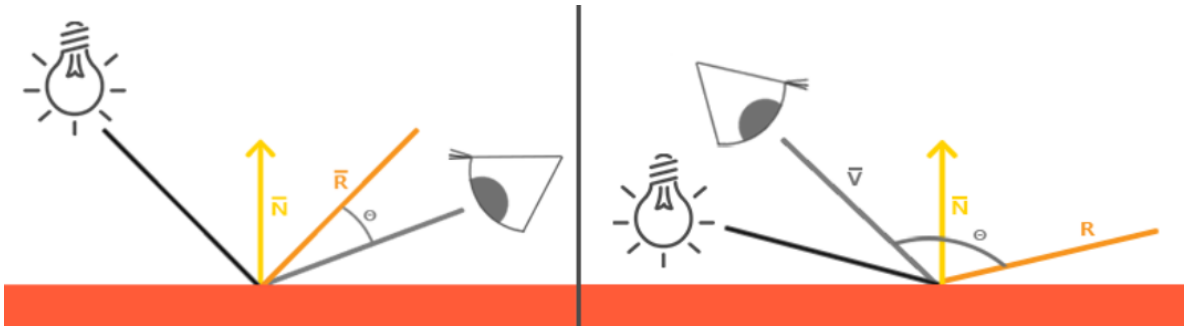
10.phong式光照模型

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。

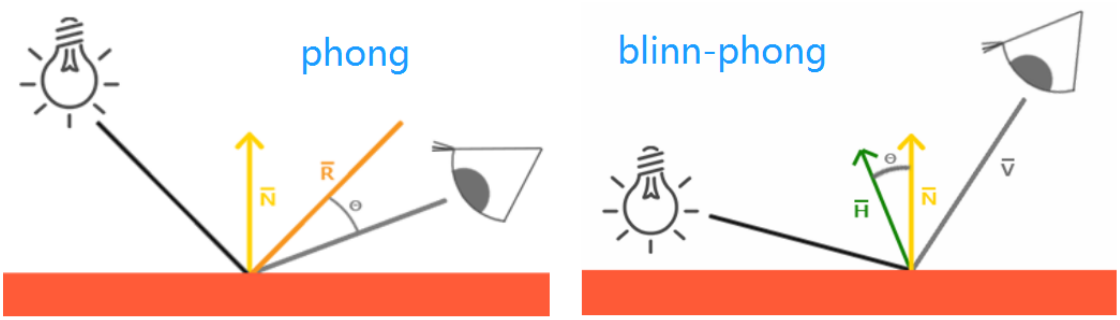


1. 环境光照：即使在黑暗的情况下，世界上也仍然有一些光亮(月亮、一个来自远处的光)，所以物体永远不会是完全黑暗的。我们使用环境光照来模拟这种情况，也就是无论如何永远都给物体一些颜色。
2. 漫反射光照：拟一个发光物对物体的方向性影响(Directional Impact)。它是冯氏光照模型最显著的组成部分。面向光源的一面比其他面会更亮。
3. 镜面光照：模拟有光泽物体上面出现的亮点。镜面光照的颜色，相比于物体的颜色更倾向于光的颜色。

为什么phong会有这种跳变的现象？算镜面光其实算的是光源关于法向量对称后的，所以即使差大于90度（右图），也不会暗，还能看到反射光。这就有问题



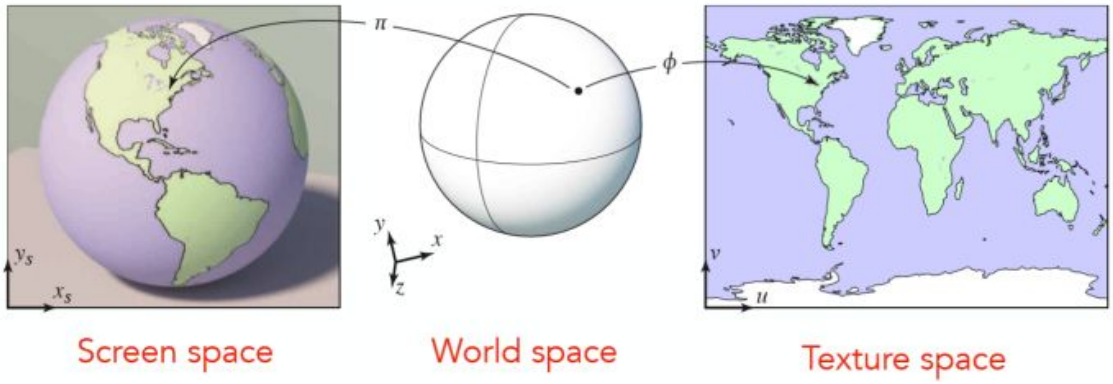
改成光源和视角夹角的半程向量。H，如果H和N的夹角越小，看到的反射光就越强。cosθ



11.纹理映射/纹理过滤/mipmap

纹理映射

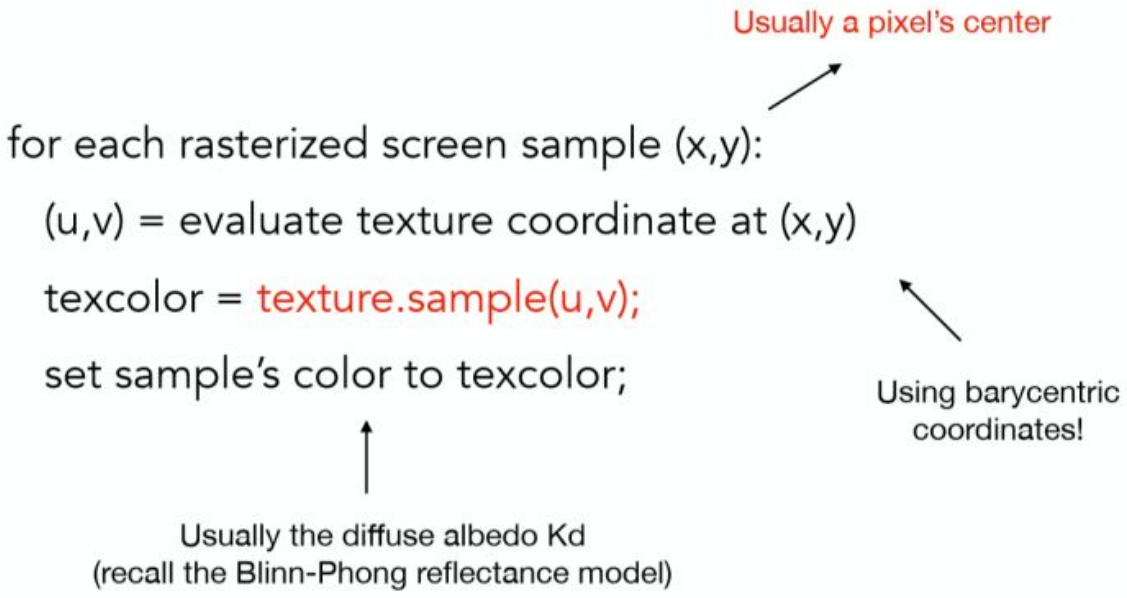
Phong反射模型的时候曾提到，一个点的颜色是由其漫反射系数决定的。我们可以将三维物体上的任意一个点都映射到一个2维平面之上，举一个简单例子，地球仪：



倘若拥有从3维World space到2维Texture space的一个映射关系，那么只需要将每个点的颜色信息即漫反射系数存储在2维的Texture之上，每次利用光照模型进行计算的时候根据映射关系就能查到这个点的漫反射系数是多少，所有点计算完之后，结果就像最左边的screen space之中，整个Texture被贴在了模型之上。

这种映射关系通过纹理坐标(UV)表示。在纹理空间之内任意一个二维坐标都在[0,1]之内，整张texture图可视化之后其实是红色和绿色的（把(u,v)坐标的两点想象成red和green就可以了）**查询一个顶点所对应在纹理空间的坐标是怎么得到的**

伪代码表示这个过程：

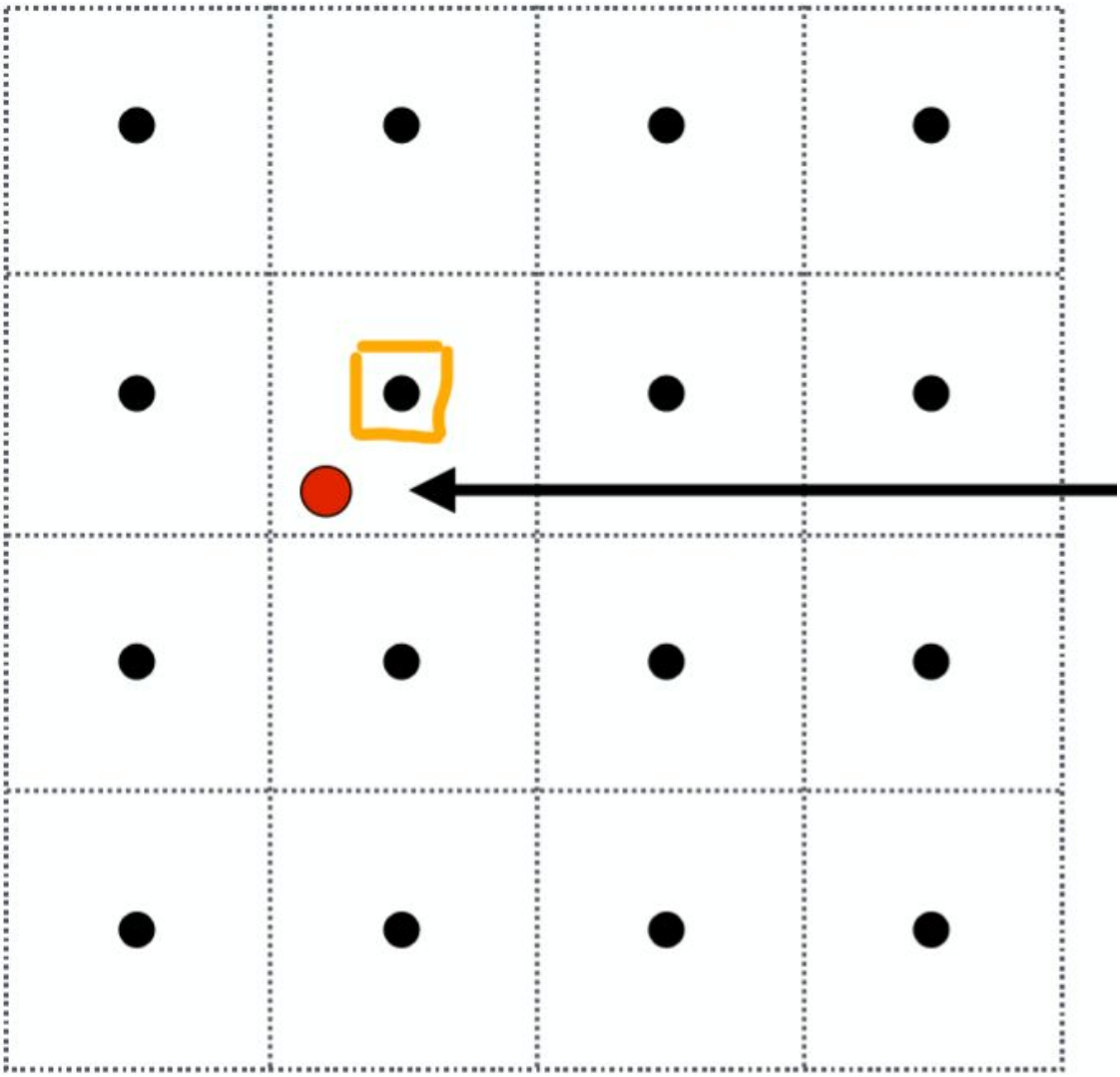


简而言之就是对每个光栅化的屏幕坐标算出它的uv坐标(利用三角形顶点重心坐标插值)，再利用这个uv坐标去查询texture上的颜色，把这个颜色信息当作漫反射系数Kd。

纹理过小和过大(相对于)引发的问题

纹理过小

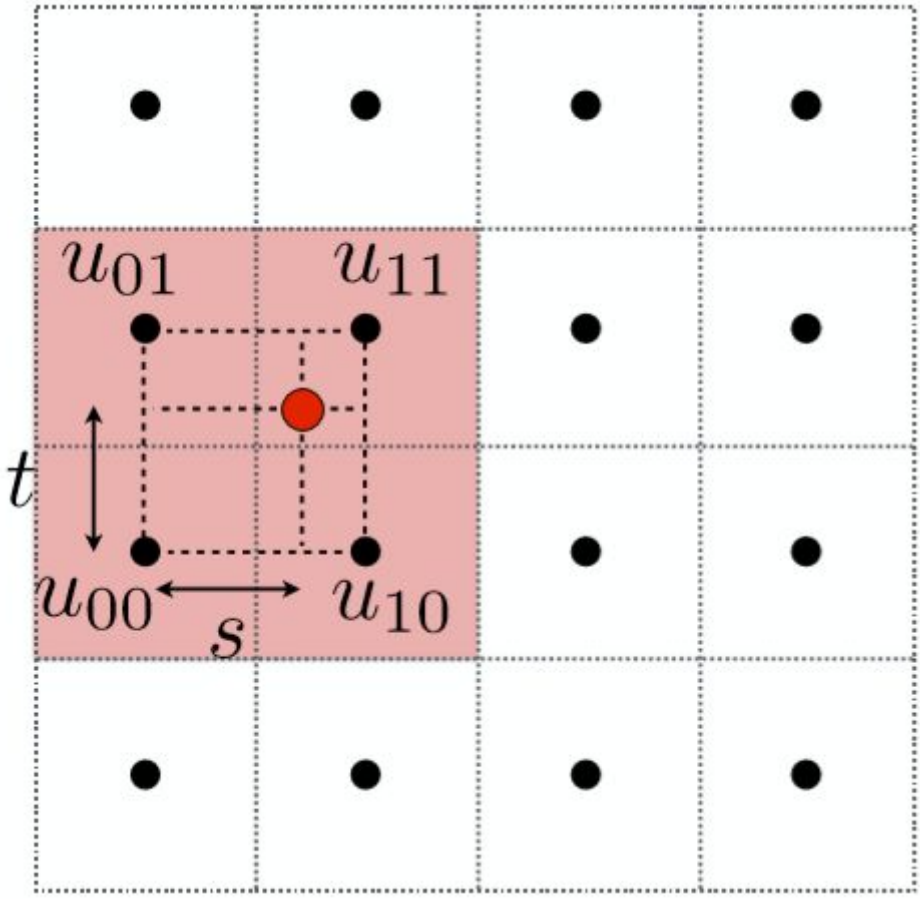
把一张100x100的纹理贴图应用在一张500x500的屏幕之上必然会导致走样失真，因为屏幕空间的几个像素点对应在纹理贴图的坐标上都是集中在一个像素大小之内。如果仅仅是引用对应(u,v)坐标在texture贴图下最近的那个像素点，往往会造成严重的走样。



这很容易理解，相当于好多个屏幕上的点都选取了同一个texel作为颜色值。

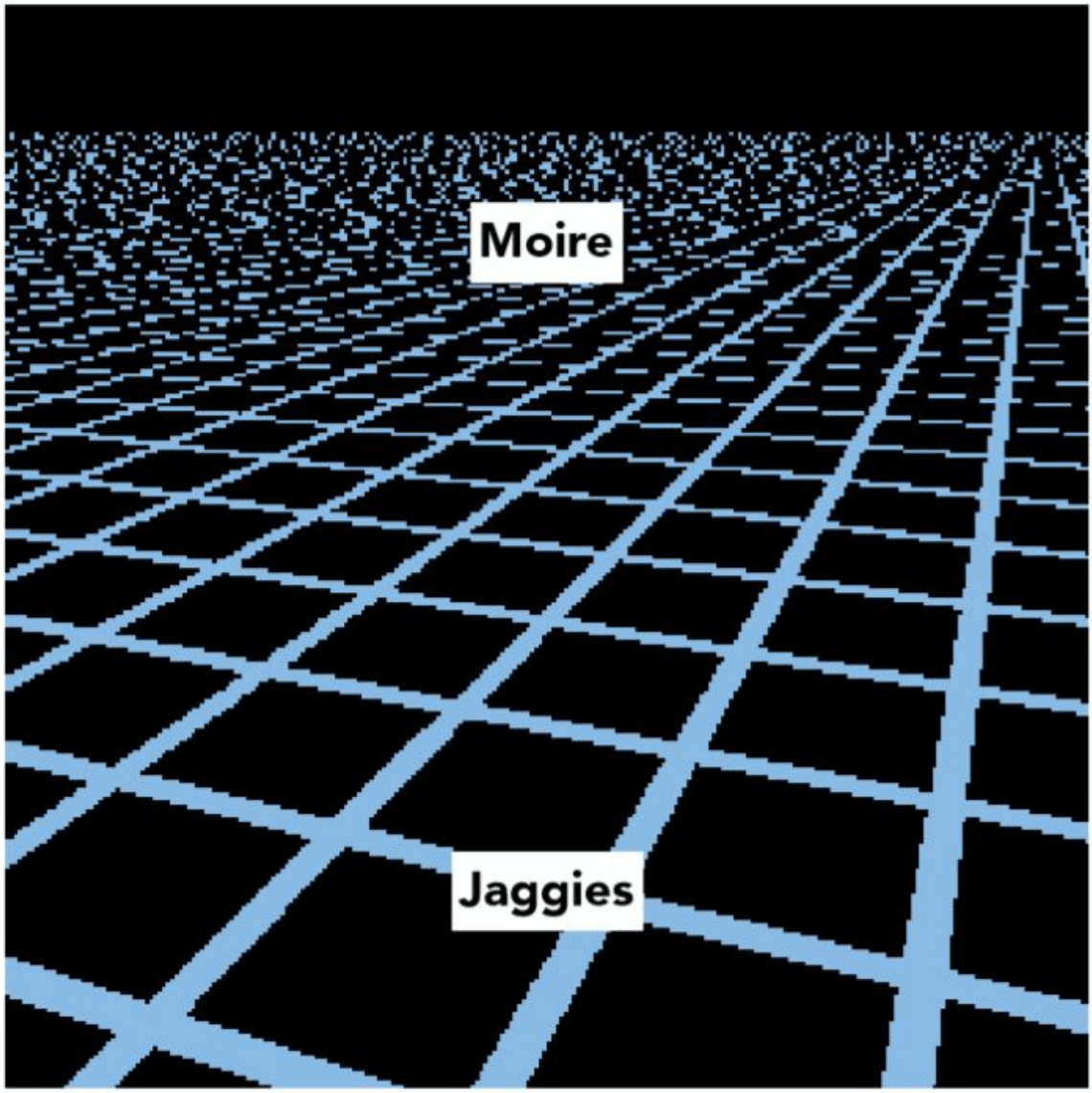
双线性插值(Bilinear Interpolation)

以上的例子可以通过双线性插值来缓解。原理如下，比较好理解。



纹理过大

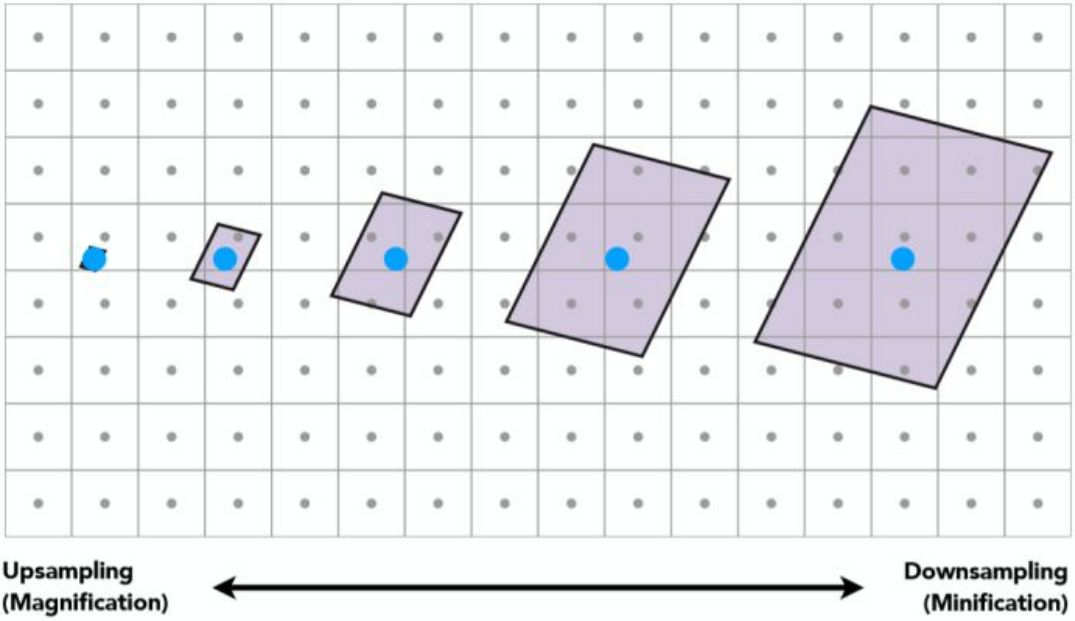
直接应用的话会出现摩尔纹以及锯齿，是非常严重的走样现象。



Point sampled

两种解释：

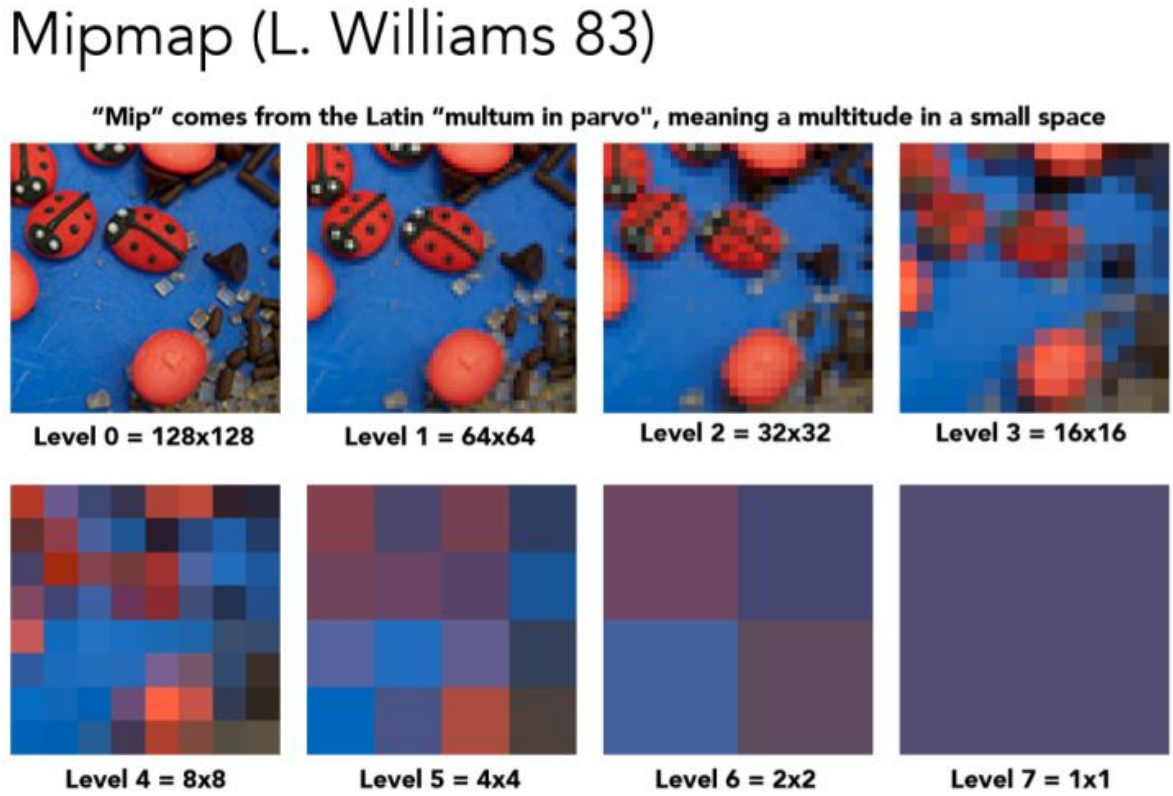
- **如开头所说，地板上铺满了重复的方格贴图，根据近大远小，远处的一张完整的贴图可能在屏幕空间中仅仅是几个像素的大小，那么必然屏幕空间的一个像素对应了纹理贴图上的一片范围的点，这其实就是纹理过大所导致的，直观来说想用一个点采样的结果代替纹理空间一片范围的颜色信息，必然会导致严重失真！(从信号的角度来说就是，采样频率过低无法还原信号原貌)**
- **换一种想法，考虑离相机很远的一个三角形面，假设该三角形面真正在纹理贴图上对应的一片区域有10个像素点。但是由于透视的关系，距离很远的三角形面投影到近平面时可能只有1个或2个像素点的大小(远远小于10个像素的原来大小)，那么这1个或2个像素采样texture的结果就要代表原来这个三角形面10个像素点的颜色信息，自然会导致失真！**



纹理贴图相对屏幕越大，一个屏幕空间像素点的所占的平均大小就越大。例如，纹理贴图大小500x500，屏幕空间100x100，将屏幕空间的像素点均匀分布在纹理空间之中，那么1个屏幕空间像素点所占的平均大小就是5x5=25个纹理空间像素。这种现象被称为屏幕像素在texture中的footprint，一个屏幕空间的蓝色像素距离相机越远，对应在texture空间的范围就越大，也就是越欠采样（采样频率达不到信号频率）。超采样可以缓解这个问题，但是计算量太大了，同时距离相机越远，更多的texel会出现在屏幕像素的一个footprint中，这样就会要求更高的超采样频率。

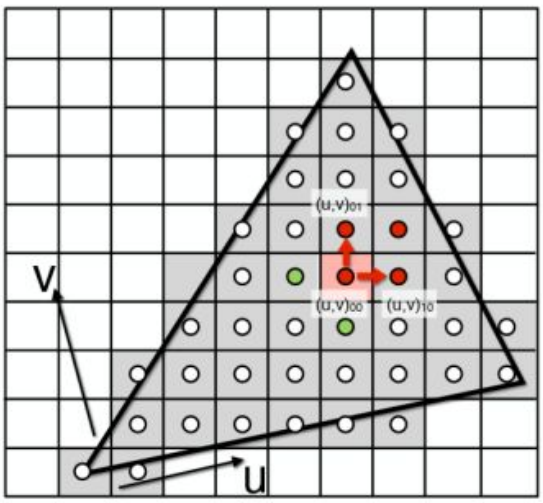
Mipmap（已经硬件支持）

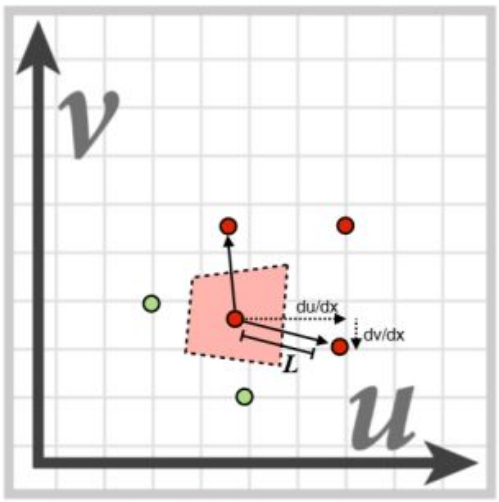
一个采样点的颜色信息不足以代表 “footprint”里一个区域的颜色信息，如果可以求出这样一个区域里面所有颜色的均值，是不是就是一种可行的方法呢？我们的目标就是从点查询Point Query迈向区域查询Range Query。



level 0代表的是原始texture，也是精度最高的纹理，随着level的提升，每提升一级将4个相邻像素点求均值合为一个像素点，因此越高的level也就代表了更大的footprint的区域查询。接下来要做的就是根据屏幕像素的footprint大小选定不同level的texture，再进行点查询即可，而这其实就相当于在原始texture上进行了区域查询！

为了确定使用哪个level的texture，利用屏幕像素的相邻像素点估算footprint大小再确定level D

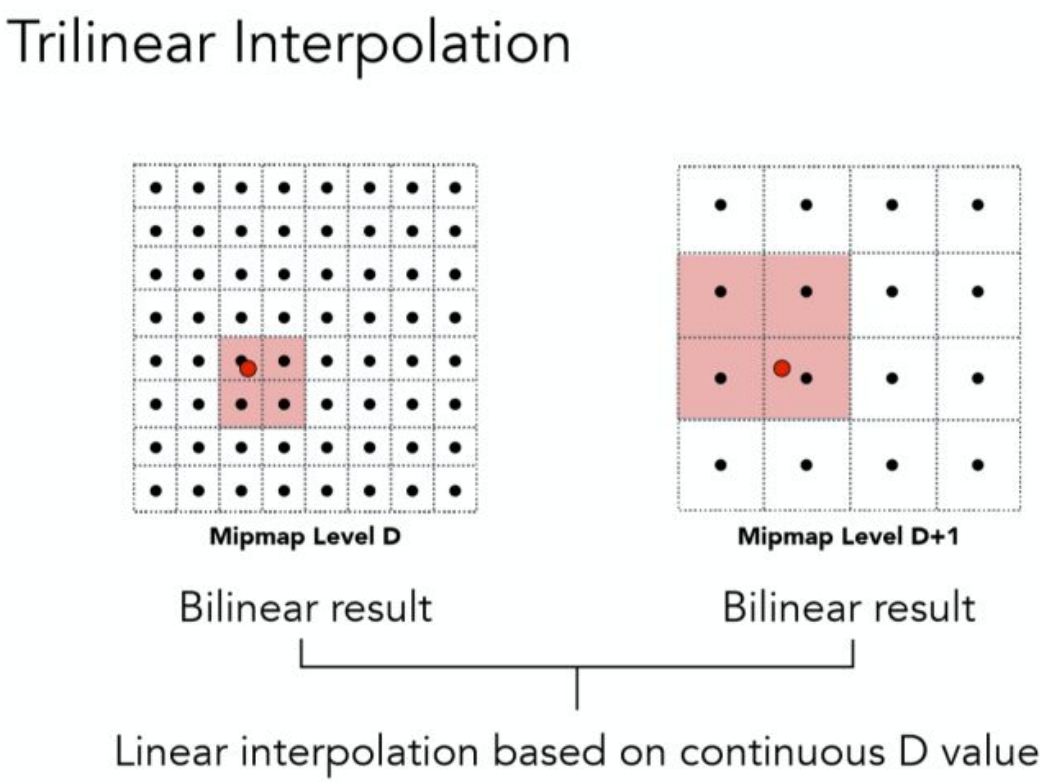


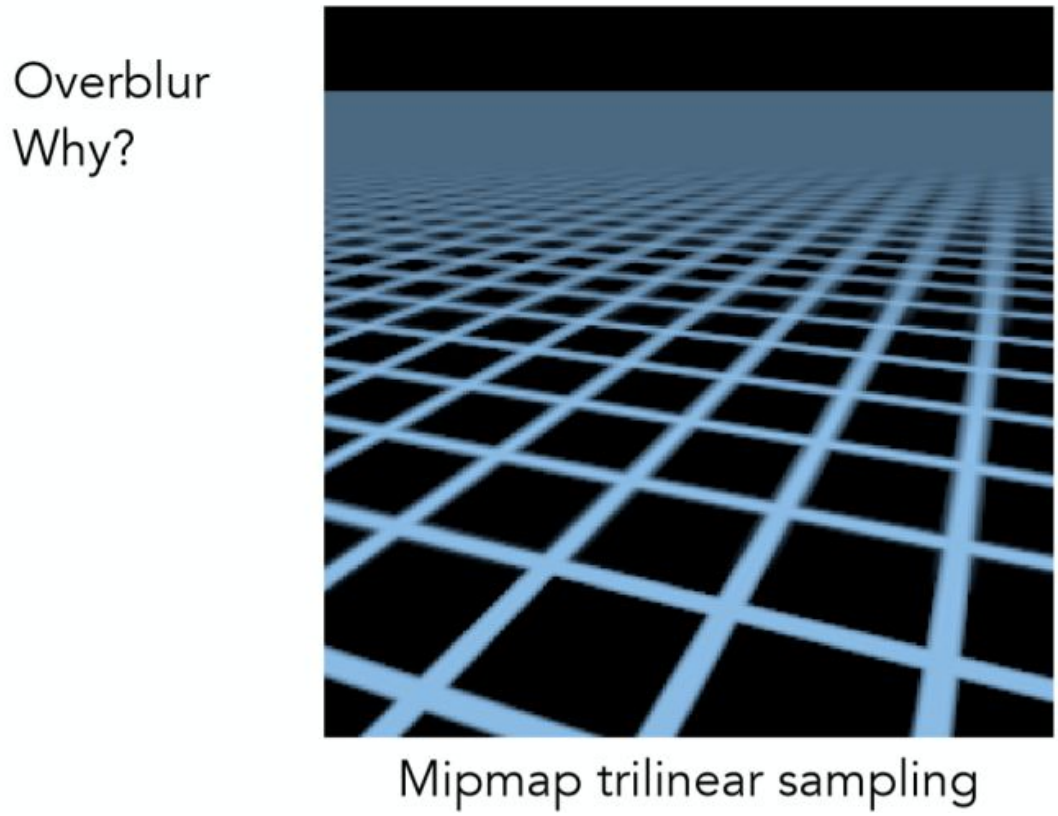


$$D = \log_2 L \qquad L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

但是这样计算出来的D值是一个连续值，并不是一个整数，有两种办法：

- 四舍五入取得最近的那个level D
- 利用D值再向下和向上取整的两个不同level进行三线性插值

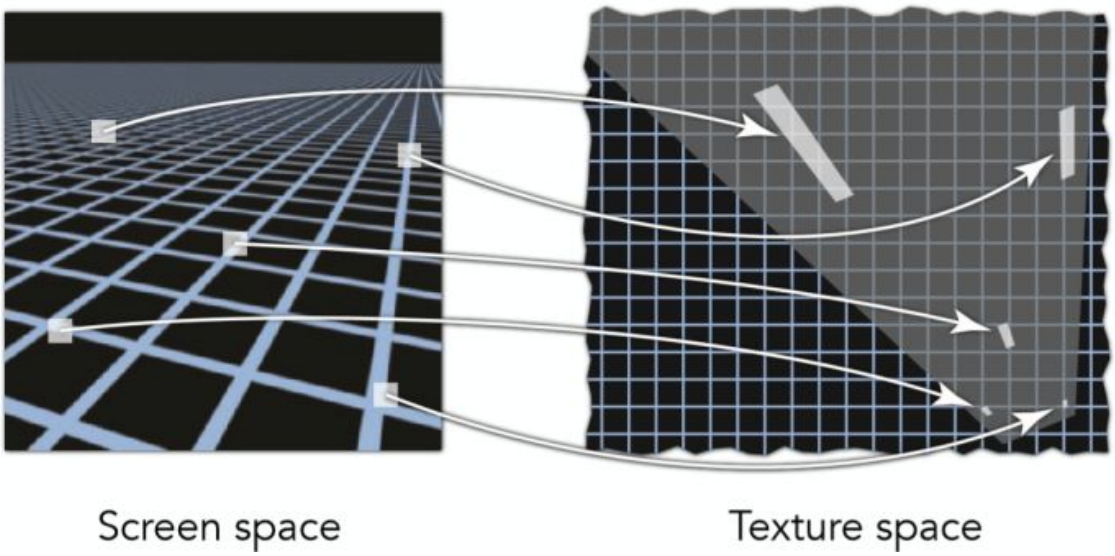




上述mipmap仍然不能解决问题，远处的地板糊在了一起

各项异性过滤Mipmap

上述现象的原因是，所采用的不同level的mipmap默认都是正方形区域的Range Query，但实际上并不是



可以看出不同screen space的像素点所对应的footprint是不同的，有长方形，甚至是不规则图形，那么针对这种情况，有的所需要的是仅仅是水平方向的高level，有的需要的仅仅是竖直方向上的高level，因此这也就启发了各向异性的过滤:

12.VAO VBO是什么

1. 顶点数组对象：Vertex Array Object，VAO

   一般提供一个vertexdata，都是很多个顶点坐标/法向量坐标/纹理坐标这样。需要一个VAO来解析他

   glattribpointer要配置顶点属性指针。

2. 顶点缓冲对象：Vertex Buffer Object，VBO

   因为顶点会需要放到GPU内存（显存）中，所以使用VBO的话，可以批量发送大量数据到显卡，如果是CPU一次发一个点到显卡那就很浪费时间。

我们通过顶点缓冲对象(Vertex Buffer Objects, VBO)管理这个内存，它会在GPU内存（通常被称为显存）中储存大量顶点。使用这些缓冲对象的好处是我们可以一次性的发送一大批数据到显卡上，而不是每个顶点发送一次。从CPU把数据发送到显卡相对较慢，所以只要可能我们都要尝试尽量一次性发送尽可能多的数据。当数据发送至显卡的内存中后，顶点着色器几乎能立即访问顶点，这是个非常快的过程。

顶点缓冲对象是我们在OpenGL教程中第一个出现的OpenGL对象。就像OpenGL中的其它对象一样，这个缓冲有一个独一无二的ID，所以我们可以使用glGenBuffers函数和一个缓冲ID生成一个VBO对象：

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

OpenGL有很多缓冲对象类型，顶点缓冲对象的缓冲类型是GL_ARRAY_BUFFER。OpenGL允许我们同时绑定多个缓冲，只要它们是不同的缓冲类型。我们可以使用glBindBuffer函数把新创建的缓冲绑定到GL_ARRAY_BUFFER目标上：

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

3. VAO VBO使用顺序

这么多了！前面做的一切都是等待这一刻，一个储存了我们顶点属性配置和应使用的VBO的顶点数组对象。一般当你**打算绘制多个物体**时，你首先要生成/配置所有的VAO（和必须的VBO及属性指针），然后储存它们供后面使用。当我们真要绘制物体的时候就**拿出相应的VAO，绑定它，绘制完物体后，再解绑VAO。**

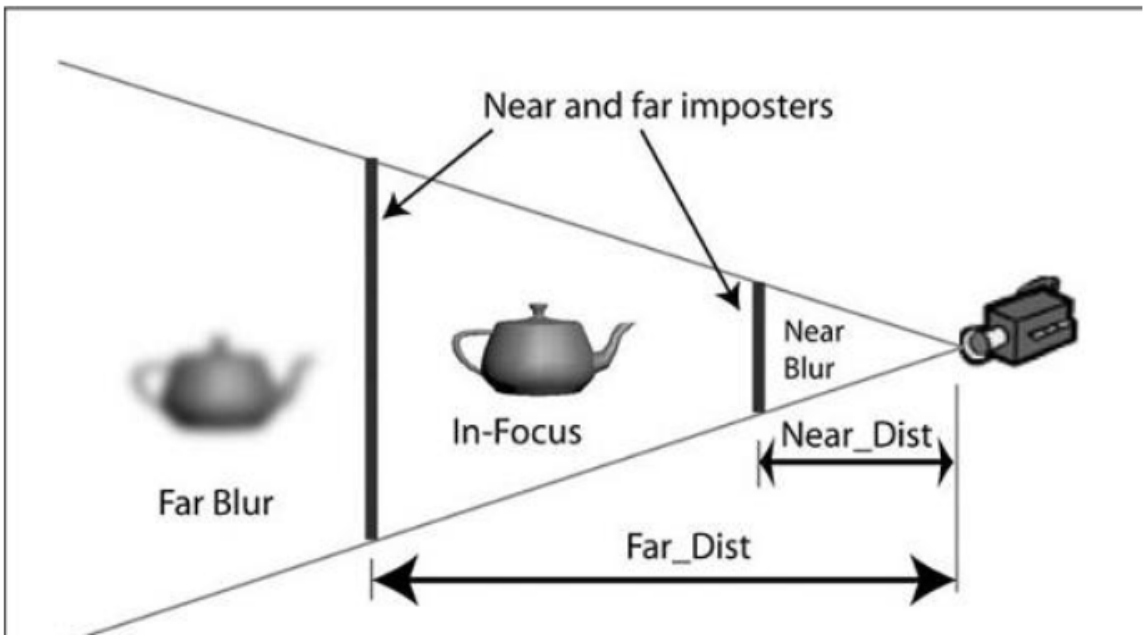
要想使用VAO，要做的只是使用glBindVertexArray绑定VAO。从绑定之后起，我们应该绑定和配置对应的VBO和属性指针，之后解绑VAO供之后使用。当我们打算绘制一个物体的时候，我们只要在绘制物体前简单地把VAO绑定到希望使用的设定上就行了。这段代码应该看起来像这样：

```
// ..:: 初始化代码（只运行一次（除非你的物体频繁改变）） :: ..
// 1. 绑定VAO
glBindVertexArray(VAO);
// 2. 把顶点数组复制到缓冲中供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

[...]

// ..:: 绘制代码（渲染循环中） :: ..
// 4. 绘制物体
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

13.z-buffer early z

现在大部分的GPU都提供一个叫做提前深度测试(Early Depth Testing)的硬件特性。提前深度测试允许深度测试在片段着色器之前运行。只要我们清楚一个片段永远不会是可见的（它在其他物体之后），我们就能提前丢弃这个片段。
片段着色器通常开销都是很大的，所以我们应该尽可能避免运行它们。当使用提前深度测试时，片段着色器的一个限制是你不能写入片段的深度值。如果一个片段着色器对它的深度值进行了写入，提前深度测试是不可能的。OpenGL不能提前知道深度值。

14.景深的原理



15.shader怎么实现bloom的效果

进行两遍处理：

1. 第一遍处理时，我们需要对原场景图进行筛选，所有小于这个阈值的像素都被筛掉，所有大于该值的像素留下来，这样，我们就得到了一张只包含需要泛光部分的贴图，其余部分是黑色的；
2. 第二遍处理时，对上面得到的图片进行模糊处理。泛光效果是由衍射效果产生的，我们现实世界中看到的泛光效果，最亮的地方实际上是会向暗的地方扩散的，也就是说在亮的地方，边界是不明显的，所以我们就需要对泛光是部分，也就是我们上一步操作的结果图片进行模糊操作，达到光溢出的效果。
3. 最后将原图像和处理过的图像进行叠加。

16.背面剔除和遮挡剔除分别在渲染管线的什么阶段

涉及到的剔除方法包括：

- **视椎体剔除** 应用程序阶段(Application Stage)

  发生在应用程序阶段(Application Stage)，一般由游戏引擎内部实现或者自己编写对应的算法来实现，运行在CPU上。裁剪的依据主要是根据摄像机的视野(field of view)以及近裁减面和远裁剪面的距离，将可视范围外的物体排除出渲染，被剔除的物体将不会进入渲染的几何阶段(Geometry Stage)。视椎体剔除是减少渲染消耗的最有效手段之一，可以在不影响渲染效果的情况下大幅减少渲染涉及到的顶点数和面数。

  - 计算包围要绘制物体的AABB盒（世界空间）
  - 获得视锥体六个面的平面方程（世界空间）
  - 判断AABB盒的最小点和最大点在六个面的内侧还是外侧
  - 剔除掉最小和最大点完全在某一面外侧的物体
- **遮挡剔除** 光栅化阶段

  感觉就是深度测试（z-buffer）

> 现代GPU中运用了Early-Z的技术，在Vertex阶段和Fragment阶段之间（光栅化之后，fragment之前）进行一次深度测试，如果深度测试失败，就不必进行fragment阶段的计算了，因此在性能上会有很大的提升。但是最终的ZTest仍然需要进行，以保证最终的遮挡关系结果正确。

- **视口剔除**

  发生在几何阶段(Geometry Stage)后期，投影变换之后屏幕映射之前，是渲染管线的必要一环。只有当图元完全或部分存在于规范立方体内部的时候，才将其返送到光栅化阶段。其中，对于完全位于规范立方体内部的图元，则直接进行下一阶段；完全处于规范立方外部的图元则完全被舍弃；部分处于规范立方体内部图元，则会根据视口进行对应的裁剪，在这一过程中可能会产生新的顶点。

  通过视口剔除可以将视口外的图元舍弃掉，减小光栅化阶段的消耗。

- **背面剔除**

  在光栅化阶段进行

- **深度剔除**

  在Fragment Shader之后，光栅化阶段末期的融合阶段执行，又叫深度检测(或Z缓存检测)。

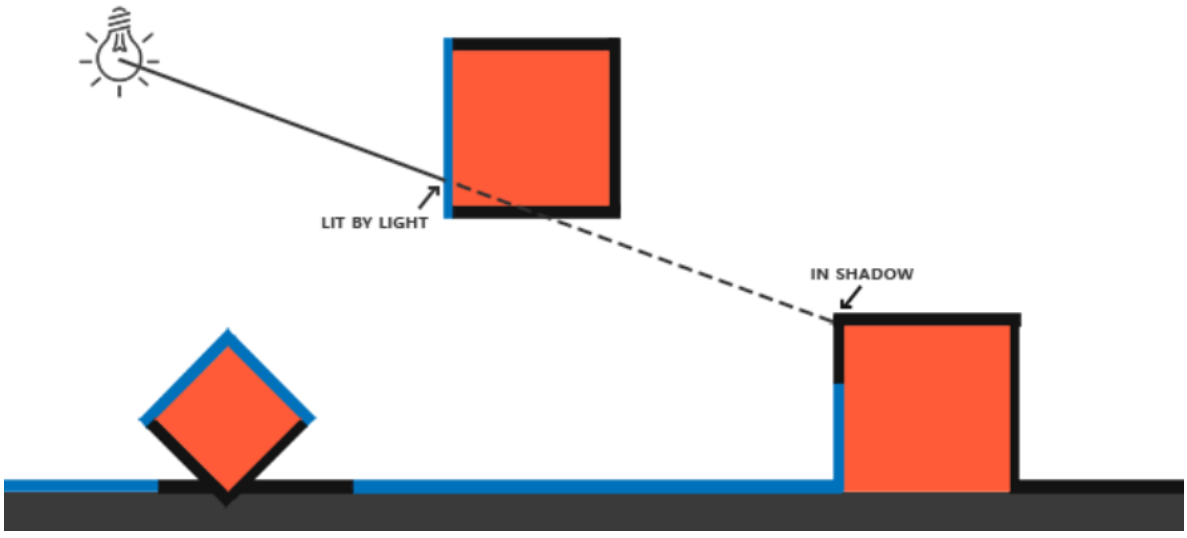
17.光照/高级光照/PBR

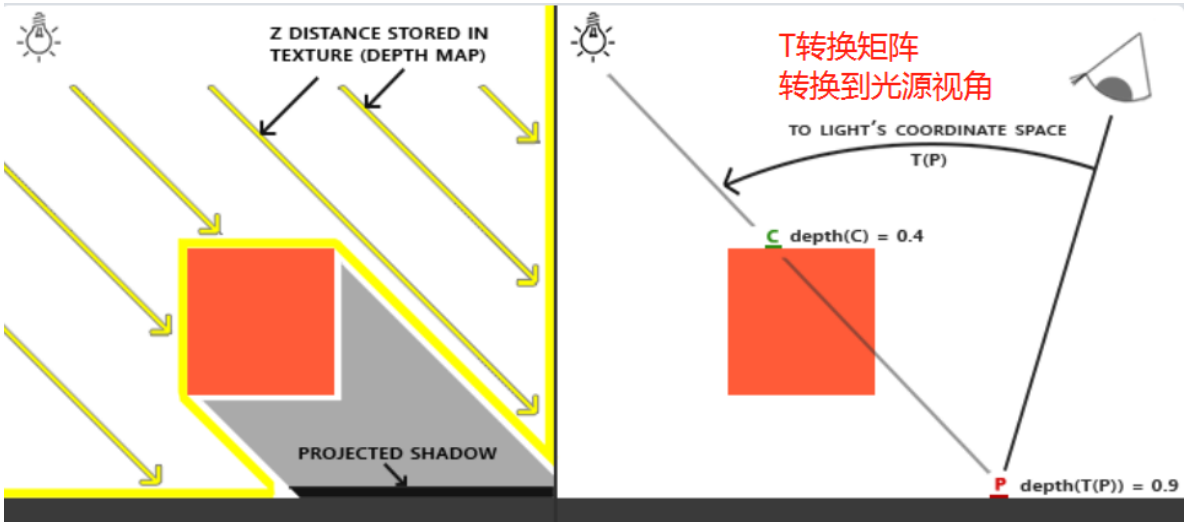
18.shadow map

shadow map是一种常用的实时阴影的生成方法。通常处理spot light，point light也可以。

2 pass的过程，第一趟从点光源出发，记录各个点的深度；第二趟从视点出发，将看到的点投影回光源的虚拟相机，看看是不是在阴影中。

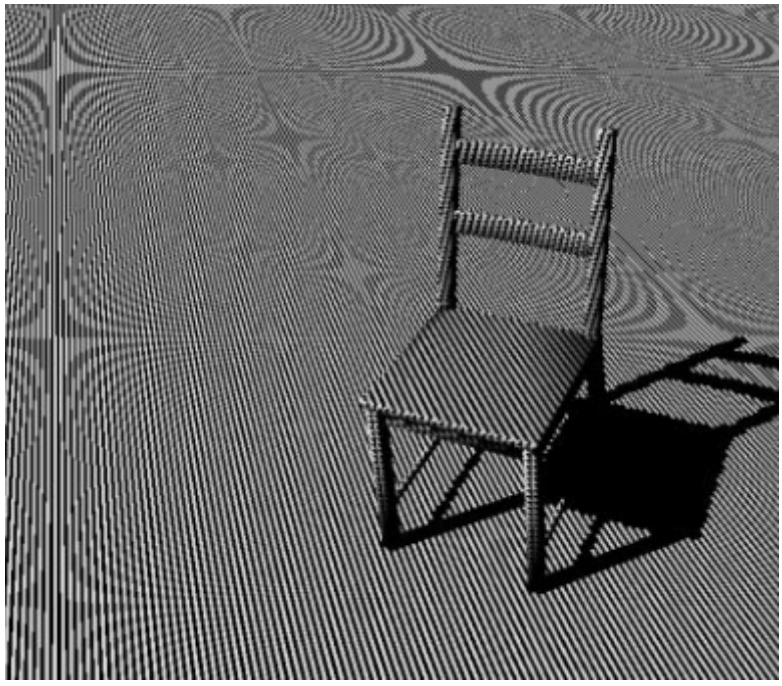
会出现问题：只能做出硬阴影，只能对点光源；质量取决于shadow map的分辨率；float比较会出现问题



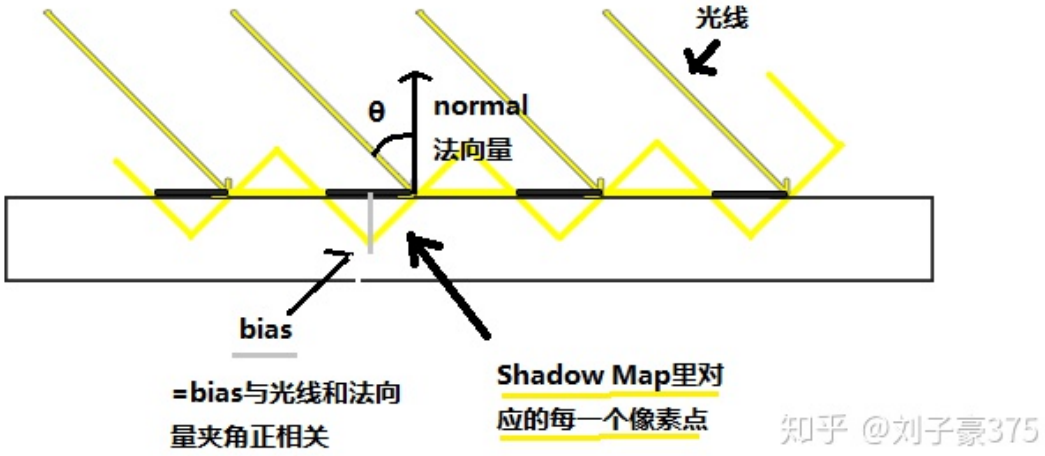


出现的问题：

- shadow acne 加bias解决



会出现以上的问题

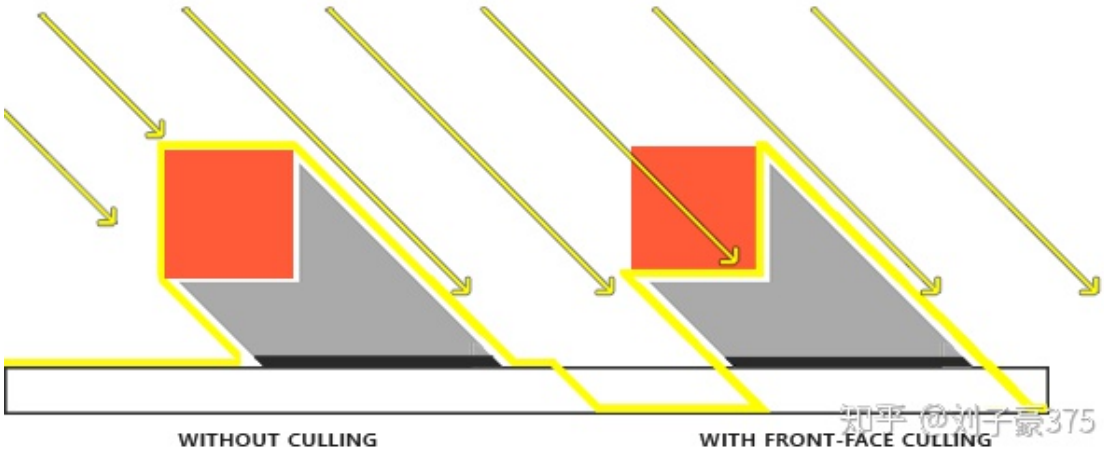


可以看见，阴影的产生和深度纹理的像素是相关的。深度纹理像素值越高，shadow acne的效果相对来说越小，然而shadow acne不可能通过提高像素来完全消除。因为只要光线和法向量的夹角不是90度，永远会有一部分的geometry会被判定处在阴影当中。这个问题可以通过给深度值加一个小小的bias来解决。通过法向量的和光线的夹角来判断bias要更为准确。

- peter panning

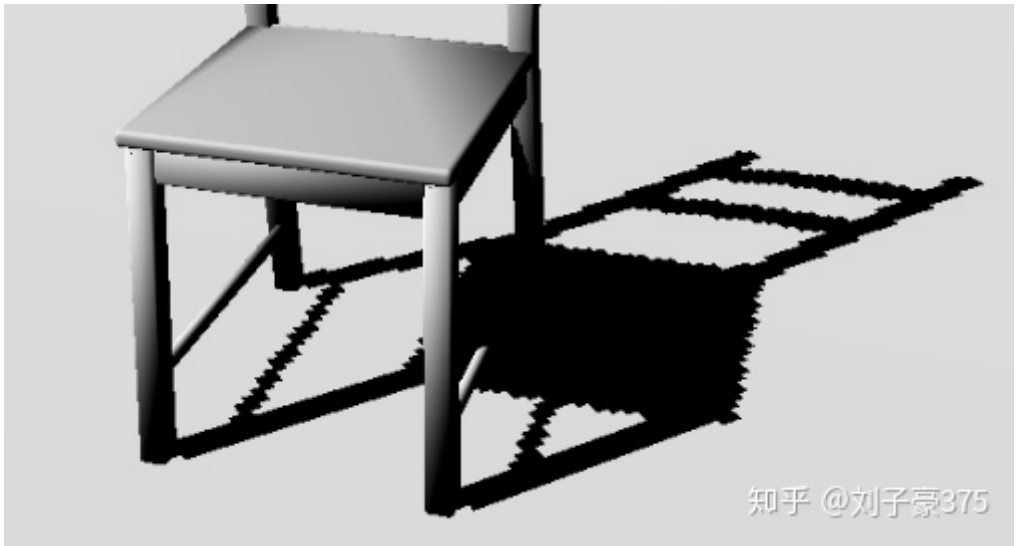
  加上bias 之后可能会出现阴影浮在空中的表现。

  也就是不计算物体正对着光的面所产生的阴影，只计算物体背面产生的阴影，他们产生的阴影相同，但是背面的bias值会小一些，所以可以有效解决peter panning的问题，原理如图：



  当然调整bias应该也是可以解决这个问题

- 阴影边缘会锯齿化



这是因为我们用的shadow map texture的分辨率的限制，会导致一个texel（texture里的像素点）会覆盖不止一个fragment，因此呢，会有很多fragments有同样的深度值采样。

解决这个最简单的方法就是提高depth texture的分辨率，这个很好理解。这里接受另外一种方法：PCF（percentage closer filtering）,其实也就是对深度纹理的texel采样取平均值。

19.延迟渲染

1. **正向渲染**

   正向渲染(Forward Rendering)，或称正向着色(Forward Shading)，是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。 传统的正向渲染思路是，先进行着色，再进行深度测试。其的主要缺点就是光照计算跟场景复杂度和光源个数有很大关系。假设有 n 个物体，m 个光源，且每个每个物体受所有光源的 影响，那么复杂度就是 O(m*n)。

   正向渲染简单直接，也很容易实现，但是同时它对程序性能的影响也很大，因为对每一个需 要渲染的物体，程序都要对每个光源下每一个需要渲染的片段进行迭代，如果旧的片段完全 被一些新的片段覆盖，最终无需显示出来，那么其着色计算花费的时间就完全浪费掉了。

2. **延迟渲染**

   可以将延迟渲染(Deferred Rendering)理解为先将所有物体都先绘制到屏幕空间的缓冲（即Gbuffer，Geometric Buffer，几何缓冲区）中，再逐光源对该缓冲进行着色的过程，从而避免了因计算被深度测试丢弃的片元的着色而产生的不必要的开销。也就是说延迟渲染基本思想是，先执行深度测试，再进行着色计算，将本来在物空间（三维空间）进行光照计算放到了 像空间（二维空间）进行处理。经典的延迟渲染复杂度为 O(n+m)。

3. **延迟渲染的过程分析**

   1、几何处理阶段(Geometry Pass)。这个阶段中，我们获取对象的各种几何信息，并将第二步 所需的各种数据储存（也就是渲染）到多个 G-buffer 中；

   2、光照处理阶段(Lighting Pass)。在这个 pass 中，我们只需渲染出一个屏幕大小的二维矩形，使用第一步在 G-buffer 中存储的数据对此矩阵的每一个片段计算场景的光照；光照计算的过 程还是和正向渲染以前一样，只是现在我们需要从对应的 G-buffer 而不是顶点着色器(和一些 uniform 变量)那里获取输入变量了。

   延迟渲染方法一个很大的好处就是能保证在 G-buffer 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。 在几何处理阶段中填充 G-buffer 非常高效，因为我们直接储存位置，颜色，法线等对象信息 到帧缓冲中，这个过程几乎不消耗处理时间。 而在此基础上使用多渲染目标(Multiple Render Targets, MRT)技术，我们可以在一个 Pass 之内 完成所有渲染工作

4.

20.PBR/渲染方程/BRDF

PBR：基于物理的渲染

基本的物理原理

1. **次表面散射**

次表面散射，Subsurface Scattering，简称 SSS(又简称 3S)，就是光射入半透明材质后在内部发生散射，最后射出物体并进入视野中产生的现象，是指光从表面进入物体经过内部散射，然后又通过物体表面的其他顶点出射的光线传递过程。

图 10 次表面散射示例 2

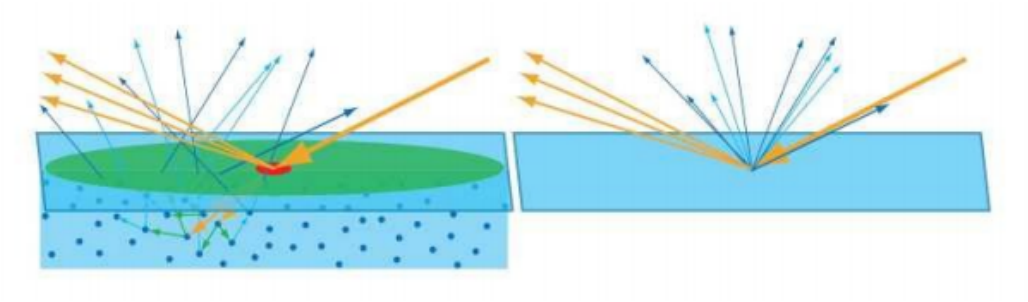


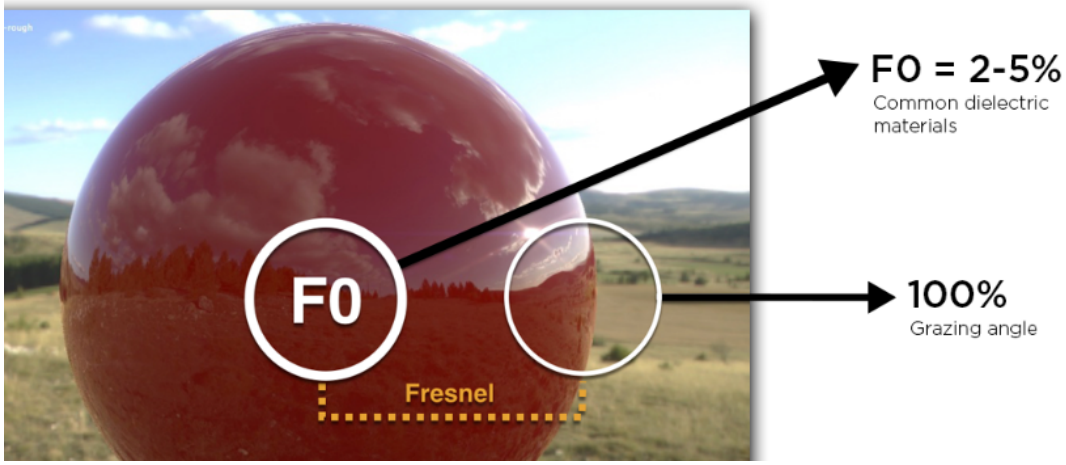
图 11 光与表面的相互作用。左图，可以看到次表面相互作用导致光从入口点重新射出。红色和绿色圆圈代表两个不同尺度下的像素所覆盖的区域。在右边，所有的次表面散射光都是从入口点发出的，忽略了表面之下的细节。

2. **菲涅尔反射**

菲涅尔反射（Fresnel Reflectance）或者菲涅尔效果（Fresnel Effect），即当光入射到折射率不同的两个材质的分界面时，一部分光会被反射，而我们所看到的光线会根据我们的观察角度以不同强度反射的现象。

相当于全反射。

菲涅尔反射率F0是什么意思？光线垂直撞表面时，镜面反射的比例F0，折射(1-F0)



3. **微平面理论**

微表面理论假设表面是由不同方向的微小细节表面，也就是微平面（microfacets）组成。每一个微小的平面都会根据它的法线方向在一个方向上反射光线。

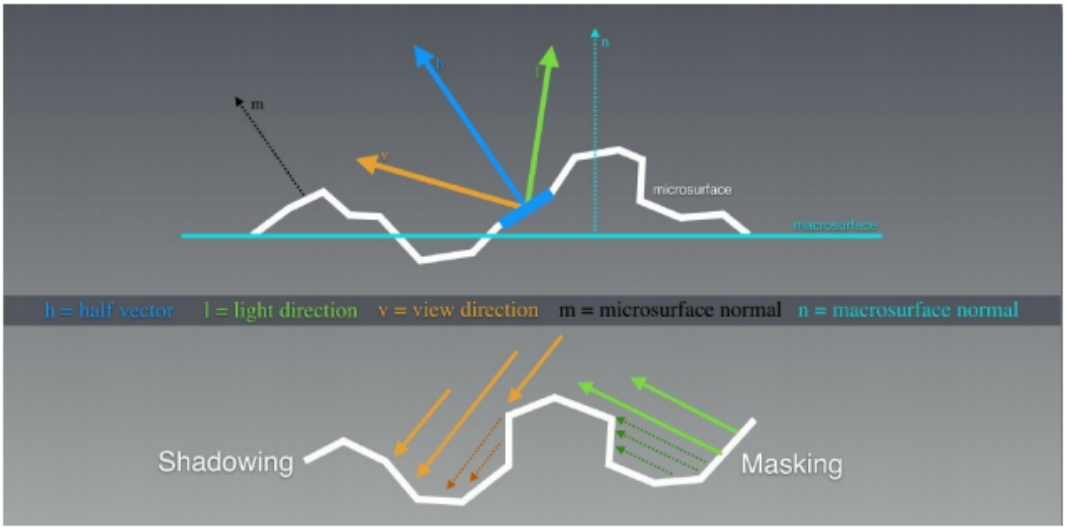


图 14 微平面理论图示

我们用法线分布函数（Normal Distribution Function，简写为 NDF），D(h)来描述组成表面一点的所有微表面的法线分布概率。则可以这样理解：向 NDF 输入一个朝向 h，NDF 会返回朝向是 h 的微表面数占微表面总数的比例，比如有 8%的微表面朝向是 h，那么就有 8%的微表面可能将光线反射到 v 方向。

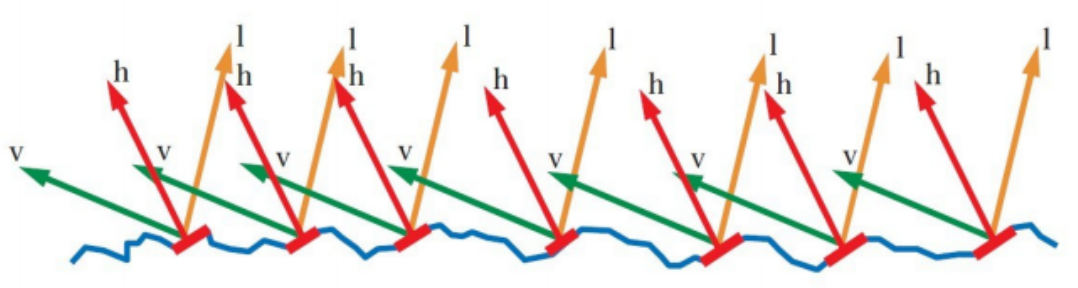


图 15 由微平面组成的表面。仅红色微平面的表面法线和半矢量 h 对齐，能参与从入射光线向量 l 到视线向量 v 的光线反射

实际工作流中，这种不平坦的表面是用粗糙度贴图或者高光度贴图来表示的.

Cook-Torrance BRDF 模型

Cook-Torrance 模型将物理学中的菲涅尔反射引入了图形学，实现了比较逼 真的效果。 Cook-Torrance 微平面着色模型（Cook-Torrance microfacet specular shading model），即 Microfacet Specular BRDF，定义为：

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n\cdot h)(n\cdot v)}$$

- F 为菲涅尔反射函数(Fresnel 函数)
- G 为阴影遮罩函数（Geometry Factor，几何因子），即未被 shadow 或 mask 的比例
- D 为法线分布函数(NDF)

基于真实世界的材质参数

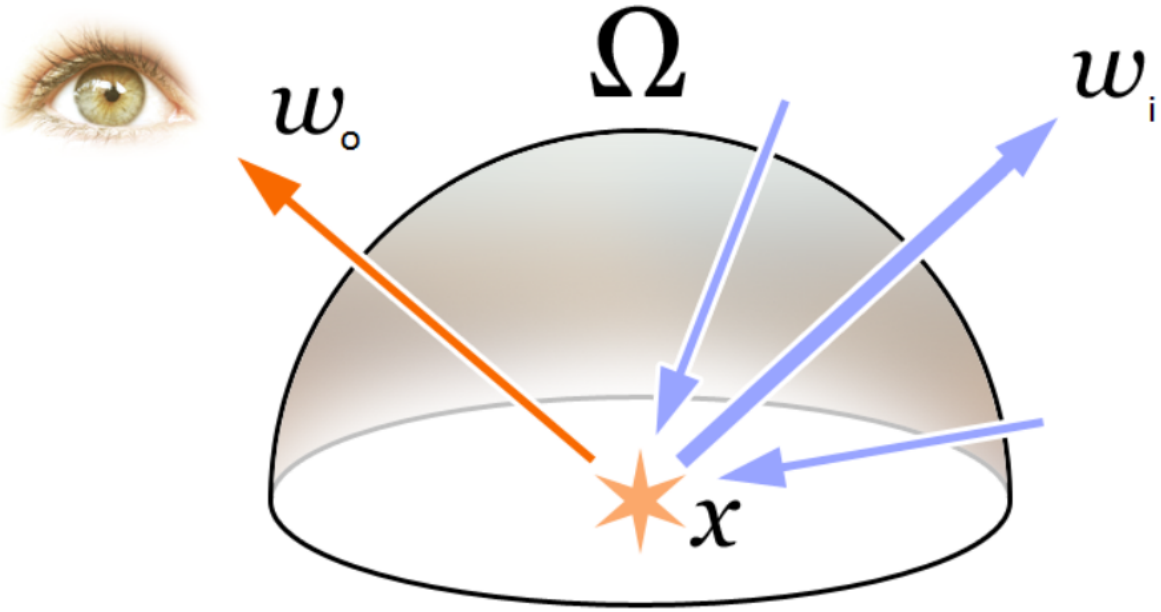
导体（金属）F0是一个float3

绝缘体（电解质，非金属）F0是一个float

	金属	非金属
镜面反射颜色	(R,G,B)三通道彩色	单通道
折射	吸收	吸收+散射
F0菲涅尔反射	0.5以上	0.05以下（很小

渲染方程

$$L_o = L_e + \int_\Omega f_r \cdot L_i \cdot (w_i \cdot n) \cdot dw_i$$



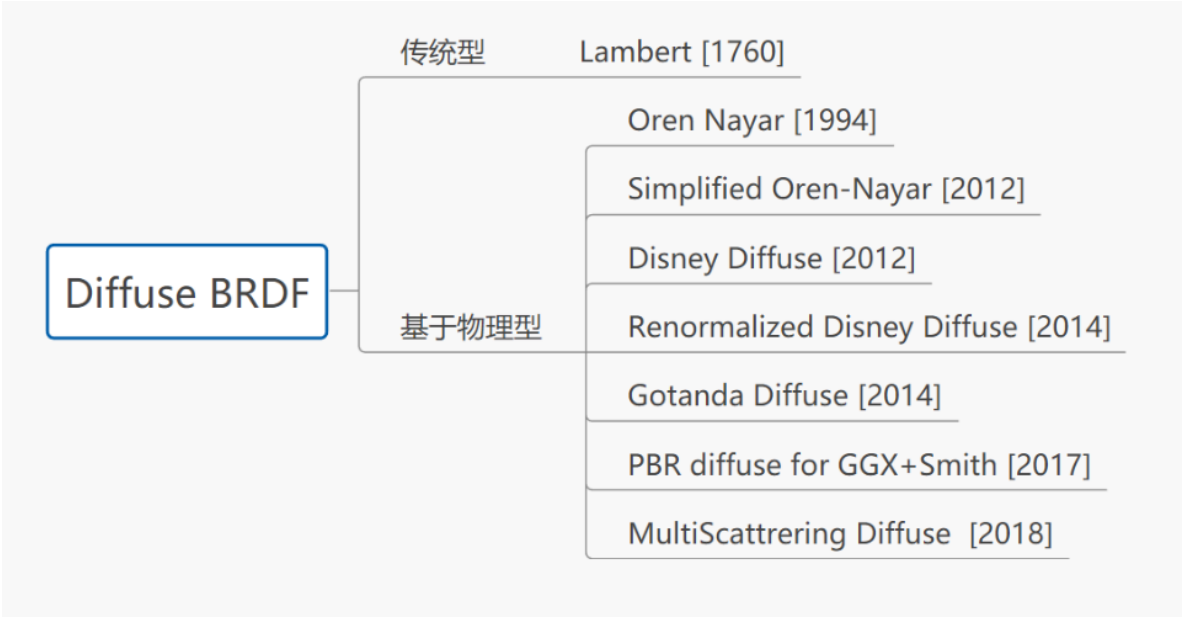
Lo是出射光亮度

Le是自发光亮度

fr是BRDF（反射方程），入射方向到出射方向光的反射比例

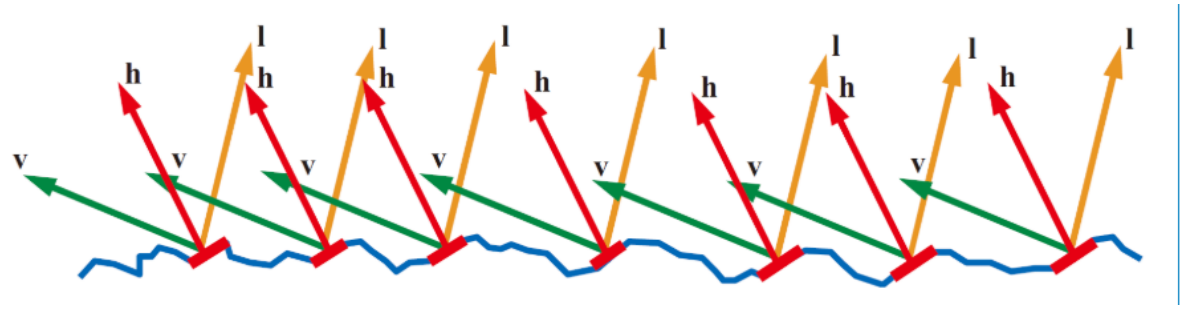
Li是入射光亮度，wi*n是入射光和法线的夹角cosθ，对每一个入射光的小方向求积分。

要求解渲染方程，需要分别求解Diffuse BRDF和Specular BRDF。所以PBR核心知识体系的第四部分是Diffuse BRDF。



游戏界目前最主流的基于物理的【**镜面反射**BRDF模型】是基于微平面理论的Microfacet Cook-Torrance BRDF

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n\cdot h)(n\cdot v)}$$



光方向l 视图方向v

所有表面点，只有那些恰好正确朝向可以将l反射到v的那些小平面(m=h)可能有助于BRDF值,才会将光线l反射到视线v的方向，其他表面点对BRDF没有贡献。h是半角矢量。

D（h）：法线分布函数 （Normal Distribution Function），**描述微面元法线分布的概率**，即**正确朝向的法线的浓度**。即具有正确朝向，能够将来自l的光反射到v的表面点的相对于表面面积的浓度。

F（l，h）：菲涅尔方程（Fresnel Equation），描述不同的表面角下表面所反射的光线所占的比率。

G(l，v，h): 几何函数（Geometry Function）：描述微平面自成阴影的属性，即m = h的未被遮蔽的表面点的百分比。

分母 4（n·l）（n·v）：校正因子（correctionfactor），作为微观几何的局部空间和整个宏观表面的局部空间之间变换的微平面量的校正。

Microfacet Cook-Torrance BRDF是实践中使用最广泛的模型，实际上也是人们可以想到的最简单的微平面模型。它仅对几何光学系统中的单层微表面上的单个散射进行建模，没有考虑多次散射，分层材质，以及衍射。Microfacet模型，实际上还有很长的路要走。

BRDF

在PBR那里已经讲过了

21.着色方式

即在给定基本光照模型使用的数学公式的基础上，对光照模型进行计算

Phong着色

在Fragment Shader中进行计算，是逐像素光照。以每个像素为基础，得到它的法线（可以是对顶点法线插值得到，也可以是从法线纹理中采样得到）

高洛德着色

在Vertex Shader 中进行计算，是逐顶点光照，在每个顶点上计算光照，然后会在渲染图元内部进行线性插值，最后输出成像素颜色。但是当光照模型中有非线性的计算的时候（比如计算高光反射的时候），逐顶点光照就会出现问题(待添加)。同时由于逐顶点光照会在渲染图元内部对顶点颜色进行线性插值，这样就会导致渲染图元内部的颜色总是低于顶点处的最高颜色值，在某些情况下会产生比较明显的棱角现象。

Flat着色

Flat shading 是最简单的着色模型，每个多边形只会呈现一个颜色，这个颜色由面法向量和光照计算得来。在该模型中，每个多边形中只有多边形的面存在法向量，而其各个顶点没有。