# （一）、Interpolation Across Triangles: Barycentric Coordinates

## 1、Interpolation Across Triangles(在三角形内部进行插值)

Why do we want to interpolate?(为什么需要插值)

Specify values at vertices(需要顶点的值)

Obtain smoothly varying values across triangles(需要三角形内部的值平滑过渡)

What do we want to interpolate?(需要插哪些值)

Texture coordinates, colors, normal vectors,(顶点坐标，颜色，法向量)

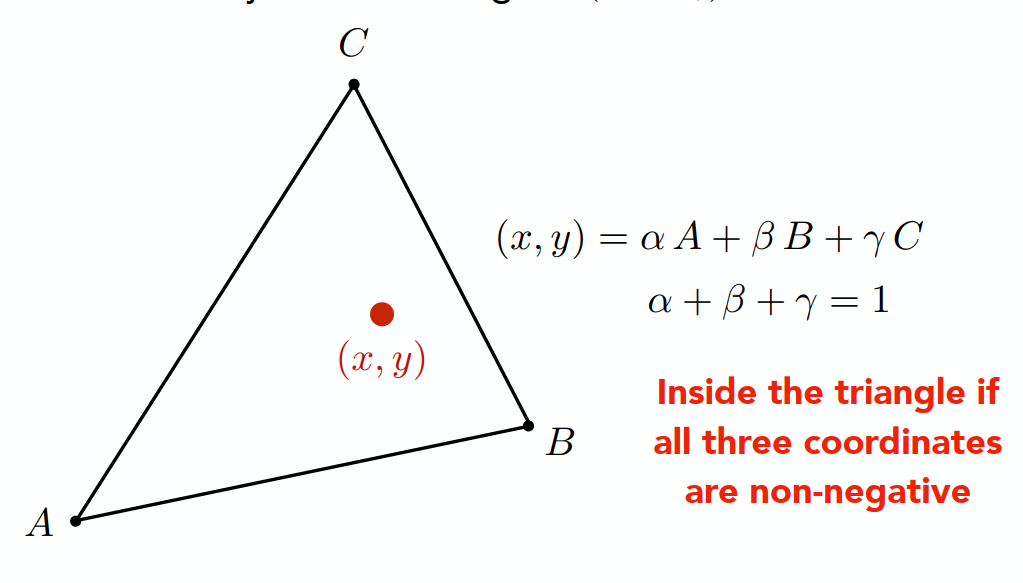
How do we interpolate?(怎么插值)

Barycentric coordinates(重心坐标)

### 2、Barycentric Coordinates

#### Barycentric Coordinates(重心坐标 已知αβγ求xy)

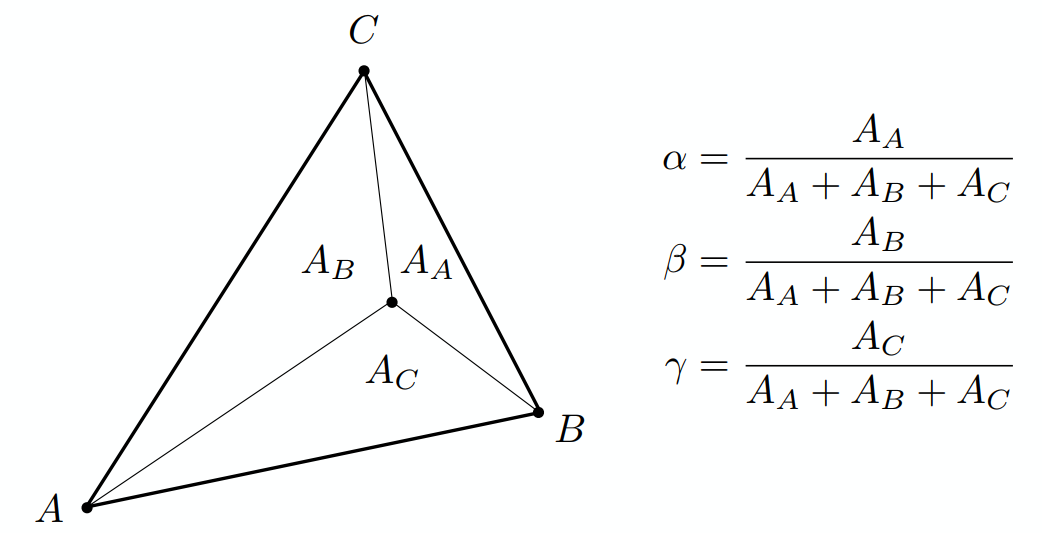
A coordinate system for triangles（α，β，γ）



Inside the triangle if all three coordinates are non-negative(非负)

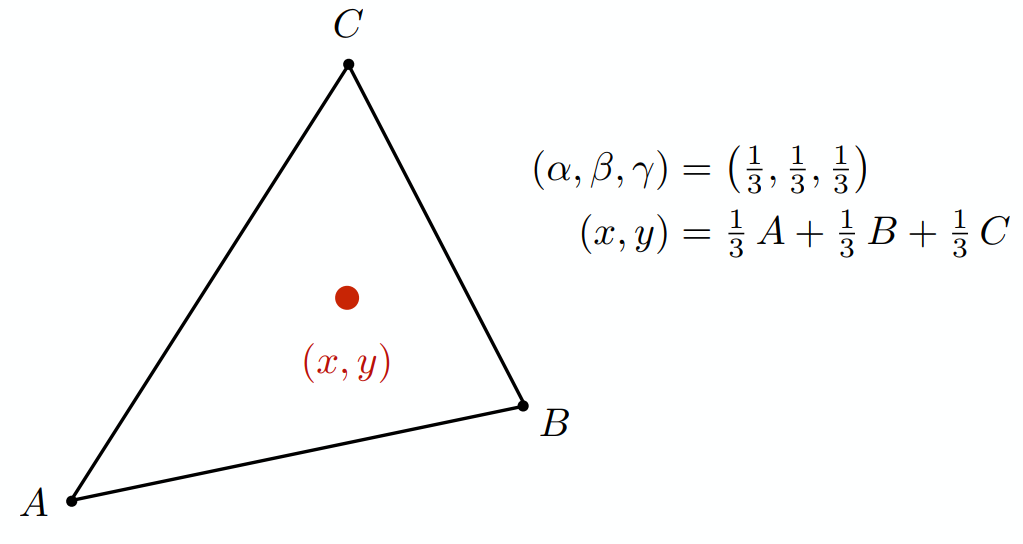
三角形所在平面上任意一个点（x，y）都可以用三个顶点A、B、C坐标的线性组合来表示，线性组合的系数α、β、γ之和等于1。如果系数都为非负数，则表示这个点在三角形内部。

#### Geometric viewpoint — proportional areas(三角形所在平面的任一点的表示)



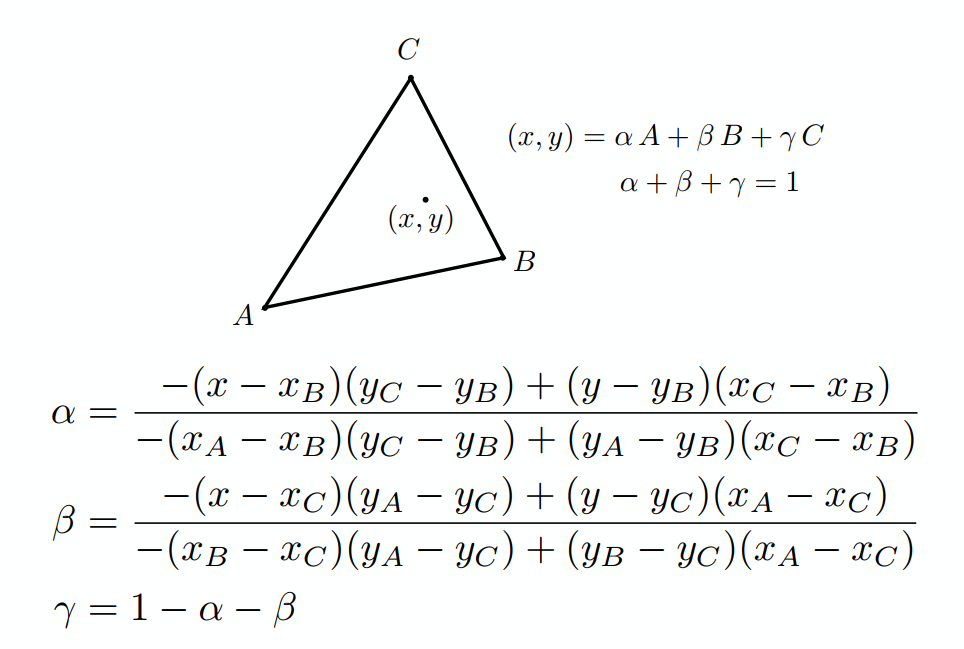
如果知道被某点（x，y）与三个顶点连接将这个三角形划分成三个小三角形各自的面积()，那么也可以求出系数α、β、γ(利用上图所示的面积比)。

#### What’s the barycentric coordinate of the centroid?(三角形质心的重心坐标表示)



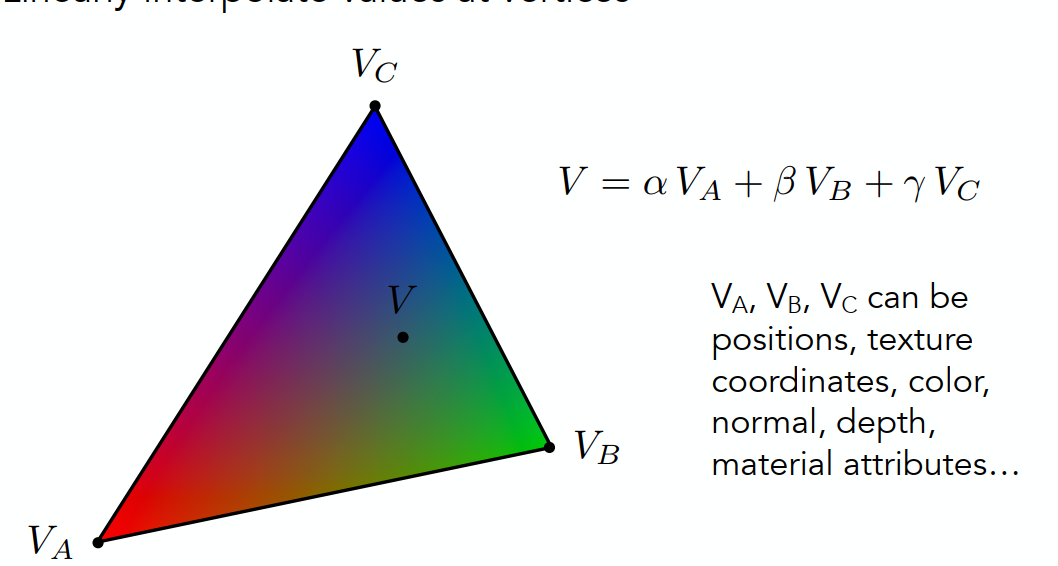
这里找一个特殊点：三角形的重心（质心）。三角形重心的重心坐标是（1/3,1/3,1/3）。

#### Barycentric Coordinates: Formulas(已知xy求αβγ)



### 3、Using Barycentric Coordinates

Linearly interpolate values at vertices



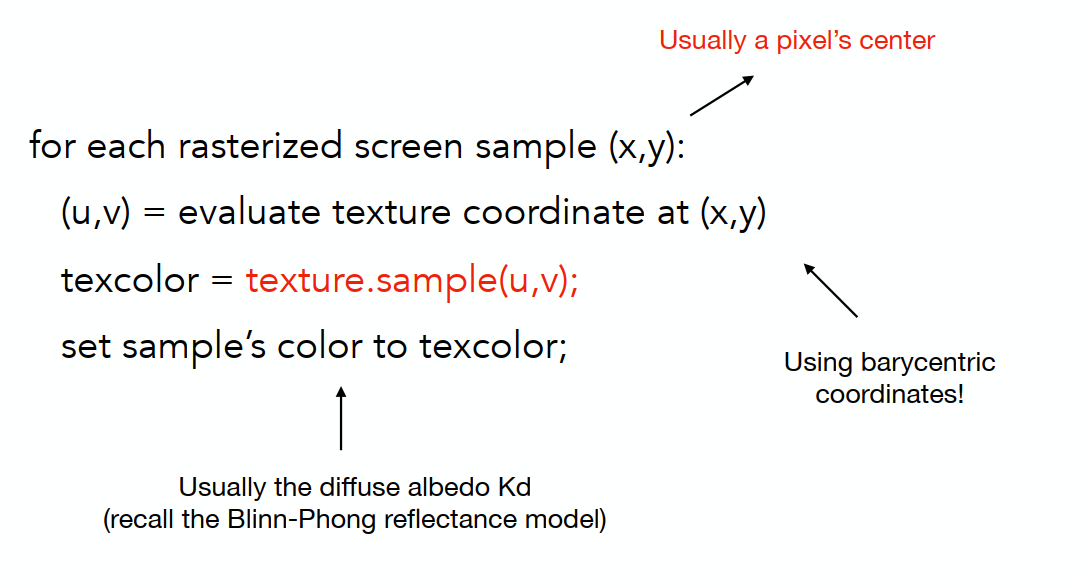
当一个点的重心坐标（α，β，γ）求出来以后，就可以将任意属性V在此点的插值求出来，此时只需要将三个顶点的该属性和系数做乘法再相加，即可求出在该点插值出来的属性。(属性包括位置，纹理坐标，颜色，法线，深度，)

但是要注意，这时候三角形已经投影到屏幕上了，尤其对于在三维空间中的属性（比如深度信息），应该找到像素中心点对应三角形的位置的三维空间坐标，然后在三维空间中将A、B、C的深度（属性）插值好，再放回来。这个过程需要做一次逆变换就可以了。

### 4、Applying Textures

屏幕上任何一个采样点（像素中心）的位置(x,y)，我们可以求出在这个位置上插值出来的(u,v)（纹理坐标）的位置。

原来纹理坐标定义在三角形的顶点上，现在对于任何一个三角形内的点，我们只需要知道其在三角形中的位置，然后用重心坐标做一个插值，就可以算出这一个点的uv。算出来这个点的uv以后，在纹理上查询一下这个uv的值，我们就知道对应纹理的颜色了，然后就可以拿来用了（比如用作漫反射的系数kd，这样就实现了贴图功能）。

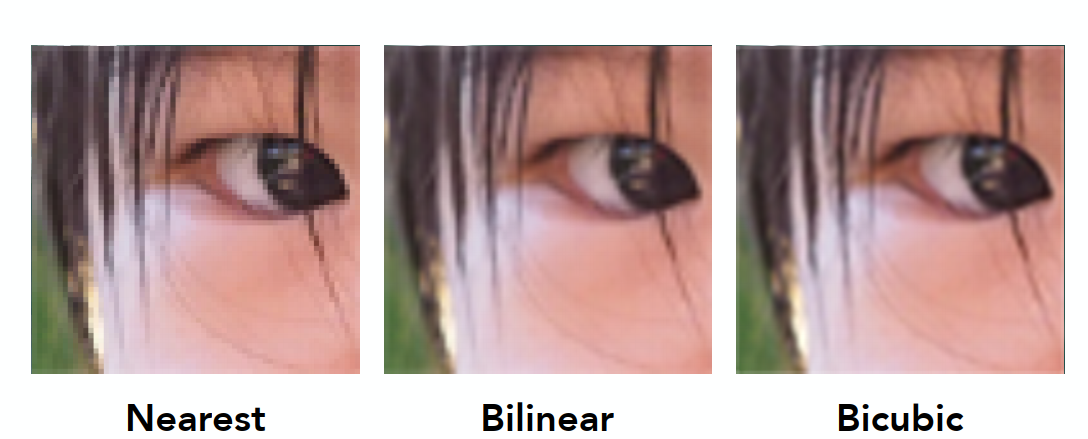


# （二）、Texture Magnification(easy case)（纹理太小了怎么办—线性插值）

## 1、Texture Magnification(纹理放大)

Generally don’t want this — insufficient texture resolution(分辨率)

A pixel on a texture — a texel(纹理元素、纹素)



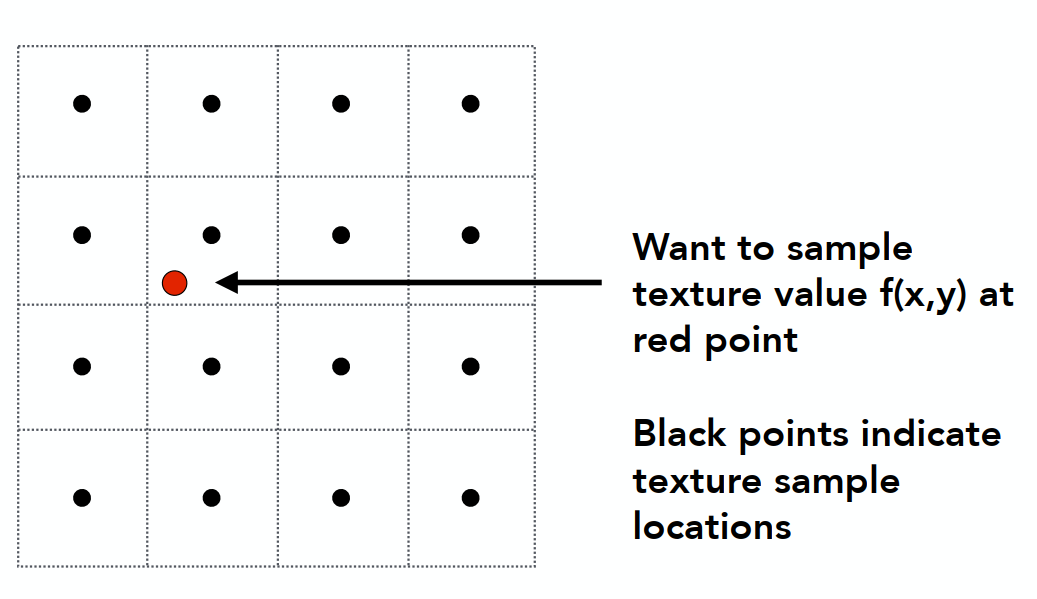
在高分辨率下（比如4k），如果纹理的分辨率比较低（比如256×256），那么在任意一个点上去查纹理，可能会查到一些非整数的值。纹理分辨率小了，就需要将其拉大。

第一种方法是取就近的像素，四舍五入，比如0.4就认为是0,0.6就认为是1，如上图左侧Nearest。但是这个样子很难看。

当查询纹理的时候，如果给了一个非整数的坐标，如何得到它的值？在这里引入双线性插值的概念。

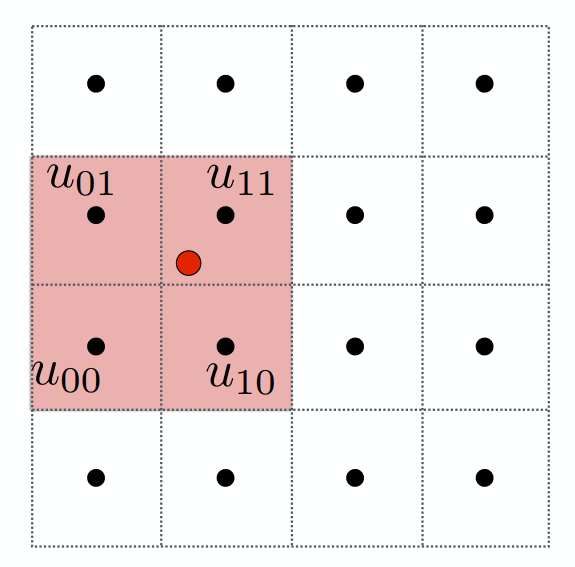
## 2、Bilinear Interpolation

如下图，红点(高分辨率的像素坐标)并没有落在像素中心(纹理的像素中心)（非整数的值），如何计算红点处的纹理值？

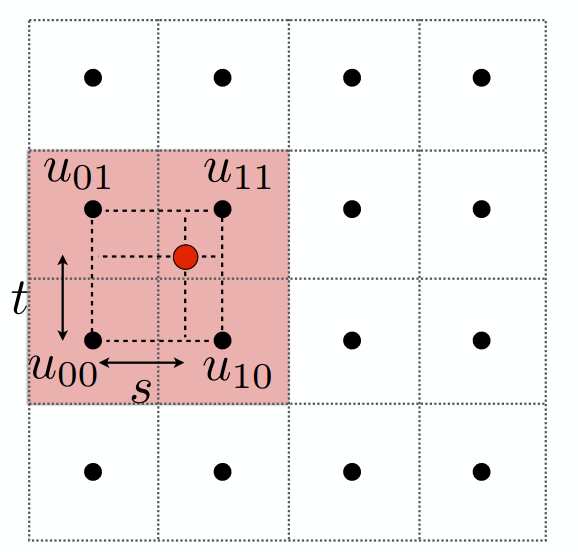


### ①Take 4 nearest sample locations, with texture values as labeled

首先，找到这个红点临近的四个像素点的中心。



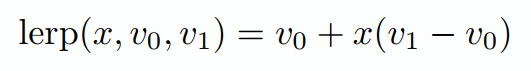
### ②And fractional offsets, (s,t) as shown



这四个点总有一个左下角，可以找到这个红点离左下角的水平距离（s）和垂直距离（t）。s和t的值肯定都是在0到1之间的，因为两个像素之间的距离是1。

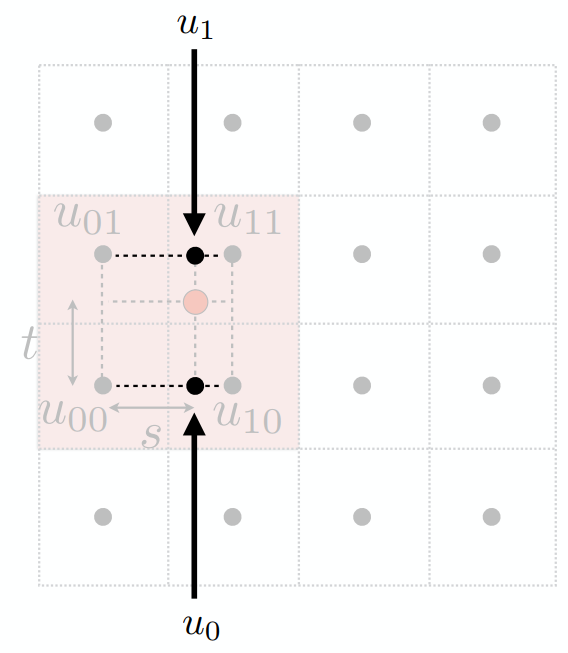
### ③Two helper lerps(horizontal)

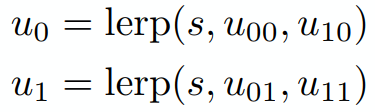
#### Linear interpolation (1D)



首先我们定义一个操作：线性插值。如上图公式。定义v0是0，v1是1，那么x就是位于0到1上的值。当x=0时，lerp的值为v0=0，x=1时，lerp的值为v1=1，x=0.5时，lerp的值为0.5

#### Two helper lerps(horizontal)

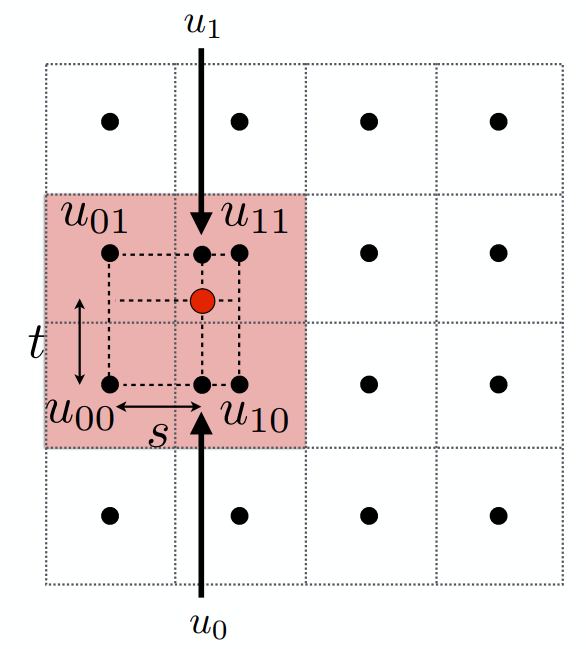




左下角和右下角可以用s做一次线性插值： 左下角u00有一个颜色，右下角u10有一个颜色，又知道u0离左边的距离是s，所以可以用s对这两个点进行一次线性插值

左上角和右上角也可以用s做一次线性插值： 左上角u01有一个颜色，右上角u11有一个颜色，又知道u1离左边的距离是s，所以可以用s对这两个点进行一次线性插值

### ④Final vertical lerp, to get result:





上一步已经通过插值得到了 u0和u1，但是最后我们想得到的是红点的值。那么我们再做一次竖直方向的插值即可。知道u0和u1，又知道红点离u0的距离t，所以可以用t对u0和u1做一次线性插值，即可得到红点的值。

整个过程做了两种插值：第一种插值是水平方向的点的插值，做了两个点（一对点），第二种插值是竖直方向的点的插值，做了一个点。这时红点的颜色就综合了u00, u10，u01, u11四个点，得到了一种平滑过渡的颜色。这个进行了两种插值的方法，就是双线性插值(就是数字图像处理里的那个)。（这里水平和竖直的先后顺序可以反过来）。

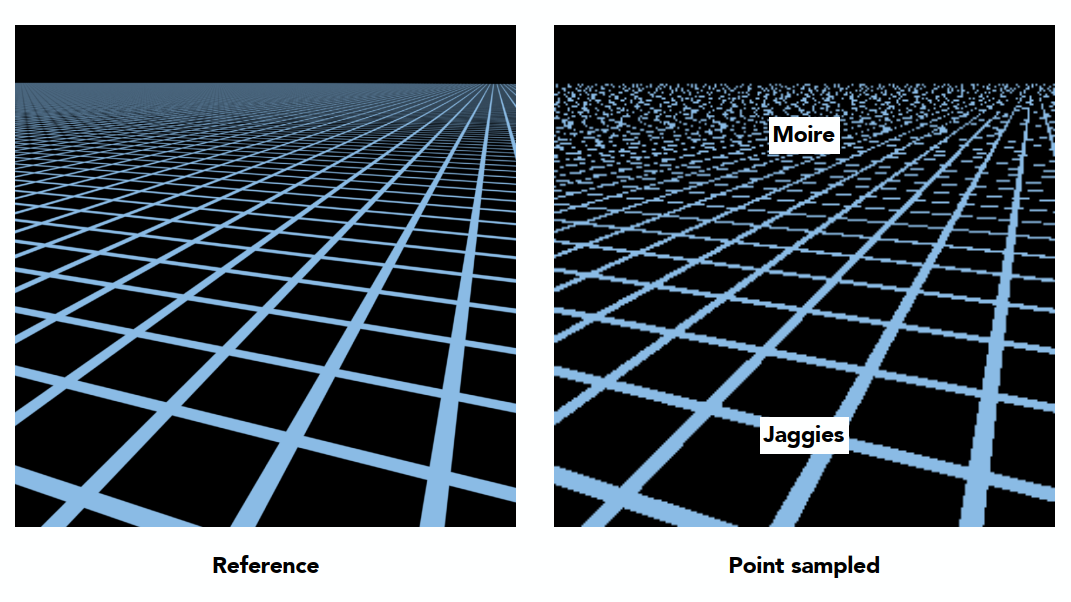
第三种方法是Bicubic，如第一幅图的最右侧效果。这种操作取的是周围16个像素而非4个像素（非线性），虽然效果好但是同时带来的弊端就是开销会很大。所以：

Bilinear interpolation usually gives pretty good results at reasonable costs.

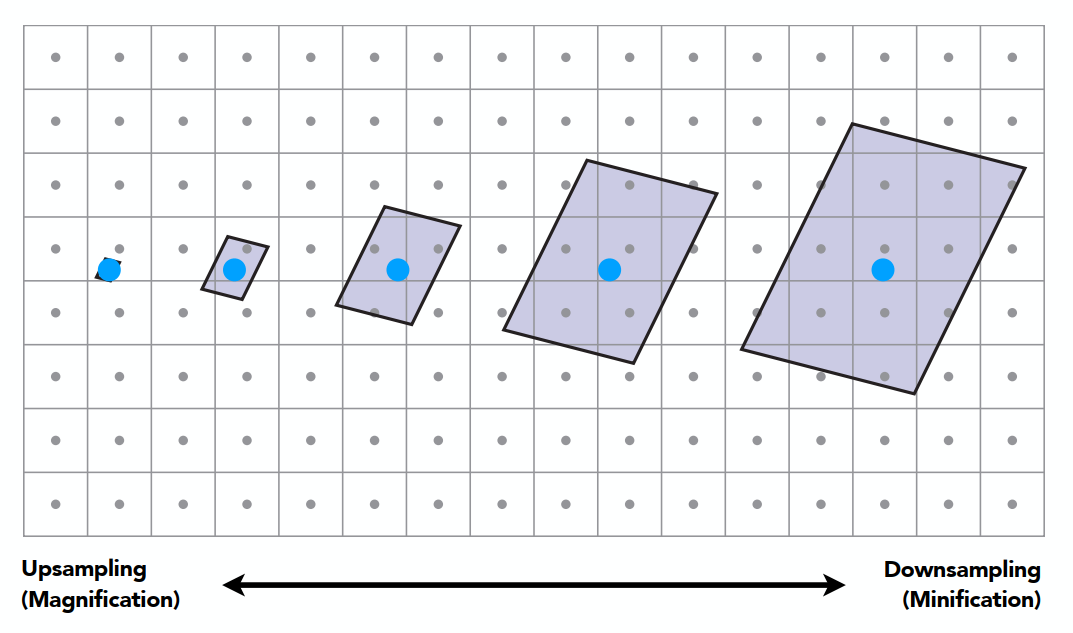
# （三）Texture Magnification (hard case) (如果纹理贴图太大了怎么办?)

## 1、Point Sampling Textures — Problem

当一张uv纹理的分辨率过大，大于屏幕像素的时候，就会出现这样的情况，近处锯齿远处摩尔纹。



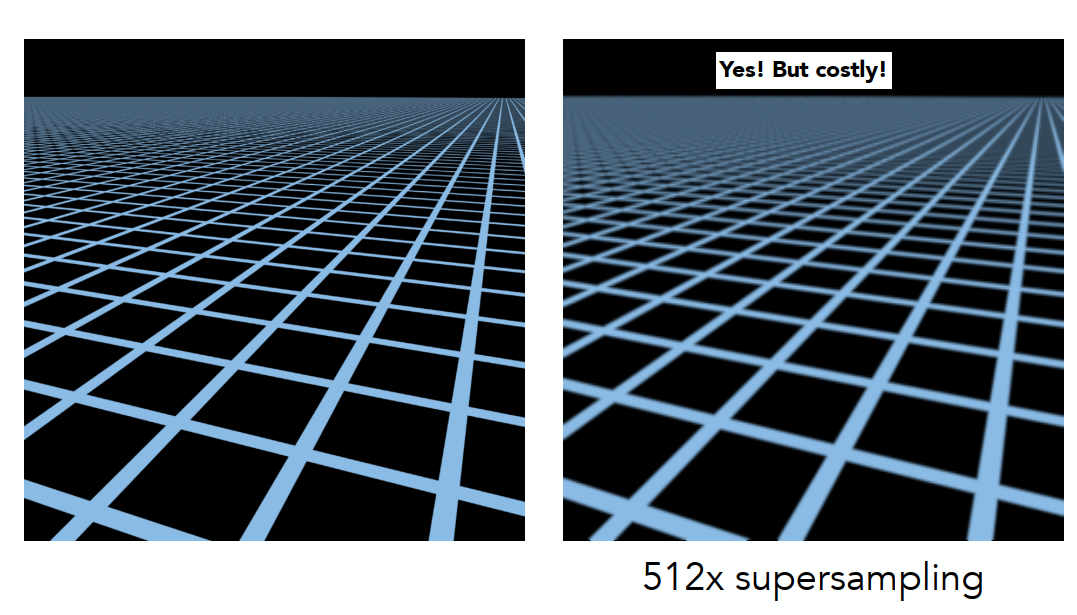
Screen Pixel “Footprint” in Texture



原因是近处一个像素覆盖的纹理区域相对较小，远处一个像素覆盖了很大一片纹理。屏幕上各个像素覆盖的纹理区域各不相同。如果一个像素覆盖的纹理区域较小，用这个像素的中心查询一下纹理的值，没有什么问题。但是一旦一个像素覆盖了很大一片区域的纹理（比如远处接近地平线的纹理），现在我还用一个像素的中心去查这块儿区域，用一个点的值去认为是这一个区域的值，那显然是不对的。用一个点的值无法代表一块纹理区域的颜色变化。

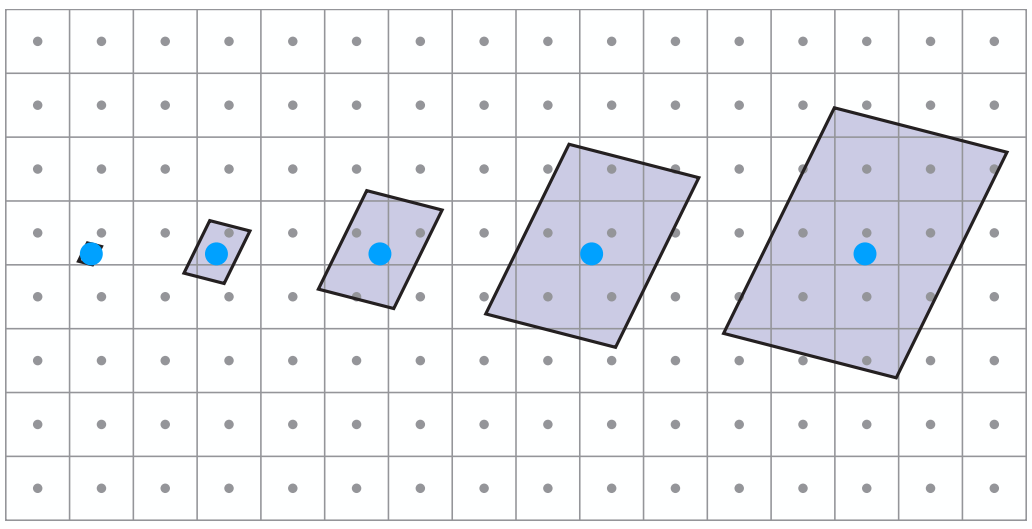
Will Supersampling Do Antialiasing?

虽然说这种问题的解决我们可以用超采样去做，但是开销极大。

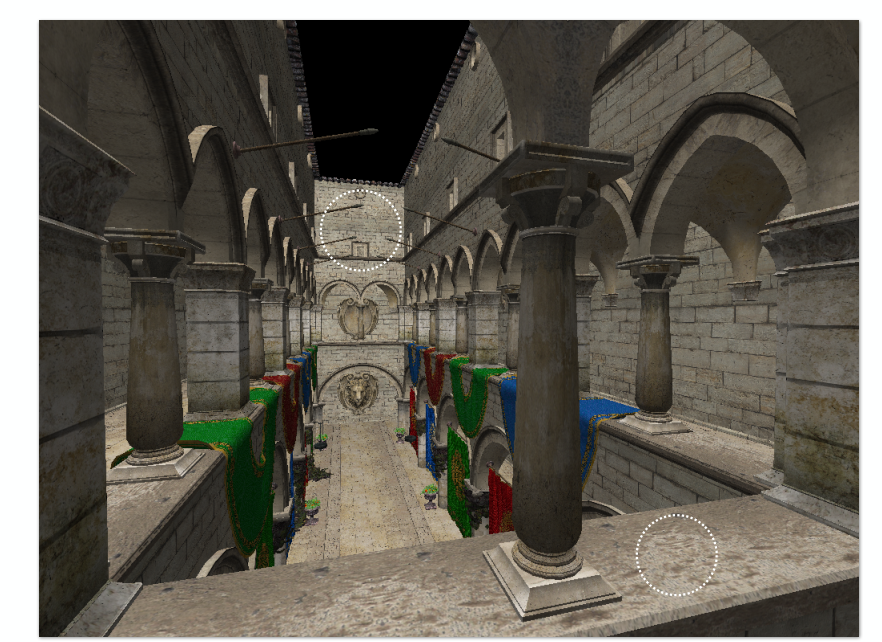


点查询：以纹理为例，给你一个点，它的值是多少，可以用双线性插值来解决，这是点查询。

范围查询：给你任何一个区域，你立刻可以得到这个范围内的平均值，这是范围查询。（范围查询还有查询一个范围内最大值和最小值，这类问题跟现在所说的完全不一样）



Different Pixels -> Different-Sized Footprints

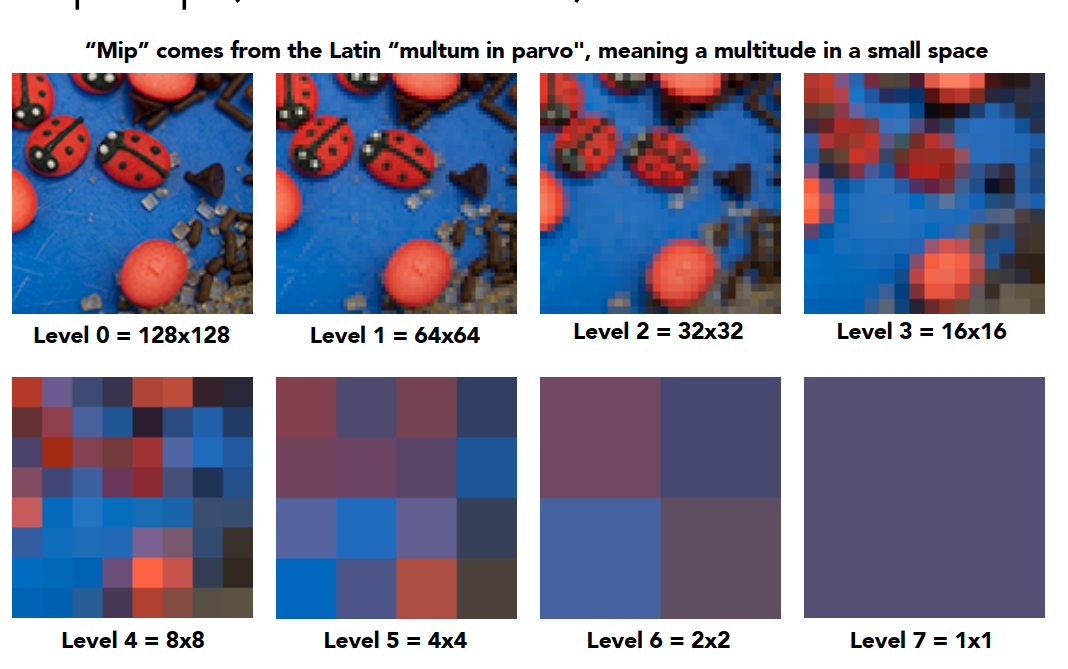


以实际的渲染图为例，近处的圆圈（右下方）一个像素覆盖的纹理区域较小，远处的圆圈（中上方）一个像素覆盖的纹理区域非常大。假设用同一个纹理，不同像素有不同的在纹理上覆盖的大小，所以范围查询应该能查询任意范围的大小。

## 2、Mipmap（Allowing (fast, approx., square) range queries）

Mipmap的范围查询三个特点：快（查询速度非常快）、大约（是查询的近似值）、方形（查询区域仅可以是方形）

“Mip” comes from the Latin “multum in parvo", meaning a multitude in a small space

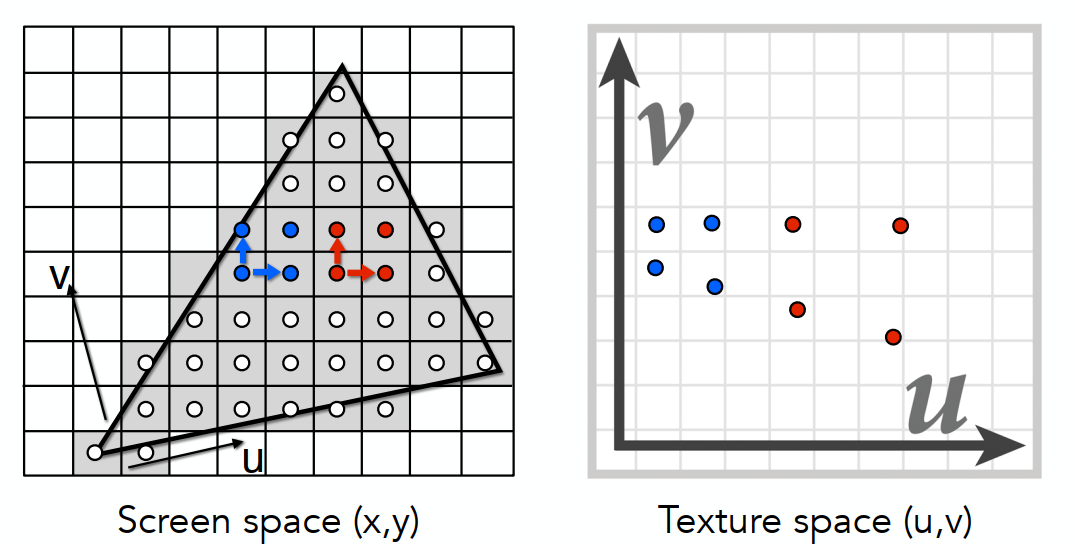


Mipmap其实就是从一张图生成一系列图(图像金字塔)。如上图，原始图是第0层，每升高一层，图像的分辨率都缩小一半。

What is the storage overhead of a mipmap?

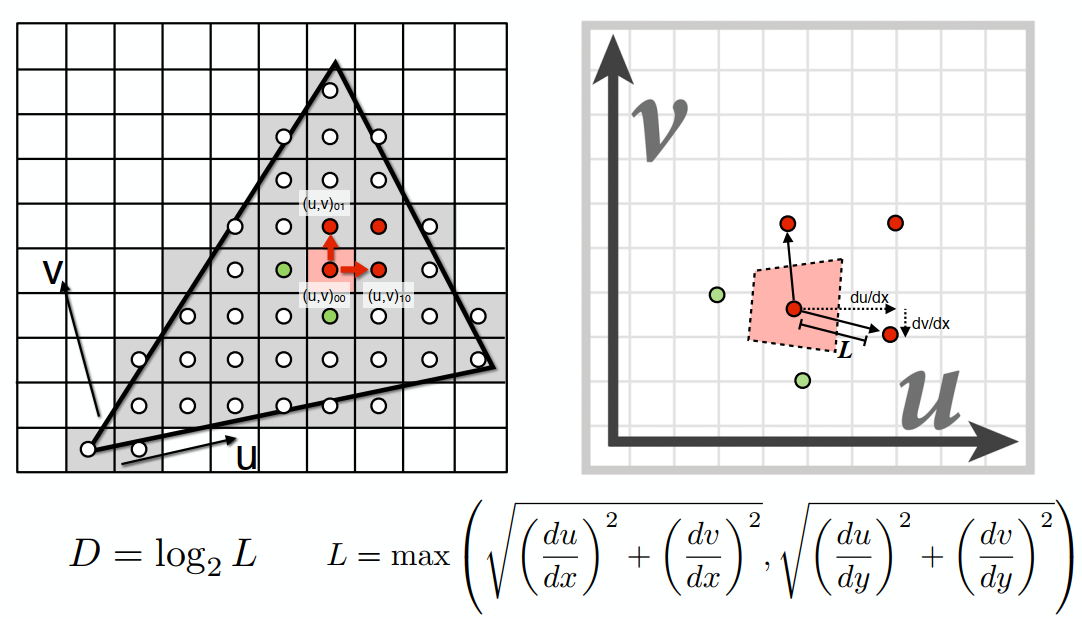
如果原图存储开销是1，因为每一层图分辨率都缩小一半，所以每一层的存储量都是上一层的四分之一。计算下来，总存储量开销是4/3 (1+1/4+1/16+1/64+…)，额外开销仅仅是1/3.

### Computing Mipmap Level D

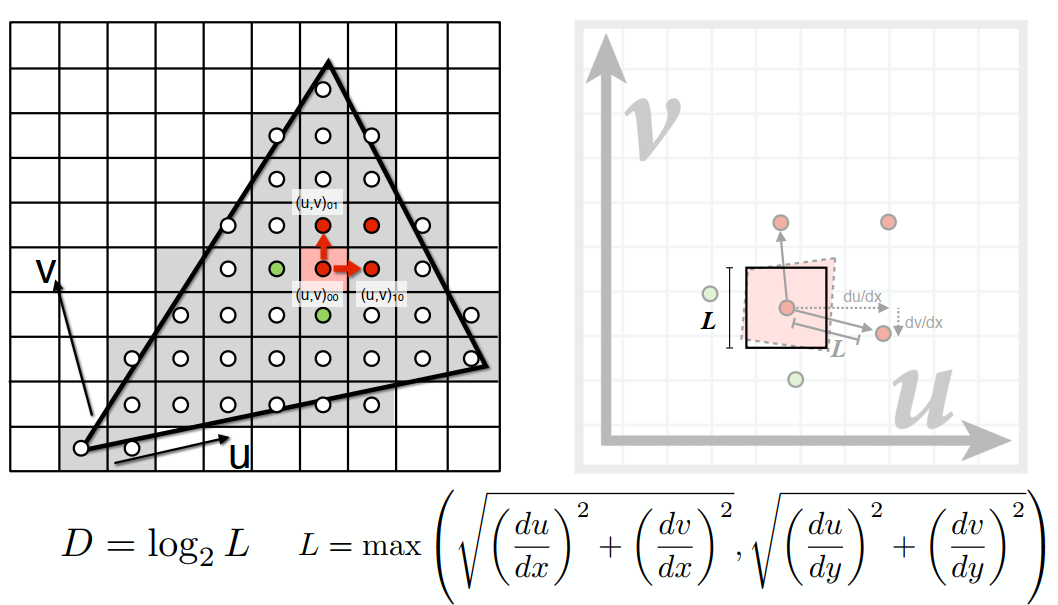


使用相邻屏幕样本的纹理坐标估计纹理足迹

任何一个像素都可以映射到纹理上的一个区域，想算一个像素的覆盖面积，可以让该像素中心和其相邻像素中心分别映射到纹理上去，然后做一个近似(左图像素红点映射到纹理图上的红点)。我们知道相邻两个像素中心的距离是1，那么对应到右边纹理上占有多远的距离L也就可以求出，这时我们就可以求出一个像素映射到纹理上占据多大的面积。



我们可以用如下图所示的一个正方形框来近似这样一个不规则(上图分红四边形)的区域。



当我们把一个像素在纹理上覆盖区域近似成一个正方形时，如何根据我刚刚已经算好的Mipmap，去查询这样一个边长为L正方形的区域的平均值是多少？

如果这个正方形区域就是1×1，那么就表明一个像素正好对应一个边长为L的正方形区域，也就可以直接在最原始（第0层，D=0）的纹理上找对应的像素，就是它的值。

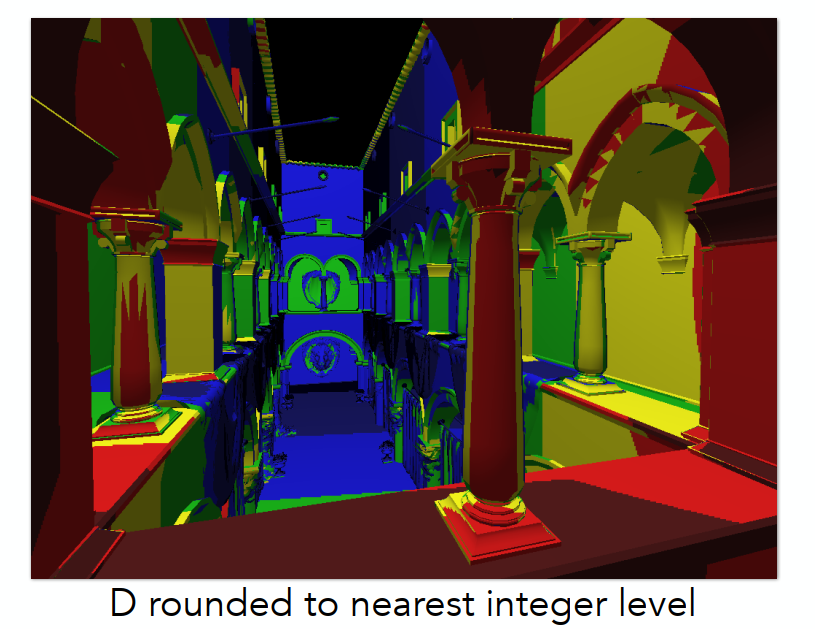
如果这个正方形区域是2×2，那么这个区域会在第1层（D=1）上对应一个像素

如果这个正方形区域是4×4，那么这个区域会在第2层（D=2）上对应一个像素

对于L×L大小的正方形，一定会在D=log2L层上对应到一个像素。

因此我们只需要算出D，即在第几层正方形的区域对应一个像素，就可以得出这个区域内平均值是多少。

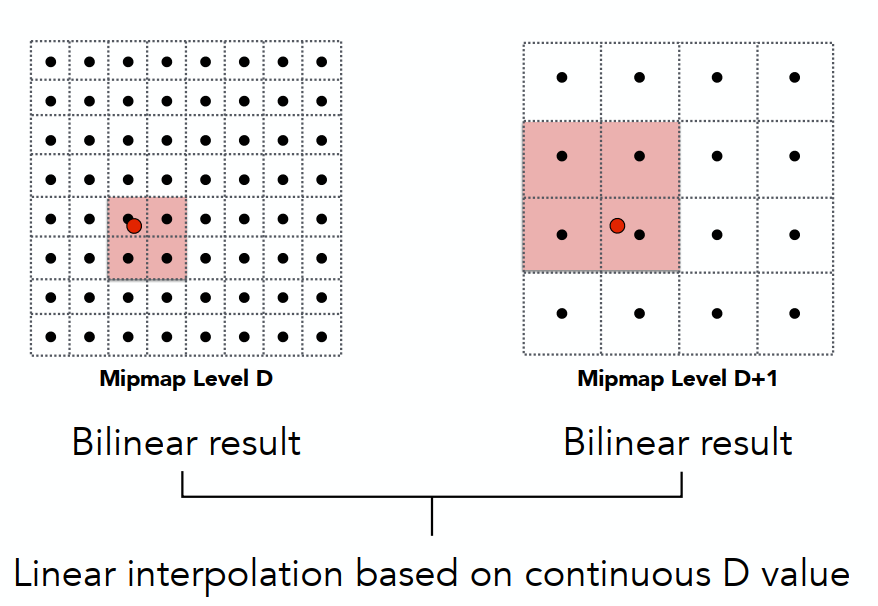
### Visualization of Mipmap Level



将这个分层的过程可视化，如上图所示。

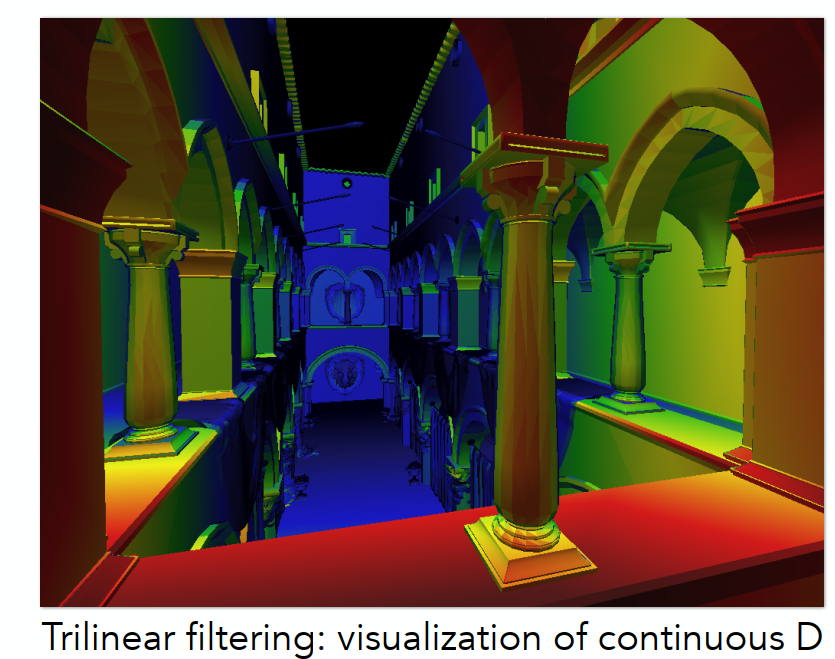
但是会发现，两层之间有很明显的边界(这是因为离散金字塔产生的)，那么在实际纹理映射的过程中可能会出现一些缝。

### Trilinear Interpolation



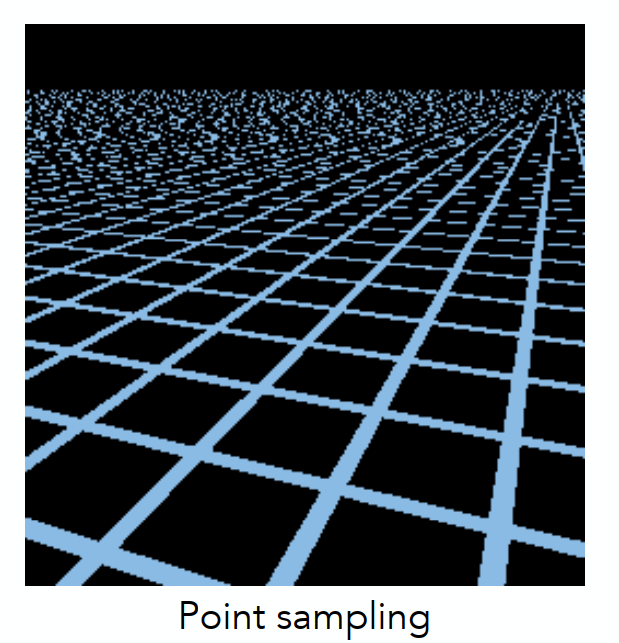
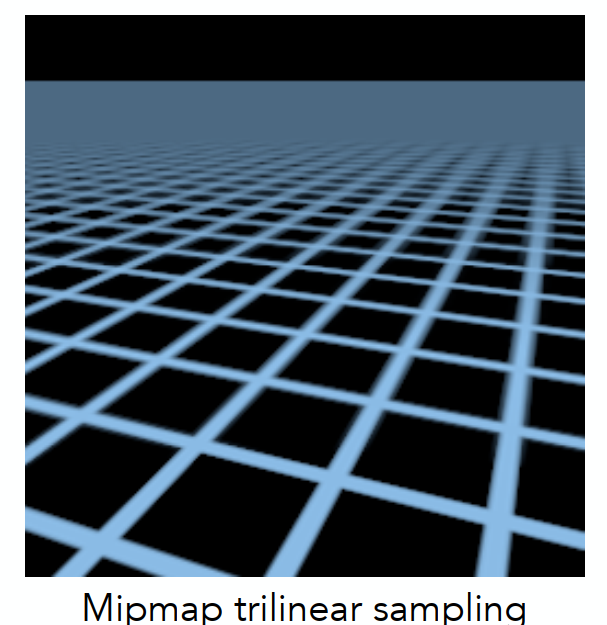
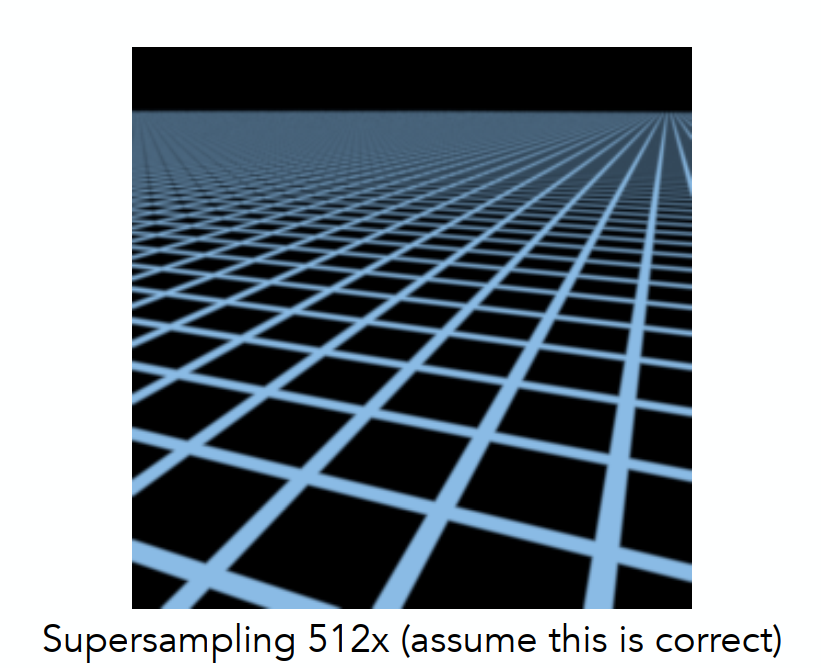
在此处我们的解决方法是，先找D层，再找D+1层，这两层内部分别用双线性插值把对应的在这两层上的查询做出来，做出来之后把这两个双线性插值的值合到一起，然后在层与层之间再做一次插值。总共做了三步插值，在双线性插值上又加了一步插值，这就是三线性插值。这样我就可以在任意层，无论是整数层还是浮点数层（例如第1.8层，D=1.8）找到像素值。

### Visualization of Mipmap Level



可以看到，三线性插值处理后的层级可视化出来，就是一个很漂亮的连续渐变的层级了。

### Mipmap Limitations

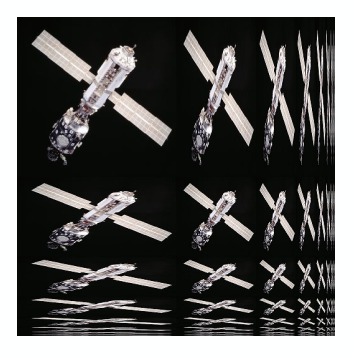
. 

在这里我们可以看到，如果假设用高开销的超采样得到的结果是相对准确的，那么用Mipmap得到的图，很明显在远处过于模糊，远处的所有细节全部糊掉。出现这个问题的原因就在于Mipmap只能查询一个正方形方块的范围。

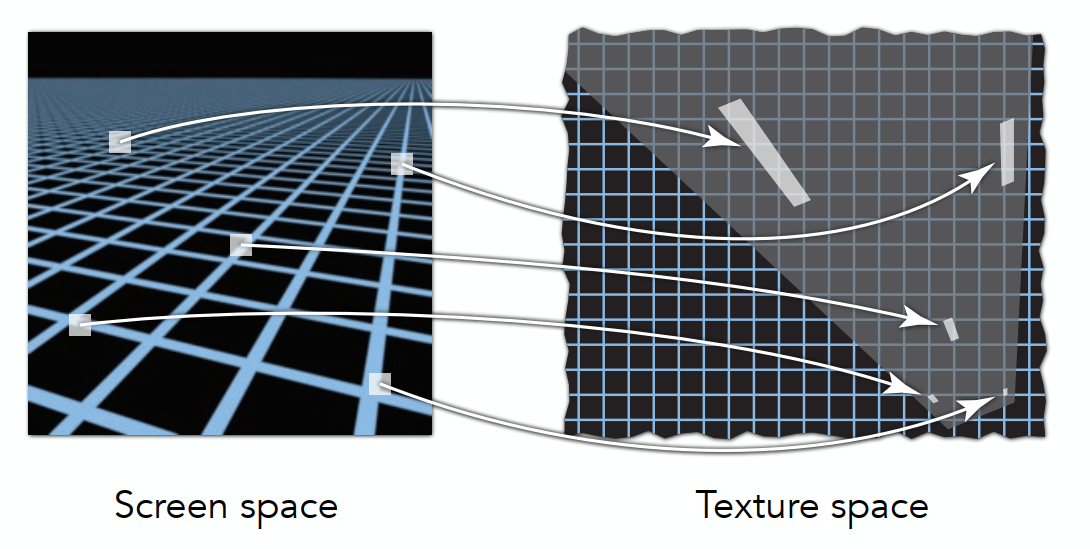
## 3、Anisotropic Filtering(各向异性过滤：三线性插值弊端的改善办法)

各向异性：在不同的方向上表现各不相同（考虑方向性）

各向同性：各个方向上表现完全相同（如刚刚的正方形，水平竖直表现完全相同）

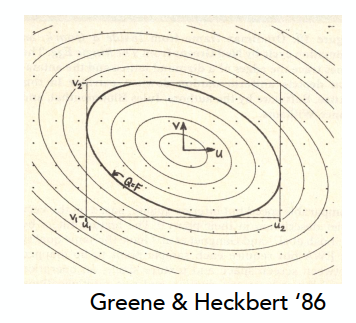


Mipmap操作的相当于是上图的对角线图片，每次长宽各缩小一半，而各向异性可以用不同的长宽比进行缩小（上图除了对角线以外的图片），也就是可以用矩形区域做范围查询（总共开销是原来的三倍）。反映到映射关系上即为下图。



但是，对于一些斜着的图形，依然没有很好的方法去查询。对上图竖直范围好查询

### EWA filtering



人们发明了另外一些方法，如EWA，对于任何一个形状，都可以拆成很多不同的圆形去覆盖这个形状。如上图查询一个椭圆，将其拆成三个圆形，每次去查询一个圆形，多次查询自然就可以得到一个区域，但是代价是“多次查询”。可见质量越高的效果，性能开销越大。