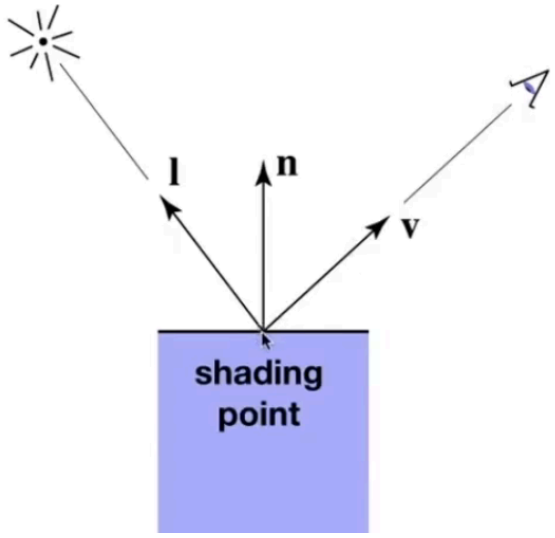
来了，遮挡关系。Z-buffering，深度缓存。

画家算法的症结在于 3d-item 之间并没有完全的深度偏序。

z-buffer 很合理。

漫反射+高光+环境光是一个很初步的光照着色模型，并不是 PBR。

漫反射



法线？我听过法线贴图！说正经的，这里的光源方向应当由观测方向决定，因为什么呢，因为直觉。（还是不正经

着色是个局部的过程，也就是局部表面反射，不涉及物体间可能的遮挡阴影。

光线方向和法线方向的夹角定义了光强，另外，法线和光线不同向时，不应该接受光线。（因为这里考虑的不是透明或半透明物体）

光强 $\frac{I}{r^2}$ 意味着可以通过光源与着色点的距离算光强。

上面几个结合起来就是课上给的公式了：

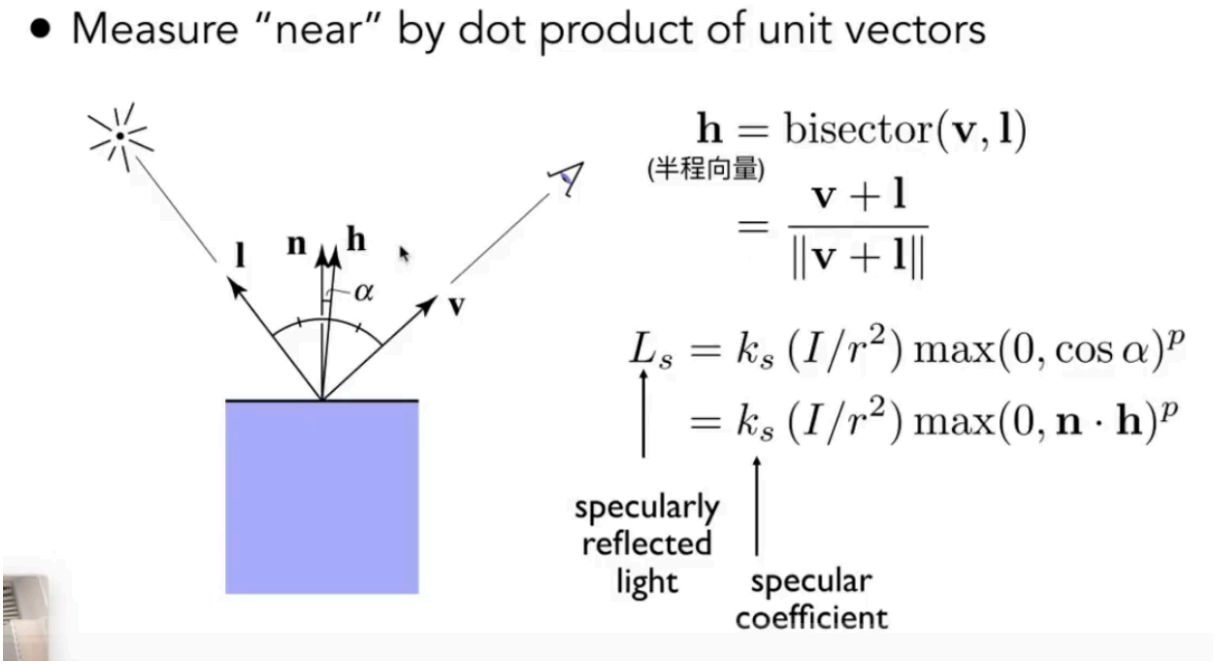
$$L_d = k_d\left(\frac{I}{r^2}\right)\max(0, l\cdot n)$$

（注意这里 l 和 n 都是单位向量，而且 cos 函数的值在 [-1,1] 这个区间）

当然这个公式描述的是漫反射，它与观测方向无关。

高光

高光是镜面反射，意味着只有在特定角度才能看见。



这个判定方式确实很聪明。对 p 次方的解释用了容忍度这个词，之前没听过，挺新颖的。

环境光

草率的假设，但对于非 PBR 足够了。

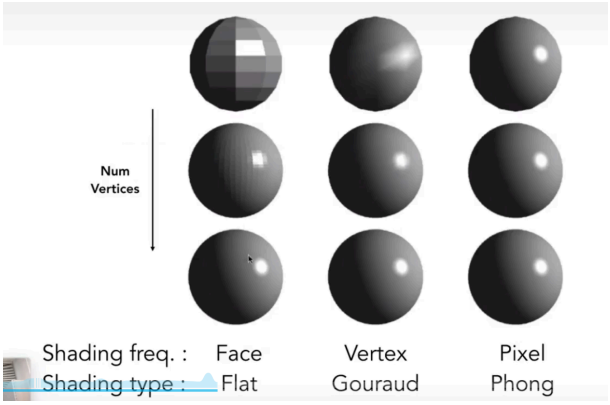
着色频率，或者说着色分辨率

flat shading：对每个平面着色

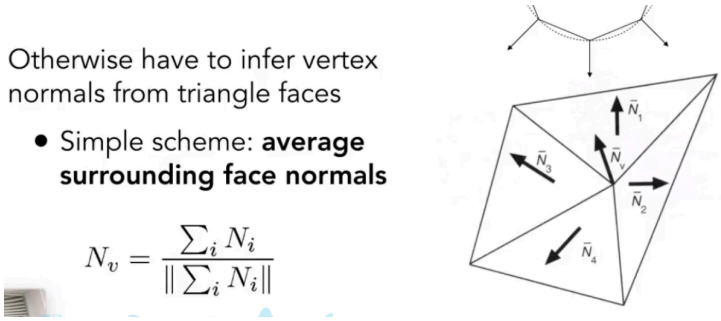
gouraud shading：对每个顶点着色，每个三角形面内部的着色基于其三个顶点的着色做插值。

phong shading：依旧是求出顶点法线，但不插值着色，而是对每个像素对应的三角形位置插值法线，然后独立着色。

可以看到，低面数情况下以 Pixel 单位着色效果最好，开销也最大。不过面数可以消弭差距哦（不只是效果，开销也会抹平）。



顶点着色的法线细节，注意要以三角形面积为基准加权平均。



纹理

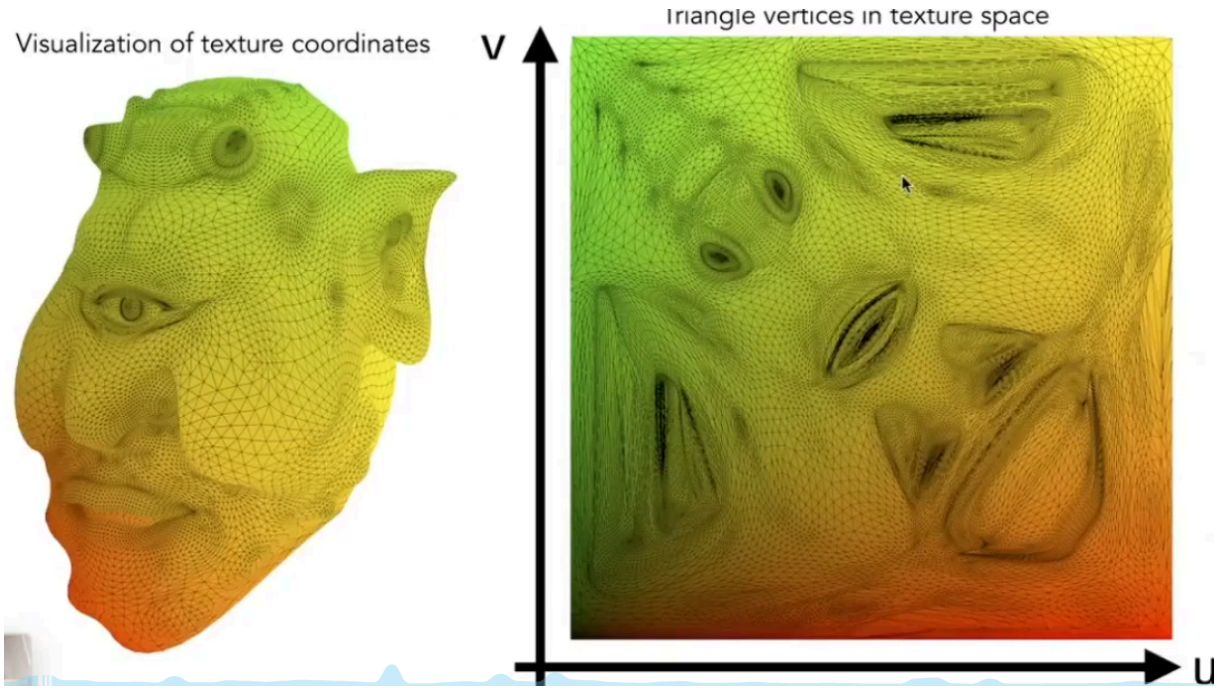


图 1　uv 贴图

uv 贴图也是个 [0,1]^2 的空间。

做一下作业 2。

首先写一下小函数。

```
static bool insideTriangle(int x, int y, const Vector3f* _v)
{
    // TODO : Implement this function to check if the point (x, y) is inside the triangle represented by _v[0], _v[1], _v[2]
    Eigen::Vector3f v1 = _v[1] - _v[0], v2 = _v[2] - _v[1], v3 = _v[0] - _v[2];
    bool _b1 = (v1.x() * (y - _v[0].y()) - (x - _v[0].x()) * v1.y()) > 0;
    bool _b2 = (v2.x() * (y - _v[1].y()) - (x - _v[1].x()) * v2.y()) > 0;
    bool _b3 = (v3.x() * (y - _v[2].y()) - (x - _v[2].x()) * v3.y()) > 0;
    return ((_b1 == _b2) && (_b2 == _b3));
}
```

然后看一下框架里的 buffer 是怎么定义的，倒是符合猜想。

```
void rst::rasterizer::clear(rst::Buffers buff)
{
    if ((buff & rst::Buffers::Color) == rst::Buffers::Color)
    {
        std::fill(frame_buf.begin(), frame_buf.end(), Eigen::Vector3f{0, 0, 0});
    }
    if ((buff & rst::Buffers::Depth) == rst::Buffers::Depth)
    {
        std::fill(depth_buf.begin(), depth_buf.end(), std::numeric_limits<float>::infinity());
    }
}

rst::rasterizer::rasterizer(int w, int h) : width(w), height(h)
{
    frame_buf.resize(w * h);
    depth_buf.resize(w * h);
}
```

于是写代码。

```
for(int i=0; i<700; ++i) {
    for(int j=0; j<700; ++j) {
        float x = i + 0.5, y = j + 0.5;

        if(insideTriangle(x, y, t.v)) {

            int idx = i * 700 + j;

            auto[alpha, beta, gamma] = computeBarycentric2D(x, y, t.v);
            float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
            float z_interpolated = alpha * v[0].z() / v[0].w() + beta * v[1].z() / v[1].w() + gamma * v[2].z() / v[2].w();
            z_interpolated *= w_reciprocal;

            if(z_interpolated < depth_buf[idx]) {
                depth_buf[idx] = z_interpolated;
                frame_buf[idx] = t.getColor();
                // 这里其实不能写 .color
            }
        }
    }
}
```

画出来了，虽然感觉不太对，嗯哼。