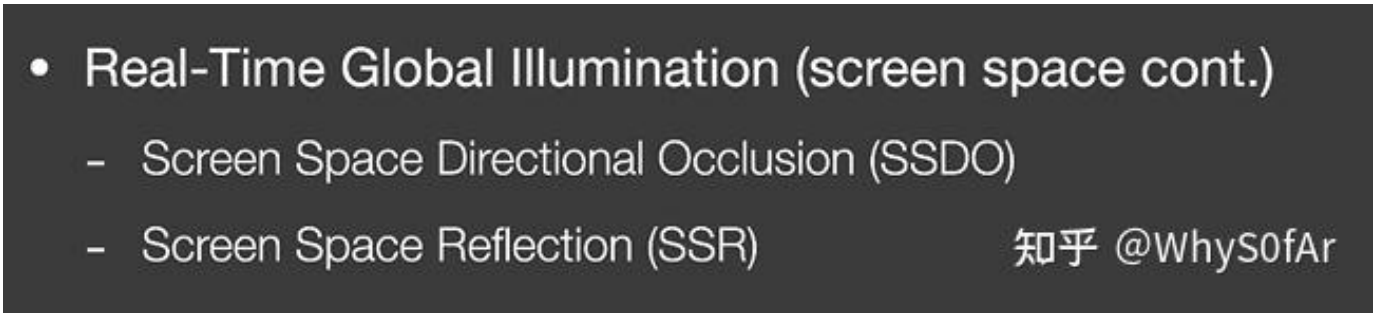
GAMES202 高质量实时渲染笔记Lecture09：Real-Time Global Illumination (Screen space cont.)

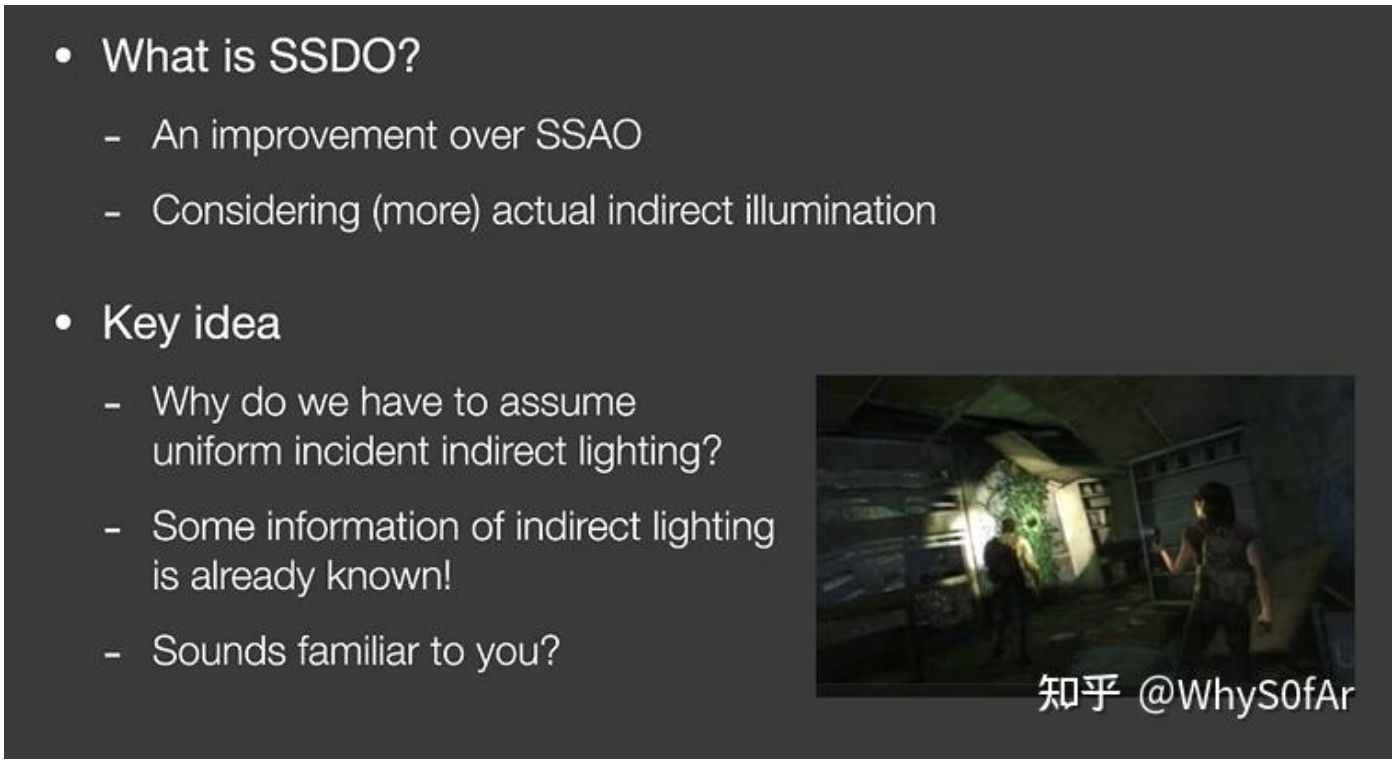
GAMES202 高质量实时渲染笔记Lecture09：Real-Time Global Illumination (Screen space cont.)



本节课主要去讲述剩余的两种Screen Space的GI算法思路:**SSDO和SSR.**

I) Screen space Direction Occlusion （SSDO）屏幕空间方向遮蔽

什么是SSDO?

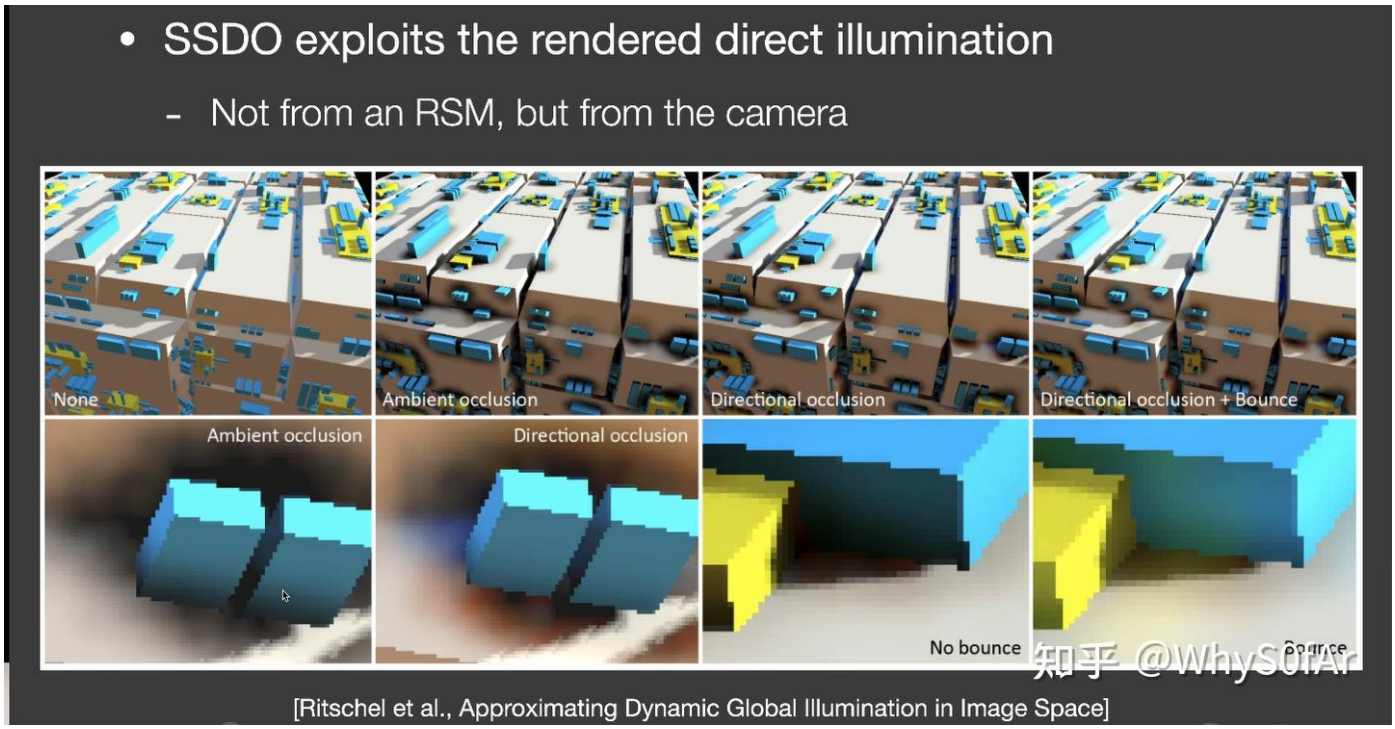


- 对SSAO的一个提高/升级；
- 比起AO考虑间接光照是一个常数,在DO里我们更精确的考虑了间接光照。

关键思路

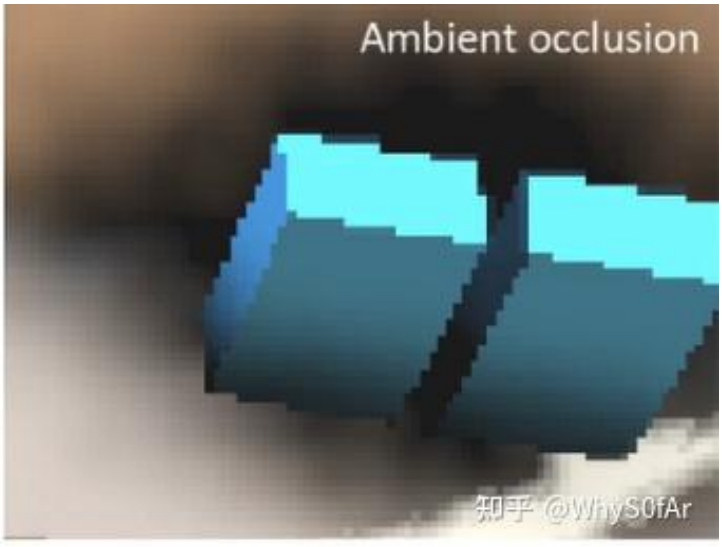
- 我们不去假设间接光照是固定不变的；

- RSM中我们用shadow map去找到接收直接光照的点当作间接光照为其他的Shading point提供直接光照,也就是说**我们一定程度上是可以已经得到间接光照的信息。**

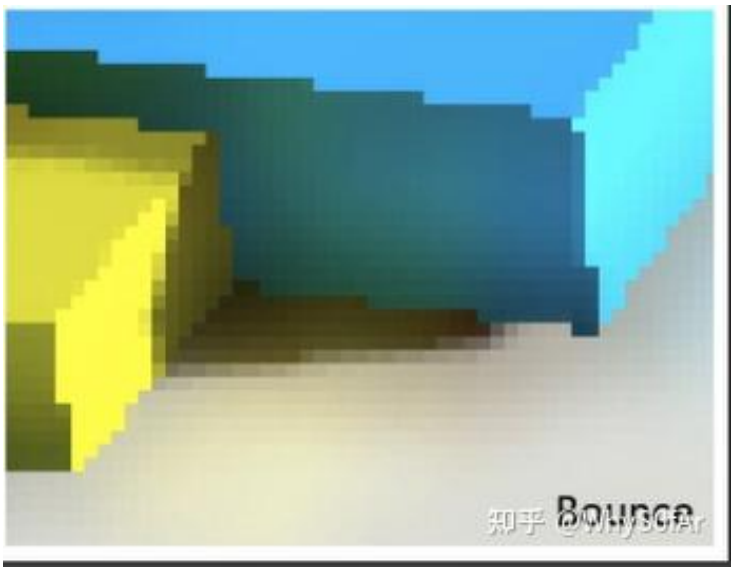


SSAO VS SSDO

通过AO和DO的对比我们可以看到,AO能够产生变暗的效果使得物体相对感更强烈,但AO并不能做到Clolor Blending（不同颜色的Diffuse会互相照亮）



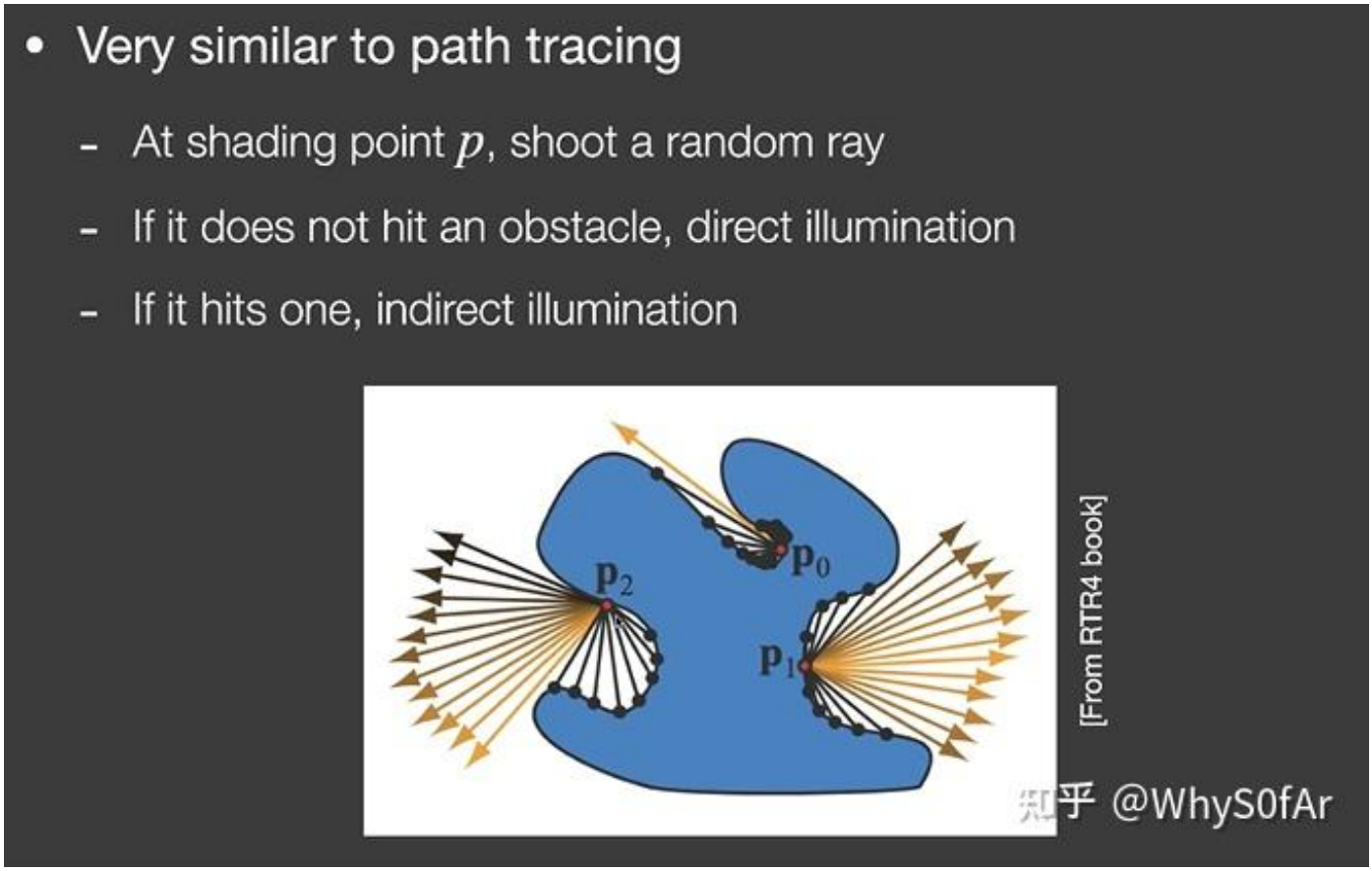
而我们想要的是更加真实具有color blending的效果



如图,我们可以看到蓝色面上会接收到一点黄光，黄色面上也会有一点蓝光,而并不是像AO一样简单的遮蔽位置整体暗一点，因此DO是更加精准的获得全局光照的一种方法。

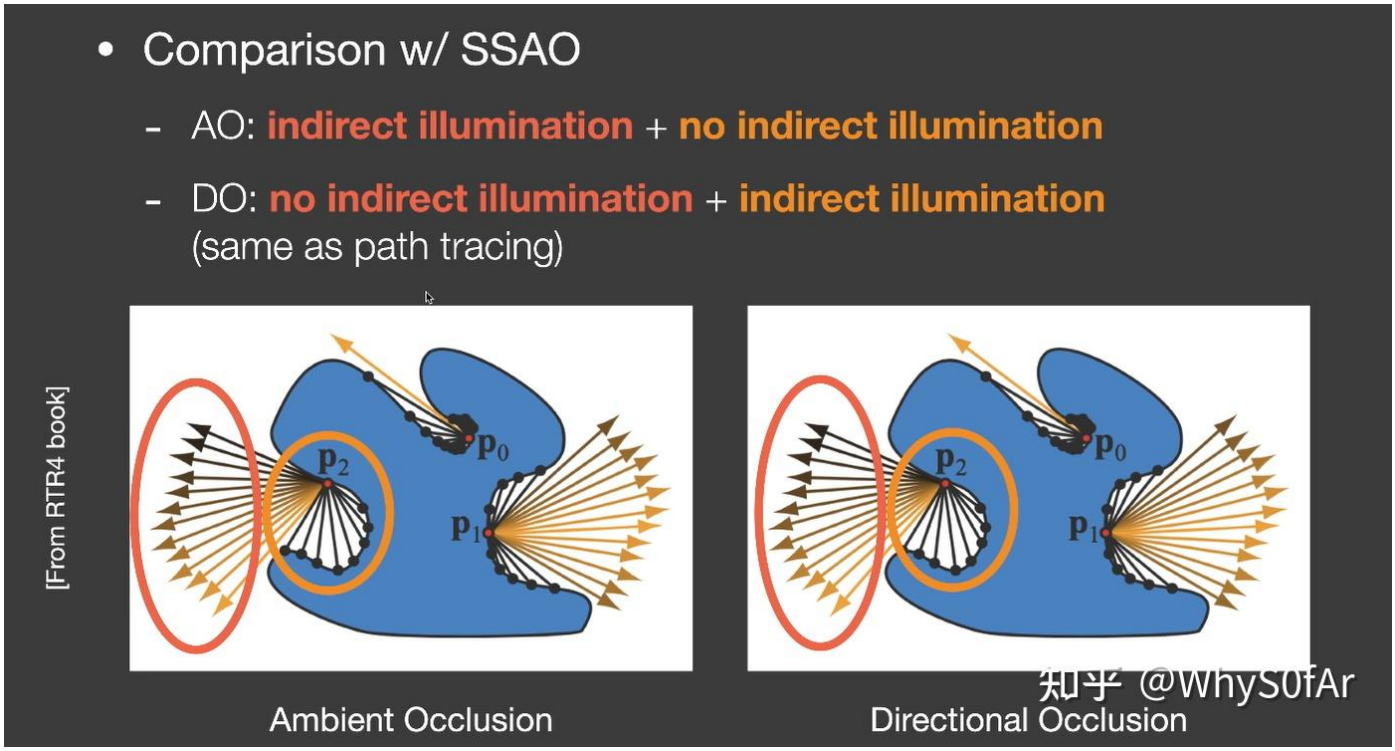
- **在SSDO中,我们要用直接光照的信息,但不是从RSM中获得,而是从screen space中得到.**

做法:



SSDO的做法于path tracing很像,假设在Shading Point的P点，随机的往某一个方向打出一根光线:

1. 如果光线没碰到物体，则认为P点这里接收直接光照
2. 如果碰到了一个点Q,那么算出Q点接受的直接光照打到P点的贡献,从而求出P点的间接光照。



我们可以发现,SSAO和SSDO是完全相反的两个假设:

AO：在AO中我们认为红色的框里能接收间接光照，黄色框里无法接收间接光照，然后求出加权平均的visibility值,也就是**假设间接光照是从比较远的地方来的**；

DO：在DO中,我们认为红色框里接收的是直接光照,而黄色框里才是接收到的间接光照.因为红色框里的光线打不到用来反射的面，因此这些方向上就不会有间接光照，黄色框里的光线能打到物体上，P点接收到的是来自红色框的**直接光照** +黄色框里的**间接光照** ,也就是**假设间接光照是从比较近的反射物来的。**

其实这两个假设都不是完全正确的，物理真实的情况是这两种的混合：近处的是DO，远距离是AO，因此AO与DO也并没有矛盾。

回到渲染方程上,将没有遮蔽的与遮蔽的方向上的光照分开考虑，那么对于DO如何解Rendering Equation:

- 当V=1时是直接光照，而DO的计算是计算间接光照的，因此这个我们完全不用去计算与考虑

$$L_o^{\text{dir}}(\mathrm{p},\omega_o)=\int_{\Omega^+,V=1} L_i^{\text{dir}}(\mathrm{p},\omega_i)f_r(\mathrm{p},\omega_i,\omega_o)\cos\theta_i\,\mathrm{d}\omega_i$$

- 当V=0时也就是间接光照的情况，这个是我们需要关注与计算的。

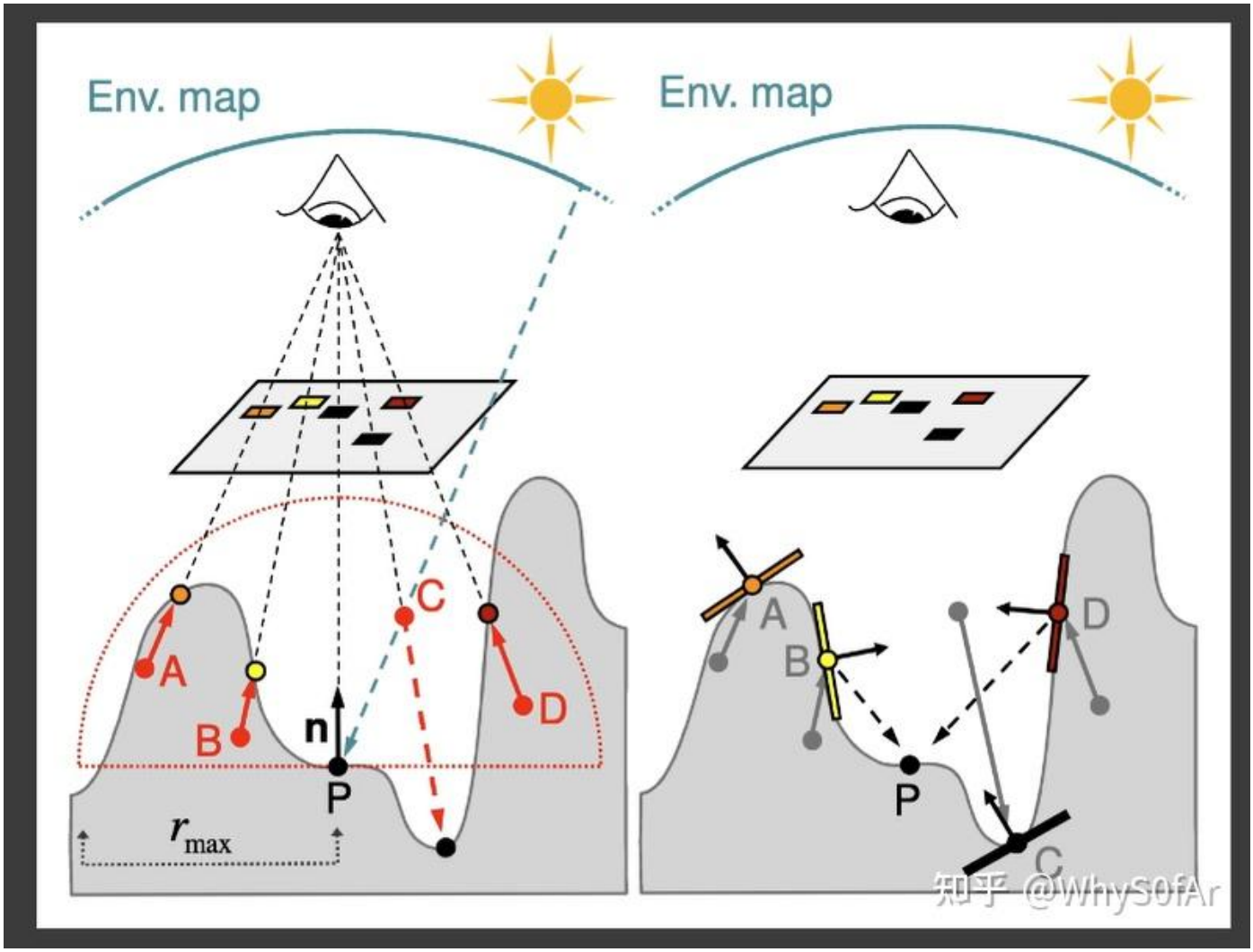
$$L_o^{\text{indir}}(\mathrm{p},\omega_o)=\int_{\Omega^+,V=0} L_i^{\text{indir}}(\mathrm{p},\omega_i)f_r(\mathrm{p},\omega_i,\omega_o)\cos\theta_i\,\mathrm{d}\omega_i$$

从一个pixel或者patch计算间接光照在lecture07-下半部分的RSM中我们讲过了。

SSDO的核心是要找哪些patch会被挡住，也就是对点P提供间接光照贡献的是哪些点，做法是与AO完全一样的。

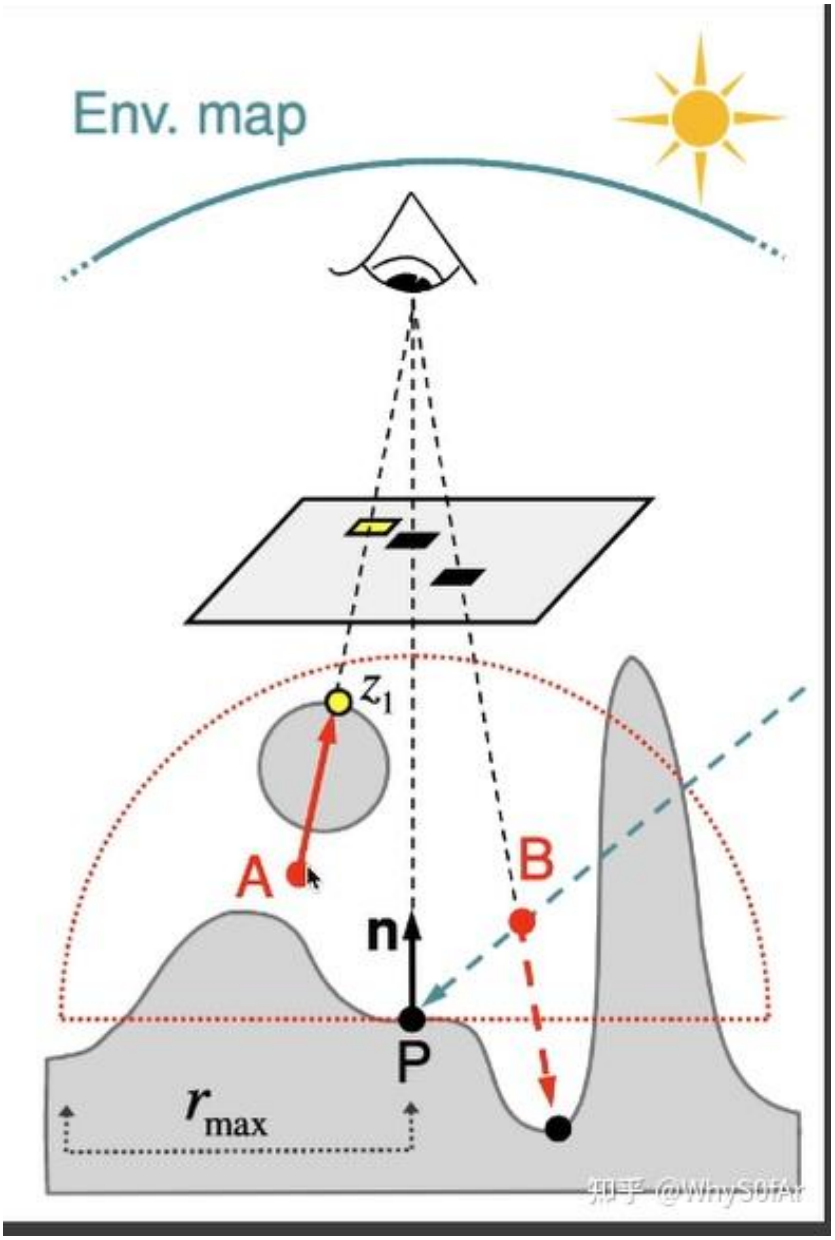
我们同样考虑点P法线部分的半球，判断从P点往A、B、C、D四个方向看会不会被挡住，由于是屏幕空间的算法,因此这里我们同样不考虑在3D场景中A,B,C,D四点会不会与P连成光线,只考虑从camera看去A,B,C,D与P连成的光线会不会被挡住。

这里A/B/D这三个点的深度比从camera看去的最小深度深,也就是说PA,PB,PD方向会被物体挡住,因此会为P点提供间接光照。然后把我们用在RSM中讲的计算间接光照的方法这些点对P的贡献加起来。



SSDO也会出现一些问题,如下图是假设与实际情况不同的情况，因为我们是在屏幕空间处理的,不判断实际上的场景物体遮挡关系，而是从camera看去半球内点的深度和最小深度相比，求出哪些点是可以提供简介光照的，因此在A点会被canmera看不到，SSDO中也认为AP之间也看不到，实际上AP之间是不会挡住的。因此在SSDO中A点错误的提供了间接光照

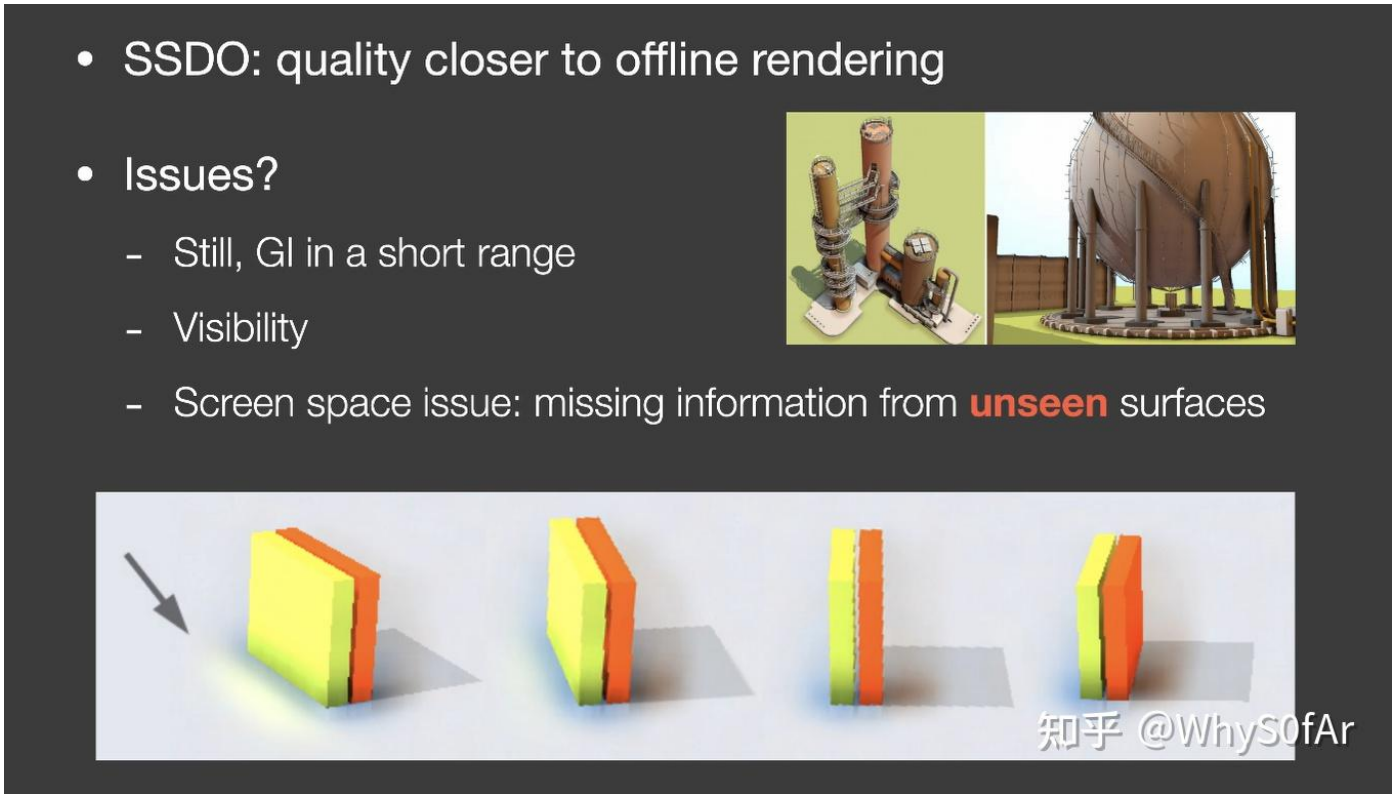
给P点。（感谢@LordedbyAlta同学的指出，已改正）



从计算量上来看与SSAO差不多，但是不同之处是，判定会被挡住的时候，会额外计算被挡住的小片的贡献，质量非常接近离线渲染。

问题:

P点对于半球上的点可见性是通过Camera对这些点的可见性来近似计算的，存在于屏幕空间中丢失信息的问题，下图是一个很明显的例子，当黄色的面朝向屏幕的时候地面的SSDO信息是正确的，而当旋转过去之后，就看不到SSDO的信息了。



SSDO只能解决一个很小范围内的全局光照，下图是接近正确的情况，而如果使用SSDO来计算，方块右边是追踪不到远处绿色的墙的，方块上也就不会有绿色的反光。

II) Screen space Reflection（SSR）屏幕空间反射

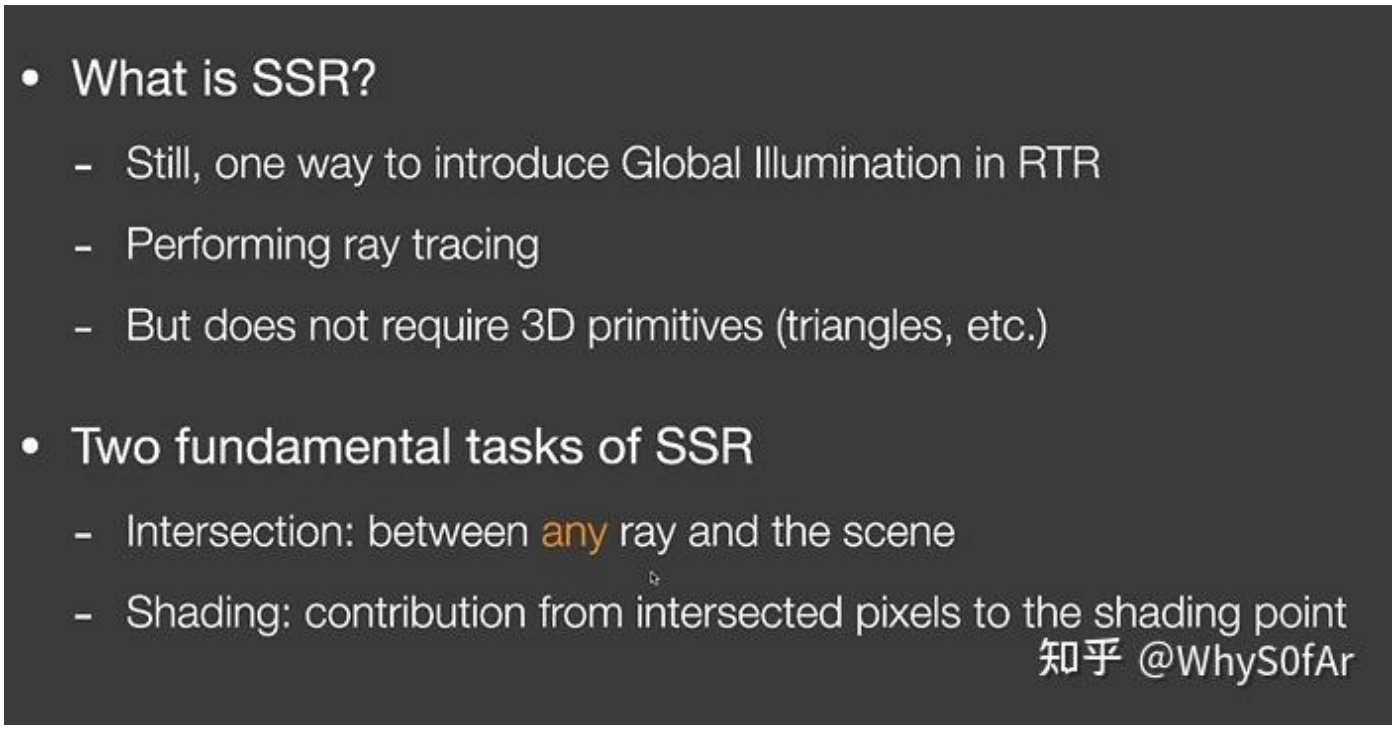
老师说更愿意把他叫做Screen space Raytracing(SSR),只把它理解成反射的话其实是听狭隘的.

首先什么是SSR?

(肯定不是酒吞,茨木,妖刀姬他们**)**

- 仍然是一种在RTR中实现GI的方式；

- 是在屏幕空间做光线追踪
- 不需要知道3D空间中的三角形、网格、加速结构等3D信息，只需要屏幕空间中已有的信息，也就是从camera看去场景的得到的这样一层“壳”。

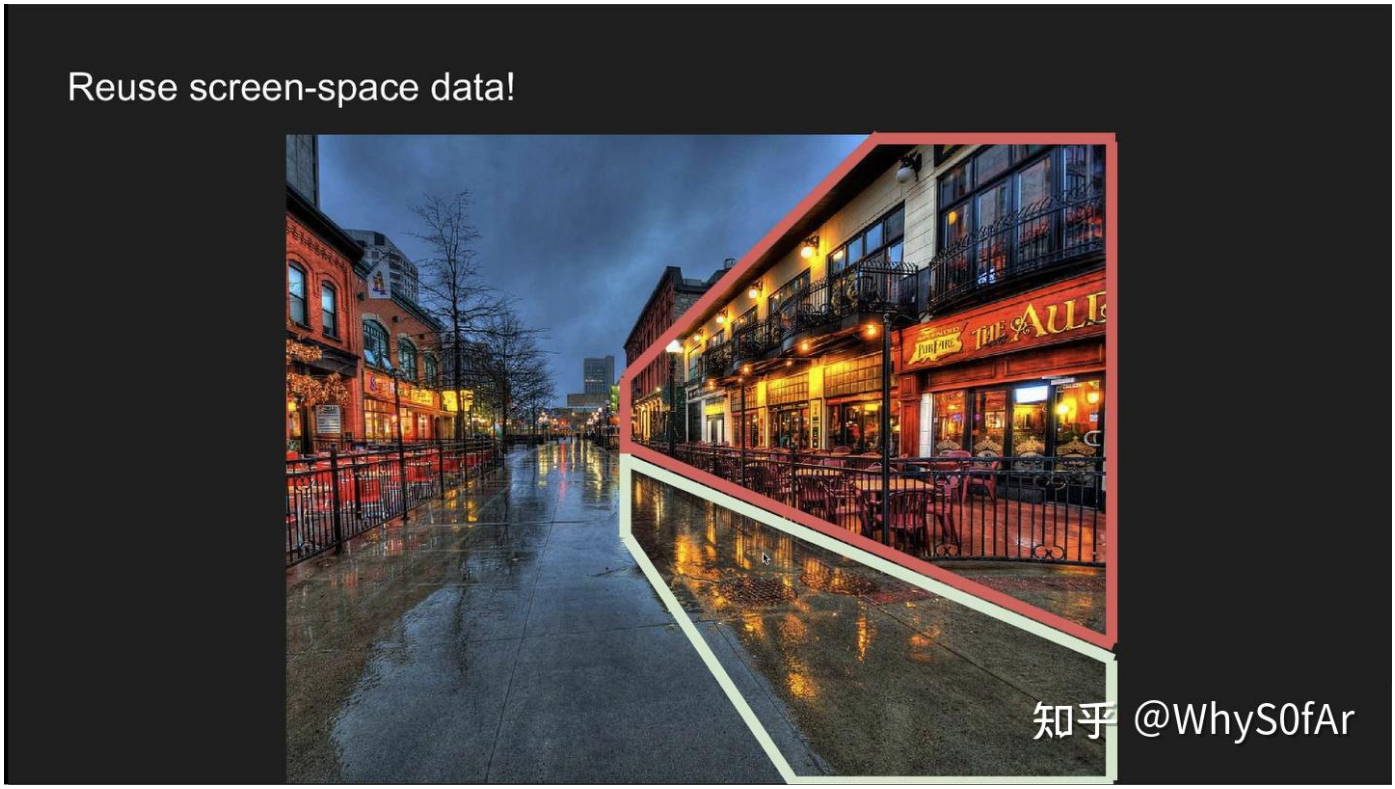


- 由于我们认为它是screen space raytracing,我们考虑的是**任何光线** (不单单是反射光)与场景中这层壳去做求交.
- 找到交点后,算出对shading point的贡献值.

我们用生活中的一些实例来看:

在这个海景房中,地面会反射出窗户,椅子之类场景中的信息.

图中地面会反射出灯光和建筑物本身的一些信息，因此反射本质上来说就是全局光照。



那么任何一个点反射的信息是什么,是从哪里得到的?

白框里反射的是红框内的场景信息,也就是说我们并不需要3D场景的什么信息,而是从屏幕空间里已有的信息得到,也就是反射的绝大多数信息是屏幕内已有的信息，这就是屏幕空间反射的核心思路。

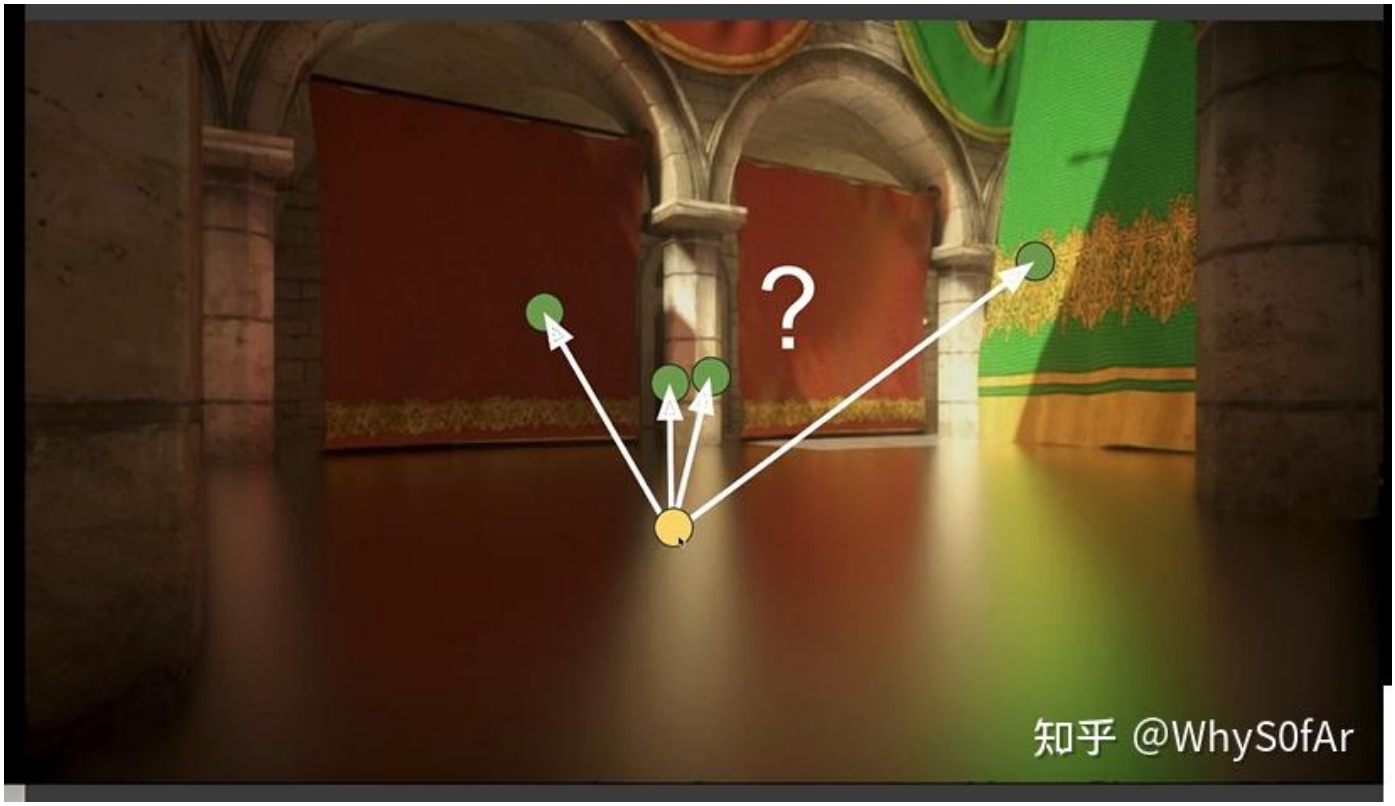
基础的SSR算法：镜面反射

假设场景中已经渲染出来了上面的部分，对于地面还没有进行渲染，如何把反射的信息加进去。

对于任何一个像素:

- 知道shading point的观察方向后,可以得出其反射方向
- 从Shading point点沿着反射方向延长找到与屏幕的壳的交点
- 将交点的颜色作为反射的颜色记录到shading point。

除了可以做specular反射,还可以做glossy反射:



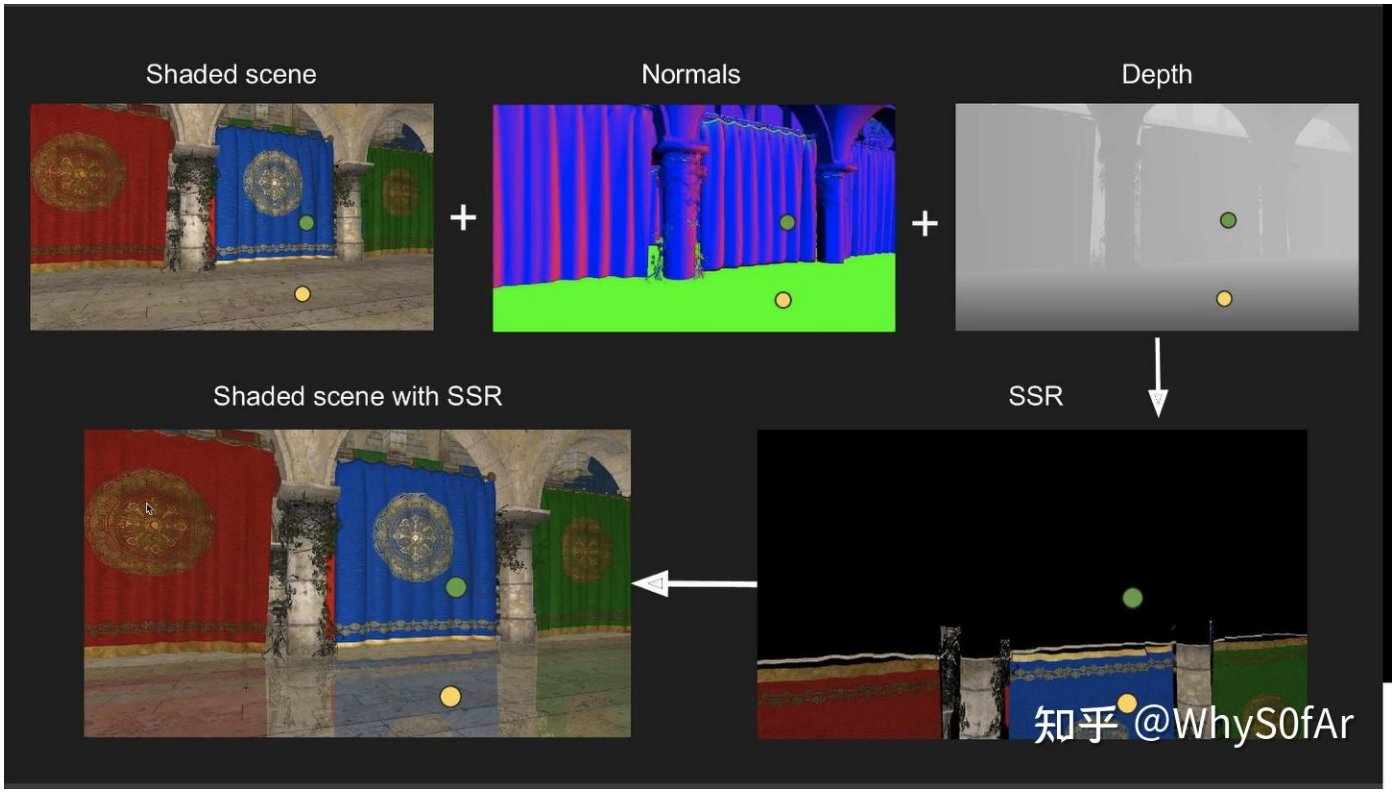
glossy反射

如果说specular反射只需要反射一个方向的话,那么对于一定roughness的材质来说我们要根据其BRDF的Lobe来考虑,如果lobe比较细,则需要很少的光线,如果lobe比较越大,像diffuse那种的话我们需要射出很多根光线

即使是地面凹凸不平,也就是在地面的各法线方向不一致的情况下,仍然可以得到反射的结果.

也就是说当我们知道几何和normal时,不论是什么物体,我们都可以往任何一个方向去trace

即在知道几何和法线信息后,SSR可以做任何的光线追踪



我们来总结一下SSR的思路:

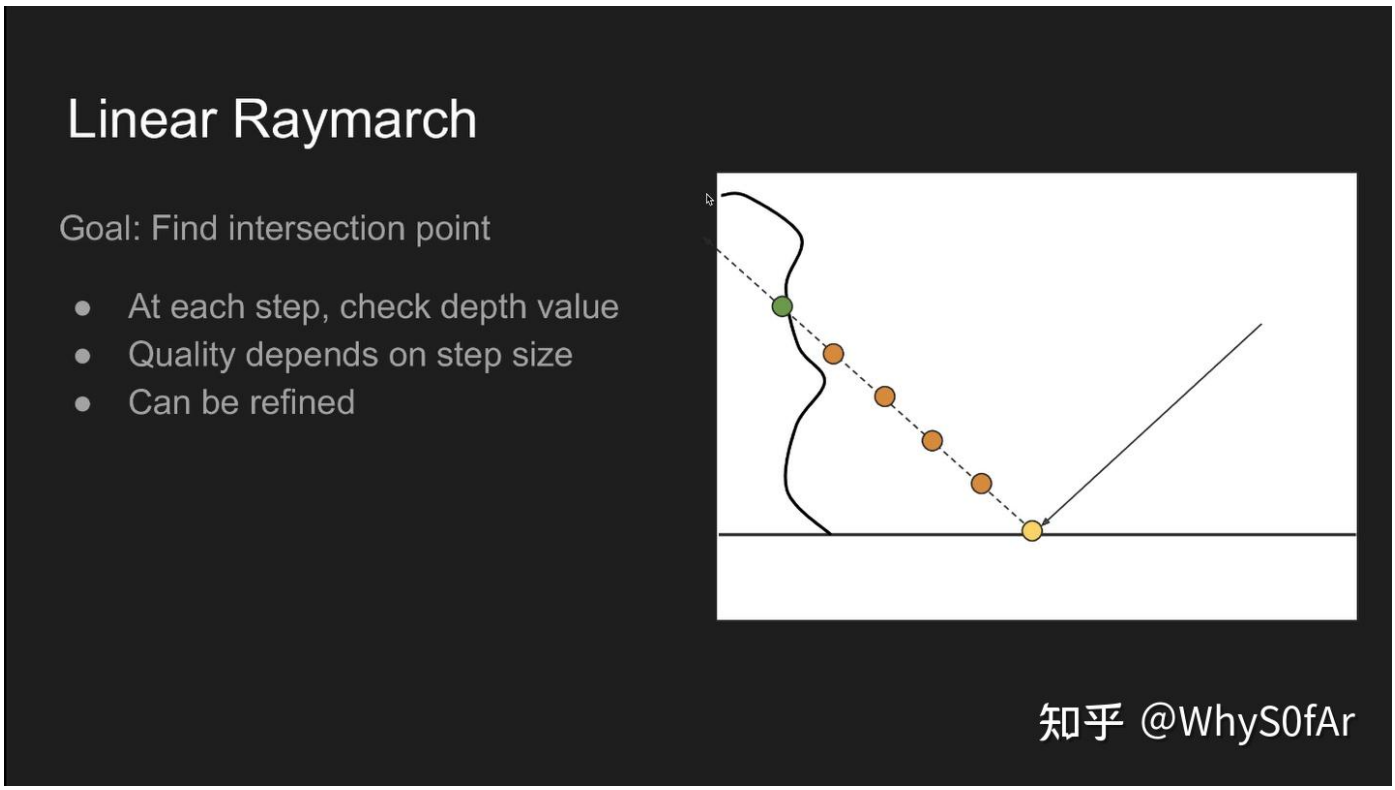
specular反射:

- 我们有一个还没有进行反射的场景,如shaded scene上的地面还并没有得到反射的结果
- 得到图中的normal信息和深度信息
- 进行SSR,我们想要的是对于地面的每一个pixel,我们都想计算出其反射到场景中的得到的值是多少
- 得到反射的值后,将结果加到场景中去,也就是在地面上的黄点反射到场景的壳上的绿色,进行求交算到的结果加入到黄点中
- 得到一个镜面反射的效果

怎么求反射光与场景的相交?

最简单的方法是:**Linear Raymarch**

黄色是shading point,虚线是反射光,假设camera在右边,那么从camera看去,场景的壳就是曲线.

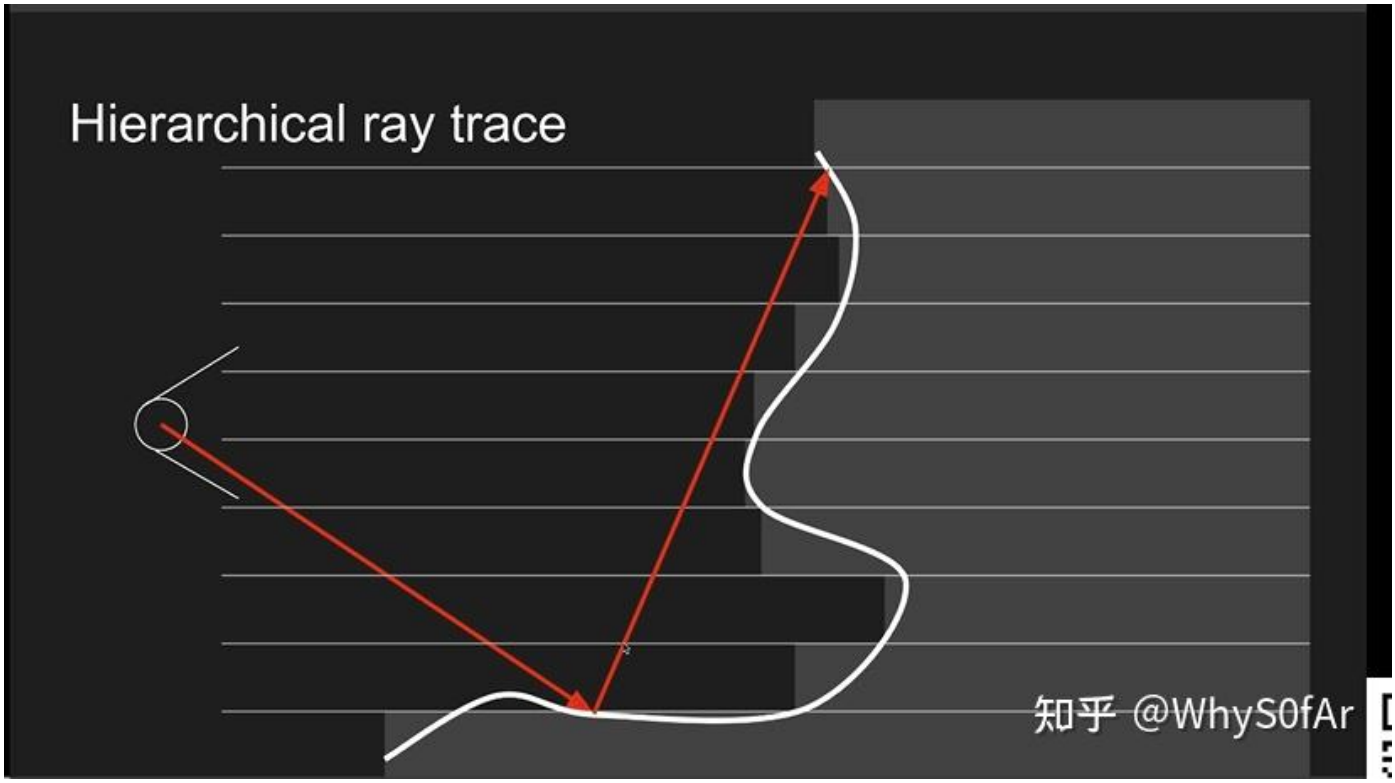


我们是为了找到反射光与场景“壳”的交点:

- 沿着反射方向以一个固定的步长逐步前进,并将每次停止时的深度与壳的深度进行比较,如果浅于壳,则继续前进,比壳深,则停止求交,也就是我们用**深度** 来进行**可见性判断**
- 质量取决于步长的大小，步长小越精准，同时计算量也越大，因此步长太大太小都不行，在没有SDF的情况下，步长只能是一个定值。

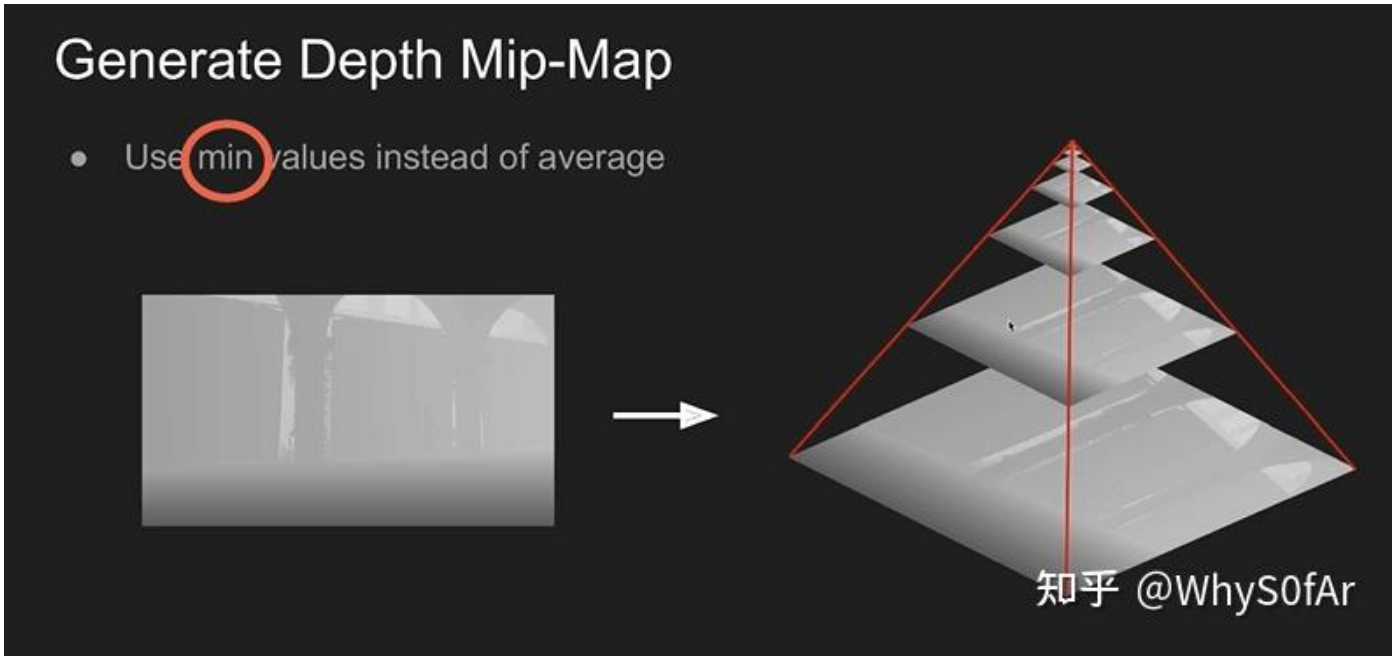
由于步长是由我们来决定的,太长太短都有其各自的问题,因此我们引入另一种动态决定步长的方法:

Hierachical ray trace

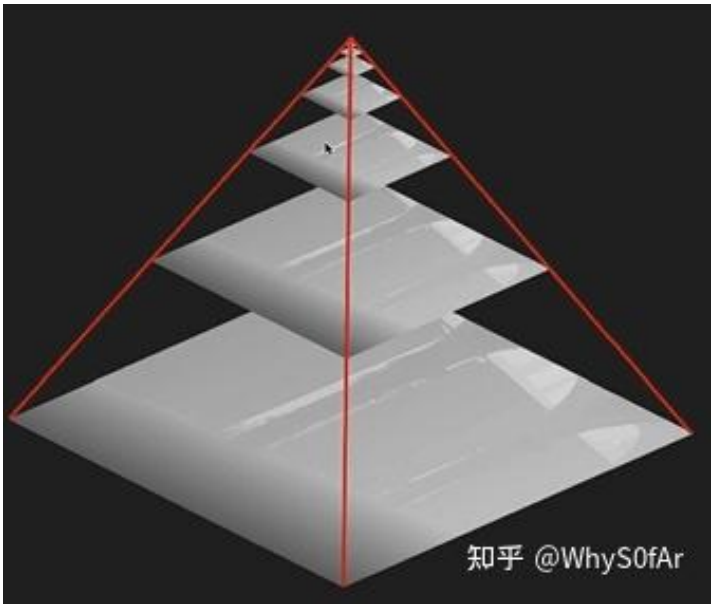


我们可以看到右边那条是反射光,我们认为一个像素是一格,我们在只trace一根反射光线的情况下,,如果按一格一格的走,每一个fragment shader都要做深度比较,那么八格就是八次,很浪费时间,而且在这个图上,我们就算一次走四格都不会与物体相交,如果我们可以快速的得到一次往前走几格这个信息的话是很美妙的.

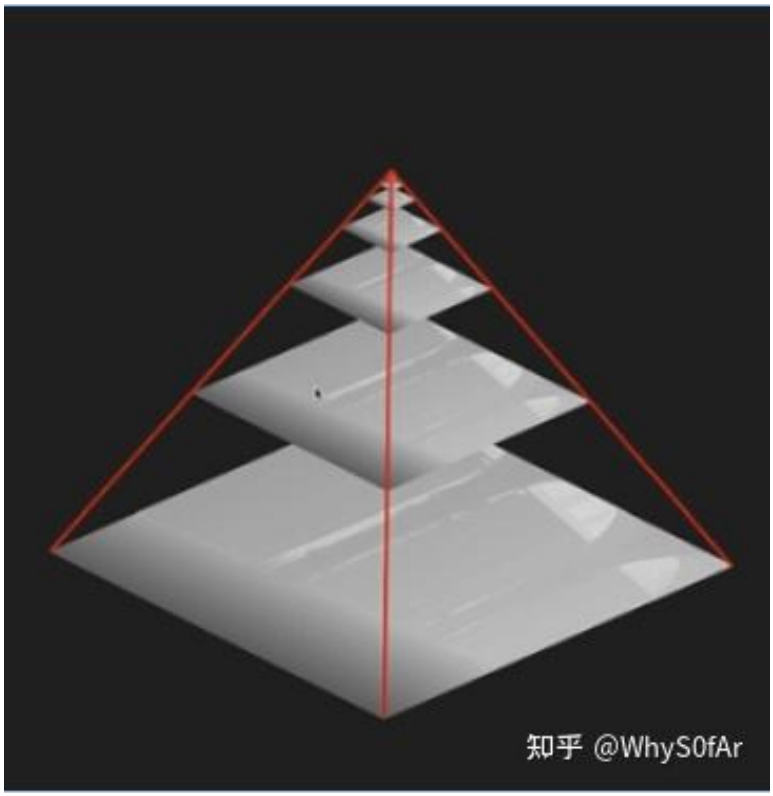
为了能得到这个信息我们需要做一个准备工作,把场景的深度图,做一个mip-map,但这个跟平常的mip-map不一样:



高一级的Mipmap存的并不是周围四个像素的深度平均，而是四个像素中深度的最小值

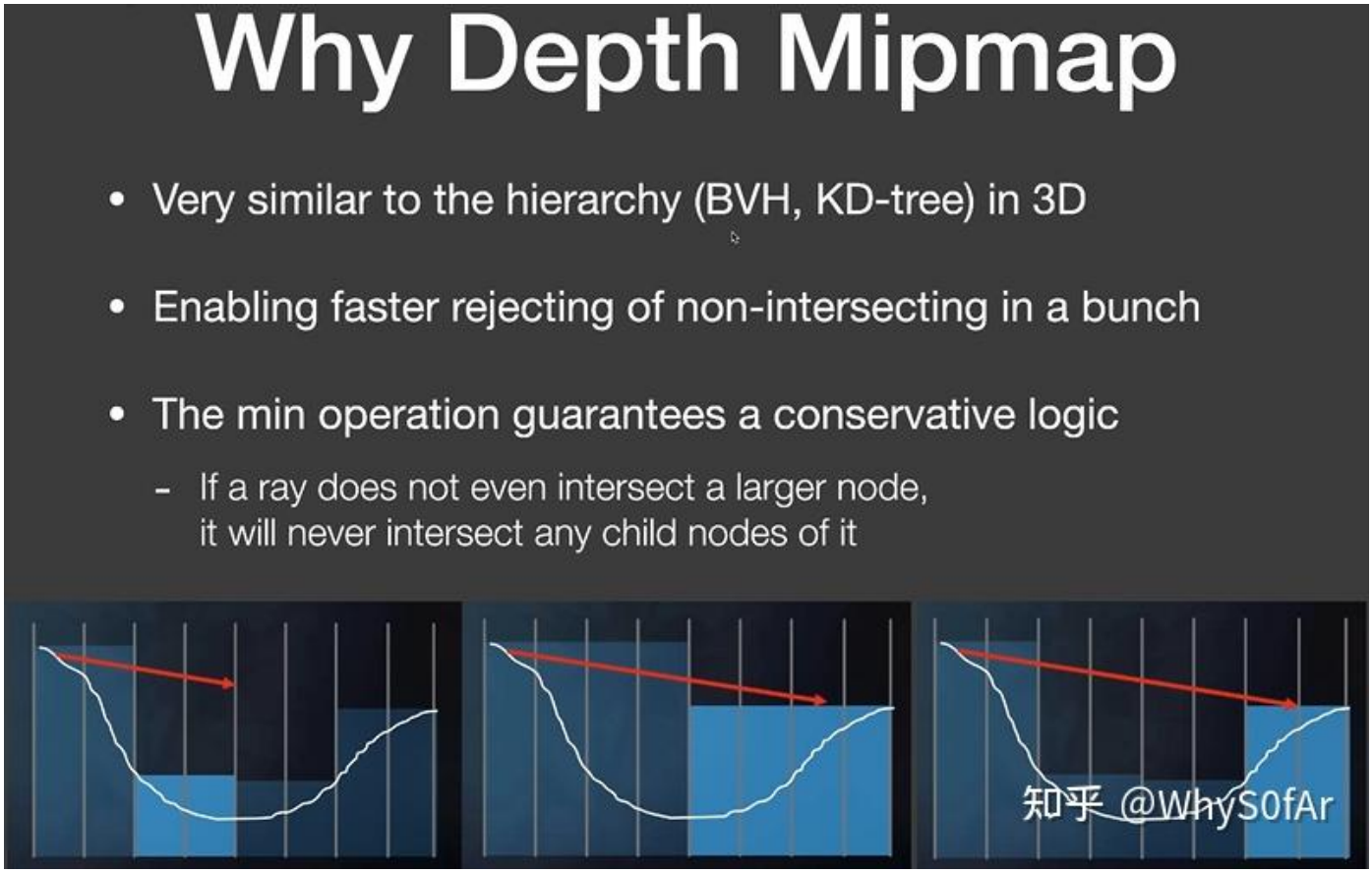


也就是鼠标所指的这一层的一个像素对应的是下图中四个像素中的最小值.



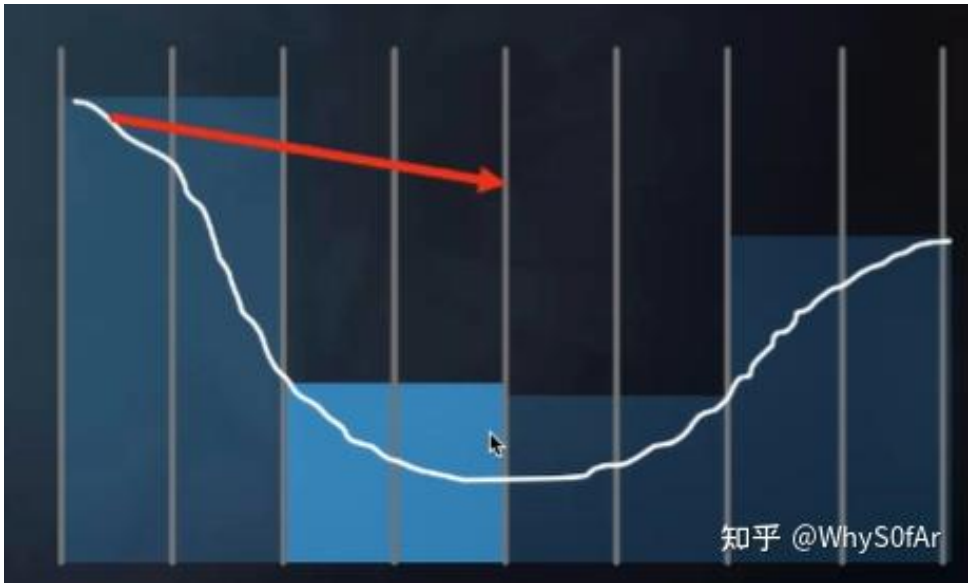
所谓的最小值,指的是四个像素中离深度最小的那个值.

为什么要做深度图mipmap呢?



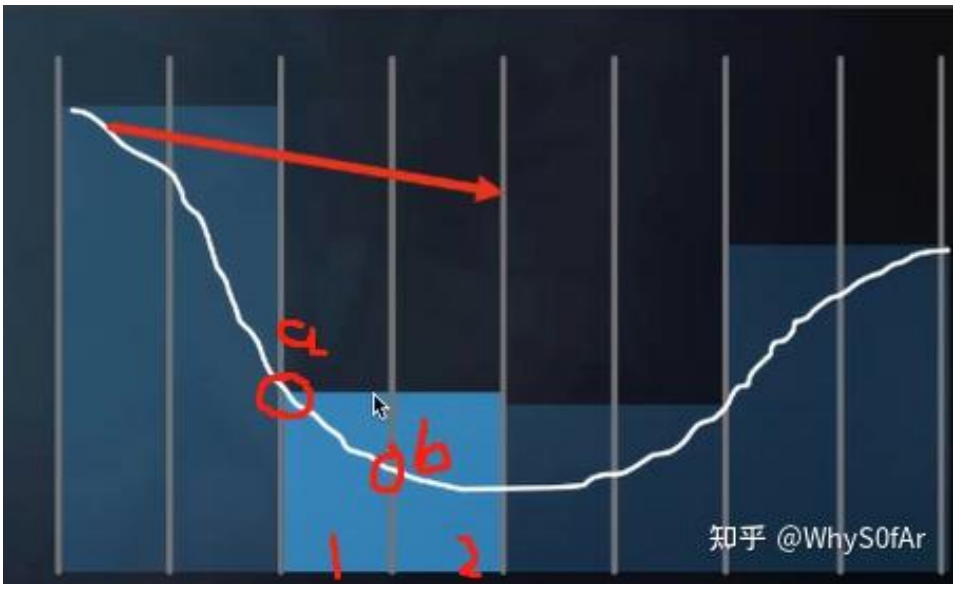
1. 在3D空间做光线追踪时,为了加速光线与场景求交，我们通常会做一个加速的层次结构（BVH/KD-tree）
2. 在屏幕空间同样做一个类似加速结构，可以快速跳过不可能相交的像素；
3. 如果我们用最小值操作的mip-map会得到一个保守逻辑:

- **如果一根光线与mip-map中的上层结点不相交,那他肯定也不会与这个结点的子节点相交.**



我们知道mip-map是一个2维的东西,这里是1维的表示,一个格子表示一个像素,一共有八个格子.

- 每格一判断是level 0
- 每两个是Level 1
- 每四个是level 2.
- 我们这里先以level 1为例子.

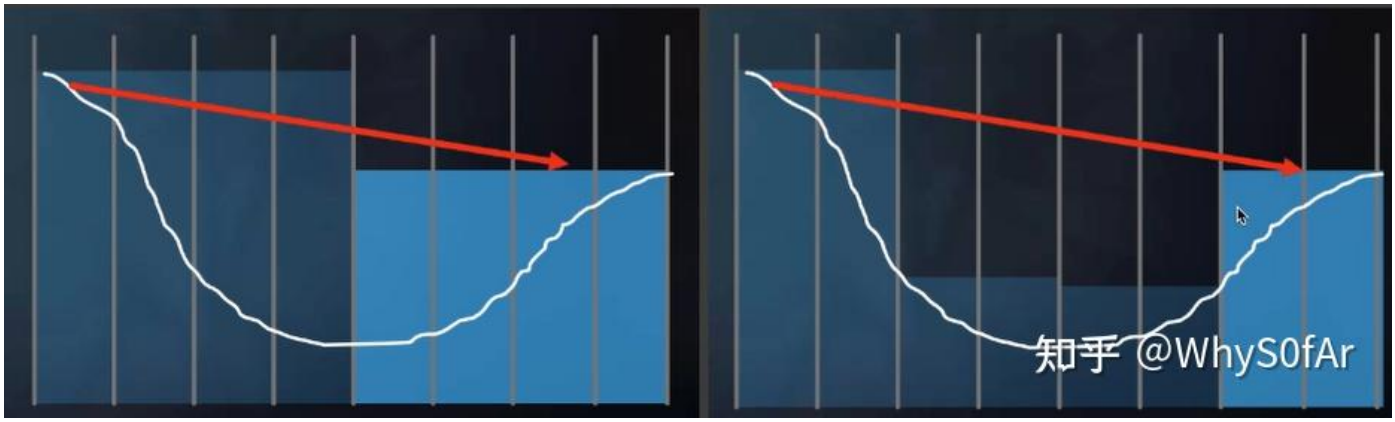


对于像素1,他记录的最浅深度为a,对于像素2它记录的最浅深度为b.

对于这一层来说由于我们要的是最小值而不是平均,因此深度取到了a.

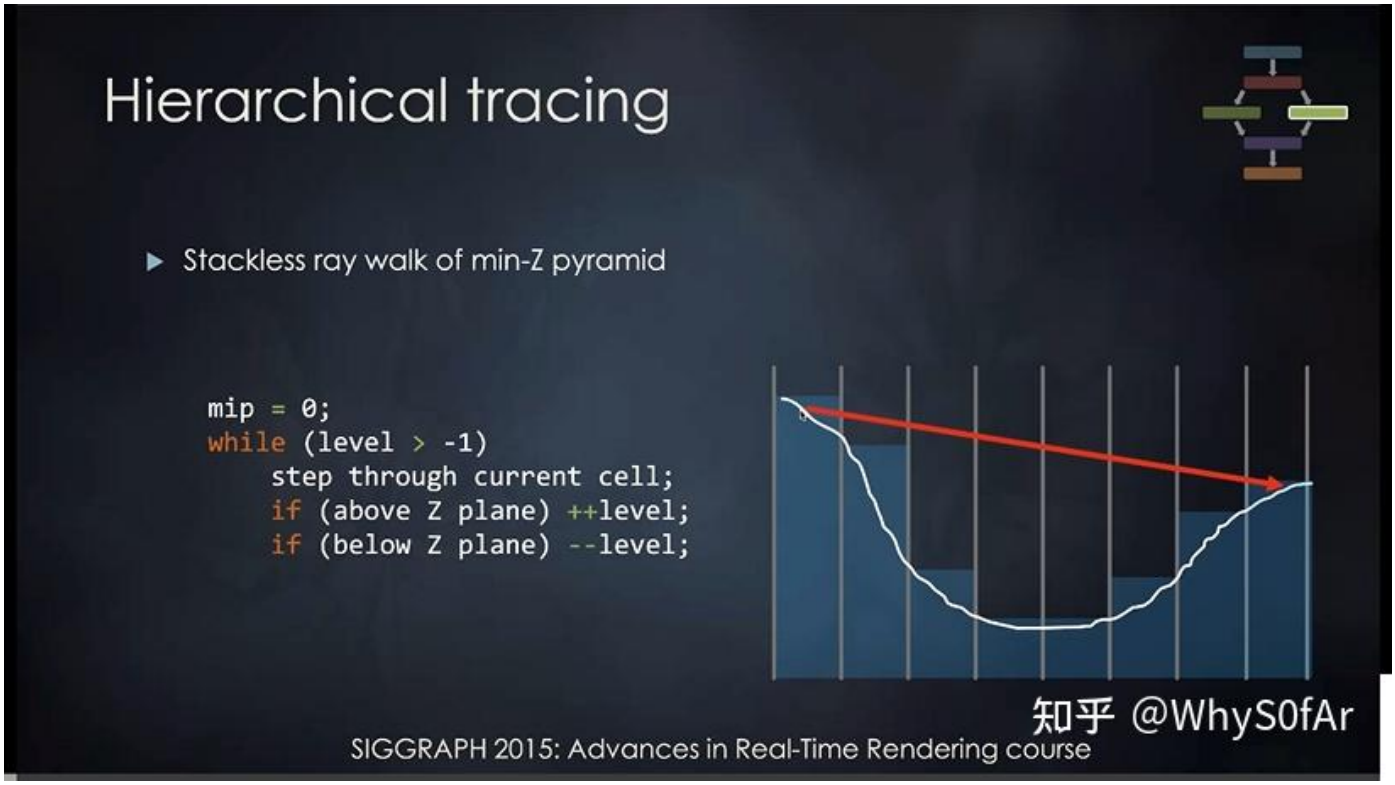
由于这根光线不会与蓝色层所表示的最小深度a相交,因此就不可能与层里的1或2相交了.

这就是他的一个基本逻辑,从而让我们可以快速的跳过很多格子

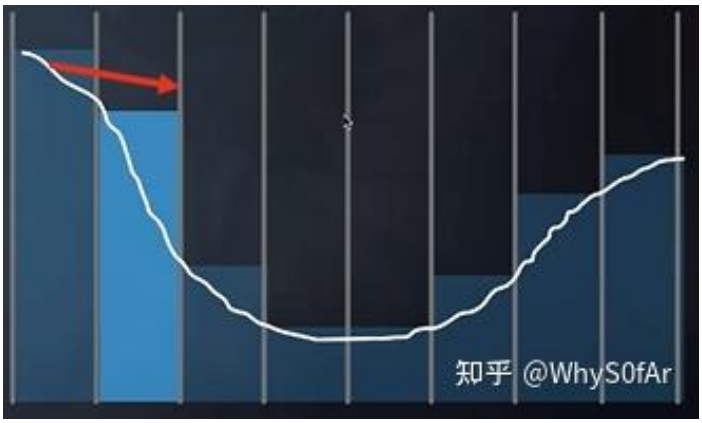


这里是level 2,当光线能够与右边部分相交时,也就是说他能够与其的两个字节点相交,因此我们要去判断与哪个子节点相交,由于左边的最小深度不符合,因此与右边的相交.

也就是我们对场景的壳从深度方面做了一个加速结构,我们来看一下他的伪代码:

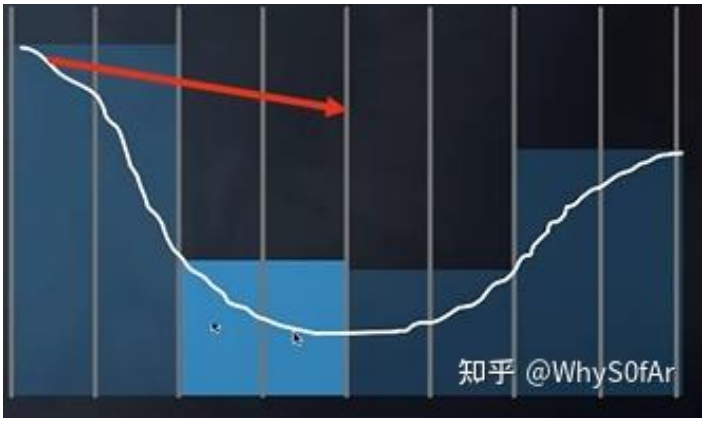


1.先走一个格子(最小的步长)，发现没有交点,那么胆子大一点



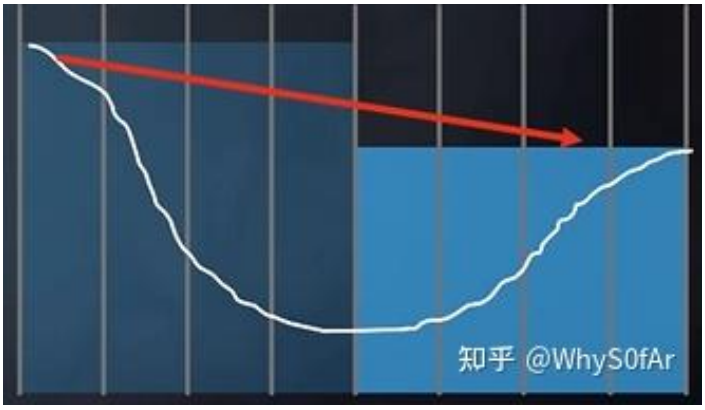
1

2.然后在原来的基础上再走两格(level 1包含的格子)，发现还没交点,胆子再大一点



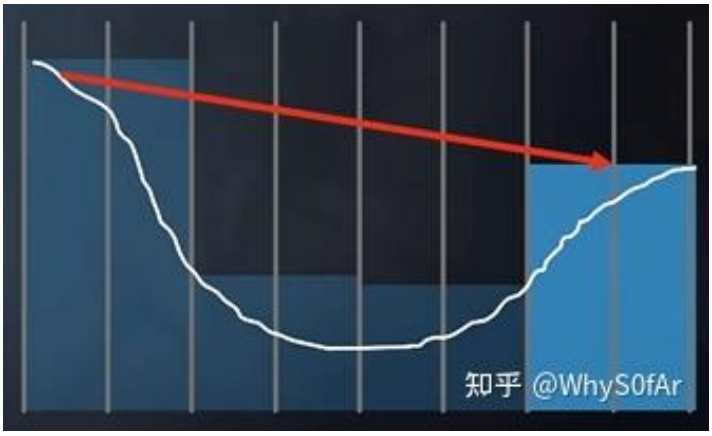
2

3.继续走4格(level 2)，有了交点；



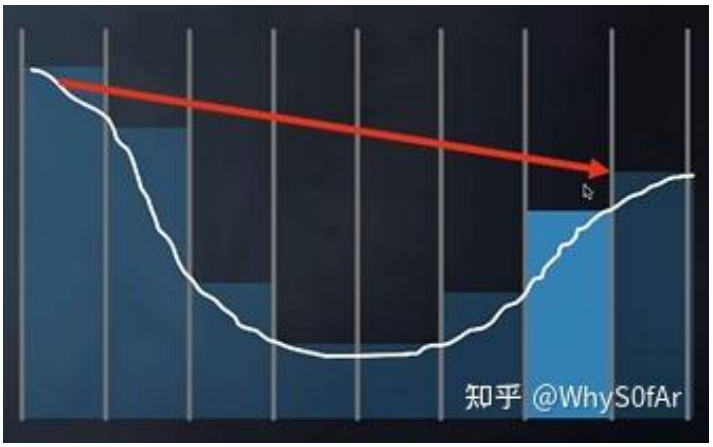
3

4.考虑更精细的层(退回level 1)，发现还是有交点

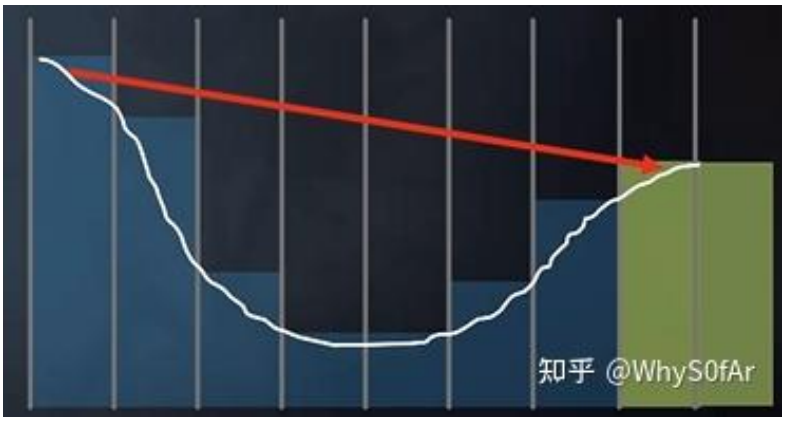


4

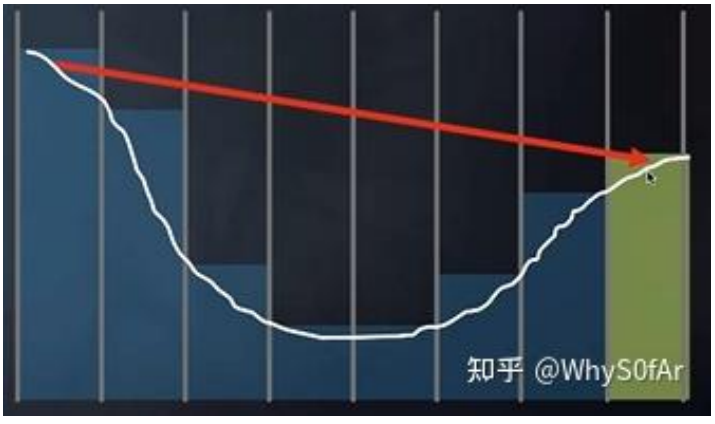
5.由于这里在考虑level 1情况时交点出现在了左半边，因此退回level 0先判断与左边有没有交点，这时候发现没有交点，进入Level 1；



6.再往前走2格发现有交点，并且交点在左半边；



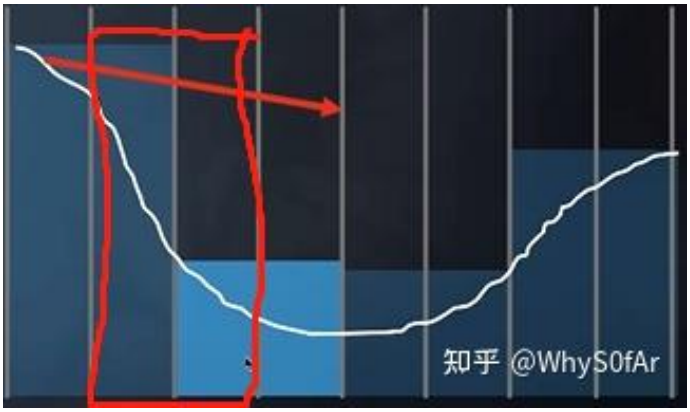
7.这时候就要退回Level 0,向前走一格最终发现了交点，求交结束。

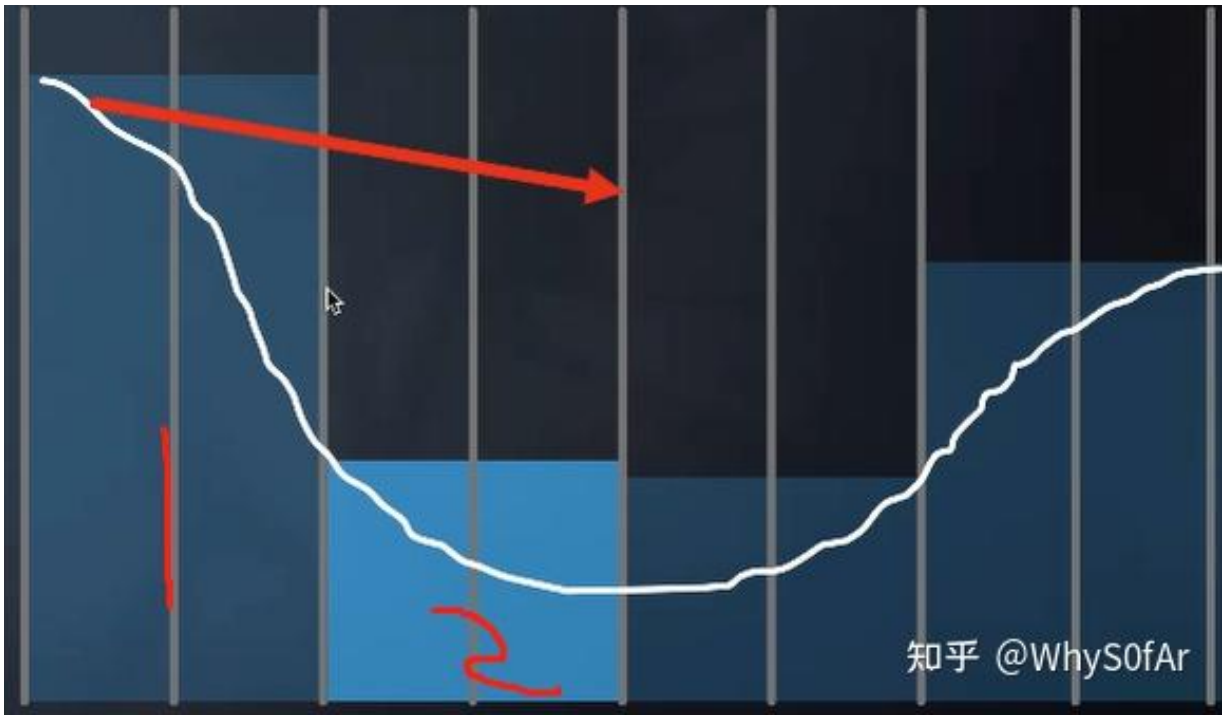


停下来的两个条件:

- 找到了交点
- 一直没找到交点

Mipmap可以做范围查询，可以做正方形查询，但是做不了准确的起点不在2的K次方上的深度的查询。





也就是红色方框区域的最小深度,我们硬要求的话,我们需要算出1和2的最小深度,然后进行插值才能得到它的最小深度.

我们来回顾一下:

由于我们不知道以一个多长的固定的步长来逐步前进求交点,因此我们用高一级的Mipmap存的并不是周围四个像素的深度平均，而是四个像素中深度的最小值来动态的决定步长,从而可以快速的求到任何光线与场景壳的交点.

但是SSR仍然有屏幕遮挡的问题:

不在屏幕中这层“壳”的信息是不会被反射的，丢失正常的反射信息。

我们可以看到手指那边得到了一个可笑的结果,这就是SSR的问题,他只会反射出camera所看到的,而不是3D场景中真实存在的所有,这也是我们学到的screen space的三个算法的一个通病,整体看来只有SSDO要稍微好点.

还有一个就是屏幕边缘信息的缺失问题:

我们可以知道地板上的红色部分是有反射的,但是黄色部分由于是在屏幕外,因为得不到反射,但是实际上他应该是有反射的.

这种问题可以根据反射光走的距离做一个衰减 减少违和感。

至此我们完成了屏幕空间光线追踪的部分,但是我们还没完成如何计算shading.

这部分与路径追踪的方法完全相同，仅仅是把光线与场景求交变成了光线与“壳”求交，因此路径追踪的算法在这里是可以直接使用的。

对于任何一个shading point，看到的radicance就是对半球进行积分,如果是specular的物体,那么相当于光线打到物体的哪里,就用它所发出的radiance就可以.

$$L_o(\mathrm{p},\omega_o)=\int_{\Omega^+}\frac{L_i(\mathrm{p},\omega_i)f_r(\mathrm{p},\omega_i,\omega_o)\cos\theta_i\,\mathrm{d}\omega_i}{L_o(\mathrm{q},q\to p)}$$

如果是glossy情况下,同样的用蒙特卡洛多采样几根光线,不管怎么所打到的物体反射过来的radiance,一定就是shading point点接收到的incident radiance.

这里我们同样需要假设反射物/次级光源 是Diffuse的情况,地板之类的接收物可以是任何物体.

问题：

1.Flux Intensity是否存在平方反比衰减的问题?

-不存在，这里做的是BRDF sampling，并不是某个指定次级光源到shading point，因此并不存在平方反比衰减的情况。

2.是否能够处理好次级光源与shading point的可见性的问题?
-由于是路径追踪Tracing计算出来的，看到的一定是能够看见的，因此是没有这个问题的。

屏幕空间反射中的自然特殊现象:
由于结果是tracing出来的，因此很多现象是自然就能做到的，并不用做trick，类似很多模糊等现象可以直接实现。
① Sharp and blurry reflection：Glossy物体反射的模糊现象。

② Contact hardening：对于物体来说，近处的物体反射清晰，远处反射模糊，当反射物是Glossy时反射的lobe是一个锥形，当距离越远锥形的截面越越大，也就会发生模糊现象。

我们可以看到爪子处的反射很清晰,而翅膀处的就比较模糊了

③ Specular elongation：反射在垂直方向被拉长的现象，在雨天常见的现象，当我们认为地面是各向同性的，也就是法线分布是一个圆，反射出去的lobe就是一个椭圆。

④ Per-piex roughness and normal：对于不同的法线与粗糙度不同的现象。

SSR总结
优点：

- 可以很快的做glossy与specular的反射
- 质量很好
- 没有spikes尖点与遮蔽的问题

缺点：

- 在Diffuse 物体上并不是非常高效

- 丢失屏幕外的信息,但这个是屏幕空间的局限性,而不是光线追踪的问题