

实时渲染基础魔法

Realtime Magic: Fundamentals

Tianyu Huang
tianyu@illumiart.net

插图代码、示范场景等补充材料：<https://github.com/CodingEric/realtime-magic-supplement>

什么是实时渲染?

Prey (2017) / Arkane Studios

在电子游戏《掠食（2017）》中，Talos I 空间站的硬件工程师 Calvino 博士发明了一个称为“通讯镜”的机器。透过这面“镜子”，观察者可以看到一个全息的虚拟世界。

“通讯镜”就是实时渲染技术的一个应用。

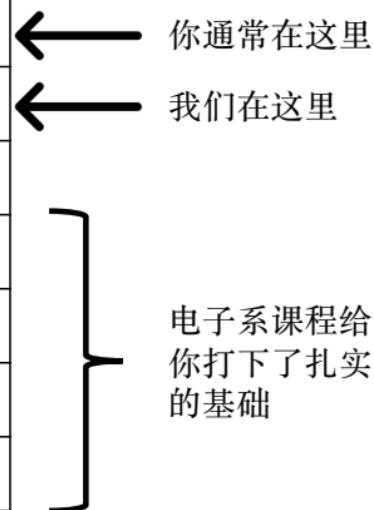
什么是实时渲染?

传统的实时渲染，或者说光栅渲染，是独立地确定将要绘制在屏幕上的每个三角形内部点的颜色的过程。

随着计算机性能的提升，现代实时渲染技术已经不再局限于此。例如光线追踪和光线步进技术也被应用于实时渲染中。

* ShaderToy 是一个充满光线追踪和光线步进等实时渲染邪教的地方：<https://www.shadertoy.com>

实时渲染：我们工作在哪个层次？



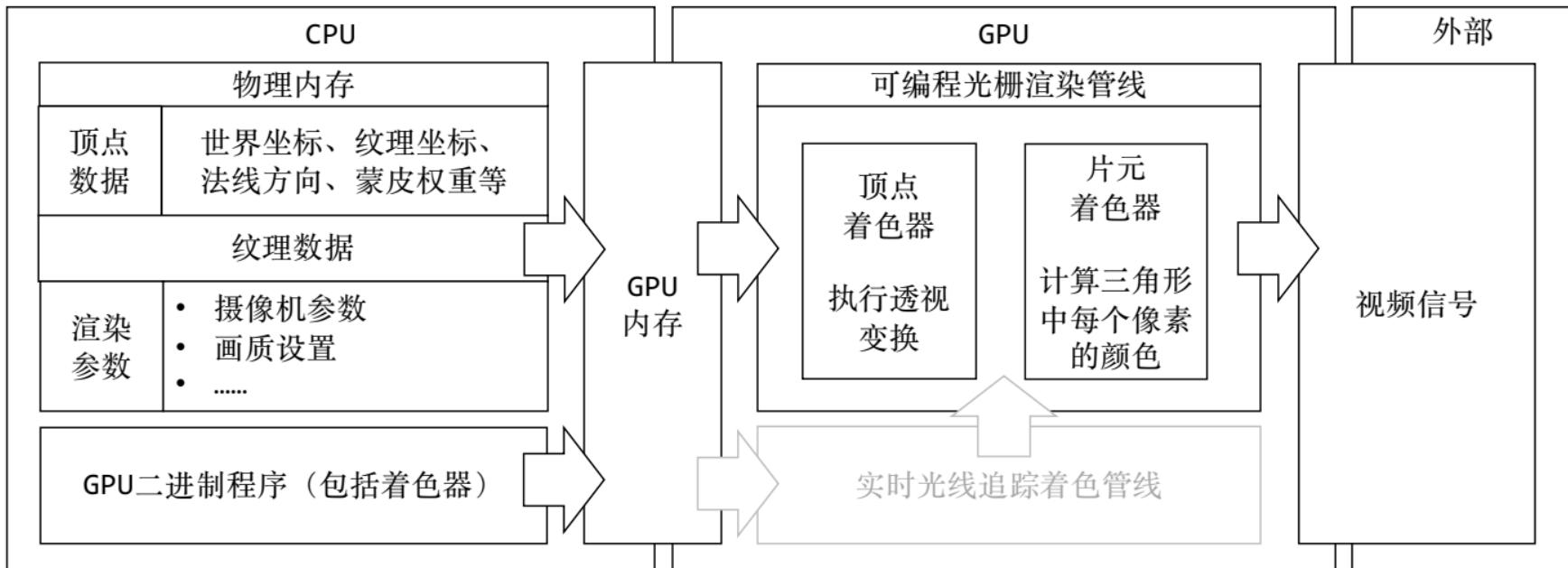
尽管在实际的游戏开发中，并不需要你去手动实现本讲座中介绍的大部分技术；但是通过本讲座，你可以知道这些游戏引擎内部预置的技术应该如何善用，以美化你的画面品质。

实时渲染的工程描述

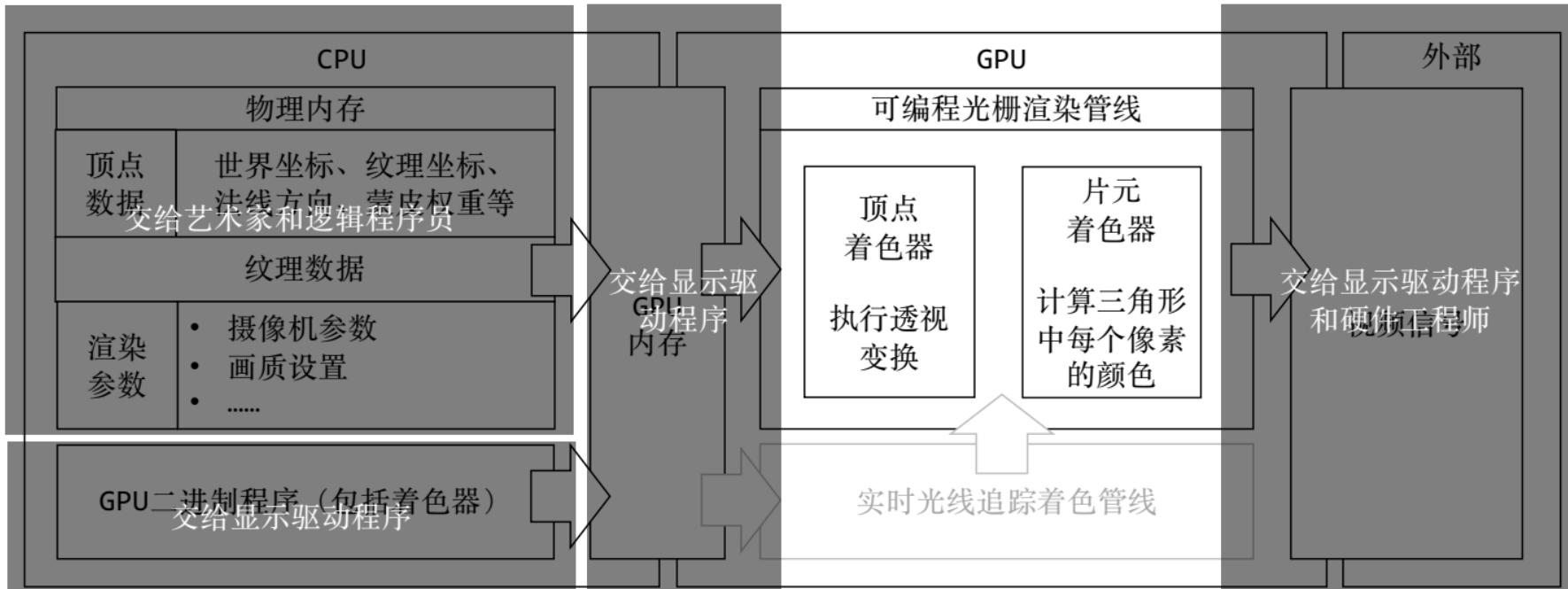
The Unfinished Swan (2012) / Santa Monica Studio and Giant Sparrow

这里的“工程描述”并不是说把实时渲染完完整整地描述下来——这需要好几本书！
相反，我们只需要给出一个简要的描述，刻画实时渲染是如何在现有的计算机软硬件系统上运行的。

实时渲染的工程描述



本讲座只关心可编程光栅渲染管线里发生的事情.....



本讲座覆盖的内容

概述

- 你正在阅读

可编程光栅着色管线

- 可编程光栅着色管线
- GLSL着色语言

物理渲染

- 颜色
- 照明
- 表面
- 体积
- 光栅阴影

非真实感渲染

实时光线追踪

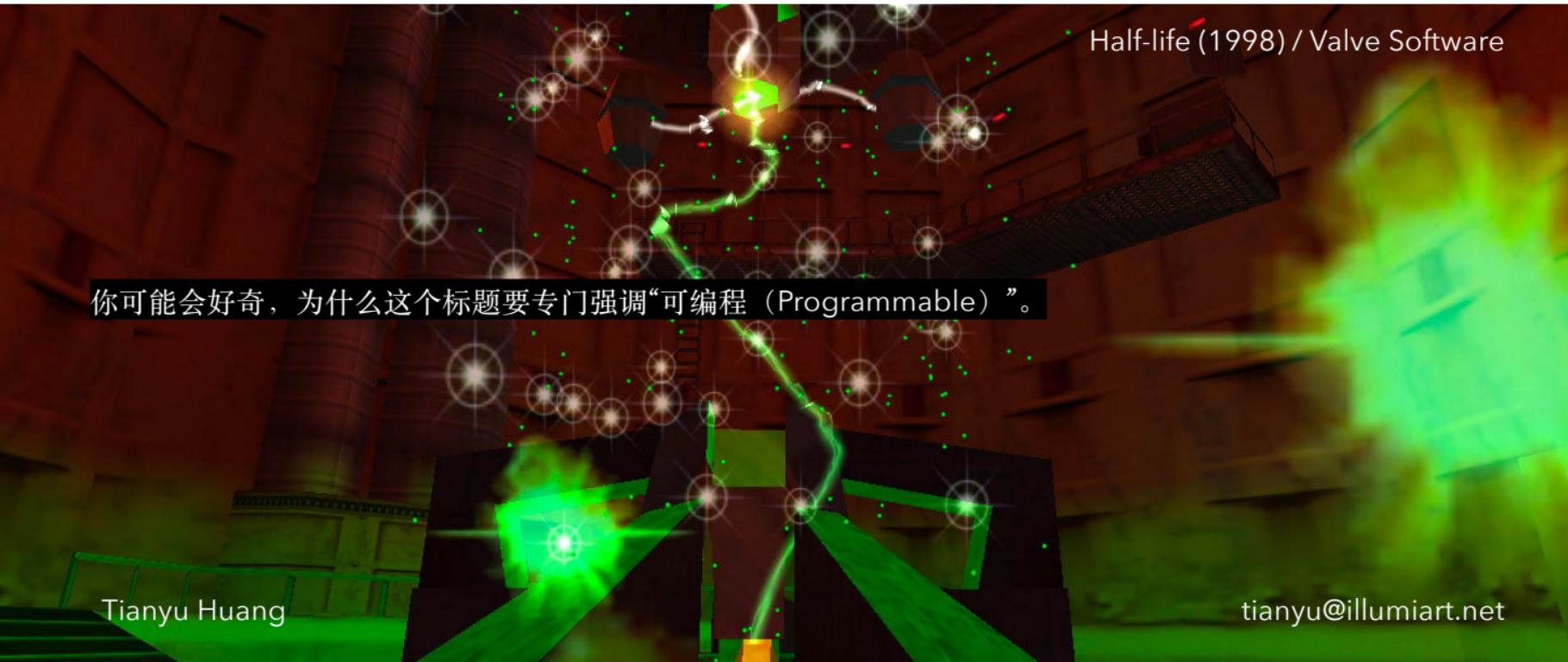
- 光线追踪
- 光线追踪加速

幻术

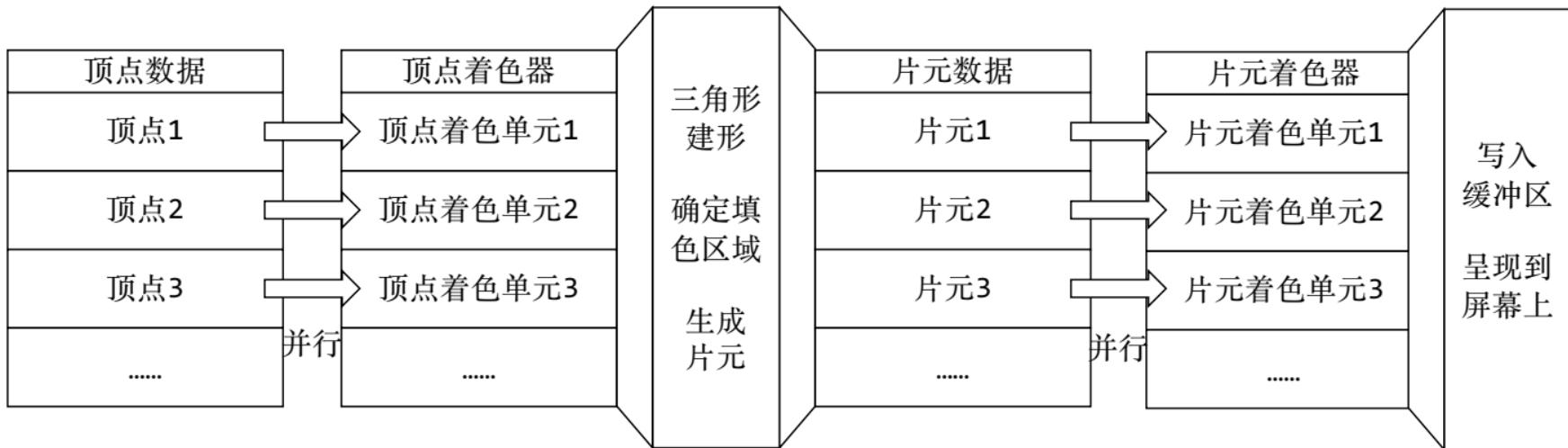
- 基于图像的渲染
- 环境光遮蔽
- 屏幕空间反射
- Bloom
- 常见实时渲染优化策略

结束语

可编程光栅着色管线

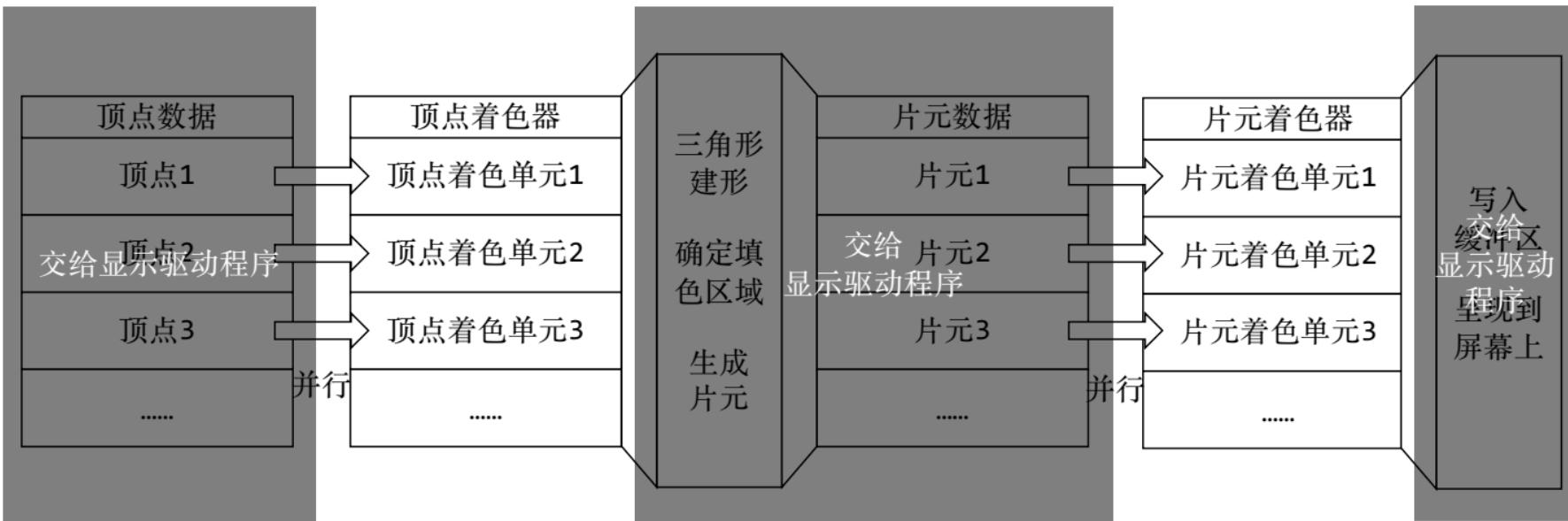


可编程光栅着色管线的最简形式



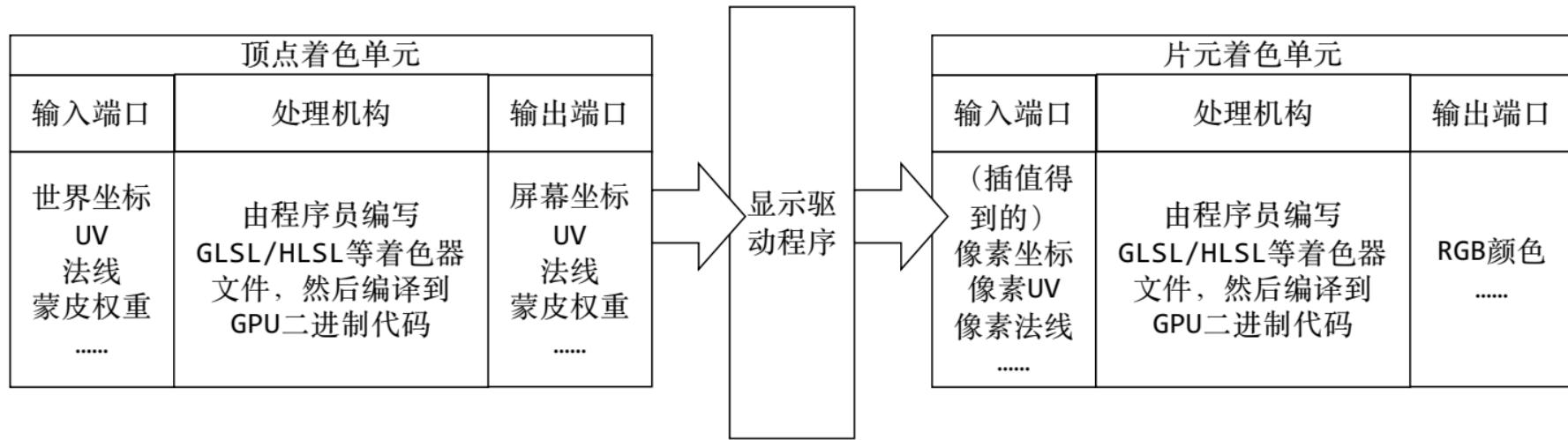
* 实际情况比这远远复杂。例如，GPU的着色单元数量往往小于顶点数量或屏幕总像素点数量。这要求进行数据切分和调度。

可编程光栅着色管线的最简形式



* 实际情况比这远远复杂。例如，GPU的着色单元数量往往小于顶点数量或屏幕总像素点数量。这要求进行数据切分和调度。

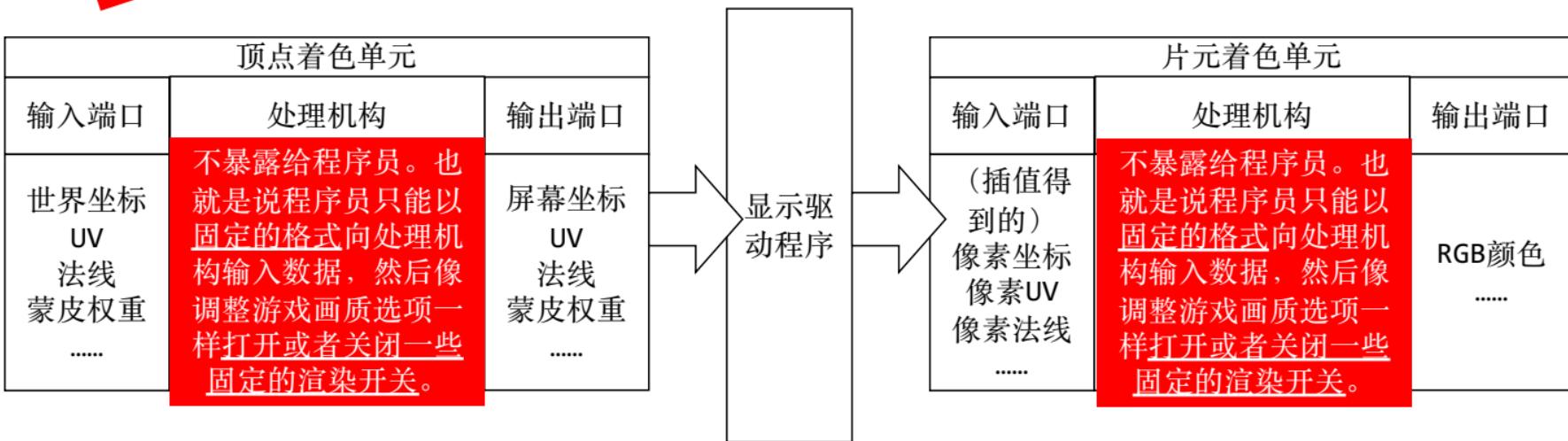
可编程光栅着色管线的最简形式



* 显示驱动程序将三个顶点合为一组，构建成三角形。然后在填充这些三角形时，交给片元着色器渲染每个待填充像素的颜色。

* 实际情况比这更加复杂。

固定光栅着色管线的最简形式



* 在固定渲染管线中，画面绘制的美观度上限往往取决于显示驱动程序或者图形API。（因为程序员只能打开或者关闭画面选项，而不具备对画面渲染的编程级别控制权）

GL着色语言 (GL Shading Language, GLSL)



GL着色语言 (GL Shading Language, GLSL)

GLSL是一种用于编程渲染管线的语言（并不仅限于光栅着色管线），可以运行在OpenGL或Vulkan上。

在大多数情况下，GLSL都在运行时由显示驱动程序编译，然后以GPU二进制代码的形式发送到GPU内存中。

GLSL和C非常相似，但是你需要注意几个问题：

- 早期的GLSL拥有非常严格的类型检查，不支持隐式转换。
- GLSL不支持递归。
- GLSL提供了vec2、vec3和vec4表示向量；提供了mat2、mat3和mat4表示矩阵。
- GLSL提供了sampler2D对内存中的材质数据块进行采样。
- GLSL的向量支持Swizzle功能。
- GLSL有若干与渲染管线紧密关联的类型限定符。
- GLSL一般情况下是无状态（stateless）的。
- 从CPU内存映射到GPU内存的结构体变量需要遵守std140内存布局规则。

GLSL向量的Swizzle功能

在GLSL中，向量使用vec2、vec3和vec4表示。

Swizzle功能可以快速使用某个向量的分量重组成新的向量。

```
vec2 someVec;  
vec4 otherVec = someVec.xyxx; // 等价于 otherVec = vec4(someVec.x, someVec.y,  
someVec.x, someVec.x)  
vec3 thirdVec = otherVec.zyy; // 同上。
```

GLSL的类型限定符 (Type Qualifier)

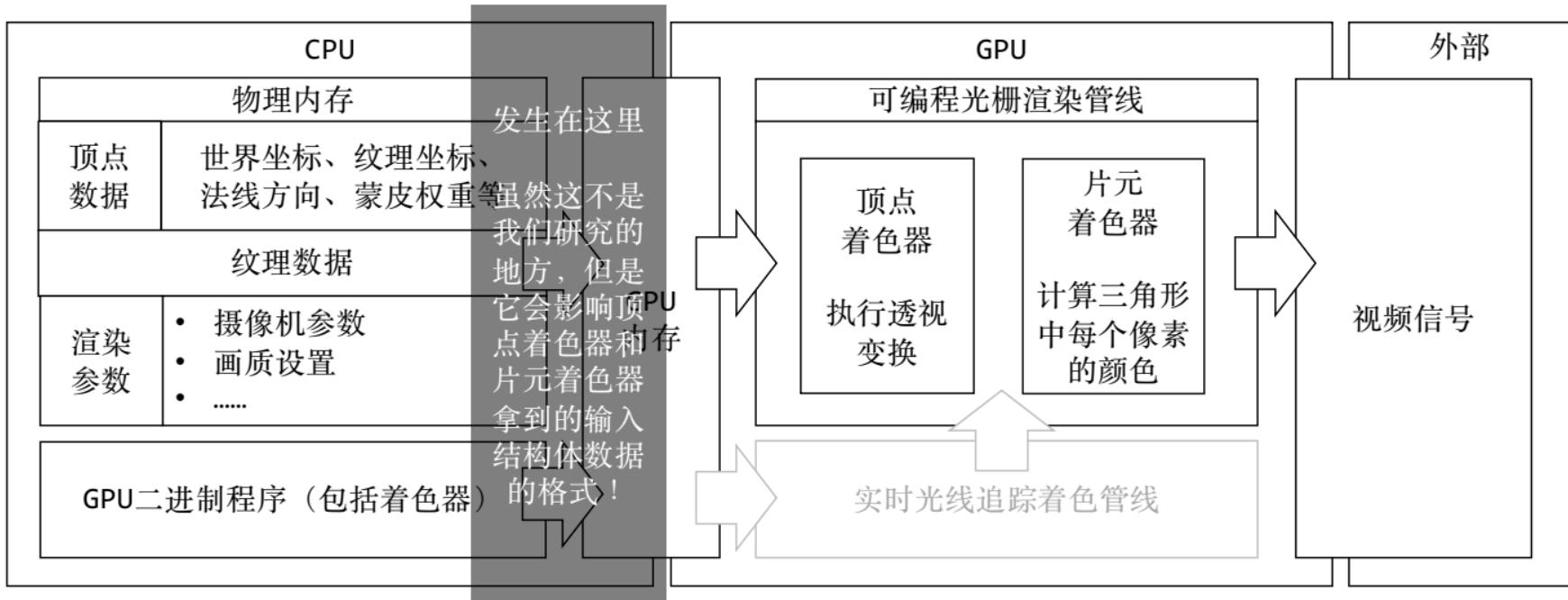
作为管线/流水线的单元，着色器需要明确指定哪些变量是输入变量，哪些变量是输出变量，哪些变量是在不同着色单元之间都不变的变量。与之相关的限定符称为存储限定符 (Storage Qualifier)。

- GLSL使用in和out标记着色器单元的输入变量和输出变量。
- 在各个着色单元之间不变的变量称为uniform变量。
- 如果着色器不处理某个输入变量，而是将其按原样输出，你需要将其标记为invariant。

为了提高速度，一些数据可以以更低的精度参与计算。与之相关的限定符称为精度限定符 (Precision Qualifier)。有三种精度限定符：highp，mediump和lowp，它们可以作为前缀加在任何数值类型变量的左侧。

从显示内存中读取数据和写入数据到显示内存中时，内存的布局很重要。与之相关的限定符称为布局限定符 (Layout Qualifier)。

std140 内存布局规则



std140 内存布局规则

由于顶点数据传入以后，往往使用layout限定符来指定内存中的位置。所以std140内存布局规则的主要作用对象并非顶点数据，而是uniform数据。

顶点着色器
顶点着色单元1
顶点着色单元2
顶点着色单元3
.....

在GPU上并行运行的不同顶点着色器接收到的顶点数据是不同的；但是它们共享着相同的uniform数据。

uniform数据往往为：摄像机信息、场景中的光源信息和渲染选项等。从工程的角度来考虑，我们很自然地知道应该让所有着色单元共享这些数据。

std140发生在以struct的形式传入uniform数据时。

std140 内存布局规则

所有的数据都按照16个字节（一个vec4的大小）对齐到显示内存中。

float			
vec3			
vec4			
mat3			
int			
bool			

我们不难注意到，vec3和mat3在这一内存布局中显示出不太好的结果。

事实上，在GLSL的文档中也明确提出，最好不要使用vec3和mat3，而使用vec4和mat4。

GLSL: 示例

顶点着色器

```
#version 410 core
in vec3 iPos;
in vec2 iUV;
out vec2 UV;
uniform mat4 projection;
uniform mat4 view;
uniform mat4 model;

void main(){
    UV = iUV;
    gl_Position = projection * view *
model * vec4(vPos, 1.0);
}
```

片元着色器

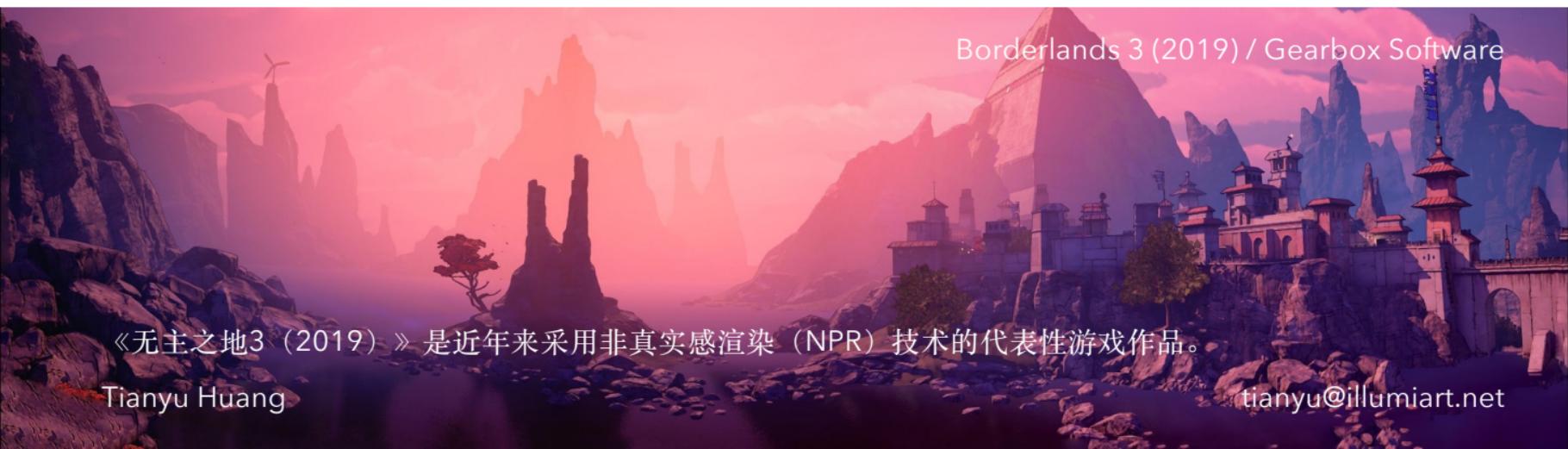
```
#version 410 core
in vec2 UV;
uniform sampler2D uTexture;

void main(){
    gl_FragColor =
texture2D(uTexture, UV);
}
```

可编程光栅着色管线：结束语

现在，你已经知道可编程光栅着色管线的基本原理，并且也大概了解如何使用GLSL对着色管线进行编程，从而赋予屏幕上的像素点颜色了。

然而这时的你，只是刚学会如何握住魔杖的学徒，才刚刚迈出了通往实时渲染魔法的第一步。如何绘制出具备真实感或者艺术感的图形，是我们下一步要讨论的话题。



《无主之地3（2019）》是近年来采用非真实感渲染（NPR）技术的代表性游戏作品。

物理渲染 (Physically-based Rendering, PBR)

Spring / Blender Studio

怎样才能让一张计算机生成的图像看起来是实拍的，或者说“具备说服力”？

我们首先要研究这个世界是“如何渲染”的。

什么是物理渲染 (Physically-based Rendering, PBR) ?

广义的物理渲染就是基于物理学的渲染，主要讨论：

- 电磁辐射功率是如何在表面与表面之间、表面与介质微粒之间和介质微粒与介质微粒之间传播的。
- 表面或者介质微粒是如何对输入的功率进行响应的，在不同入射角和出射角下会产生什么现象。
- 如何模拟高级物理现象，例如光谱色散、焦散、等离子体积、光的波动现象甚至相对论效应等。

在本讲义中（和大部分PBR资料中），物理渲染是指一种在计算机中模拟电磁辐射功率与不同表面和介质微粒的相互作用的渲染手段，也就是上面讨论的前两点。

物理渲染：我们工作在哪个层次？

几何光学
波动光学
量子物理

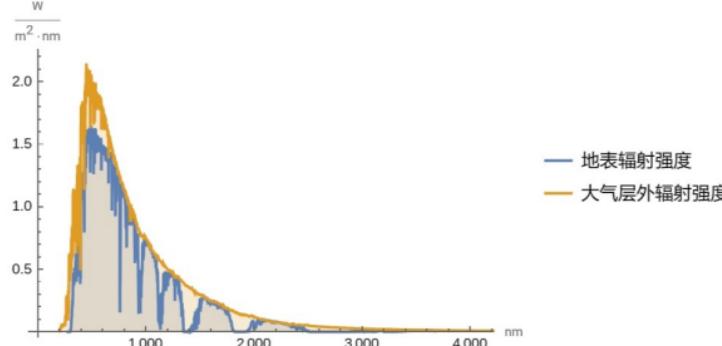
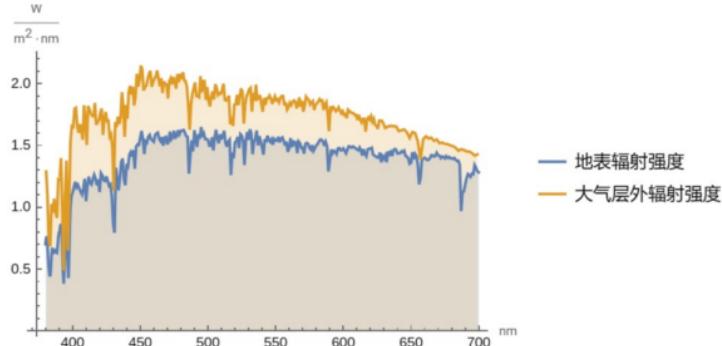
你可能会尝试在左侧的三个层次中寻找物理渲染的影子，但事实上物理渲染并不严格属于其中任何一个层级。

物理渲染研究电磁辐射功率与不同表面和介质微粒的相互作用。
但我们并不关心能量的载体是什么：波还是粒子，我们无所谓。我们需要且只需要遵守能量守恒。

颜色



太阳辐射光谱



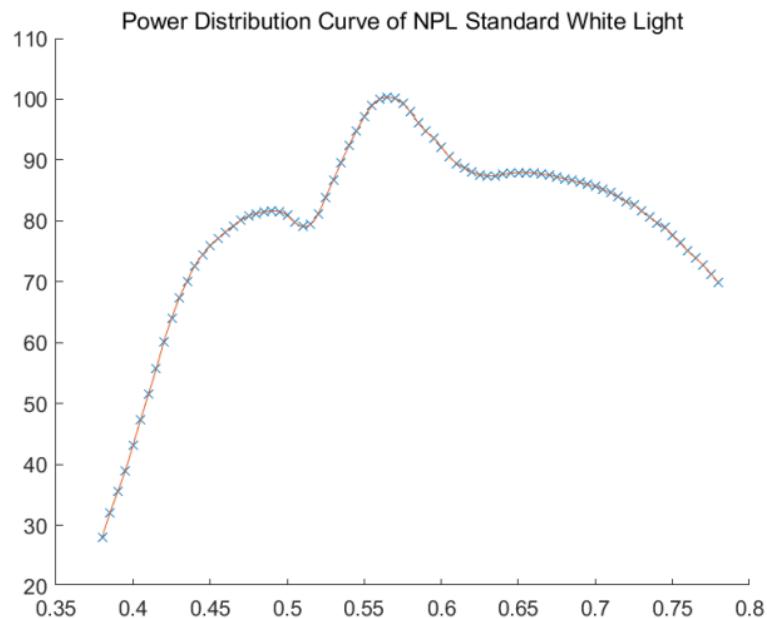
对人类来说，太阳发射光谱在可见光区域的颜色就是白色。

在漫长的生物进化史中，我们的视觉系统认为在可见光区域中分布得比较均匀的光谱是白色，例如阳光。

我们发现，白光/可见光恰好分布在5778K黑体辐射线能量最高且最平稳的区域。

这不禁使我们产生一些联想：假如在一个恒星温度比较低的红矮星星系里产生了智慧生命，又或者在一个恒星温度非常高的蓝巨星星星系里诞生了智慧生命，他们的“白色”是什么样的呢？

标准白色



左图是NPL标准白光的功率-波长分布示意图。

白色的确定非常的关键，它就像纷繁复杂的色彩空间中的一个锚定点。使用白色作为颜色标准的制定基础要远远好过使用单色光，因为白色本身就是各种单色光的总和！

归一化颜色

在接下来的讨论中，我们暂时先讨论归一化颜色。

36,228,228

18,114,114

9,57,57

上面三种颜色的归一化值都是：

$$R = \frac{r}{r + g + b} = 0.0732$$

$$G = \frac{g}{r + g + b} = 0.463$$

$$B = \frac{b}{r + g + b} = 0.463$$

通过归一化操作，我们忽略了亮度信息，而专注于色相。

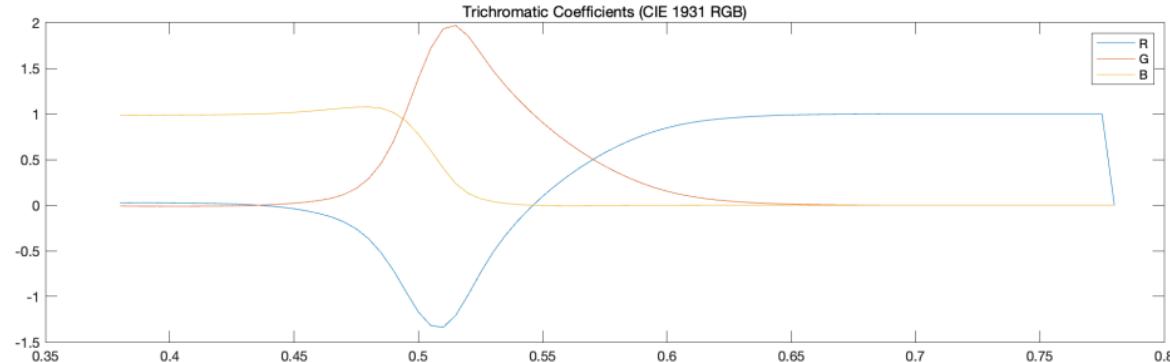
标准颜色测定过程

标准颜色测定过程实验分为以下两步：

- 从光谱中抽取一个颜色；
- 人为调整三个单色光源，使得三个单色光源合成出这种颜色。

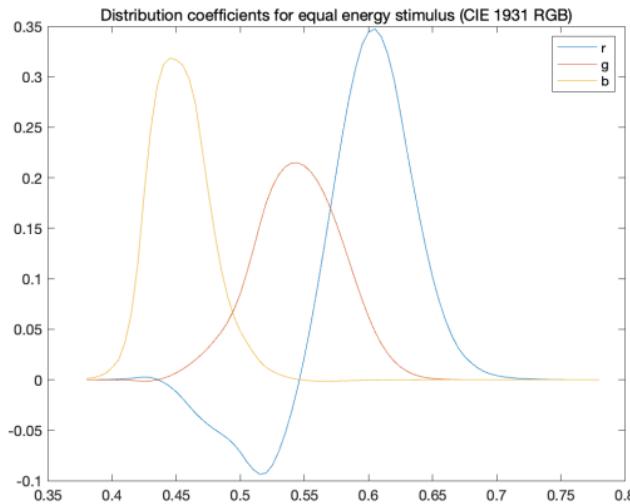
使用大量的志愿者重复上述实验，取平均值。

由此，我们建立起光谱到三个单色光源的RGB仪器值的映射。



数据源：The C.I.E.
colorimetric standards and
their use, T Smith and J
Guild 1931 Trans. Opt. Soc.
33 73

标准颜色测定过程



接下来我们对三个光源的能量进行归一化。
这张图片就是知名的CIE 1931 RGB曲线了。

如何计算rgb分量？

任何一个光谱都可以分解为大量单色光的总和。

$$R(\lambda) = R(\lambda) * \delta(\lambda) = \int R(\lambda_0) \delta(\lambda - \lambda_0) d\lambda_0$$

我们定义函数

$$rgb(R(\lambda)) : \mathbb{R} \rightarrow \mathbb{R}^3$$

那么

$$\begin{aligned} rgb(R(\lambda)) &= \int rgb(R(\lambda_0) \delta(\lambda - \lambda_0)) d\lambda_0 = \int R(\lambda_0) rgb(\delta(\lambda - \lambda_0)) d\lambda_0 = \int R(\lambda_0) \begin{pmatrix} r(\lambda_0) \\ g(\lambda_0) \\ b(\lambda_0) \end{pmatrix} d\lambda_0 \\ &= \int R(\lambda) \begin{pmatrix} r(\lambda) \\ g(\lambda) \\ b(\lambda) \end{pmatrix} d\lambda \end{aligned}$$

其实就是亮度曲线和三条rgb曲线的内积。

反照率 $\rho(\lambda)$

已经知道了如何把连续光谱表示成三个刺激值，现在我们可以研究世界上的一切物体为什么呈现出它们的颜色了。

反照率是表面对光谱的响应。

我们知道，树叶呈现出绿色是因为树叶反射了绿色的光。更本质的描述是，树叶的反照率在绿色附近最大。

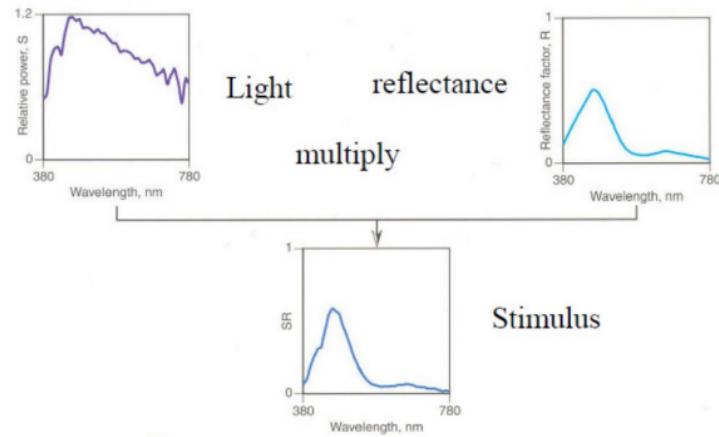


Image Credit: MIT 6.837 Computer Graphics (Fall 2012)

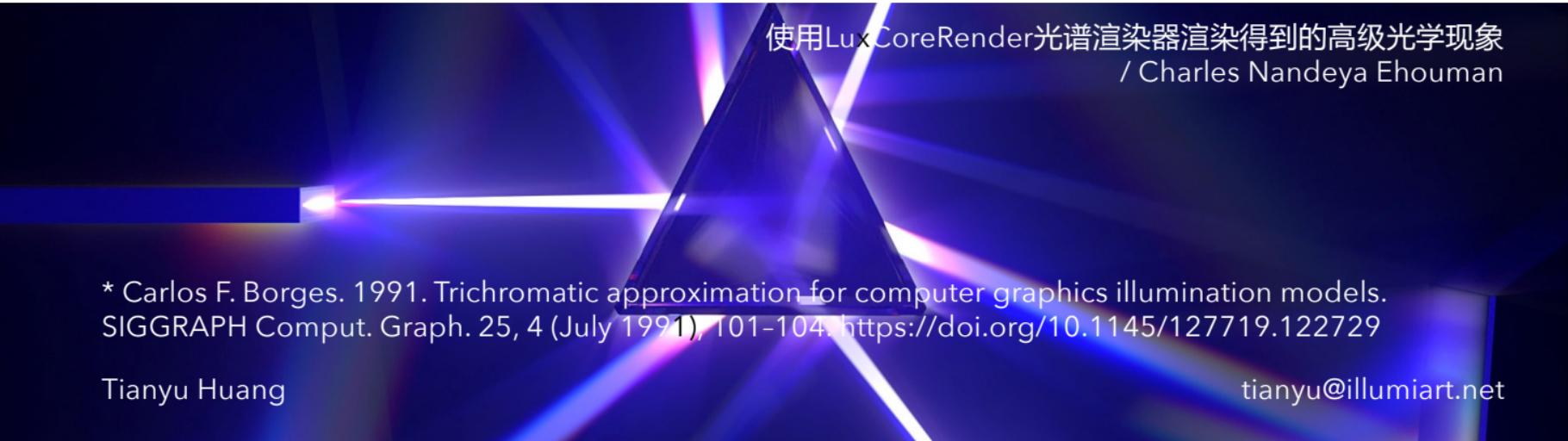
tianyu@illumiart.net

反照率 ρ 的三刺激值近似

存储光源的发射光谱和反射光谱的内存开销是巨大的，如果我们只存储三刺激值，然后对三刺激值进行乘法，在数学上是否可行？

首先它们肯定不相等，但是已经有人分析过*，这一误差在容许范围内。

现代的所有实时光栅渲染系统以及大部分离线渲染框架都采用三刺激值近似。

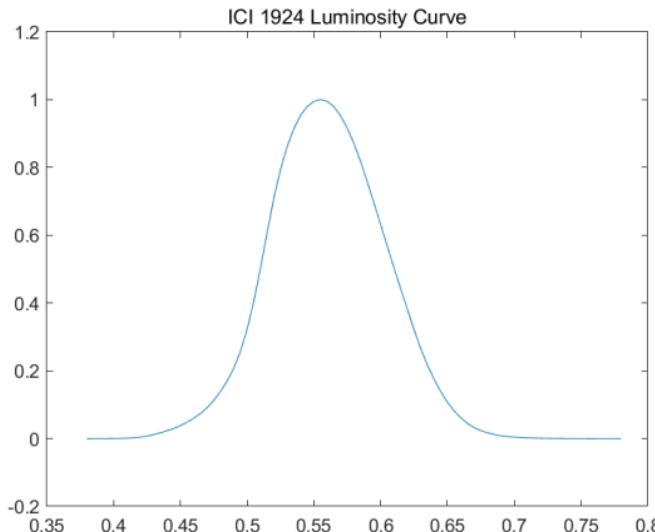


使用LuxCoreRender光谱渲染器渲染得到的高级光学现象
/ Charles Nandeya Ehouman

* Carlos F. Borges. 1991. Trichromatic approximation for computer graphics illumination models. SIGGRAPH Comput. Graph. 25, 4 (July 1991), 101–104. <https://doi.org/10.1145/127719.122729>

照明效率曲线 $Y(\lambda)$

人眼对相同能量的不同波长的亮度感知是不一样的。举一个非常简单的例子：100W的绿色灯泡和100W的红外灯泡，显然红外灯泡是无法被人眼看见的。就算在可见光区域，人眼对不同波长的颜色的响应也是不同的，这一特性可以使用照明效率曲线 $Y(\lambda)$ 反映。



这个曲线在555nm处达到峰值。说明在输出照明功率相同的情况下，555nm的单色光能产生最大的视亮度（Luminous）。
这是大部分应急标识都使用绿色灯光的原因之一。

刚才提到亮度，亮度的物理量符号为 L ，与辐射度 R 区别。

将辐射度转换到亮度的公式是：

$$R(\lambda) = Y(\lambda)L(\lambda)$$

按量纲分析，亮度的单位应该和辐射度是一致的，不过为了便于区分，我们引入流明（lm）。进而规定，功率为1W的，555nm的单色光，产生的亮度为683 lm。

使用 R 或者 L 无关紧要

如果你读过一些基于物理的渲染文献，你可能会发现有些文献使用辐射 R ，有些文献使用亮度 L 。这是无关紧要的，因为二者只相差一个系数 Y 。通过给方程两边乘上 Y 或者除以 Y ，就能实现 R 和 L 方程的互相转换。

照明

Alan Wake II (2023) / Remedy Entertainment

要有光。

现实中的光源



人造光源往往
是空间中的一个很
小的点源。
Credit: Chris Coe



太阳光源是最常
见的自然光源。
Credit: Laura Adai

光源抽象：点光源和方向光

点光源常用来表示人造光源



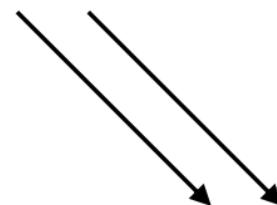
必须为点光源存储的数据

- 世界空间坐标 P_L
- 颜色
- 功率（强度）

$$\omega_i = P_L - P$$

P 是物体表面点的坐标。

方向光源常用来表示阳光



必须为方向光存储的数据

- 入射方向 ω_L
- 颜色
- 功率（强度）

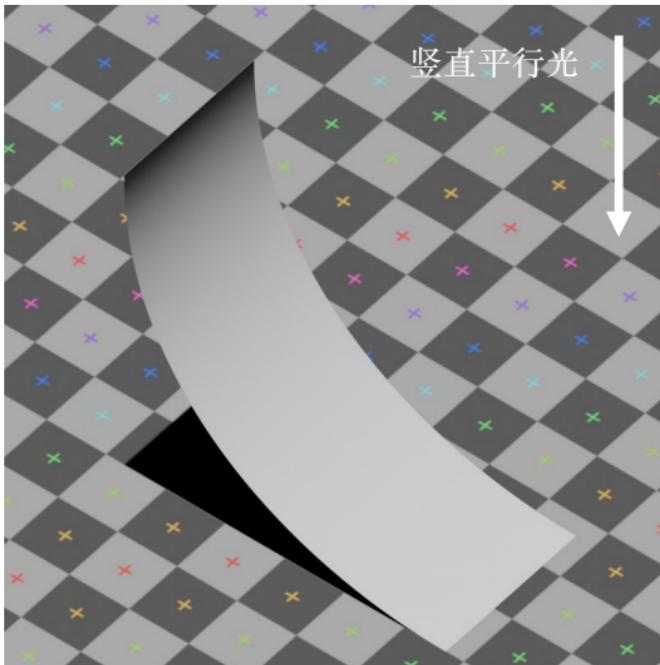
$$\omega_i = -\omega_L$$

不管物体表面点的坐标是什么，方向都一样。

* 在物理渲染中，我们习惯将点光源称为delta-光源，因为它在蒙特卡洛采样中的行为和 δ 函数一致。

** 别忘了我们约定表面上的入射方向和出射方向都是指离表面的。

Lambert余弦定律



方向为 ω_i 的光打到法线方向为 n 的微小面元上，原始功率为 R 。那么微小面元接收到的功率为：

$$R_{recv} = R \omega_i \cdot n$$

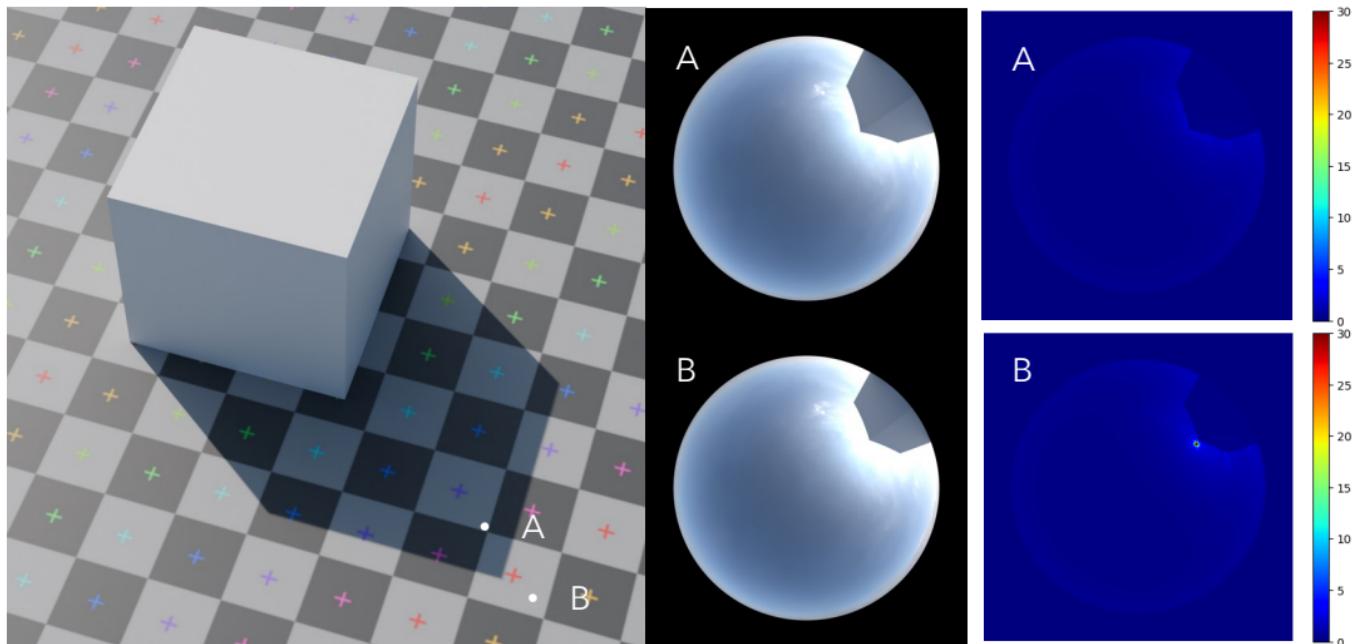
在许多文献中，这个内积被写作 $\cos \theta$ 。在后续的讨论中，这两种表示都可能使用。

* 如果你有电动力学或者流体力学基础，你可能会发现Lambert余弦定律实际上计算的就是通过微小面元的功率通量。或者说，平行于表面的功率分量不被计入表面的接受功率。

** 如果你好奇为什么功率可以进行直角分解，我认为学了电动力学以后你能很好地理解这个问题。

照明原理

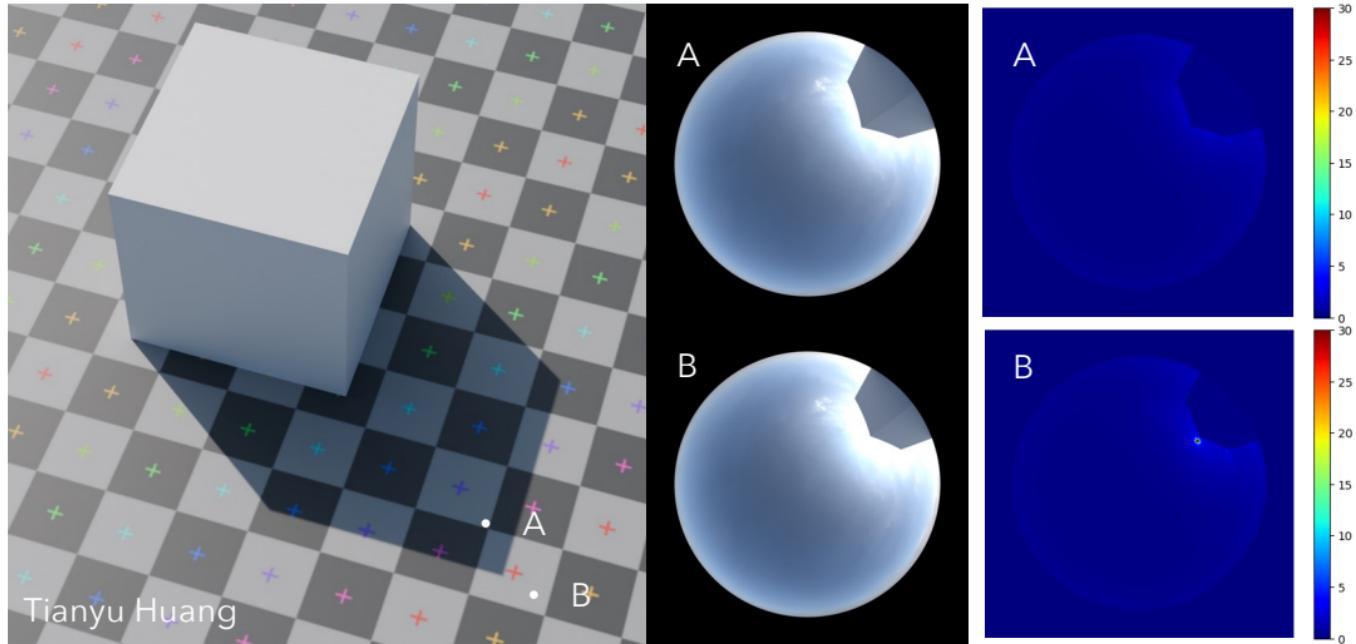
我们需要一个统一的理论，来描述明面、暗面和阴影。



照明原理

42

描述明面、暗面和阴影的统一理论：基于辐射功率！



不管是A点还是B点，从外界环境接收到的总功率都是对半球面的积分，可以用下面的公式描述：

$$\iint_{\theta, \varphi} R_i(\theta, \varphi) \cos \theta \sin \theta d\theta d\varphi = \int_{Hemisphere} R_i(\omega) \cos \theta d\omega$$

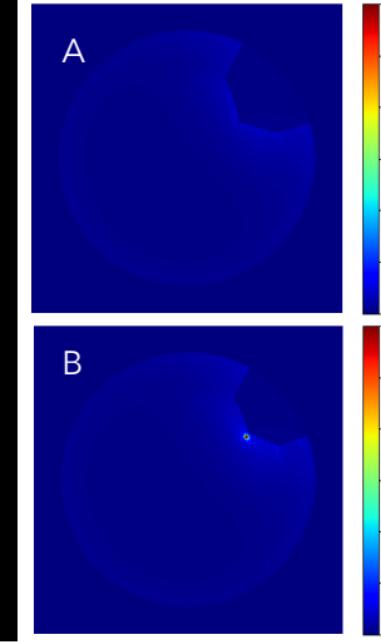
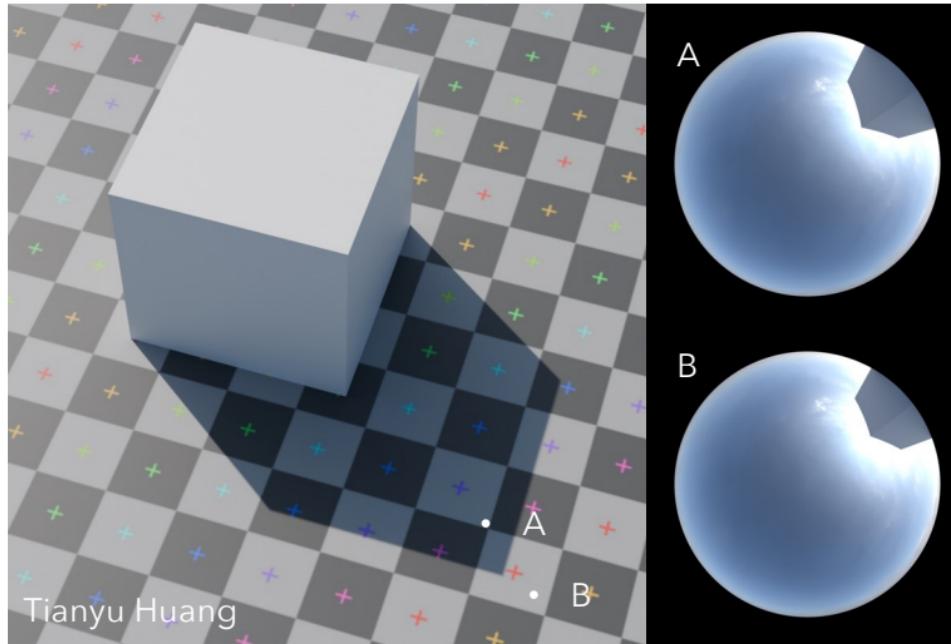
这个公式的结果被称为辐照度 (Irradiance)。这里的 $\cos \theta$ 就是Lambert余弦定律。

使用鱼眼镜头可以很好地直观展示球面积分的作用对象！

照明原理

43

描述明面、暗面和阴影的统一理论：基于辐射功率！



使用鱼眼镜头可以很好地直观展示球面积分的作用对象！

现在我们可以解释为什么平面上距离很近的A和B两个点，亮度差异如此巨大了：

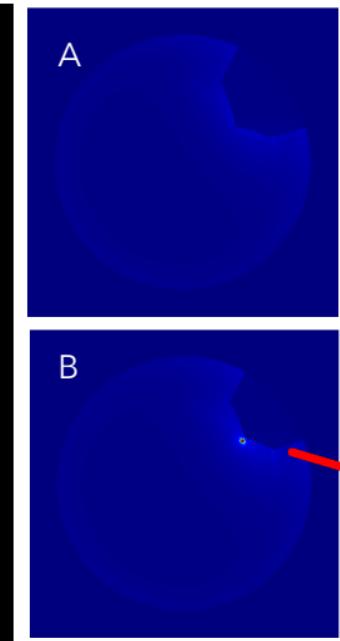
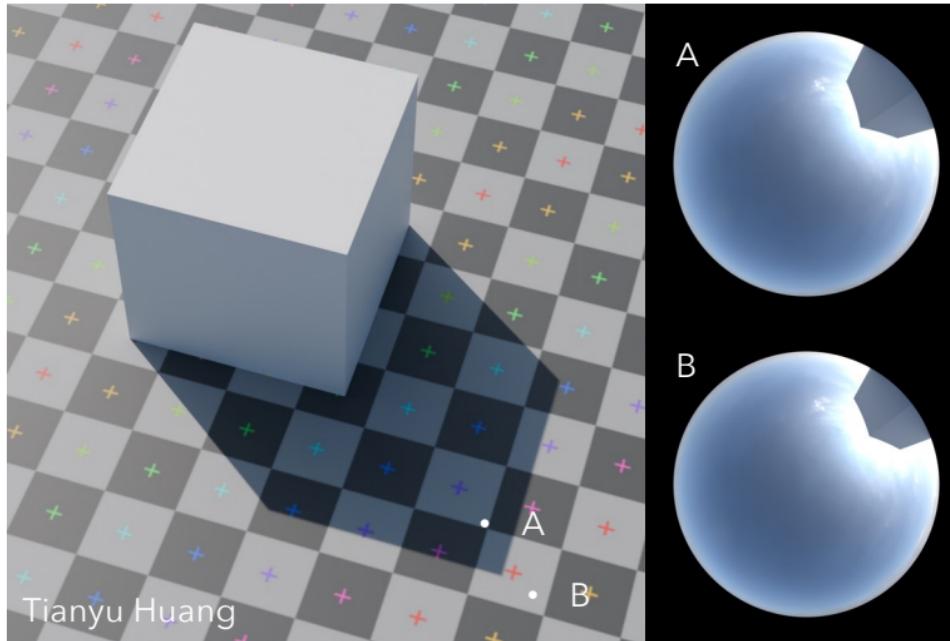
在B的辐照度积分中，太阳光源极大增大了积分结果。而太阳光源是一个非常狭窄的脉冲，从而能在遮挡物附近呈现出锐利的变化*。

* 在未专门进行优化的可微路径追踪中，这种情况会导致梯度爆炸。

照明原理

44

描述明面、暗面和阴影的统一理论：基于辐射功率！



使用鱼眼镜头可以很好地直观展示球面积分的作用对象！

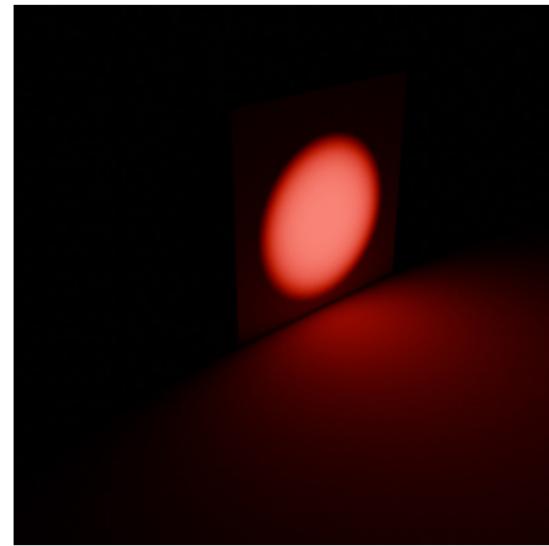
辐照度公式适用于物理世界几乎所有的宏观照明现象。

但是我们注意到，辐照度公式具备全局光照特性：它是一个积分方程，并且被积函数不解析，甚至可能不可微！在严格的基于物理的渲染，也就是路径追踪中，我们使用蒙特卡洛方法来计算这个积分；然而对于实时渲染来说，这是不现实的。

辐照度方程具备全局光照性质



月光照亮大地 / Nathanaël Desmeules



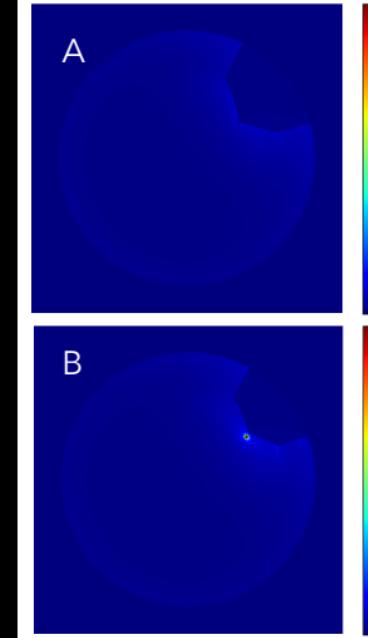
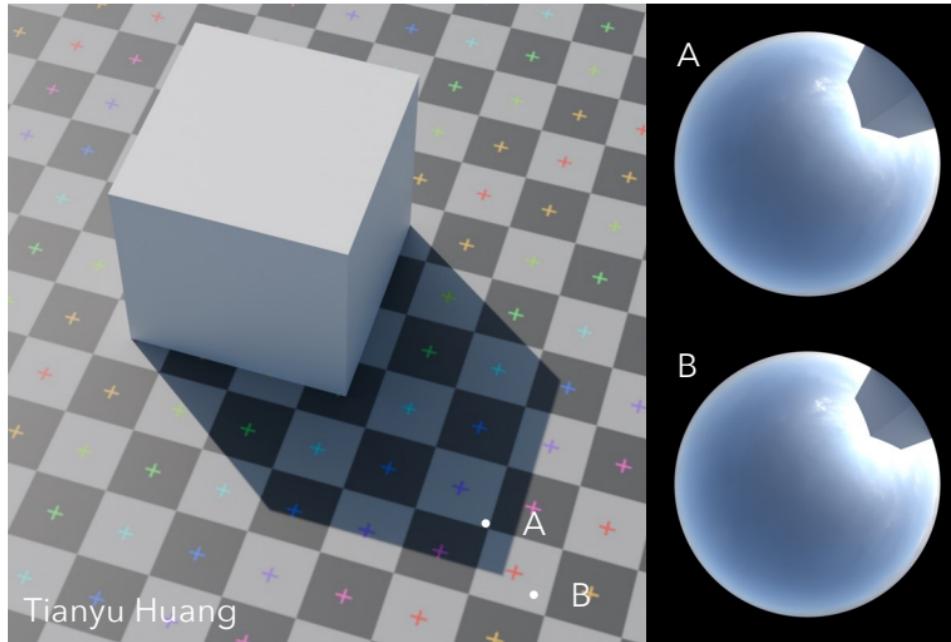
红色探照灯打到白色表面上，间接照亮地面

辐照度方程不关心谁是光源，它只是忠实地对半球入射功率进行积分。环境中所有的入射功率都会对辐照度起贡献。

照明原理

46

描述明面、暗面和阴影的统一理论：基于辐射功率！



使用鱼眼镜头可以很好地直观展示球面积分的作用对象！

辐照度公式适用于物理世界几乎所有的宏观照明现象。

但是我们注意到，辐照度公式具备全局光照特性：它是一个积分方程，并且被积函数不解析，甚至可能不可微！在严格的基于物理的渲染，也就是路径追踪中，我们使用蒙特卡洛方法来计算这个积分；然而对于实时渲染来说，这是不现实的。

辐照度方程的实时渲染简化

辐照度方程：

$$L_{in} = \int_{Hemisphere} R_{in}(\omega) \cos \theta d\omega$$

我们分离入射功率项：

$$R_{in}(\omega) = R_{light}(\omega) + R_{environment}(\omega)$$

然后进行重要简化：对场景中的所有点， $R_{environment}(\omega)$ 与所处空间位置无关，而完全人为给定（游戏引擎会自动生成一个看起来比较真实的反射盒参数）。

这个简化的实质是关闭了全局光照，而只考虑直接光源！

由此，辐照度方程可以简化为：

$$L_{in} = \int_{Hemisphere} R_{light}(\omega) \cos \theta d\omega + L_{env}$$

辐照度方程的实时渲染简化

由此，辐照度方程可以简化为：

$$L_{in} = \int_{Hemisphere} R_{light}(\omega) \cos \theta d\omega + L_{env}$$

还记得我们此前将光源抽象成点源吗？

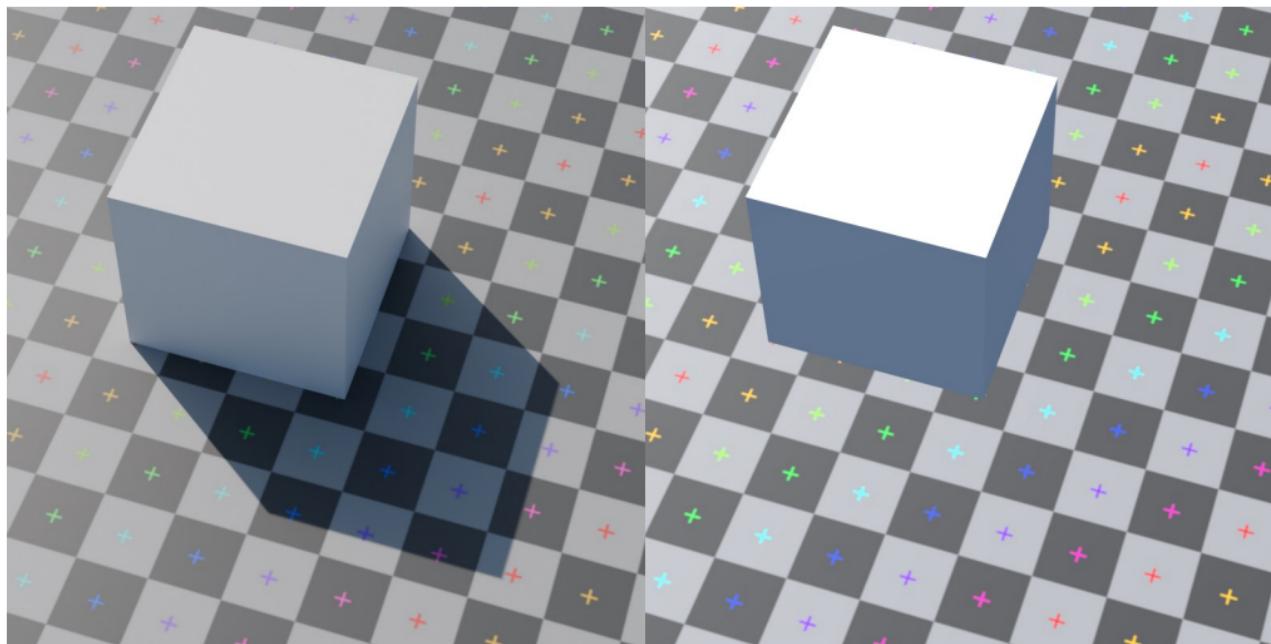
$$L_{in} = \int_{Hemisphere} R_{light} \cdot \delta_{light}(\omega) \cos \theta d\omega + L_{env} = L_{light} + L_{env}$$

其中 L_{light} 和 L_{env} 为可以在 $O(1)$ 时间内确定的值。

如果你有一定的实时渲染基础，那么你可能见过这个方程。是的，这就是Phong光照模型！

使用Phong简化以后.....

我们注意到明暗关系能被表示出来，但是阴影却消失了！



阴影消失是因为阴影是一个空间相关的现象。

我们需要做一些额外的工作把阴影请回来，稍后会介绍。

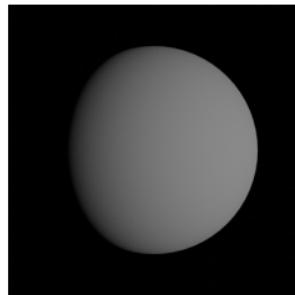
表面



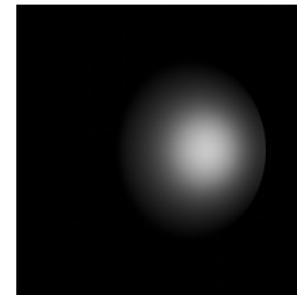
表面的建模：双向反射分布函数 (BRDF)

在现实世界中，不同的表面对不同的入射角度和反射角度有不同的反应。

- 对于漫射表面。
 - 固定观察方向，更改入射光方向，我们发现入射光方向越偏离法线方向，表面越暗。
 - 固定入射光方向，更改观察方向，我们发现似乎对于所有观察方向，表面的颜色都是一致的。
- 对于高光表面。
 - 固定观察方向，更改入射光方向，我们同样发现入射光方向越偏离法线方向，表面越暗。
 - 固定入射光方向，更改观察方向，我们发现当观察方向越接近于理想镜面反射方向，那么观察到的表面就越亮。



漫射球受到方向光源照射



高光球受到方向光源照射

表面的建模：双向反射分布函数 (BRDF)

在现实世界中，不同的表面对不同的入射角度和反射角度有不同的反应。

- 对于漫射表面。
 - 固定观察方向，更改入射光方向，我们发现入射光方向越偏离法线方向，表面越暗。
 - 固定入射光方向，更改观察方向，我们发现似乎对于所有观察方向，表面的颜色都是一致的。
- 对于高光表面。
 - 固定观察方向，更改入射光方向，我们同样发现入射光方向越偏离法线方向，表面越暗。
 - 固定入射光方向，更改观察方向，我们发现当观察方向越接近于理想镜面反射方向，那么观察到的表面就越亮。

如何将上述观察抽象成计算机能理解的数学形式？

表面的建模：双向反射分布函数 (BRDF)

定义反射分布函数 (Reflectance Distribution Function, RDF) *:

$$RDF(\omega_o; \omega_i) : \omega_o \mapsto \rho$$

表示在入射光 ω_i 固定的情况下，表面从 ω_o 看过去的反照率。

* 这似乎是只在本讲义中才有的概念，引用请注明出处。

表面的建模：双向反射分布函数 (BRDF)

定义反射分布函数 (Reflectance Distribution Function, RDF) *:

$$RDF(\omega_o; \omega_i) : \omega_o \mapsto \rho$$

表示在入射光 ω_i 固定的情况下，表面从 ω_o 看过去的反照率。

为了在一个函数中同时考虑入射角和反射角，我们定义双向反射分布函数 (Bidirectional Reflectance Distribution Function, BRDF) :

$$BRDF(\omega_o, \omega_i) = RDF(\omega_o; \omega_i)$$

* 这似乎是只在本讲义中才有的概念，引用请注明出处。

BRDF和渲染方程

定义反射分布函数（Reflectance Distribution Function, RDF）*：

$$RDF(\omega_o; \omega_i) : \omega_o \mapsto \rho$$

表示在入射光 ω_i 固定的情况下，表面从 ω_o 看过去的反照率。

为了在一个函数中同时考虑入射角和反射角，我们定义双向反射分布函数（Bidirectional Reflectance Distribution Function, BRDF）：

$$BRDF(\omega_o, \omega_i) = RDF(\omega_o; \omega_i)$$

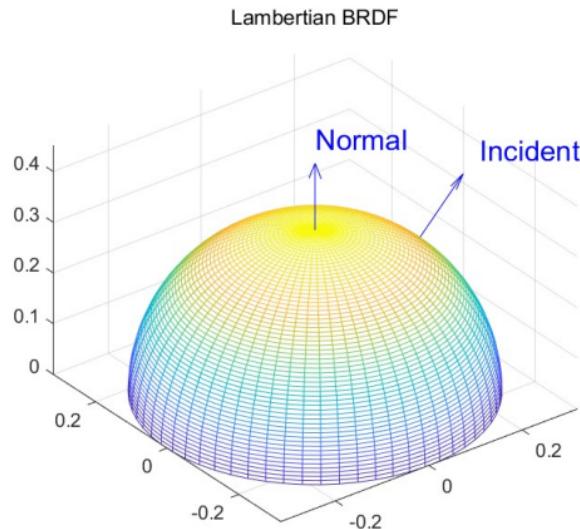
我们知道入射光不止一个方向，为了考虑全向入射光在出射光方向上产生的贡献，我们对BRDF进行积分：

$$R_o = \int_{Hemisphere} BRDF(\omega_o, \omega_i) \cdot \cos \theta \cdot R_i(\omega_i) d\omega_i$$

这就是BRDF表示的渲染方程。

* 这似乎是只在本讲义中才有的概念，引用请注明出处。

表面的建模：Lambertian漫射表面BRDF



Lambertian漫射表面的BRDF是：

$$BRDF(\omega_o, \omega_i) = \frac{\rho_0}{\pi}$$

ρ_0 是表面在完全白色照明*下呈现出的反照率。 π 是物理渲染为满足能量守恒引入的项（和球面角积分有关），在很多早期的电子游戏代码中看不到这一项。

* 想象一个电影特效有时会出现的画面：一个从各个方向看过去都是纯白的宇宙。

表面的建模：Oren-Nayar漫射表面BRDF



现实世界的满月

Lambert 描绘的满月

观察边缘！

现实世界的漫射表面被平行光照射时，边缘没有明显的亮度衰减。

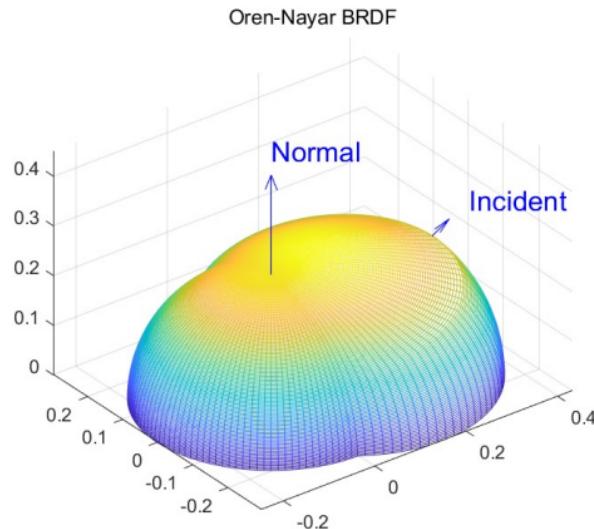
小时不识月，呼作白玉盘。

右图月球按1:1737建模

摄像机焦距为2m

Real moon photo credit:
Gregory H. Revera

表面的建模：Oren-Nayar漫射表面BRDF



Oren-Nayar漫射表面的BRDF是：

$$BRDF(\omega_o, \omega_i)$$

$$= \frac{\rho_0}{\pi} (A + B \max(0, \cos(\varphi_i - \varphi_o)) \sin \alpha \tan \beta)$$

其中

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}$$

$$B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max(\theta_i, \theta_o)$$

$$\beta = \min(\theta_i, \theta_o)$$

ρ_0 是表面在完全白色照明*下呈现出的反照率。

表面的建模：Oren-Nayar漫射表面BRDF

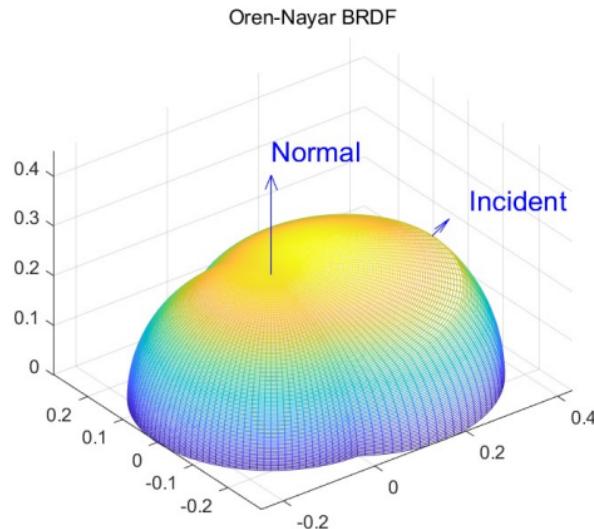
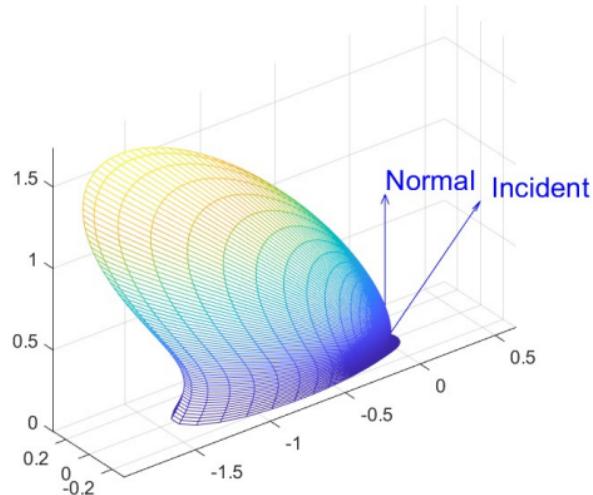


Image Credit: M. Oren and S. Nayar

表面的建模：GGX高光表面BRDF

GGX BRDF (Burley D distribution, GGX-Smith G distribution)



GGX高光表面的BRDF是：

$$BRDF(\omega_o, \omega_i) = \frac{\rho_0 DG}{4(\mathbf{n} \cdot \omega_i)(\mathbf{n} \cdot \omega_o)}$$

$$\alpha_x = roughness_x^2$$

$$\alpha_y = roughness_y^2$$

$$h = \frac{\omega_o + \omega_i}{|\omega_o + \omega_i|}$$

$$D = \frac{1}{\pi \alpha_x \alpha_y} \frac{1}{\left((\mathbf{h} \cdot \mathbf{n})^2 + \frac{(\mathbf{h} \cdot \mathbf{u})^2}{\alpha_x^2} + \frac{(\mathbf{h} \cdot \mathbf{v})^2}{\alpha_y^2} \right)^2}$$

$$G(\omega_o, \omega_i, h) = \frac{heaviside(\omega_i, h)heaviside(\omega_o, h)}{\Lambda(\omega_i) + \Lambda(\omega_o) + 1}$$

$$\Lambda(\omega_j) = \frac{1}{2} \left(\sqrt{\alpha^2 \tan^2 < \omega_j, \mathbf{n} >} + 1 - 1 \right)$$

你无需关心推导的详细过程，因为现代图形框架都内置了这一BRDF。

表面的建模：GGX高光表面BRDF

GGX高光表面的BRDF是：

$$BRDF(\omega_o, \omega_i) = \frac{\rho_0 D G}{4(\mathbf{n} \cdot \omega_i)(\mathbf{n} \cdot \omega_o)}$$

$$\alpha_x = roughness_x^2$$

$$\alpha_y = roughness_y^2$$

$$\mathbf{h} = \frac{\omega_o + \omega_i}{|\omega_o + \omega_i|}$$

$$D = \frac{1}{\pi} \frac{1}{\alpha_x \alpha_y} \frac{1}{\left((\mathbf{h} \cdot \mathbf{n})^2 + \frac{(\mathbf{h} \cdot \mathbf{u})^2}{\alpha_x^2} + \frac{(\mathbf{h} \cdot \mathbf{v})^2}{\alpha_y^2} \right)^2}$$

$$G(\omega_o, \omega_i, h) = \frac{heaviside(\omega_i, \mathbf{h})heaviside(\omega_o, \mathbf{h})}{\Lambda(\omega_i) + \Lambda(\omega_o) + 1}$$

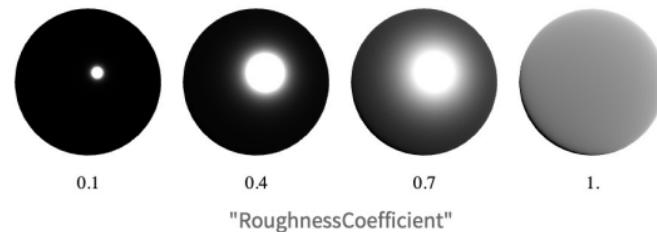
$$\Lambda(\omega_j) = \frac{1}{2}(\sqrt{\alpha^2 \tan^2 < \omega_j, \mathbf{n} >} + 1 - 1)$$

你无需关心推导的详细过程，因为现代图形框架都内置了这一BRDF。

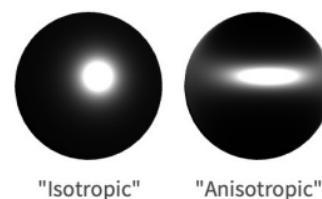
Tianyu Huang

$roughness_x$ 和 $roughness_y$ 是这一模型唯二需要控制的参数。

当 $roughness_x = roughness_y = roughness$ 时，表面各向同性。 (Image credit: Wolfram Research)



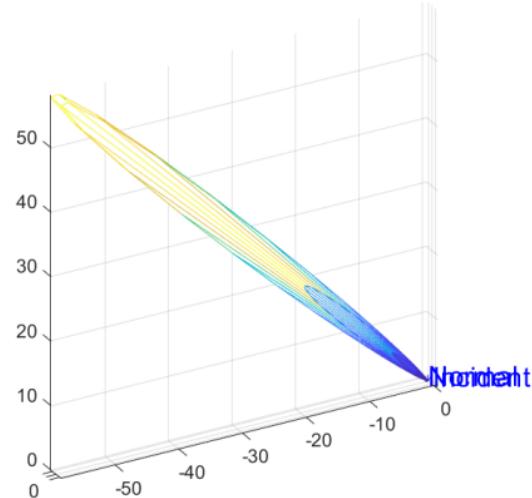
当 $roughness_x \neq roughness_y$ 时，表面各向异性，这种现象能在很多金属中观察到。



tianyu@illumiart.net

表面的建模：理想镜面BRDF

GGX BRDF (Burley D distribution, GGX-Smith G distribution)



将粗糙度调整为0.2，产生的GGX lobe如图所示。

我们注意到，随着粗糙度的减小，GGX BRDF越来越接近于delta函数。

$$\delta(\omega) = \begin{cases} \infty, \omega = 0 \\ 0, \text{otherwise} \end{cases}$$

$$\int_{Hemisphere} \delta(\omega) d\omega = 1$$

表面的建模：理想镜面BRDF

在基于点光源的着色系统里，无法渲染理想镜面！

在介绍完理想镜面BRDF以后，我们能更理解为什么是这样的：

$$BRDF(\omega_o, \omega_i) = \rho_0 \delta(\omega_i - \omega_r)$$

如果着色系统是基于点光源的，那么理想镜面BRDF中点光源产生的光斑是无限小的，所以没有任何渲染效果！

表面的建模：双向散射分布函数 (BSDF)

定义反射分布函数 (Reflectance Distribution Function, RDF) *:

$$RDF(\omega_o; \omega_i) : \omega_o \mapsto \rho$$

表示在入射光 ω_i 固定的情况下，表面从 ω_o 看过去的反照率。

为了在一个函数中同时考虑入射角和反射角，我们定义双向反射分布函数 (Bidirectional Reflectance Distribution Function, BRDF) :

$$BRDF(\omega_o, \omega_i) = RDF(\omega_o; \omega_i)$$

类似的，我们可以定义双向透射分布函数 $BTDF(\omega_o, \omega_i)$ 。

从而定义双向散射分布函数：

$$BSDF(\omega_o, \omega_i) = BRDF(\omega_o, \omega_i) + BTDF(\omega_o, \omega_i)$$

* 这似乎是只在本讲义中才有的概念，引用请注明出处。

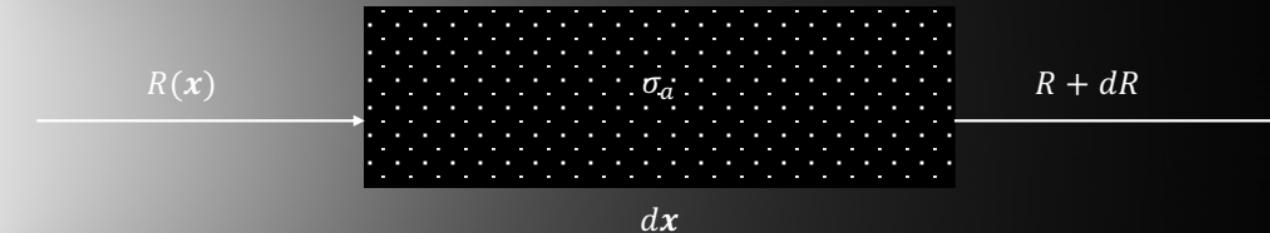
体积

Death Stranding (2019) / Kojima Productions

体积=宏观传输方程+微粒相函数

- 体积雾

惰性*各向同性体积的传输方程：吸收



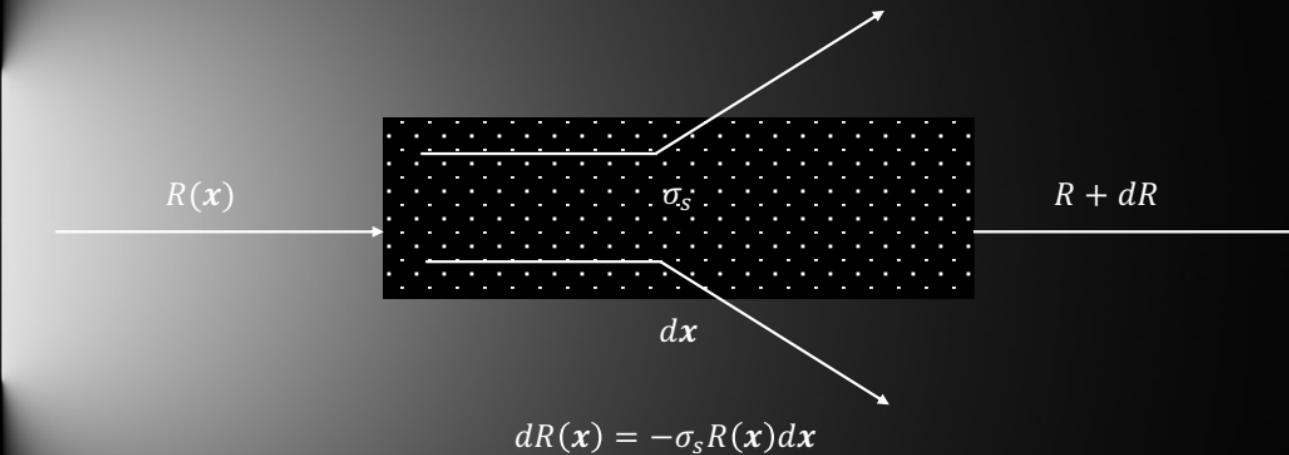
$$dR(\mathbf{x}) = -\sigma_a R(\mathbf{x}) d\mathbf{x}$$

* 不自发光。

Tianyu Huang

tianyu@illumiart.net

惰性*各向同性体积的传输方程：向外散射

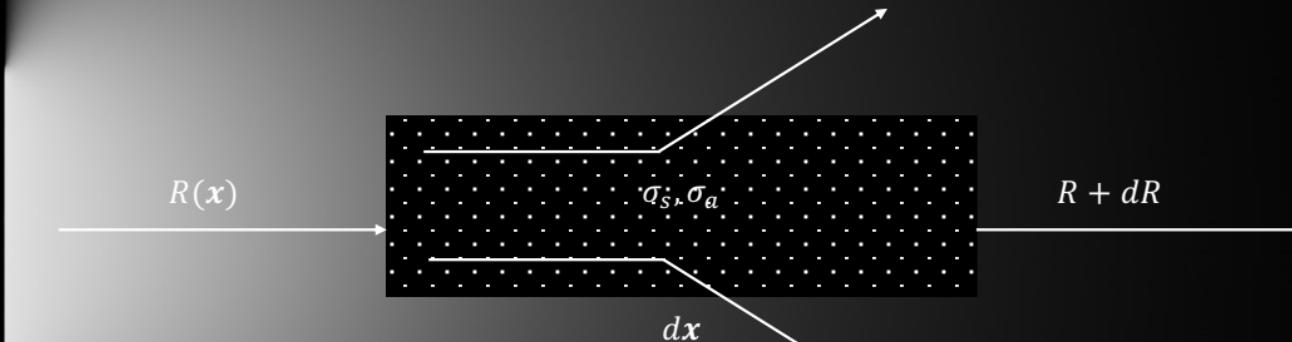


* 不自发光。

Tianyu Huang

tianyu@illumiart.net

惰性*各向同性体积的传输方程：消逝（Extinction）**



$$dR(x) = -(\sigma_s + \sigma_a)R(x)dx = -\sigma_t R(x)dx$$

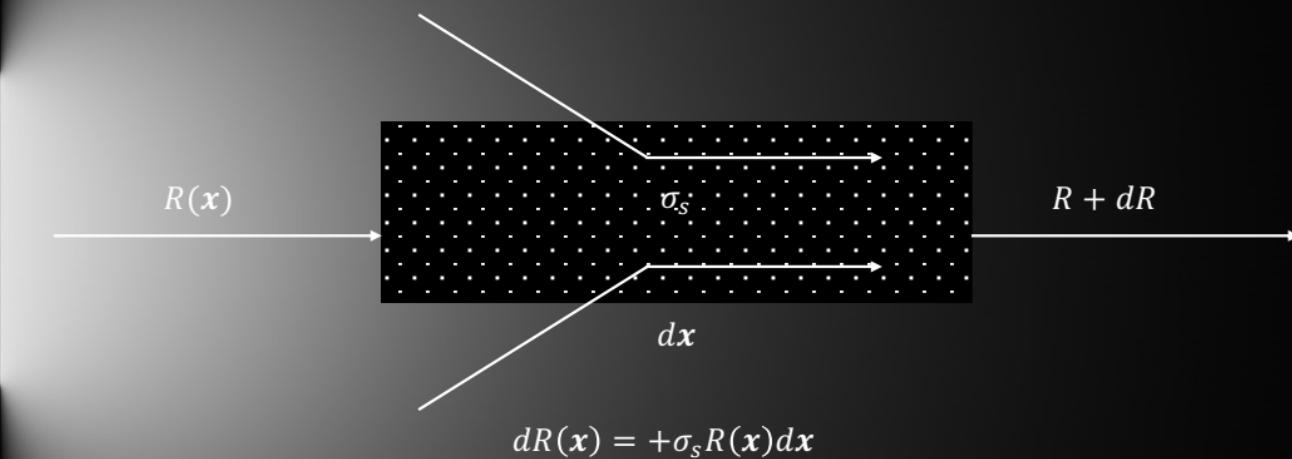
这个方程可以求解：

$$R(x) = R_0 \exp \int_{x_0}^x -\sigma_t dx = R_0 e^{-\sigma_t(x-x_0)}$$

* 不自发光。

** 一些文献称之为衰减（Attenuation）。

惰性*各向同性体积的传输方程：向内散射



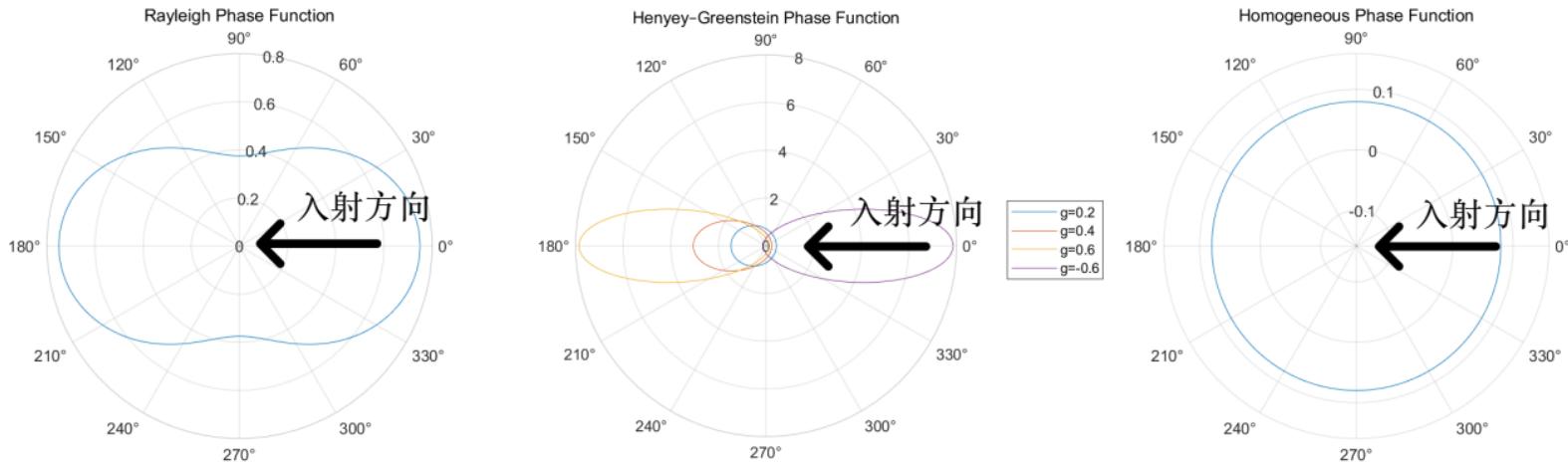
* 不自发光。

Tianyu Huang

tianyu@illumiart.net

体积微粒的相函数 (Phase Function)

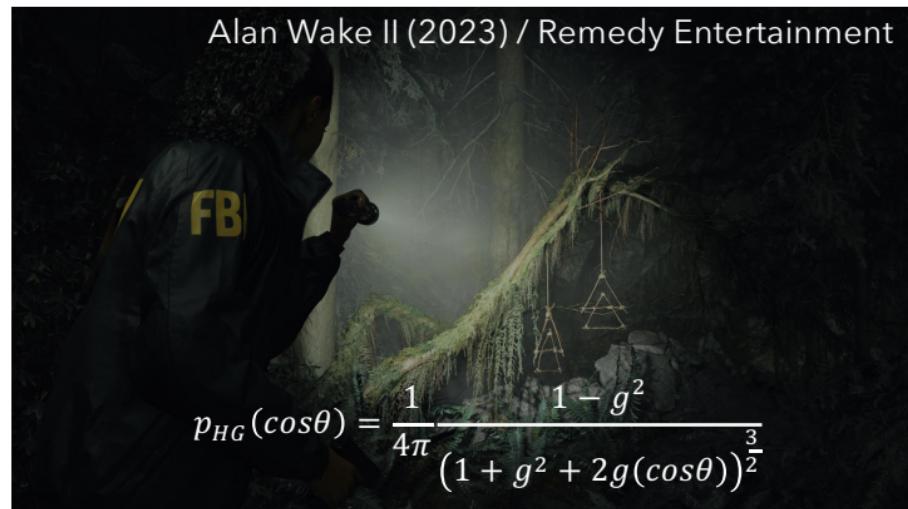
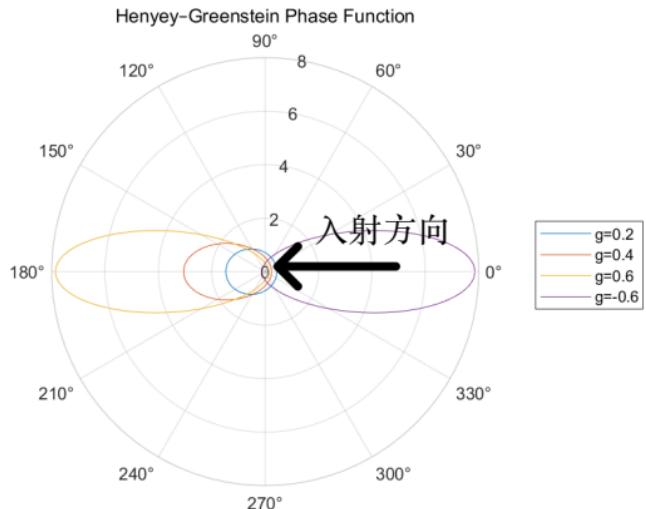
相函数刻画了单位能量从入射方向 (0° 方向) 进入体积微粒后的向外辐射情况。



Heney-Greenstein 相函数

Heney-Greenstein 相函数是一个经验公式，但是可以较好的在只修改一个参数的情况下拟合现实中的体积。

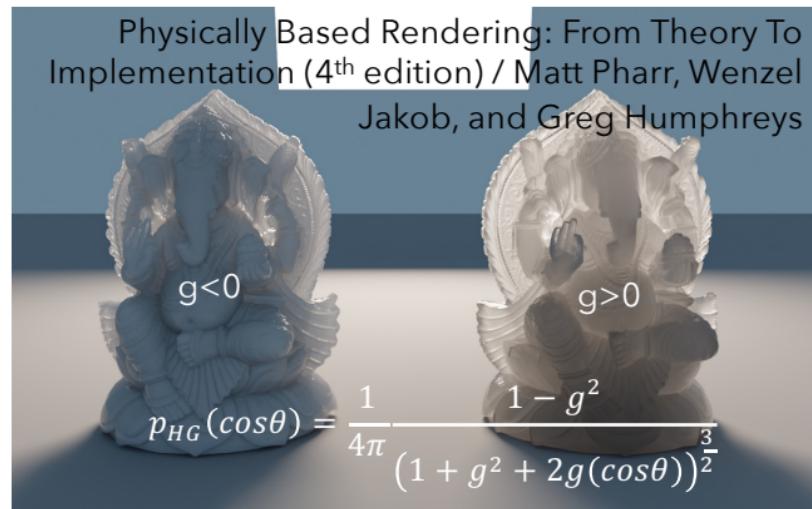
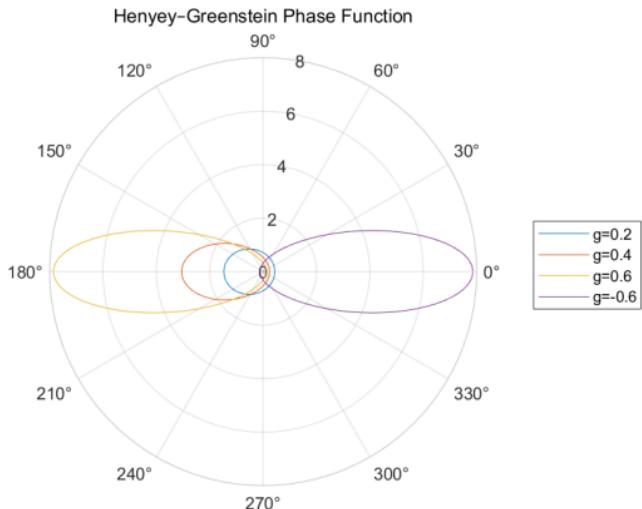
当 $g > 0$ 时，光击中体积微粒以后，主要的功率还是沿着原来的方向传播，但是有一些功率向两边扩散。空气水雾可以用这种情形模拟。



Henyey-Greenstein 相函数

Henyey-Greenstein 相函数是一个经验公式，但是可以较好的在只修改一个参数的情况下拟合现实中的体积。

当 $g > 0$ 时，光击中体积微粒以后，主要的功率还是沿着原来的方向传播，但是有一些功率向两边扩散。空气水雾可以用这种情形模拟。

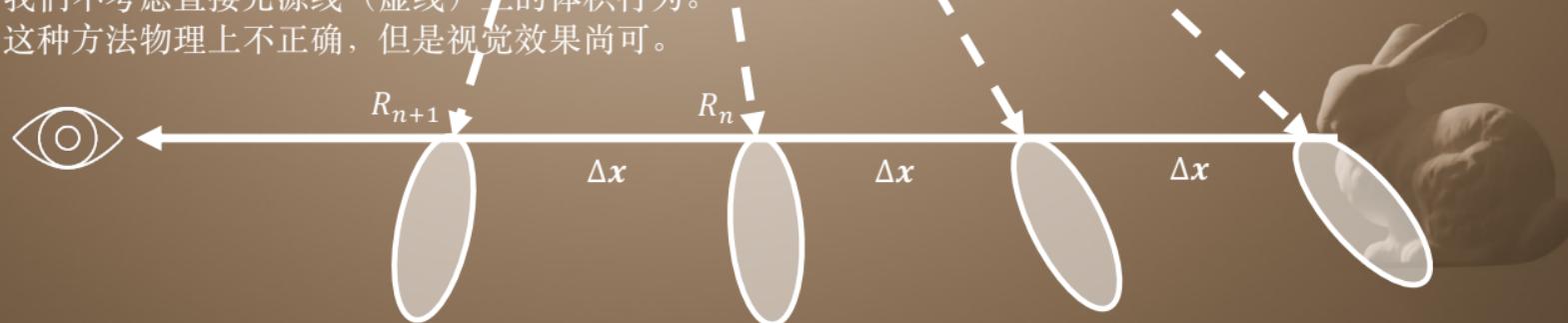


单次散射 (Single Scattering) 体积光线步进 (Ray Marching)

$$R_{n+1} = e^{-\sigma_t \Delta x} \left(R_n + \rho_{ss} \sum_{lights} p_{HG}(\cos \theta) R_{light} \right)$$

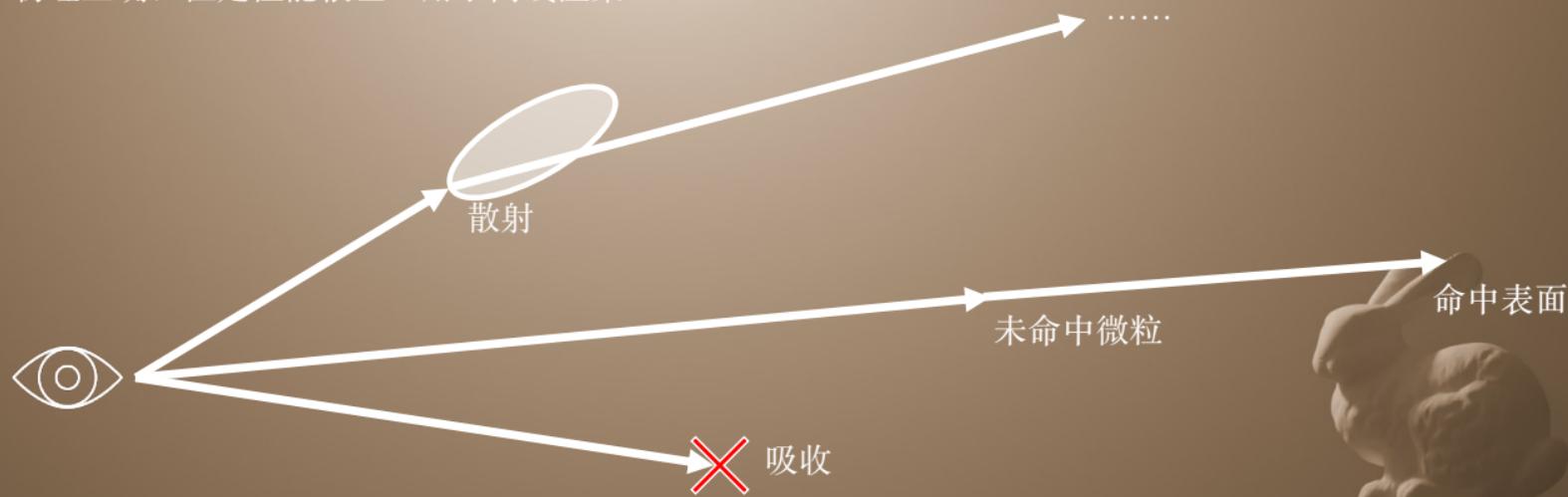
$$\rho_{ss} = \frac{\sigma_s}{\sigma_t}$$

我们不考虑直接光源线（虚线）上的体积行为。
这种方法物理上不正确，但是视觉效果尚可。



多重散射 (Multiple Scattering) / 随机游走 (Random Walk) Monte Carlo 体积光线步进

物理正确，但是性能很差。用于离线渲染。

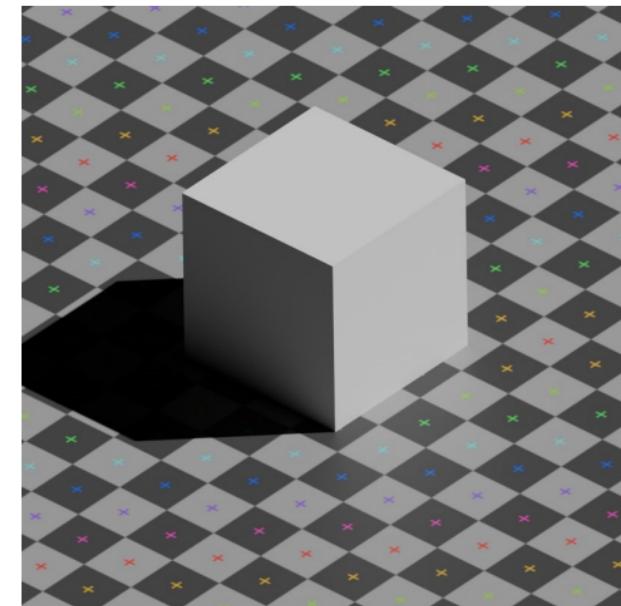
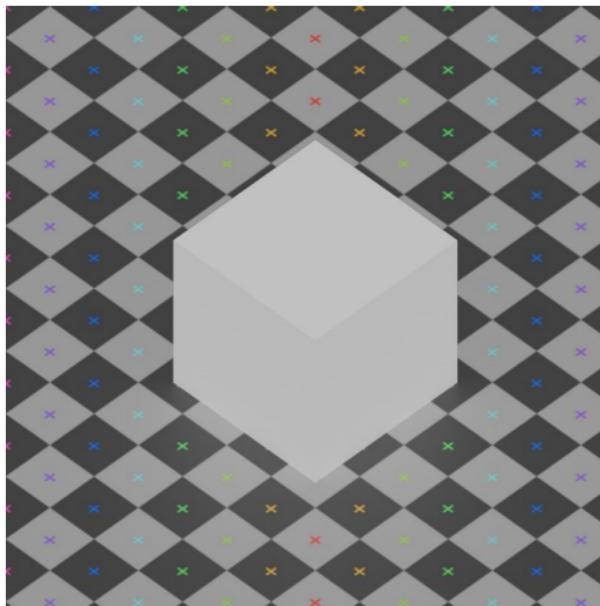


光栅阴影

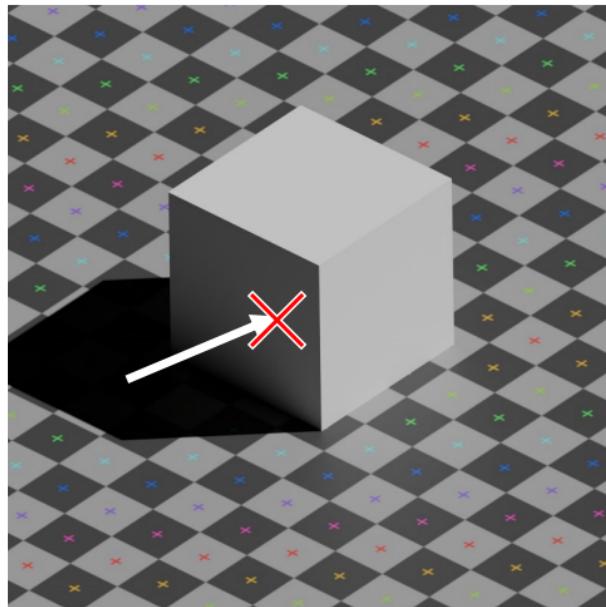


点光源和摄像机没有本质区别

光源是发光的眼睛，看到的一切都会被照亮，未看到的部分显示为阴影。（左图为光源视角）



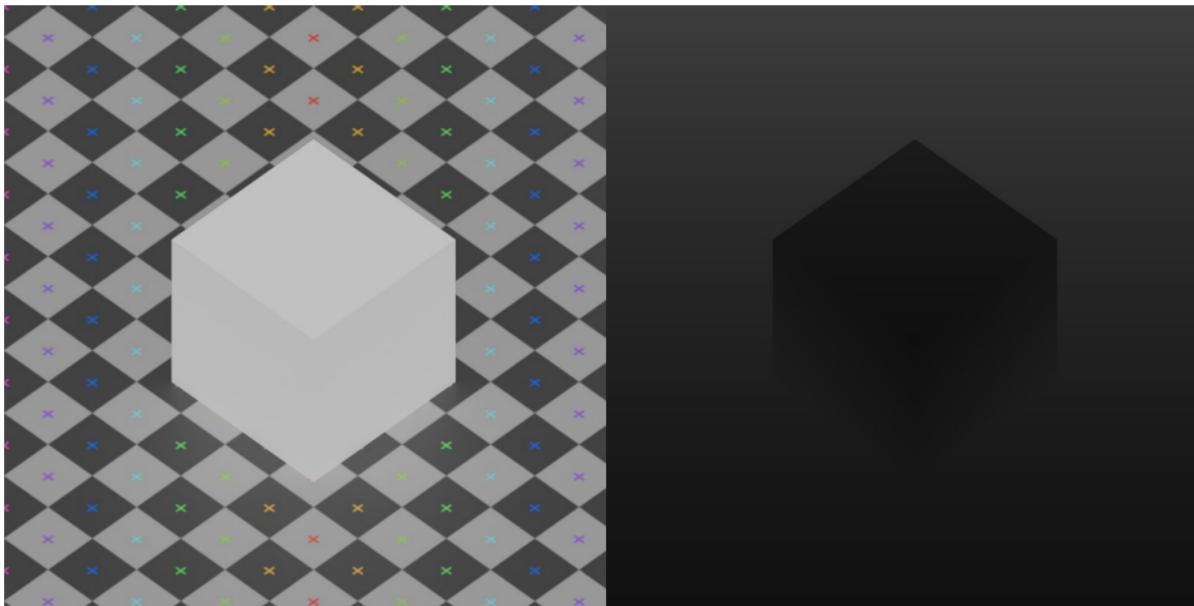
阴影生成的光线追踪实现



向光源方向发出射线，与场景中的物体求交。

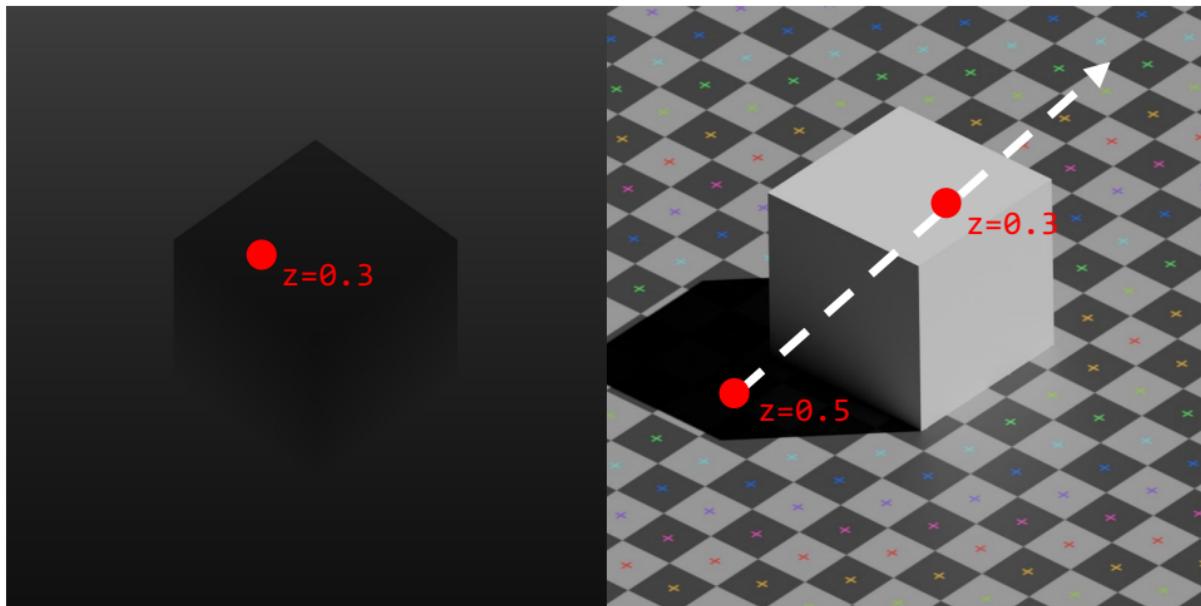
这种方法简单粗暴，但是与光栅管线不兼容，需要另外建立一套光线追踪管线，性能较差。

阴影生成的光栅实现



在光源处对深度通道进行一次光栅化。

阴影生成的光栅实现



在主摄像机处，我们对每一个点也可以得到深度信息。接着，我们使用摄像机变换矩阵将主摄像机处得到的深度转换成光源相机空间中的深度。

如果主摄像机的深度经过变换以后 $>$ 光源相机对应点处的深度，那么判定为阴影。

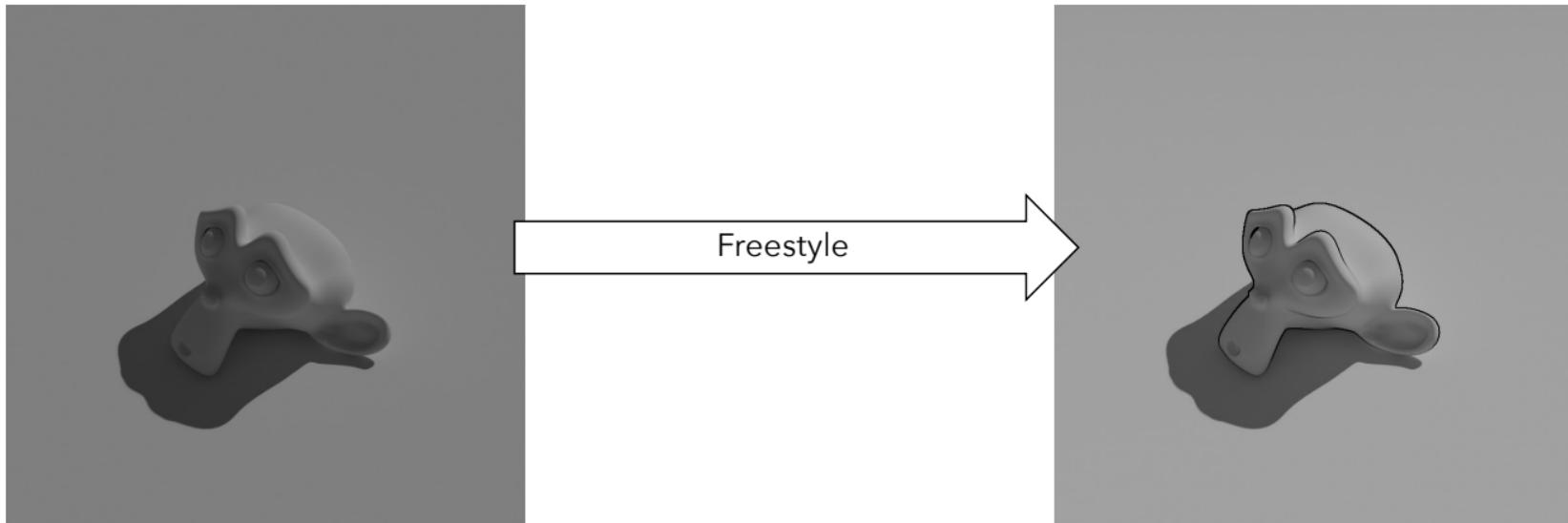
非真实感渲染 (Non-photorealistic Rendering, NPR)

BanG Dream! It's MyGO!!!! (2023) / Bushiroad

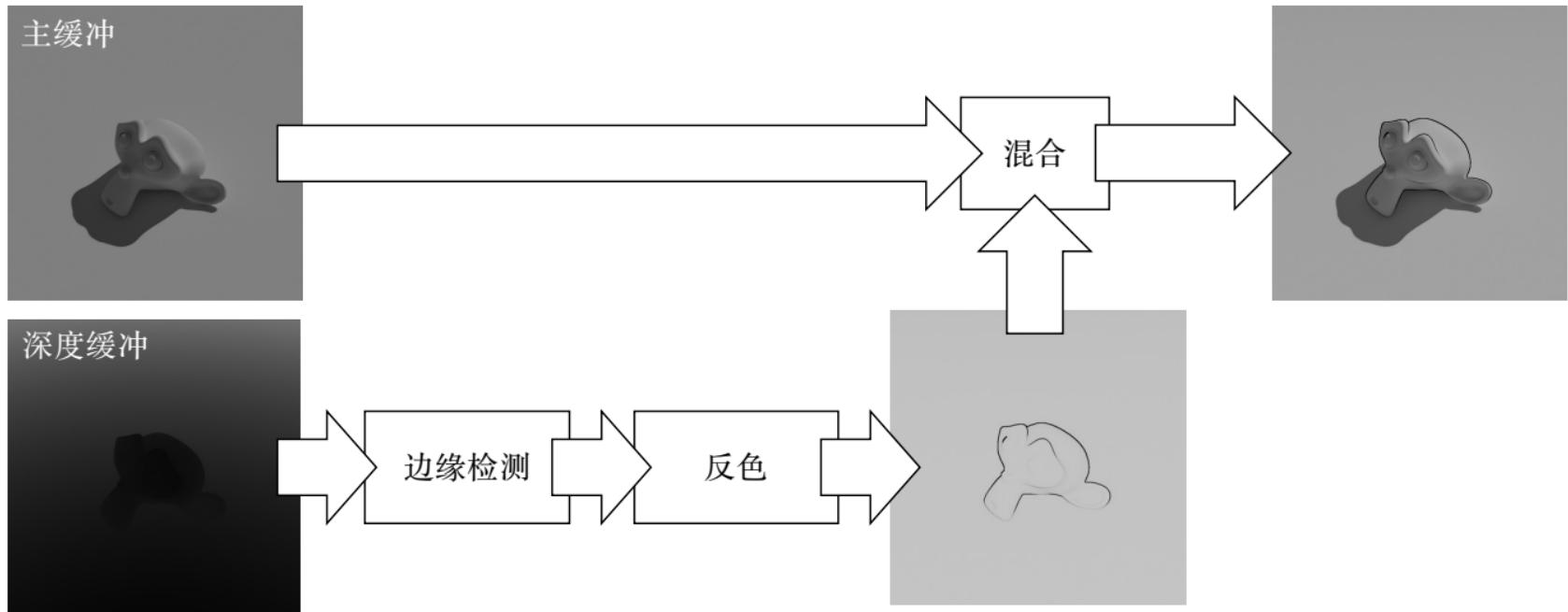
是的，三渲二就是NPR.....

Freestyle / Outlines

这项技术为物体边缘加上线框。

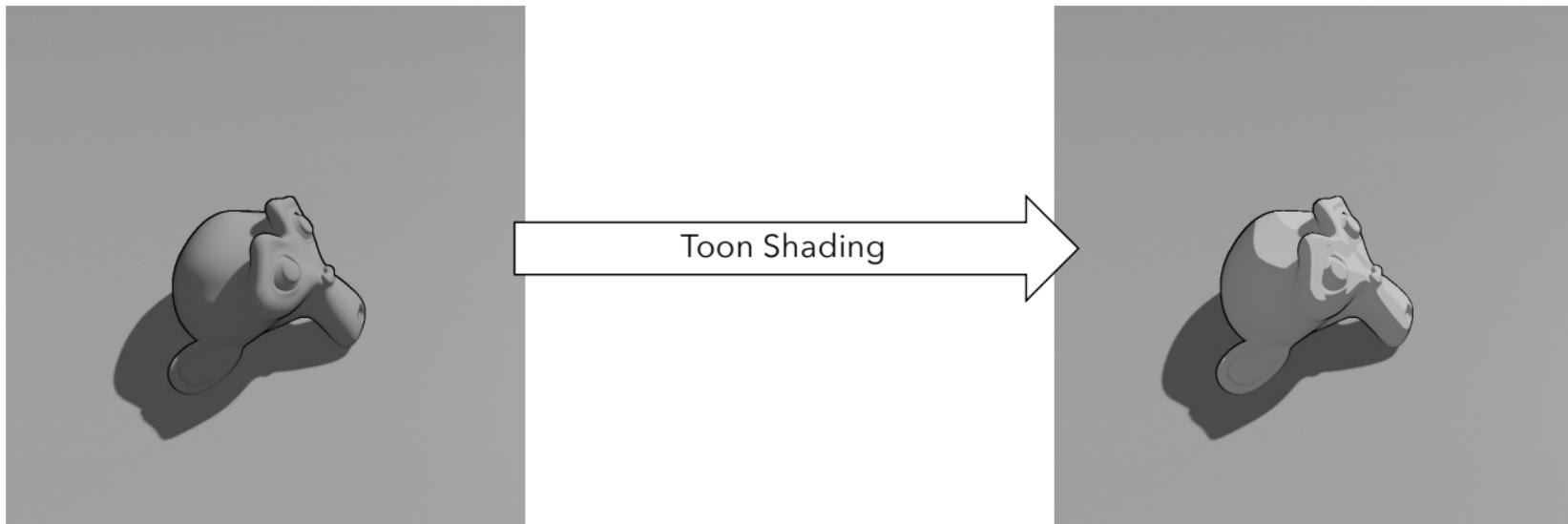


Freestyle / Outlines



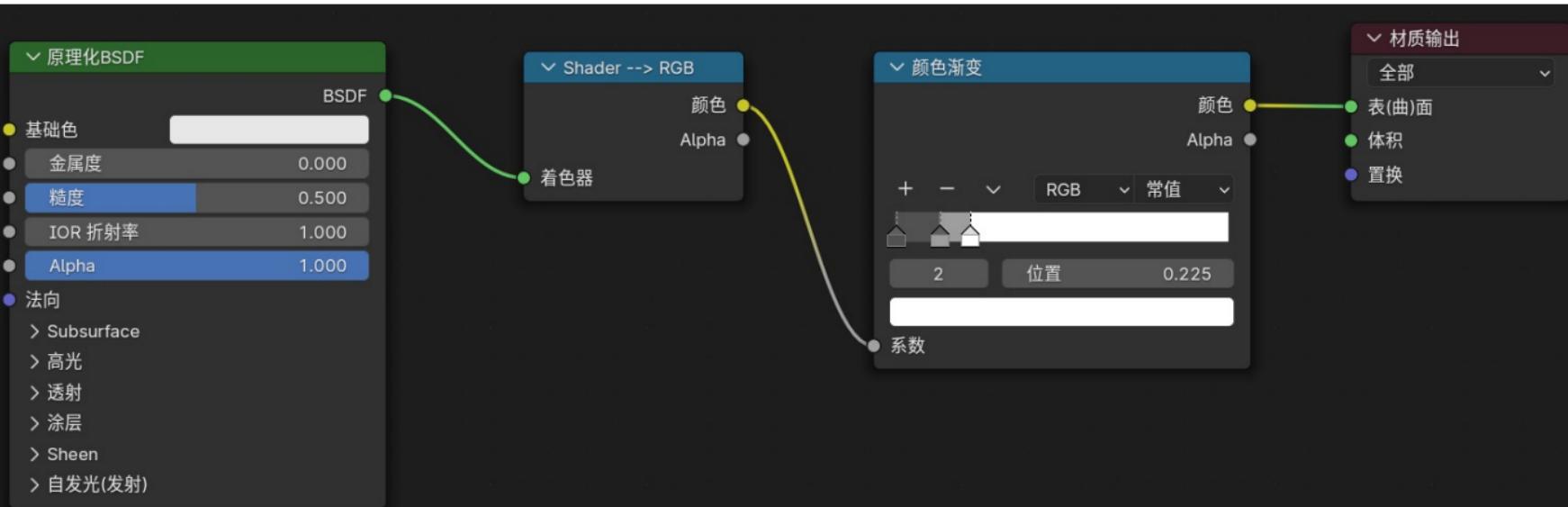
卡通 (Toon) 着色

模仿赛璐璐的上色风格，明暗具备非常明确的分界。

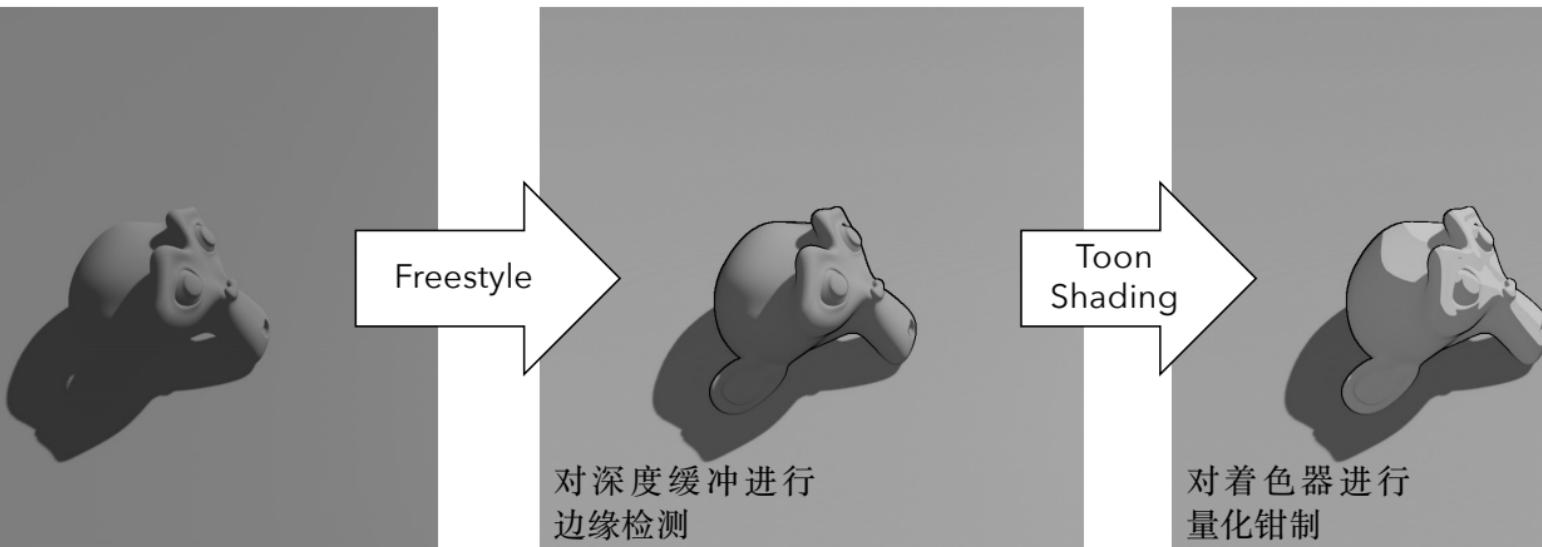


卡通 (Toon) 着色

对材质输出分段着色即可。



这就是最常见的NPR策略



光线追踪/路径追踪



图形学界的共识：光线追踪是物理渲染的完美算法。

但为什么实时渲染长期不使用光线追踪？

* 本讲义中，我们不区分光线追踪和路径追踪。这是因为目前主流的“光线追踪”程序实际上使用的都是路径追踪。

Tianyu Huang

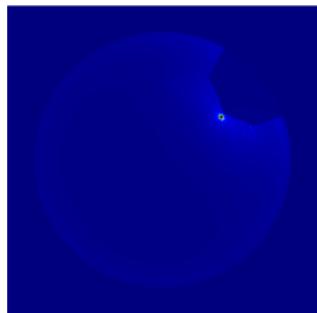
tianyu@illumiart.net

光线追踪是求解渲染方程积分最准确的方法

BSDF表示的渲染方程：

$$R_o = \oint BSDF(\omega_o, \omega_i) \cdot |\cos \theta| \cdot R_i(\omega_i) d\omega_i + R_{emission}$$

在大多数情况下，*BSDF*函数解析，但是不一定可微；*R_i*不解析也不可微。



回忆此前介绍照明的插图，这一个鱼眼半球视角的函数显然是无法给出简单的解析解的。图中天空和立方体之间具有非常巨大的梯度，可以认为是不可微的。

针对这种复杂积分，最优的方法是使用Monte Carlo方法。

渲染方程求解：Monte Carlo采样

BSDF表示的渲染方程：

$$\begin{aligned} R_o &= \oint BSDF(\omega_o, \omega_i) \cdot |\cos \theta| \cdot R_i(\omega_i) d\omega_i + R_{emission} \\ &= \sum_{k=1}^{samples} \frac{BSDF(\omega_o, \omega_{i,k}) \cdot |\cos \theta_k| \cdot R_i(\omega_{i,k})}{Samples \cdot PDF(\omega_{i,k})} + R_{emission} \end{aligned}$$

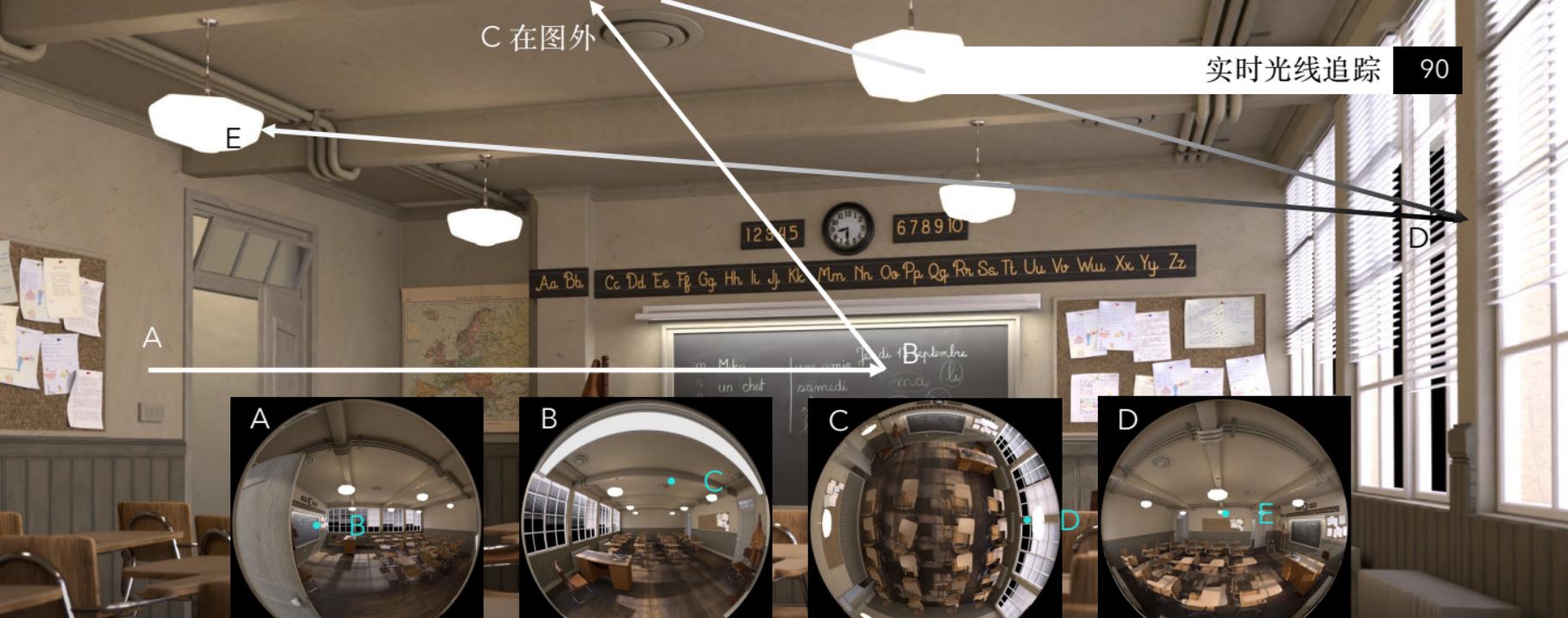
每次采样时生成 $\omega_{i,k}$ ，然后BSDF, PDF和余弦项唯一确定，但是 $R_i(\omega_{i,k})$ 如何确定？

我们只需要把 $R_i(\omega_{i,k})$ 放在渲染方程的左边，再进行一次积分即可，由此无限递归下去，直到：

- 达到递归限制。
- 下一层递归的贡献可以忽略不计。
- 光线命中无穷远处。
- 俄罗斯轮盘赌算法判决当前递归路径终止。
- 体积渲染器中，微粒模式为“吸收”。

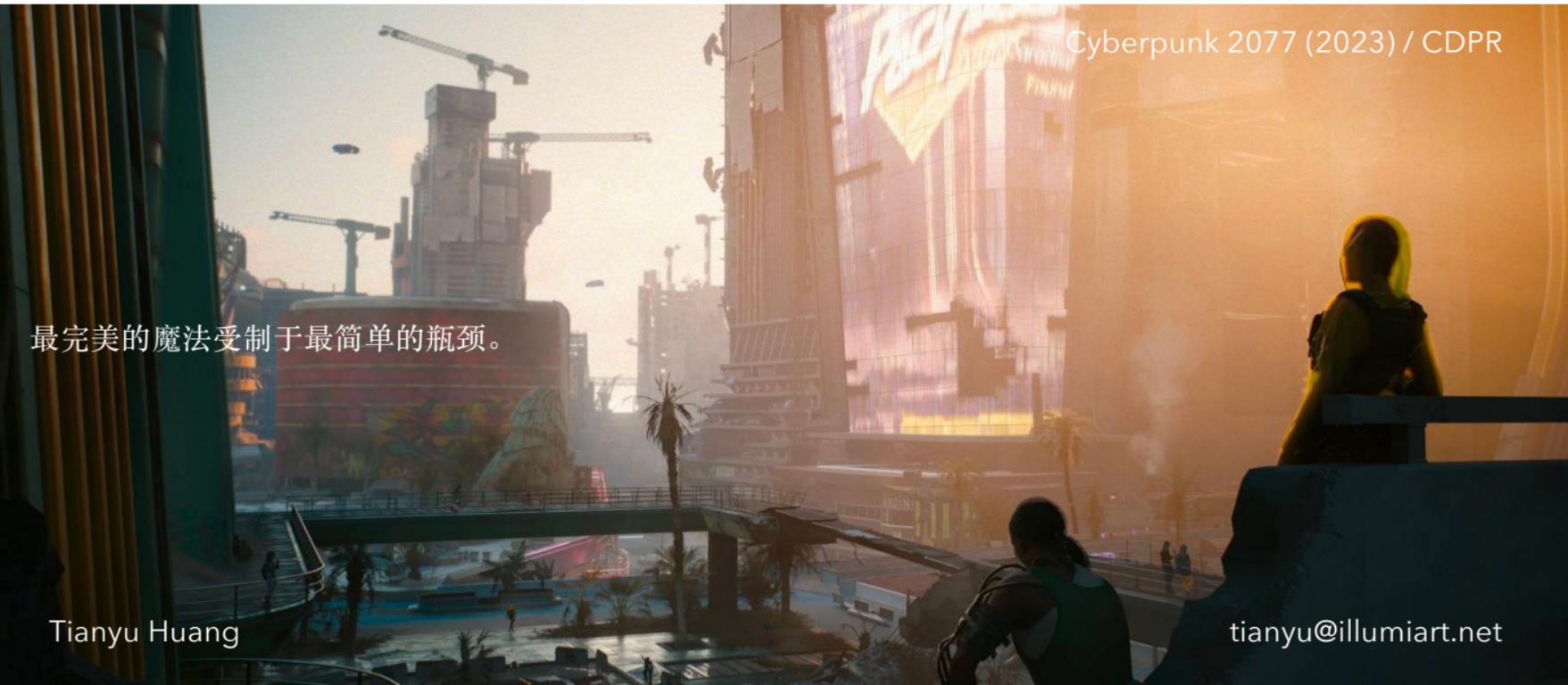


$$R_o = \oint BSDF(\omega_o, \omega_i) \cdot |\cos \theta| \cdot R_i(\omega_i) d\omega_i + R_{emission} = \sum_{k=1}^{Samples} \frac{BSDF(\omega_o, \omega_{i,k}) \cdot |\cos \theta_k| \cdot R_i(\omega_{i,k})}{Samples \cdot PDF(\omega_{i,k})} + R_{emission}$$



“光线”（实质是蒙特卡洛采样线）在空间中走出了一条路径，所以这种手段也被称为路径追踪。

光线追踪加速



最完美的魔法受制于最简单的瓶颈。

Cyberpunk 2077 (2023) / CDPR

光线追踪：时间复杂度分析

光线追踪的性能瓶颈是三角形求交。

- 若场景三角形数量为 n ，那么在BVH优化下，单个像素进行一次采样和一次反弹的时间复杂度为 $O(\log n)$ 。
- 三角形求交本身涉及到复杂的浮点操作，所以时间复杂度带有一个较大的常数。
- 要使图像具备较低的噪点，需要增加采样次数。
- 要使图像符合物理世界的全局光照，需要增加反弹次数。

光线追踪

$$O(C \cdot width \cdot height \cdot samples \cdot bounces \cdot \log n)$$

$C > 1$ 且不可忽略

为了保证全局光照准确性，没有太多手段减小 n

光栅

$$O(C \cdot n)$$

可以通过许多手段减小 n

在理想情况下，屏幕上的像素点最多只被访问一次

光线追踪：加速手段

加快求交速度

- 并行化 (Intel Embree, RTX GPUs)
- 硬件 BVH (RTX GPUs)
- 使用体素网格而非三角形网格
- LOD

减少求交次数

- 下一事件预测 (Next Event Estimation, NEE) , 在每一跳都对光源进行采样
- 路径引导 (Path Guiding) , 这和下一事件预测类似, 是一个新兴的研究领域
- (神经网络) 辐照缓存
- 神经网络多重重要性采样
- AI 降噪 (Intel OpenImageDenoise, NVIDIA OptiX)
- 超分辨率与插帧 (NVIDIA DLSS etc.)



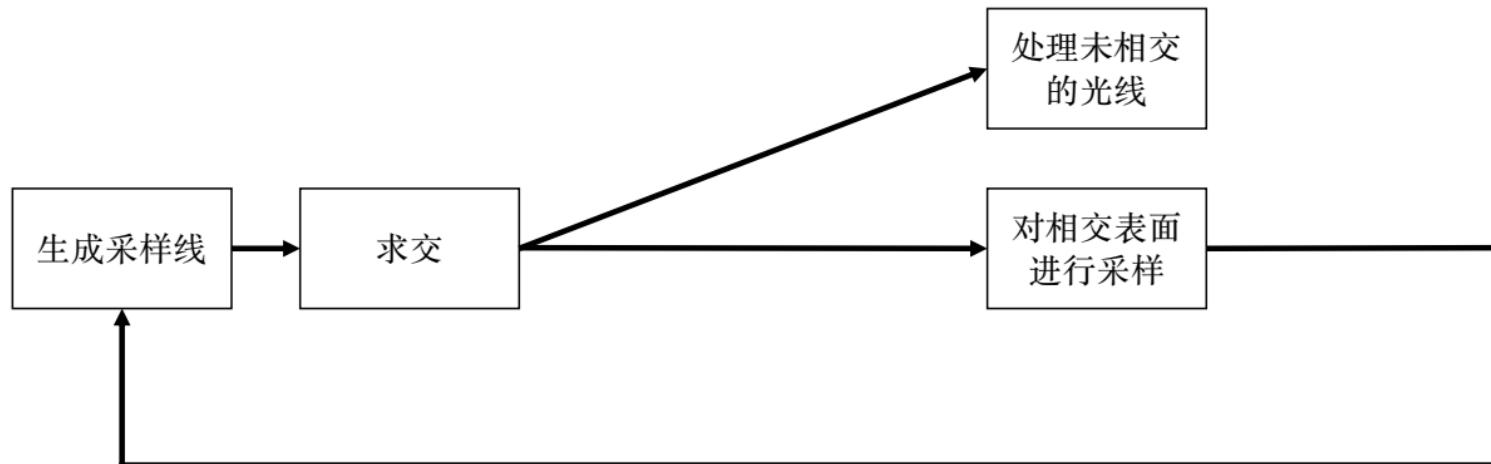
$$R_o = \sum_{k=1}^{Samples} \frac{BSDF(\omega_o, \omega_{i,k}) \cdot |\cos \theta_k| \cdot R_i(\omega_{i,k})}{Samples \cdot PDF(\omega_{i,k})} + \sum_{sampled\ lights} \frac{BSDF(\omega_o, \omega_{light}) \cdot |\cos \theta_{light}| \cdot R_{light}(\omega_{light}) \cdot w_{nee}}{Samples \cdot PDF(\omega_{light})} + R_{emi} \cdot w_{nee}$$

$$R_o = \sum_{k=1}^{Samples} \frac{BSDF(\omega_o, \omega_{i,k}) \cdot |\cos \theta_k| \cdot R_i(\omega_{i,k})}{Samples \cdot PDF(\omega_{i,k})} + R_{emission}$$

$R_i(\omega_{i,k})$ 和 $PDF(\omega_{i,k})$ 是在线训练的神经网络输出。

波前 (Wavefront) 路径追踪渲染管线的最简形式

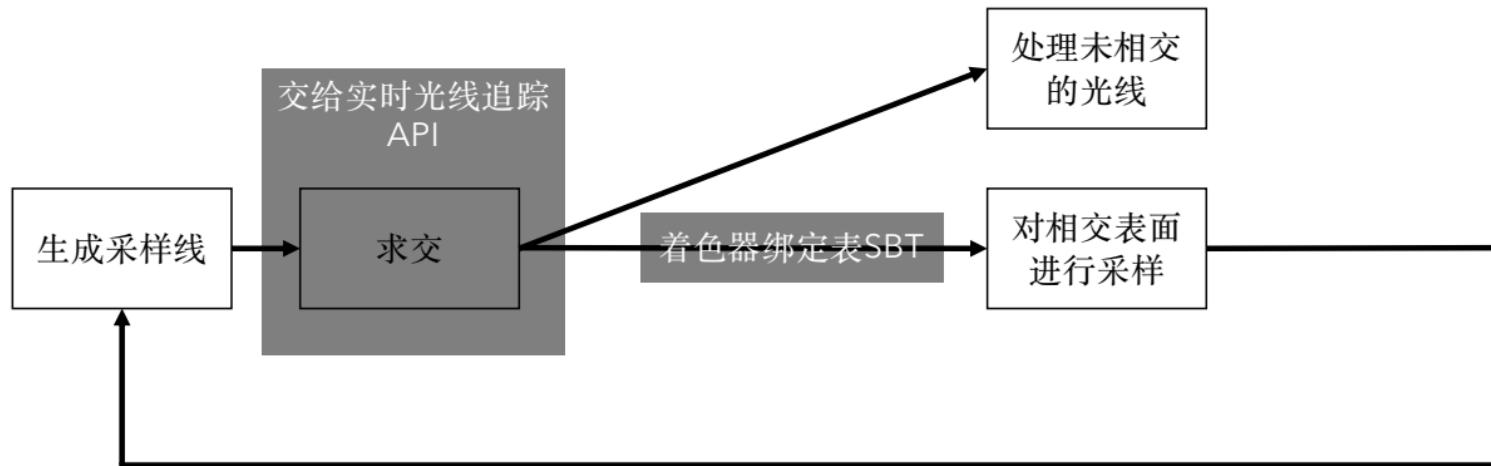
如何将路径追踪拆分成流水线运行？波前 (Wavefront) 方法是一种可行模式。



NVIDIA的OptiX库就以这种形式运行。（当然，由于NEE和体积的参与，OptiX的实现更加复杂）

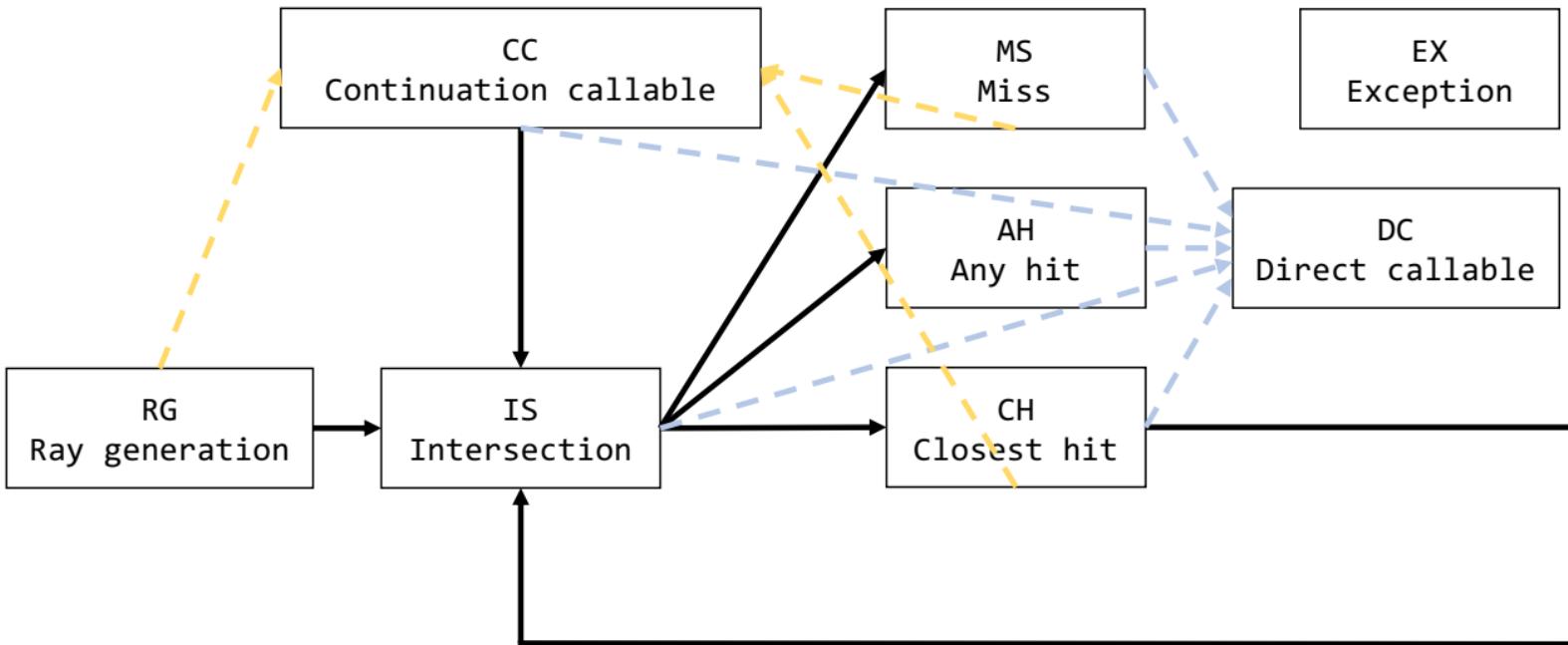
波前 (Wavefront) 路径追踪渲染管线的最简形式

如何将路径追踪拆分成流水线运行？波前 (Wavefront) 方法是一种可行模式。

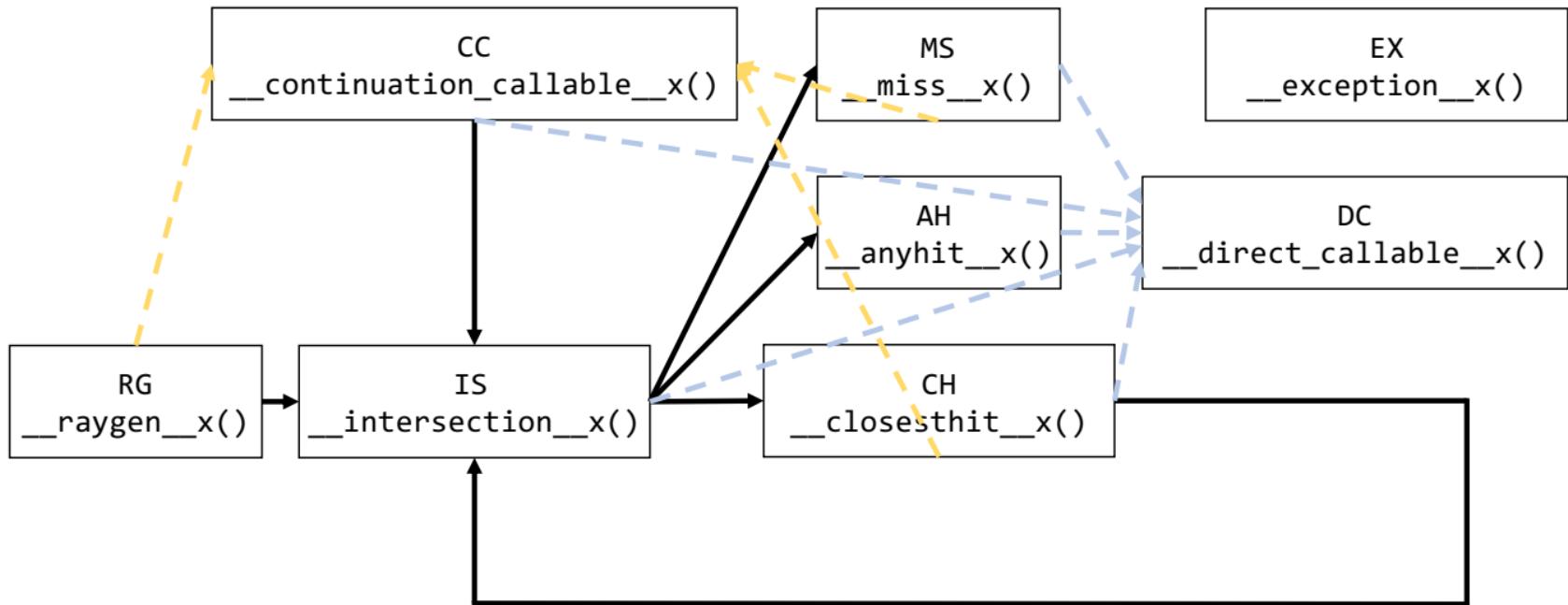


NVIDIA的OptiX库就以这种形式运行。 (当然，由于NEE和体积的参与，OptiX的实现更加复杂)

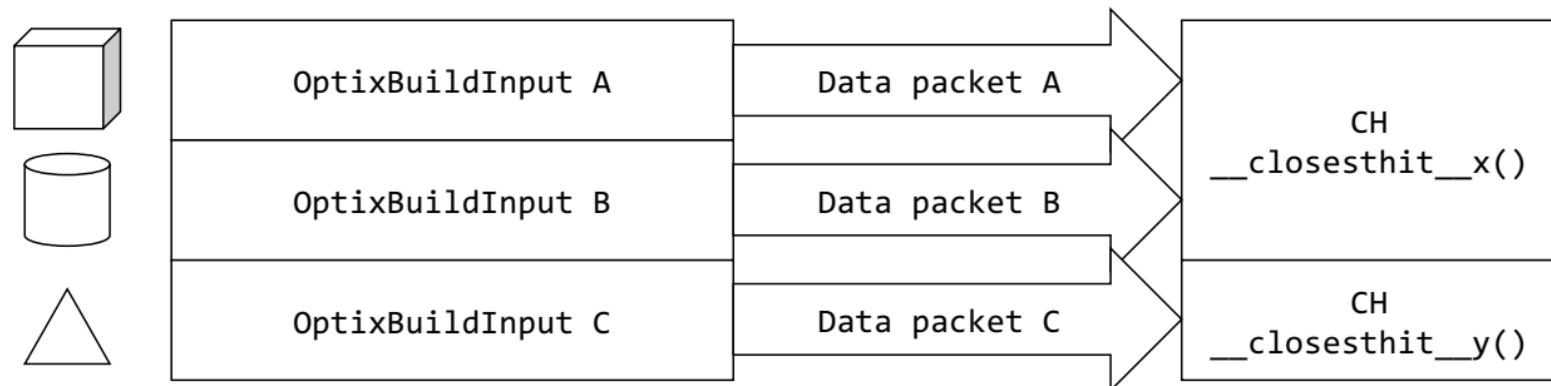
NVIDIA OptiX 8.0 流水线结构



每一个方框都是一个程序组（Program Group）



着色器绑定表 (SBT) 选择程序组并且装载数据



常用于对不同模型赋予不同材质。

在GPU内存中存储顶点： AOS还是SOA?

AOS: Array-of-Structures

v0.x	v0.y	v0.z	v0.u	v0.v	v1.x	v1.y	v1.z	v1.u	v1.v
------	------	------	------	------	------	------	------	------	------

SOA: Structure-of-Arrays

v0.x	v0.y	v0.z	v1.x	v1.y	v1.z	v2.x	v2.y	v2.z	v3.x
v0.u	v0.v	v1.u	v1.v	v2.u	v2.v	v3.u	v3.v	v4.u	v4.v

```
struct {  
    float3 pos;  
    float2 uv;  
} aos[64];
```

```
struct soa {  
    float3 pos[64];  
    float2 uv[64];  
};
```

```
struct soa {  
    float3 *pos;  
    float2 *uv;  
};
```

GPU同样受到局部性原理约束。在大多数情况下，SOA都优于AOS。

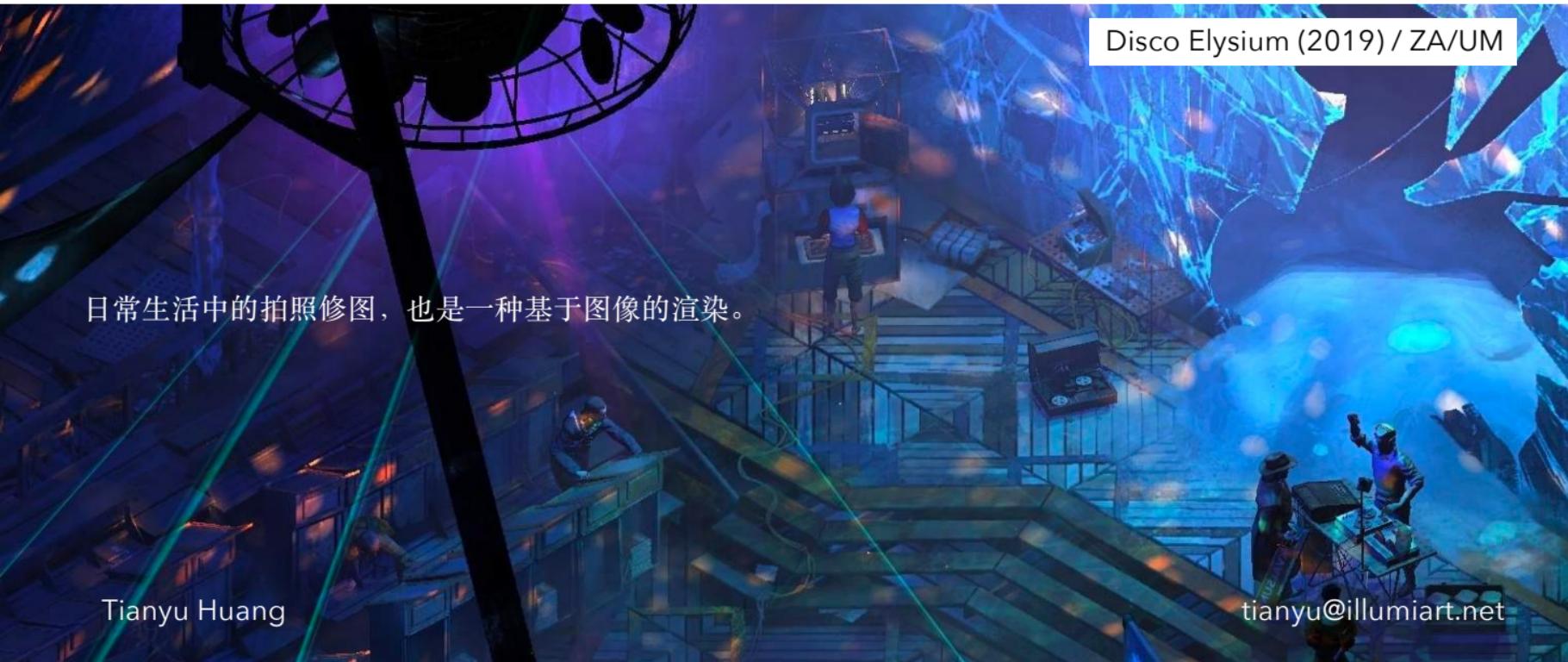
幻术

OCTOPATH TRAVELER II (2023) / Square Enix

限制艺术家发挥的重大瓶颈，就是实时渲染至少30帧的帧率限制。

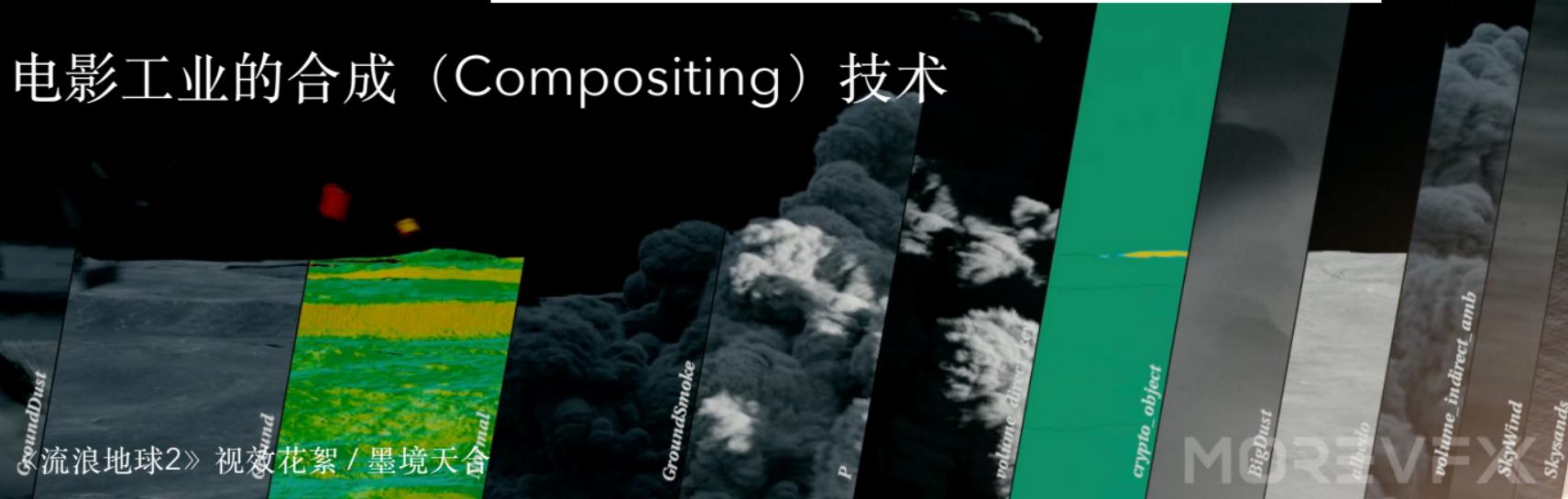
如果我们没法完美地模拟物理现象，那为什么不去试试欺骗眼睛呢？

基于图像的渲染 (Image-based Rendering)



日常生活中的拍照修图，也是一种基于图像的渲染。

电影工业的合成（Compositing）技术



合成（Compositing）就是使用两张或者多张图片合并成一张图片，从而创造出特殊的视觉效果的技术。

合成技术：

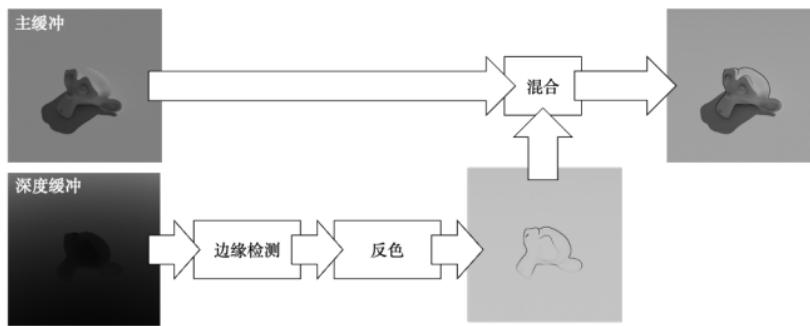
- 基于已经拍摄或者渲染完成的图片
- 将这些图片通过某种手段混合在一起

电影工业的合成（Compositing）技术



在渲染领域，我们称合成技术为一种基于图像的（Image-based）渲染技术。因为合成的对象都是已经渲染完成的图像（尽管这些图片可能效果并不好，是半成品）。

是不是有点熟悉？



在“非真实感渲染”一节中，我们介绍的NPR渲染流程就是一种基于图像的渲染技术。

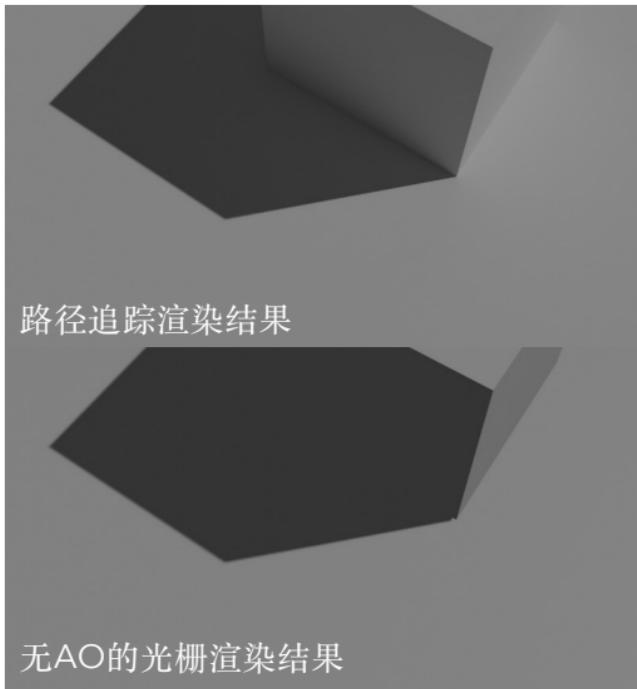
流程图左侧的两张图像已经走完了完整的光栅渲染流程，但我们并没有将主缓冲直接呈现到屏幕上，而是对图片进行处理以后再呈现到屏幕上。

事实上，几乎所有的后期处理技术都属于基于图像的渲染技术。

环境光遮蔽/环境吸收 (Ambient Occlusion, AO)



采用简化光照模型带来的问题



前面已经介绍过，在光栅渲染管线中，我们避免了渲染方程积分的计算，这使得着色失去了大部分的空间信息。

在面与面相交处的点，大部分的照明立体角都被遮挡，所以表现得比其他地方更暗—这是渲染方程的积分项可以精确描述的，但在光栅渲染管线中却无法呈现这一信息，导致暗面一片死黑。

为了解决这个问题，我们使用环境光遮蔽/环境吸收，为光栅渲染打上一个补丁。

环境光遮蔽是渲染方程的极致简化

正如其名，在环境光遮蔽的计算中，我们只考虑“遮蔽”。

或者说，我们考虑在完全白色照明*下，使用渲染方程描述的白色漫射世界（白模）的颜色。

$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{Hemisphere} V(\mathbf{p}, \boldsymbol{\omega}) \cos\theta d\boldsymbol{\omega}$$

$V(\cdot)$ 是遮挡函数，表示从表面上的点 \mathbf{p} 沿着 $\boldsymbol{\omega}$ 方向发出一条无穷远的射线，是(0)否(1)会被遮挡。



* 在材质模型部分介绍过这个概念，就是一个朝向各个方向看去，只要没有遮挡都呈现为纯白色的世界。

环境光遮蔽的精确计算

$$AO(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{Hemisphere} V(\mathbf{p}, \boldsymbol{\omega}) \cos\theta d\boldsymbol{\omega} = \frac{1}{\pi} \sum_{Samples} \frac{V(\mathbf{p}, \boldsymbol{\omega}_{sample}) \cos\theta}{Samples \cdot PDF(\boldsymbol{\omega}_{sample})}$$

精确计算环境光遮蔽需要实现进行一次反弹的路径追踪器。尽管由于只进行一次反弹，这种路径追踪器运行得确实很快，但是它和光栅渲染管线并不兼容。

pbtrt-v4实现了AOIntegrator类，用于精确计算环境光遮蔽，可供参考。

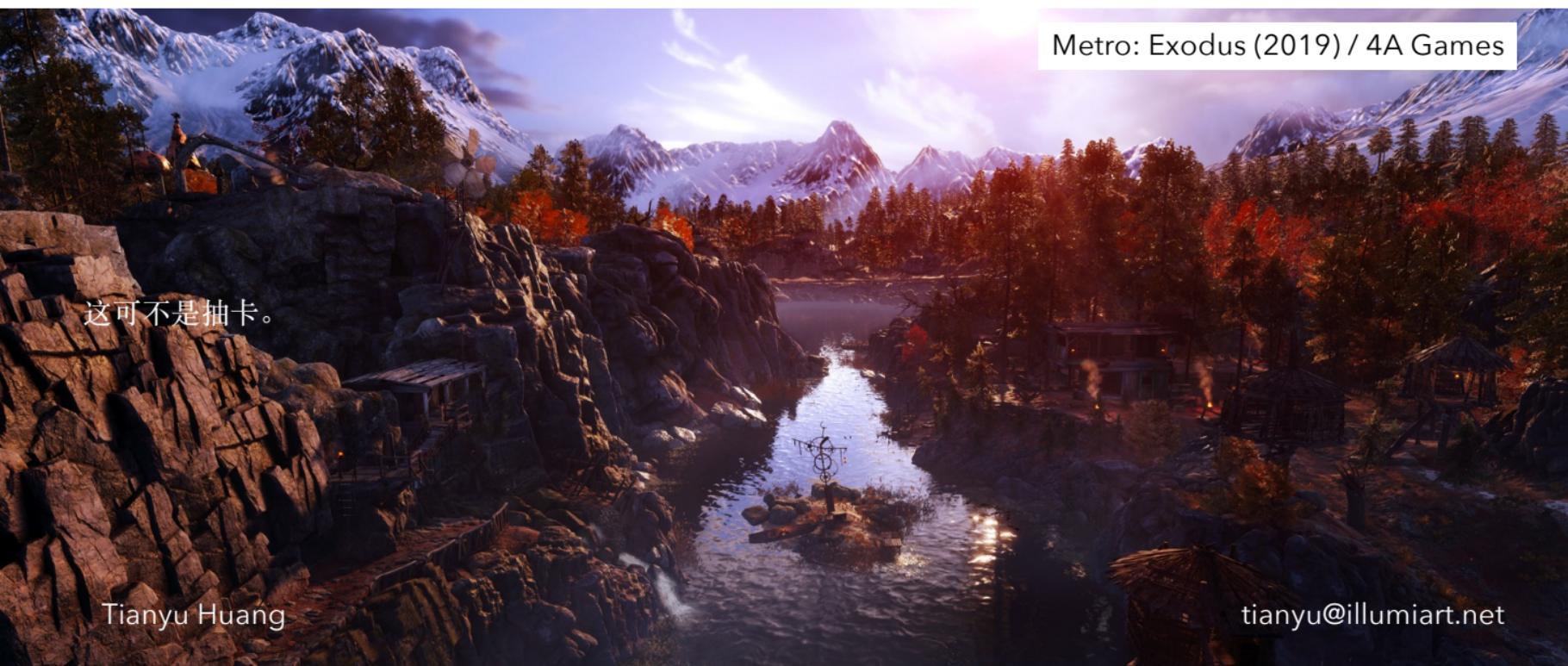
* 在材质模型部分介绍过这个概念，就是一个朝向各个方向看去，只要没有遮挡都呈现为纯白色的世界。

现代渲染框架内置实时AO实现

- 屏幕空间环境光遮蔽 (Screen Space Ambient Occlusion, SSAO)
- 基于视平线的环境光遮蔽 (Horizon-based Ambient Occlusion, HBAO)
- 体素加速环境光遮蔽 (Voxel-accelerated Ambient Occlusion, VXAO)

你不用关心具体是怎么实现的，因为现代渲染框架的相关实现都非常完善了，而且部分实时AO的具体实现目前并没有公开。

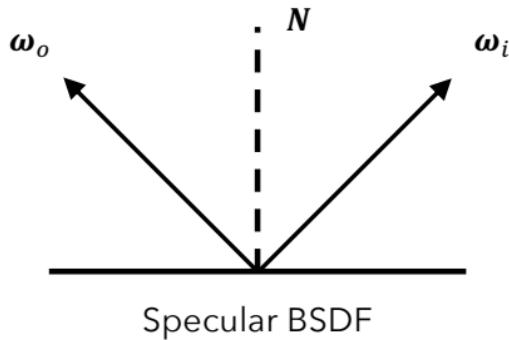
屏幕空间反射 (Screen Space Reflection, SSR)



镜面反射的光线追踪实现

不计算第一次光线命中点的颜色，而在镜面表面发射反射光线，并将反射光线得到的颜色作为最终颜色。

你可能会想到一种奇异的情况：第二次光线也命中镜面。在这里，我们不讨论这种情况。事实上这种情况用光线追踪算法很好解决，因为光线追踪是递归式的；但是屏幕空间反射确实不会处理这种情形。



起源引擎的水面反射

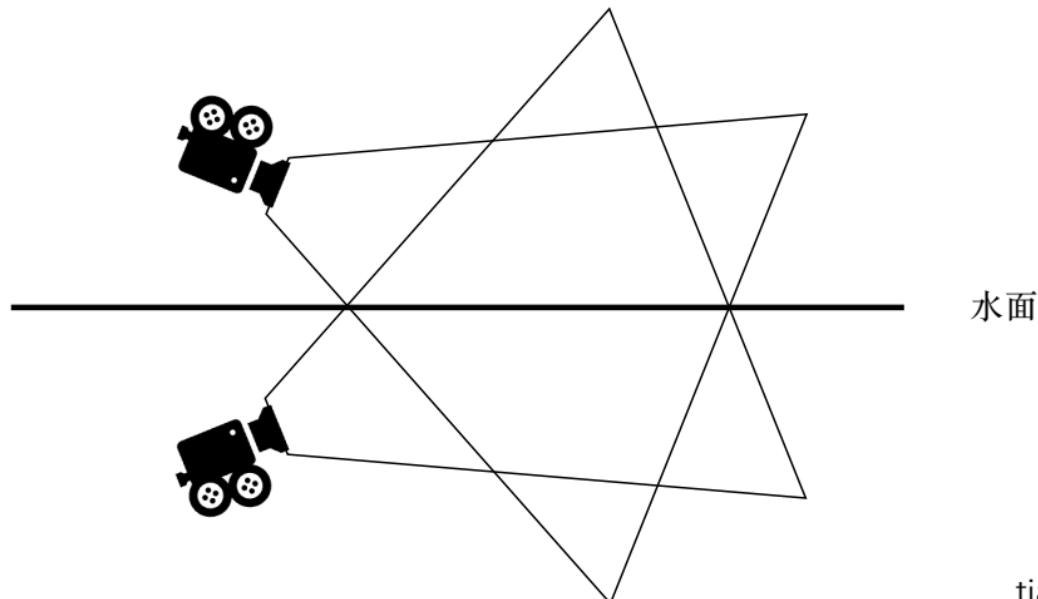
起源引擎的水面反射是一个非常特殊的实现，非常值得一提。



Half-life 2 (2004) / Valve Software

起源引擎的水面反射

起源引擎渲染水面时，在水面下方对称放置了一个虚拟摄像机。然后将虚拟摄像机的渲染结果投影到水面上。



起源引擎的水面反射

放置虚拟摄像机不仅仅可以处理水面反射，它也能处理虚拟门径的渲染。在Valve Software的Portal (2007)中，大量采用了类似的技术。



Portal (2007) / Valve Software

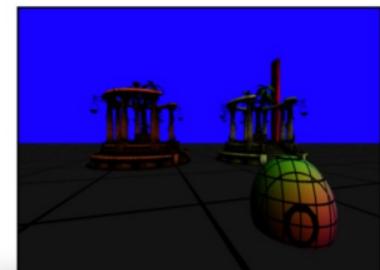
起源引擎的水面反射：优势和问题

优势：虚拟摄像机可以渲染主缓冲以外（无法直接通过主摄像机看见）的像素。

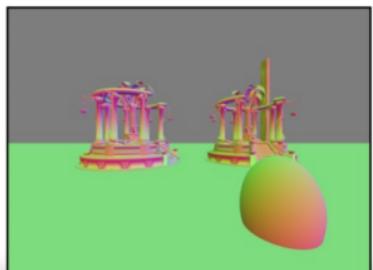
问题：虚拟摄像机一次只能处理一个平面。

- 如果有多个水面，那么要创建大量虚拟摄像机，性能会急剧下降。
- 无法处理曲面，因为曲面的法线是处处变化的。

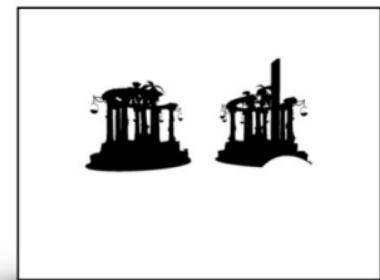
屏幕空间反射基本原理



Scene Color



Scene Normal



Reflection Mask



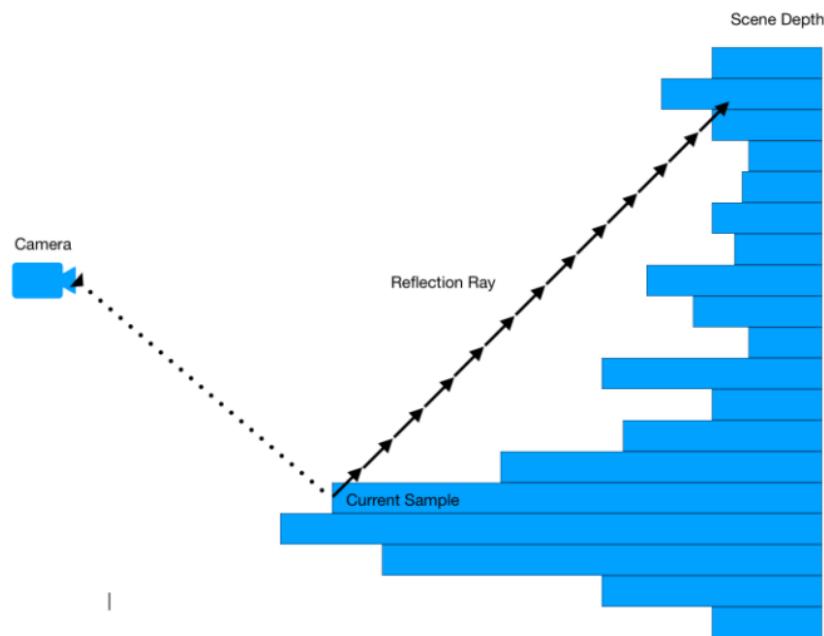
Scene Depth

SSR通过4个渲染通道的合成实现。

- Reflection Mask用于区分表面是否发生反射，对于发生反射的表面，执行SSR的后续步骤。
- 根据屏幕空间坐标和当前摄像机矩阵，我们可以还原世界空间中的入射光方向，进而通过法线通道计算出反射光的方向。
- 在深度通道上进行步进，计算反射光与场景中物体的交点。
- 在颜色通道中取出交点处的颜色，交点处的颜色就是反射到摄像机的颜色。

(Image credit: Sugu Lee)

屏幕空间反射基本原理



SSR通过4个渲染通道的合成实现。

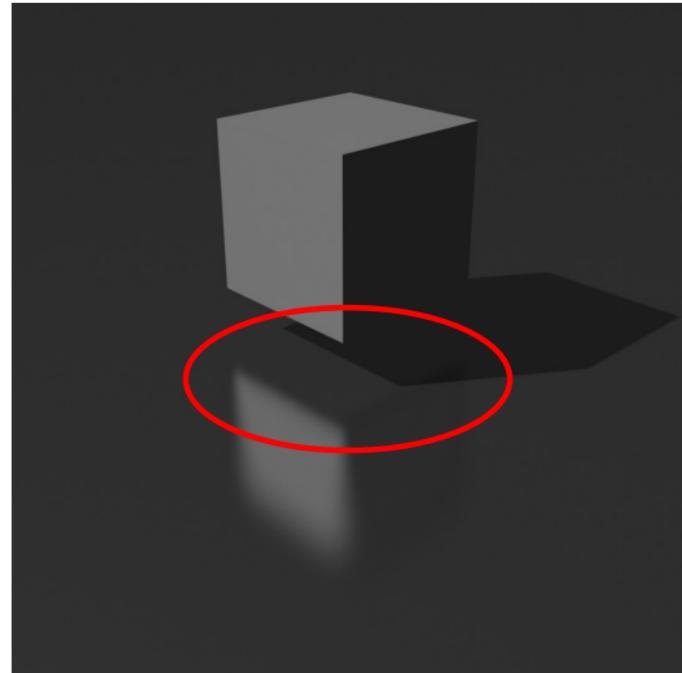
- Reflection Mask用于区分表面是否发生反射，对于发生反射的表面，执行SSR的后续步骤。
- 根据屏幕空间坐标和当前摄像机矩阵，我们可以还原世界空间中的入射光方向，进而通过法线通道计算出反射光的方向。
- 在深度通道上进行步进，计算反射光与场景中物体的交点。
- 在颜色通道中取出交点处的颜色，交点处的颜色就是反射到摄像机的颜色。

(Image credit: Sugu Lee)

屏幕空间反射：优势和问题

优势：只需要渲染一次场景颜色，因为反射的颜色也是从主缓冲中取得的。

劣势：主摄像机看不见的部分不会被反射出来。



Bloom

Armored Core VI: Fires of Rubicon (2023) / From Software

光污染可以转移人的注意力。

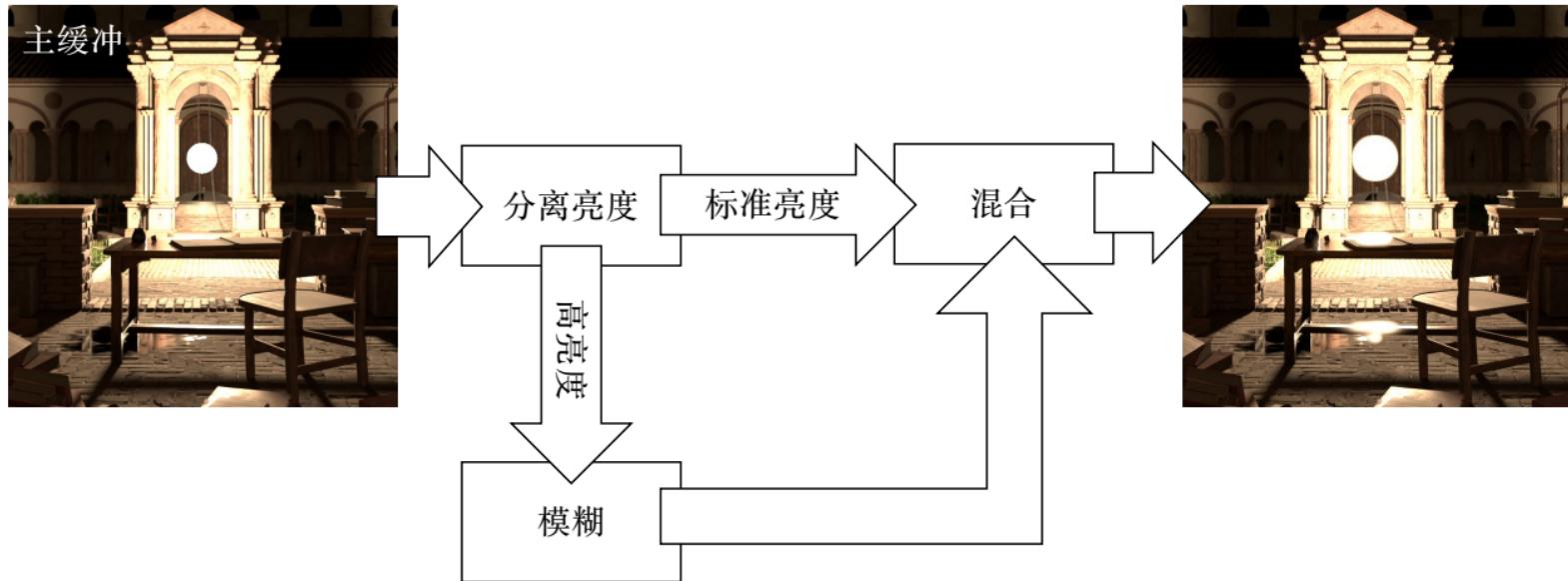
Bloom源自于对屏幕亮度的妥协

对于SDR屏幕，可以呈现的亮度范围仅有0.0~1.0（0~255），最大值通常被映射到100nits或者400nits（和用户对屏幕亮度的设置也有关系）。

对于HDR屏幕，HDR10的理论亮度极限是10000nits，不过大部分屏幕的亮度都在 10^3 nits量级。

如何表现出爆表的亮度？使用Bloom。

Bloom的一种实现思路

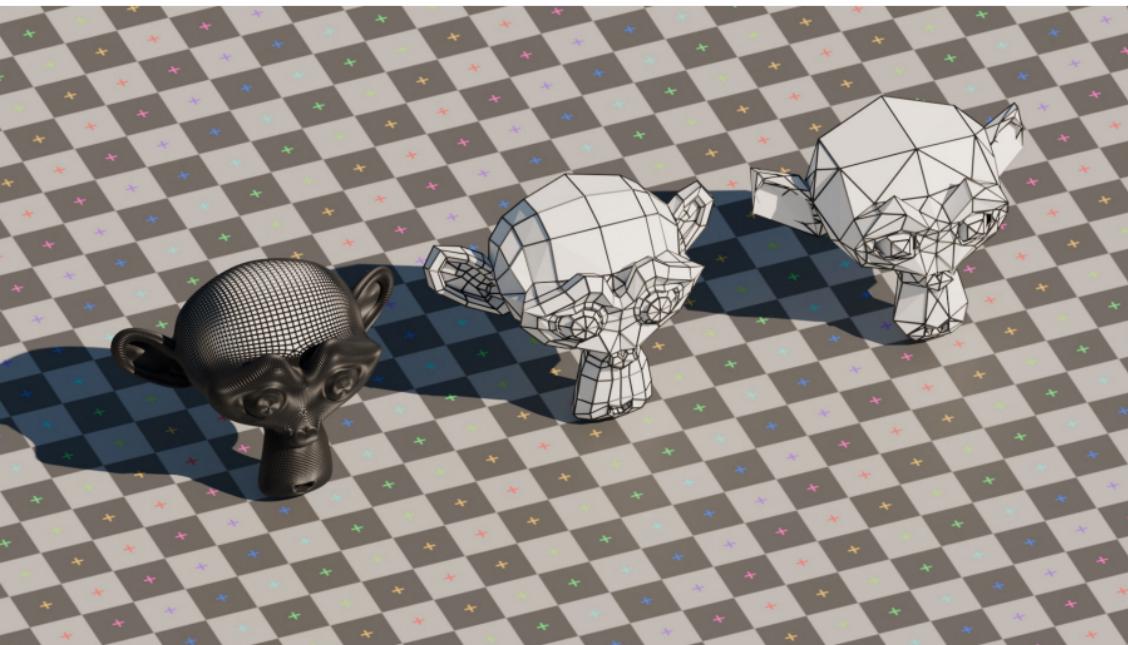


常见实时渲染优化策略



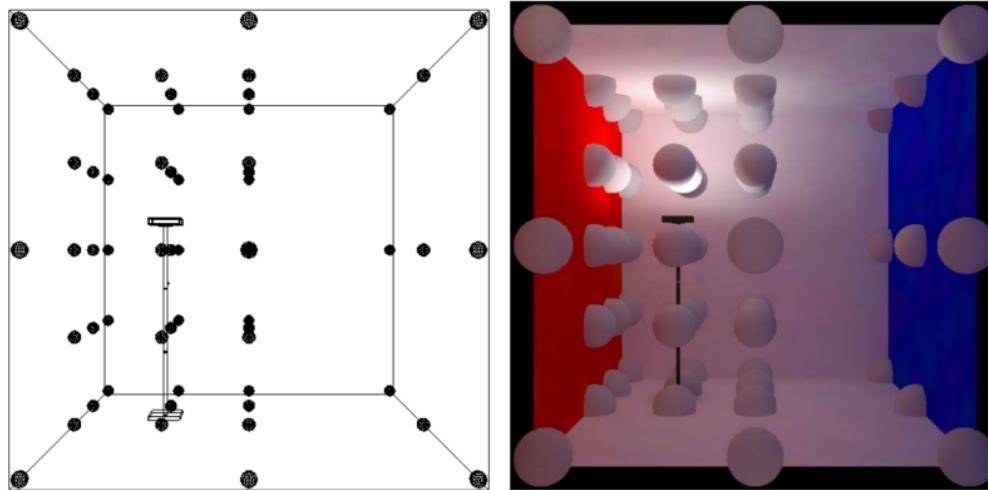
Hellblade: Senua's Sacrifice (2017) / Ninja Theory

细节层次 (Level of Details, LOD)



近处的物体使用高模，远处的物体使用低模。以此减少传入GPU的三角形面数，从而加快渲染速度。

辐亮度探针 (Probes)



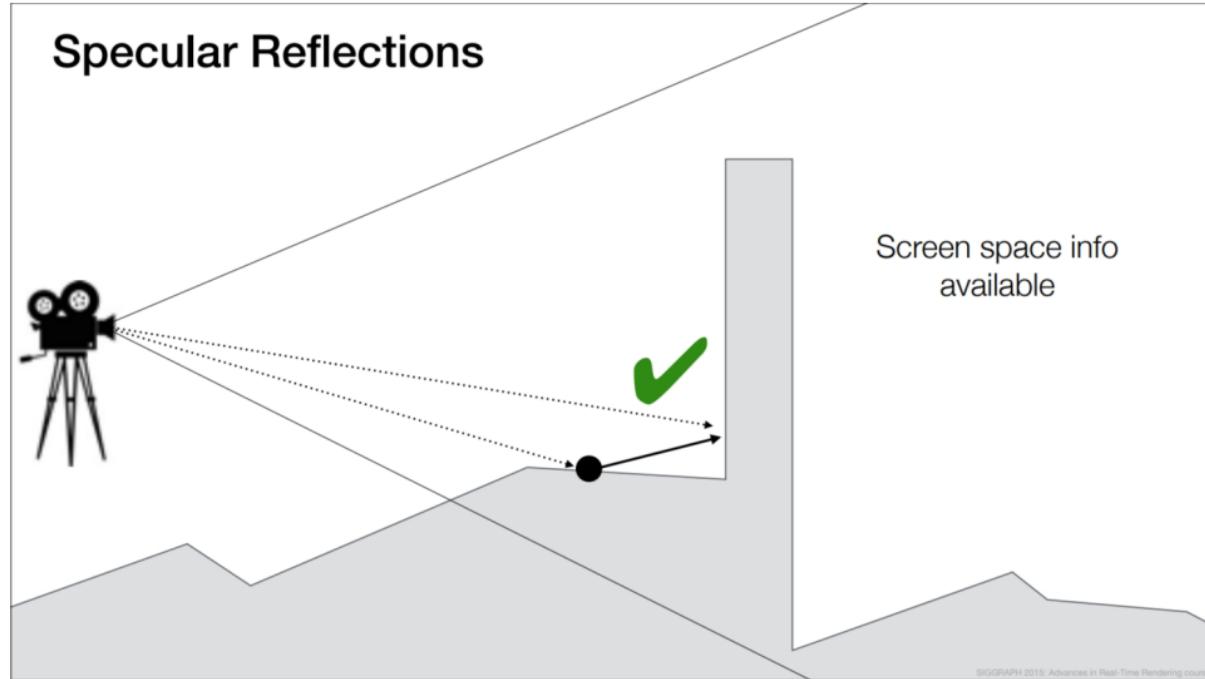
G. Greger, P. Shirley, P. M. Hubbard and D. P. Greenberg, "The irradiance volume," in IEEE Computer Graphics and Applications, vol. 18, no. 2, pp. 32-43, March-April 1998, doi: 10.1109/38.656788.

通过等距分布的探针将空间连续的辐亮度分布以离散的形式存储下来。进行查询时，取出最近的探针，然后进行插值即可。

这是一种在实时环境下实现全局光照的方法。

反射探针

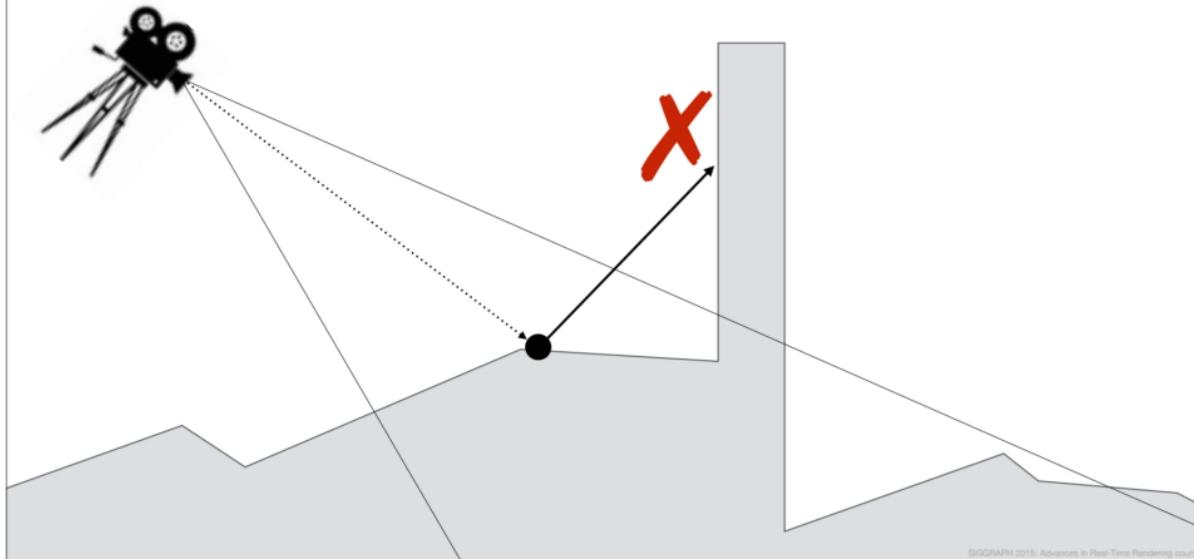
Specular Reflections



Multi-Scale Global
Illumination in Quantum
Break / Ari Silvennoinen
(Remedy), Ville Timonen
(Remedy)

反射探针

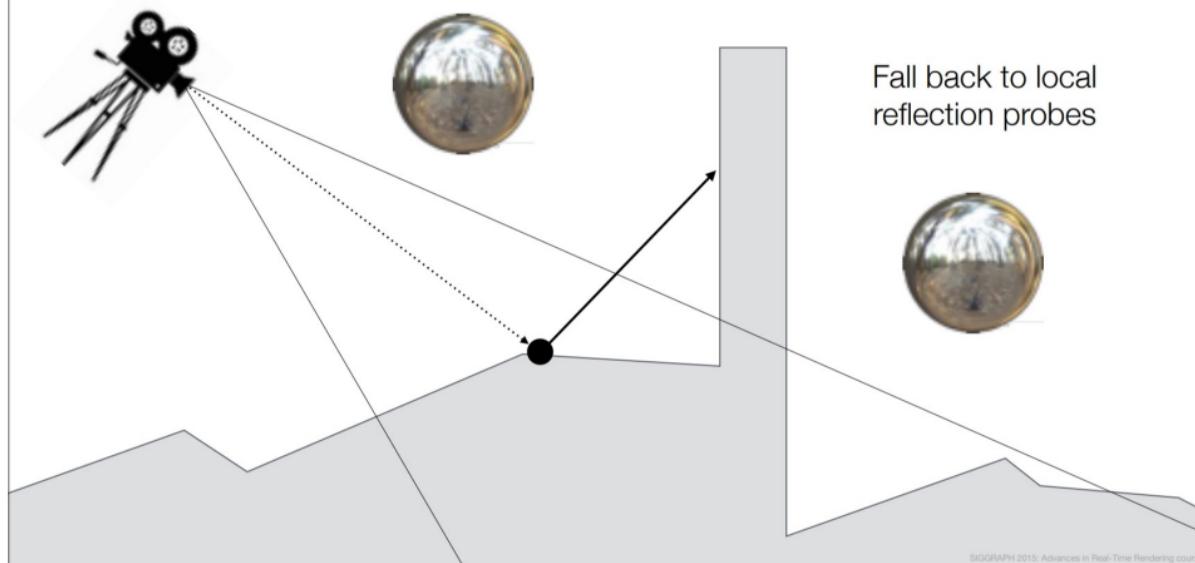
Specular Reflections



Multi-Scale Global
Illumination in Quantum
Break / Ari Silvennoinen
(Remedy), Ville Timonen
(Remedy)

反射探针

Specular Reflections



Multi-Scale Global
Illumination in Quantum
Break / Ari Silvennoinen
(Remedy), Ville Timonen
(Remedy)

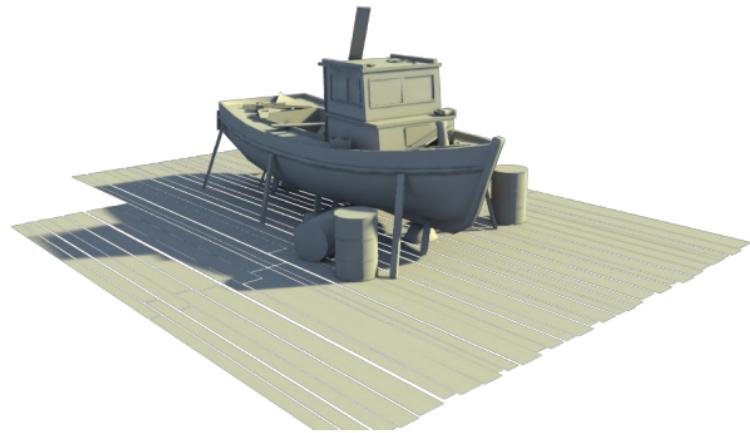
反射探针



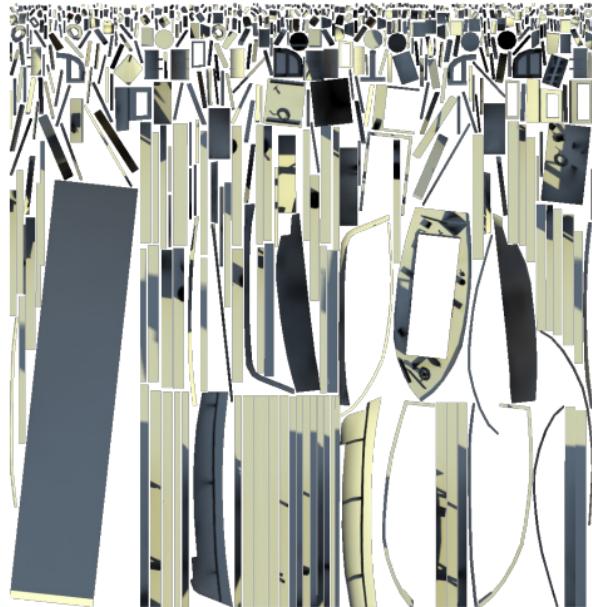
现代渲染引擎都提供了放置反射探针、进行反射烘焙和自动计算反射的工具。

Multi-Scale Global
Illumination in Quantum
Break / Ari Silvennoinen
(Remedy), Ville Timonen
(Remedy)

烘焙 (Baking)

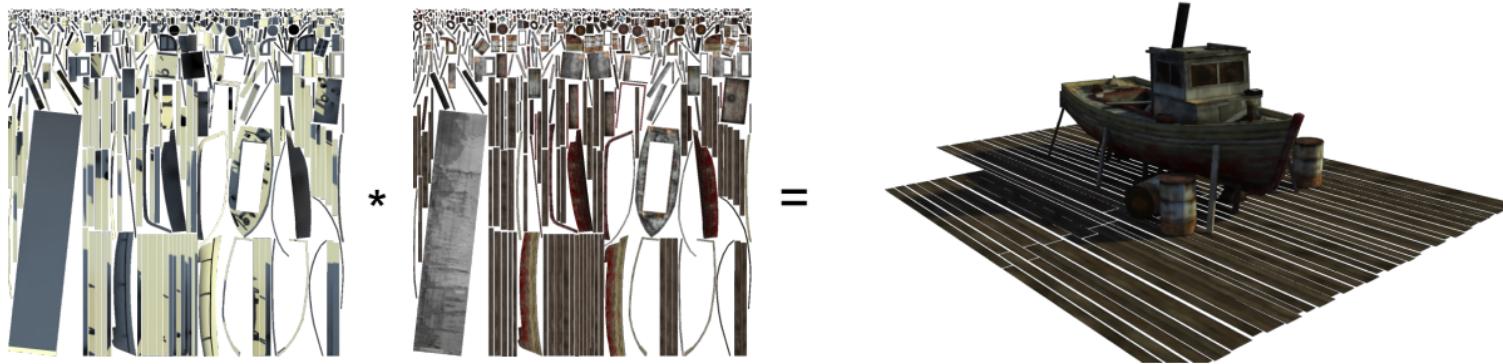


对于场景中的静态物体，其光照自然也是静态的。我们可以把高质量的光照预渲染到贴图上，这一过程称为“烘焙”。



Pre-computing Lighting in Games / David Larsson, Autodesk Inc.

烘焙 (Baking)



Pre-computing Lighting in Games / David Larsson, Autodesk Inc.

对于场景中的静态物体，其光照自然也是静态的。我们可以把高质量的光照预渲染到贴图上，这一过程称为“烘焙”。

结束语

《古剑奇谭3 (2018)》 / 网元圣唐

以一个“学院派”的身份，
聊聊国产单机的未来。