【CHAPTER 23】

Interactive Light Map and Irradiance

Volume Preview in Frostbite

*Diede Apers, Petter Edblom, Charles de Rousiers, and Sébastien Hillaire*

*Electronic Arts*

# 摘要

本章将介绍Frostbite引擎中的实时全局光照（GI）预览系统。该系统基于GPU上运行的蒙特卡罗光线追踪（Monte Carlo path tracing）方法，使用DirectX光追（DXR）API实现。本文提出了一种根据构成场景的元素实时更新光图和辐照度体积区域（irradiance volumes）的方法。同时，我们还将讨论如何提高像素的视图优先级排序（view prioritization）速度与辐照度换缓存（irradiance caching）更新速度的方法。除此之外，系统还使用了一套光图降噪方法（light map denoiser）来保证屏幕上呈现的图像一直具有较好的质量。这套解决方案能让艺术家渐进式地实时预览当前编辑的光照，而不用像之前使用CPU GI计算那样渲染一次要等上数分钟甚至数小时。虽然渐进预览过程中的图像品质可能历经数秒也未必能够收敛到成品质量，但是该质量用于辅助艺术家大致确定场景最终的光照效果已经完全够用。这种渐进式预览可以加快艺术家们迭代光照方案的速度，从而帮助他们更好更快地做出优秀的场景光照。

# 介绍

基于光图的预计算光照方案最早出现在1996年的游戏*Quake*中。从那时起，这类方法被不断提升，视觉上的拟真度已经达到了相当高的水平。然而，该方案直到现在还是存在烘焙时间漫长的问题，这种低效让艺术家和图形工程师在工作和debug时备受困扰。本文的目标，就是针对这个问题在Frostbite编辑器中提供一套实时的漫反射全局光照（diffuse GI）预览方案。

美国电艺（Electronic Arts）出品过种各样具有不同光照复杂度的游戏作品：静态光照为主的*星战：前线2* （Star Wars Battlefront 2），基于动态昼夜变换系统（time of day）的*极品飞车*（Need for Speed），甚至还有需要根据场景破坏情况和实时光照情况实时更新全局光照（GI）的作品，比如*战地*系列（Battlefield series）。在这一章中，我们主要讨论全局光照为静态时候的情形，这类情形也是Frostbite 全局光照解算器可以支持的。如图23-1所示，基于预烘焙光图和探针（probes）的静态全局光照方案具有较高的可行性：打开高质量间接光进行烘焙，可以得到视觉真实度非常高的渲染结果，又不需要很多运行时的计算量。反观动态全局光照，仅就可实时移动的动态光源这一种光照成分而言，就需要大量的离线烘焙。运行时的效果也只是一个较为粗糙的近似，同时还伴随着可观的计算量开销。

**图23-1.** 使用本文中提到的Frostbite 全局光照预览系统渲染得到的三个不同环境的截图。左图：Granary. (感谢 Evermotion.)中图: SciFi test scene. (感谢

Frostbite, . 2018 Electronic Arts Inc.) 右图: Plants vs. Zombies Garden Warfare 2 中的 Zen Peak关卡.( 感谢 Popcap Games, . 2018 Electronic Arts Inc.)

光图中每一个像素值的计算非常适合于并行化，它们的值在计算过程中不受彼此的影响，可以单独地被计算[3,9]。像素值可以通过光线追踪与蒙特卡洛积分计算得到[5]，计算过程中每一条路径同样可以单独被计算，互不干扰。最近在DXR[11]与Vulkan Ray Tracing [15]中新加入的与光线追踪相关的部分能够让用户更加方便的发挥GPU强大的并行运算的能力，从而将这种强大的运算力用于计算射线-三角形求交、表面估计、光线追踪回归与着色当中。Frostbite中的全局光照漫反射结算器就是基于DXR API的。

本章将讨论Frostbite光线追踪方案实现实时全局光照预览的方法。第23.2节讨论了用于生成光图与辐照度体积区域的光线追踪方案细节。23.3节讨论使用视图优先级与纹理元优先级的加速方法来减少全局光照预览时的计算开销。23.4节将讨论当艺术家与场景中的光、材质、模型交互时，全局光照信息的生成与更新时机。最后，23.5节将讨论不同的光照计算加速组件对预览准确性与性能的影响。

# 23.2 全局光照解算管线（GI SOLVER PIPELINE）

本节将讨论本文解决方案中用于预览光图与辐照度体积区域的全局光照解算器。23.2.1小节将讨论全局光照解算器的输入（场景几何模型、材质和灯光）与输出（光图数据），以及他们是如何在GPU中存储的。随后，23.2.2小节讨论管线中每一部分的概况。最后，在23.2.3中，我们将讨论方案中的光照计算是如何进行的。

379

## 23.2.1 INPUT AND OUTPUT

### 23.2.1.1 INPUT

**几何数据**：场景几何数据使用三角形网格模型表示。每一个网格模型带着一个特殊的UV参数，以便将光图映射到网格模型上。为了与最终运行时的网格模型（in-game meshes）加以区分，这些网格模型在本文中被称为“代理网格模型”（proxy meshes）。代理网格模型一般经由运行时网格模型减面得到，如图23-2所示。预览用的光图受内存占用限制，一般像素密度会很小。使用代理网格模型可以减轻由于光图像素密度过于小而产生的自相交问题。（self-intersection issues）代理模型上的UV参数可以通过专门的算法自动生成，或者由艺术家手动指定。无论用哪种方法，目的都是为了让生成的UV尽量避免光图素映射时产生的拉伸和产生带有像素重叠的模型（漏光）。这些非流形网格模型（Non-manifold meshes）的UV会被划分成若干块，每个块之间留有一定距离以防止线性采样光图时块与块之间离太近产生的漏光。一个模型的多个实例（instances）会共用同一个UV参数，并将模型映射到光图中的不同区域。如果模型实例的体积被放大了，那么它覆盖的光图区域面积也会相应变大。这样可以保持场景中所有几何物体映射光图使用的像素密度一致。当然，艺术家可以选择禁用这个映射面积缩放，从而节省实例模型占用光图空间的大小。

**图23-2.** *星战：前线2* （Star Wars Battlefront 2）中使用的光图。左图：使用代理网格模型映射的光图，使用了全局光照追踪。右图：将光图通过代理网格模型的UV投射到最终运行时的网格模型上后映射的光图，同时场景还有法线贴图。(感谢 DICE, . 2018 Electronic Arts Inc.)

**材质：**每一个场景几何模型都有若干个材质属性：漫反射率（diffuse albedo），自发光颜色（emissive color），以及一个“背面射线检测行为描述”（backface behavior）（译者注：指的是当入射光线与该三角面背面求交成功时要做的事情，例如光线完全被遮挡或是穿过三角面继续进行光追）。漫反射率用于确定物体间最基本的漫反射，三角面的朝向会影响光线与三角面背面求交成功时的处理方法。由于我们只关注物体的漫反射光照，因此物体的材质只需要一个简单的Lambert漫反射光照模型[10]即可。高金属度的物体会被当做非金属处理。基于这种简化处理方法，漫反射率的数值可以基于物体表面的“反射率”（reflectance）与粗糙度决定[7]。也因此，本文中的全局光照解算器不会解算与焦散（caustics）相关的内容。

**光源信息：**一个场景中会包含多种光源：局部点光源，面积光，方向光与天空光（sky dome）[7]。每一种光产生的效果要与他们在实时光照中的效果类似，来保证预览烘焙的结果与运行时的效果具有高度的一致性。天空光的信息存在一张分辨率较小的cubemap中。

**辐照度体积区域**：除了光图之外，全局光照解算器还可以对动态物体的光照进行可视化预览。这些动态物体的光照来源于布置在场景中的辐照度体积区域，如图23-3（a）所示。每一个辐照度体积区域存储的是一个布满了球谐函数系数（spherical harmonics coefficients）的三维网格。

【图】

1. **辐照度体积区域的可视化结果 （b）光线求交的可视化结果**

图23-3， 编辑器中的debug可视化。(a) 一个未来风格走廊场景中的辐照度体积区域，该体积区域用于对动态物体产生照明。(b)透明物体与阴影射线求交的可视化结果。黄线代表阴影射线，红色叉子表示计算物体透射时使用的“求交着色器程序”（*any-hit shader*）的调用次数。

场景的几何体会做一个预处理，来为光图图素（light map texel）产生一种叫“采样位置”（sample locations）的东西。这些处在世界空间中的点遍布于场景模型之中，用作光线追踪时物体表面发出的追踪射线的起始点。每一个样本点的位置均落在每一个光图图素所覆盖的区域范围之内以及图素的边界上。这些样本点会与没有展过UV的场景几何体求交，然后再使用求交物体的空间变换信息转换到世界空间，如图23-4所示。与模型求交失败的点会被丢弃掉不做运算，求交成功的传至GPU参与到稍后进行的光线追踪计算过程中。采样位置产生的算法采用贪心思路，此算法会在图素内一直产生样本直到一个图素涵盖的范围内包含8个（左右）采样位置为止。计算采样位置时，算法利用低偏差霍顿序列（low-discrepancy Halton sequence）在代理模型的UV空间内撒点采样。在某些情况下，撒点无法产生有效的样本，例如一个处在撒点之间面积非常小的三角。遇到这种情况，算法会将这个小三角形剪裁（clipping）并归属到边界离它最近的那个图素里，然后在三角新的几何中心位置生成一个采样位置。另外，使用这种算法还会遇到一个样本点在UV空间内产生多个采样位置的情况。这类情况一般发生在若干个几何体共用一块光图UV空间的情况。我们的算法对于这种不希望出现的情况会进行弹性处理，接受这种情况的出现。

**图23-4.** 左图：三维空间中的一个几何体。 中图：这个几何体在UV空间中的映射。 采样位置基于一个覆盖一整个图素的低差异样本序列产生。右图：只有与三角形相交（译者注：指在UV空间内落在三角形UV内部）的样本会被保留作为可用的样本点。

场景中的几何体存储在一个双层层次包围盒（bounding volume hierarchy(BVH)）中(DXR’的加速结构 [11])。底层BVH节点上存储每一个不同的网格，顶层BVH会生成每个底层BVH节点中网格的实例，每个实例在空间中处于不同位置。虽然遍历这种双层BVH结构的速度不如单层的BVH快，但是这种双层结构有利于加速在编辑场景时候频繁发生的场景几何体更新。举个例子：当一个几何物体在场景中被平移时，这种BVH只需要更新顶层那一级的BVH中对应这个模型那一个节点的数据即可，而不需要把这个模型的数据在底层BVH种进行移动。

### 23.2.1.2 OUTPUT

全局光照解算器会产生几种输出，这些输出会存储在光图或者辐照度体积区域内。（几何）实例会在运行时使用较为粗略的方法快速排布到一张或者二几张光图图集(light map atlases)中。

**辐照度：**这部分数据是全局光照解算器生成的主要输出。这部分数据主要表示了光图或者辐照度体积区域中的“有向辐照度”（directional irradiance）。 一般来说，游戏运行时使用的模型会比用来做烘焙的模型更精致一些，同时还利用诸如法线贴图（normal mapping）的技术承载更多的法线细节信息。这类法线细节不会参与到烘焙之中，如图23-2所示。在运行时中，有向辐照度允许我们计算得到细节中法线方向上的入射辐照度。辐照度的朝向表示方法有几种，例如平均值、主成分方向（principal direction,）以及球谐函数（spherical harmonics）[9]等，都可以支持。

备注------------------------

1.将几何体打成图集时，每一块子元素之间加入一些空间避免世界空间中不相接的两个几何体在两个图素之间插值。

2.有向辐照度存储入射光的方法允许从多个方向上采样。

**天空可见度：**这部分数据记录了光图中每个图素和辐照度体积区域内某一点的天光可见度[7]。天空可见度的数值会参与到多个运行时计算当中，例如反射混合与材质的特效等你。

**环境光遮蔽**：这个数据记录了光图图素和辐照度体积区域内某一点的环境遮蔽情况[7]。环境光遮蔽数值会在运行时参与到反射遮蔽的计算中。

## 23.2.2 GI SOLVER PIPELINE OVERVIEW

本文中的全局观照解算管线（简称解算管线，下同）管线的设计目标是让预览光照的速度尽可能的快。对于真实游戏产品中出现的关卡场景规模，若要等到光照计算的结果完全收敛，可能需要数秒甚至数分钟。因此，本文中的解算管线使用几个迭代步骤，逐渐精细化光照结果。如图图23-5所示，每一个迭代步骤会完成如下操作：

- 更新场景（Update scene）：上一次迭代结束后有关于场景的所有改动会被纳入更新的范畴，例如模型的位移或者更改灯光的颜色等。相关改动会被推动到GPU上。这些内容将在23.4节中讨论。

- 更新缓存（update cache）：若辐照度缓存（irradiance caches）无效或者还未计算完毕，使用光线追踪计算缓存中入射光的辐照度。这些缓存会用于加速管线中光线追踪的计算。这些内容将在23.3.3节中讨论。

- 规划待计算图素（Schedule texels）：依据当前预览场景相机的视锥，找出最有可能被观察到的光图图素以及辐照度体积区域，并将它们规划为“待计算图素”。这些内容将在23.3.1节中讨论。

-光追图素（Trace texels）：对每一个待计算的图素和辐照度采样点使用若干条光路（path）进行光追，收敛光追结果。这些光路可以用来计算入射光辐照度，也可以用来计算天光可见度以及环境光遮蔽。这些内容将在23.2.3节中讨论。

- 合并图素：将新的计算结果累加到持久化的输出里。这些内容将在23.2.6节中讨论。

- 输出结果后处理：对输出结果进行边缘扩张（Dilation）（译者注：指的是将贴图中UV边界的像素向UV边界外复制的操作。这种操作可以保证UV边界上的映射不出现“漏光”）与去噪点，为用户呈现无噪点的输出。这些内容将在23.2.8节中讨论。

**图：**

图23-5：全局光照解算器管线总览。光图与辐照度体积区域的更新采用迭代式的方法。场景预览相机的视点会被用于确定光图图素中那些图素需要进行光追计算（待计算图素）。辐照度缓存可用于精细化全局光照数据。光追的计算结果会与前面几帧的计算结果合并在一起，然后再进行后处理（边缘扩张与去噪点），最后返回给运行时。

## 23.2.3 LIGHTING INTEGRATION AND PATH CONSTRUCTION

欲计算每个光图图素的辐照度E（irradiance），我们需要对（光照表面的）法线上半球Ω内的入射光的光强度（radiance）使用投影立体角的值*ω*⊥（projected solid angle）做带权积分：

【公式1】

计算入射光强度L，需要解光传播方程（light transport equation）。该方程根据入射光强度Li以及光受着色表面材质属性的影响计算得出出射光光照强度。在本方案中，我们只关注漫反射的部分。对于漫反射材质，材质的漫反射率ρ（albedo），自发光光照强度*Le*，有如下方程：

【公式2】

由于公式1的积分对象是高维的（相对于一维来说），因此这个方程的求解很困难。欲求解这类问题，蒙特卡洛（Monte Carlo）积分法这类基于统计学的方法是不错的选择。原因有几点：第一，结果无偏（unbiased）。这意味着只要采样次数够多，这类方法最终一定会收敛到一个固定的真值上。第二，最终结果可以使用迭代的方法计算得到。这个特点非常适合于渐进精化的预览的使用情形。最后，每个光图图素的渐进式精化过程的计算是各自独立的，因此非常适合并行运算。使用蒙特卡洛积分求解公式1，有：

【公式3】

简单来说，上述公式表达的含义为：对积分域内n个随机位置所对应的值使用概率分布函数pL进行加权平均，所得结果会收敛于真值。这一性质在计算时非常方便。

欲求解方程1，一种简单的解法是构建若干条由光图图素上的点到光源的光路，然后用它们做光追。这些点均依附在几何体表面上。几何体材质中的属性，例如漫反射率等，会减弱对应光路的能量通量（throughput.），该能量通量决定了光追过程中光的强度。做光追时，我们迭代式的构建从图素到光源的光路，如表23-1中的核心代码所示：

表23-1 对入射光积分时使用的核心代码。

表23-1中的算法大致说明了计算每个光图图素辐照度的方法。该方法的计算速度较慢且扩展性不佳。接下来，我们将讨论如何使用传统方法来提升计算速度。

- 重要性采样（Importance sampling）：使用光照法线上半球内的投影值作为权重进行重要性采样，会比使用均匀分布样本进行采样要更高效。从掠角（grazing angle）方向（译者注：与光照表面几乎水平的方向）入射的光在光照贡献上远远小于垂直入射的光[10]。

- 使用随机数构建光路：为了减小辐照度计算时每次采样间的方差，每条路径最开始的两个路径点需要被精细选择。首先，使用23.2.1节中提到的低差异Halton样本序列预生成样本位置（spatial sample locations）。这一步保证了采样域中样本的位置是均匀分布的。第二步，再使用低差异Halton序列生成样本的方向（directional samples）。为了避免空间内相邻的样本本方向过于一致（correlation issues），产生的样本会使用随机抖动进行方向上的扰动。这种构造样本的方法保证了样本在四维域（译者注：即样本的位置坐标，与样本朝向的角度）中的随机性。直接使用一个四维序列生成样本比使用两个二维序列更加高效一些，然而出于简便性，我们没有在第一个实现版本中使用四维序列。其他剩余的光路的路径点的方向使用均匀分布的数值决定，点的位置由射线求交结果决定。

- 后续预估（*Next event estimation*）：构建光路时，随意选择一个方向直到光路通向光源这种做法是非常低效的。当光源数量减小体积大小变小时，光路通向光源的概率会大大降低。考虑极端情况，当场景仅仅被局部点光源照亮的时候，使用随机方向（构建光路）几乎不可能让光路抵达光源。一种解决该问题的简单的方法，是将构成一条光路路径中的每一个点与光源连接起来并计算它们的对光路的贡献。这种方法被称之为后续预估。使用这种方法，可以使用已经构造好的子路径来构建更多的光路。这种方法简单且高效的构造方法极大的提高了构建路径的收敛速度。为了避免重复统计子路径的贡献值，光源的几何体不被算作场景中常规的几何体。

上面所有的方法在表23-2中的简化核心代码中有所体现：

**表23-2** 描述光照的核心代码

23.2.24 LIGHT SOURCES

一个场景会包含多个点光源及面光源。当光路被构造好以后，这条光路折线的每一个点回去计算局部光源、方向光（例如太阳）以及天光对这条光路的贡献（后续预估）：

- 局部点光源：这种光照的辐照度计算很简单。辐照度的强度由光源到着色表面的距离以及光线入射的角度决定[7]。点光源的强度是根据距离平方的倒数衰减的，艺术家也可以通过调整光源的包围盒（bounding volume）来调整光照的影响范围。当着色表面不断接近光源时，光照强度数值会变得无限大。为了避免这种情况，我们将光与着色表面间的最小距离设定为1厘米。

**- 面光源（Area Lights）：**面光源的辐照度是对面光源可见的表面积分得到的。为了计算该值，我们在面光源的表面的可见部分取很多的点[10]然后把它们与光路中的点相连。面光源光照需要采样较多的样本点来求解。这一计算，不需要计算辐照度贡献，还计算了用于产生软阴影的采样点可见度（visiblility）。对面光源采样生成光路时，路径上的点会根据光源对该点照明的立体角大小在面光源上采等比例数量的样本点。为了更好地估算样本点对积分的贡献值，样本点需要在光照积分域内平均分布，如图23-6所示。面光源之于着色点的立体角在采样时是已知的，因此光源上的采样点可以用数量一定的低差异 Hammersley 序列（low- discrepancy Hammersley sequence）产生。为了防止样本方向重合从而导致阴影被重复计算（shadow repllicates），路径上每一个点对光源产生样本时所用的低差异序列都会加入一个随机偏移用于去除样本点在方向上的相关性。

**图 23-6** 光图图素辐照度计算过程。欲计算辐照度，需要构造一条光路（绿色）。光路路径上的每一个点使用向面光源投射射线的方法计算该光源直接光光照贡献（这一过程就是后续预估）。路径点采样光源的样本点数与光源之于路径点的立体角大小成正比。射线没有与任何光源或者几何体相交，则会使用天光的强度进行计算。

**- 方向光（Directional Lights）：方向光的**辐照度在光路路径中的每一个反射点上被采样。运行时的方向光是以小圆盘形状的面光源形式出现并计算的（译者注：这种做法是考虑到太阳在积分域内是有立体角的，而并不是一个向量）。然而，因为光图分辨很低，因此计算光图时没必要使用这种（更精确的）方法。

- **天光（sky dome）**：当光路中的点投射线没有与场景中的任何几何体相交时候，就会让这些射线去计算天光贡献的辐照度。可以使用[18]中的“别名算法”（Alias Method，译者注：一种用于离散分布随机取样的方法）加速光路中每一个点对天空的重要性采样。

23.2.5 SPECIAL MATERIALS

除了通常的漫反射以外，材质自己还可以产生自发光，或是让光线穿透自己（透明材质）：

- **自发光表面（Emissive surfaces）**：具有自发光属性的几何体实例会产生光照，这让一般的几何物体都变成了潜在光源。在构建光路时，物体表面的自发光是在光路中的每一个转折点被计算的。这种方法虽然可以产生正确的结果，但是需要较多的采样，尤其是那些发光面积较小的表面。为了解决这个问题，带有自发光的三角形会被纳入一个光照加速结构，详细请见23.3.2节。这些自发光三角形会被当做一般的直接光进行计算。

**- 半透明物体（Translucency）**：物体材质使用“背面特性”（backface behavior）来描述材质表面的透明属性。对于这类表面，光会以散射形式穿透到几何体的另外一边，如图23-7所示。光的透过量是由物体表面的漫反射率与透光率（translucency factor）决定。基于透光量，我们可以根据透光的概率概率控制入射光照表面的光线应该穿透物体还是应该反射。直接光计算只在反射光的那面进行，因此在计算直接光时，如果光源与构建的光路之间有半透物体，光是不会传递到照明表面的。这条光路上的点只会在这条路径能够与半透物体表面相连时候才会被着色。

**图23-7.** 带有透明平面的场景。左图：半透明被禁用，物体不会对光进行散射。右图：半透明被启用，平面允许光散射到场景内。半透散射让场景笼罩了一层红色。

- 透明物体**（Transparency）**：后续估计阶段（参见23.2.3节）会使用一条通往光源的射线进行计算。光线与带有透明度的物体相交会影响光线的可见度。使用 DXR， 这一效果可以在求交着色器（any-hit shader）中将透射度（transmittance）乘以一个可见度来完成。材质中不包含透明属性的几何体会被标记为 D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_OPAQUE。 当光线与这类物体相交时，光线传播会被终止。若物体材质包含半透属性，则会被标记为D3D12\_RAYTRACING\_GEOMETRY\_FLAG\_NO\_DUPLICATE\_ANYHIT\_INVOCATION来防止二次贡献光照计算。图23-3(b)说明了求交着色器是如何只在透明物体上被触发调用的。

23.2.6 SCHEDULING TEXELS

本节将讨论管线中的第一部分与第三部分，如图23-5所示。在进行光追计算之前，系统需要将待计算的光图图素进行规划。规划这一步工作会运行多次“启发式算法”来决定哪些图素需要被计算。这些启发式算法包括“视图优先级法”（参见23.3.1节）和“质量收敛剔除法”（convergence culling）（参见23-4节）。质量收敛剔除法会根据图素已经被计算的次数以及是否已经收敛来避免已收敛的图素被再次被计算（译者注：质量收敛意味着monte carlo积分中的样本数已经足够多的，积分值已经趋近稳定，再多的样本不会对计算结果产生大的影响）。当以个图素被规划为待渲染，我们会从这个图素对应的几何表面选一个样本，并将其加入到一个缓冲区（buffer）中。

当所有需要被计算的样本都已被加入到缓存后，管线中的第二部分会开始处理它们。依据系统性能不同，每一个样本会被计算若干次（参见23.2.7节）。图23-8描述了图素计算时的调度策略。为了将计算分摊到尽可能多的并行线程，充分使用硬件资源，我们会将对一个图素安排多次运算，ns = 图素对应的样本数。它们的计算结果会存到一个按照图素进行划分的大缓存里， nt = 图素的数量。除此之外，在每一条运算线程里还有一个循环，用于从一个线程里面发射多条射线做运算，ni = 运算迭代数。总的采样数为*n*t × *n*s × *n*i。对于值nt，该值是由视图优先级产生的像素数决定的，1次采样对应屏幕空间的16个像素。*n*t 与 *n*s会被表23-3中的sampleRatio缩放。

**图23-8。**左图：光追核心的调度策略。 右图：采样样本被存在一个大的缓冲区内，光图上的每一个图素由这些样本的采样结果累加或者合并得到。

将多个样本计算结果合并成一个图素的操作在计算着色器（compute shader）内完成。注意，我们不需要担心一个图素被多次规划并运算这种情况。因为视图优先级算法保证了每一个图素只被规划渲染一次。最后的输出会用当前运算的结果与前一帧运算的记过合并得到。

23.2.7 PERFORMANCE BUDGETING

光线追踪的性能开销是不好预测的，这有可能造成瞬时的卡顿。为了保证艺术家编辑场景时候的流畅性，我们实现了一套性能预算系统（performance budget system）监控GPU进行光线追踪的时间。这个系统会动态调整光追时候的样本数来保证运算性能符合性能指标预算（以毫秒记）。详见表23-3。

**表23-3.** 性能预算系统控制样本数的代码

23.2.8 POST-PROCESS

全局光照计算产生的输出是渐进式生成的。因此，光照的最终结果往往需要几秒钟才能算出来。为了给艺术家提供有效的预览结果以及可控性，当前输出的结果需要尽可能的靠近最终输出的效果。在这方面，有三个问题需要被解决：

- 图素为黑：当一个图素还没有进行过运算，或者这个图素的样本发出射线与某个三角面的背面相交，这个时候图素会变为黑色，如图23-9所示。为了避免这两种情况，输出的光图会进行一次颜色溢出操作（dilation filter）。该操作保证所有的图素在运行时线性UV采样都不会出现黑色。某些图素会的局部会被覆盖，例如某些图素中的采样点被挡住没有接收到任何光照。这类图素不会参与到颜色溢出操作当中，因为它们的辐照度值只会根据没有被遮挡的样本做运算。这样做可以避免几何体连接处变黑。

**图23-9** 使用了颜色溢出操作烘焙的 Cornell Box 场景。左图：几何体覆盖了部分像素产生的错误辐照度，其原因是因为部分光追射线与几何体的内部产生了相交。这些图素会被被标记为“错误”。中图：使用颜色溢出操作处理过的错误图素。右侧：从上方观察一个物体所的合法图素（绿色）与错误图素（红色）。

- 图素噪点（noisy texels）：使用概率方法进行数值积分时，经常会出现欠采样（underdsampling）现象。噪点会随着时间推进减弱，因为后续增加的采样会让积分的平均值逐渐逼近于期望值。为了想用户呈现更理想的数值积分结果，我们采用了一种预测收敛期望值的降噪算法对积分结果进行处理，如图23-10所示。我们采用了[12]中提到的“方差引导滤波器”方法，核心思想是追踪图素的方差，并使用方差值控制这个像素进行邻域过滤处理（neighborhood filter）时候的强度。这一滤波处理作用在整个光图上，并使用几何物体UV分块的ID辨识图光图中的UV边界，不处理几何体UV之外的图素，如图23-11所示。这一滤波器是分层的，它会稀疏地扫描图素周围的像素，随着迭代次数增加，扫描的半径也会增加。即便噪声出现在多个不同的频率上，这种方法也能求得图素的收敛均值（译者注：因为扫描版半径随迭代次数变大）。由于这个滤波器是作用在光图图素空间上的，因此图素的滤波器扫描场景时候覆盖到的像素密度是较为恒定的。若场景的光图图素密度变大了，那么滤波器处理单个像素的迭代次数也需要相应增加。当图素与邻域的数值方差下降以后，滤波器扫描的半径会缩小，扫描时使用的样本间距离也会变小，使得最终过滤结果平滑地收敛到实际的均值。

**图23-10.** 左图：进行过合图与颜色溢出操作后的光图可视化结果。右图：光图数据经过去噪之后的样子。两幅图对光图的采样都禁用了线性插值采样，以便于更好的展示去噪的效果。

**图23-11**。光图空间中“á-Trous denoiser” [12]方法的三个步骤。每个步骤使用一个5x5的核进行过滤（绿色、浅蓝色、蓝色的部分）。一开始聚在一起的采样点会逐渐远离图素的中心，使得不同频率的噪点可以被过滤。滤波会根据图素自身的亮度以及几何体UV块ID跳过某些像素，以防止过模糊（overblurring）和几何物体UV边缘的漏光。在下排右图中的图素亮度直方图中，图素2（绿点）是正在被过滤的图素，只有图素4（深蓝色点）会被纳入滤波考虑的样本点内。其他的像素或者UV ID与当前操作的图素不一样，亦或者与当前操作的图素亮度差异太大。（译者注：图素0的UV ID与图素2不一样， 图素3的亮度没有落入滤波范围之内所以也不被考虑）

每个像素的均值方差3使用Welford在线方差计算方法(Welford’s online variance algorithm)[19]计算得到。方差数值不是在每次滤波操作时更新，而是在图素做光追用的样本块（bucket）都被计算完之后。因为蒙特卡洛积分的收敛速度是二次方的，因此样本块的大小会随着每次滤波操作而倍增。样本块在起始时的大小只包含12个样本，在后续的操作处理中每次增加一倍。使用方差信息，我们可以通过标准差（standard error）[20]了解当前处理的图素数值是否已经达到了置信区间（confidence interval）。我们认为置信区间为95%的时候可以认为图素已经完全收敛了。表23-4中展示了这个处理的样例代码。

- 接缝问题（Chart seams）：光图里的UV块之间会产生接缝问题。当两个图素在世界空间中对应的几何体相连接但是图素自己在贴图中不相连时，就会产生这个问题。这个问题是光图经常出现的问题。我们的GPU全局光照解算器工具目前没有处理这个问题，而是使用我们已有的CPU接缝修补方案[4]处理这个问题。

表 23-4 计算方差的代码示例：

# 23.3 ACCELERATION TECHNIQUES

在进行渐进式计算时，全局GI解算器依赖于若干加速方法来减少运算量。使用这些方法的目的，是为了在保证输出具有一致性的情况下（见23.5.2节）尽可能快的收敛全局光照计算结果。

## 23.3.1 VIEW PRIORITIZATION

当编辑视图摄像机只观察到场景的一小部分时，重新渲染光图中所有的图素是完全没有必要的。我们可以根据当前视图中包含的图素为它们安排渲染优先度，从而让光追运算更快收敛，这一点对艺术家来说很重要。这一方法被称为“图素优先级排序”，它们在规划待渲染像素时候执行，如23.2.6节所述。

为了保证视图中的每一个图素至少会被标记为待渲染至少一次，我们会记录视图中图素在若干帧里面是否可见，如图23-19所示。在每一帧中，我们使用nv条从视图摄像机发射到场景中的光线进行光线求交，如下面的伪代码所示。如果一个图素在可见性求交中多次成功，那么在后面进行合图的时候该像素就需要被特殊处理。我们使用原子逻辑操作来保证在一帧当中，一个像素只被规划成待渲染一次。

（伪代码）

每个图素数值的方差在光照积分计算过程中会被计算（参见23.2.8节），这一数据会被用来决定未收敛的图素是否需要被渲染。这意味着只有当某个图素数值的方差低于一个低到一个限度一下，它才会被安排等待渲染。

## 23.3.2 LIGHT ACCELERATION STRUCTURE

每个关卡会包含大量的灯光。因此计算光照时候向每一个光源投射射线进行后续预估的开销非常可观（参见23.2.3节），而且这一运算会发生在光路路径上的每一个反射点上。为了产生巨大的运算开销，我们会使用一光照加速结构对光源进行过滤，只计算潜在有可能对着色点产生光照的光源。

光照加速解构是一种空间哈希函数[16]。世界空间会被加速结构划分成很多轴向对齐的包围盒，每个包围盒的大小为*S*3D。每个包围盒会被映射到一个具有ne个元素的一维哈希函数上。当场景被加载时，哈希表会在GPU上被创建一次，其中包含了所有的灯光。每个哈希元素里面存储了一个灯光能够被影响到的所有的光追采样样本点索引。在本方案中，场景中的每一个包围盒都不可能包含场景中所有的灯光。这是因为寒霜引擎出于性能上的考虑，要求所有灯光影必须有一个随亮度变化而变化的有限影响范围[7]。（译者注：因此，灯光不会和场景中所有的包围盒产生交集，所以某一个灯光不会落入场景中所有的包围盒中。）

每个包围盒的体积*S*3D是根据灯光影响范围的平均值除以一个常数系数算得的。这个系数可以用来改变包围盒的默认体积，默认值是8。哈希表的数量ne会被设成一个较大的质数，例如524或者287。由于潜在的哈希冲突（hash collision），多个世界空间中不相邻的包围盒可能会被索引到同一个哈希元素上。理论上，这种情况会让某些完全不在灯光影响范围内的区域被计算（false-positive lights），但是我们在实际工程中并没有遇到这个问题。使用加速结构计算的光照结果如图23-12所示。更多细节请参考原论文[16]。

**图23-12.** 每个图素需要被计算的光源数：蓝色为灯光数为0的地方，橙黄色是密度最高的地方。左图：建筑内部充满了很多局部光源。右图：开放环境内的碰撞体基本上不会索引哈希表里面的灯光，因为它们很少有机会能与局部光源产生交集。

23.3.3 IRRADIANCE CACHING

23.2节中讨论了如何使用光路追踪计算光图和辐照度体积区域[9]。 在光路路径上的每一个转折点上，这一点附近的局部光源会被计算得到局部光源的直接照明成分。在这过程中，转折点会向光源投射射线来检测光源之于点的可见性。然而，一个场景会包含非常多的光源，计算光路中每一个转折点到每一个光的可见度开销极大。另外，光路的构建都是各自独立进行的，因此光路方向变得非常发散的机会很大。光路发散会产生非常大的计算消耗，发散的方向会访问空间中不相连的结构，使得内存访问碎片化产生较大的延迟。为了避免巨大的开销，我们在解决方案中使用辐照度缓存（irradiance caching）速光路的构建。

23.3.3.1 DIRECT IRRADIANCE CACHE LIGHT MAPS

辐照度缓存的核心思想，是将入射光线按照物体表面的分块存储到一个便于快速访问的数据结构中。辐照度缓存的完整描述可见Křivanek等人的工作[6]。寒霜引擎的全局光解算器根据场景中有关全局光光图的参数，存储光图空间中的“直接辐照度”（direct irradiance），如图23-13。光照构成中的每一种组成会有各自独有的缓存：局部光照，阳光和天光。这种按光照成分将缓存分离的策略至关重要，这种策略可以让我们分别更新不同成分各自的光照，避免全量更新。例如天光改变的时候只改变天光的缓存而不去重新计算局部光照（参见23.4.2节）。当这些缓存被计算以后，它们会被存储到贴图中供光路上的转折点采样直接光辐照度时候使用，而不是显式地去采样灯光（参见23.2.3节）

**图23-13** 场景“Granary”的辐照度可视化。该场景被局部光源，方向光，天光照亮。左上图：间接光照光图。右上图：方向光的辐照度缓存。左下图：局部光源的光照辐照度缓存。右下图：红色的点显示了辐照度缓存被计算时使用的采样点。

如图23-14所示，光源的辐照度，在之前的讨论中是使用投射很多射线做光追的方法得到的（图23-6），若使用辐照度缓存，则计算方法就变成了若干次贴图采样。这种方法可以充分利用硬件线性插值过滤。因此，使用缓存的方法会比让光路上的点直接对光源进行光线追踪的消耗小得多。若想提高缓存的精确性只需提高缓存存储的贴图即可。若缓存贴图图素对应的面积过大，则复杂几何遮蔽产生的一些小细节会丢失，使得最后的全局光照结果变得非常模糊。提高缓存的分辨率还可以减缓光图因为线性采样产生的模糊。

**图23-14。**为了在使用光路追踪进行后续预估时更精确的计算辐照度，光路上的每一个转折点（绿色点）必须向光源发射射线（黄色虚线）检测光源的可见度。当使用辐照度缓存时，每个点只需要一次贴图采样就能知道场景中每一个光源贡献的辐照度。 缓存可以节省调所有与光源求交用的射线计算，并加速光路追踪的速度。辐照度缓存用物体表面小的长方形表示，黄色的长方向代表有辐照度（E>0），黑色代表没有辐照度（E=0）。

使用辐照度缓存对性能提升有较大的帮助，相关讨论参见23.5.1节。有关时间的信息，请参见表23-3。在不同的场景、灯光设定与视口位置的组合下，计算的收敛速度获得了极大的提升，如图23-17所示。

23.3.3.2 CACHE UPDATE PROCESS

在引擎编辑器刚刚打开的时候，缓存处于一个不可用的状态。当场景中的太阳改变角度和亮度发生改变时，也会导致直接光部分的缓存不可用（译者注：需要更新），对于天光与局部点光源，情况也类似。若缓存不可用，那么系统会开始对它进行更新。拿局部点光源举例子，当它需要被更新时，会做如下事情：

1. 对于光图缓存中的每个图素，选取nic数量的样本。

2. 对于每个样本，使用光追计算辐照度。

3. 对于面光源，在光源表面选取若干样本。选取的数量会根据光源之于着色点所对应的立体角进行调整[10]。立体角越大，需要的样本越多。

4. 样本采样结果累加到光图的辐照度当中。

上述操作也会对阳光与天光进行，采样太阳的样本均匀散布在表示太阳圆盘的表面上，采样天空时样本均匀的分布在天空半球内。由于对每一种光源要采多少样本数量是已知的，因此我们可以使用低差异序列生成样本。对于太阳，只使用nic\_sun= 8个样本产生边缘清晰的阴影。对于局部光源，样本数nic\_L最多可达128。由于面光源会产生软阴影，因此会给予更多的样本数。对于天光，使用 nic\_sky=128 个样本。天空需要的样本数较多，因为基于物理模拟的天空的辐照度变化范围很大，可能会产生高频变化信息，尤其是当太阳处在地平线的时候[2]。在缓存更新的这几步中，我们忽略掉透明表面的影响（参见23.2节），因为若考虑透明产生的影响，光线需要穿过透明物体表面。然而辐照度缓存存储的数据是无遮蔽（non-occluded）的数据（可以直接被看到的光源对辐照度产生的贡献）。由于每一次更新需要计算nic=8个样本，因此当太阳迭代更新过1次，局部光源迭代16次，天光迭代16次以后，我们就认为每一种光源的缓存更新完成了。缓存更新使用的样本个数可以由使用者根据游戏项目进行调整。例如，某些游戏的光照强依赖于阳光与天光而不依赖于局部光照，那么用户就可以让阳光与天光使用的样本多一些。

23.3.3.3 FUTURE IMPROVEMENTS

**间接光缓存** 除了直接照明的缓存（direct light caching），累积并计算间接光的辐照度会减少光追的运算量。如23.2.8节所述，光图图素的计算收敛情况是会被跟踪记录的。因此当一个图素已经收敛以后，这个图素就可以参与间接光照的计算了。这样（其他像素发出的）光路就不用再进行扩展光路扩展，减少运算量。

**自发光表面** 目前自发光表面没有计算到直接光辐照度缓存中。在23.2.4节中，我们提到发光表面可以被转化成三角形的面光源。转换成的面光源可以直接被当成局部光源进行采样并被纳入到辐照度缓存的范畴内。若效此法，则只有艺术家改变了带有自发光表面的模型时，自发光缓存才需要被更新。

# 23.4 LIVE UPDATE

23.4.1 LIGHTING ARTIST WORKFLOW IN PRODUCTION

使用寒霜引擎开发游戏的团队里，艺术家的职责包含设置场景的灯光及场景整体的气氛。为达成此目的，艺术家通常会对场景做若干种操作。最常见的操作包括布置光源，调整光源，移动物体，改变物体的材质等等。其他的操作还包括更改某个物体占用光图的大小，以及控制光照物是被光图照亮还是被辐照度体积区域照亮。后面这两种操作的主要目的是调整光图存储大小以及光图内资源的规划调整。

如23.1节所述，寒霜引擎的离线全局光照解算器是运行在CPU上的光线追踪系统。以前想要在游戏编辑器里面获得游戏场景全局光照的更新，需要等待数分钟甚至几小时。将解算器改为GPU运算以后，这一过程只需几秒到几分钟。为了实现这种质变，计算场景全局光照的性能在此变得非常重要。加速这一过程的方法在23.3节中有介绍。在这一节内容中，我们主要关注如何针对艺术家的一些特定场景编辑操作进行优化。

23.4.2 SCENE MANIPULATION AND DATA INVALIDATION

对于一个场景来说，光图，辐照度体积区域以及辐照度缓存均可看作状态机，其状态在每次全局光照迭代计算完成后被刷新。当场景中的灯光，模型，材质有变动或更新时，当前全局光照的状态会被标志为不可用。对于艺术家来说，提供相对正确的结果可以更好地理解当前场景照明的状态。为了保证预览结果正确，一个最简单的方法是（每次发生改动时）全量更新所有的光照数据。同时，为了保证用户操作的体验（流畅），我们也需要避免舍弃过多的已计算数据，导致每次用户编辑场景时候都要重新计算，使得预览结果产生太多噪点。

表23-1展示了不同的操作执行后，哪些种类的辐照度缓存会被更新。光图（以及辐照度体积区域）除了在改变光图分辨率操作以外，任何操作之后都需要全量更新。缩放光图大小不会触发场景预览中辐照度缓存的更新，因为场景预览的时候，每一个模型网格都有他们自己专属的光图（译者注：只用于预览场景）。渲染预览场景的模型时，我们使用网格专属贴图的概念以及“无绑定技术”（bindless techniques）使用这些贴图渲染场景。材质改变会影响间接光照，不过它们并不影响缓存因为缓存里面存储的是直接光照的辐照度，它的数值并不受材质的影响。需要注意到，（灯光的）辐照度缓存只有在灯光种类改变时才会改变。

**表23-1.** 当输入改变时（左列），全局光照状态机需要被重置（上行）。输入与重置项目的对应关系用叉子表示。

这一做法中的最大瓶颈来自于模型的改变。当模型被添加、更新、变换或者删除时，所有的光图和辐照度缓存都需要被重置。只因为一个网格产生变化而重置全场景的光图是非常浪费的。将来如果能对此进行改进，则需要将网格在场景中的位置考虑进去，只更新网格周围区域的光图。更进一步的，也可以考虑通过将网格位置周围范围内的光图图素采样次数变为使得这些图素得到更新。这样可以让艺术家在编辑场景时更加的顺滑，也不会产生错误的预览结果。

# 23.5 PERFORMANCE AND HARDWARE

## 23.5.1 METHOD

在最近有关光线追踪的研究工作中，性能的衡量标准一直关注的是纯技术层面上的，例如每秒多少千兆数量的射线（giga-rays）或者每秒每帧的采样数目。在本文中，我们关注的更多是艺术家编辑灯光时候的体验，因此我们的衡量指标更多反映的是体验的质量。艺术家眼中的“性能指标”更多指的是（光照）更新频率（refresh rate）与运算收敛速度（convergence rate）。这两项指标分别从运算频率与计算结果质量两方面衡量用户得到的体验。

全局光照解算器的渐进式输出的质量主要决定于用户主观上对图像变化产生的感知，同时输出产生的错误也会随着时间进行而被记录。对于我们的使用情景，“L1量度法” [13]（L1 Metric）（译者注：此处说的L1就是指L1范数（L1 norm））是一种不错的衡量方法。我们在光图和辐照度体积区域上使用L1量度法。只有在视图内的像素会参与到计算当中。若一个像素在所有迭代更新中所占的计算比重达到了1%，则这个像素会被认为是一个关键需要被统计的像素。为了限制独立参数的个数，测量的更新频率被锁定在33毫秒。

为了计算图片之间的感知差别，我们需要一个绝对正确的图片参照（Ground Truth）。这一绝对正确值使用的是本文中全局光照解算器的基础版本，不使用任何优化、简化也不限制计算时间。图素的收敛程度会被益智计算，每一个图素都收敛时（详见23.2节），输出的光图与辐照度体积区域结果会被存储起来作为绝对正确参照。

应用在全局光照解算器上的某一个特定加速技术的性能提升效果可以被衡量。这类测试从辐照度缓存状态机被清零时开始。具体想确定相关技术是否对某一种特定的缓存计算有帮助，可以使用不同的用户输入当作测试内容（详见23.4节）。

使用了加速技术后的运算结果产生的错误，以及它与绝对正确参考之间的差别会被记录下来，然后用图表的方式随时间展示出来。性能提升参照的对象是没有使用加速技术的版本。

23.5.2 RESULTS

衡量性能时，我们会使用两个场景，每个场景设立一个观察点。这两个场景要能够代表艺术家编辑场景的典型状态。第一个是一个室内场景，主要由局部光源照亮（Granary by Night）。第二个场景是一个真实项目中的开放大世界场景（*Plants vs. Zombies Garden Warfare 2* 中的 Zen Peak）。图23-15与表23-2分辨展示了场景的视觉复杂度与各项统计数据。除了表中显示的数据以外，还有一个具有相关性的数据是每个场景局部光源影响范围的重叠程度。Zen Peak 的灯光布满整个场景，而 Granary by Night 中的影响范围包围盒里面会出现有10个等在一个包围盒里面的情况。后面这个场景的高重叠让计算的开销变得较大，表23-3列举了计算辐照度缓存所用的时间。

**图23-15。** 两个测试场景，（a）Granary by Night. (由Evermotion提供) (b) Plants vs. Zombies Garden Warfare 2 中的ZenPeak (由 Popcap Games. 2018 Electronic Arts Inc.提供)

**表23-2.** 测试场景复杂度。局部点光源包含点光源，聚光灯与锥形光三种。

**表23-3.** 每一个计算迭代的平均耗时。如23.3.3.2所说，当辐照度缓存完全收敛时候其计算消耗会变为0（当所有的样本点都被计算完之后）。

表 23-16 中的图表展示了随时间变化的错误程度，y轴按照指数倍率缩放。注意观察会发现事发场景收敛的更快，因此曲线在Y轴上会产生差异。

很明显，图素视图优先级排序会让收敛速度产生较大的提升。如表23-2所述，不同的场景需要被渲染的图素数量很接近。提升的速度应该是可见区域的图素与总的图素数量的比值。使用优先级排序（详见23.3.1节）的渲染结果如图23-16所示。测试的起始时刻所有的数据都被重置（都需要从头计算），两条曲线分别代表了使用和不使用加速技术的错误收敛情况。如曲线所示，第一个测试场景中的加速提升非常明显，但是第二个场景因为我们使用的图素估计算法的原因提升不那么明显。这是因为第二个场景中光图的图素的视图重要性均一致，因此优先级规划算法不会优先规划一些小尺寸上的图素而是全场景都算上了，因此收敛速度比预期慢。

**图 23-16。**在两个场景中使用视图优先级规划前后，性能速度提升使得收敛曲线产生差异。两个场景分别为Granary by Night（左图）与Zen Peak（右图）。两条线展示了与对照组场景光图的L1误差。 使用了视图优先级规划（虚线）可以提高收敛速度，尤其是对于收敛过程最初的几秒钟，对比全量计算所有光图图素的情况（红线）。

本文第二个讨论的加速技术是辐照度缓存。在运算起始时，这种方法会占用计算资源计算缓存中的数据，这部分数据并不直接被使用。缓存会计算场景中所有光图图素，并不会只计算光线追踪光路中经过的图素。这个步骤会影响场景渲染初始阶段的收敛速度，图23-17显示了这种影响。但是，仅仅过了一秒以后，使用缓存的计算过程就远远快于（不使用缓存的）基础版本的运算过程。辐照度缓存在第二张图中的收敛速度也很快，但是会产生约4.5%的错误。这个错误的引入是因为缓存分辨率太低，因此在灯光较为丰富细节较多的高频区域无法表现的很好。请参考23.3.3节了解有光直接光辐照度缓存的问题细节。注意，辐照度缓存一旦被计算完，则很长一段时间都不用再被计算，直到用户做了特殊的场景操作为止。详细讨论请参考23.4节。

**图23-17。**使用辐照度缓存带来的收敛速度上的收益图。两个场景分别为Granary by Night（左图）与Zen Peak（右图）。图中显示了计算结果与真值参照图的相对误差（L1）。使用了辐照度缓存（虚线）的收敛速度要比单纯使用光线追踪做后续估计（红色线）的收敛速度要快。速度的差异在Granary场景尤为明显，该场景拥有很多的局部光源因此需要发射更多的求交射线检测光源可见度。

去噪的主要目的是为了让用户获得视觉上更佳的输出结果。如图23-18所示，使用去噪后收敛速度还是会提高。这一点至关重要，因为去噪过程不会对收敛计算产生影响，只会影响用户看到的图像。图23-19总结了相关技术应用后对图像产生的影响差异。

**图23-18** 使用去噪方法带来的收敛速度上的收益图。两个场景分别为Granary by Night（左图）与Zen Peak（右图）。图中显示了计算结果与真值参照图的相对误差（L1）。去噪过程去除了大部分的高频与中频噪声，使得去噪后的结果更接近收敛时的状态。

**图23-19** 使用了不同加速方法的场景在不同时间点上的视觉对比。第一行：所有的图素同时开始渲染，且光路中的每一个点都进行后序预估。第二行：使用视图优先级排序规划之后的结果。第三行：使用辐照度缓存（IC）代替后序预估后的结果。第四行：同时使用试图优先级排序与辐照度缓存的结果。第五行：将两种技术与去噪合并起来一同作用的结果。

23.5.3 HARDWARE SETUP

如果当前的系统中只有一个GPU，那么引擎编辑器与全局光照解算器会将使用同一个GPU。在这种情况下，操作系统会平均分摊两者的进程时间片。然而，这种默认的设置体验并不流畅，因为这种调度策略不可能将运算量分的很均匀，达到我们想要的帧率。要么巨大的光追运算任务造成引擎编辑器严重卡顿，要么光线追踪工作无法及时被消化使得场景预览需要等待很久才能收敛。我们推荐在系统中使用两个GPU，以防止编辑器与解算器同时竞争一个GPU的运算资源。

如图23-30所示，当系统存在两个GPU时，第一个被用来渲染编辑器和游戏，第二个处理全局光照解算器的光追任务。当光追任务迭代更新完成一次后，光图和光照探针体积区域（light probe volumes）（译者注：这里说的光照探针应该和前文说的辐射度体积区域是一个东西）会被拷贝到另外一个GPU上提供给编辑器用于预览场景。为了达成这个目标，我们使用了DirectX 12的multi-adapter mode。这种模式下GPU的控制是被系统通过显式操作完成的[14]。最适合做光追运算的GPU会被用来计算光线追踪任务，而另外一个只要支持Direct X 11就可以运行引擎编辑器。

图23-20。寒霜引擎编辑器的运算在GPU0上完成，而全局光照解算使用另外一个协处理GPU（GPU1）。这种配置可以让每一个进程分别占用一个GPU，防止卡顿。

将来，我们可以将解算器扩展到n+1个GPU上，n个GPU做光线追踪，剩下的一个提供给艺术家运行编辑器。目前的双GPU设置已经可以较好地适应艺术家的操作，为寒霜引擎编辑器提供流畅无卡顿的操作体验。

# 23.6 CONCLUSION

本文讨论了一个基于寒霜引擎应用于多款电艺产品制作中的实时全局光照预览系统。该系统允许灯光美术师在几秒钟内快速预览场景的全局光照效果，而最终版本的效果往往需要烘焙几个小时。我们坚信这一系统会极大的提高艺术家的工作效率，并让他们将有限的精力投入到纯美术与创作过程中。使用这套系统可以让美术花更多的时间在迭代、打磨场景品质上，从而间接地提高了最终产品的质量。

本方案使用了若干种加速技术，例如光图图素的渲染规划与辐照度缓存。这些技术能够提高运算结果的收敛速度，并且几乎不会影响最终输出结果的质量。我们还将去噪技术融入到解决方案内，（在光图需要被重新计算时）使用较少的采样数为用户提供品质较高的预览图像。

放眼未来，我们可以继续研究更高级的缓存方式，例如较长光路路径的引导与双向光路追踪[17]（bidirectional path tracing）。为获得更好的用户体验，我们推荐使用带有两个GPU的机器配置使用本系统来实现异步无锁的全局光照更新。更进一步地，我们可以设立支持DXR 的GPU农场并将它们安装在世界各地，为全世界的电艺工作室中的每一个人提供高品质的全局光照预览服务。