第十三章　图形管线

图 13.1: 城市渲染效果图 ©Unreal Engine 5

在前面的课程中，我们已经学习了如何绘制三角形．然而，现代的大型实时游戏场景中包含大量的三角形面片，为了让人眼不感到间断，每秒至少需要渲染 30 帧以上场景．在 CPU 上想要串行执行这样规模的计算几乎是一件不可能的事情．为了高效地完成实时图形渲染，人们对渲染的流程进行了很长时间的优化和演进，最终形成了图形管线（Graphics Pipeline）的概念．由于图形管线中很多操作可以进行高度并行化，专用图形硬件——GPU 应运而生，随之而来的是各种对 GPU 进行编程的 API，包括 OpenGL、DirectX、Metal、Vulkan 等．本章我们将从光栅化渲染的回顾开始，介绍图形管线的基本构成，然后介绍主要硬件 API 的编程模型、GPU 的基本结构．最后，我们将介绍图形管线在近年来的一些新进展．

13.1　光栅化渲染总结

我们在前面的课程中已经了解了如何在屏幕上绘制一个简单的三角形（遍历判断像素点是否位于三角形内）、如何在简单光源下给三角形着色（Bling-Phong 着色模型）、如何处理三角形的遮挡和走样问题（z-buffer，super-sampling）．将这些要素组合到一起，就可以实现一个基本的软件渲染（对应于使用 GPU 的硬件渲染）流程：

1. 输入待渲染模型（若干三角形面片及其光照属性）、各光源属性．

2. 对所有三角形，计算其顶点的投影坐标以及深度，存储其法向量等着色信息.
3. 取一个三角形，遍历判断所有像素是否在投影后的三角形内部.
4. 对三角形内的像素点，遍历所有光源，将着色叠加得到像素点颜色.
5. 计算三角形在此处的深度值，如果相对于 z-buffer 深度更浅，更新 z-buffer，并将此处颜色写入 framebuffer.
6. 对所有三角形进行 3-5 步操作.
7. 将 framebuffer 同步到显示器，渲染结果被显示在屏幕上.

随着计算机图形学的发展，现在 GPU 上运行的渲染程序和这一流程已经产生了不少细节上的差异，但总体的结构并没有太多变化. 图形管线的基本结构如下图所示.

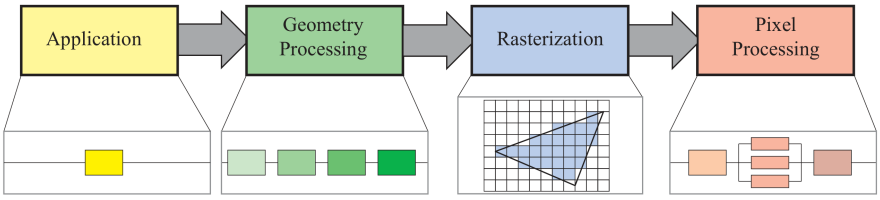

图 13.2: 图形管线示意图 ©Real Time Rendering 4

从应用输入的场景和模型开始，渲染流程中首先对输入进行几何处理. 基础的几何处理包括投影、计算顶点颜色等，简称 T&L（Transform and Lighting）. 几何处理输出的一系列三角形在接下来的步骤中被光栅化，得到处于三角形投影内部的像素点. 然后，这些像素点根据光源属性、光照模型以及后面我们要提到的纹理映射被着色，最后通过 z-buffer、alpha-blending 等算法确定最后的颜色输出，显示到屏幕上.

13.2　图形管线与硬件 API

上面介绍的图形管线仅仅是一个大概的结构，而各个图形硬件厂商对此的具体实现都是不一样的. 以 OpenGL 为例，其 API 中的图形管线实现如下图所示. 图中所有的过程都是在 GPU 上执行的.

在 OpenGL 中，**几何处理**阶段被进一步细化为顶点着色（Vertex Shading）、曲面细分（Tessellation）、几何着色（Geometry Shading）、顶点后处理（Vertex Post-Processing）、图元组装（Primitive Assembly）等几个阶段. 这些阶段中，顶点着色、曲面细分、几何着色是可编程的（使用 OpenGL 着色器语言 GLSL 编写），而其余部分在硬件驱动中实现，仅能通过一些选项调整其参数. 大部分情况下需要手动编写的仅有**顶点着色器（Vertex Shader）**一项，这一项需要处理顶点到投影空间的坐标变换，并计算光照有关的输入数据. 除此之外，顶点着色器还可以修改顶点的位置，利用这一特性可以实现刚体旋转、关节动画、软体拉伸变形、水面波纹、地形起伏等一系列复杂的特效. 顶点着色器仅能修改顶点位置而不能增加/删除顶点，为此 OpenGL 的图形管线后续加入了曲面细分和几何着色阶段，大大增加了几何处理阶段的自由发挥空间，详见后文 Mesh Shader 介绍.

下面展示一个简单顶点着色器的 GLSL 代码：

```
#version 450
layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inNormal;
layout(binding = 0) uniform GlobalUniformData {
```

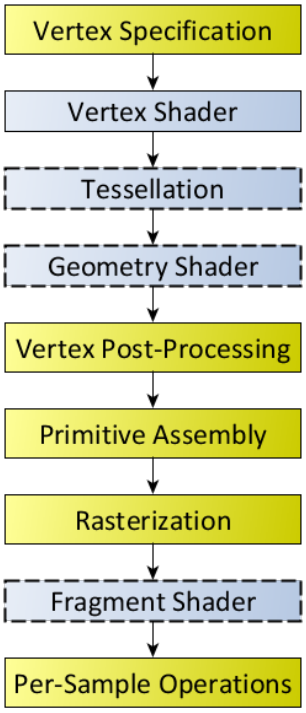

图 13.3: OpenGL 图形管线示意图 ©OpenGL Wiki

```
    mat4 proj;
    mat4 view;
} global_data;
layout(location = 0) out vec3 fragColor;
void main() {
    gl_Position = global_data.proj * global_data.view * vec4(inPosition, 1.0);
    fragColor = inNormal;
}
```

其中，#version 450声明这是 OpenGL 4.5 的着色器代码，in、out 分别修饰着色器的输入和输出，uniform 代表全局数据，layout 表示数据在内存中存储的顺序．整个几何处理的运行逻辑位于main函数中，gl_Position则是 OpenGL 定义的全局变量，表示输出的顶点位置，是顶点着色器中唯一必须输出的量．fragColor作为输出，后续将传给下面提到的片段着色器，其内容是可以自由指定的．例如，这里将输入三角形的法向量编码为颜色传给后续片段着色器．上面这一段代码会在 GPU 中遍历所有输入的顶点并行执行，得到所有顶点的输出后交由 OpenGL 进行光栅化操作．

光栅化过程是光栅化渲染的核心部分，这一部分被硬编码在 GPU 中，并针对性地做了大量优化，仅能通过参数微调．光栅化之后，GPU 并行处理的对象就不再是顶点，而是屏幕中的像素，对每一个像素的着色操作被定义在**片段着色（Fragment Shading）**阶段．这一阶段是可编程的，所有的光照、着色都在这一阶段完成，所编写的程序成为**片段着色器（Fragment Shader）**或**像素着色器（Pixel Shader）**．片段着色结束后，OpenGL 会对着色器的输出颜色根据深度、透明度等进行裁剪与混合，同时根据指定的反走样算法执行反走样操作．在 OpenGL 中，这些操作都是编码在 GPU 中的，无法自定义实现．

下面展示一个简单片段着色器的 GLSL 代码：

```
#version 450
in vec3 fragColor;
out vec4 outColor;
void main() {
    outColor = vec4(fragColor, 1.);
}
```

其中，#version 450声明这是 OpenGL 4.5 的着色器代码，in、out 分别修饰片段着色器的输入和输出，整个着色的运行逻辑位于main函数中．值得一提的是，这里fragColor是与上面提到的顶点着色器中输出相对应的量．在 OpenGL 中，如果片段着色器的输入和顶点着色器输出的名称和类型都相同，那么 OpenGL 就会自动将二者关联起来．

在我们展示的代码中，顶点着色器和像素着色器中都有类型为vec3的fragColor的参数，这就意味着像素着色器得到的输入是由顶点着色器的输出决定的．但与简单的参数传递不同，顶点着色器输出的是逐顶点的值，像素着色器接受的输入是逐像素的输入．这中间进行的转换正是光栅化这一步进行的操作．在默认情况下，OpenGL 会把顶点着色器的输出通过我们上一章介绍的**透视矫正的线性插值**方法插值到每个像素上，喂给像素着色器使用．因此即使我们在像素着色器中没有进行任何的插值操作，直接输出得到的就是光滑插值之后的结果．到这里你也能够更好地理解我们之前介绍的 Gouraud 着色与 Phong 着色之间的差别．Gouraud 着色就是在顶点着色器计算 Blinn-Phong 光照模型，输出得到颜色之后通过光栅化插值；而 Phong 着色是在顶点着色器输出每个顶点的位置、法向等信息，在像素着色器中计算 Blinn-Phong 光照模型．事实上，OpenGL 提供了三种插值选项，由**插值限定符（Interpolation Qualifier）**决定．flat in vec3 fragColor 指这个变量在光栅化阶段不会进行插值，像素着色器会得到三角形第一个顶点上的值．noperspective in vec3 fragColor 指这个变量会在屏幕空间做线性插值．smooth in vec3 fragColor 指这个变量会通过透视矫正的方法插值，默认情况可以去掉smooth不写．

OpenGL 之外，还有许多流行的图形 API，其中最有名的是微软的 DirectX．OpenGL 和 DirectX 几乎是同时演进的，特别是在 OpenGL 3.3 引入 core profile 之后，二者之间在功能上几乎一致．随着图形渲染需求越来越广泛，还诞生了面向移动设备的 OpenGL ES 和在浏览器里运行的 WebGL 等 API．大约在 2016 年，维护 OpenGL 的开源组织 Khronos Group 推出跨平台的 Vulkan API 作为 OpenGL 的后继，苹果、微软也相继推出了自己的新 API Metal 和 DirectX 12．新 API 中包含了一些图形管线上的新进展，这些内容将在本章的最后讲述．

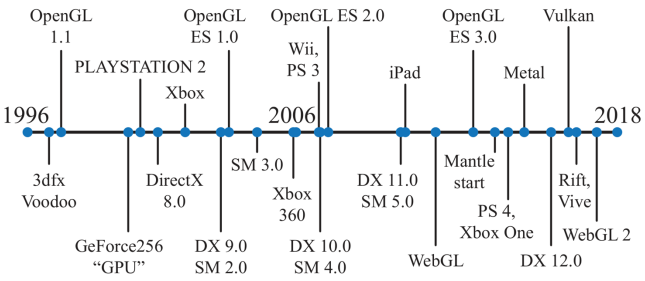

图 13.4: 主要图形硬件和图形 API 的出现节点 ©RTR4

13.3　专用图形硬件——GPU

看到这里，你很可能会有疑问：图形硬件是怎样做到比 CPU 计算快这么多，以至于大家愿意去编写复杂的 GPU 代码而不是在 CPU 上渲染再传给 GPU？答案就在 GPU 的架构上．以 NVIDIA 的 Maxwell 架构为例，其架构图如下图所示．

图 13.5: NVIDIA Maxwell GM204 架构图 ©Life of a triangle

以 GM204 架构为例，GPU 中包含一个统管任务的 Giga Thread Engine 和几个不同的 GPC（Graphics Processing Cluster），每个 GPC 又包含若干 SM（Streaming Multiprocessor）和一个 Raster Engine．每个 SM 中包含很多个核心，这些核心都可以并发执行计算．作为对比，一块消费级 CPU 可能包含 8~24 个核心，但一块消费级显卡通常可以达到 3000~10000 个核心，这使得 GPU 可以同时处理大量数据．当然，GPU 堆核心也是有代价的：GPU 上的单个核心通常计算能力不高，并且 GPU 核心的访存延迟明显高于 CPU 核心，这也意味着 GPU 上算法的数据局域性非常重要．总而言之，GPU 的特点是：高吞吐、高延迟．对于图形渲染，GPU 的这些缺陷并不致命，但其超高并发的特性非常适合用于处理含有大量数据重复操作的图形管线任务．

如图，应用程序首先调用图形 API 的 drawcall，此时 CPU 传递一系列指令和数据给 GPU．顶点缓存里的数据被分给各个核心并发执行，这里执行的命令包括顶点着色器、几何着色器、坐标变换等．在 GPU 中，数据加载和命令执行是可以同时进行的，这样数据一旦到达 GPU 就可以参与计算．此后，这些顶点数据通过 Work Distribution Crossbar 被分配给各个包含三角形面片的 GPC．Raster Engine 接到数据后，执行光栅化计算，并完成数据插值等任务，将必要的着色的数据传给片段着色器．所有的片段着色器也是并行执行的．片段着色器给出颜色后，经过 ROP（Render Output Unit）处理一下遮挡、混合，就可以写入帧缓冲（Frame Buffer）等待屏幕显示了．在图形管线中，计算量最大的部分往往是对所有三角形顶点循环的顶点着色运算，和对所有像素、所有三角形循环的片段着色运算．在 GPU 上，这两个循环可以展开到不同的核心上进行计算，从而大大提高了计算效率．

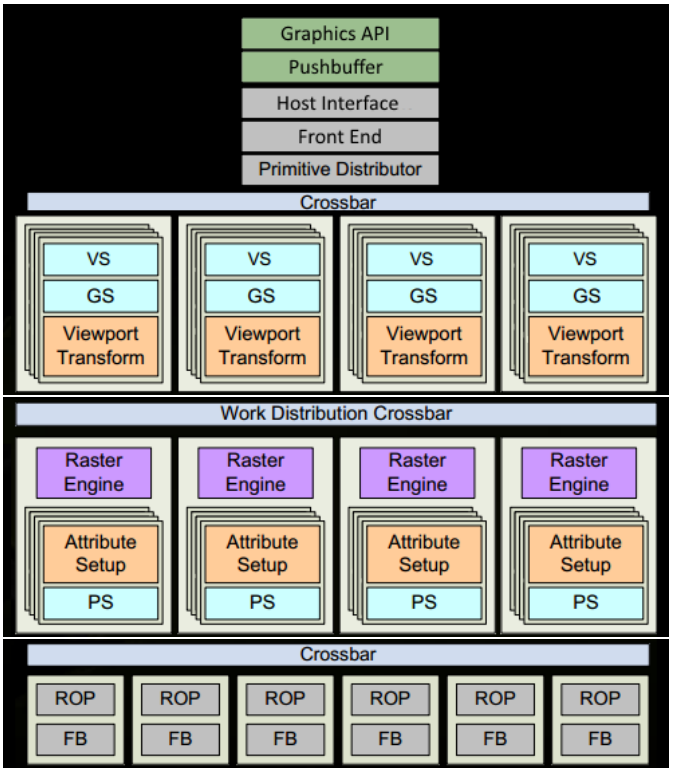

图 13.6: NVIDIA GPU 图形管线 ©Life of a triangle

GPU 的用途并不止于图形计算．自从可编程管线出现，许多人便开始研究是否可以利用 GPU 高度并行的特点加速现有的一些其他领域计算．早期这些计算需要把数据使用位置矢量的方式编码到着色器的输入中，然后提取分量进行计算，十分麻烦．CUDA、OpenCL、ROCm 等计算 API 的推出大大降低了使用 GPU 进行通用计算的门槛，目前，GPU 已经成为数值模拟、神经网络训练、图像处理、视频剪辑、虚拟货币挖矿等领域中非常重要的加速硬件．在图形管线中，粒子系统模拟、一部分几何处理任务也可以借助 GPU 的通用计算功能来实现，现代图形 API 中把这部分代码称为 Compute Shader，详见后文 Compute Shader 的介绍．

13.4　图形管线的新进展

OpenGL 诞生于 1992 年，彼时计算机图形学还处在 2D 绘制阶段，3D 渲染技术尚不成熟．直到 2010 年 OpenGL 3.3 发布，才初步形成上面介绍的 OpenGL 图形管线（在此之前，OpenGL 默认所有过程都由硬件驱动实现，仅可调整部分参数）．与 OpenGL 竞争的 DirectX API 在早期也存在类似的问题．到了 2016 年前后又出现了 Vulkan、Metal 等新的图形 API，这些 API 与当时新发布的 DirectX 12 都在图形管线上引进了一些新变化．以 Vulkan 为例，其图形管线示意图如下

我们这里介绍一部分图形管线上的新进展，有兴趣进一步了解的同学可以查阅相关文献（Real Time Rendering）．

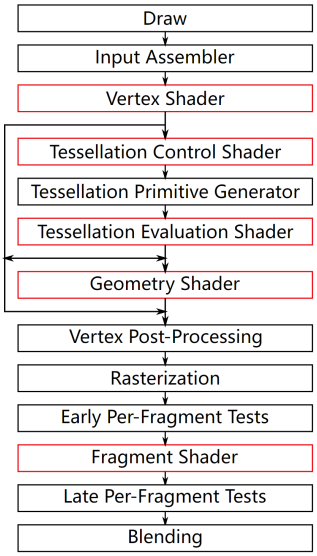

图 13.7: Vulkan 图形管线示意图 ©Vulkan Registry

13.4.1 Deferred Rendering

细心的读者可能已经注意到了，我们上面介绍的渲染管线里，每一个三角形都经过了着色，但处理完遮挡等问题后，大部分的着色结果都被舍弃了．有没有办法可以“按需着色”，只着色那些会对最终渲染结果产生影响的三角形部分呢？答案是有的，这一算法称为 Deferred Rendering，与之对应的传统图形管线着色算法称为 Forward Rendering．二者的区别是，Forward Rendering 中深度测试在着色之后，而 Deferred Rendering 中着色被推迟，先计算深度测试和着色所需的属性，再完成着色过程．相对于 Forward Rendering，Deferred Rendering 大大减少了实际执行的着色计算量，而缺点是对透明物体的处理比较麻烦，只能提前裁剪一部分着色，无法产生明显效果；并且 Deferred Rendering 需要存储额外的 G-buffer 数据，这会带来相当高的显存开销．后一问题可以通过一种名为 Deferred Texturing 的技术改善，限于篇幅原因此处不再详细展开．

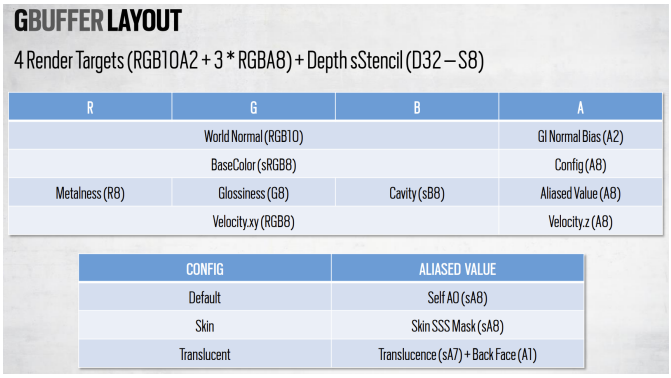

图 13.8: Ubisoft《彩虹六号：围攻》中 G-buffer 的内存排布 ©Rendering Rainbow Six | Siege

在 Forward Rendering 的流程中，计算着色所需的投影点三维坐标、法向量、漫反射颜色、材质等变量都由顶点着色器直接传给片段着色器进行处理，相当于每个三角形都传了一个二维数组给片段着色器．实际上，这些量只有在着色像素点上的值才有意义，因此对于一个被部分遮挡的三角形，遮挡部分可以无需给出这些变量．如果没有透明物体存在，片段

着色器只需要知道自己对应的那个表面点上的着色信息，所有需要的着色信息数量仅构成一个二维的数组（对应二维的屏幕像素分布），这就大大减少了着色需要的数据传输和计算. 这个数组通常被称为 G-buffer. G-buffer 中元素的设计是相当自由的，渲染程序可以根据自身需要设置 G-buffer 中的数据排布. 在 Vulkan API 中，存在 Early Per-Fragment Tests 和 Late Per-Fragment Tests 两个阶段，这一划分使得 Deferred Rendering 实现起来更加自然.

13.4.2 Compute Shader

前面提到了可以利用 GPU 做通用计算. 而在渲染过程中，传统的图形管线之外有时也需要用到并行计算. 例如，在着色完毕向屏幕输出时，需要对输出图像进行卷积. 这时候要么在 CPU 上完成这一过程，要么借助 GPU 另写一份通用计算代码，前者将引入不必要的 CPU-GPU 数据传输开销，而后者在数据交互上非常麻烦. 为了解决这类问题，Compute Shader 应运而生. Compute Shader 可以在 GPU 上执行通用计算任务，但和顶点着色器、片段着色器同属图形 API 范围内，且具有相似的交互逻辑；这使得在渲染过程中调用 GPU 进行通用计算变得尤为方便. Compute Shader 最大的意义是它大幅扩展了在图形管线中进行自定义的可能性，想要在图形管线上作出改变，最直接的办法就是引入 Compute Shader.

下面展示一个简单 Compute Shader 的 GLSL 代码：

```
#version 450 core
layout(local_size_x = 1, local_size_y = 1, local_size_z = 1) in;
layout(rgba32f, binding = 0) uniform image2D imgOutput;

void main() {
    vec4 value = vec4(0.0, 0.0, 0.0, 1.0);
    ivec2 coord = gl_WorkGroupID.xy;
    float width = 1000;

    value.x = mod(float(coord.x), width) / (gl_NumWorkGroups.x);
    value.y = float(coord.y) / (gl_NumWorkGroups.y);
    imageStore(imgOutput, coord, value);
}
```

这里gl_NumWorkGroups表示启动的线程组数量，gl_WorkGroupID表示当前线程组的编号；每个线程组内的线程数由变量in的 layout 指定，例如这里指定线程组内为单线程. 上面代码的作用是根据线程组编号对一幅图片的相应位置上色.

有了 Compute Shader 之后，甚至可以绕开传统图形管线在 GPU 上实现渲染. 例如，Unreal Engine 5 中的 Nanite 技术使用 Compute Shader 绕过了固定的光栅化流程，对非常小的三角形自己实现了光栅化流程，大大加速了小三角形的光栅化，使得千万面片场景的实时渲染成为可能. 本章开头展示的城市图片就是使用 Nanite 技术渲染的场景. 未来的图形管线中，光栅化也很可能成为可编程管线的一部分.

13.4.3 Mesh Shader

在大型游戏中，场景的模型通常不会直接拿到管线中进行渲染. 对于远处的精细模型，我们需要利用第 8 章提到过的 LOD（Level Of Detail）等技术对顶点处理后再输入到管线.

而对于近处的粗糙模型，我们则往往需要利用顶点法线插值等方法对三角形面进行加密再输入到管线．这些操作都要求对顶点的增加/删除，而这仅仅用顶点着色器是无法实现的．在曲面细分和几何着色器加入管线之前，这些操作只能在 CPU 上进行，然后传递到 GPU，这大大影响了渲染程序的渲染效率．而现在，借助图形管线中的几何处理部分，这些算法都可以在 GPU 上实现，从而减少了 CPU-GPU 的通信成本，提高了渲染性能．

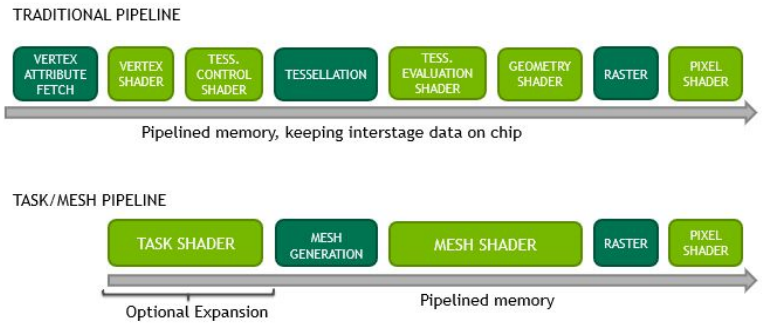

图 13.9: 当前图形管线与 Mesh Shader 对比 ©Introduction to Turing Mesh Shaders

尽管加入了更多的几何处理着色器，现有的图形管线中仍区分了顶点着色器、曲面细分着色器、几何着色器等多种流程，并且其顺序是固定的，这给几何处理带来了不便．2018 年，NVIDIA 率先提出了 Mesh Shader 的概念，将几何处理统一为图形管线中一个完全可编程的步骤，几何处理的灵活性和性能都将大幅提高．在 Mesh Shader 中，GPU 线程不再与顶点或者三角形绑定，并且执行的指令也不再要求是相互独立的，一个线程可以同时获得多个顶点或三角形的信息；顶点处理、曲面细分、几何处理的顺序也更加灵活，可以把顶点处理放在最后以降低显存压力．有了 Mesh Shader，复杂几何体的剔除、程序化几何体生成等处理都可以更加自然地放在 GPU 上进行，而不必再借助 Compute Shader 来实现．

13.4.4　全局光照

在光栅化渲染过程中，全局光照是一个困扰大家很久的问题．前面所提到的各种简单光照模型，几乎都只考虑了光源的直接照射，没有考虑环境的二次反射；而三次以上反射，尤其是漫反射、各向异性反射的全局光照更是难以实现．如果使用后面介绍的光线追踪算法，则可以轻松解决全局光照问题，但光线追踪的计算成本高昂，并且并非所有的图形硬件都带有实时光追单元、支持硬件光追，这限制了光线追踪在实时渲染中的应用．为了在性能与效果之间取得平衡，一种办法是结合使用光栅化管线与光线追踪思想进行渲染．

三角形面片的光线求交需要硬件单元支持，但是有一种数据结构可以实现更简单的光线求交：那就是我们之前提到的隐式表面表示 Signed Distance Field．因此，只要对场景中的每个物体都建立一个低精度的 Signed Distance Field 作为近似，就可以通过光线追踪算法，在片段着色阶段以较低成本实现相对准确的全局光照效果．由于 Signed Distance Field 存储在网格上而非像素或者表面上，帧与帧之间的表面信息是连续的，因此可以利用上一帧的表面光照信息辅助加速全局光照的计算．低精度的光线追踪有其缺点，那就是容易产生噪点；但借助 Compute Shader，在光追结果上进行滤波降噪，即可消除这一问题的影响．Unreal Engine 5 中处理全局光照的 Lumen 技术就运用了这一思想．未来，光栅化与光线追踪的结合很有可能成为下一代图形管线的一部分．