第三部分

反射、折射和阴影

任何光线追踪渲染都会导致关于效果是否支持以及如何支持的一些设计决策。光线追踪的主要历史诉求是它能够很好地处理阴影、反射和折射。然而，当你真的坐下来实现了一个支持这些效果的系统，你将面临一些不明显的设计决策。本书的这一部分描述了几个具体的达成这些设计决策的方法。

虽然对一个清晰的玻璃球进行光线追踪是一蹴而就的，但面对更复杂的模型，就有点令人头疼了。例如，一个简单的装有水的玻璃杯就呈现了光线在三种不同类型的材质之间传递的行为：水和空气，玻璃和空气以及玻璃和水。为了使折射正确，被追踪的光线不仅需要知道自己穿过了哪个接触面，还得知道是从哪种材质穿入哪种材质的。指望美术工作者用这些接触面建模是有问题的；想象一下在玻璃杯中装满水。第[11](#第11章)章《自动处理嵌套体积的材质》中描述了一个巧妙并且经过实战考验的方法来处理这个问题。

困扰几乎所有光线追踪程序的一个问题是当法线贴图产生了物理上不可能的表面法向量时应该怎么处理。显然，用硬编码的条件分支方案忽略这些不可能的法向量会导致不和谐的颜色不连续。每个光线追踪器都有自己的小技巧来处理这个问题。第[12](#第12章)章《用基于微表面的阴影函数解决凹凸不连续的问题》中，提出了一个基于简单的统计学方法的简洁实现。

光线追踪的屏幕空间方法使其在生成精确的屏幕空间阴影方面特别强大，而不会有阴影贴图方法带来的所有锯齿问题。然而光线追踪的计算能快到可以进行实时交互的程度吗？第[13](#第13章)章《光线追踪阴影：维持实时帧率》提供了关于如何做到这一点的详细解释。

大部分简单的光线追踪器都会从眼睛发出光线。通常这些程序几乎不可能产生焦散——就像我们用聚焦光图案制作装有液体的玻璃杯、游泳池或者湖泊。用“几乎”来形容是因为计算结果伴随了太多的噪点。但是，将光线从光源发射到环境中是一种产生焦散的可行方法。事实上，这甚至可以通过实时计算完成，正如第[14](#第14章)章所描述的《使用DXR在单一散射介质中实现光线引导的体积水焦散》。

总之，一个基础的光线跟踪器是非常简单的。把光线跟踪器部署到生产环境需要对基本效果进行一些小心处理，本书的第三部分提供了处理这些基本效果的几种有用的方法。

Peter Shirley

**117**

第11章

自动处理嵌套体积的材质

Carsten Wächter and Matthias Raab

英伟达

摘要

我们提出了一种新颖而简单的算法来自动处理嵌套体积和体积之间的转换，启用按钮操作的渲染功能。唯一的要求是使用封闭的水密体积（使用诸如英伟达的RTX这样的保证水密遍历和相交的光线追踪实现）并且相邻的体积不打算相互交叉，除了正好在（译者注：相邻）体积之间建立起边界的小重叠以外。

11.1对体积进行建模

粗暴的路径追踪已成为模拟光传输的核心技术并用于在许多大制作的渲染系统中渲染逼真的图像[1,2]。对于任何基于光线追踪的渲染器，都需要处理场景中的不同材质之间的关系以及不同几何对象之间的关系：

* 要正确模拟反射和折射，需要知道表面的正面和背面的折射索引（译者注：顺序）。要注意这不仅仅是针对有折射属性的材质，也包括了由菲涅尔效应产生的反射光。
* 可能会需要先确定好体积的参数（散射和吸收），例如，当光线离开具有光吸收性的体积时应用衰减。

因此，对体积数据（译者注：包括嵌套介质）的处理是渲染生产场景的基本要求，必须紧密集成到渲染核心。光线追踪底层对层级结构的遍历和几何体的求交，总会受制于浮点数的精度，甚至就连几何数据的表示（译者注：比如用浮点数变换来进行网格实例化）都会进一步引入浮点精度的问题。因此，为了处理体积之间的过渡至少需要仔细地建立起体积和包围它们的几何体外壳。在下文中，我们将分三种情况对相邻体积建模。见图[11.1](#图11_1)。

**119**

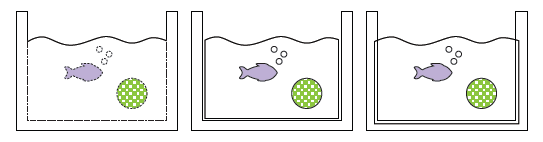


图11.1：左图：用虚线标记了体积之间的明确的边界交叉。中图：加入空气间隙以避免数值（译者注：精度）问题。右图：重叠的体积。

11.1.1单独的边界

用看似简单的方式在相邻体积间共享一个清楚地描述了两个（或更多）介质间的边界的单独表面，也就是两个透明物体的接触面，比如玻璃和水（译者注：的接触面），用一个单独的表面网格替换掉原来的两个（译者注：物体的接触部分各自的）网格，并且把这个网格标记为特殊类型。美术通常无法以这种方式创建体积，因为需要手动把单个模型按照它分别和哪个相邻体积接触拆分成多个子区域。就以常见的装了苏打水的玻璃杯为例，杯中有气泡并且气泡和玻璃杯壁接触，要想手动拆分网格（译者注：的子区域）是行不通的，尤其是当场景还带动画的时候。这个方案的另外一个主要的复杂因素是每个表面都需要为正面和背面提供单独的材质以明确区分（译者注：两种不同材质的边界）。

请注意这个单独表面也可以通过复制来为每个体积提供独立的封闭外壳，这样就能避免所有繁琐的细分工作。然而，在实践当中，要想使表面完全匹配上是相当困难的。美术人员或者他们所使用的建模/动画/模拟软件，可能会（译者注：对相邻体积）选用不同级别的网格细分，或者相邻体积的外壳会因为浮点数精度问题在进行实例化变换的时候产生轻微的不同。因此，光线追踪的实现与渲染核心结合的时候需要经过精心设计，使其能够处理这种（译者注：不一致）表面的匹配。光线追踪的核心，包括用于加速的层级结构构建（译者注：算法），必须保证总能反馈所有最接近的（译者注：不同材质体积的）交叉表面，渲染核心也必须能以正确的顺序对相交面列表进行排序。

11.1.2额外的空气间隙

第二种方法允许相邻体积之间存在微小的空气间隙以缓解前面提到的大多数建模问题。不幸的是，这导致了一般的光线追踪实现中常见的新的浮点数问题：在生成新的追踪路径时，每条光线的原点都需要加一个极小的偏移，以避免自相交[4]。这导致在对相邻体积的外壳求交时，可能会有一个（或多个）体积过渡（因偏移）被完全跳过，所以建立的空气间隙要大于这个偏移值是很重要的。插入小空气间隙的另一个主要甚至更加严重的缺点是空气间隙会大幅改变渲染出来的外观，因为发生了更多非预期的体积过渡/折射。见图[11.2](#图11_2)。

**120**

|  |  |
| --- | --- |
|  |  |
| 图11.2：用微小的空气间隙对鱼缸建模。 | 图11.3：让水体和缸壁稍微重叠。 |

11.1.3重叠外壳

为了避免前两个方案的缺点，我们可以迫使相邻体积略有重叠。见图[11.3](#图11_3)。不幸的是，这第三种方法引入了一个新问题：路径/体积交叉的顺序和数量将不再正确。Schmidt等人[3] 通过为每个体积安排优先级强行保证了正确的顺序，但这需要美术人工来给出明确的设置，对于复杂的设置来说可能会很繁琐，尤其是在制作动画的时候。

请注意，除了提到的三种方案之外，还有另一个值得注意的特殊情况：一个嵌套／封闭的体积完全包含在另外一个体积中，参见图[11.1](#图11_1)中的彩色物体（译者注：鱼和小球）。这些通常被认为会从包围它的体积中减掉（译者注：布尔运算减法），一些渲染实现可能也允许对重叠或嵌套的体积进行混合。这显然对降低实现的复杂度完全没有帮助，在进入或离开相邻体积时之前提到的那些问题仍然存在。由于允许路径一次通过多个体积而使这些过渡甚至更难以检测和正确处理。因此，我们提供的方案是针对一次只处理一个体积的渲染器。

在下文中，我们描述了一种新的算法在使用外壳重叠方法时恢复正确的路径/体积交叉顺序，而不需要手动安排优先级，它作为Iray渲染系统[1]的一部分成功用于生产已超过十年。

11.2算法

我们的算法用一个栈结构管理着所有当前激活（嵌套）的材质，每一次光线到达表面，我们将表面的材质加到栈顶并确定光线进入的材质和边界的背面（译者注：光线离开的材质）。基本的想法是如果材质在栈上被引用了奇数次说明是进入了这个体积，如果被引用偶数次说明离开了这个体积。因为我们假设重叠，所以栈的处理还需要确保光线路径上的两个表面中有一个实际上是作为体积的边界，通过检查光线是否在进入了当前材质后又进入了另一种材质来过滤掉第二个边界以实现这一目标。为了效率，我们为每个栈元素存储两个标记：一个表示栈元素是否是这种材质的最顶部的一个（译者注：栈中按照进入、离开、进入、离开的顺序存储了好多条该材质的栈元素，最顶部的这个表明是在光线的路径上最近一次的进入／离开这种材质），另一个表示它是奇数或偶数次的引用（译者注：这表明光线是进入／离开这种材质）。一旦着色完成而（译者注：光线的）路径还在继续，我们需要区分三种情况：

1. 对于反射，我们将栈中最顶层的元素出栈并将上一个相同材质的实例标记为该种材质的最顶部的一个。
2. 对于已经确定从最新进栈的这种材质中离开的传输（比如折射），我们不仅需要将最顶层的栈元素出栈，还需要移除前一个引用该材质的栈元素。
3. 对于相同材质的边界（这种情况直接跳过）和已经确定了是进入一种新材质的传输，保持栈不变。

注意，在（光线）路径轨迹发生分裂的情况下，例如追踪多重光泽的反射光线时，产生的每条光线需要一个单独的栈。

在相机本身位于体积内的情况下，需要建立一个反映该体积的嵌套状态的初始栈，可以用一条从场景的边界框外部发出的光线朝摄像机位置进行递归的追踪来填充这个初始栈。

11.2.1实现

下文提供了我们对体积栈算法的实现的代码段，一个重要的实现细节是堆栈可能永远不会是空的，一开始就应该包含一个人工（加入）的真空材质（被标记为奇数次并且在栈顶）或者如果相机是在体积的内部的话可以从预处理阶段复制一个初始栈。

如清单[11.1](#代码11_1)所示，体积栈的数据结构中需要保存材质（以这种：在场景中）的索引、栈元素的奇偶性（译者注：是进入或离开材质）和是否为栈顶的标记（译者注：最近一次的进入／离开这种材质）。此外，我们需要能够访问场景中的材质并假设他们是可以比较的，根据实现情况，对材质索引进行比较实际上可能会很有用。

1. **struct** volume\_stack\_element
2. {
3. **bool** topmost : 1, odd\_parity : 1;
4. **int** material\_idx : 30;
5. };
6. scene\_material \* material;

代码清单11.1：材质索引、标记和场景材质。

当光线照射到物体表面时，我们将该物体的材质的索引值加到栈顶并确定此光线实际进入和离开的材质的索引值（译者注：由于相邻材质的边界有留有空隙、刚好吻合和略微重叠三种情况，所以这里说的实际进入和离开的材质也是根据这种三种情况来进行判断的）。在索引相同的情况下，光线追踪代码应该跳过这个边界，leaving\_material这个变量为true表明越过这个边界就是从这种材质离开了，需要在Pop函数中遵守此规则。参见清单[11.2](#代码11_2)。

1. **void** Push\_and\_Load(
2. // 结果
3. **int** & incident\_material\_idx, **int** & outgoing\_material\_idx,
4. **bool** & leaving\_material,
5. // 相交的几何体的材质（在场景材质数组中的索引）
6. **const** **int** material\_idx,
7. // 栈状态（栈数组和栈顶索引值）
8. volume\_stack\_element stack[STACK\_SIZE], **int** & stack\_pos)
9. {
10. **bool** odd\_parity = **true**;
11. **int** prev\_same;
12. // 自栈顶向下遍历，找到和新加入元素的材质相同的栈元素。
13. // （获取它的奇偶性并取消它的栈顶标记）
14. **for** (prev\_same = stack\_pos; prev\_same >= 0; --prev\_same)
15. **if** (material[material\_idx] == material[stack[prev\_same].material\_idx]) {
16. // 注意：必须保证它之前是栈顶元素（信赖其它函数对栈的维护结果，这里不做额外保护）
17. stack[prev\_same].topmost = **false**;
18. odd\_parity = !stack[prev\_same].odd\_parity;
19. **break**;
20. }
21. // 自栈顶向下遍历，找到前一个进入的材质。
22. // （同时满足是奇数次出现、被标记为栈顶并且和新加入元素的材质不相同的栈元素）
23. **int** idx;
24. // 由于相机体的存在，索引值总是非负的。
25. **for** (idx = stack\_pos; idx >= 0; --idx)
26. **if** ((material[stack[idx].material\_idx] != material[material\_idx]) &&
27. (stack[idx].odd\_parity && stack[idx].topmost))
28. **break**;
30. // 现在把新加入元素的材质索引加入到栈顶。
31. // 防止过多的体积嵌套导致栈溢出。
32. **if** (stack\_pos < STACK\_SIZE - 1)
33. ++stack\_pos;
34. stack[stack\_pos].material\_idx = material\_idx;
35. stack[stack\_pos].odd\_parity = odd\_parity;
36. stack[stack\_pos].topmost = **true**;
37. **if** (odd\_parity) { // 假设正在进入这个新加入的材质。
38. incident\_material\_idx = stack[idx].material\_idx;
39. outgoing\_material\_idx = material\_idx;
40. } **else** { // 假设正在离开这个新加入的材质。
41. outgoing\_material\_idx = stack[idx].material\_idx;
42. **if** (idx < prev\_same)
43. // 不留下重叠（区域）
44. // 由于我们还没有进入另一种材质（又回到了嵌套材质里面）。
45. incident\_material\_idx = material\_idx;
46. **else**
47. // 留下重叠（区域）
48. // 表明这个边界应该被跳过。
49. incident\_material\_idx = outgoing\_material\_idx;
50. }
51. leaving\_material = !odd\_parity;
52. }

代码清单11.2：进栈操作

当渲染代码继续光线追踪时，我们需要从栈中弹出材质，如代码清单[11.3](#代码11_3)所示。对于传输事件，只有在设置了leaving\_material为true时才会调用，这种情况下，将从栈中删除两个元素（栈顶元素以及从栈顶往下找到的第一个和栈顶相同材质的元素）。

|  |  |
| --- | --- |
|  |  |
| 图11.4：按照在威士忌和玻璃杯之间留出轻微空气间隙的方式进行建模。 | 图11.5：把威士忌和玻璃的体积略微重叠。 |

1. **void** Pop (
2. // 由（前面那个）Push\_and\_Load函数的逻辑决定的是否已离开当前材质的布尔值
3. **const** **bool** leaving\_material,
4. // 栈状态（栈数组和栈顶索引值）
5. volume\_stack\_element stack [STACK\_SIZE], **int** & stack\_pos)
6. {
7. // 把最后一个加入的栈元素从栈中取出并把栈顶索引往下降一格。
8. **const** scene\_material &top = material[stack[stack\_pos].material\_idx];
9. --stack\_pos;
10. // 我们是否需要从栈中取出（并移除）两个栈元素？
11. **if** (leaving\_material) {
12. // 从栈顶往下查找最近的一个和栈顶元素相同材质的栈元素。
13. **int** idx;
14. **for** (idx = stack\_pos; idx >= 0; --idx)
15. **if** (material[stack[idx].material\_idx] == top)
16. **break**;
17. // 防止输入的栈存在错误导致数组访问越界。
18. // （这种情况出现在Push\_and\_Load函数内部出现栈溢出的时候）
19. **if** (idx >= 0)
20. // 将找到的栈元素上面所有的栈元素依次下移一格以移除这个栈元素。
21. **for** (**int** i = idx + 1; i <= stack\_pos ; ++i)
22. stack[i - 1] = stack[i];
23. --stack\_pos;
24. }
25. // 从栈顶往下找一个和被移除的栈顶元素相同材质的栈元素更新它的栈顶状态。
26. **for** (**int** i = stack\_pos; i >= 0; --i)
27. **if** (material[stack[i].material\_idx] == top) {
28. // 注意：必须保证它之前不是栈顶元素（信赖其它函数对栈的维护结果，这里不做额外保护）。
29. stack[i].topmost = **true**;
30. **break**;
31. }
32. }

代码清单11.3：出栈操作

**121-122**

11.3限制

我们的算法在遇到重叠的时候，总是丢弃第二个边界。因此，几何体的相交实际上取决于光线的轨迹和光源的位置。特别是，不可能把光到相机的路径反过来计算出一条相同的（相机到光），这使得方法对于双向的光传输算法会稍微不一致，比如双向路径追踪。一般来说，缺少明确的删除边界的顺序可能导致去除了错误的重叠部分。例如，水将会为先进入玻璃的那些光线把重叠区域从图[11.3](#图11_3)中的玻璃碗中分割开来。如果重叠如预期般的足够小，那还不会导致视觉可见的问题。然而，如果场景中有大的重叠，就像图[11.4](#图11_4)和[11.5](#图11_5)里面部分浸没的漂浮的冰块，产生的错误可能会很大（虽然那个场景中视觉可见的影响到底有多大可能也存在争议）。因此，应该避免使用明知道会交叉的体积，但这（译者注：即使是交叉了也）并不会破坏算法或损害路径上后续体积交互的正确性。

通过分配优先级[3]使体积保持明确的顺序能够解决这种模糊性，代价是会导致失去一键自动渲染的便利性。这个解决方案有其局限性，因为易用性对很多用户来说是必不可少的，例如那些依赖于现成的资产库、灯光设置和材质，而不知道任何技术细节的用户。

为每个路径管理一个栈会增加（译者注：维护渲染）状态（译者注：所需的内存）大小，因此高度并行的渲染系统可能需要小心地限制体积栈的大小。虽然我们提供的实现捕捉了溢出（的异常），但也只是避免了程序崩溃（译者注：渲染结果还是会不正确）。

致谢

该算法的早期版本是和Leonhard Grunschlob合作构思的，鱼缸的场景部分由Turbosquid提供，其他由Autodesk提供。

参考

1. Keller, A., Wächter, C., Raab, M., Seibert, D., van Antwerpen, D., Korndörfer, J., and Kettner, L. The Iray Light Transport Simulation and Rendering System. arXiv, <https://arxiv.org/abs/1705.01263>, 2017.
2. Pharr, M. Special Issue On Production Rendering and Regular Papers. ACM Transactions on Graphics 37, 3 (2018).
3. Schmidt, C. M., and Budge, B. Simple Nested Dielectrics in Ray Traced Images. Journal of Graphics Tools 7, 2 (2002), 1-8.
4. Woo, A., Pearce, A., and Ouellette, M. It's Really Not a Rendering Bug, You See... IEEE Computer Graphics and Applications 16, 5 (Sept 1996), 21-25.

**125**