Journal of System Simulation

Volume 28 | Issue 10 Article 17

8-13-2020

Online Fragment Culling for Efficient Rendering of Multi-Fragment Effects

Zhou Guo
1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;; 2. Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190;; 3. University of Chinese Academy of Sciences, Beijing 100049, China;

Dengming Zhu
1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;; 2. Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190;;

Zhaoqi Wang
1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;; 2. Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190;;

Wei Yi
1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;; 2. Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190;;

Online Fragment Culling for Efficient Rendering of Multi-Fragment Effects

Abstract

Abstract: Multi-fragment effects such as real-time transparency are rendered by visiting the per-pixel fragments in depth order. The *k*-buffer algorithm rendered the scene in a single pass to obtain several layers of fragments. The read-modify-write hazard while updating the buffer should be avoided thus the performance was likely to be hit hard. The temporal and spatial coherence among frames were exploited to cull the input fragments in an online manner. A fragment cache based on a semi-heap was proposed. The worst case scenario of the time complexity was analyzed. The reverse reprojection was employed to construct the culling interface on the fly while rendering a frame. Since the buffer updating operations are effectively reduced and no preprocessing is required, the complexity scene during walkthrough or changing can be rendered efficiently.

Keywords

depth peeling, heap sort, fragment interlock, reverse reprojection

第 28 卷第 10 期 系统仿真学报© Vol. 28 No. 10
2016 年 10 月 Journal of System Simulation Oct., 2016

在线片元剔除的多片元效果高效绘制方法

周果[1,2,3]，朱登明[1,2]，王兆其[1,2]，魏毅[1,2]
（1.中国科学院计算技术研究所，北京 100190；2.移动计算与新型终端北京市重点实验室，北京 100190；3.中国科学院大学，北京 100049）

摘要：多片元效果有实时透明等重要应用，它按深度顺序处理每个像素的所有片元。已有的 *k*-buffer 算法在单遍绘制场景时，剥取并缓存距离视点最近的多个片元，这需要避免缓存更新时的读写冲突，降低了在复杂场景上的计算效率。对此提出*利用帧间的时空相关性，允许单遍绘制时在线地剔除缓存外片元*的方法。描述建立半堆结构的片元缓存的过程，分析最坏情况下的时间复杂度，在绘制每帧时反向重投影构造剔除界面。由于减少了缓存更新操作且不需要预处理，复杂场景特别在被漫游或发生变化时的绘制效率被显著提高。

关键词：深度剥离；堆排序；片元临界区；反向重投影
中图分类号：TP391.9 文献标识码：A 文章编号：1004-731X (2016) 10-2407-08

Online Fragment Culling for Efficient Rendering of Multi-Fragment Effects

Zhou Guo[1,2,3], *Zhu Dengming*[1,2], *Wang Zhaoqi*[1,2], *Wei Yi*[1,2]
(1. Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China; 2. Beijing Key Laboratory of Mobile Computing and Pervasive Device, Beijing 100190; 3. University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Multi-fragment effects such as real-time transparency are rendered by visiting the per-pixel fragments in depth order. The *k*-buffer algorithm rendered the scene in a single pass to obtain several layers of fragments. The read-modify-write hazard while updating the buffer should be avoided thus the performance was likely to be hit hard. The temporal and spatial coherence among frames were exploited to cull the input fragments in an online manner. A fragment cache based on a semi-heap was proposed. The worst case scenario of the time complexity was analyzed. The reverse reprojection was employed to construct the culling interface on the fly while rendering a frame. Since the buffer updating operations are effectively reduced and no preprocessing is required, the complexity scene during walkthrough or changing can be rendered efficiently.

Keywords: depth peeling; heap sort; fragment interlock; reverse reprojection

引言

图形处理器(graphics processing unit，GPU)采用管线的思想流式地光栅化图元，被三角面片覆盖的像素获得对应的片元。每个像素遍历从属面片生成的多个片元，可以实现例如透明和半透明材质、构造实体几何布尔操作等效果[1]。这就需要保证每个像素的所有片元深度有序，在 GPU 上根据相邻片元间的距离和遮挡关系进行实时绘制。

复杂场景在光栅化后，每个像素对应的片元数量差异较大。这给单指令、多线程架构的 GPU 提出了挑战，即针对高深度复杂度我们需要实现显存的高效管理和计算任务的负载均衡。已有的工作需要将整个场景读取多遍，每次光栅化后按深度顺序剥取一层或者多层片元。这类方法通过

收稿日期：2016-05-09 修回日期：2016-07-22；
基金项目：国家自然科学基金(61173067，61379085，61532002)，国家 863 计划项目(2015AA016401)；
作者简介：周果(1987-)，男，陕西，博士生，研究方向为计算机图形学；朱登明(1973-)，男，安徽，博士，副研究员，研究方向为自然现象模拟和可视化。

多次迭代遍历所有片元，每次仅剥离少量片元来减小存储开销，但浪费了大量显存带宽。另一类方法单遍读取场景但缓存所有片元然后排序，需要占用大量存储有溢出的风险。为了节约存储，可以对缓存中的片元进行压缩或采用高效的场景组织方式。前者在屏幕空间只保留对绘制结果贡献较大的片元(比如 *k*-buffer 算法及其扩展[1])，但是对输入的图元顺序敏感无法保证绘制效率。后者在物体空间将场景剖分使得每块场景最大深度有确定上界，虽然场景块能被分别绘制但预处理代价高故不适用于动态场景。

针对 *k*-buffer 算法在复杂场景上效率较低的问题，本文提出一种改进的方法实现快速剔除，能够在线地丢弃不重要的片元，在满足单遍绘制场景的前提下改进了绘制效率。我们首先描述构建半堆结构的片元缓存的方法，在临界区中使用最大堆维护部分片元；其次分析了算法的执行效率与图元输入顺序间的关系，给出了最坏情况下的时间复杂度；最后反向重投影缓存后面的一层片元，构造了深度边界作为当前帧的缓存的剔除依据。本文方法利用帧间的时空相关性来减少缓存更新操作，不需要预处理故能够高效地绘制复杂场景特别是动态场景。

1 相关工作

多片元效果的高效绘制关键在于保证每个像素对应的所有片元深度有序。经典的 *z*-buffer(即深度缓存)为每个像素记录深度最小的片元，对每个输入片元进行深度比较来决定是否进行替换，最终能够获得距离视点最近的片元。而考察多个片元的效果，需要每个像素尽可能记录部分或所有片元。为了满足这个需求，已有方法多遍或单遍绘制场景来遍历片元。

1.1 多遍场景绘制的准确结果

复杂场景在光栅化后生成的片元总量往往超过显存上限，所以通常采取多遍绘制而每次仅处理部分片元。深度剥离[2]相对视点从前往后采用两个深度缓存交替测试，每次迭代剥取一层片元，然后在下一次迭代中它被用来丢弃前面的片元从而获得下一层片元。这里每次都要光栅化整个场景，故需要读取的数据量很大。而使用遮挡查询判断是否剥取结束，又需要 GPU 同步等待故效率低下。在每次迭代中可以剥取多层片元，减少迭代的次数从而间接提高效率，比如双深度剥离[3]使用最大最小深度缓存每次剥取最前面和最后面两层。还有使用多绘制对象(multiple render targets，MRT)对其进行扩展的桶深度剥离[4-5]，使用桶排序在多个深度区间内执行双深度剥离。该方法预先光栅化一次场景，获得每个像素的近似深度直方图，然后确定每个桶对应的深度区间，但映射到相同桶的不同片元发生竞争故绘制会有明显瑕疵。本文在片元临界区中更新缓存，能够完全避免读写冲突。

1.2 单遍场景绘制的近似结果

多遍绘制方法每次保留的片元只对应场景中的局部区域，大部分区域产生的片元在比较深度后被丢弃，因此浪费了大量带宽和计算资源。单遍绘制方法为每个像素维护所有片元链表然后进行排序[6]，但这需要消耗大量存储空间。由于预先并不知道生成片元的数量，这类方法往往不能准确地分配显存，很有可能造成浪费或者溢出。为了降低内存开销，一类方法尝试使用逐像素固定长度的数组来获得近似结果。通过保留对绘制结果贡献最大的多个片元，比如使用原子操作[7-10]或者片元临界区[11]保留距离视点最近的多个片元，而后者也可以用来丢弃整个缓存中贡献较小的片元[12-13]。这类方法虽然可以获得较好的绘制结果，但是绘制效率受场景的组织方式影响很大。

高效的场景组织可以减小甚至消除近似引入的误差。通过绘制时缓存部分无序片元，*k*-buffer 以流的方式在缓存中实现比较、重排、混合和丢弃操作。它要求输入图元深度有序，使得片元按顺序生成以减少读写冲突，故预先将所有面片按它们的

重心进行排序确定遮挡关系。由于面片被多个 GPU 线程并行地光栅化，生成的片元往往只能基本有序。当重心排序不能很好地描述面片的遮挡关系时，输出片元的顺序无法保证故导致瑕疵。将场景凸多面体剖分为部件，可以确定每个部件的最大深度[14-15]。利用体绘制中的网格投影法，可以在绘制时确定部件的遮挡关系来逐个绘制。这类方法绘制效率很高，但在漫游时每帧部件排序耗时、预处理代价高故适用于小规模静态场景。

2 本文方法

透明效果需要每个像素 p 的所有片元按深度顺序计算混合方程[3]：

$$\begin{aligned} C_{\text{dst}} &= A_{\text{dst}}(A_{\text{src}}C_{\text{src}}) + C_{\text{dst}} \\ A_{\text{dst}} &= (1 - A_{\text{src}})A_{\text{dst}} \end{aligned} \tag{1}$$

这里 $(C_{\text{dst}}, A_{\text{dst}})$ 给出了帧缓存中记录的颜色和不透明度，其中 A_{dst} 初值为 1 而其他为 0，而 $(C_{\text{src}}, A_{\text{src}})$ 为输入片元的颜色。本文的目的是获取所有片元中距离视点最近的多个片元然后按深度顺序计算(1)式，尽早地剔除缓存外的片元来减少缓存更新操作、提高绘制效率。

2.1 半堆结构的片元缓存

我们采用的大顶堆数据结构为每个像素存储部分片元，而剩下的片元直接存储到数组形式的序列中。本文方法在绘制前不需要对图元重心排序，在更新缓存时不会发生读写冲突，同时不需要现有 OpenGL 标准外的图形硬件特性。为了避免缓存被多个线程同时读写，使用基于原子操作的转锁在片元着色器中实现了临界区。利用它我们在缓存中记录了距离视点最近的 k 个片元，然后在接下来的步骤在寄存器中对这些片元统一进行插入排序。单遍绘制场景以在片元着色器中缓存构建的主要过程如下：

算法1：片元临界区实现的缓存更新

```
void main() {
    float c = pack(color, opacity); // 打包片元颜色和不透明度
    // 分配输入面片的索引
    uint i = atomicAdd(C, fragCoord.xy, 1);
    if (i < uint(depth)) { // 缓存未满时直接存入
        imageStore(fragImg, (fragCoord.xy, depth - int(i)), (c, fragCoord.z));
    }
    else if (fragCoord.z < imageLoad(F, (fragCoord.xy, 0)).g)
    { // 尽早剔除
        bool retry = true;
        while (retry)
        {
            if (atomicExchange(M, fragCoord.xy, 1) == 0)
            {
                arrayInsert(c);
                memoryBarrierImage();// 保证前面的更新操作完成
                imageStore(M, fragCoord.xy, 0);
                retry = false;
            }
        }
    }
    discard;
}
```

这里使用两个视口大小的二维无符号(r32ui)纹理 C 和 M 为每个像素分别记录片元的编号、临界区状态，使用三维无符号浮点(rg32f)纹理 F 以大小为 depth 的一维数组记录每个像素的片元。其中 C 被初始化为 1 以跳过缓存中的堆顶(即数组中堆的部分)，F 使用两个通道分别存储了输入片元 f 打包的颜色、不透明度 c 和深度 z。首先根据输入片元的索引判断缓存的当前状态，若未满则将它直接存入。否则先与堆顶的片元(即 F 中的第一个元素)比较深度，当较大时输入片元必定位于缓存外，故直接剔除就能避免访问缓存、减少线程为进入临界区的忙等待。注意此时堆顶可能会被其他线程更新，而堆顶的片元只会被更近的片元替换故不影响结果的正确性。

当输入片元 f 更近时 F 中的片元半堆结构需要被更新。而多个 GPU 线程并行修改缓存会发生读写冲突，故需要临界区保护堆的更新过程。每个像素使用 M 标志当前临界区的状态，每个线程反复尝试使用原子操作将 M 置位。当多个线程的原子操作发生碰撞时，它们会被串行化故最终都能依次执行临界区中的代码。这里布尔变量 retry 标志是否需要重试原子操作。在使用 f 更新堆后，当前线程需要释放临界区给其他线程。这就要求缓存更新后的结果在 M 被复位前被其他线程可见，故使用屏障(barrier)保证内存操作的顺序。而文献[10]并没有注意到这个问题，可能导致缓存结果不完整。

算法 2 展示了在临界区中用 f 更新堆结构的过程。在更新前需要重新与堆顶比较进行剔除测试，这是由于之前的测试在临界区外，而此时堆顶可能已经被其他线程更改。由于更新完整的堆结构访存代价太高，我们在 F 中为每个像素维护大顶堆和序列两个部分有 (H,S)，其中 H 仅维护完整堆中靠近堆顶的部分，而 **S** 直接乱序存储剩下的片元。在片元着色器中假设当前最远的片元 b=f，每个像素 H 的更新过程如下：

1. 遍历 **S** 行深度比较，而 $\forall a \in \mathbf{S}$ 如果它的深度更大则令 $b = a$ 并记录它的索引 j；

2. 用 b 来替换堆顶有 $\mathrm{S} = \mathrm{S} \setminus \{b\}$ 然后更新 H，然后将 f 放到 b 的原始位置即 $\mathrm{S} = \mathrm{S} \cup f$ 。

由于算法 2 中 H 仅包含一个元素即堆顶故直接替换即可。对于更大的堆，第 i 个元素的两个孩子分别为 $2i$ 和 $2i+1$。更新 H 时从堆顶开始递归地将每个元素与它的两个孩子比较和交换即可。这里我们将文献[10]中的无序序列和堆结合，能够根据实际的硬件平台情况调整两者的比例以获得最佳性能。在单遍绘制场景后 F 就缓存了距离视点最近的 $k = \mathrm{depth}$ 个片元，它们在另一个片元着色程序中被读取到寄存器进行插入排序然后按(1)式混合。

算法 2： 片元缓存堆结构更新

```
void arrayInsert(float c)
{
    if (fragCoord.z < imageLoad(F, (fragCoord.xy, 0)).g)
    {
        vec2 b = (c, fragCoord.z);
        int j = 0; // 堆顶片元将被替换
        for (int i = depth - 1; i > 0; i--)
        { // 找到S中最远的片元
            vec2 a = imageLoad(F, (fragCoord.xy, i)).rg;
            if (a.g > b.g) {
                b = a;
                j = i;
            }
        }
        // 用次远片元更新H并将输入片元放到S中次远片元的原始位置
        imageStore(F, (fragCoord.xy, 0), (b.r, b.g));
        imageStore(F, (fragCoord.xy, j), (c, fragCoord.z));
    }
}
```

2.2 缓存更新算法复杂度分析

更新无序序列和堆结构都需要在片元临界区中进行，故代价很高。每个像素的线程更新缓存越频繁，其他线程的忙等待时间就越长，而这与输入图元是否深度有序有很强的正相关性。给定输入面片集合

$$T = \{t_i \mid i = 1, \ldots, N\} \tag{2}$$

假定 T 在被视点投影变换后，光栅化得到的片元都覆盖同一个像素有片元集合

$$F = \{f_j \mid j = 1, \ldots, N\} \tag{3}$$

其中：3 N 为面片数量，而面片和片元满足有 $g: T \to F$ 。这里将光栅化和片元的深度排序操作一起看作函数关系 g，显然它是一个双射。由于面片被多个 GPU 线程并行地光栅化，而每个线程的执行时间又受缓存、内存访问延迟和指令调度策略等各方面影响，现有图形管线无法保证 T 和 F 的排列(permutation)的顺序完全相同。

$$K(\sigma,\tau)=\left|\left\{(i,j):i<j,\begin{pmatrix}\sigma(i)<\sigma(j)\wedge\\ \tau(i)>\tau(j)\end{pmatrix}\vee\begin{pmatrix}\sigma(i)>\sigma(j)\wedge\\ \tau(i)<\tau(j)\end{pmatrix}\right\}\right| \tag{4}$$

为了度量这两个排列的差异，我们定义这两个集合的所有排列的集合分别为 T_N 和 F_N 。对于 $\sigma\in T_N$ 、 $\tau\in F_N$ 和 $\tau=g(\sigma)$， $\sigma(i)$ 和 $\tau(j)$ 分别给出了面片 t_i 和它生成的片元 f_j 的排名(ranking)。而 σ 和 τ 间的差异使用(4)式的 Kendall's tau 距离[16]来描述有它表达了通过交换 σ 中相邻元素来获得 τ 所需的次数，显然该度量是对称的。当使用单线程计算 g 时，σ 和 τ 排名一致故距离为零。而 k-buffer 算法中 σ 被并行处理导致局部排名发生变化，缓存作为滑动窗口只记录 τ 中的一段并排序，故窗口的尺寸必须大于两个排列位于窗口中的部分的距离，这样才能保证单遍绘制面片时保证排名一致。为了尽可能缩小窗口，文献[1]需要对 σ 采用耗时的重心排序以减小 K。

当仅保留距离视点最近的 k 个片元时，算法 2 中 K 与窗口的尺寸(即 $|\mathrm{H}|+|\mathrm{S}|$)无关，但它决定了访问临界区的时间和使用 H 进行剔除的效率。当 $K=0$ 时首先片元被直接存储到 S，其次在临界区中更新 H 有平均时间复杂度 $O(\log|\mathrm{H}|)$。对于后面的 $N-k$ 个片元，它们直接与堆顶比较深度后被丢弃，故总的时间复杂度为

$$O(|\mathrm{H}|\log|\mathrm{H}|) \tag{5}$$

然而当面片排名 σ 和片元排名 τ 完全相反时两者距离为 $K=N(N-1)/2$，后面的每个片元都要在临界区中逐个遍历 H 和 S，故总的时间复杂度为

$$O(N(\log|\mathrm{H}|+|\mathrm{S}|)) \tag{6}$$

由于往往 N 比 $|\mathrm{H}|$ 和 $|\mathrm{S}|$ 大得多，这就导致在临界区中消耗了大量时间，加剧了线程忙等待时间而降低了绘制效率。本文利用帧间的相关性来解决这个问题，改进了最坏时间复杂度情况下的效率。

2.3 反向重投影的剔除界面

文献[17-18]提出绘制每一帧时复用前一帧的缓存绘制结果，避免每帧都执行片元着色程序来减小开销。当每帧视点或场景变化较小时，当前帧的绘制结果和以前帧有很大的冗余。但是他们的方法只针对常规的表面绘制，故只需要考虑距离视点最近的单个片元。对于例如次表面散射等低频着色模型，片元的颜色也可以复用而没有明显的视觉错误。多片元绘制中像素的颜色由多个片元共同决定，直接使用他们的方法构造复用缓存开销很大，而视点或场景发生变化时缓存的重用率低、绘制结果有瑕疵。

为了减少维护复用缓存的开销，我们只在每帧记录第 k+1 层片元的深度，然后以它为界面在下一帧快速剔除外面的片元。我们创建帧缓存对象(frame buffer object，FBO)，然后在执行算法 1 和算法 2 时将片元写入 FBO 中，在下一帧绘制时绑定并读取 FBO 的深度纹理进行剔除测试。在视点或场景发生变化的情况下，这里的关键是如何为每个片元找到它在前一帧中的屏幕坐标，然后访问深度纹理中对应像素。

令场景在第 t 帧在屏幕空间的深度即深度纹理的内容为 d_t，它描述了除前 k 层外最近的片元而在下一帧被重投影(reprojection)。 $d_t(\mathrm{p})\in[0,1]$ 代表了屏幕空间每个像素 $\mathrm{p}=(x,y)\in N^2$ 的深度值，我们需要确定它在前一帧中在屏幕空间中的位置

$$(x',y',z')=\pi_t(\mathrm{p}) \tag{7}$$

这里 π_t 为重投影函数，它将从像素 p 可见的场景中的点映射到了前一帧的像素坐标 $\mathrm{p}'=(x',y')$ 和深度 z'。然后跟界面比较进行剔除测试有

$$z'-d_{t-1}(\mathrm{p}')>\epsilon \tag{8}$$

其中：$\epsilon\in\mathbb{R}^+$ 为较小的阈值允许相近的结果也可以被重用。如果测试通过直接将片元输出，完全不需要进入临界区(算法 1 中的第二个分支)。若 $\pi_t(\mathrm{p})=\varnothing$ 即 p 被映射到屏幕外或对应位置没有片元覆盖，我们令 $d_{t-1}(\mathrm{p}')=1$ 间接地关闭剔除测试。

为了利用 GPU 的投影矫正的插值确定每个像素的映射 π_t，我们使用投影变换后的齐次坐标来表示场景。给定前一帧和当前帧的变换矩阵 $\boldsymbol{M}_{t-1}$

和 $\boldsymbol{M}_t$，基于算法 1 实现的在线面片剔除、构造剔除界面的过程如下：

1. 在顶点着色程序中，每个输入顶点 $\mathbf{v}=(x,y,z,1)$ 被变换两次有

$$\begin{aligned}\mathbf{v}_{t-1}&=\boldsymbol{M}_{t-1}\mathbf{v}\\ \mathbf{v}_t&=\boldsymbol{M}_t\mathbf{v}\end{aligned}\tag{9}$$

其中：$\mathbf{v}_{t-1}$ 作为 $\mathbf{v}_t$ 的属性一起被传递给光栅器，而多个 $\mathbf{v}_t$ 在装配后构成的图元就被光栅化和透视校正插值，同时 $\mathbf{v}_t$ 插值后在每个像素进行透视除法。

2. 在片元着色程序中，将每个像素的 $\mathbf{v}_{t-1}$ 进行透视除法就得到了 π_t 中的屏幕空间坐标 x'、y' 和 z'，利用它读取复用缓存中的深度 d_{t-1} 然后与 z' 比较计算(8)式。这一步在算法 1 进入临界区前进行。若测试通过则直接将当前片元输出到帧缓存，否则更新半堆缓存、在 F 中找到更远的片元替换当前片元输出。

这里变换矩阵描述了相机或物体的变化，构造剔除界面需要两个 FBO 在每帧进行轮换以避免读写冲突。由于读取深度纹理 d_{t-1} 时 p' 一般不准确对应像素位置，故采用双线性对 d_{t-1} 重采样。我们设置 d_{t-1} 的边界值为 $(1,0,0,0)$，在 p' 访问越界时不影响剔除的结果。

图 1 展示了漫游场景时，连续两帧构造的剔除界面 d_{t-1}、d_t 和它的颜色，可以看到当视点变化很小时它们的变化很小。我们正是利用这种帧间相关性使用前一帧 d_{t-1} 在线地剔除片元，同时为下一帧构造 d_t，而这都只需要单次绘制场景就能高效地实现多片元效果。为了保证剔除界面的有效性，我们采用和文献[17]一样的方法，将屏幕划分为像素块(quad)来根据 t 跳过剔除实现完全更新。由于片元着色程序是以 2×2 的块为最小单位同步执行的，所以块的尺寸必须更大才有较好的性能表现。

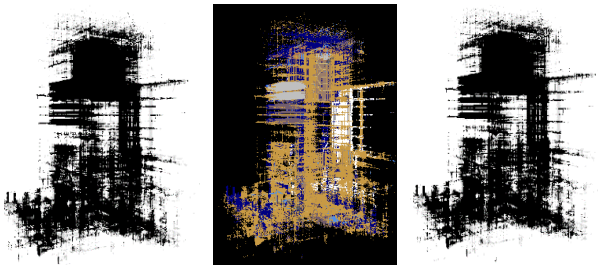

图 1　利用相邻帧构造剔除界面

3　实现细节和实验结果

我们使用 OpenGL 4.5 和 C++实现了提出的方法，使用无符号整型变量 t 从零开始为每帧编号。首先绘制覆盖全屏幕的矩形，每个像素将 C 置为 1、M 置为 0、将 F 的第一个元素置为 1；其次开启深度测试并绑定输入、输出两个 FBO，在将 d_t 清空为 1(即远裁剪平面)后绘制场景；再次绘制覆盖全屏幕的矩形读取 F 中的所有片元，插入排序后计算(1)式混合片元到帧缓存；最后再次绘制矩形根据片元坐标得到棋盘格背景并混入帧缓存。这里利用 t 的奇偶性将两个 FBO 轮换，分别记录 d_{t-1} 和 d_t。多个视点和多个关键帧的绘制结果见图 2。

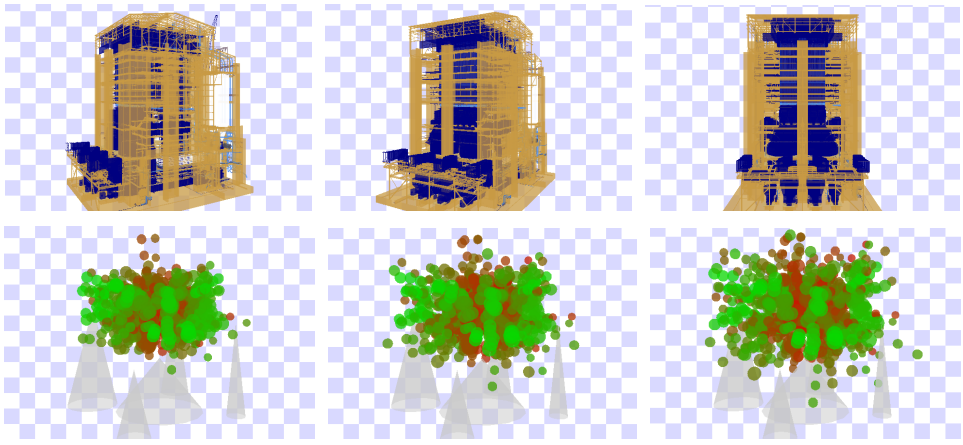

图 2　本文方法的绘制结果

3.1 实验设置

我们在多个场景上测试了本文的在线剔除的方法，发现当场景复杂、每个像素深度复杂度很高时能取得显著的加速效果。像素的片元缓存一维数组的长度为 $k=16$，其中堆部分由于更新成本高故取长度 $|\mathrm{H}|=1$。这里选取了漫游 powerplant 场景的多个视点和 N-body 仿真 balls 场景的几个关键帧，给定不透明度 0.5，图 2 展示了实时透明的绘制结果。所有绘制的图像分辨率为 640×480，并以棋盘格作为背景。每帧的绘制效率由表 1 给出(在线剔除所在列)，使用 ARB_timer_query 扩展度量了绘制每帧花费的时间(ms)，这里不包括场景被读取到内存和被上传到显存所花费的时间。

表 1 在线剔除每帧绘制时间比较 /ms

场景	最大深度	重心正序	在线剔除	重心反序
powerplant 11891501	354	52.17	54.21(0.50)	81.57
	202	51.63	52.19(0.55)	81.04
	157	48.94	50.02(0.21)	63.71
	284	53.06	54.96(0.33)	81.78
balls 434508	62	11.04	13.74(0.34)	18.39
	56	10.74	12.32(0.38)	16.95
	48	10.12	13.81(0.12)	15.48
	50	11.74	13.85(0.23)	17.96

表 1 中括号中的数值给出了本文方法相对于最坏时间的加速比，即重心反序耗时减去本文方法耗时后再除以后者。其中最大深度使用经典的深度剥离得到，从前往后绘制场景并使用遮挡查询判断是否有片元生成。本文在微机上进行实验，配置为 Intel Q9550 2.83GHz 的中央处理器，4 GB 内存和驱动版本为 359.00 的 Nvidia GTS 450 显卡。

3.2 绘制效率讨论

为了评价输入面片的顺序对片元缓存更新效率的影响，所有面片在被提交给 GPU 前按文献[1]在 CPU 上进行重心排序。首先计算每个面片的包围盒，然后取包围盒的中心并将它乘以变换矩阵得到它在眼空间中的位置，最后按位置的 z 分量排序。这里利用了眼空间中视点位于原点的性质，同时 z 分量皆为负值故当面片符合递减顺序时有最好算法复杂度。文献[1]使用重心排序来减少读写冲突，但对于本文基于临界区的方法没有冲突而面片顺序只影响绘制效率。

根据前面的算法复杂度分析可以得知，文献[10]绘制复杂场景的效率受面片的排列顺序影响很大。最好和最坏时间复杂度情况下的绘制效率分别由表 1 的重心正序和反序两列给出，即面片在被给 GPU 前按重心的深度顺序递减或递增排序，这就给出了使用本文在线剔除方法绘制的下界和上界。为了度量本文的在线剔除法的鲁棒性，我们生成自然数随机序列来扰动面片的排名，表 1 中的在线剔除列给出了十个随机序列绘制耗时的平均值和相对于重心反序列的加速比。

根据结果可以看到，在最好时间复杂度情况下(即重心正序列)文献[11]的方法对深度变化并不敏感，这与前文的算法复杂度分析是吻合的。而本文的方法无需对场景的面片进行深度排序，通过快速剔除有效地改进了算法的平均效率。当然利用帧间相关性构造剔除界面要求每帧视点或场景变化较小，这样才能有效地复用 d_{t-1} 来提高绘制效率。而与 GPU 基数排序结合可以进一步提高鲁棒性，将从高位到低位的多遍重排分摊到多个帧，渐进地保证所有面片基本深度有序。

4 结论

本文提出了一种利用帧间相关性提高多片元效果绘制效率的方法，通过复用深度信息构造界面在线地剔除片元。我们首先减少了已有方法的分支发散，将大顶堆和无序序列结合为每个像素实现半堆结构的片元缓存；然后分析缓存更新的时间复杂度，讨论它和面片(即图元)深度顺序间的关系；最后根据最坏情况下的时间复杂度，提出复用前一帧的深度构造剔除界面来提高绘制效率。

在单遍绘制场景的前提下，本文的方法利用前一帧直接剔除了较远的片元，降低进入临界区

的概率故减少了 GPU 线程的忙等待时间，同时又为下一帧构造了新的剔除界面。由于不需要对面片深度排序，本文的方法能够有效地适用于复杂、动态场景的漫游。同时对于存在循环遮挡甚至自相交面片的场景，我们在片元的粒度上构造剔除界面，故绘制效率不受影响。

我们将在下一步的工作中考察使用帧间相关性进行超采样抗锯齿[19]，对于多片元绘制问题如何有效地将多个片元的多个样本分摊到相邻的帧是个非常有意义的问题，这里需要在保证片元深度有序的同时减少存储开销。

参考文献：

[1] Bavoil L, Callahan S P, Lefohn A, et al. Multi-fragment effects on the GPU using the k-buffer [C]// Proceedings of the Symposium on Interactive 3D Graphics and Games. New York, USA: ACM Press, 2007: 97-104.

[2] Everitt C. Interactive Order-Independent Transparency [R]. Santa Clara, USA: Nvidia Corporation, 2001.

[3] Bavoil L, Myers K. Order independent transparency with dual depth peeling [R]. Santa Clara, USA: Nvidia Corporation, 2008.

[4] Liu F, Huang M C, Liu X H, et al. Efficient depth peeling via bucket sort [C]// Proceedings of the Symposium on High Performance Graphics. Saarbrucken, Germany: ACM Press, 2009: 51-57.

[5] 刘芳, 黄梦成, 刘学慧, 等. 基于桶内动态融合的透明现象的高效绘制 [J]. 计算机辅助设计与图形学学报, 2010, 22(3):382-387.

[6] Yang J C, Hensley J, Grun H, et al. Real-time concurrent linked list construction on the GPU [J]. Computer Graphics Forum (S0167-7055), 2010, 29(4): 1297-1304.

[7] Liu F, Huang M C, Liu X H, et al. FreePipe: a programmable parallel rendering architecture for efficient multi-fragment effects [C]// Proceedings of the Symposium on Interactive 3D Graphics Games. Washington D C, USA: ACM Press, 2010: 75-82.

[8] 黄梦成, 刘芳, 刘学慧, 吴恩华. 基于 CUDA 渲染器的顺序独立透明现象的单遍高效绘制 [J]. 软件学报, 2011, 22(8): 1927-1933.

[9] Maule M, Comba J, Torchelsen R, et al. Hybrid transparency [C]// Proceedings of the Symposium on Interactive 3D Graphics Games. Orlando, USA: ACM Press, 2013: 103-118.

[10] Wyman C. Exploring and Expanding the Continuum of OIT Algorithms [C]// Proceedings of the Symposium on High Performance Graphics. Dublin, Ireland: The Eurographics Association, 2016.

[11] Vasilakis A A, Papaioannou G, Fudos I. An Efficient, Memory-Friendly and Dynamic k-buffer Framework [J]. IEEE Transaction on Visualization and Computer Graphics (S1077-2626), 2015, 21(6): 688-700.

[12] Salvi M, Montgomery J, Lefohn A. Adaptive transparency [C]// Proceedings of the Symposium on High Performance Graphics. Vancouver, Canada: ACM Press, 2011: 119-126.

[13] Salvi M, Vaidyanathan K. Multi-layer alpha blending [C]// Proceedings of the Symposium on Interactive 3D Graphics and Games. San Francisco, USA: ACM Press, 2014: 151-158.

[14] 谢国富, 王文成. 单遍数据读取的 GPU 上的多片元效果绘制 [J]. 计算机学报, 2011, 34(3): 473-481.

[15] Wang W, Xie G. Memory-efficient single-pass GPU rendering of multifragment effects [J]. IEEE Transaction on Visualization and Computer Graphics (S1077-2626), 2013, 19(8): 1307-1316.

[16] Kumar R, Vassilvitskii S. Generalized distances between rankings [C]// Proceedings of the 19th international conference on World wide web. Raleigh, USA: ACM Press, 2010: 571-580.

[17] Scherzer D, Yang L, Mattausch O, et al. Temporal Coherence Methods in Real-Time Rendering [J]. Computer Graphics Forum (S0167-7055), 2012, 31(8): 2378-2408.

[18] Nehab D, Sander P, et al. Accelerating Real-Time Shading with Reverse Reprojection Caching [C]// Proceedings of the 22nd ACM symposium on Graphics hardware, Sarajevo, Bosnia and Herzegovina. USA: ACM, 2007: 25-35.

[19] Yang L, Nehab D, Sander P, et al. Amortized Supersampling [J]. ACM Transactions on Graphics (S 0730-0301), 2009, 28(5): 238-250.