MiniRTRender New Feature--基于Volume Stack机制 自动处理嵌套体积

曹琦 FDU Software Engineering 22307110076

问题

光线追踪渲染在处理多个嵌套或重叠透明介质时，如何正确识别当前光线所处的材质环境是一个关键问题。传统的建模方式要求由建模者手动定义复杂的界面或在体积间留下微小空隙，不仅增加了制作成本，而且在动画或动态场景中难以维护。如果模型本身的重叠不做特殊处理则会出现视觉效果的错误。

Ray Tracing Gems CH11 提出volume Stack自动处理机制，通过在光线追踪过程中动态维护一个材质状态栈，实现了对嵌套和重叠体积的自动识别与管理。该机制根据材质在栈中出现次数的奇偶性判断光线是否处于某一体积内部，并结合微小重叠策略避免了手动的优先级设定。

测试场景

为观察和验证Volume Stack机制在嵌套透明材质中的表现，我在Mini compute shader RTRender 项目中构建了一个包含玻璃杯、威士忌液体和漂浮冰块的测试场景。在建模过程中，我有意让威士忌液体与玻璃杯之间存在微小重合区域，以模拟真实渲染中可能出现的建模重叠。

图1：玻璃杯+威士忌+冰块，未引入体积栈机制

图2：玻璃杯

图3：威士忌

图4：冰块

场景中涉及到不同材质的折射率数值设置如下：

- 空气：IOR = 1.0
- 玻璃： IOR = 1.5
- 威士忌（液体）： IOR = 1.36
- 冰块： IOR = 1.31

目标效果

目标是通过修改原始代码引入体积栈机制，在原有建模体积有嵌套的情况下，实现建模体积留有air gap的视觉效果。

图5：调整建模留下air gap，折射效果正确

算法

维护一个所有当前活跃（嵌套）材质的栈结构 。每当光线与一个表面相交时，我们将该表面的材质压入栈中，并确定当前边界在入射侧和背面所对应的材质。 基本思想是：如果某个材质在栈中出现的次数为奇数次 ，则表示当前处于该体积内部；若为偶数次，则表示已退出该体积。由于我们假设体积之间存在重叠，栈的处理还需要确保：在路径上沿着两个重叠表 面移动时，只有其中一个表面被认定为体积边界。我们通过检查是否在进入当前材质之后又进入了另一 种材质，来过滤掉第二个边界。

在每个栈元素中存储两个标志位：

- 一个标志位表示该栈元素是否是该材质的最顶层引用；
- 另一个标志位表示该材质被引用的次数是奇数次还是偶数次 。

完成着色并继续追踪路径，需要区分以下三种情况：

1. 对于反射情况，我们从栈中弹出最顶层的元素，并更新该材质在栈中上一次出现实例 的“最顶层标志位”。
2. 对于透射情况，如果判断光线已经离开最新压入栈的材质，我们不仅需要弹出栈顶元素，还需要移除该材质之前的引用记录 。
3. 对于相同材质之间的边界（应被跳过）以及判断为进入新材质的透射情况，保持栈不变 。

需要注意的是，在路径轨迹发生分裂的情况下，每条生成的光线都需 要拥有独立的材质栈。 当摄像机本身位于某个体积内部时，还需要初始化一个材质栈，以反映该体积的嵌套状态。为了构建这个初始栈，可以从场景包围盒外部向摄像机位置递归地发射一条光线，并根据其与各体积的交点来填充 栈结构。

实现

体积栈的逻辑集中实现在compute shader中，参考RT Gems代码中给出了的栈结构

1. 定义栈结构，实现工具函数

```
87  struct VolumeStackElement {
88      Material material; // 为了适应原有代码结构将material_index修改为直接存完整材质，后续可以优化
89      bool topmost;
90      bool odd_parity;
91  };
92  // 体积栈上限常量
93  const int MAX_VOLUME_STACK = 32;
94
95  // 用于判断volume stack中的结构体是否相等
96  bool isEqualMaterial(Material a, Material b) {
97      return (a.albedo == b.albedo &&
98              a.diffuse_specular == b.diffuse_specular &&
99              a.refractive == b.refractive);
100 }
101
102 // 查找栈中相同材质的位置 ,不存在时返回-1
103 int FindMaterialInStack(VolumeStackElement stack[MAX_VOLUME_STACK], int stack_pos, Material mate
104     for (int i = stack_pos; i >= 0; i--) {
105         if (isEqualMaterial(stack[i].material,material)) {
106             return i;
107         }
108     }
109     return -1;
110 }
111
```

2.修改PathSegment结构体

由于在路径轨迹发生分裂的情况下，每条生成的光线都需要拥有独立的材质栈。故我在这里考虑把volume stack信息保存在用于递归的`PathSegment`结构体中

```
404  struct PathSegment {
405      vec3 origin;
406      vec3 direction;
407      vec3 throughput;
408      int depth;
409      int stack_pos;
410      VolumeStackElement volume_stack[MAX_VOLUME_STACK];
411  };
```

3.新增体积栈操作函数（push pop）

```
113  void Push(inout VolumeStackElement stack[MAX_VOLUME_STACK],
114            int stack_pos,
115            Material material,
116            out Material incident_material,
117            out Material outgoing_material,
118            out bool leaving)
119  {
120      // 查找相同材质
121      int prev_same = FindMaterialInStack(stack, stack_pos, material);
122      bool odd_parity = true;
123      if (prev_same >= 0) {
124          stack[prev_same].topmost = false;
125          odd_parity = !stack[prev_same].odd_parity;
126      }
127      // 查找当前最顶层材质
128      int idx = -1;
129      for (int i = stack_pos; i >= 0; i--) {
130          if (!isEqualMaterial(stack[i].material,material)  &&
131              stack[i].odd_parity && stack[i].topmost) {
132              idx = i;
133              break;
134          }
135      }
136      // 压入新元素
137      if (stack_pos < MAX_VOLUME_STACK - 1) {
138          stack_pos++;
139          stack[stack_pos].material = material;
140          stack[stack_pos].topmost = true;
141          stack[stack_pos].odd_parity = odd_parity;
142      }
143      // 设置进出材质
144      if (odd_parity) {
145          incident_material = (idx >= 0) ? stack[idx].material : air_material;
146          outgoing_material = material;
147      } else {
148          outgoing_material = (idx >= 0) ? stack[idx].material : air_material;
149          incident_material = (idx < prev_same) ? material : outgoing_material;
150      }
151      leaving = !odd_parity;
152  }
```

```
void Pop(inout VolumeStackElement stack[MAX_VOLUME_STACK],
            int stack_pos,
            bool leaving)
{
    // 弹出栈顶元素
    if (stack_pos ≥ 0) {
        // 记录当前栈顶材质
        VolumeStackElement top_element = stack[stack_pos];

        // 简单出栈：减少栈指针
        stack_pos--;
        // 如果标记为 "leaving"，需要移除前一个相同的材质
        if (leaving && stack_pos ≥ 0) {
            int idx = -1;
            // 从栈顶向下查找相同材质
            for (int i = stack_pos; i ≥ 0; i--) {
                if (isEqualMaterial(stack[i].material,top_element.material)) {
                    idx = i;
                    break;
                }
            }
            // 找到后，移除该条目
            if (idx ≥ 0) {
                // 将后面的元素前移一位（填补空缺）
                for (int i = idx + 1; i ≤ stack_pos; i++) {
                    stack[i - 1] = stack[i];
                }
                // 减少栈指针两次（弹出两个元素）
                if (stack_pos > 0) {
                    stack_pos--;
                }
            }
        }
        // 更新该材质在栈中的前一个实例为 "topmost"
        for (int i = stack_pos; i ≥ 0; i--) {
            if (isEqualMaterial(stack[i].material,top_element.material)) {
                stack[i].topmost = true;
                break;
            }
        }
    }
}
```

4. 修改 `cast_ray()` 函数

在 `cast_ray()` 初始化根路径段时：初始化体积栈为空，所有材质条目标记为无效（默认材质、`topmost=false`、`odd_parity=false`），表示初始处于外部介质（如空气）中。

```glsl
vec3 cast_ray(vec3 orig, vec3 dir) {
    // TODO-finished
    // You need to modify this function to add background
    // 初始化一个体积栈,压入空气材质
    VolumeStackElement vstack[MAX_VOLUME_STACK];
    int stack_pos = 0;
    vstack[0] = VolumeStackElement(air_material,true,true);

    vec3 color = vec3(0.0);
    PathSegment stack[MAX_STACK_SIZE];
    int stackSize = 0;

    PathSegment rootSegment;
    rootSegment.origin = orig;
    rootSegment.direction = dir;
    rootSegment.throughput = vec3(1.0);
    rootSegment.depth = 0;
    rootSegment.stack_pos = 0;
    for (int i = 0; i < MAX_VOLUME_STACK; i++) {
        rootSegment.volume_stack[i] = VolumeStackElement(air_material, false, false);
    }
    stack[stackSize++] = rootSegment;
```

在求到光线场景交点后调用 `Push()` 更新堆栈：复制父PathSegment的堆栈到局部变量（避免污染原始堆栈）。

调用 `Push()` 根据当前材质更新堆栈状态，判断进入或退出体积（通过 `odd_parity` 和 `leaving`）。并在折射计算前，获取当前所处材质前后的折射率：

```glsl
467         // Push 材质到堆栈
468         Material incident_material;
469         Material outgoing_material;
470         bool leaving;
471         Push(localStack, localStackSize, hit.material, incident_material, outgoing_material, leaving
472         float eta_out = outgoing_material.refractive.x;
473         float eta_in = incident_material.refractive.x;
474         //获取当前所处的材质（即栈顶有效材质）
475         Material current_material;
476         for (int i = localStackSize - 1; i >= 0; i--) {
477             if (localStack[i].odd_parity) {
478                 current_material = localStack[i].material;
479                 break;
480             }
481         }
```

在折射路径中传递更新后的栈

```glsl
512     // 折射处理：使用当前材质和目标材质的折射率
513     if (hit.material.albedo.w > 0.0 && stackSize < MAX_STACK_SIZE) {
514         vec3 refract_dir = custom_refract(segment.direction, n, eta_out,eta_in );
515         if (length(refract_dir) > 0.0001) {
516             refract_dir = normalize(refract_dir);
517             vec3 offset = dot(refract_dir, n) < 0.0 ? -n * ep : n * ep;
518             vec3 next_origin = p + offset;
519             vec3 next_throughput = segment.throughput * outgoing_material.albedo.w;
520
521             PathSegment new_segment;
522             new_segment.origin = next_origin;
523             new_segment.direction = refract_dir;
524             new_segment.throughput = next_throughput;
525             new_segment.depth = segment.depth + 1;
526             for (int i = 0; i < MAX_VOLUME_STACK; i++) {
527                 new_segment.volume_stack[i] = localStack[i];
528             }
529             new_segment.stack_pos = localStackSize;
530             stack[stackSize++] = new_segment;
531             //stack[stackSize++] = PathSegment(next_origin, refract_dir, next_throughput, segment.depth
532         }
```

在反射路径中（包括全反射）调用 `Pop()`，透射离开材质时调用 `Pop()`

```
557     // 反射
558     if (hit.material.albedo.z > 0.0 && stackSize < MAX_STACK_SIZE && skip_reflect == false) {
559         vec3 next_dir = normalize(reflect(segment.direction, n));
560         vec3 offset = dot(next_dir, n) < 0.0 ? -n * ep : n * ep;
561         vec3 next_origin = p + offset;
562         vec3 next_throughput = segment.throughput * outgoing_material.albedo.z;
563         PathSegment new_segment;
564         new_segment.origin = next_origin;
565         new_segment.direction = next_dir;
566         new_segment.throughput = next_throughput;
567         new_segment.depth = segment.depth + 1;
568         for (int i = 0; i < MAX_VOLUME_STACK; i++) {
569             new_segment.volume_stack[i] = localStack[i];
570         }
571         new_segment.stack_pos = localStackSize;
572         Pop(new_segment.volume_stack, new_segment.stack_pos, leaving);
573         stack[stackSize++] = new_segment;
574         //stack[stackSize++] = PathSegment(next_origin, next_dir, next_throughput, segment.depth + 1,s
575     }
576     // 如果是透射并离开当前材质，Pop 堆栈
577     if (hit.material.albedo.w > 0.0 && leaving && stackSize < MAX_STACK_SIZE) {
578         Pop(localStack, localStackSize, leaving);
579     }
```

最终实现效果

引入体积栈机制后看起来并没有明显的视觉变化，可能代码实现还有corner case没有考虑。

图6：引入Volume stack机制后的效果