

计算机图形学基础 光线追踪大作业

叶宸 软82 2018013400

目录

- 一、 光线追踪
- 二、 景深、软阴影、抗锯齿与贴图
- 三、 参数曲面解析法求交
- 四、 GPU并行加速
- 五、 光线求交加速
- 六、 代码结构

一、光线追踪

采用蒙特卡罗路径追踪算法，通过多次采样从相机发出的光线并追踪其路径，计算路径上的发光、反射、折射等带来的颜色权重，最后求平均以求解物体表面的渲染方程，这个过程求解的渲染方程结果是无偏的，缺点就是收敛速度极慢，采样数直接影响了渲染结果的质量，采样数的增加又会导致渲染时长增加。且由于只追踪从相机发出的光线（即单向路径追踪，区别于双向路径追踪），导致当光源面积很小时很难收敛，也难以模拟焦散等特征。

实现路径追踪的基础算法时主要参考了[smallpt](#)，同时借助[SmallPT —— 99 行代码光线追踪解析](#)以理解算法。实现了漫反射、镜面反射与折射三种表面材质类型。

光线追踪中用到的求交逻辑除了参数曲面外均基于前面几次PA。

代码见 `include/pathtracer.cuh`

二、景深、软阴影、抗锯齿与贴图

1. 景深

原先的针孔相机模型中，相机（发出初始光线的位置）即为空间中一个点；而在模拟景深，即带光圈的相机时，相机（发出初始光线的位置）可以是光圈位置的任何一个点，所以通过指定光圈大小，随机化地选择初始出射光线来模拟景深，为此另外定义一个焦距参数。

在与相机所在位置的距离等于焦距的平面上，物体应该清晰可见，光圈中随机发出的光线需要满足这一特征。采用的方法是，先在光圈内随机产生一个点作为光线的出射点，然后计算这个光线的方向，使得其能刚好射向焦点，出射点加方向即构成完整的出射光线。

景深相机的代码见 `include/camera.cuh` 的 `getRay` 部分

2. 软阴影

通过对面光源采样，被物体阻挡的部分并不是完全无法采样到光照，而是随被遮挡的程度逐渐加深而呈现出逐渐变深的阴影，因此可以得到带有渐变过渡的阴影。在实现中由于不定义光线，而直接给物体材质加上发光这一特征，因此可以方便地实现面光源，即自然实现了软阴影。

不附上具体代码，实现结合在pt当中

3. 抗锯齿

参考smallpt，实现的是SSAA*4抗锯齿，即将每个像素分为四个子像素进行采样，采样完毕后将四个结果取平均作为对该像素本次采样的结果。SSAA的优点是实现非常简单，但相应的增加了数倍采样次数，大大增加了渲染开销。

代码见 `main.cu` 的 `renderPixel` 部分，每个像素都分成四个子像素渲染再压缩

4. 贴图

实现了平面与球体类的纹理映射。对于平面，采用了指定区域平铺+拉伸的映射方法，在创建平面时可以选择一个平铺的基准点以及平铺的两个方向向量，方向的长度同时也决定了材质的拉伸程度。对于球体，采用了墨卡托投影，直接通过交点处的法向换算出uv空间坐标。

在材质类中能够通过图片创建并保存颜色、发光、表面材质类型与梯度信息四个矩阵，其中颜色、发光直接通过读入ppm图片的rgb数据得到；表面材质的输入图片也为ppm，判断每个像素上RGB中权重最大的值，如果R的权重大则存为漫反射（默认），G的权重大则为反射，B的权重大则为折射；梯度信息的输入为灰度图，通过像素间的灰度变化（实际上为了计算方便，只采用了R通道）算出u,v方向的梯度（区间为 $[-1, 1]$ ），用于计算凹凸贴图的法向，最终法向 $norm_{uv}$ 的计算公式为

$$norm_u = (1 - |\Delta u|) * norm + dirU * \Delta u$$

$$norm_{uv} = (1 - |\Delta v|) * norm_u + dirV * \Delta v$$

其中 $norm$ 是该点原来的法向

每次求交时，球体和平面会计算出交点在 $[0, 1] \times [0, 1]$ 空间上的uv坐标，换算成材质类中的矩阵坐标后取出对应点的材质信息，用于后续计算。

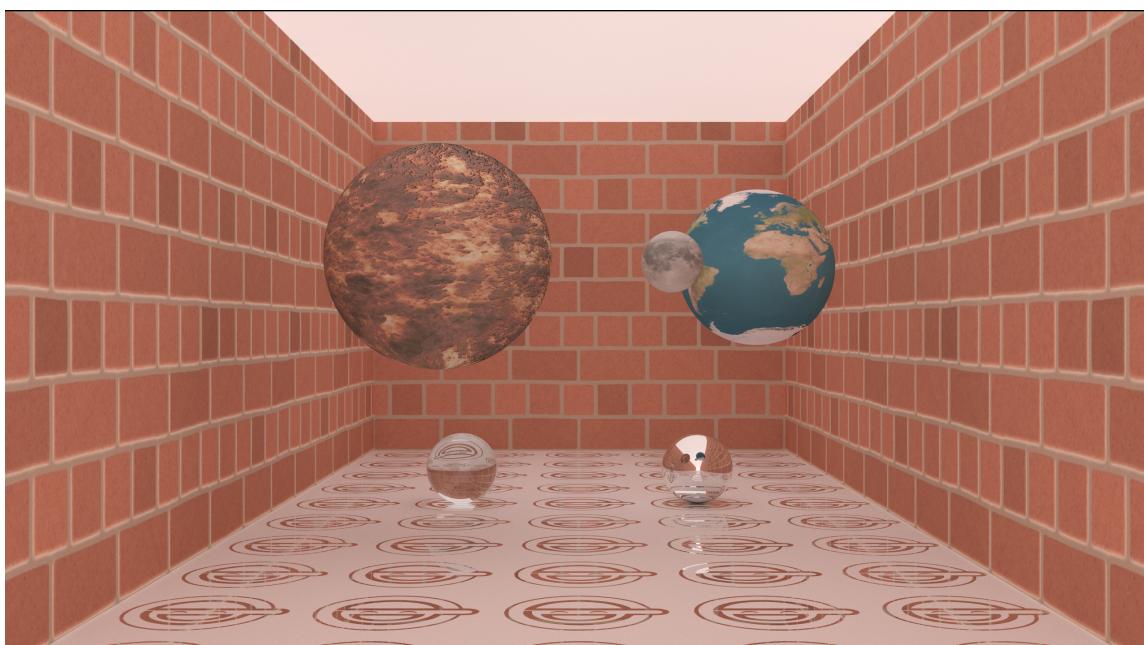
材质主要代码见 `include/material.cuh`

`include/plane.cuh` 和 `include/sphere.cuh` 中有相关的映射逻辑（计算uv坐标），均在对应的 `intersect` 函数中

渲染效果

1. 球面的颜色及凹凸贴图、平面凹凸贴图（两侧砖墙）、表面材质类型贴图（地面）

2560x1440 每个像素采样800次 664秒



2. 景深，三个100k龙模型（材质分别为带颜色的折射、带颜色的反射、漫反射）

2560x1440 每个像素采样800次 9169秒



三、参数曲面解析法求交

参数曲线基于PA3，实现了Bspline曲线和Bezier曲线，在此基础上增设旋转曲面，将xy平面上的曲线绕y轴旋转以形成参数曲面。光线与参数曲面的求交采用牛顿迭代法进行，对参数曲面的参数 $u \in [0, 1]$, $\theta \in [0, 2\pi]$ 与光线的参数 $t \in [t_{min}, \infty]$ 三个参数按照习题课2上的公式进行迭代。

其中光线参数 t 的初值可由参数曲面的包围盒与光线的相交测试得出，而 u, θ 的初值采用暴力撒点的方法给出，实现中通过一定次数的尝试，每次随机生成定义域内的 u, θ 值并开始迭代，在迭代收敛、参数超出定义域或是超出最大迭代次数后退出迭代，并取所有相交结果中 t 最小的作为最终与参数曲面的相交结果。

在展示时拿出的参数曲面结果中，每隔一些像素就会出现一次光线交不上的情况，且每次渲染都是一样的结果，检查求交过程却找不到问题，卡了很久，最后想到自己用的是自己写的伪随机算法，且随机数种子是和像素位置相关的，后修改成curand后果然解决了问题。教训是随机算法的随机性还是十分重要的。

相关代码见 `include/curve.cuh` （两种参数曲线）及

`include/revsurface.cuh`，`revsurface` 的 `intersect` 包含了解析法求交的过程

渲染效果

金色全反射酒杯

1920x1080 每个像素采样1000次 9653秒



四、GPU并行加速

由于串行的版本中每个像素是独立渲染的，具有天然的可并行性，于是使用cuda为每个像素发起一个thread进行渲染。在将串行程序改为cuda版本时主要遇到了以下困难

- cuda不允许深递归，而原本的pt算法以及各种树的遍历都写成了递归的。采用解决办法是，对于pt算法，由于其层数不深，直接用函数模板在编译时展开递归，对于树遍历则改成非递归版本
- 沿用前几次PA中的框架，CPU端创建的多态特性无法很好地继承到GPU（虚函数表丢失），解决办法是直接在GPU端完成物体的构造
- 大量数据结构需要重构，stl与库函数不可用，先前PA中给的vecmath库不可直接使用。这些均需要手动完成移植，因此几乎对所有算法相关的代码进行了重构。由于时间有限完成仓促，所以最后实现的版本存在一些性能问题且可拓展性不佳

而且由于实现的版本中CPU发起所有线程后就基本处于闲置状态，资源浪费很大，因此加速效果也没有特别理想，只到了纯CPU版的几倍水平（测试平台为3700x+RTX2060s，CPU版采用openmp 16线程）

五、光线求交加速

求交的算法加速主要分为两个部分，一是对于场景中的物体，利用AABB包围盒及层次包围盒的形式组织，形成树状结构以减少每次求交时需要计算的物体数目。二是对于mesh，利用kdtree加速求交过程。

AABB BVH

每个物体都有其AABB包围盒，包围盒大小由物体的最小坐标（X,Y,Z均最小）及最大坐标（X,Y,Z均最大）确定，平面较为特殊，本次只用到了平行于坐标平面的平面，采用了指定某个区域为其包围盒的做法，多个物体之间若要确定一个大包围盒，则由所有子包围盒的最小坐标和最大坐标决定。

BVH根据包围盒创建，每个结点包含了一个包围盒与一组物体的id及两个子节点指针，标识该结点的包围盒内的所有物体。首先对于整个场景构造一个大包围盒作为根节点的包围盒，其物体id初始化为包含所有物体。随后通过计算每个子包围盒中心点坐标（归一化后）的MortonCode，按MortonCode为键排序并均匀分割成两部分，两部分包含的物体构成该节点的两个子节点，递归地构造整棵BVH树直到每个叶子结点都是单独的一个物体（BVH仅在CPU上构建一次，因此这部分递归不需要展开）。MortonCode分割的思路参考了[NVIDIA官方提供的BVH构建思路](#)，原理为对点在空间中的位置进行排序并做出间隔距离最大的划分，由于实现的场景中不包含太多物体（不像mesh一样有那么多面片），这样的构造已经足够达成减少不必要求交的目的。

在需要对场景求交时，遍历这棵树，剔除所有包围盒不与光线相交的结点，只对可能与光线相交的物体求交。遍历采用了依赖栈的非递归方法，以便在GPU上进行。

相关代码见 `include/AABB.cuh` `include/bvh.cuh`

`include/objects.cuh` 的 `intersect` 函数中包含了调用bvh求交的过程

KD-Tree

KD-Tree加速仅针对mesh展开，每个包含mesh的物体都包含一棵kd-tree，kd-tree实现了对于一条光线，判断其可能与哪些面片相交的接口，以最小化每次与mesh求交需要遍历的面片数。

kd-tree的节点定义为包含一个包围盒和两个子结点指针，同时留出一个指针用于保存叶结点包围盒内的所有面片id。构造过程为读入一个obj文件并保存所有三角面片，从根节点开始，先构造一个包裹所有面片的包围盒并保存，再从X轴开始，树的每层轮流选择X,Y,Z中的一个维度进行划分，每次划分都让该维度上分割点两侧的面片数尽量一样多，这两部分分别构成两个子节点，递归构造直到达到深度上限或者每个结点的面片数小于某个阈值。一个面片如果穿过了分割点，则同时被包含在两个结点中。叶子结点中用一个数组保存该节点内的所有面片编号，内部结点不保存这个信息，因为面片求交最终只在叶子结点上进行。树的构造仅在创建场景时完成一遍，故可以在CPU上进行。

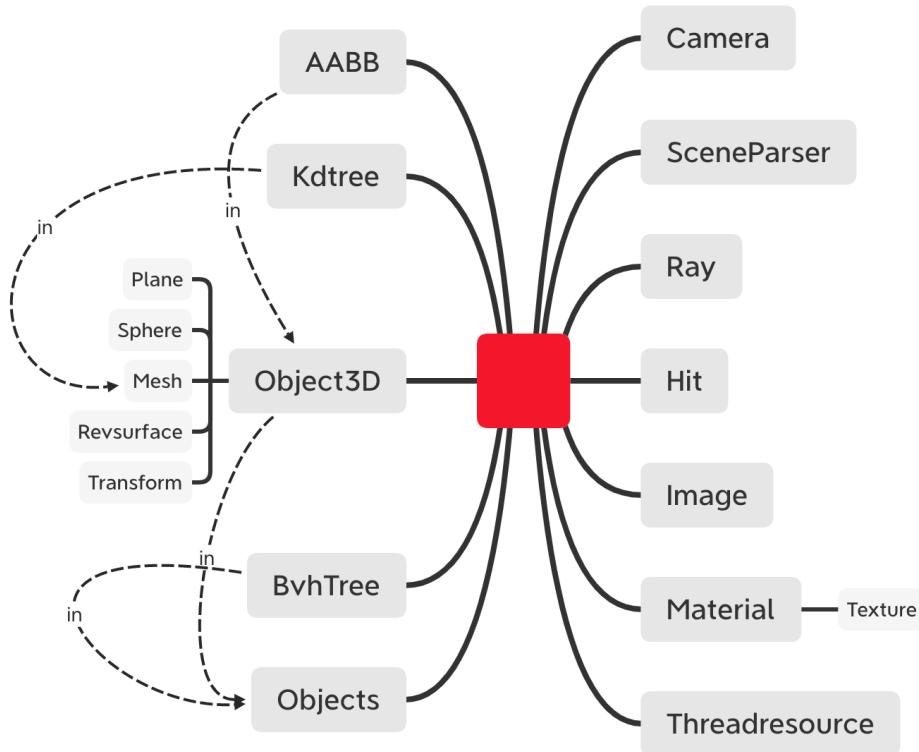
在需要对mesh求交时，遍历这棵树，剔除所有包围盒不与光线相交的结点，最终选出那些包围盒与光线相交的叶子结点，用它们包含的所有面片与光线求交。遍历采用了依赖栈的非递归方法，以便在GPU上进行。

相关代码见 `include/AABB.cuh` `include/kd-tree.cuh`

`mesh.cuh` 的 `intersect` 函数中包含了调用kd-tree求交的过程

六、代码结构

主要类结构如下，用到的数学类基于Vecmath，此处不列出。每个类的完整代码均在同名头文件中



- 其中右侧的类功能基本与前几次PA中相同，Material类中增加了Texture类以满足贴图

Texture类的主要功能是根据uv坐标返回一个包含所有渲染需要的材质信息的数据结构，该接口定义如下

```
1  __device__ void getMaterialAt(const float u, const float v,
2      MaterialFeature& ft) {
3      if (u < 0 || u > 1 || v < 0 || v > 1) {
4          // debug log
5          printf("Incorrect u,v: %f %f\n", u, v);
6          return;
7      }
8      int mapped_x = (int)(u * width);
9      int mapped_y = (int)(v * height);
10     int index = Image::index(mapped_x, mapped_y, width,
11         height);
12     if (emission_map) {
13         ft.emission = emission_map[index];
14     }
15     if (color_map) {
16         ft.color = color_map[index];
17     }
18     if (type_map) {
19         ft.type = type_map[index];
20     }
21 }
```

```

18     }
19     if (bump_map) {
20         ft.gradientU = bump_map[index].y;
21         ft.gradientV = bump_map[index].z;
22     }
23 }
```

- Threadresource类用来储存一些多线程的必要资源，此处可以忽略
- Camera类增加了景深相关的参数

增加的参数为焦距 `flength` 与光圈大小 `aperture`

Camera类保存了相机位置与图片大小信息，其最主要的功能是产生从某个像素点上发出的光线，接口定义如下

```

1 __device__ Ray getRay(const int x, const int y, const int sx,
2 const int sy, uint* Xi) const {
3     double rp1 = 2 * rand(Xi),
4             dx = rp1 < 1 ? sqrt(rp1) - 1 : 1 - sqrt(2 - rp1);
5     double rp2 = 2 * rand(Xi),
6             dy = rp2 < 1 ? sqrt(rp2) - 1 : 1 - sqrt(2 - rp2);
7     Vector3f d = cx * (((sx + .5 + dx) / 2 + x) / w - .5) +
8                 cy * (((sy + .5 + dy) / 2 + y) / h - .5) +
9                 getDirection();
10
11    if (flength == 0) {
12        return Ray(getOrigin(), d);
13    }
14    else {
15        d = normalize(d) * flength;
16        double rand1 = rand(Xi) * 2 - 1.;
17        double rand2 = rand(Xi) * 2 - 1.;
18
19        Vector3f v1 = r1 * rand1 * aperture;
20        Vector3f v2 = r2 * rand2 * aperture;
21
22        Vector3f sp = getOrigin() + v1 + v2;
23        Vector3f fp = d + getOrigin();
24        d = fp - sp;
25        return Ray(sp, d);
26    }
27}
```

- AABB类即为AABB包围盒，被应用在物体、bvhtree和kdtree上

其实现了与光线求交、与三角形面片求交（用于构建kdtree）的方法

- KdTree作为Mesh的成员变量，用来加速Mesh求交

其求交逻辑主要如下，利用一个栈来实现非递归的树遍历，其中mark数组用来记录哪些面片已经求过交，以避免重复的求交

```

1      __device__ bool intersect(const Ray& r, Hit& h, float
2          tmin, ThreadResource* thread_resource) const {
3          BoolBitField *mark = thread_resource->repeat_mark;
// pre allocated
4          int mark_size = thread_resource->mark_size;
5          memset(mark, 0, sizeof(BoolBitField) * mark_size);
6
7          bool result = false;
8          KdNode* stack[MAX_KDTREE_DEPTH * 2];
9          KdNode** stackPtr = stack;
10         *(stackPtr++) = nullptr;
11
12         if (!mainNode->getLeftChild() && !mainNode-
13             >getRightChild()) {
14             intersect(r, h, tmin, mark, mark_size, mainNode,
15             result);
16         }
17
18         KdNode* node = mainNode;
19         do {
20             KdNode* lchild = node->getLeftChild();
21             KdNode* rchild = node->getRightChild();
22             bool lcNull = (lchild == nullptr);
23             bool rcNull = (rchild == nullptr);
24
25             bool intersectL = lcNull ? false : lchild-
26             >getBox().isIntersect(r);
27             bool intersectR = rcNull ? false : rchild-
28             >getBox().isIntersect(r);
29
30             if (intersectL && lchild->isLeaf()) {
31                 intersect(r, h, tmin, mark, mark_size,
32                 lchild, result);
33             }
34
35             if (intersectR && rchild->isLeaf()) {
36                 intersect(r, h, tmin, mark, mark_size,
37                 rchild, result);
38             }
39
40             bool traverseL = (intersectL && !lchild-
41             >isLeaf());
42             bool traverseR = (intersectR && !rchild-
43             >isLeaf());
44
45             if (!traverseL && !traverseR) {
46                 node = *(--stackPtr); //pop
47             }
48             else {

```

```
40         node = (traverseL) ? lchild : rchild;
41         if (traverseL && traverseR) {
42             *(stackPtr++) = rchild; //push
43         }
44     }
45 }
46 while (node != nullptr);
47 return result;
48 }
```

- 所有物体都是虚基类Object3D的派生类，Object3D要求物体拥有AABB包围盒与材质的成员变量，并要求物体必须实现统一的求交接口
- Objects类负责保存场景中的所有物体，控制其创建与回收，并负责在初始化场景时构建一棵bvhtree并保存，用于后续求交。在光追算法中对场景中物体求交时，仅调用Objects的bvhtree的求交接口