作业6

代码框架分析
global.hpp
clamp()
solveQuadratic()
get_random_float()
UpdateProgress()
Vector.hpp
Light.hpp
Object.hpp
Triangle.hpp
Sphere.hpp
AreaLight.hpp
Bounds3.hpp
BVH.hpp与BVH.cpp
基础部分
提高部分

代码框架分析

global.hpp

clamp()

限制值v在[lo, hi]范围内

solveQuadratic()

根据传入的一元二次方程未知数参数a，b，c 求出该方程的解，并返回bool值判断方程是否有解

get_random_float()

提供[0,1]的平均分布的随机数

UpdateProgress()

显示当前进程完成率

Vector.hpp

没有使用Eigen库的Vector3f和Vector2f，而是自定义了其相关实现和操作运算符

Light.hpp

定义了Light类，其成员变量为Vector3f类型的position和Vector3f类型的intensity

Object.hpp

定义了Object基类，其成员函数包括检测相交 intersect() 函数，得到交点相关信息 getIntersection() 函数，获得三角形表面属性 getSurfaceProperties() 函数，计算漫反射颜色 evalDiffuseColor() 函数，得到目前物体所处包围盒 getBounds() 函数

Triangle.hpp

- 定义了继承于Object类的MeshTriangle类，有初始化MeshTriangle()函数，该函数将该三角形所在的物体划分到一个包围盒中、intersect()函数、获得三角形所在包围盒getBounds()函数、得到三角形表面属性函数getSurfaceProperties()函数、估计漫反射颜色evalDiffuseColor()函数、获得光线与BVH交点信息getIntersection()函数
- 另外一个Triangle类也是类似的，不过其getBounds()函数返回的是这个三角形的包围盒，getIntersection()函数返回的是光线与三角形的交点信息

Sphere.hpp

定义了继承于Object类的Sphere类

AreaLight.hpp

定义了区域光的相关属性

Bounds3.hpp

定义了包围盒的相关属性及操作

BVH.hpp与BVH.cpp

定义了包围盒

基础部分

```
for (uint32_t j = 0; j < scene.height; ++j) {
        for (uint32_t i = 0; i < scene.width; ++i) {
            // generate primary ray direction
            float x = (2 * (i + 0.5) / (float)scene.width - 1) *
                      imageAspectRatio * scale;
            float y = (1 - 2 * (j + 0.5) / (float)scene.height) * scale;
            // TODO: Find the x and y positions of the current pixel to get the
            // direction
            //  vector that passes through it.
            // Also, don't forget to multiply both of them with the variable
            // *scale*, and x (horizontal) variable with the *imageAspectRatio*

            // Don't forget to normalize this direction!
            Ray ray = Ray(eye_pos, normalize(Vector3f(x, y, -1)));
            framebuffer[m++] = scene.castRay(ray, 0);

        }
        UpdateProgress(j / (float)scene.height);
    }
```

- 调整函数调用即可

```
inline Intersection Triangle::getIntersection(Ray ray)
{
    Intersection inter;

    if (dotProduct(ray.direction, normal) > 0)
        return inter;
    double u, v, t_tmp = 0;
```

```
    Vector3f pvec = crossProduct(ray.direction, e2);
    double det = dotProduct(e1, pvec);
    if (fabs(det) < EPSILON)
        return inter;

    double det_inv = 1. / det;
    Vector3f tvec = ray.origin - v0;
    u = dotProduct(tvec, pvec) * det_inv;
    if (u < 0 || u > 1)
        return inter;
    Vector3f qvec = crossProduct(tvec, e1);
    v = dotProduct(ray.direction, qvec) * det_inv;
    if (v < 0 || u + v > 1)
        return inter;
    t_tmp = dotProduct(e2, qvec) * det_inv;

    // TODO find ray triangle intersection

    if (t_tmp < 0)
        return inter;
    inter.distance = t_tmp;//光线经过的时间
    inter.happened = true;//是否与三角形相交
    inter.m = m;//三角形的材质
    inter.obj = this;
    inter.normal = normal;//三角形面的法线
    inter.coords = ray(t_tmp);
    return inter;
}
```

- 补全返回相交相关信息代码即可

```
inline bool Bounds3::IntersectP(const Ray& ray, const Vector3f& invDir,
                                const std::array<int, 3>& dirIsNeg) const
{
    // invDir: ray direction(x,y,z), invDir=(1.0/x,1.0/y,1.0/z), use this because Multiply is faster that Division
    // dirIsNeg: ray direction(x,y,z), dirIsNeg=[int(x>0),int(y>0),int(z>0)], use this to simplify your logic
    // TODO test if ray bound intersects
    //计算进入x、y、z截面的最早和最晚时间

    float min_x = (pMin.x - ray.origin.x) * invDir[0];
    float max_x = (pMax.x - ray.origin.x) * invDir[0];

    float min_y = (pMin.y - ray.origin.y) * invDir[1];
    float max_y = (pMax.y - ray.origin.y) * invDir[1];

    float min_z = (pMin.z - ray.origin.z) * invDir[2];
    float max_z = (pMax.z - ray.origin.z) * invDir[2];

    //如果方向为负（反向），就交换最早和最晚时间
    if (dirIsNeg[0])
    {
        std::swap(min_x, max_x);
    }

    if (dirIsNeg[1])
    {
        std::swap(min_y, max_y);
    }

    if (dirIsNeg[2])
    {
        std::swap(min_z, max_z);
    }

    float enter = std::max({min_x,min_y, min_z});
    float exit = std::min({max_x,max_y, max_z});

    if (enter < exit && exit > __FLT_EPSILON__)
    {
        return true;
    }
```

```
    return false;
}
```

- 按照老师在课堂上讲授的代码逻辑来写即可，需要注意的是，由于浮点数无法判断是否与0相等，因此笔者将 `exit >=0` 修改为 `exit>__FLT_EPSILON__`

```
Intersection BVHAccel::getIntersection(BVHBuildNode* node, const Ray& ray) const
{
    // TODO Traverse the BVH to find intersection
    Vector3f invDir(1.f / ray.direction.x, 1.f / ray.direction.y, 1.f / ray.direction.z);

    std::array<int, 3> dirIsNeg;
    dirIsNeg[0] = ray.direction.x<0;
    dirIsNeg[1] = ray.direction.y<0;
    dirIsNeg[2] = ray.direction.z<0;

    //若光线没有与包围盒相交，返回空
    if (!node->bounds.IntersectP(ray, invDir, dirIsNeg))
    {
        return {};
    }

    //若包围盒为叶节点，测试包围盒内的该物体是否与光线相交
    if (node->left == nullptr && node->right == nullptr)
    {
        return node->object->getIntersection(ray);
    }

    //若包围盒为中间节点，则继续递归判断
    Intersection leftChild = BVHAccel::getIntersection(node->left, ray);
    Intersection rightChild = BVHAccel::getIntersection(node->right, ray);
    return leftChild.distance < rightChild.distance ? leftChild : rightChild;
}
```

- 最后判断光线是否与包围盒相交，否则递归下去

提高部分

```
BVHBuildNode* BVHAccel::recursiveBuild(std::vector<Object*> objects)
{
    BVHBuildNode* node = new BVHBuildNode();

    // Compute bounds of all primitives in BVH node
    Bounds3 bounds;
    for (int i = 0; i < objects.size(); ++i)
        bounds = Union(bounds, objects[i]->getBounds());
    if (objects.size() == 1) {
        // Create leaf _BVHBuildNode_
        node->bounds = objects[0]->getBounds();
        node->object = objects[0];
        node->left = nullptr;
        node->right = nullptr;
        return node;
    }
    else if (objects.size() == 2) {
        node->left = recursiveBuild(std::vector{objects[0]});
        node->right = recursiveBuild(std::vector{objects[1]});

        node->bounds = Union(node->left->bounds, node->right->bounds);
        return node;
    }
    else {
        Bounds3 centroidBounds;
        for (int i = 0; i < objects.size(); ++i)
            centroidBounds =
                Union(centroidBounds, objects[i]->getBounds().Centroid());
```

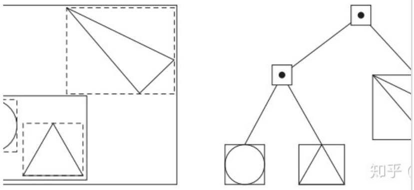
```
        float SN = centroidBounds.SurfaceArea();
        int B = 10;
        int minCostIndex = 0;
        float minCost = std::numeric_limits<float>::infinity();
        int dim = centroidBounds.maxExtent();
        switch (dim) {
        case 0:
            std::sort(objects.begin(), objects.end(), [](auto f1, auto f2) {
                return f1->getBounds().Centroid().x <
                       f2->getBounds().Centroid().x;
            });
            break;
        case 1:
            std::sort(objects.begin(), objects.end(), [](auto f1, auto f2) {
                return f1->getBounds().Centroid().y <
                       f2->getBounds().Centroid().y;
            });
            break;
        case 2:
            std::sort(objects.begin(), objects.end(), [](auto f1, auto f2) {
                return f1->getBounds().Centroid().z <
                       f2->getBounds().Centroid().z;
            });
            break;
        }

        for (int i = 1; i < B; i++)
        {
            auto beginning = objects.begin();
            auto middling = objects.begin() + (objects.size()*i/B);
            auto ending = objects.end();
            auto leftshapes = std::vector<Object*>(beginning, middling);
            auto rightshapes = std::vector<Object*>(middling, ending);
            Bounds3 leftBounds, rightBounds;
            for (int j = 0; j < leftshapes.size(); j++)
                leftBounds = Union(leftBounds,leftshapes[j]->getBounds().Centroid());
            for (int j = 0; j < rightshapes.size(); j++)
                rightBounds = Union(rightBounds,rightshapes[j]->getBounds().Centroid());
            float SA = leftBounds.SurfaceArea();
            float SB = rightBounds.SurfaceArea();
            float cost = 0.125f + (leftshapes.size()*SA+rightshapes.size()*SB)/SN;
            if(cost < minCost)
            {
                minCost = cost;
                minCostIndex = i;
            }
        }

        auto beginning = objects.begin();
        auto middling = objects.begin() + (objects.size() * minCostIndex / B);
        auto ending = objects.end();
        auto leftshapes = std::vector<Object *>(beginning, middling);
        auto rightshapes = std::vector<Object *>(middling, ending);

        assert(objects.size() == (leftshapes.size() + rightshapes.size()));

        node->left = recursiveBuild(leftshapes);
        node->right = recursiveBuild(rightshapes);

        node->bounds = Union(node->left->bounds, node->right->bounds);
    }

    return node;
}
```

- 算法介绍见

PBRT-E4.3-层次包围体(BVH)（一）

层次包围体（Bounding volume hierarchies, BVH） 是一种基于图元（Primitive，构成场景的基本元素，如三角形、球面等）划分的空间索引结构。说它是基于图元的，是因为它是将场景中的图元划分成不相交集的层次结构（Hierarchy of disjoint sets）；与之相对的基

知 https://zhuanlan.zhihu.com/p/50720158



至此，作业6完成！