Assignment 3 : Realistic Rendering with Ray Tracing

NAME: DONGHAO ZHAO
STUDENT NUMBER: 20096438
EMAIL: ZHAODH@SHANGHAITECH.EDU.CN

1 INTRODUCTION

In this assignment, meshes are rendered more realistically within the framework of ray tracing. To have realistic rendering, more general lighting distributions are considered, where area light sources with indirection illumination from many reflected surfaces are taken into account. Ray tracing starts by shooting rays from the camera imaging plane, and back-trace these rays to accumulate all possible light energy to form the pixel intensity and color. Since ray tracing tries to back-trace all light rays in space, the final formed image is much more realistic than the one rendered by OpenGL.

2 IMPLEMENTATION DETAILS

Camera Model and Ray Generation

We can back-trace these rays starting from the imaging plane and through each pixel. These back traced rays will hit or not hit any objects in space. If the ray hits an object, it will be reflected and refracted about the surface normal of the hit (intersection) point. Light energy will be accumulated along all the light ray paths.
A camera can be specified by its imaging center. The viewport is further discretized into pixels with a certain resolution to form a digital image.To determine the camera coordinate system which can be located either at the imaging center or the focal point, we can define an up direction vector, and use the orthogonalization process described in the course to calculate the camera coordinate system.
We use the approach of sampling sufficient rays within a pixel, so anti-aliasing is automatically involved.

Ray-geometry Intersection

We need to determine the intersection point of a ray with a geometry, then we can get the surface normal, texture coordinate etc. at that point for further rendering. The algorithms of solving intersection point of sphere, triangle etc. are complete already, so the code of getIntersection is just at the end.

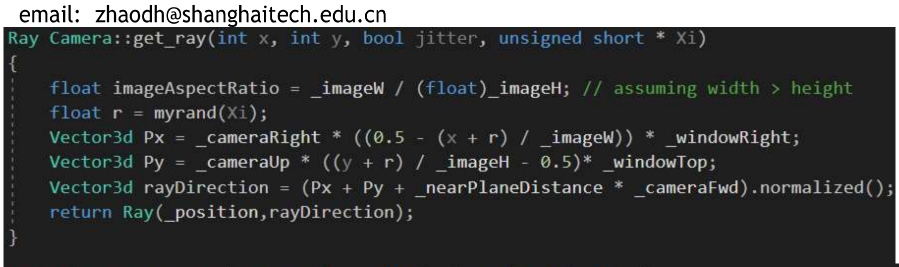
Bounding Box and KD Tree

To render the image, we cast a ray from the camera point through each pixel of the image, and test that ray for intersections with the scene objects. The object is located in the center of the frame, but rays cast through the corner and edge pixels will still check for intersections with the object, despite not being anywhere near it. We can improve this case by implementing simple bounding boxes. We give the object a bounding box slightly bigger than the objects largest dimensions, and then all the rays we cast initially only have to test for intersection with the rectangular planes of the bounding box, which is much simpler than testing for intersections with every one of the thousands of triangles.
If a ray does intersect with the bounding box, we again have to test for intersection with each of the thousands of triangles. It would be better if we could limit the number of possible triangles we have to test. We can do this by using a KD-tree. For my ray tracer, I created a KD tree of bounding boxes. Each node in the tree has a list of pointers to triangles contained within the bounding box, a bounding box that surrounds those triangles, and pointers to child nodes. The KDNode is showed in Fig1.
To build the tree, we start at the root, which contains all the triangles in the object and a bounding box surrounding the whole thing. At each level down the tree, we split on a different axis. We could do it in order X, Y, Z, X, Y, Z, but I did order by longest axis. For each level of the tree, we:
1. Find the midpoint of all the triangles in the node
2. Find the longest axis of the bounding box for that node
3. For each triangle in the node, check whether, for the current axis, it is less than or greater than the overall midpoint
if less than, push the triangle to the left child
if greater than, push the triangle to the right child
Now, this is almost enough to create the tree. We just need to tell the tree when to stop subdividing. We could subdivide all the way down to one triangle per bounding box/node, but that is unnecessary. There are various metrics that can be used to determine when to stop subdividing. I decided to just stop when 50 percents of the triangles in each child match. The ::build function is showed in Fig2.
Hit function is altered to use this data structure. Hit function is recursive we first check the root node for intersections with the bounding box, and then recurse down the structure until we reach a leaf, where we test the ray for intersection with all of the leafs triangles. The ::hit function is showed in Fig3.

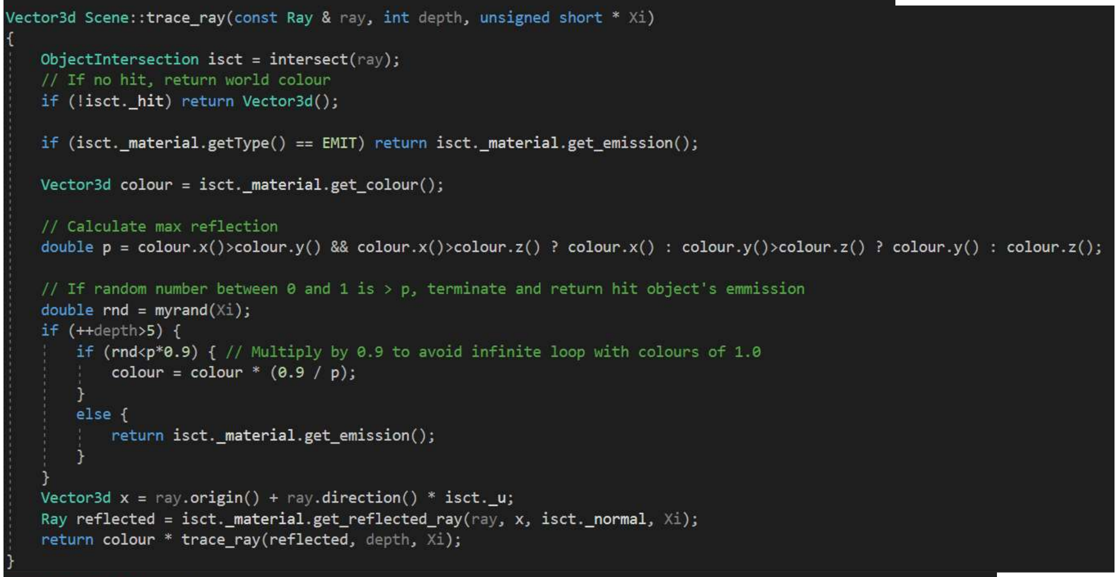
3 RESULTS

I download another obj file which is simpler and render it.

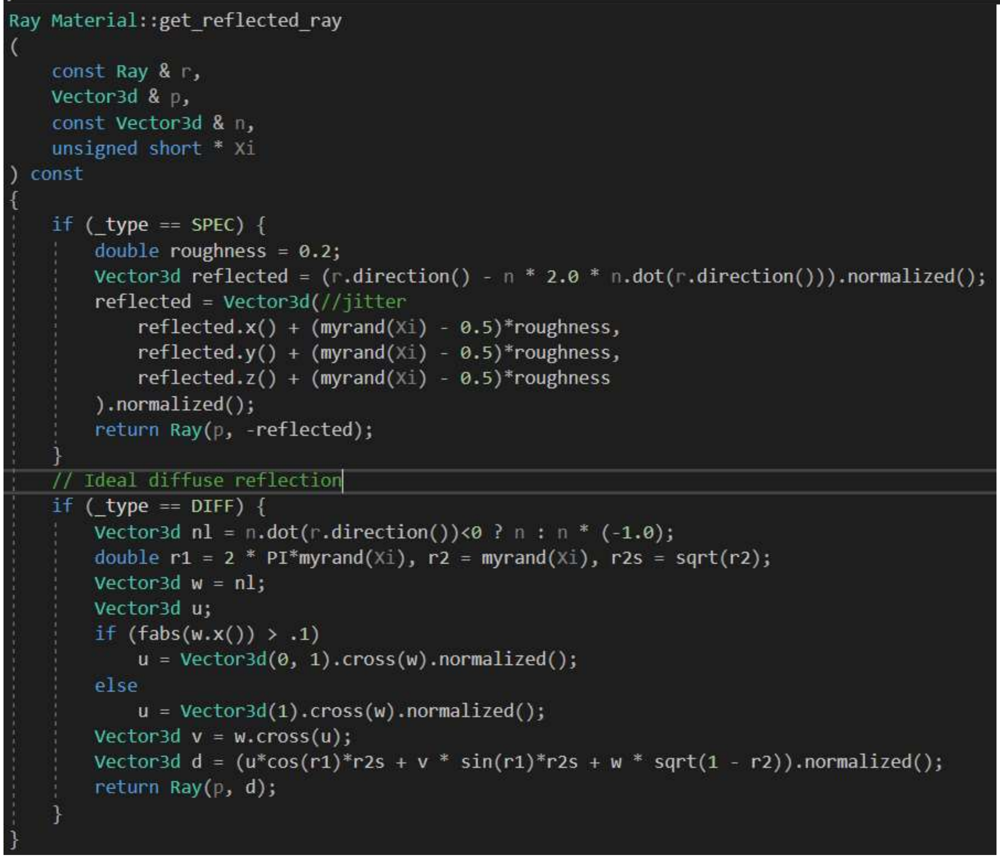
1:2 • Name: Donghao Zhao
student number: 20096438
email: zhaodh@shanghaitech.edu.cn



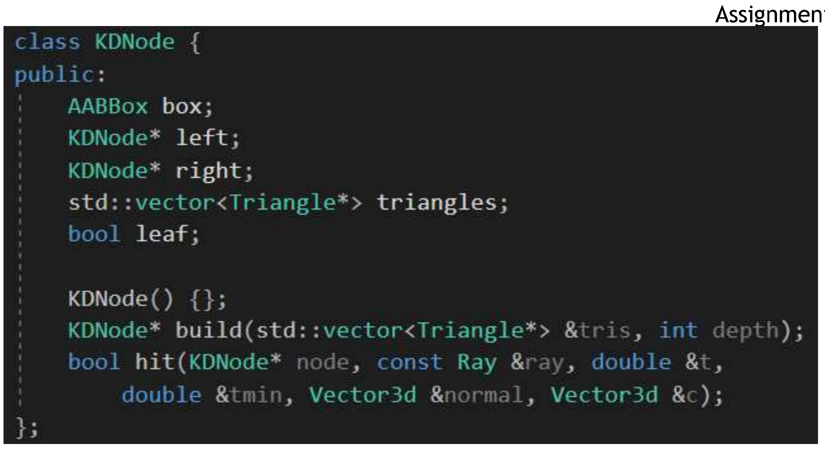
```
Ray Camera::get_ray(int x, int y, bool jitter, unsigned short * Xi)
{
    float imageAspectRatio = _imageW / (float)_imageH; // assuming width > height
    float r = myrand(Xi);
    Vector3d Px = _cameraRight * ((0.5 - (x + r) / _imageW)) * _windowRight;
    Vector3d Py = _cameraUp * ((y + r) / _imageH - 0.5)* _windowTop;
    Vector3d rayDirection = (Px + Py + _nearPlaneDistance * _cameraFwd).normalized();
    return Ray(_position,rayDirection);
}
```



```
Vector3d Scene::trace_ray(const Ray & ray, int depth, unsigned short * Xi)
{
    ObjectIntersection isct = intersect(ray);
    // If no hit, return world colour
    if (!isct._hit) return Vector3d();

    if (isct._material.getType() == EMIT) return isct._material.get_emission();

    Vector3d colour = isct._material.get_colour();

    // Calculate max reflection
    double p = colour.x()>colour.y() && colour.x()>colour.z() ? colour.x() : colour.y()>colour.z() ? colour.y() : colour.z();

    // If random number between 0 and 1 is > p, terminate and return hit object's emmission
    double rnd = myrand(Xi);
    if (++depth>5) {
        if (rnd<p*0.9) { // Multiply by 0.9 to avoid infinite loop with colours of 1.0
            colour = colour * (0.9 / p);
        }
        else {
            return isct._material.get_emission();
        }
    }
    Vector3d x = ray.origin() + ray.direction() * isct._u;
    Ray reflected = isct._material.get_reflected_ray(ray, x, isct._normal, Xi);
    return colour * trace_ray(reflected, depth, Xi);
}
```



```
Ray Material::get_reflected_ray
(
    const Ray & r,
    Vector3d & p,
    const Vector3d & n,
    unsigned short * Xi
) const
{
    if (_type == SPEC) {
        double roughness = 0.2;
        Vector3d reflected = (r.direction() - n * 2.0 * n.dot(r.direction())).normalized();
        reflected = Vector3d(//jitter
            reflected.x() + (myrand(Xi) - 0.5)*roughness,
            reflected.y() + (myrand(Xi) - 0.5)*roughness,
            reflected.z() + (myrand(Xi) - 0.5)*roughness
        ).normalized();
        return Ray(p, -reflected);
    }
    // Ideal diffuse reflection
    if (_type == DIFF) {
        Vector3d nl = n.dot(r.direction())<0 ? n : n * (-1.0);
        double r1 = 2 * PI*myrand(Xi), r2 = myrand(Xi), r2s = sqrt(r2);
        Vector3d w = nl;
        Vector3d u;
        if (fabs(w.x()) > .1)
            u = Vector3d(0, 1).cross(w).normalized();
        else
            u = Vector3d(1).cross(w).normalized();
        Vector3d v = w.cross(u);
        Vector3d d = (u*cos(r1)*r2s + v * sin(r1)*r2s + w * sqrt(1 - r2)).normalized();
        return Ray(p, d);
    }
}
```



```
class KDNode {
public:
    AABBox box;
    KDNode* left;
    KDNode* right;
    std::vector<Triangle*> triangles;
    bool leaf;

    KDNode() {};
    KDNode* build(std::vector<Triangle*> &tris, int depth);
    bool hit(KDNode* node, const Ray &ray, double &t,
        double &tmin, Vector3d &normal, Vector3d &c);
};
```