图形学实验报告

因为导出pdf后发现代码段比较混乱，因此将markdown文件一并提交。

# 得分点

路径追踪PT

渐进式光子映射PPM（使用kdtree数据结构）

景深

运动模糊

软阴影

抗锯齿

UV纹理映射

凹凸贴图

参数曲面牛顿迭代法求交

复杂网格模型读取以及法向插值

复杂网格模型层次包围盒的求交加速（使用BVH tree数据结构）

代码

PT

主要代码在pt.hpp里，下面是代码段：大体思路是创建一个Path_Tracer类，函数trace()是实现路径追踪的主要函数，函数getPtColor是每个sample采样得到颜色的函数。

```
#ifndef PT_H
#define PT_H

#include "camera.hpp"
#include "constant.h"
#include "group.hpp"
#include "hit.hpp"
#include "image.hpp"
#include "light.hpp"
#include "ray.hpp"
#include "scene_parser.hpp"
#include "random_producer.hpp"
using namespace std;

Vector3f getPtColor(Ray ray, Group* group, int depth, bool de =
false) {

    Hit hit;
    bool isIntersect = group->intersect(ray, hit, 0, RND2);

    if(isIntersect) {

        Material* material = hit.getMaterial();
        bool is_texture;
        Vector3f hit_color = hit.get_color(is_texture);
        if(!is_texture)
            hit_color = material->color;

        if(material->emission != Vector3f::ZERO)
            return material->emission;

        float p = hit_color.x() > hit_color.y() && hit_color.x()
> hit_color.z() ? hit_color.x() : hit_color.y() > hit_color.z() ?
hit_color.y() : hit_color.z();
        p = p < 0.75 ? p : 0.75;
        if(depth > pt_max_depth) {
            if(RND2 < p) { //轮盘赌决定是否终止
                hit_color = hit_color / p;
            } else {
                return material->emission;
            }
        }
        Vector3f hit_emission = material->emission;
        Vector3f hit_normal = hit.getNormal().normalized();
        Vector3f next_origin = ray.getOrigin() + hit.getT() *
ray.getDirection();
        Vector3f ray_direction = ray.getDirection().normalized();

        float type_decision = RND2; //轮盘赌决定表面材质种类
        float b = Vector3f::dot(ray_direction, hit_normal);
        if(type_decision < material->type.x()) {//漫反射
            Vector3f z_ = Vector3f::cross(ray_direction,
hit_normal);
            Vector3f x_ = Vector3f::cross(z_, hit_normal);
            z_.normalize();
            x_.normalize();
            Vector3f next_direction;
            if(b < 0)
                next_direction = RND1 * z_ + RND1 * x_ + RND2 *
hit_normal;
            else
                next_direction = RND1 * z_ + RND1 * x_ - RND2 *
hit_normal;
            next_direction.normalize();

            Vector3f next_color = getPtColor(Ray(next_origin,
next_direction), group, depth + 1, de);

            if(de) {

                cout << "depth:" << depth << endl <<
"ru_direct:";
                ray_direction.print();
                cout << "type:";
                material->type.print();
                cout << "matirial color:";
                material->color.print();
                cout << "normal:";
                hit_normal.print();
                cout << "chu_direct:" ;
                next_direction.print();
                cout << "origin:" ;
                next_origin.print();
                cout << "t: " << hit.getT() << endl;
                next_color.print();
            }

            return hit_emission + hit_color * next_color;

        } else if(type_decision < material->type.x() + material-
>type.y()) {//镜面反射
            Vector3f next_direction = ray_direction - hit_normal
* (b * 2);
            next_direction.normalize();
            Vector3f next_color = getPtColor(Ray(next_origin,
next_direction), group, depth + 1, de);
            if(de) {
                cout << "depth:" << depth << endl <<
"ru_direct:";
                ray_direction.print();
                cout << "normal:";
                hit_normal.print();
                cout << "chu_direct:" ;
                next_direction.print();
                next_origin.print();
                next_color.print();
            }
            return hit_emission + hit_color * next_color;

        } else {//折射
            float n = material->refractive_index;
            float R0 = ((1.0 -n) * (1.0 - n)) / ((1.0 + n) * (1.0
+ n));
            if(b > 0) {
                hit_normal.negate();
            } else {
                n = 1.0 / n;
            }

            float cos1 = -Vector3f::dot(hit_normal,
ray_direction);
            float cos2 = 1.0 - n * n * (1.0 - cos1 * cos1);
//cos(theta2)的平方
            Vector3f reflect = (ray_direction + hit_normal *
(cos1 * 2));
```

```
            if(cos2 < 0) {
                return hit_emission + hit_color *
getPtColor(Ray(next_origin, reflect), group, depth + 1, de);
            }
            //Schlick估计菲涅尔项
            float Rprob = R0 + (1.0 - R0) * pow(1.0 - cos1, 5.0);

            Vector3f refrac =  ((ray_direction * n) + (hit_normal
* (n * cos1 - sqrt(cos2)))).normalized();


            float P = 0.25 + 0.5 * Rprob;
            if (depth <= 1)
                return hit_emission + hit_color * (Rprob *
getPtColor(Ray(next_origin, reflect), group, depth + 1, de) + (1
- Rprob) * getPtColor(Ray(next_origin, refrac), group, depth + 1,
de));

            if (RND2 < P) { //轮盘赌决定反射和折射的占比
                return hit_emission + hit_color * ((Rprob / P) *
getPtColor(Ray(next_origin, reflect), group, depth + 1, de));
            } else {
                return hit_emission + hit_color * (((1 - Rprob) /
(1 - P)) * getPtColor(Ray(next_origin, refrac), group, depth + 1,
de));
            }
        }
    }
    else {

        return Vector3f::ZERO; //递归终止，返回背景色，方便起见统一设为
黑色
    }
}

class Path_Tracer {
    const SceneParser& scene;
    const char* output_file;
    bool debug;
    public:
    Path_Tracer(const SceneParser& scene, const char* output,
bool debuger = false):scene(scene), output_file(output),
debug(debuger){
    };

    void trace() {
        Camera *camera = scene.getCamera();
        int w = camera->getWidth();
        int h = camera->getHeight();
        Image myImage(w, h);
        Vector3f background = scene.getBackgroundColor();
        Group* group = scene.getGroup();
        for (int x = 0; x < w; ++x) {
            for (int y = 0; y < h; ++y) {

                Vector3f finalColor = Vector3f::ZERO;
                for (int s = 0; s < pt_samples; s++) {

                    Vector2f ori = Vector2f(x + RND1 / 2, y +
RND1 / 2);
                    Ray camRay = camera->generateRay(ori);

                    // if(s == 2) {
                    //     cout << "<<<<<<<<<<<<<<" << endl;
                    finalColor += getPtColor(camRay, group, 0,
debug);

                }
                finalColor = finalColor / float(pt_samples);
                finalColor = Vector3f(toFloat(finalColor.x()),
toFloat(finalColor.y()), toFloat(finalColor.z()));
                // cout << "finalcolor: ";
                // finalColor.print();
                myImage.SetPixel(x, y, finalColor);



            }
            if (x % 5 == 0)
                cout << "Finished:" << (float(x) * 100) / w <<
"%" << endl;
        }
         myImage.SaveImage(output_file);

    }
```

```
};

#endif
```

PPM

主要代码在ppm.hpp里，光子相关的kdtree则在photon.hpp里。下面是代码段，代码解释见注释：

```
#ifndef PPM_H
#define PPM_H

#include "camera.hpp"
#include "constant.h"
#include "group.hpp"
#include "hit.hpp"
#include "image.hpp"
#include "light.hpp"
#include "ray.hpp"
#include "scene_parser.hpp"
#include "random_producer.hpp"
#include "photon.hpp"
using namespace std;

struct ViewRecord { //用于记录每个像素的漫反射面观察点，以及观察点收集的光子数、能量等
    Vector3f direction;
    Vector3f position;
    Vector3f power_rate;
    Vector3f accumulate_power;
    Vector3f accumulate_photon_power;
    int photon_num;
    float radius;
    int pixel_x;
    int pixel_y;
    ViewRecord(int x, int y, Vector3f pos, float r) {
        pixel_x = x;
        pixel_y = y;
        position = pos;
        radius = r;
        photon_num = 0;
        accumulate_photon_power = Vector3f::ZERO;
    }
};

class PPM { //渐进式光子追踪的主要类
    public:
    const SceneParser& scene;
    const char* output_file;
    Camera* camera;
    int w;
    int h;
    Vector3f background;
    Group* group;

    PPM(const SceneParser& _scene, const char*
output):scene(_scene), output_file(output) {
        camera = scene.getCamera();
        w = camera->getWidth();
        h = camera->getHeight();
        background = scene.getBackgroundColor();
        group = scene.getGroup();
        photon_total = 0;
        caustic_photon_total = 0;
    };

    void trace();
    void pt_and_record_view_point();
    void postprocess();
    void traverse_viewpoint(Photon_KDtree& tree, bool first =
false);
    void caustic();
    void volume_light();

    private:
    int caustic_photon_total;//焦散光子总数
    int photon_total;//实际上总共有效光子数
    std::vector<ViewRecord> view_pointmap;
//      std::unique_ptr<PhotonMap> m_photonMap;
//
};
```

```
void PPM::trace() { //实现PPM的主函数

    Image myImage(this->w, this->h);
    pt_and_record_view_point(); //每个像素从相机发射若干条光线，记录最
终落到哪个漫反射面上


    cout << "View points have been record." << endl;

    postprocess(); //后处理，即光源渐进式地发射光子，并遍历每个观察点收集
光子
    caustic(); //增强焦散效果，发射并收集焦散光子
    Vector3f img[w * h];
    for(int i = 0; i < w * h; ++i)
        img[i] = Vector3f::ZERO;
    for(auto it = view_pointmap.begin(); it !=
view_pointmap.end(); ++it) {
        Vector3f total_photon_power =
(*it).accumulate_photon_power / (M_PI * (*it).radius *
(*it).radius * (caustic_photon_total + photon_total));
        Vector3f color = (*it).accumulate_power +
(*it).power_rate * total_photon_power;
        img[(*it).pixel_y * w + (*it).pixel_x] += color;
    }
    for (int x = 0; x < w; ++x) {
        for (int y = 0; y < h; ++y) {
            Vector3f finalColor = img[y * w + x] / ppm_samples;
            finalColor = Vector3f(toFloat(finalColor.x()),
toFloat(finalColor.y()), toFloat(finalColor.z()));
            myImage.SetPixel(x, y, finalColor);
        }
    }
    myImage.SaveImage(this->output_file);
}

void PPM::pt_and_record_view_point() {
    cout << "begin pt_and_record." << endl;
   for (int x = 0; x < w; ++x) {
        for (int y = 0; y < h; ++y) {
            for (int s = 0; s < ppm_samples; s++) {
                Vector2f ori = Vector2f(x + RND1 / 2, y + RND1 /
2);
```

```
                Ray camRay = camera->generateRay(ori);

                Vector3f new_power = Vector3f(1, 1, 1);
                Vector3f accumulate_power = Vector3f::ZERO;
                int dep = 0;

                while(true) { //和pt类似的求交
                    if(dep > ppm_max_depth)
                        break;
                    else if (new_power.x() < 1e-3 &&
new_power.y() < 1e-3 && new_power.z() < 1e-3)
                        break;

                    dep++;

                    Hit hit;
                    bool isIntersect = group->intersect(camRay,
hit, 0, RND2);


                    if(isIntersect) {
                        Material* material = hit.getMaterial();
                        bool is_texture;
                        Vector3f hit_color =
hit.get_color(is_texture);
                        if(!is_texture)
                            hit_color = material->color;

                        Vector3f hit_emission = material-
>emission;
                        Vector3f hit_normal =
hit.getNormal().normalized();
                        Vector3f next_origin = camRay.getOrigin()
+ hit.getT() * camRay.getDirection();
                        Vector3f ray_direction =
camRay.getDirection().normalized();
                        Vector3f next_direction;

                        for (int li = 0; li <
scene.getNumLights(); li++) {
                            Light* light = scene.getLight(li);
                            Vector3f r = next_origin -light-
>position;
```

```
                        if(r.squaredLength() < ((light-
>radius) * (light->radius))) {
                            auto dot_ = Vector3f::dot(r,
light->get_direction());
                            if(dot_ < 1e-3 && dot_ > -1e-3) {
                                accumulate_power += light-
>emission * new_power;
                                // cout << x << " " << y << "
" << s << endl;
                            }

                        }
                    }


                    float type_decision = RND2;
                    float b = Vector3f::dot(ray_direction,
hit_normal);

                    camRay.origin = next_origin;

                    accumulate_power += material->emission *
new_power;
                    new_power = new_power * hit_color;
                    if(type_decision < material->type.x())
{//漫反射，达到漫反射面即终止，并记录下来。
                        ViewRecord vr(x, y, next_origin,
ini_photon_radius);
                        vr.power_rate = new_power;
                        vr.accumulate_power =
accumulate_power;
                        view_pointmap.push_back(vr);
                        break;
                    } else if(type_decision < material-
>type.x() + material->type.y()) {//镜面反射
                        Vector3f next_direction =
ray_direction - hit_normal * (b * 2);
                        next_direction.normalize();
                        camRay.direction = next_direction;
                    } else  {
                        float n = material->refractive_index;
                        float R0 = ((1.0 -n) * (1.0 - n)) /
((1.0 + n) * (1.0 + n));
```

```
                            if(b > 0) {
                                hit_normal.negate();
                            } else {
                                n = 1.0 / n;
                            }
                            float cos1 = -
Vector3f::dot(hit_normal, ray_direction);
                            float cos2 = 1.0 - n * n * (1.0 -
cos1 * cos1); //cos(theta2)的平方
                            Vector3f reflect = (ray_direction +
hit_normal * (cos1 * 2));
                            //Schlick估计菲涅尔项
                            float Rprob = R0 + (1.0 - R0) *
pow(1.0 - cos1, 5.0);
                            Vector3f refrac =  ((ray_direction *
n) + (hit_normal * (n * cos1 - sqrt(cos2)))).normalized();

                            if (RND2 < Rprob || cos2 <= 0) {
                                camRay.direction = reflect;
                                // cout << x << " " << y << " "
<< s << " rprob:" << Rprob << " fanshe at";
                                // next_origin.print();
                                // camRay.direction.print();
                            } else {
                                camRay.direction = refrac;

                            }
                        }
                    } else {
                        break;
                    }
                }
            }
        }
    }
    cout << "View Finished, totally " << view_pointmap.size() << "
view points." << endl;

}

void PPM::postprocess() {
    int _pass_count = pass_count;
    int per_light_photon_count =  per_sample_photon_count /
scene.getNumLights();
    while(_pass_count--) { //一轮一轮地发射光子
        vector<Photon*> PhotonMap;
        for (int li = 0; li < scene.getNumLights(); li++) {
            for (int i = 0; i < per_light_photon_count; i++) {
                Light* light = scene.getLight(li);
                Vector3f photon_power;
                Ray light_ray = light->get_ray(photon_power);
                int dep = 0;
                // bool is_zheshe = false;
                while(true) {
                    Hit hit;
                    if(dep > ppm_max_depth)
                        break;
                    else if (photon_power.x() < 1e-3 &&
photon_power.y() < 1e-3 && photon_power.z() < 1e-3)
                        break;
                    bool is_intersect = group-
>intersect(light_ray, hit, 0, RND2);
                    dep++;

                    if (is_intersect) {
                        Material* material = hit.getMaterial();
                        bool is_texture;
                        Vector3f hit_color =
hit.get_color(is_texture);
                        if(!is_texture)
                            hit_color = material->color;

                        Vector3f hit_emission = material-
>emission;
                        Vector3f hit_normal =
hit.getNormal().normalized();
                        Vector3f next_origin =
light_ray.getOrigin() + hit.getT() * light_ray.getDirection();
                        Vector3f ray_direction =
light_ray.getDirection().normalized();
                        Vector3f next_direction;

                        float type_decision = RND2;
                        float b = Vector3f::dot(ray_direction,
hit_normal);
```

```
                        light_ray.origin = next_origin;

                        if(type_decision < material->type.x())
{//漫反射
                            Vector3f z_ =
Vector3f::cross(ray_direction, hit_normal);
                            Vector3f x_ = Vector3f::cross(z_,
hit_normal);
                            z_.normalize();
                            x_.normalize();
                            Vector3f next_direction;
                            if(b < 0)
                                next_direction = RND1 * z_ + RND1
* x_ + RND2 * hit_normal;
                            else
                                next_direction = RND1 * z_ + RND1
* x_ - RND2 * hit_normal;
                            next_direction.normalize();
                            light_ray.direction = next_direction;
                            Photon* pho = new Photon(next_origin,
ray_direction, photon_power);//达到漫反射，先记录到光子图中去
                            PhotonMap.push_back(pho);
                            //轮盘赌判断漫反射表面的光子是否被吸收
                            float p = hit_color.x() >
hit_color.y() && hit_color.x() > hit_color.z() ? hit_color.x() :
hit_color.y() > hit_color.z() ? hit_color.y() : hit_color.z();
                            p = p < 0.75 ? p : 0.75;
                            if(RND2 < p) {
                                hit_color = hit_color / p;
                            } else {
                                break;
                            }
                        } else if(type_decision < material-
>type.x() + material->type.y()) {//镜面反射
                            Vector3f next_direction =
ray_direction - hit_normal * (b * 2);
                            next_direction.normalize();
                            light_ray.direction = next_direction;
                        } else  {
                            float n = material->refractive_index;
                            float R0 = ((1.0 -n) * (1.0 - n)) /
((1.0 + n) * (1.0 + n));
```

```
                            if(b > 0) {
                                hit_normal.negate();
                            } else {
                                n = 1.0 / n;
                            }
                            float cos1 = -
Vector3f::dot(hit_normal, ray_direction);
                            float cos2 = 1.0 - n * n * (1.0 -
cos1 * cos1); //cos(theta2)的平方
                            Vector3f reflect = (ray_direction +
hit_normal * (cos1 * 2));
                            //Schlick估计菲涅尔项
                            float Rprob = R0 + (1.0 - R0) *
pow(1.0 - cos1, 5.0);
                            Vector3f refrac =  ((ray_direction *
n) + (hit_normal * (n * cos1 - sqrt(cos2)))).normalized();

                            if (RND2 < Rprob || cos2 <= 0) {
                                light_ray.direction = reflect;
                            } else {
                                light_ray.direction = refrac;
                            }
                        }
                        photon_power = hit_emission + hit_color *
photon_power;
                    } else {//没交，这次光子结束
                        break;
                    }
                }
            if(i % 100000 == 0) {
                cout << _pass_count << "th pass, emit photon: "
<< i << endl;
            }
            }
        }

        photon_total += PhotonMap.size();
        Photon_KDtree tree(PhotonMap.begin(), PhotonMap.end());
//根据光子图构建kdtree
        if(pass_count == _pass_count + 1) //根据kdtree进行近邻搜索，
搜索半径为radius
            traverse_viewpoint(tree, true);
        else
```

```
            traverse_viewpoint(tree);

        cout << "There will be " << _pass_count << " passes" <<
endl;
    }
}

void PPM::traverse_viewpoint(Photon_KDtree& tree, bool first) {
//遍历观察点，收集周围光子
    for (auto it = view_pointmap.begin(); it !=
view_pointmap.end(); ++it) {
        // float next_radius;
        // float
        float next_radius;
        int M;
        Vector3f phi_i(0, 0, 0);
        float itradius = (*it).radius;
        if(first)
            itradius = ini_photon_radius;
        tree.search((*it).position, itradius, M, phi_i);

        if(first) { //第一次搜索，直接把总能量、总数量记录下来
            (*it).photon_num += M;
            (*it).accumulate_photon_power += phi_i;

        }
        else { //后来的搜索，需要根据公式计算出衰减后的半径，下一轮使用衰减
后的半径搜索。
            int ini_num = (*it).photon_num;
            (*it).photon_num += ppm_alpha * M;
            float rate = float((*it).photon_num) / float(ini_num
+ M);

            (*it).radius = (*it).radius * sqrt(rate);
            (*it).accumulate_photon_power += phi_i;
            (*it).accumulate_photon_power *= rate;
        }

    }
}

void PPM::caustic() { //处理焦散光子，即光子图只保留仅反射和折射的落在漫
反射面上的光子
```

```
    int per_light_photon_count =  photon_total / (2 *
scene.getNumLights());

    vector<Photon*> PhotonMap;
    for (int li = 0; li < scene.getNumLights(); li++) {
        for (int i = 0; i < per_light_photon_count; i++) {
            Light* light = scene.getLight(li);
            Vector3f photon_power;
            Ray light_ray = light->get_ray(photon_power);
            int dep = 0;
            bool HasSpecularOrGlossy = false;
            // bool is_zheshe = false;
            while(true) {
                Hit hit;
                if(dep > ppm_max_depth)
                    break;
                else if (photon_power.x() < 1e-3 &&
photon_power.y() < 1e-3 && photon_power.z() < 1e-3)
                    break;
                bool is_intersect = group->intersect(light_ray,
hit, 0, RND2);
                dep++;

                if (is_intersect) {
                    Material* material = hit.getMaterial();
                    bool is_texture;
                    Vector3f hit_color =
hit.get_color(is_texture);
                    if(!is_texture)
                        hit_color = material->color;

                    Vector3f hit_emission = material->emission;
                    Vector3f hit_normal =
hit.getNormal().normalized();
                    Vector3f next_origin = light_ray.getOrigin()
+ hit.getT() * light_ray.getDirection();
                    Vector3f ray_direction =
light_ray.getDirection().normalized();
                    Vector3f next_direction;

                    float type_decision = RND2;
                    float b = Vector3f::dot(ray_direction,
hit_normal);
```

```
                    light_ray.origin = next_origin;

                    if(type_decision < material->type.x()) {//漫反
射
                        if(!HasSpecularOrGlossy)
                            break;
                        Vector3f z_ =
Vector3f::cross(ray_direction, hit_normal);
                        Vector3f x_ = Vector3f::cross(z_,
hit_normal);
                        z_.normalize();
                        x_.normalize();
                        Vector3f next_direction;
                        if(b < 0)
                            next_direction = RND1 * z_ + RND1 *
x_ + RND2 * hit_normal;
                        else
                            next_direction = RND1 * z_ + RND1 *
x_ - RND2 * hit_normal;
                        next_direction.normalize();
                        light_ray.direction = next_direction;
                        Photon* pho = new Photon(next_origin,
ray_direction, photon_power);
                        PhotonMap.push_back(pho);
                        //只记录第一次的光子
                        break;

                    } else if(type_decision < material->type.x()
+ material->type.y()) {//镜面反射
                        Vector3f next_direction = ray_direction -
hit_normal * (b * 2);
                        next_direction.normalize();
                        light_ray.direction = next_direction;
                        HasSpecularOrGlossy = true;
                    } else  {
                        float n = material->refractive_index;
                        float R0 = ((1.0 -n) * (1.0 - n)) / ((1.0
+ n) * (1.0 + n));
                        if(b > 0) {
                            hit_normal.negate();
                        } else {
                            n = 1.0 / n;
                        }
                        float cos1 = -Vector3f::dot(hit_normal,
ray_direction);
                        float cos2 = 1.0 - n * n * (1.0 - cos1 *
cos1); //cos(theta2)的平方
                        Vector3f reflect = (ray_direction +
hit_normal * (cos1 * 2));
                        //Schlick估计菲涅尔项
                        float Rprob = R0 + (1.0 - R0) * pow(1.0 -
cos1, 5.0);
                        Vector3f refrac =  ((ray_direction * n) +
(hit_normal * (n * cos1 - sqrt(cos2)))).normalized();

                        if (RND2 < Rprob || cos2 <= 0) {
                            light_ray.direction = reflect;
                        } else {
                            light_ray.direction = refrac;
                        }
                        HasSpecularOrGlossy = true;
                    }
                    photon_power = hit_emission + hit_color *
photon_power;

                } else {//没交，这次光子结束
                    break;
                }
            }
            if(i % 100000 == 0) {
                cout << "emit caustic photon: " << i << endl;
            }
        }
    }

    caustic_photon_total += PhotonMap.size();
    cout << "emit caustic_photon: " << caustic_photon_total <<
endl;
    Photon_KDtree tree(PhotonMap.begin(), PhotonMap.end());
    traverse_viewpoint(tree, true);

    cout << "Caustic photon finished" << endl;

}
```

```
#endif
```

```
#ifndef PHOTON_H
#define PHOTON_H

#include "camera.hpp"
#include "constant.h"
#include "group.hpp"
#include "hit.hpp"
#include "image.hpp"
#include "light.hpp"
#include "ray.hpp"
#include "scene_parser.hpp"
#include "random_producer.hpp"
#include "photon.hpp"
#include <queue>
#include <algorithm>

using namespace std;

struct Photon { //一个个光子，记录位置、方向、能量
    Vector3f position;
    Vector3f direction;
    Vector3f power;
    Photon(Vector3f _position, Vector3f _direction, Vector3f
_power) {
        position = _position;
        direction = _direction;
        power = _power;
    }
};

struct KDTreeNode
{
    int depth;
    Photon* photon;
    KDTreeNode* lc;
    KDTreeNode* rc;
    KDTreeNode* parent;
    KDTreeNode(KDTreeNode* e) {
        parent = e;
        lc = nullptr;
        rc = nullptr;
        photon = nullptr;
    }
};

Vector3f Pos;

struct cmp
{
     bool operator()(const Photon* a, const Photon*b)
    {
         return (a->position - Pos).squaredLength() < (b-
>position - Pos).squaredLength();
    }
};


bool find_median_x(Photon* pp, Photon* qq) {
    return pp->position.x() < qq->position.x();
}

bool find_median_y(Photon* pp, Photon* qq) {
    return pp->position.y() < qq->position.y();
}

bool find_median_z(Photon* pp, Photon* qq) {
    return pp->position.z() < qq->position.z();
}

bool cmp_x(Vector3f& pp, Vector3f& qq) {
    return pp.x() < qq.x();
}

bool cmp_y(Vector3f& pp, Vector3f& qq) {
    return pp.y() < qq.y();
}

bool cmp_z(Vector3f& pp, Vector3f& qq) {
    return pp.z() < qq.z();
}
```

```
class Photon_KDtree {
    public:
    KDTreeNode* root;
    vector<Photon*>::iterator start;
    vector<Photon*>::iterator end;
    // priority_queue<Photon*, vector<Photon*>, cmp> k_near;
    vector<Photon*> k_near;

    void buildtree(vector<Photon*>::iterator _start,
vector<Photon*>::iterator _end, int depth, KDTreeNode* e);
    void search(Vector3f pos, float& radius, int& M, Vector3f&
phi_i);
    Photon_KDtree(vector<Photon*>::iterator _start,
vector<Photon*>::iterator _end) : start(_start) , end(_end){
        buildtree(start, end, 0, root);
        KDTreeNode* p = root;
    };

    void recurve(KDTreeNode* e, float radius);

};

void Photon_KDtree::buildtree(vector<Photon*>::iterator _start,
vector<Photon*>::iterator _end, int depth, KDTreeNode* e) { //递归
建立kdtree
    int nums = distance(_start, _end);
    e->depth = depth;
    if(nums == 0)
        return;
    int total = nums / 2;
    bool (*cmp_func) (Photon* p1, Photon* p2);
    switch (depth % 3)
    {
    case 0:
        cmp_func = find_median_z;
        break;
    case 1:
        cmp_func = find_median_y;
        break;
    default:
        cmp_func = find_median_x;
        break;
    }
    sort(_start, _end, cmp_func);
    auto it = _start + total;
    e->photon = *it;
    KDTreeNode* e1 = new KDTreeNode(e);
    KDTreeNode* e2 = new KDTreeNode(e);
    e->lc = e1;
    e->rc = e2;
    buildtree(_start, it, depth + 1, e1);
    buildtree(it + 1, _end, depth + 1, e2);
}

//以radius为半径进行搜索
void Photon_KDtree::search(Vector3f pos, float& radius, int& M,
Vector3f& phi_i) {
    k_near.clear();
    Pos = pos;
    recurve(root, radius);
    // float new_radius = (k_near.top()->position -
Pos).length();
    M = (int)k_near.size();
    //叠加power:
    for (auto it = k_near.begin(); it != k_near.end(); ++it)
        phi_i += (*it)->power;

}

void Photon_KDtree::recurve(KDTreeNode* e, float radius) {
    if(!e->photon)
        return;

    float axis_dist;
    bool (*cmp_func) (Vector3f& pp, Vector3f& qq);
    switch (e->depth % 3)
    {
    case 0:
        cmp_func = cmp_z;
        axis_dist = Pos.z() - e->photon->position.z();
        break;
    case 1:
        cmp_func = cmp_y;
        axis_dist = Pos.y() - e->photon->position.y();
```

```
            break;
        default:
            cmp_func = cmp_x;
            axis_dist = Pos.x() - e->photon->position.x();
            break;
        }

        if(cmp_func(Pos, e->photon->position))
            recurve(e->lc, radius);
        else
            recurve(e->rc, radius);

            if((e->photon->position - Pos).squaredLength() < radius *
radius)
                k_near.push_back(e->photon);
            if (radius > abs(axis_dist)) {
                if(axis_dist < 0)
                    recurve(e->rc, radius);
                else
                    recurve(e->lc, radius);
            }

        // }
}

#endif
```

景深

在camera.hpp里新继承了一个LensCamera类。需要多给出成像面（焦平面）到相机的实际距离以及相机的圆盘半径大小。

```
class LensCamera : public Camera {
    float fxy;
    float distance;//成像平面（画布）到其的距离
    float radius;//相机光圈半径
public:
    LensCamera(const Vector3f &center, const Vector3f &direction,
            const Vector3f &up, int imgW, int imgH, float angle,
float distance, float radius) : Camera(center, direction, up,
imgW, imgH) {
        // angle is in radian.
        this->fxy = imgH / (2 * tan(angle / 2) * (distance + 1));
        this->distance = distance;//在真实空间的距离
        this->radius = radius;//在真实空间的半径
    }

    Ray generateRay(const Vector2f &point) override {
        // 每次取样时，在相机的圆中随机取一点，与成像平面相连作为相机射出的
光线

        Vector3f drc(point.x() - width/2, height/2 - point.y(),
fxy * (distance + 1));
        drc.normalize();
        Vector3f drw = drc.x() * horizontal - drc.y() * up +
drc.z() * direction;
        drw.normalize();
        float theta = 2 * M_PI * RND2;
        float r = RND2 * radius;
        Vector3f point2 = r * sin(theta) * up +  r * cos(theta) *
horizontal;
        Vector3f r2 = drw * ((distance + 1) / (Vector3f::dot(drw,
direction))) - point2;
        r2.normalize();
        return Ray(this->center + point2, r2);
    }
};
```

运动模糊

在读取物体时新增了velocity选项，给出运动向量，在求交时新增一个时间项，随机取[0, 1]的时间。

以plane的求交为例，其他物体均相同：

```
    bool intersect(const Ray &r, Hit &h, float tmin, float time_)
override {
        Vector3f origin = r.getOrigin();
        Vector3f direction = r.getDirection();
        if(is_moving)
            origin -= time_ * this->velocity;//求time_时刻物体的位
置，实际上相当于光线起点向相反方向移动。其余与普通求交相同

        //.......以下代码省略
```

软阴影

在PPM中我使用的是面光源，而PPM光子发射的原理本来就自带软阴影的效果。

面光源的实现如下：

```
class RoundDisk_Light : public Light {
public:
    RoundDisk_Light() = delete;

    RoundDisk_Light(const Vector3f &p, const Vector3f &dir, const
Vector3f &c, float r, const Vector3f& em) {
        position = p;
        color = c;
        radius = r;
        emission = em;
        direction = dir.normalized();
        if(direction.x() < 1e-4 && direction.x() > -1e-4)
            if(direction.z() < 1e-4 && direction.z() > -1e-4)
                if(1 - direction.y() < 1e-4 || 1 + direction.y()
< 1e-4) {
                    x_axis = Vector3f::RIGHT;
                    y_axis = Vector3f::FORWARD;
                    return;
                }
        if(abs(direction.x()) < 1e-3 && abs(direction.z()) < 1e-
3) { //和y轴平行
            x_axis = Vector3f(1, 0, 0);
            y_axis = Vector3f(0, 0, 1);
        } else {
            x_axis = Vector3f::cross(direction,
Vector3f::UP).normalized();
            y_axis = Vector3f::cross(direction,
x_axis).normalized();
        }

    }

    ~RoundDisk_Light() override = default;

    void getIllumination(const Vector3f &p, Vector3f &dir,
Vector3f &col) const override {
        dir = (position - p);
        dir = dir / dir.length();
        col = color;
    }

    Ray get_ray(Vector3f & ini_power) const {
        float alpha = RND2 * 2 * M_PI;
        Vector3f p = position + radius * cos(alpha) * x_axis +
radius * sin(alpha) * y_axis; //在圆盘上随机取一点
        Vector3f new_dir = direction * RND2 + RND1 * x_axis +
RND1 * y_axis;
        new_dir.normalize();//随机方向
        if(new_dir.squaredLength() < 1e-4)
            new_dir = direction;
        ini_power = (M_PI * M_PI * radius * radius) * emission;
        // new_dir.print();
        // p.print();
        return Ray(p, new_dir); //随机选择发射方向、发射源头
    }

    Vector3f get_direction() const {
        return direction;
    }

private:

    Vector3f color;

    Vector3f direction;
    Vector3f x_axis, y_axis;
```

```
};
```

抗锯齿

在PPM和PT的相机发射光线取样时，均做了在周围画布网格随机扰动的操作，达到抗锯齿效果。

（PPM和PT代码已在上面列出，与抗锯齿相关的是下面这一句）

```
Vector2f ori = Vector2f(x + RND1 / 2, y + RND1 / 2);
```

UV纹理映射、凹凸贴图

主要完成了plane类、mesh类和sphere类的UV纹理映射，以及sphere类的凹凸贴图：

纹理映射使用的类是Texture，在Texture.cpp中。

```
#ifndef TEXTURE_H
#define TEXTURE_H

class Texture {
    public:
    unsigned char *picture;
    int width, height, channel;
    bool is_texture;
    Texture(const char *filename);
    Vector3f get_color(float u, float v);
    Vector3f get_pic_color(int x, int y);
    float get_gray(int index);
    float get_disturb(float u, float v, Vector2f &grad);
    int get_index(float u, float v);
};

#endif
```

```
#define STB_IMAGE_IMPLEMENTATION //使用了stb_image解码库来读取纹理图
片
#include <vecmath.h>
#include "stb_image.h"
```

```
#include <cstring>
#include "texture.h"

float Texture::get_gray(int index) { return (picture[index] /
255.0 - 0.5) * 2; }
//灰度图的转换

int Texture::get_index(float u, float v) { //根据计算得的uv位置获得纹
理图中的位置
    u -= int(u);
    v -= int(v);
    u = u < 0 ? 1 + u : u;
    v = v < 0 ? 1 + v : v;
    int x = u * width;
    int y = v * height;
    x = x < 0 ? 0 : x;
    x = x > width - 1 ? width - 1 : x;
    y = y < 0 ? 0 : y;
    y = y > height - 1 ? height - 1 : y;
    int index = (y * width + x) * channel;
    return index;
};

Texture::Texture(const char* filename) {
    if (std::strlen(filename) == 0) {
        is_texture = false;
        return;
    }
    this->picture = stbi_load(filename, &width, &height,
&channel, 0);//读取图片，获得宽高和通道数
    is_texture = true;
}

Vector3f Texture::get_color(float u, float v) { //通过uv浮点数得到颜
色

    u -= int(u);
    v -= int(v);
    u = u < 0 ? 1 + u : u;
    v = v < 0 ? 1 + v : v;
    u = u * width;
    v = height * (1 - v);
    int iu = (int)u, iv = (int)v;
    Vector3f ret_color = Vector3f::ZERO;
    float s = u - iu;
    float t = v - iv;
    //双线性插值
    Vector3f color1 = (1 - s) * get_pic_color(iu, iv + 1) + s *
get_pic_color(iu + 1, iv + 1);
    Vector3f color2 = (1 - s) * get_pic_color(iu, iv) + s *
get_pic_color(iu + 1, iv);
    ret_color += (1 - t) * color2;
    ret_color += t * color1;
    return ret_color;
}

Vector3f Texture::get_pic_color(int x, int y) {

    x = x < 0 ? 0 : x;
    x = x > width - 1 ? width - 1 : x;
    y = y < 0 ? 0 : y;
    y = y > height - 1 ? height - 1 : y;
    int index = (y * width + x) * channel;
    return Vector3f(picture[index + 0], picture[index + 1],
picture[index + 2]) / 255.0;
}


float Texture::get_disturb(float u, float v, Vector2f &grad) {//
凹凸贴图得到梯度

    if(!is_texture) {
        return 0.0;
    }

    int idx_ = get_index(u, v);
    float disturb = get_gray(idx_);
    float du = 1.0 / width, dv = 1.0 / height;
    //计算梯度
    grad[0] = width * (get_gray(get_index(u + du, v)) -
get_gray(get_index(u - du, v))) * 0.5;
    grad[1] = height * (get_gray(get_index(u, v + dv)) -
get_gray(get_index(u, v - dv))) * 0.5;
    return disturb;
}
```

material类中：

```
Vector3f get_color(float u, float v) {
        if(!texture.is_texture)
            return color;
        return texture.get_color(u, v);
}
```

plane类中：

```
void get_uv(const Vector3f& p, float& u, float& v) {
        v = Vector3f::dot(p - offset * normal, bi_normal) / 100;
        u = Vector3f::dot(p - offset * normal, main_tangent) /
100;
     // (100这个常数可以调节，取决于希望铺开的大小，bi_normal和
main_tangent是平行与法线的一组基向量)
}
```

```
//求交时调用:
float uu = 0, vv = 0;
get_uv(next_origin, uu, vv);
h.set(t, this->material, new_normal, material->get_color(uu, vv),
next_origin, material->texture.is_texture);
```

Sphere类中，代码段位于求交成功后的处理中：

```
float u;
u = 0.5 + atan2(old_normal.x(), old_normal.z()) / (2 * M_PI);
float v = 0.5 - asin(old_normal.y()) / M_PI; //极坐标思想确定一个u、
v

Vector2f grad = Vector2f::ZERO;
float f = material->bump.get_disturb(u, v, grad);//bump是Texture类
型，得到法线扰动

Vector3f new_normal = p;
if (!(f < 1e-4 && f > -1e-4)) {//需要法线扰动
    float phi = u * 2 * M_PI, theta = M_PI - v * M_PI;
    Vector3f pu(-p.z(), 0, p.x()), pv(p.y() * cos(phi), -radius *
sin(theta), p.y() * sin(phi));

    new_normal = Vector3f::cross(pu + old_normal * grad[0] / (2 *
M_PI), pv + old_normal * grad[1] / M_PI).normalized();//两个方向的
扰动叉乘得到新的法线

}
Vector3f new_color = material->get_color(u, v);//查询贴图颜色
h.set(final_t / dir_len, this->material, new_normal, new_color, p
+ this->center, material->texture.is_texture);
return true;
```

mesh类中：

mesh的uv由obj文件储存，详见“复杂网格模型读取及法向插值”部分。

参数曲面牛顿迭代法求交

具体实现在curve.hpp和revsurface.hpp中

curve.hpp（省去了PA2中要求实现的离散点的计算：discretize函数）

```
#ifndef CURVE_HPP
#define CURVE_HPP

#include "object3d.hpp"
#include <vecmath.h>
#include <vector>
#include <utility>

#include <algorithm>

struct CurvePoint {
    Vector3f V; // Vertex
    Vector3f T; // Tangent  (unit)
    float t;//对应的t
};

Vector3f now_xy;
```

```
bool dist(const CurvePoint& a, const CurvePoint& b) {
    return (a.V - now_xy).squaredLength() < (b.V -
now_xy).squaredLength();
}

class Curve : public Object3D {
protected:
    std::vector<Vector3f> controls;
public:
    std::vector<CurvePoint> mydata;

    explicit Curve(std::vector<Vector3f> points) :
controls(std::move(points)) {}

    bool intersect(const Ray &r, Hit &h, float tmin, float t)
override {
        return false;
    }

    std::vector<Vector3f> &getControls() {
        return controls;
    }

    virtual void discretize(int resolution,
std::vector<CurvePoint>& data) = 0;

    virtual bool valid_t(float t) = 0;

    void get_t(float x_, float y_, float& t) { //根据x, y估计t, 期望
获得一个比较精确的初始t
        now_xy = Vector3f(x_, y_, 0.0);

        //取和估计点之间距离最小的那个离散点来估计t
        t = (*min_element(mydata.begin(), mydata.end(), dist)).t;
    };

    bool is_on_curve(const Vector3f& vv) {
        now_xy = Vector3f(-sqrt(vv.x() * vv.x() + vv.z() *
vv.z()), vv.y(), 0.0);
        auto it  = min_element(mydata.begin(), mydata.end(),
dist);
        if(((*it).V - now_xy).length() > 1e-2)
            return false;
        return true;
    }

    virtual void get_point(float, Vector3f&, Vector3f&) = 0; //根据t获得精确的point和切线
};

class BezierCurve : public Curve {
public:
    explicit BezierCurve(const std::vector<Vector3f> &points) : Curve(points) {
        if (points.size() < 4 || points.size() % 3 != 1) {
            printf("Number of control points of BezierCurve must be 3n+1!\n");
            exit(0);
        }
        this->discretize(1000, mydata);//先计算1000个离散点储存起来以便估计t
    }

    bool valid_t(float t) { //得到的t是否有效
        return (t >= 0 && t <= 1);
    }

    void get_point(float t_, Vector3f& point, Vector3f& grad) override {//根据t计算曲线上的点以及对应梯度

        std::vector<Vector3f> A = controls;
        int n = this->controls.size() - 1;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n - i; j++) {
                if(i == n - 1)
                    grad = A[1] - A[0];
                A[j] = (1 - t_) * A[j] + t_ * A[j + 1];

            }
        point = A[0];
    }

protected:

};
```

```
class BsplineCurve : public Curve {
public:
    inline float t(float i) {return i / (n + k + 1); };

    BsplineCurve(const std::vector<Vector3f> &points) :
Curve(points) {
        if (points.size() < 4) {
            printf("Number of control points of BspineCurve must
be more than 4!\n");
            exit(0);
        }
        this->discretize(100, mydata);//先计算若干离散点储存起来以便估
计t
    }

    bool valid_t(float t) {//得到的t是否有效
        n = controls.size() - 1;
        return (t >= this->t(k) && t <= this->t(n));
    }

    void get_point(float t_, Vector3f& point, Vector3f& grad)
override {//根据t计算曲线上的点以及对应梯度

        n = controls.size() - 1;
        CurvePoint ft;
        ft.V = Vector3f::ZERO;
        ft.T = Vector3f::ZERO;
        float t = t_;
        float B[n + k + 1][k + 1]; // [0, n + k][0, k]
        //std::cout << t;
        int index = k;
        for(int i = k; i <= n; i++) {
            if(t > this->t(i))
                index = i;
            else
                break;
        }
        for(int i = 0; i < n + k + 1; i++)
            (i == index) ? B[i][0] = 1 : B[i][0] = 0;

        for(int p = 1; p <= k; p++)
            for(int i = 0; i + p + 1 <= n + k + 1; i++)
```

```
                B[i][p] = ((t - this->t(i)) / (this->t(i + p) - 
this->t(i))) * B[i][p - 1] + ((this->t(i + p + 1) - t) / (this-
>t(i + p + 1) - this->t(i + 1))) * B[i + 1][p - 1];

        for(int i = 0; i <= n; i++)
            ft.V += B[i][k] * controls.at(i);
        for(int i = 0; i <= n; i++)
            ft.T += (k * (B[i][k - 1] / (this->t(i + k) - this-
>t(i)) - B[i + 1][k - 1] / (this->t(i + k + 1) - this->t(i + 
1)))) * controls.at(i);

        point = ft.V;
        grad = ft.T;
    }

protected:
    int n = 0;
    int k = 3;
};

#endif // CURVE_HPP
```

revsurface.hpp中：

```
#ifndef REVSURFACE_HPP
#define REVSURFACE_HPP

#include "object3d.hpp"
#include "curve.hpp"
#include <tuple>
#include "constant.h"

class RevSurface : public Object3D {

    Curve *pCurve;

    float x_max, y_max, y_min;

public:
    RevSurface(Curve *pCurve, Material* material) : 
pCurve(pCurve), Object3D(material, Vector3f::ZERO) {
        // Check flat.
        float xmax = 0, ymax = -1e38, ymin = 1e38;
        for (const auto &cp : pCurve->getControls()) {
            if (cp.z() != 0.0) {
                printf("Profile of revSurface must be flat on xy
plane.\n");
                exit(0);
            }
            if (abs(cp.x()) > xmax) xmax = abs(cp.x());
            if (cp.y() > ymax) ymax = cp.y();
            if (cp.y() < ymin) ymin = cp.y();
        }
        x_max = xmax;
        y_max = ymax;
        y_min = ymin;//得到AABB包围盒的顶点
    }

    ~RevSurface() override {
        delete pCurve;
    }

    float F_(const Vector3f& dir, const Vector3f& origin, float
x_t, float y_t) {
        float x1 = dir.y() * dir.y() * x_t * x_t;
        float y1 = y_t - origin.y();
        float x2 = pow((y1 * dir.x() + dir.y() * origin.x()), 2);
        float x3 = pow((y1 * dir.z() + dir.y() * origin.z()), 2);
        return x2 + x3 - x1;
    }//牛顿迭代的那个函数，以t为自变量

    float F_grad(const Vector3f& dir, const Vector3f& origin,
float x_t, float y_t, float x_grad_t, float y_grad_t) {
        float x1 = 2 * dir.y() * dir.y() * x_t * x_grad_t;
        float y1 = y_t - origin.y();
        float x2 = 2 * dir.x() * y_grad_t * (y1 * dir.x() +
dir.y() * origin.x());
        float x3 = 2 * dir.z() * y_grad_t * (y1 * dir.z() +
dir.y() * origin.z());
        return x2 + x3 - x1;
    }//牛顿迭代的那个函数对t求导

    bool inter_AABB(const Ray &r, Hit &h, float & x_, float & y_)
{//判断与AABB包围盒是否相交。
        Vector3f direction = r.direction;
        Vector3f origin = r.origin;
        if(abs(r.origin.x()) < x_max && abs(r.origin.z()) < x_max
&& y_min < r.origin.y() && r.origin.y() < y_max) {
            x_ = r.origin.x() * r.origin.x() + r.origin.z() *
r.origin.z();
            x_ = - sqrt(x_);
            y_ = r.origin.y();
            return true;
        }

        float t_xmin = -1e38;
        float t_xmax = 1e38;
        float t_ymin = -1e38;
        float t_ymax = 1e38;
        float t_zmin = -1e38;
        float t_zmax = 1e38;

        if(direction.x() > 1e-4) {
            t_xmin = (-x_max - origin.x()) / direction.x();
            t_xmax = (x_max - origin.x()) / direction.x();
        } else if(direction.x() < -1e-4) {
            t_xmax = (-x_max - origin.x()) / direction.x();
            t_xmin = (x_max - origin.x()) / direction.x();
        } else if(origin.x() > x_max || origin.x() < -x_max)
            return false;

        if(t_xmax <= 0)
            return false;

        if(direction.y() > 1e-4) {
            t_ymin = (y_min - origin.y()) / direction.y();
            t_ymax = (y_max - origin.y()) / direction.y();
        } else if(direction.y() < -1e-4) {
            t_ymax = (y_min - origin.y()) / direction.y();
            t_ymin = (y_max - origin.y()) / direction.y();
        } else if(origin.y() > y_max || origin.y() < y_min)
            return false;

        if(t_ymax <= 0)
            return false;

        if(direction.z() > 1e-4) {
            t_zmin = (-x_max - origin.z()) / direction.z();
            t_zmax = (x_max - origin.z()) / direction.z();
        } else if(direction.z() < -1e-4) {
            t_zmax = (-x_max - origin.z()) / direction.z();
            t_zmin = (x_max - origin.z()) / direction.z();
        } else if(origin.z() > x_max || origin.z() < -x_max)
            return false;

        if(t_zmax <= 0)
            return false;

        float t0, t1;
        t0 = max(t_zmin, max(t_xmin, t_ymin));
        t1 = min(t_zmax, min(t_xmax, t_ymax));

        if(t0 < t1) {
            float the_t = (t0 > 0) ? t0 : t1;
            Vector3f next_ori = origin + the_t * direction;
            x_ =  next_ori.x() * next_ori.x() + next_ori.z() *
next_ori.z();
            x_ =  - sqrt(x_);
            y_ = next_ori.y();
            return true;
        } else {
            return false;
        }

    }

    //对于curve上的点(x, y),其旋转后为(xcos, y, xsin)
    bool intersect(const Ray &r, Hit &h, float tmin, float time_)
override {
        float AABB_x, AABB_y;
        bool is_interAABB = inter_AABB(r, h, AABB_x, AABB_y);
        if(!is_interAABB) {
            // printf("Not hit aabb\n");
            return false;
        }
        //与包围盒相交，得到包围盒的顶点，以此得到一个估计的平面点(x, y),
然后估计一下t。

        float estimate_t;
        float ini_ft;
        pCurve->get_t(AABB_x, AABB_y, estimate_t);
        //牛顿迭代
        int dep = 0;
        while(dep < newton_depth) {
            dep++;
            if(!pCurve->valid_t(estimate_t))
                return false;

            Vector3f now_point;
            Vector3f now_grad;
            pCurve->get_point(estimate_t, now_point, now_grad);
            float ft = F_(r.direction, r.origin, now_point.x(),
now_point.y());
            float ft_grad = F_grad(r.direction, r.origin,
now_point.x(), now_point.y(), now_grad.x(), now_grad.y());

            if(abs(ft) < 1e-5) {
                float tr;
                if(abs(r.direction.y()) > 1e-3) {
                    tr = (now_point.y() - r.origin.y()) /
(r.direction.y());
                }
                else {

                    float a = r.direction.z() * r.direction.z() +
r.direction.x() + r.direction.x();
                    float b = 2 * (r.direction.z() * r.origin.z()
+ r.direction.x() * r.origin.x());
                    float c = pow(r.origin.x(), 2) +
pow(r.origin.z(), 2) - pow(now_point.x(), 2);
                    tr = (sqrt(pow(b, 2) - 4 * a * c) - b) / (2 *
a);//一元二次方程
                }
                Vector3f next_origin = tr * r.direction +
r.origin;

                if(tr > tmin && pCurve->is_on_curve(next_origin))
{//得到的t确实有效后

                    Vector2f plane_normal(now_grad.y(), -
now_grad.x());
                    plane_normal.normalize();//在二维参数曲线上的法线

                    if(abs(now_point.x()) < 1e-4) {
                            h.set(tr, material, now_point.y() > 0 ?
Vector3f(0, 1, 0) : Vector3f(0, -1, 0));
                            return true;
                        }
                        float costheta = (r.direction.x() * tr +
r.origin.x()) / now_point.x();
                        float sintheta = (r.direction.z() * tr +
r.origin.z()) / now_point.x();
                        Vector3f new_normal = Vector3f(costheta *
plane_normal.x(), plane_normal.y(), sintheta *
plane_normal.x());//旋转后得到三维参数曲面上的法线
                        h.set(tr, material, new_normal);
                        return true;
                    }
                }
                float step = ft / ft_grad;
                if(step > 0.05)
                    step = 0.05;
                else if(step < -0.05)
                    step = -0.05;//限制步长
                estimate_t -= step;
            }
            return false;
        }
};

#endif //REVSURFACE_HPP
```

复杂网格模型读取及法向插值

代码主要在mesh.hpp和mesh.cpp中：

```
//mesh类储存了以下信息:
struct TriangleIndex {
    TriangleIndex() {
        x[0] = 0; x[1] = 0; x[2] = 0;
    }
    int &operator[](const int i) { return x[i]; }
    int x[3]{};
};

vector<TriangleIndex> vIndex, tIndex, nIndex;//对应一个平面的三个点的
顶点下标、贴图下标、法线下标
vector<Vector3f> v, vn;//对应obj文件中的一系列点和法向
vector<Vector2f> vt;//对应obj文件中贴图uv位置
vector<Object3D *> triangles;//储存每一个三角形
```

Mesh类的初始化函数如下：

```
Mesh::Mesh(const char *filename, Material *material, const
Vector3f& ve) : Object3D(material, ve) {//获得传入的文件名等信息，ve
指的是运动向量
    std::ifstream f;
    f.open(filename);
    if (!f.is_open()) {
        std::cout << "Cannot open " << filename << "\n";
        return;
    }
    std::string line;
    std::string vTok("v");
    std::string fTok("f");
    std::string texTok("vt");
    std::string vnTok("vn");

    char bslash = '/', space = ' ';
    std::string tok;
    bool texture_and_normal = false;
    while (true) {
        std::getline(f, line);
        if (f.eof()) {
            break;
        }
        if (line.size() < 3) {
            continue;
        }
        if (line.at(0) == '#') {
            continue;
        }
        std::stringstream ss(line);
        ss >> tok;

        if (tok == vTok) {
            Vector3f vec;
            ss >> vec[0] >> vec[1] >> vec[2];
            v.push_back(vec);
        } else if (tok == fTok) {
            TriangleIndex vrig, trig, nrig;
            //限定两种输入方式:
            //f 2 3 4与f 2/3/4 3/4/5 4/5/6
            if (line.find(bslash) != std::string::npos) {
                //有顶点、法向、贴图的下标
                texture_and_normal = true;
                std::replace(line.begin(), line.end(), bslash,
space);
                std::stringstream facess(line);
                facess >> tok;
                for (int ii = 0; ii < 3; ii++) {
                    facess >> vrig[ii] >> trig[ii] >> nrig[ii];
                    trig[ii]--;
                    vrig[ii]--;
                    nrig[ii]--;
                }
            } else {
                for (int ii = 0; ii < 3; ii++) {//只有顶点下标
                    ss >> vrig[ii];
                    vrig[ii]--;
                }
            }
            vIndex.push_back(vrig);
            tIndex.push_back(trig);
            nIndex.push_back(nrig);
        } else if (tok == texTok) {
            Vector2f texcoord;
            ss >> texcoord[0];
            ss >> texcoord[1];
            vt.push_back(texcoord);
        } else if (tok == vnTok) {
            Vector3f vnvec;
            ss >> vnvec[0] >> vnvec[1] >> vnvec[2];
            vn.push_back(vnvec);
        }
    }
    f.close();

    if(!texture_and_normal)
        computeNormal();
    else
        computeNormal_and_texture();

    cout << "Normal and (possible) texture have been computed."
<< endl;

    computeAABB();

    cout << "AABB box has been computed." << endl;
    cout << "triangle num:" << triangles_info.size() << endl;

    setup_bvh_tree();

    cout << "BVH tree has set up." << endl;

}

void Mesh::computeNormal() {//没有贴图和顶点法线，直接根据三个顶点计算面
法线
    vn.resize(vIndex.size());
    for (int triId = 0; triId < (int) vIndex.size(); ++triId) {
        TriangleIndex& triIndex = vIndex[triId];
        Vector3f a = v[triIndex[1]] - v[triIndex[0]];
        Vector3f b = v[triIndex[2]] - v[triIndex[0]];
        b = Vector3f::cross(a, b);
        vn[triId] = b / b.length();

        Triangle* triangle = new Triangle(v[triIndex[0]],
                                 v[triIndex[1]], v[triIndex[2]],
material, Vector3f::ZERO);
        triangle->normal = vn[triId];
        triangles.push_back(triangle);
    }
}
```

```cpp
void Mesh::computeNormal_and_texture() {//有贴图和顶点法线
    for (int triId = 0; triId < (int) vIndex.size(); ++triId) {

        TriangleIndex& triIndex = vIndex[triId];

        Triangle* triangle = new Triangle(v[triIndex[0]],
v[triIndex[1]], v[triIndex[2]], material, Vector3f::ZERO);

        triangles.push_back(triangle);

        if(!vt.empty()) {
            TriangleIndex& tIdx = tIndex[triId];
            ((Triangle *)triangles.back())->set_vt(vt[tIdx[0]],
vt[tIdx[1]], vt[tIdx[2]]);
        }

        if(!vn.empty()) {
            TriangleIndex& nIdx = nIndex[triId];
            ((Triangle *)triangles.back())->set_vn(vn[nIdx[0]],
vn[nIdx[1]], vn[nIdx[2]]);
        }

    }
}
```

其中调用到的set_vt和set_vn函数如下：

```
void set_vt(const Vector2f& _at, const Vector2f& _bt, const
Vector2f& _ct) {
        at = _at;
        bt = _bt;
        ct = _ct;
        is_texture = true;
}

void set_vn(const Vector3f& _an, const Vector3f& _bn, const
Vector3f& _cn) {
    an = _an;
    bn = _bn;
    cn = _cn;
    is_norm = true;
}
```

三角形计算法线插值以及uv纹理时的函数如下：

```
void get_uv(const Vector3f& p, float& u, float& v) {
    if (!is_texture)
        return;
    Vector3f va = (vertices[0] - p), vb = (vertices[1] - p), vc =
(vertices[2] - p);
    float ra = Vector3f::cross(vb, vc).length();//叉乘的大小正比于面
积
    float rb = Vector3f::cross(vc, va).length();
    float rc = Vector3f::cross(va, vb).length();
    Vector2f uv = (ra * at + rb * bt + rc * ct) / (ra + rb + rc);
//用面积加权平均
    u = uv.x();
    v = uv.y();
}

Vector3f get_norm(const Vector3f& p) {
    if(!is_norm)
        return this->normal;
    Vector3f va = (vertices[0] - p), vb = (vertices[1] - p), vc =
(vertices[2] - p);
    float ra = Vector3f::cross(vb, vc).length();
    float rb = Vector3f::cross(vc, va).length();
    float rc = Vector3f::cross(va, vb).length();
    return (ra * an + rb * bn + rc * cn).normalized();//加权平均
}
```

复杂网格模型层次包围盒的求交加速

使用BVH层次包围体求交加速，代码主要在mesh.hpp和mesh.cpp中：

mesh.hpp：

```
struct Pel {//每个图元的AABB包围盒
    Vector3f minxyz;//最小顶点
    Vector3f maxxyz;//最大顶点
    Vector3f centroid;//中点。粗略的质心
    int index;
    Pel(float minx, float miny, float minz, float maxx, float
maxy, float maxz, int idx) {
        minxyz = Vector3f(minx, miny, minz);
        maxxyz = Vector3f(maxx, maxy, maxz);
        centroid = (minxyz + maxxyz) * 0.5;
        index = idx;
    }
};



struct BVH_TreeNode { //BVH树的节点，储存父子关系以及这一节点的AABB包围
盒数据，如果是叶子节点，储存包含的图元AABB包围盒（连续储存，因此储存首个Pel的
迭代器就行）
    BVH_TreeNode* parent;
    BVH_TreeNode* lc;
    BVH_TreeNode* rc;
    Vector3f minxyz;
    Vector3f maxxyz;
    int pel_num;
    vector<Pel>::iterator first_pel; //有这个的是叶子节点
    BVH_TreeNode(BVH_TreeNode* a, vector<Pel>::iterator it, int
_num);
};


bool Ray_hit_AABB(BVH_TreeNode* e, const Ray &r, float& t);

class Mesh : public Object3D {
```

```
public:
    BVH_TreeNode* produce_child(int axis, BVH_TreeNode*
now_root);//axis:0为x轴，1为y轴，2为z轴

    Mesh(const char *filename, Material *m, const Vector3f& v);
    ~Mesh();
    struct TriangleIndex {
        TriangleIndex() {
            x[0] = 0; x[1] = 0; x[2] = 0;
        }
        int &operator[](const int i) { return x[i]; }
        int x[3]{};
    };

    vector<TriangleIndex> vIndex, tIndex, nIndex;
    vector<Vector3f> v, vn;
    vector<Vector2f> vt;
    vector<Object3D *> triangles;

    bool intersect(const Ray &r, Hit &h, float tmin, float time_)
override;
    bool hit_intersect(const Ray &r, Hit &h, float tmin,
BVH_TreeNode* e);
    BVH_TreeNode* root;
    vector<Pel> triangles_info;

    void delete_node(BVH_TreeNode* e);


private:

    // Normal can be used for light estimation
    void computeNormal();
    void computeNormal_and_texture();
    void computeAABB();

    void setup_bvh_tree();
    void produce_child(BVH_TreeNode* e);
};

#endif
```

mesh.cpp中（mesh初始化函数在上一部分已经列出，不再赘述）：

求交加速分为两部分：初始化时的建树和求交时的搜索。

建树相关函数是`void computeAABB()`、`void setup_bvh_tree()`、`void produce_child(BVH_TreeNode* e)`。

求交相关函数是`bool intersect(const Ray &r, Hit &h, float tmin, float time_) override`、`bool hit_intersect(const Ray &r, Hit &h, float tmin, BVH_TreeNode* e)`、`bool Ray_hit_AABB(BVH_TreeNode* e, const Ray &r, float& t);`;

```
#include "mesh.hpp"
#include <fstream>
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <utility>
#include <sstream>
#include "constant.h"

using namespace std;

float now_center;

bool Pel_min_x(Pel x, Pel y) {
    return x.minxyz.x() < y.minxyz.x();
}

bool Pel_min_y(Pel x, Pel y) {
    return x.minxyz.y() < y.minxyz.y();
}

bool Pel_min_z(Pel x, Pel y) {
    return x.minxyz.z() < y.minxyz.z();
}

bool Pel_max_x(Pel x, Pel y) {
    return x.maxxyz.x() < y.maxxyz.x();
}

bool Pel_max_y(Pel x, Pel y) {
    return x.maxxyz.y() < y.maxxyz.y();
}

bool Pel_max_z(Pel x, Pel y) {
    return x.maxxyz.z() < y.maxxyz.z();
}

bool Pel_cen_x(Pel x) {
    return x.centroid.x() < now_center;
}

bool Pel_cen_y(Pel x) {
    return x.centroid.y() < now_center;
}

bool Pel_cen_z(Pel x) {
    return x.centroid.z() < now_center;
}

BVH_TreeNode::BVH_TreeNode(BVH_TreeNode* a, vector<Pel>::iterator
it, int _num) {//BVH树节点的初始化
        parent = a;
        lc = nullptr;
        rc = nullptr;
        first_pel = it;
        pel_num = _num;

        if(_num) {
            float minx = (*min_element(it, it + _num,
Pel_min_x)).centroid.x();
            float miny = (*min_element(it, it + _num,
Pel_min_y)).centroid.y();
            float minz = (*min_element(it, it + _num,
Pel_min_z)).centroid.z();
            float maxx = (*max_element(it, it + _num,
Pel_max_x)).centroid.x();
            float maxy = (*max_element(it, it + _num,
Pel_max_y)).centroid.y();
            float maxz = (*max_element(it, it + _num,
Pel_max_z)).centroid.z();
            minxyz = Vector3f(minx, miny, minz);
            maxxyz = Vector3f(maxx, maxy, maxz);
        }
    }
```

```
bool Ray_hit_AABB(BVH_TreeNode* e, const Ray &r, float &the_t)
{//包围盒求交

    if(e->pel_num == 0)
        return false;

    Vector3f origin = r.getOrigin();
    Vector3f direction = r.getDirection().normalized();

    float t_xmin = -1e38;
    float t_xmax = 1e38;
    float t_ymin = -1e38;
    float t_ymax = 1e38;
    float t_zmin = -1e38;
    float t_zmax = 1e38;

    if(direction.x() > 1e-4) {
        t_xmin = (e->minxyz.x() - origin.x()) / direction.x();
        t_xmax = (e->maxxyz.x() - origin.x()) / direction.x();
    } else if(direction.x() < -1e-4) {
        t_xmax = (e->minxyz.x() - origin.x()) / direction.x();
        t_xmin = (e->maxxyz.x() - origin.x()) / direction.x();
    } else if(origin.x() > e->maxxyz.x() || origin.x() < e-
>minxyz.x())
        return false;

    if(t_xmax <= 0)
        return false;

    if(direction.y() > 1e-4) {
        t_ymin = (e->minxyz.y() - origin.y()) / direction.y();
        t_ymax = (e->maxxyz.y() - origin.y()) / direction.y();
    } else if(direction.y() < -1e-4) {
        t_ymax = (e->minxyz.y() - origin.y()) / direction.y();
        t_ymin = (e->maxxyz.y() - origin.y()) / direction.y();
    } else if(origin.y() > e->maxxyz.y() || origin.y() < e-
>minxyz.y())
        return false;

    if(t_ymax <= 0)
        return false;
    if(direction.z() > 1e-4) {
        t_zmin = (e->minxyz.z() - origin.z()) / direction.z();
        t_zmax = (e->maxxyz.z() - origin.z()) / direction.z();
    } else if(direction.z() < -1e-4) {
        t_zmax = (e->minxyz.z() - origin.z()) / direction.z();
        t_zmin = (e->maxxyz.z() - origin.z()) / direction.z();
    } else if(origin.z() > e->maxxyz.z() || origin.z() < e-
>minxyz.z())
        return false;

    if(t_zmax <= 0)
        return false;

    float t0, t1;
    t0 = max(t_zmin, max(t_xmin, t_ymin));
    t1 = min(t_zmax, min(t_xmax, t_ymax));

    if(t0 < t1) {
        the_t = (t0 > 0) ? t0 : t1;
        return true;
    } else {
        return false;
    }

};

bool Mesh::hit_intersect(const Ray &r, Hit &h, float tmin,
BVH_TreeNode* e) {//已经通过AABB包围盒测试的
    if (e->rc == nullptr && e->lc == nullptr) {//叶子节点
        bool result = false;
        for (auto it = e->first_pel; it != e->first_pel + e-
>pel_num; ++it) {
            result |= ((Triangle *)triangles[(*it).index])-
>intersect(r, h, tmin, 0);
        }
        return result;
    }
    float lc_t;
    float rc_t;
    bool lc_hit = Ray_hit_AABB(e->lc, r, lc_t);
    bool rc_hit = Ray_hit_AABB(e->rc, r, rc_t);

    if(!(lc_hit || rc_hit))
```

```
        return false;

    bool real_hit;

    //看左右子节点哪个包围盒求交的结果近，先求近的，如果近的求交成功无需求远
的，否则需要求远的
    if(lc_hit && rc_hit) {
        if(lc_t < rc_t) {
            real_hit = hit_intersect(r, h, tmin, e->lc);
            if(real_hit)
                return true;
            else
                real_hit = hit_intersect(r, h, tmin, e->rc);
            return real_hit;
        } else {
            real_hit = hit_intersect(r, h, tmin, e->rc);
            if(real_hit)
                return true;
            else
                real_hit = hit_intersect(r, h, tmin, e->lc);
            return real_hit;
        }
    } else if(lc_hit)
        return hit_intersect(r, h, tmin, e->lc);
    else
        return hit_intersect(r, h, tmin, e->rc);

};


bool Mesh::intersect(const Ray &r, Hit &h, float tmin, float
time_) {

    // Optional: Change this brute force method into a faster
one.
    bool result = false;
    // for (int triId = 0; triId < (int) triangles.size();
++triId) {
    //     result |= ((Triangle *)triangles[triId])->intersect(r,
h, tmin);
    // }
    // return result;
    float t;
    if(!Ray_hit_AABB(root, r, t)) //先判断是否和AABB包围盒交
        return false;

    if(!is_moving)
        result = hit_intersect(r, h, tmin, root);
    else
        result = hit_intersect(Ray(r.origin - time_ * velocity,
r.direction), h, tmin, root);
    return result;
}


void Mesh::computeAABB() { //计算每个三角形的AABB包围盒，并储存为图元
Pel
    if((int) triangles.size() <= 0) {
        printf("Obj File Errors.\n");
        exit(0);
    }

    for (int i = 0; i < (int) triangles.size(); ++i) {
        float now_minx, now_maxx, now_miny, now_maxy, now_minz,
now_maxz;
        for (int j = 0; j < 3; j++) {
            if(j == 0) {
                now_minx = now_maxx = ((Triangle*) triangles[i])-
>vertices[j].x();
                now_miny = now_maxy = ((Triangle*) triangles[i])-
>vertices[j].y();
                now_minz = now_maxz = ((Triangle*) triangles[i])-
>vertices[j].z();
            } else {
                if(((Triangle*) triangles[i])->vertices[j].x() <
now_minx)
                    now_minx = ((Triangle*) triangles[i])-
>vertices[j].x();
                else if(((Triangle*) triangles[i])-
>vertices[j].x() > now_maxx)
                    now_maxx = ((Triangle*) triangles[i])-
>vertices[j].x();

                if(((Triangle*) triangles[i])->vertices[j].y() <
now_miny)
```

```
                    now_miny = ((Triangle*) triangles[i])-
>vertices[j].y();
                else if(((Triangle*) triangles[i])-
>vertices[j].y() > now_maxy)
                    now_maxy = ((Triangle*) triangles[i])-
>vertices[j].y();

                if(((Triangle*) triangles[i])->vertices[j].z() <
now_minz)
                    now_minz = ((Triangle*) triangles[i])-
>vertices[j].z();
                else if(((Triangle*) triangles[i])-
>vertices[j].z() > now_maxz)
                    now_maxz = ((Triangle*) triangles[i])-
>vertices[j].z();
            }
        }
        Pel t = Pel(now_minx, now_miny, now_minz, now_maxx,
now_maxy, now_maxz, i);
        this->triangles_info.push_back(t);
    }
}

void Mesh::delete_node(BVH_TreeNode* e) {
    if(!e)
        return;
    if (e->rc == nullptr && e->lc == nullptr) {
        delete e;
        return;
    }
    delete_node(e->lc);
    delete_node(e->rc);
    delete e;
    return;
}

Mesh::~Mesh() {//删树节点
    for (int i = 0; i < triangles.size(); i++)
        delete triangles[i];
    delete_node(root);
}

void Mesh::setup_bvh_tree() {//建树的函数
```

```
    root = new BVH_TreeNode(nullptr, triangles_info.begin(),
(int) triangles_info.size());
    produce_child(root);
}

void Mesh::produce_child(BVH_TreeNode* e) {//递归建树，每一轮以物体中
心跨度最大的那个维度进行切割，切割按跨度的中点切
    if(e->pel_num < leaf_max_pel) //限制叶子图元最大数目
        return;
    bool (*comp)(Pel x);
    float dx = e->maxxyz.x() - e->minxyz.x();
    float dy = e->maxxyz.y() - e->minxyz.y();
    float dz = e->maxxyz.z() - e->minxyz.z();
    if (dx > dy && dx > dz) {
        comp = Pel_cen_x;
        now_center = e->minxyz.x() + dx / 2;
    } else if (dy > dz){
        comp = Pel_cen_y;
        now_center = e->minxyz.y() + dy / 2;
    } else {
        comp = Pel_cen_z;
        now_center = e->minxyz.z() + dz / 2;
    }

    auto bound = partition(e->first_pel, e->first_pel + e-
>pel_num, comp);//找到中点，对图元进行分类
    int distance1 = distance(e->first_pel, bound);
    int distance2 = e->pel_num - distance1;
    BVH_TreeNode* l_child = new BVH_TreeNode(e, e->first_pel,
distance1);
    BVH_TreeNode* r_child = new BVH_TreeNode(e, bound,
distance2);
    produce_child(l_child);
    produce_child(r_child);
}
```

# 渲染结果

原图在result文件夹中，markdown文件可能看不到图片，请移步pdf或直接前往result文件夹查看。

第一幅图是采用了法向插值使水面和鱼平滑，使用PPM达到焦散效果。

第二幅图采用法向插值使兔子平滑，天花板、墙壁、球和军刀使用纹理贴图，使用PPM达到焦散效果，球也有软阴影效果。

第三幅图采用PT算法，篮球是纹理贴图，绿球是凹凸贴图，军刀有运动模糊效果。

第四幅图采用PPM算法，水杯是参数曲面，有景深、焦散效果。