山东大学软件学院实验报告

实验题目：RayTracing		学号：201400301005
日期：2017-05-28	班级： 14 基地班	姓名： 邢博
软件环境： Win7 CodeBlock Opengl		
实验目的：了解 Phong 光照模型，熟练使用 opengl 编写程序		

实验内容：

1 简介

本程序是一个基于光线追踪的3D 渲染程序. 可以选用Phong 模型作为局部光照模型或Path Tracing作为全局光照模型,渲染3D 场景. 支持平面、球、三角面片、三角网格(可从obj 文件读入)几种几何对象,并可方便的扩展. 渲染效果上,支持软阴影,抗锯齿,景深,自定义纹理等功能, 并可对三角网格进行简化. 三角网格,全局渲染,网格简化均采用特定数据结构(KD 树及堆)与多线程加速,效率很高. 同时,将渲染功能嵌入了图形界面,可以在图形界面上支持obj 文件的预览及简化.

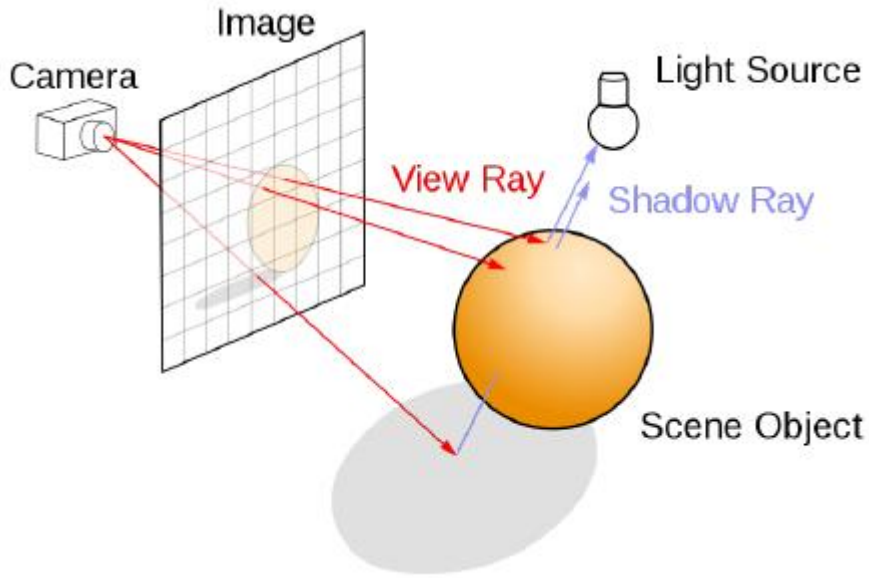
1.1 依赖

1. 本程序用C++11 编写,需要编译器支持C++11 中的ranged loop, initializer list,
type inference 等语法, 且需标准库包含std::shared_ptr, std::future 类.建议使
用g++_ 4.8 编译.
2. OpenCV2 1
3. ImageMagick 2
4. Qt4 3 (可选)

1.2 编译

在src目录中,使用make命令和make gui命令分别编译命令行程序与图形界面程序.

2.1 光线追踪及局部光照模型
光线追踪的原理如下图所示.



选定视点位置及一观察屏, 从视点到屏上各点发出光线, 与空间中物体求交. 在交点处根据局部光照模型计算颜色, 再递归的计算反射、透射光颜色, 混合后显示在屏上.
此程序使用的局部光照模型为Phong 模型, 其主要公式为

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\overrightarrow{L_m} \cdot \overrightarrow{N} i_{m,d} + k_s(\overrightarrow{R_m} \cdot \overrightarrow{V})^{\alpha} i_{m,s}))$$

其中*Ks, Kd, Ka;* 分别为物体表面该点处的高光系数, 漫反射系数, 环境光系数, 亮度. 向量*Lm*为该点指向光源的向量, 向量*N*, *向量Rm*为表面法向量*Rm* 为光源指向该点的光线经理想反射后的指向, *向量V*为视点到表面交点处的向量.
实现见下：

```
using
namespace
std;
            void Space::add_light(const Light& light) {
                    if (use_soft_shadow) {
                            real_t delta_theta = 2 * M_PI / SOFT_SHADOW_LIGHT;
                            real_t theta = 0;
                            REP(k, SOFT_SHADOW_LIGHT) {
                                    Vec diff = Vec(cos(theta), sin(theta), 0) *
            SOFT_SHADOW_RADIUS;
                                    theta += delta_theta;
                                    lights.push_back(make_shared<Light>(
                                                            light.get_src() + diff,
```

```
light.color, light.intensity / SOFT_SHADOW_LIGHT));
                }
        } else
                lights.push_back(make_shared<Light>(light));
}
void Space::add_obj(const rdptr& objptr) {
        if (!objptr->infinity()) {
                auto k = objptr->get_aabb();
                bound_min.update_min(k.min - Vec::eps());
                bound_max.update_max(k.max + Vec::eps());
        }
        objs.push_back(objptr);
}
void Space::finish() {              // called from View::View()
        if (use_tree)
                build_tree();
}
void Space::build_tree() {
        // the final objs contains a kdtree and all the infinite obj
        list<rdptr> infinite_obj;
        for (auto itr = begin(objs); itr != end(objs);) {
                if ((*itr)->infinity()) {
                        infinite_obj.push_back(*itr);
                        objs.erase(itr++);
                } else
                        ++itr;
        }
        auto finite_tree = make_shared<KDTree>(objs, AABB(bound_min,
bound_max));
        P(finite_tree->get_aabb());
        objs = move(infinite_obj);
        if (finite_tree) objs.emplace_back(finite_tree);
}
shared_ptr<Trace> Space::find_first(const Ray& ray, bool include_light)
const {
        real_t min = numeric_limits<real_t>::max();
        shared_ptr<Trace> ret;
        for (auto & obj : objs) {
                auto tmp = obj->get_trace(ray, min ==
numeric_limits<real_t>::max() ? -1 : min);
                if (tmp) {
                        real_t d = tmp->intersection_dist();
                        if (update_min(min, d)) ret = move(tmp);
                }
        }
        if (include_light)          // also look for light
```

```
                for (auto &l : lights) {
                        auto tmp = l->get_trace(ray, min ==
numeric_limits<real_t>::max() ? -1 : min);
                        if (tmp) {
                                real_t d = tmp->intersection_dist();
                                if (update_min(min, d)) ret = move(tmp);
                        }
                }
        return ret;
}
bool Space::find_any(const Ray& ray, real_t max_dist) const {
        for (auto & obj : objs) {
                auto tmp = obj->get_trace(ray, max_dist);
                if (tmp) return true;
        }
        return false;
}
```

2.2 全局光照模型

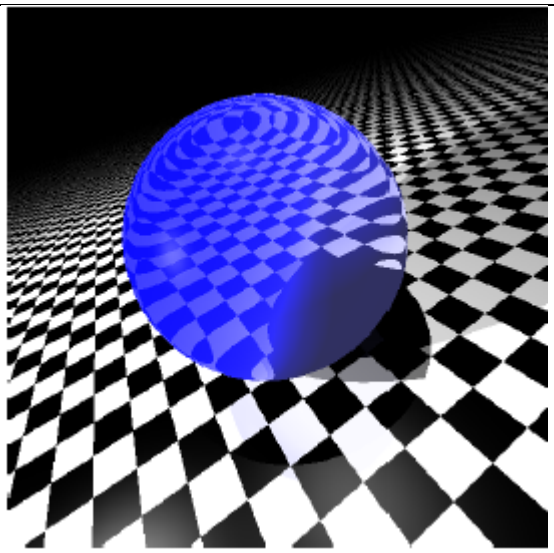
程序还支持了基于Path Tracing4 的全局光照模型,其渲染方程如下:

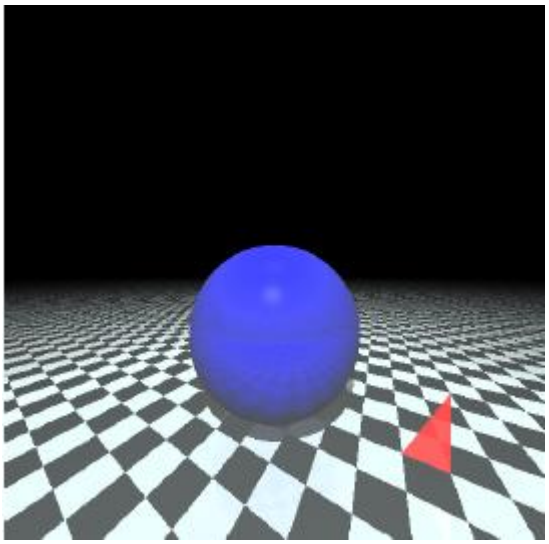
$$L(x \to v) = L_e(x \to v) + \int_\Omega L(\Phi \to x) F_s(x, \Phi \to v) \cos\theta_\Phi d\Omega$$

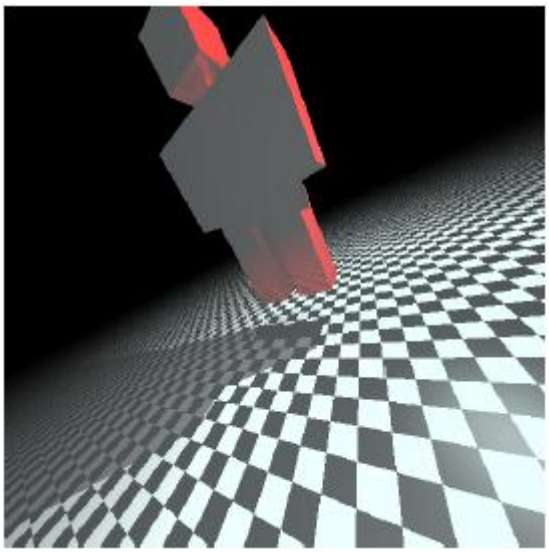
其中, $L(a—>b)$ 表示从b 处观察a 处所得结果,Le 为物体自身发光亮度,$F_s(x, \Phi \to v)$为物体表面BRDF 函数.

采用Monte Carlo Path Tracing对此方程进行逼近,其方法是: 在每次光线与物体的相交后, 由交点处处随机向其他方向发射多条光线, 递归求解后取平均值作为此点颜色. 随机时依据表面材质的不同选取不同的概率分布: 对于漫反射,遵循空间均匀分布,实现中, 在半球面中随机采样. 对于反射,分布近似一个尖峰函数,仅在反射方向上概率非0.对于透射,依照Fresnel 方程计算透射比与反射比,随机选择一者进行递归. 具体实现见上面代码

Monte Carlo Path Tracing 需要大量的采样才能够得到较好效果,因而其效率较低, 一张质量较好的图需要二十分钟渲染. 但它得到的图像更真实, 如下图.







2.3 几何对象表示及计算

程序支持了平面、球、三角面片、包围盒四类基本几何物体，物体都需要各自拥有与光线求交的方法.光线光线是一条射线,包含一个起始点及方向. 见include/geometry/ray.hh，代码如下

```
#pragma
once
        #include <cmath>
        #include "geometry/geometry.hh"
        class Ray {
                public:
                        bool debug = false;
                        Vec orig, dir;
                        real_t density;
                        Ray(){}
                        Ray(const Vec & _orig, const Vec& _dir, real_t _density =
        bool normalize = false):
                                orig(_orig), dir(_dir), density(_density) {
                                if (normalize)
                                        dir.normalize();
                        }
                        virtual ~Ray(){}
                        Vec get_dist(real_t d) const
                        { return orig + dir * d; }
                        real_t distance(const Vec& p) const
                        { return sqrt(sqrdistance(p)); }
                        real_t sqrdistance(const Vec& p) const
                        { return (project(p) - p).sqr(); }
                        Vec project(const Vec& p) const {
                                real_t t = (p - orig).dot(dir);
                                return get_dist(t);
                        }
                        bool contains(const Vec& p) const
                        { return (fabs(sqrdistance(p)) < EPS) && ((p.x - orig.x)
        (dir.x) >= -EPS); }
                        friend std::ostream& operator << (std::ostream & os, cons
        Ray& ray)
                        { return os << "orig: " << ray.orig << " , dir:" << ray.d
        };
```

无穷平面为了计算方便,使用平面法向及平面到原点的距离作为确定平面的方式. 平面与光线求交时,首先通过光线指向判断是否相交,再通过光线在平面法向方向的投影长度计算交点.见include/geometry/infplane.hh, renderable/plane.cc，代码见下面

```
#include
"renderable/plane.hh"
                        using namespace std;
                        shared_ptr<Trace> Plane::get_trace(const Ray& ray, real_t dist)
                        const {
                                auto ret = make_shared<PlaneTrace>(*this, ray);
                                if (ret->intersect(dist))
                                        if (fabs(dist + 1) < EPS or ret-
                        >intersection_dist() < dist)
                                                return ret;
                                return nullptr;
                        }
                        AABB Plane::get_aabb() const { error_exit("shoudl not be
                        here"); exit(0); }
                        Vec Plane::surf_dir() const {
                                Vec ret(plane.norm.y, -plane.norm.x, 0);
                                // possibily get a (0,0,0) !
                                // but cannot be all zero
                                if (ret == Vec::zero())
                                        ret = Vec(0, plane.norm.z, -plane.norm.y);
                                m_assert(!(ret == Vec::zero()));
                                ret.normalize();
                                return ret;
                        }
                        bool PlaneTrace::intersect(real_t max_dist) const {
                                dist_to_plane = plane.plane.dist(ray.orig);
                                if (fabs(dist_to_plane) < EPS or (max_dist > 0 and
                        dist_to_plane > max_dist)) // source on the plane
                                        return false;
                                dir_dot_norm = plane.plane.norm.dot(ray.dir);
                                if (fabs(dir_dot_norm) < EPS)  // parallel to plane
                                        return false;
                                if ((dist_to_plane > 0) ^ (dir_dot_norm < 0))  // ray
                        leaves plane
                                        return toward = false;
                                if (plane.radius < EPS)              // is an infinite
                        plane
                                        return true;
                                else {
                                        inter_dist = - dist_to_plane / dir_dot_norm;
                                        Vec inter_point = ray.get_dist(inter_dist);
                                        real_t dist = (inter_point -
                        plane.center).mod();
                                        if (dist >= plane.radius)
                                                return false;
                                        else
```

```
                                        return true;
                        }
                }
                real_t PlaneTrace::intersection_dist() const {
                        if (isfinite(inter_dist))
                                return inter_dist;
                        inter_dist = -dist_to_plane / dir_dot_norm;
                        return inter_dist;
                }
                Vec PlaneTrace::normal() const {     // norm to the ray side
                        Vec ret = plane.plane.norm;
                        if (plane.plane.in_half_space(ray.orig))
                                return ret;
                        return -ret;
                }
                shared_ptr<Surface> PlaneTrace::transform_get_property() const
                {
                        Vec diff = intersection_point() - plane.center;
                        real_t x = diff.dot(plane.surfdir);
                        real_t y =
                diff.dot(plane.surfdir.cross(plane.plane.norm));
                        return plane.get_texture()->get_property(x, y);
                }
```

球球由球心及半径唯一确定.球与光线求交时，利用球心到它在光线所在直线上的投影的距离判断是否相交,在根据投影位置及勾股定理计算交点. 求交时要考虑光线起始点在球内部的情形,以供计算相对折射率.
见include/geometry/sphere.hh, renderable/sphere.cc，代码如下

```
#include
"renderable/sphere.hh"
                #include "const.hh"
                using namespace std;
                shared_ptr<Trace> Sphere::get_trace(const Ray& ray, real
                dist) const {
                        auto ret = make_shared<SphereTrace>(*this, ray);
                        if (ret->intersect(dist)) {
                                if (dist == -1 or ret->intersection_dist()
                dist)
                                        return ret;
                        }
                        return nullptr;
                }
                AABB Sphere::get_aabb() const {
                        Vec diff = Vec(sphere.r, sphere.r, sphere.r),
                                min = sphere.center - diff,
                                max = sphere.center + diff;
```

```
        return AABB(min, max);
}
bool SphereTrace::intersect(real_t) const {
        if (!toward && !inside)      // ray leaves sphere
                return false;
        proj = ray.project(sphere.sphere.center);
        m_assert(isfinite(proj.x));
        sqrdiff = sqr(sphere.sphere.r) - (proj -
sphere.sphere.center).sqr();
        if (!inside && (sqrdiff < 0))  // dist > r
                return false;
        return true;
}
real_t SphereTrace::intersection_dist() const {
        if (!inside)
                inter_dist = (proj - ray.orig).mod() -
sqrt(sqrdiff);
        else if (toward)
                inter_dist = (proj - ray.orig).mod() +
sqrt(sqrdiff);
        else
                inter_dist = sqrt(sqrdiff) - (proj -
ray.orig).mod();
        m_assert(isfinite(proj.x));
        m_assert(isfinite(sqrdiff));
        m_assert(inter_dist > 0);
        inter_point = Trace::intersection_point();
        return inter_dist;
}
Vec SphereTrace::intersection_point() const {
        m_assert(isfinite(inter_point.x));
        return inter_point;
}
Vec SphereTrace::normal() const {
        Vec ret = inter_point - sphere.sphere.center;
        ret.normalize();
        if (!inside) return ret;
        else return -ret;
}
shared_ptr<Surface> SphereTrace::transform_get_property()
const {
        m_assert(fabs(sphere.north.sqr() - 1) < EPS);
        Vec norm = inter_point - sphere.sphere.center;
        Vec projxy = Vec(norm.x, norm.y, 0).get_normalized(),
                projyz = Vec(0, norm.y,
norm.z).get_normalized();
```

```
                real_t arg1 = acos(projxy.dot(sphere.north)),
                // 0 to pi
                        arg2 = acos(projyz.dot(sphere.north));
                return sphere.get_texture()->get_property(arg1 *
        sphere.sphere.r, arg2 * sphere.sphere.r);
        }
        real_t SphereTrace::get_forward_density() const {
                if (inside) return AIR_REFRACTIVE_INDEX;
                else return DEFAULT_REFRACTIVE_INDEX;
        }
```

三角面片三角面片用三个顶点坐标存储. 为了性能,与光线的求交参考了[1, 3]的算法及实现,其基本思想是求解满足方程

$$\overrightarrow{Orig}+t\overrightarrow{Dir}=x\overrightarrow{v_1}+y\overrightarrow{v_2}+(1-x-y)\overrightarrow{v_3}, t>0, x, y\in(0,1), x+y\leq 1$$

的(t, x, y)在求交时同时能得到交点的重心坐标 ,便于之后进行法向插值. 见include/renderable/face.hh, renderable/face.cc

轴平行包围盒轴平行包围盒(Axis Align Bounding Box)用最小坐标与最大坐标两个向量存储. 为了效率,包围盒与光线求交部分参考了的算法. 其思想是对每个面逐一计算交点位置, 再更新最近交点. 见geometry/aabb.hh

```
#pragma
once
        #include <utility>
        #include <iostream>
        #include <limits>
        #include "geometry/ray.hh"
        class AAPlane {
                public:
                        enum AXIS { AXIS_X = 0, AXIS_Y, AXIS_Z, ERROR};
                        AXIS axis;
                        real_t pos;
                        AAPlane():
                                axis(ERROR) {}
                        AAPlane(int _axis, real_t _pos):
                                axis(static_cast<AXIS>(_axis)), pos(_pos)
                        {}
        };
        // axis-aligned bounding box
        class AABB {
                public:
                        Vec min = Vec::max(),
                                max = -Vec::max();
                        AABB(){}
                        AABB(const Vec& _min, const Vec& _max):
                                min(_min), max(_max) {}
```

```
                void set(const Vec& vmin, const Vec& vmax) { min = vmin,
max = vmax; }
                Vec size() const { return max - min; }
                bool empty() const { return (min.x >= max.x or min.y >=
max.y or min.z >= max.z); }
                real_t area() const { return (max - min).area(); }
                bool contain(const Vec& p) const { return p < max && min <
p; }
                // override >
                //
                friend std::ostream& operator << (std::ostream & os, const
AABB& b)
                { return os << "min: " << b.min << " , max:" << b.max; }
                void update(const AABB& b) {
                        min.update_min(b.min);
                        max.update_max(b.max);
                }
                inline void update(const Vec& v) {
                        update_min(v);
                        update_max(v);
                }
                inline void update_min(const Vec& v) { min.update_min(v); }
                inline void update_max(const Vec& v) { max.update_max(v); }
                std::pair<AABB, AABB> cut(const AAPlane& pl) const {
                        AABB l = *this, r = *this;
                        if (!BETW(pl.pos, min[pl.axis] + 2 * EPS,
max[pl.axis] - 2 * EPS))
                                // !! otherwise, l.max or r.min can be equal
to *this.max / *this.min,
                                // resulting a same boundbox in child
                                throw "Outside";
                        l.max[pl.axis] = pl.pos;            // to loose
                        r.min[pl.axis] = pl.pos;
                        return std::make_pair(l, r);
                }
                // An efficient and robust ray-box intersection algorithm
                // Williams, etc. SIGGRAPH 2005
                bool intersect(const Ray& ray, real_t &mind, bool &inside)
{
                        if (empty())
                                return false;
                        Vec inv(1.0 / ray.dir.x, 1.0 / ray.dir.y, 1.0 /
ray.dir.z);
                        real_t t_max = std::numeric_limits<real_t>::max(),
                                t_min = -t_max;
#define UPDATE(t) \
```

```
                        do { \
                                if (fabs(ray.dir.t) > EPS) { \
                                        bool sign = (inv.t < 0); \
                                        real_t tmp_min = ((sign ? max : min).t
- ray.orig.t) * inv.t; \
                                        real_t tmp_max = ((sign ? min : max).t
- ray.orig.t) * inv.t; \
                                        ::update_max(t_min, tmp_min); \
                                        ::update_min(t_max, tmp_max); \
                                        if (t_min + EPS > t_max) return false;
\
                                } \
                        } while (0)
                        UPDATE(x); UPDATE(y); UPDATE(z);
#undef UPDATE
                        if (t_max < 0) return false;
                        if (t_min < 0) mind = t_max, inside = true;
                        else mind = t_min, inside = false;
                        if (ray.debug)
                                std::cout << "intersect with box " << (*this)
<< " at " << mind << ", inside: " << inside << std::endl;
                        return true;
                }
};
```

2.4 视图模型

一个视图应包括视点及矩形屏幕,且应使视点在屏幕的中轴线上. 视图类View存储了视点坐标,屏幕中心坐标,屏幕尺寸,屏幕边缘的两个方向向量, 并保证视点到屏幕中心的连线与屏幕边沿指向垂直. 这样可以方便的进行视图旋转,视图平移,缩放等导航操作.见include/view.hh, view.cc

view.hh

```
#pragma
once
#include <memory>
#include "geometry/geometry.hh"
#include "render/space.hh"
class View {
        private:
                const Geometry geo;
                inline void normalize_dir_vector()
                { dir_w.normalize(); dir_h.normalize(); }
                inline void restore_dir_vector() {
                        //  we should have:  |dir_w| * geo.w  == size
                        //  as well as:              |dir_w| == |dir_h|
```

```
                                dir_w = dir_w.get_normalized() * (size / geo.w);
                                dir_h = dir_h.get_normalized() * (size / geo.w);
                        }
                        const Space& sp;
                public:
                        Vec view_point;
                        Vec mid;
                        real_t size;          // length the img cover in the scene
                        Vec dir_w, dir_h;
                        Vec origin_norm;              // the initial view
                        bool use_dof = false;
                        bool use_bended_screen = false;
                        View(const Space& _sp, const Vec& _view_point,
                                        const Vec& _mid, real_t w, const Geometry&
        _geo);
                        void zoom(real_t ratio);      // r > 1: zoom in
                        void twist(int angle); // -180 ~ 180
                        void rotate(int angle); // -180 ~ 180
                        void orbit(int angle); // -180 ~ 180
                        void shift(real_t dist, bool horiz);
                        void move_screen(real_t dist);
                        Color render(int i, int j) const;    // i row j column
                        Color render_bended(int i, int j) const;
                        Color render_antialias(const Vec& dest, int sample) const;
        // render with n^2 sample at each pixel
                        Color render_dof(const Vec& dest) const;
                        const Geometry& get_geo() const
                        { return geo; }
        };



#include
"view.hh"
            using namespace std;
            View::View(const Space& _sp, const Vec& _view_point,
                                        const Vec& _mid, real_t _size, const Geo
            m_geo):
                                geo(m_geo), sp(_sp), view_point(_view_point),
            mid(_mid), size(_size) {
                    Vec norm = (view_point - mid).get_normalized();
                    origin_norm = norm;
                    Vec tmp;
                    if ((fabs(norm.y) > EPS) or (fabs(norm.x) > EPS))
                            tmp = Vec(-norm.y, norm.x, 0);
                    else
```

```
                tmp = Vec(-norm.z, 0, norm.x);
        dir_w = tmp;
        dir_h = norm.cross(dir_w);
        restore_dir_vector();
}
Color View::render_antialias(const Vec& dest, int sample) const {
        real_t unit = 1.0 / sample, unit2 = unit / 2;
        Color ret = Color::NONE;
        REP(dx, sample) REP(dy, sample) {
                Vec new_dest = dest - dir_h * (dx * unit + drand48() *
unit2) + dir_w * (dy * unit + drand48() * unit2);
                if (!use_dof) {
                        Ray ray(view_point, new_dest - view_point, 1, true);
                        Color diff = sp.trace(ray);
                        ret += diff;
                } else {
                        ret += render_dof(new_dest);
                }
        }
        ret = ret * (1.0 / ::sqr(sample));
        return ret;
}
Color View::render_dof(const Vec& dest) const {
        Vec intersec = view_point + (dest - view_point) *
DOF_SCREEN_DIST_FACTOR;
        real_t theta;
        Color dof_res = Color::NONE;
        REP(k, DOF_SAMPLE_CNT) {
                theta = drand48() * 2 * M_PI;
                Vec diff = dir_w * cos(theta) + dir_w * sin(theta);
                diff.normalize();
                diff = diff * (drand48() * DOF_SAMPLE_RADIUS);
                Vec neworig = intersec + diff;
                Ray ray(neworig, dest - neworig, 1, true);
                dof_res += sp.trace(ray);
        }
        dof_res = dof_res * (1.0 / DOF_SAMPLE_CNT);
        return dof_res;
}
Color View::render(int i, int j) const {
        if (use_bended_screen)
                return render_bended(i, j);
        Vec dest = mid + dir_h * (geo.h - 1 - i - geo.h / 2) + dir_w * (j -
geo.w / 2);
        return render_antialias(dest, ANTIALIAS_SAMPLE_CNT);
}
```

```
Color View::render_bended(int i, int j) const {
	Vec left_mid = mid - geo.w * dir_w / 2,
		up_mid = mid - geo.h * dir_h / 2,
		dir = (mid - view_point).get_normalized(),
		point_to_left = left_mid - view_point,
		point_to_up = up_mid - view_point;
	real_t dtheta = acos(dir.dot(point_to_left.get_normalized()));
	dtheta /= geo.w / 2;
	real_t dphi = acos(dir.dot(point_to_up.get_normalized()));
	dphi /= geo.h / 2;
	real_t theta = (j - geo.w / 2) * dtheta;
	real_t phi = (geo.h / 2 - i) * dphi;
	real_t r = (mid - view_point).mod();
	Vec new_norm = dir * cos(theta) + dir_w.get_normalized() *
sin(theta);
	new_norm = new_norm * cos(phi) + dir_h.get_normalized() * sin(phi);
	Vec dest = view_point + new_norm.get_normalized() * r;
	return render_antialias(dest, 1);
}
void View::twist(int angle) {
	normalize_dir_vector();
	Vec norm = view_point - mid;
	real_t alpha = M_PI * angle / 180;
	dir_w = dir_w * cos(alpha) + dir_h * sin(alpha);
	m_assert(fabs(dir_w.sqr() - 1) < EPS);
	dir_h = norm.cross(dir_w);
	restore_dir_vector();
}
void View::zoom(real_t ratio) {
	ratio = 1.0 / ratio;
	size *= ratio;
	Vec dir = (view_point - mid);
	view_point = mid + dir * ratio;
	restore_dir_vector();
}
void View::orbit(int angle) {
	normalize_dir_vector();
	real_t alpha = M_PI * angle / 180;
	Vec norm = (view_point - mid).get_normalized();
	norm = norm * cos(alpha) + dir_w * sin(alpha);
	view_point = mid + norm * (view_point - mid).mod();
	m_assert(fabs(dir_w.sqr()) - 1 < EPS);
	dir_w = -norm.cross(dir_h);
	restore_dir_vector();
}
void View::shift(real_t dist, bool horiz) {
```

```
        Vec diff = (horiz ? dir_w : dir_h) * dist;
        view_point = view_point + diff;
        mid = mid + diff;
}
void View::move_screen(real_t dist) {
        real_t old_dist_to_screen = (mid - view_point).mod();
        mid = view_point + (mid - view_point).get_normalized() *
(old_dist_to_screen + dist);
        size *= (old_dist_to_screen + dist) / old_dist_to_screen;
        restore_dir_vector();
}
void View::rotate(int angle) {
        normalize_dir_vector();
        real_t alpha = M_PI * angle / 180;
        Vec norm = (mid - view_point).get_normalized();
        norm = norm * cos(alpha) - dir_w * sin(alpha);
        mid = view_point + norm * (mid - view_point).mod();
        m_assert(fabs(dir_w.sqr()) - 1 < EPS);
        dir_w = norm.cross(dir_h);
        restore_dir_vector();
}
```

2.5 KD 树

KD 树是一种空间划分树,本用于在K 维空间中快速查找区间内的点, 可利用它在光线追踪中对物体及其包围盒进行索引. 原理是,另树中每个节点对应一个包围盒,选取一个恰当的轴平行平面将包围盒一分为二, 作为两个子节点,递归下去直至深度太大或节点内物体太少, 叶节点不仅存储包围盒, 也维护盒中物体. 注意与切分面相交的物体在左右节点中都应维护.

2.5.1 建树

传统的KD 树中,按照使切分面两边点的个数尽量接近的原则选取切分面,其理由是假设了各个点被查询的概率均等. 而在光线追踪中,一般采用面积启发式(SAH-based)的切分面选取方式,选取切分面使得

$H = S1N1 + SrNr$

最大化,其中$S1; Sr$ 为左右包围盒的表面积, $N1;Nr$ 为左右包围盒管理的物体个数. 这样的建树可以使KD 树在查询时效率更高.对于启发式的建树算法,若逐一枚举切分平面,计算包含物体个数, 需要$O(x^2)$ 复杂度(n 为物体个数),程序实现了中提供的$O(n \log 2\ n)$ 算法. 其优化方法是: 在每一维度上,将各物体按坐标大小顺次选做切分平面的坐标, 因而可增量的计算平面两边包含物体的个数.实现见KDTree::build.

对于含有20W 面片的龙模型6, 采用不同方法单线程建树的用时及在几个固定视角的渲染耗时如下(单位: 秒):

	建树	视角1	视角2	视角3	视角4	视角5
均分建树（终止条件：100层，15个）	0.6	1.93	2.51	3.26	4.38	5.89
SAH建树（终止条件：100层，20个）	5.41	0.29	0.37	0.45	0.59	0.78
SAH建树（终止条件，100层，15个）	7.82	0.24	0.33	0.41	0.52	0.68

注：1.建树时,以树深度及当前节点所管理的物体个数作为建树终止的判定条件.
2.此实验的视角1 为main.cc中test_kdtree()提供的视角,其余视角由视角1 zoom in 依次得到.
3.可在lib/kdtree.cc中通过注释KDTree::build()函数中相应代码可切换两种建树算法.
由表可见SAH 建树的查询效率有很大提高,但建树缓慢. 建树效率与渲染效率之间存在trade-off,可以通过改变终止条件来调整.
另外,不使用KD 树时,视角1 渲染时间约为800s. (不使用KD 树时, 各像素所需时间大致相同,可由部分渲染时间估算总时间).,具体实现见KdTree.cc KdTree.hh

```
#pragma
once
        #include <list>
        #include "geometry/geometry.hh"
        #include "geometry/aabb.hh"
        #include "renderable/renderable.hh"
        class KDTree : public Renderable {
                public:
                        struct Node;
                        class RenderWrapper {
                                public:
                                        rdptr obj;
                                        AABB box;
                                        RenderWrapper(const rdptr& _obj, const AAB
        _box):
                                                obj(_obj), box(_box){}
```

```
                        };
                        Node* root;
                        Vec bound_min = Vec::max(), bound_max = -Vec::max();
                        KDTree(const std::list<rdptr>& objs, const AABB& space);
                        ~KDTree();
                        shared_ptr<Trace> get_trace(const Ray& ray, real_t max_dist)
            const override;
                        AABB get_aabb() const override
                        { return AABB(bound_min, bound_max); }
                    private:
                        Node* build(const std::list<RenderWrapper*>& objs, const AABB&
            box, int depth);
                        AAPlane cut(const std::list<RenderWrapper*>& objs, int depth)
            const;
            };
```

KdTree.cc

```
 #include
 <algorithm>
                #include <future>
                #include "kdtree.hh"
                #include "const.hh"
                #include "lib/utils.hh"
                #include "lib/debugutils.hh"
                #include "lib/Timer.hh"
                using namespace std;
                struct KDTree::Node {
                        AABB box;
                        Node* child[2];
                        // TODO use enum type to identify leaf or non-leaf, save spa
                for child
                        union {
                                AAPlane pl;           // for non-leaf node
                                list<rdptr> objs; // for leaf node
                        };
                        Node(const AABB& _box, Node* p1 = nullptr, Node* p2 =
                nullptr) :
                                box(_box), child{p1, p2}, objs() { }
                        ~Node() {
                                delete child[0]; delete child[1];
                                if (leaf()) objs.~list<rdptr>();
                        }
                        bool leaf() const
                        { return child[0] == nullptr && child[1] == nullptr; }
                        void add_obj(const rdptr& obj)
                        { objs.push_back(obj); }
                        shared_ptr<Trace> get_trace(const Ray& ray, real_t inter_dis
```

```
real_t max_dist = -1) const {
                // call when know to intersect
                if (leaf()) {
                        // find first obj
                        real_t min = numeric_limits<real_t>::max();
                        shared_ptr<Trace> ret;
                        for (auto & obj : objs) {
                                auto tmp = obj->get_trace(ray, max_dist);
                                if (tmp) {
                                        real_t d = tmp-
>intersection_dist();
                                        if (update_min(min, d)) ret = tmp;
                                }
                        }
                        return ret;
                }
                real_t min_dist = -1;
                bool inside;
                real_t pivot = ray.get_dist(inter_dist)[pl.axis];
                int first_met = (int)(pivot > pl.pos);
                for (int index : {first_met, 1 - first_met})  {
        // iterate over two children
                        Node* ch = child[index];
                        if (!ch or !ch->box.intersect(ray, min_dist,
inside)) ch = nullptr;
                        if (ch && (max_dist == -1 or min_dist <
max_dist)) {
                                auto ret = ch->get_trace(ray, min_dist,
max_dist);
                                if (ret) {
                                        if (ray.debug) {
                                                cout << "ray intersect
with box " << ch->box << endl;
                                                cout << "interpoint: " <<
ret->intersection_point() << endl;
                                                cout << endl;
                                        }
                                        if (!ch->leaf())      // then
definitely contain, directly return
                                                return ret;
                                        Vec inter_point = ret-
>intersection_point();
                                        if (ch->box.contain(inter_point))
                                                return ret;
                                } else if (ray.debug)
                                        cout << "not intersect with box"
```

```
                << ch->box << endl;
                        }
                }
                if (ray.debug)
                        cout << "reaching end of node->get_trace()" <<
endl;
                return nullptr;
        }
};
KDTree::KDTree(const list<rdptr>& objs, const AABB& space) {
        list<RenderWrapper*> objlist;
        for (auto & obj : objs) {
                m_assert(!obj->infinity());
                AABB aabb = obj->get_aabb();
                objlist.emplace_back(new RenderWrapper(obj, aabb));
                bound_min.update_min(aabb.min),
bound_max.update_max(aabb.max);
        }
        Timer timer;
        root = build(objlist, space, 0);
        for (auto objptr : objlist)
                delete objptr;
        printf("Build tree spends %lf seconds\n", timer.get_time());
}
KDTree::~KDTree() { delete root; }
shared_ptr<Trace> KDTree::get_trace(const Ray& ray, real_t max_dist)
const {
        if (!root) return nullptr;          // empty kd-tree
        real_t min_dist; bool inside;
        if (!(root->box.intersect(ray, min_dist, inside)))
                return nullptr;
        if (min_dist > max_dist and max_dist >= 0) return nullptr;
        return root->get_trace(ray, min_dist, max_dist);
}
AAPlane KDTree::cut(const list<RenderWrapper*>& objs, int depth) const
{
        AAPlane ret(depth % 3, 0);
        vector<real_t> min_list;
        for (auto objptr : objs)
                min_list.push_back(objptr->box.min[ret.axis]);
        nth_element(min_list.begin(), min_list.begin() +
min_list.size() / 2, min_list.end());
        // partial sort
        ret.pos = min_list[min_list.size() / 2] + 2 * EPS;        //
SEE what happen
        return ret;
```

```
}
KDTree::Node* KDTree::build(const list<RenderWrapper*>& objs, const
AABB& box, int depth) {
	if (objs.size() == 0 or depth > KDTREE_MAX_DEPTH) return
nullptr;
#define ADDOBJ \
	for (auto objptr : objs) ret->add_obj(objptr->obj)
	Node* ret = new Node(box);
	int nobj = objs.size();
	if (nobj <= KDTREE_TERMINATE_OBJ_CNT) {
		ADDOBJ;
		return ret;
	}
	pair<AABB, AABB> par;
	AAPlane best_pl;
	real_t min_cost = numeric_limits<real_t>::max();
/**
 *	// algorithm 1 (naive kdtree)
 *	best_pl = cut(objs, depth);
 *	try {
 *		par = box.cut(best_pl);
 *	} catch (...) {
 *		ADDOBJ;
 *		return ret;		// pl is outside box, cannot go further
 *	}
 *	ret->pl = best_pl;
 *
 *	vector<RenderWrapper> objl, objr;
 *	for (auto obj : objs) {
 *		if (obj->box.max[best_pl.axis] >= best_pl.pos - EPS)
objr.push_back(obj);
 *		if (obj->box.min[best_pl.axis] <= best_pl.pos + EPS)
objl.push_back(obj);
 *	}
 *	// end of algo 1
 */
	// algorithm 2 (SAH kdtree)
	// "On building fast kd-trees for ray tracing, and on doing
that in O (N log N)"
	// Wald, Ingo and Havran, Vlastimil
	// O(n log^2(n)) build
	typedef pair<real_t, bool> PDB;
	auto gen_cand_list = [&objs](int dim) {
		vector<PDB> cand_list;
		for (auto objptr: objs)
			cand_list.emplace_back(objptr->box.min[dim] -
```

```
                EPS, true),
                        cand_list.emplace_back(objptr->box.max[dim] +
EPS, false);
                sort(cand_list.begin(), cand_list.end());
                return cand_list;
        };
        future<vector<PDB>> task[3];
        const int THREAD_DEPTH_THRES = 1;
        bool parallel = depth < THREAD_DEPTH_THRES;
        if (parallel) REP(dim, 3)
                task[dim] = async(launch::async, bind(gen_cand_list,
dim));
        REP(dim, 3) {
                // true: min, false: max
                vector<PDB> cand_list =
                        parallel ? task[dim].get() : gen_cand_list(dim);
                int lcnt = 0, rcnt = nobj;
                auto ptr = cand_list.begin();
                do {
                        AAPlane pl(dim, ptr->first);
                        try {
                                auto par = box.cut(pl);
                                real_t cost = par.first.area() * lcnt +
par.second.area() * rcnt;
                                if (lcnt == 0 or rcnt == 0 or (lcnt +
rcnt == nobj - 1 && rcnt == 1))
                                        cost *= 0.8;          // this is a
hack
                                if (update_min(min_cost, cost)) best_pl =
pl;
                        } catch (...) {}
                        if (ptr->second) lcnt ++; else rcnt --;
                        auto old = ptr++;
                        while (ptr != cand_list.end() && ptr->first -
old->first < EPS) {
                                if (ptr->second) lcnt ++;
                                else rcnt --;
                                old = ptr++;
                        }
                } while (ptr != cand_list.end());
        }
        if (best_pl.axis == AAPlane::ERROR) { // didn't find best_pl
                ADDOBJ;
                return ret;
        }
        // fount the best cutting plane
```

```
                ret->pl = best_pl;
                par = box.cut(best_pl);
                list<RenderWrapper*> objl, objr;
                for (auto obj : objs) {
                        if (obj->box.max[best_pl.axis] >= best_pl.pos)
objr.push_back(obj);
                        if (obj->box.min[best_pl.axis] <= best_pl.pos)
objl.push_back(obj);
                }
                // end of algorithm 2
                if (parallel) { // parallel when depth is small
                        future<Node*> lch_future = async(launch::async,
                                        [&]() {
                                                return build(objl, par.first,
depth + 1);
                                        });
                        future<Node*> rch_future = async(launch::async,
                                        [&]() {
                                                return build(objr, par.second,
depth + 1);
                                        });
                        ret->child[0] = lch_future.get();
                        ret->child[1] = rch_future.get();
                } else {
                        ret->child[0] = build(objl, par.first, depth + 1);
                        ret->child[1] = build(objr, par.second, depth + 1);
                }
                // might fail to build children
                if (ret->leaf()) ADDOBJ;            // add obj to leaf node
                return ret;
#undef ADDOBJ
}
```

2.5.2 求交

光线KD 树中物体求交的基本方法是,递归寻找两子树中最近物体(而不是叶子节点的
包围盒) 的交点,取较近者为结果返回.

实现时,在每个节点处保存了两子节点的切分平面, 这样可以预先判断出首先与光线相交的包围盒, 若光线与此包围盒内物体相交, 则不用考虑另一包围盒, 提高了效率. 使用此方法应注意,若光线与较近包围盒所管理的物体相交,应确认与最近物体的交点在此包围盒内.因为若不包含,则光线首先打到的物体可能并不是此物体,而是第二个包围盒管理的物体.

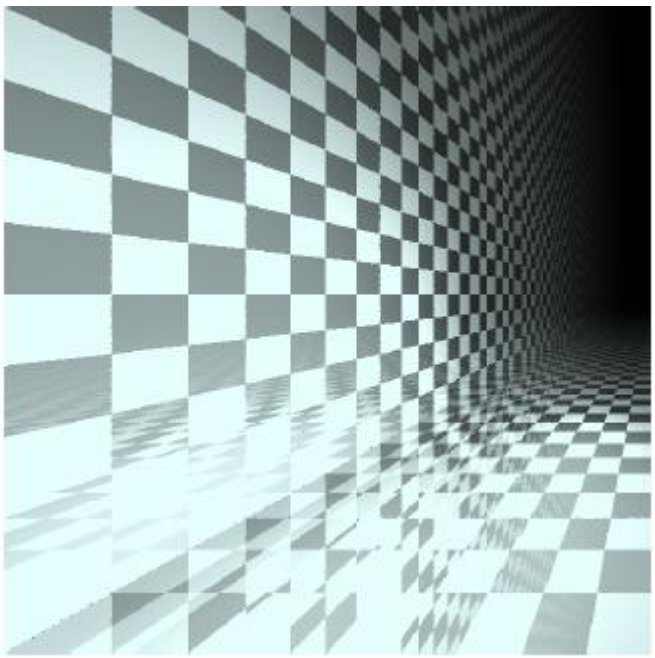
2.6 纹理

一种纹理相当于一个二维坐标到表面属性的映射. 程序中的“表面属性”除包括Phong模型参数外,还包括了用于折射判定的透明度,以及全局光照模型需要使用的发光强度.见include/material/surface.hh. 程序实现了均匀纹理、网格纹理、图片纹理三类纹理,并可通过继承Texture类进行扩展.

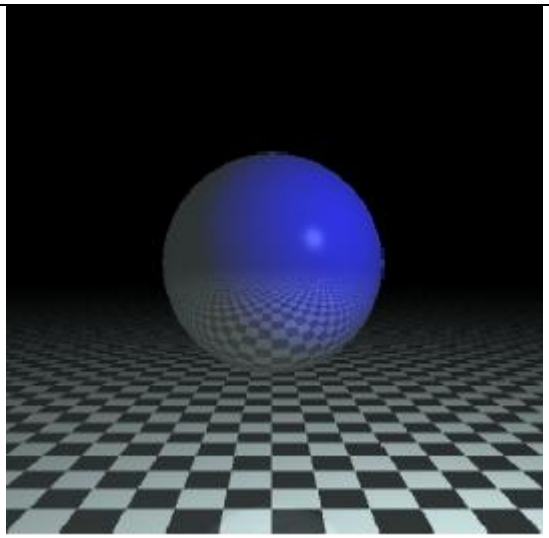
2.7 法向插值

法向插值可以使三角网格表面更光滑,只需找到一个面片上的连续函数,就可以得到较好的效果. 此程序采用了线性插值.在读入网格数据后,令各顶点法向为其相邻各面法向的平均值. 在面片与光线求交后,据交点的重心坐标及面片各顶点的法向,即可插值出交点处法向. 效果见下图

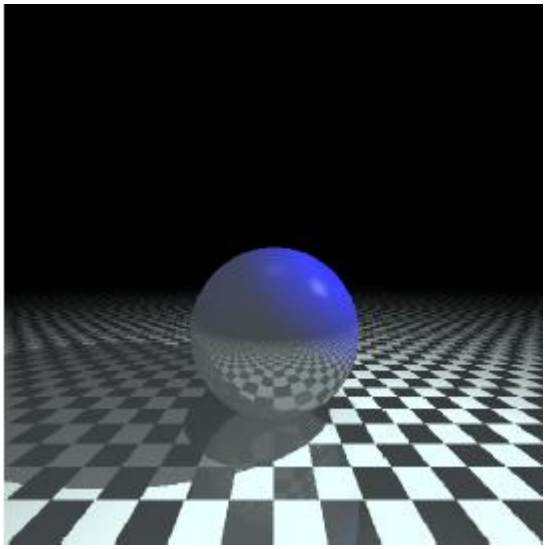
1. 考虑了Phong 模型中的高光,并对$x - y$ 平面和$y - z$ 平面加入了反射系数,有了互相反射的效果,并且左下部分能够看到高光.



2: 使用球模型,可以看到明显的高光效果。同时由于球面specular参数高,使得球下部反射了平面。

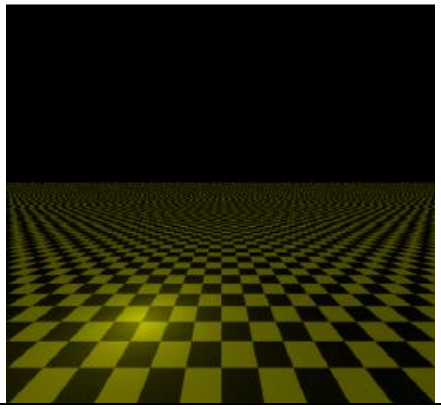


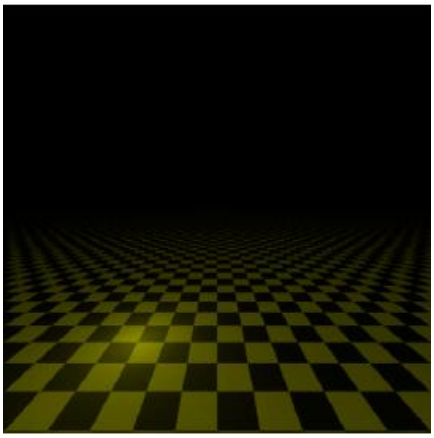
3:对于找到的交点,判断它与光源之间是否被挡住,若被挡住就不计算漫反射和高光. 这样实现了阴影效果



2.8 抗锯齿

1. 使用Beer-Lambert 定律,使得光线亮度按传播距离指数衰减,可以有效消除远处纹理密集处的畸形. 见下图对比





2. 使用全屏抗锯齿(FSAA),对图片整体应用卷积盒

$$\begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}$$

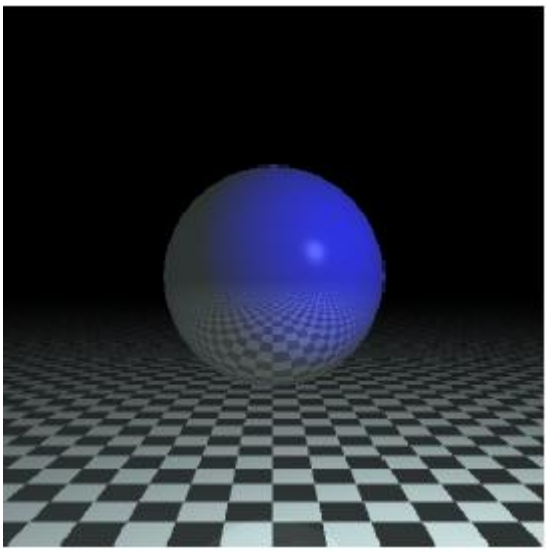
消除直线锯齿的同时使图片模糊,影响视觉效果.

3. 对每一像素,计算其与周围像素距离平方之和,若小于某一阈值则应用如上卷积盒,有效果. 见CVRender::antialias()

2.9 Phong 模型中的软阴影

对场景中每一点光源,将其替换为点周围的多个密集点光源以模拟面光源的效果, 即

可实现软阴影.效果如下图所示, 比起Path Tracing 的软阴影的话还是十分不真实.



2.10 景深

程序中景深7 的实现方法为,以焦平面作为屏幕,取焦平面与视点之间某处建一感光器平面. 对视点到焦平面的每条光线,在它与感光器平面的交点周围随机采多个样本点,以这些样本点为观察点向屏幕同一位置发射光线并执行光线追踪,以各样本点颜色的平均值作为最终颜色. 这样就可以使得焦平面上物体清晰,而其余位置模糊.main.cc中的dof_ball_scene()生成一个演示景深的场景,场景中可通过键盘控制焦平面位置随机取点会造成焦平面以外有无规律噪点,在生成视频后会比较明显, 因而对输出图像统一做了高斯模糊,使噪点不太明显.

2.11 小量处理

在如下情形需要特别注意实数运算中的误差.

1. 反射折射时的交点判定

考虑光线斜射平面的情形,若由于误差, 交点落在了平面异侧,则反射光会再次打到平面, 若落在了平面同侧,则透射光会再次打到平面.因而计算反射光线时, 应将其起始点回退EPS,计算透射光线时,应将其起始点前进EPS.

2. 平行判定

判定直线与面片或平面是否平行时,利用直线与法线点乘的绝对值<EPS 判定,否则可能导致交点坐标过大, 使其他计算溢出.

3. KD 树

建树时,对于物体与包围盒相交的判定应略微宽松,与包围盒距离<EPS 的物体都应归入包围盒管理, 查询时对光线与面片求交也可判的宽松一些,否则渲染的图片中容易出现黑点.对面片求包围盒时应注意最小值应减去EPS,最大值应加上EPS, 否则可能出现0 体积包围盒,影响算法实现.

2.12 网格简化

网格的坍缩简化算法参考实现. 基本思路是,对每对顶点估算出简化代价,每次选取代价最小的顶点对执行简化操作, 操作后更新相关的代价值.由于算法具有优先队列结构,因此使用了std::priority_queue进行堆加速. 但由于需要执行堆元素修改操作,但STL 库不支持, 因而对堆结构进行了如下处理:

对每个顶点,维护它相邻顶点中最适合坍缩的一个顶点指针. 堆元素存储一顶点指针, 它与相应邻点的坍缩代价,以及一时间戳. 堆元素按照坍缩代价保持堆性质,坍缩一对顶点后更新了附近顶点的最优代价,就将新的值打上新的时间戳插入堆中, 而取堆顶元素时,若从时间戳发现其不是最新就抛弃. 这样就可以替代修改操作.见include/mesh_simplifier.hh, mesh_simplifier.cc

Mesh.cc

```
#include
<algorithm>
                #include "mesh_simplifier.hh"
                #include "renderable/mesh.hh"
                #include "lib/objreader.hh"
                #include "lib/Timer.hh"
                using namespace std;
```

```
Mesh::Mesh(std::string fname, const Vec& _pivot, real_t _zsize, const
shared_ptr<Texture>& _texture):
	Renderable(_texture), pivot(_pivot), zoom_size(_zsize) {
	ObjReader::read_in(fname, this);
	cout << "nvtx = " << vtxs.size() << ", nface = " <<
face_ids.size() << endl;
	transform_vtxs();
}
void Mesh::simplify(real_t ratio) {		// only require face_ids
	Timer timer;
	MeshSimplifier s(*this, ratio);
	s.simplify();
	printf("Simplification spends %lf seconds\n", timer.get_time());
	cout << "after simplification: nvtx = " << vtxs.size() << ",
nface = " << face_ids.size() << endl;
}
void Mesh::transform_vtxs() {
	Vec sum;
	real_t zfactor = zoom_size / (bound_max - bound_min).get_max();
	// * factor
	for (auto &k : vtxs) sum = sum + k.pos;
	sum = pivot - sum / vtxs.size();
	for (Vertex& v : vtxs)
		v.pos = pivot + (v.pos + sum - pivot) * zfactor;
	bound_min = pivot + (bound_min + sum - pivot) * zfactor;
	bound_max = pivot + (bound_max + sum - pivot) * zfactor;
}
void Mesh::finish() {		// build tree, calculate smooth norm
	if (smooth) {		// calculate vtx norm
		if (face_ids.size() < 30) {
			printf("Number of faces is too small, cannot use
smooth!\n");
			smooth = false;
		} else {
			int nvtx = vtxs.size();
			struct NormSum {
				Vec sum = Vec(0, 0, 0);
				int cnt = 0;
				void add(const Vec& v) { sum = sum + v, cnt
++; }
			};
			NormSum* norm_sum = new NormSum[nvtx];
			for (auto & t : face_ids) {
				int a = t[0], b = t[1], c = t[2];
				Vec tmp_norm = Triangle(vtxs[a].pos,
vtxs[b].pos, vtxs[c].pos).norm;
```

```
                                        // norms might be in opposite direction ?
                obj gives vertexes in correct order
                                        norm_sum[a].add(tmp_norm);
                                        norm_sum[b].add(tmp_norm);
                                        norm_sum[c].add(tmp_norm);
                                }
                                REP(k, nvtx)
                                        vtxs[k].norm = norm_sum[k].sum /
                norm_sum[k].cnt;
                                delete[] norm_sum;
                        }
                }
                for (auto &ids : face_ids) add_face(ids);
                if (use_tree)
                        tree = make_shared<KDTree>(list<rdptr>(begin(faces),
                end(faces)), get_aabb());
                }
                shared_ptr<Trace> Mesh::get_trace(const Ray& ray, real_t dist) const {
                if (use_tree)
                        return tree->get_trace(ray, dist);
                shared_ptr<Trace> ret;
                real_t min = numeric_limits<real_t>::max();
                for (auto & face : faces) {
                        auto tmp = face->get_trace(ray, dist);
                        if (tmp && update_min(min, tmp->intersection_dist()))
                                ret = tmp;
                }
                return ret;
                }
```

Mesh.hh

```
#pragma
once
           #include <vector>
           #include <unordered_set>
           #include <queue>
           #include "renderable/mesh.hh"
           using std::vector;
           using std::unordered_set;
           using std::priority_queue;
           class MeshSimplifier {
                  private:
                         Mesh& mesh;
                         struct Vertex;
                         struct Face {
```

```
                        Vertex * vtx[3];
                        Vec norm;
                        Face(Vertex* a, Vertex* b, Vertex* c): vtx{a, b, c}
                        { norm = (c->pos - a->pos).cross((b->pos - a-
>pos)).get_normalized(); }
                        // delete this face (containing u, v) when collapsing
from u to v
                        void delete_from(Vertex* u, Vertex* v);
                        // change vertex u to v when collapsing from u to v
                        void change_to(Vertex* u, Vertex* v);
#ifdef DEBUG
                        inline bool contain(Vertex* v) const
                        { return (v == vtx[0]) + (v == vtx[1]) + (v == vtx[2])
== 1; }
                        inline int count(Vertex* v) const
                        { return (v == vtx[0]) + (v == vtx[1]) + (v ==
vtx[2]); }
#endif
                };
                struct Vertex {
                        Vec pos;
                        int id = -1;
                        bool erased = false;
                        unordered_set<Vertex*> adj_vtx;
                        unordered_set<Face*> adj_face;
                        int cost_timestamp = 0;
                        real_t cost;
                        Vertex* candidate = nullptr;
                        Vertex(Vec _pos, int _id = -1):pos(_pos), id(_id) {}
                        inline void add_face(Face* f) {
                                adj_face.insert(f);
                                REP(k, 3) if (f->vtx[k] != this)
                                        adj_vtx.insert(f->vtx[k]);
                        }
                        // change adj_vtx, u to v
                        void change_to(Vertex* u, Vertex* v);
                };
                struct Elem {
                        // element type to use in the heap
                        Vertex* v;
                        int cost_timestamp;
                        Elem(Vertex* _v) :
                                v(_v), cost_timestamp(v->cost_timestamp) {}
                        bool operator < (const Elem& r) const
                        { return v->cost > r.v->cost; }
                        inline bool outofdate() const
```

```
                { return cost_timestamp < v->cost_timestamp; }
        };
        vector<Vertex> vtxs;
        vector<Face> faces;
        int target_num;                    // shrink to this number
        priority_queue<Elem> heap;
        real_t cost(Vertex*, Vertex*) const;
        void update_cost(Vertex*);
        int collapse(Vertex*, Vertex*);            // return the
number of faces it simplified
        void do_simplify();
        void write_back();
    public:
        MeshSimplifier(Mesh& mesh, real_t _ratio);
        void simplify();
};
```

结论分析与体会：通过这次实验，我认识到了学习一门新的编程语言需要耗费多么大的精力。课上学习轻松，但是课下做实验很难。所以在这方面还要提示。

更重要的是学会了做事情的思路，解决核心问题之前先解决核心问题的附属问题而不是一上来就直接去解决核心问题。就拿这个图形学实验来说，main.cc 是核心问题，直接决定了最后的展示效果，如果一上来就写 main.cc 会发现 sphere, ray, plane, phong, surface, trace, kdtree, mesh 等这些一系列需要地部分都没定义好。所以先将问题分解成一个又一个模块，先解决枝叶问题，即先把 phere, ray, plane, phong, surface, trace, kdtree, mesh 的等都定义好，再去写 main，这时写 main 无论需要用什么都可以直接调用。