装 订 线

同濟大學

TONGJI UNIVERSITY

# 计算机图形学课程项目报告

课程名称 数据库系统原理

学院(系) 电子与信息工程学院

专　　业 计算机科学与技术

组员 1 朱可仁 学号 1452286

组员 2 周宇星 学号 1352652

组员 3 章鹏飞 学号 1452233

组员 4 张翔 学号 1452229

2016-2017 学年 第 1 学期

# 目 录



装 订 线

# 1 引言

## 1.1 项目的背景

精美的 CG 效果图，与真实世界毫无区别的电影视觉效果，我们对这些并不陌生。而在游戏中对水面之类的场景我们也不陌生，不过它所生成的画面效果，好像永远都不那么真实。即使人们尽再大的努力，它的画面始终还是动画，和人们心目中的“电影级别的画质”总是差那么一点。这是因为，我们目前的游戏，无一例外都在使用光栅化算法。而在这些电影中，则采用的是光线追踪算法。在 3DSMax、Maya、SoftimageXSI 等软件中，也都无一例外地采用了这一算法。

光线追踪技术是由几何光学通用技术衍生而来。它通过追踪光线与物体表面发生的交互作用，得到光线经过路径的模型。简单地说，3D 技术里的光线追踪算法，就是先假设屏幕内的世界是真实的，显示器是个透明的玻璃，只要找到屏幕内能透过人眼的光线，加以追踪就能构建出完整的 3D 画面。

光线追踪算法分为两种：正向追踪算法和反向追踪算法。其中，正向追踪算法是大自然的光线追踪方式，即由光源发出的光经环境景物间的多次反射、透射后投射到景物表面，最终进入人眼。反向追踪算法正好相反，它是从观察者的角度出发，只追踪那些观察者所能看见的表面投射光。就目前而言，所有 3D 制作软件的光线追踪算法都是采用反向追踪法，原因是这种算法能够最大程度地节省计算机的系统资源，而且不会导致渲染质量的下降。

我们以反向追踪法为基础，实现了一个支持场景描述、光线反射、软阴影、材质贴图的简易光线追踪器。

## 1.2 本文所作的工作

本文从项目背景，整体架构，物体模型，物体与射线碰撞检测，光线反射，阴影处理，物体材质纹理，并行优化介绍了本小组课程项目：简易光线追踪器的实现。在项目实现过程中，小组成员通力合作，主要分工为：

朱可仁：摄像机，程序总体架构，程序调试。

张　翔：光线追踪算法的反射，阴影处理，程序调试。

周宇星：物体的模型构建、与射线碰撞检测、材质纹理，程序调试。

章鹏飞：场景描述文件，并行优化，程序调试。

装 订 线

# 2 整体架构

## 2.1 程序框架

本项目程序主要分为两大部分：外部交互，算法实现。
其中外部交互主要负责读入算法所需参数、输出算法渲染结果。
算法实现主要负责根据输入的参数，完成光线追踪渲染。

## 2.2 交互模块简介

1.输入：程序通过运行参数获取配置文件，包括了摄像机参数及物体模型描述：

Rule := command-name: value1 <value2> <value3> ...
# comment

**Camera**

```
resolution:   640 480 256
file-name:    scene1.ppm
camera-origin: 2 -5 3
camera-focus: -0.5 1 -.5
camera-zoom:  1
```

**Object**

```
object: plane
origin: 0 0 0
normal: 0 0 1
texture: checker
texture-scale: 2
color: .1 .1 .1
color2: 1 1 1
reflection: 0
```

2.输出：程序渲染的结果在执行目录下保存为.ppm 格式文件。

## 2.3 算法模块简介

本项目程序的算法实现主要分为以下两大部分：
（1）3D 基础类，
为方便之后核心算法的实现，我们定义了一个 3 维向量类，并定义函数或重载运算符实现了取模、点积、叉积等常用运算，具体内容如下：

```
class Vec {
private:
    float x[3];
public:
    float AngleX();
    float AngleY();
    float& operator[] (int i);
    Vec();
    Vec(float a, float b, float c);
    void set(float a, float b, float c);
    float magnitude();
    void normalize();
    Vec GetNormalized();
    Vec cross(Vec v);
    Vec blend(Vec, float);
```

```
        float dot(Vec v);
        float max();
        Vec(const Vec &v);
        Vec& operator=(const Vec &v);
        Vec operator+(const Vec &v);
        Vec operator-(const Vec &v);
        Vec operator*(const float &f);
        Vec operator/(const float &f);
        Vec operator*=(const float &f);
        void print();
};
```

（2）光线追踪核心算法类

此类实现了光线追踪渲染所需的各项计算，并以涉及对象具体类型划分，具体如下：

```
class Texture {
public:
        Texture();
        enum Pattern{NONE,CHECKER,IMAGE};
        Vec color;
        Vec color2;
        Vec color3;
        float reflection;
        float opacity;
        float refraction;
        Pattern pattern;
        float scale;
        fimage* image;
        float imageVert[3][2];
        float intensity;
};
class Object {
public:
        enum Geometry{NONE,PLANE,TRIANGLE,POLY,SPHERE};
        Texture texture;
        Vec origin;
        virtual void Speak();
        virtual hit Trace(Vec, Vec);
        virtual void SetNormal(Vec);
        virtual Vec GetNormal();
        virtual void SetNormal(float, float, float);
        virtual void Setu(Vec);
        virtual void SetRadius(float);
        virtual void SetIntensity(float);
        virtual void AddVert(float, float, float);
        virtual void AddVert(Vec);
        virtual void SetVert(int, float, float, float);
        virtual void SetVert(Vec, Vec, Vec);
        virtual Vec Color(hit);
};
```

装 订 线

```
struct hit {
	Vec location;
	Vec normal;
	float distance;
	bool contact;
	Object *object;
	Vec barycentric;
	hit(Object * o, Vec l, float d, Vec n) {
		object = o;
		location = l;
		normal = n;
		distance = d;
		contact = 1;
	}
	hit(Object * o, Vec l, float d, Vec n, Vec b) {
		barycentric = b;
		object = o;
		location = l;
		normal = n;
		distance = d;
		contact = 1;
	}
	hit(bool c) {
		contact = c;
		object = NULL;
	}
};
class Plane : public Object {

	Vec normal;
	Vec u; // rotation
public:
	hit Trace(Vec o, Vec d);
	void SetNormal(Vec v);
	void SetNormal(float x, float y, float z);
	Vec GetNormal();
	void Setu(Vec v);
	Vec Getu();
	Vec Color(hit);
	void Speak();
	void SetIntensity(float);
};
class Triangle : public Object {
public:
	Vec vert[3];
	Vec normal;
	Geometry type();
	void Speak();
	void AddVert(float, float, float);
	void AddVert(Vec);
	void SetVert(int, float, float, float);
	void SetVert(Vec x, Vec y, Vec z);
	void CalcNorm();
	hit Trace(Vec o, Vec d);
```

装 订 线

```
        Vec Color(hit);
        bool Inside(Vec v);
        Vec Barycentric(Vec);
        void SetIntensity(float);
};
class Poly : public Object {
public:
        Vec *vertices;
        float nVertices;
        Vec normal;
        void Speak();
        void SetIntensity(float);
};
class Sphere : public Object {
public:
        float radius;
        Vec u; //rotation
        Vec v; //rotation
        void SetRadius(float r);
        hit Trace(Vec o, Vec d);
        Vec Color(hit);
        void Speak();
        void SetIntensity(float);
};
class Light : public Object {
public:
        float intensity;
        void SetIntensity(float);
        void SetRadius(float r);
        Vec RandomSpot();
        float radius;
        Light();
};
```

装 订 线

# 3 物体模型

本项目程序支持球体、三角形、平面三种实际物体模型，下文分别介绍其构建及与射线的碰撞检测。

## *3.1 球体*

在我们的光线追踪算法中，球体一般不以三角形近似。

考虑球体的表面，中心点为 c、半径为 r 的球体表面可用等式表示：

$$\|\mathbf{x}-\mathbf{c}\| = r$$

需要计算光线和球体的最近交点，只要把光线 x = r(t)代入球体等式，把该等式求解就是交点。为简化方程，设 v=o - c，则:

$$\begin{aligned}
\|\mathbf{v}+t\mathbf{d}\| &= r \\
\|\mathbf{v}+t\mathbf{d}\|^2 &= r^2 \\
(\mathbf{v}+t\mathbf{d})\cdot(\mathbf{v}+t\mathbf{d}) - r^2 &= 0 \\
\mathbf{v}^2 + 2t(\mathbf{d}\cdot\mathbf{v}) + t^2\mathbf{d}^2 - r^2 &= 0 \\
(\mathbf{d}^2)t^2 + (2\mathbf{d}\cdot\mathbf{v})t + (\mathbf{v}^2 - r^2) &= 0
\end{aligned}$$

因为 d 为单位向量，所以二次方的系数可以消去。 t 的二次方程式的解为：

$$\begin{aligned}
t &= \frac{-2\mathbf{d}\cdot\mathbf{v} \pm \sqrt{(2\mathbf{d}\cdot\mathbf{v})^2 - 4(\mathbf{v}^2 - r^2)}}{2} \\
&= -\mathbf{d}\cdot\mathbf{v} \pm \sqrt{(\mathbf{d}\cdot\mathbf{v})^2 - (\mathbf{v}^2 - r^2)}
\end{aligned}$$

若根号内为负数，即相交不发生。另外，由于这里只需要取最近的交点，因此正负号只需取负号。

## *3.2 平面*

光线和平面的位置关系有平行（重合）、远离或相交（且交点唯一）
利用平面法线和光线方向向量点乘，容易判断出位置关系。
再利用投影，就可以求出碰撞点的位置。

## *3.3 三角形*

三角形可以先用平面的方法处理，求出碰撞点后，再判断是否位于三角形内部。
利用点积、叉积的几何性质，求出有向面积，有以下判断方式：

```
float ABC = normal.dot((vert[1] - vert[0]).cross(vert[2] - vert[0]));
float PBC = normal.dot((vert[1] - location).cross(vert[2] - location));
float PCA = normal.dot((vert[2] - location).cross(vert[0] - location));
float u = PBC / ABC;
float v = PCA / ABC;
```

判断 u,v,1-u-v 是否均非负即可。

## 4 反射计算

本项目采用光线逆向追踪。顾名思义，逆向追踪的光线从视点出发，经过成像平面上的某个像素，经过数次反射、折射，最终到达光源。对那些到达不了光源的光线，我们舍弃它们。虽然这样同样会产生无用计算，但是相比正向光线追踪对性能的影响，要小得多。通过控制递归层数，我们认为经过 m（m 是我们给定的常数）次反射后的光线到达不了光源，从而停止递归，舍弃这些光线。

```
if (ob->texture.reflection > 0.0 && depth > 0) {
    Vec reflect = d - (closestHit.normal * 2.0 * d.dot(closestHit.normal));
    reflect = Cast(closestHit.location,reflect,depth-1);
    PixelColor = PixelColor  + (reflect * ob->texture.reflection * 0.75);
}
```

由于我们进行光线追踪最后得到的是一张图片，图片是由一个个像素构成的，每个像素的颜色，是通过这个像素点的光线的颜色决定的。我们对每个像素点，将它分成 16 个子像素，进行 16 次采样取颜色平均值，以达到抗锯齿的效果，使画面更加自然。

```
if (sampleMethod == 0 || superSample == 1.0){
    for (float m1 = 0; m1 < superSample; m1++) {
        for (float m2 = 0; m2 < superSample; m2++) {
            float mx = (float)x + m1 / superSample;
            float my = (float)y + m2 / superSample;
            getPixelVector(mx,my,&o,&d);
            co = co + Cast(o, d, recursion);
        }
    }
```

首先我们要确定光线的方向。由于过视平面中点的法线通过了视点，可以构造一个由视点指向视平面中点的向量，再加上视平面指向当前采样像素点的向量，便可得到光线的方向向量。

对于射出（沿直线传播过程中）无法触碰到场景里的任何物体的光线，我们认为它射向了天空，从而根据视仰角，决定它在成像平面的颜色。这个颜色是由蓝（天空）向白（地平面）渐变的过程。

```
if (closestHit.contact == 0) {
    PixelColor[0] = PixelColor[1] = 1.0 - d[2] * 1.0;
    PixelColor[2] = 1.0;
    return PixelColor;
} else {
```

对于能触碰到物体的光线，我们认为这条光线由两部分组成：已经经过数次反射，且射向这个入射点发生反射的光线，且反射光线的路径与当前逆向追踪的光线路径重合；光源直接打到入射点，经过一次反射直接射入眼睛的光线。对于前者，我们递归地进行计算，值得注意的是，当一种颜色为 A 的光线打在材质颜色为 B 的物体上，反射的光线的颜色应当是颜色向量 A 与颜色向量 B 每个分量做乘法，重新映射到[0,1]区间上的结果。同时，反射光线的亮度还和材质的反射率有关，要与反射率做一些列的运算。对于后者，我们计算每个光源发出的光线的反射情况。如果一个光源和视点不在反射平面的同侧，这个光源对当前采样的像素点颜色是没有贡献的。如果在同侧，那么我们认为它发出的的光线对颜色的贡献和当前追踪的光线夹角有关，夹角越小，亮度越强。

```
float b = h->normal.GetNormalized().dot(lightDir.GetNormalized());
float c = h->normal.dot(*d);
if ((b < 0.0) == (c < 0.0)) //If the light is on the opposite side o
    return 0.0;
float lightDist = lightDir.magnitude();
hit obstacle = Closest(h->location, lightDir);
if (obstacle.contact && obstacle.distance < lightDist)
    return 0.0;
return fabs(b);
```

产生贡献的光线的颜色是由光源颜色、光线强度、亮度三者进行一系列的运算得到的。其中的亮度我们也对它进行若干次采样取平均值。

```
brightness = brightness + ((Lights[i]->texture.color * Lights[i]->intensity)
                           * (brightness2 / (float)(init + addSamples)));
```

装 订 线

## 5 阴影处理

本项目根据是否产生软阴影的选项，适当地增加随机采样次数。此外，我们认为光源是一个有大小的物体，我们同样根据是否产生软阴影的选项，决定光源射出的光线是只在一个点发出的（此时不取平均值），还是从多个点发出的（此时要取平均值）。

```
cur = SampleLight(&h, &d, Lights[i],softShadows);
brightness2 = brightness2 + cur;
```

我们根据得到的反射光线结果，通过经验公式算得每个像素的颜色。

```
PixelColor = PixelColor  + (reflect * ob->texture.reflection * 0.75);
```

关于产生阴影的问题，阴影分为了硬阴影和软阴影。硬阴影就是影子全黑，没有渐变的过程，真实感不强，如果光源是一个点，那么对于每个点，它的亮度只能由一条光线决定，那么就一定会产生硬阴影无疑。反之，如果光源是一个面，那么一个点的亮度可能由多条光线决定，此时如果产生阴影，那么在它的边缘会有一个渐变的过程，因为越靠阴影外缘，有机会打到上面的光线会越多。因此，如果是硬阴影模式，我们模拟光源为点光源；如果是软阴影模式，我们模拟光源是一个有大小的物体，并在上面多次采样。

```
if (rnd)
    lightDir = l->RandomSpot() - h->location;
else
    lightDir = l->origin - h->location;
```

```
for (int j = 0; j < addSamples; j++){
    brightness2 = brightness2 + SampleLight(&h, &d, Lights[i],1);
}
```

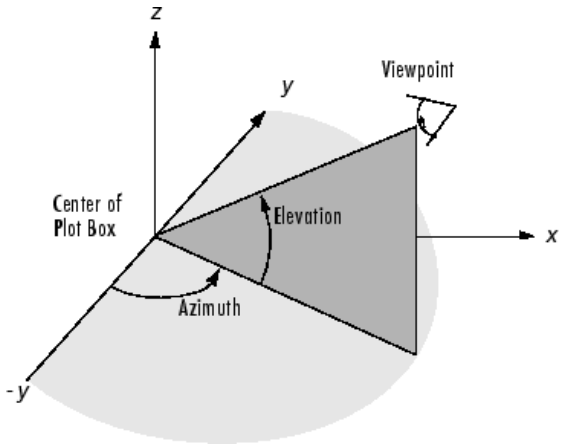
装 订 线

# 6 材质纹理

本项目程序采用了基于坐标映射的纹理处理方式，该纹理映射技术可以通过两种方法来实现，一种是将物体的属性存储在一个二维数组中，该数组一般称为纹理图，然后使用物体的某个位置信息或者其他信息在纹理图中查找属性值，称图像纹理或者也可以使用一个算法来计算物体每个位置上的属性，该实现方法通常称为过程纹理。

## *6.1 映射到平面*

本项目程序中仅有球体表面不是一个平面，故我们需构造一个映射(x,y,z)->(u,v)将其映射到一个二维平面。注意到球体表面任何一点到球心距离为常数，故我们可以用两个角度俯仰角$Elevation \in \left[-\frac{\pi}{2},\frac{\pi}{2}\right]$，方位角$Azimuth \in [0,2\pi)$ 来描述球体表面任何一点，这样一来仅有两个变量，可以映射到一个[0,1]X[0,1]的平面：




如图

对于点(x,y,z)，设球体半径为 r，我们有

x=r*cos(E)*sin(A)

y=-r*cos(E)*cos(A)

z=r*sin(E)

整理得

E=asin(z/r)

A=atan2(-y,x)

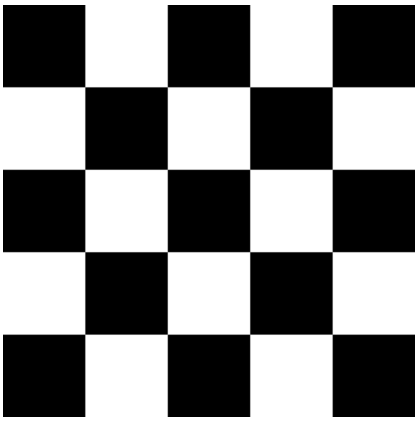
再由 E,A 的值域，我们有：

u=(E+$\pi/2$)/$\pi$

v=A/(2∗$\pi$）

### *6.2 过程纹理*

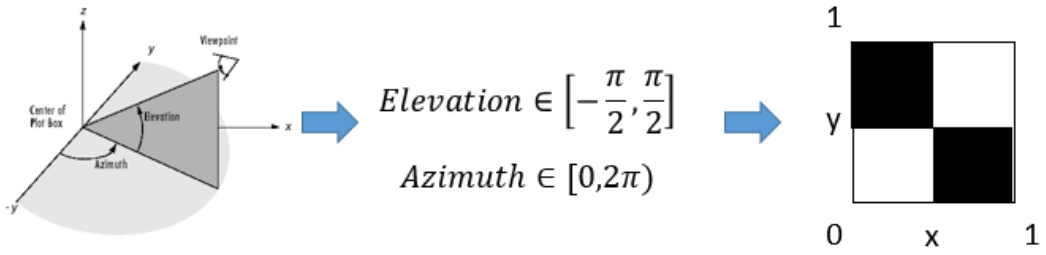
过程纹理为一个映射 F(x,y,z) = color，主要用于以周期性重复纹理为主的简单纹理。
以经典的黑白棋盘图为例：



其典型的特征：相邻块异色可以用 0，1 异或 1 取相异值来模拟。
故有

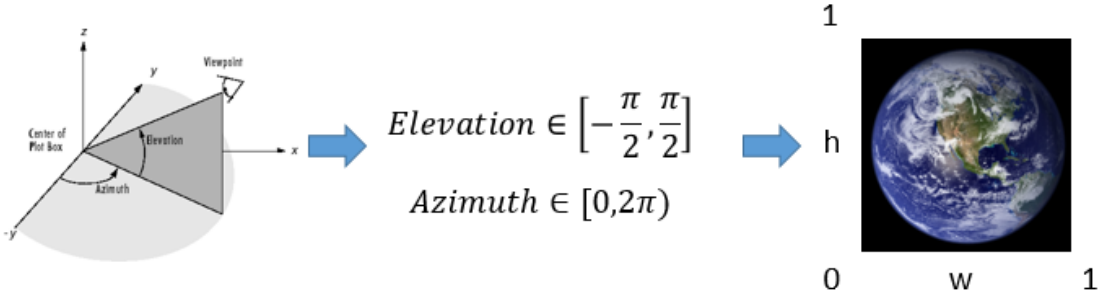
```
if ((int)((x + 0.001) / texture.scale) % 2)
    col ^= 1;
if ((int)((y + 0.001) / texture.scale) % 2)
    col ^= 1;
```

其中+0.001 是为了防止程序因实数运算带来的误差导致黑白边界处颜色混乱。
对于平面物体，我们可以直接投影到(x,y)平面，然后按上述方法处理。
对于球体，可以按 6.1 的方法，映射到[0,1]X[0,1]平面，然后处理：




### *6.3 图像纹理*

对于平面物体，我们可以直接投影到(x,y)平面，缩放后直接获取纹理图中的属性值。
对于球体，可以按 6.1 的方法，映射到[0,1]X[0,1]平面，缩放后获取纹理图中的属性值：



## 7 并行优化

在本项目的程序实现中，每条光线都是独立计算的，为了充分利用现有处理器的多核计算能力，提升渲染速度，可以考虑并行优化。

我们采用了实现简易的 OpenMP 进行优化处理，OpenMp 是由 OpenMP Architecture Review Board 牵头提出的，并已被广泛接受的，用于共享内存并行系统的多处理器程序设计的一套指导性的编译处理方案，在本项目中，用于将串行实现的 for 语句并行化。

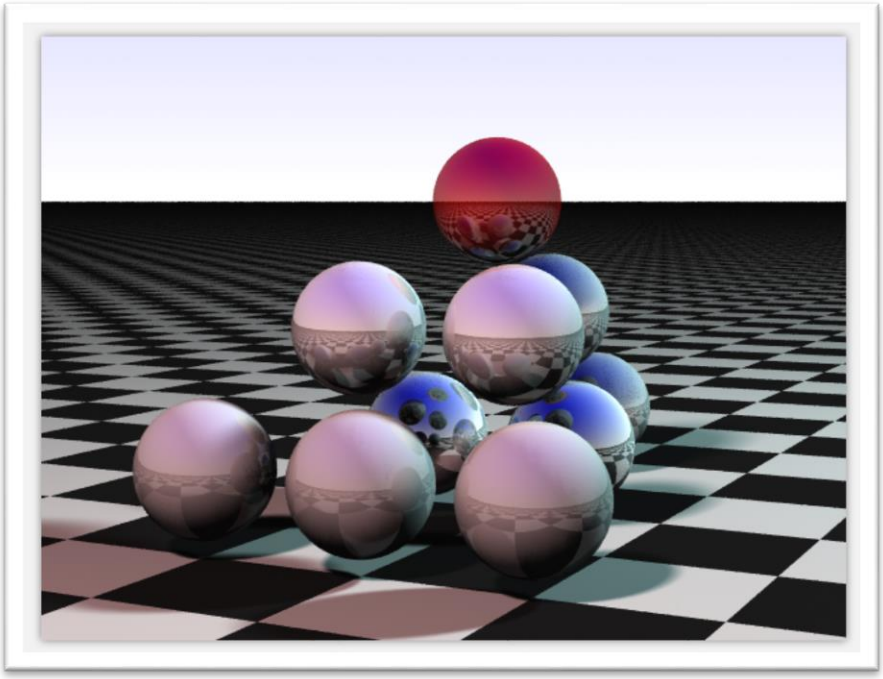
具体的实现如下：

```
#pragma omp parallel for schedule(dynamic, 1)
    for (int y = 0; y < image.height; y++) {
        for (int x = 0; x < image.width; x++) {
            Vec co;
            if (sampleMethod == 0 || superSample == 1.0){
                for (float m1 = 0; m1 < superSample; m1++) {
                    for (float m2 = 0; m2 < superSample; m2++) {
                        float mx = (float)x + m1 / superSample;
                        float my = (float)y + m2 / superSample;
                        getPixelVector(mx,my,&o,&d);
                        co = co + Cast(o, d, recursion);
                    }
                }
            }
    …
```

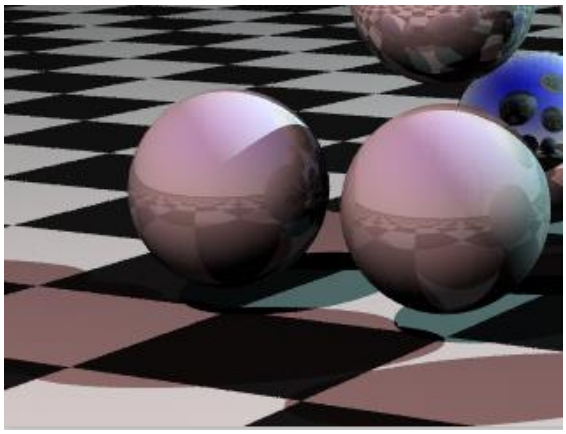
装 订 线

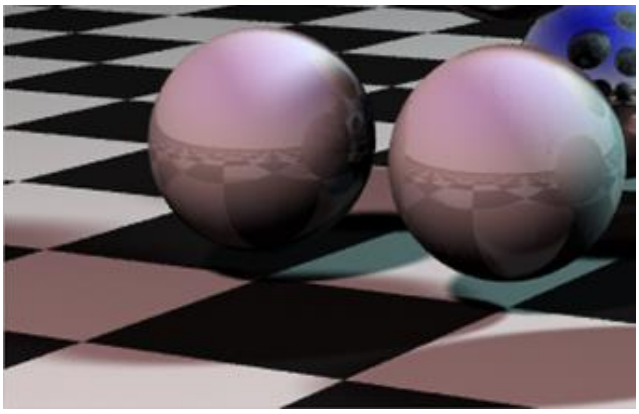
# 8 渲染示例

由于球体比较能反映出渲染效果，下面是本项目的一些以球体为主的场景渲染示例



示例 1.一组球体组合。





示例 2.阴影效果对比

示例 3.纹理效果对比



装 订 线

## 参考文献

[1] Line–sphere intersection [DB/OL].
https://en.wikipedia.org/wiki/Line%E2%80%93sphere_intersection
[2] Möller–Trumbore intersection algorithm [DB/OL].
https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
[3] A Minimal Ray-Tracer: Rendering Simple Shapes [DB/OL].
http://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection
[4] PPM - Netpbm color image format [DB/OL].
http://netpbm.sourceforge.net/doc/ppm.html
[5] Floyd–Steinberg dithering [DB/OL].
https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering
[6] How to get a reflection vector? [DB/OL].
http://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector

装 订 线