

# 基于光线追踪(Ray-Tracing) 原理的物理渲染器实现

余畅 (98) (2019091621002)  
张清安 (95) (2019091621014)  
伍桐雨 (95) (2019091619028)  
肖荣博 (95) (2019091621017)

Software Engineering, UESTC

January 4, 2020

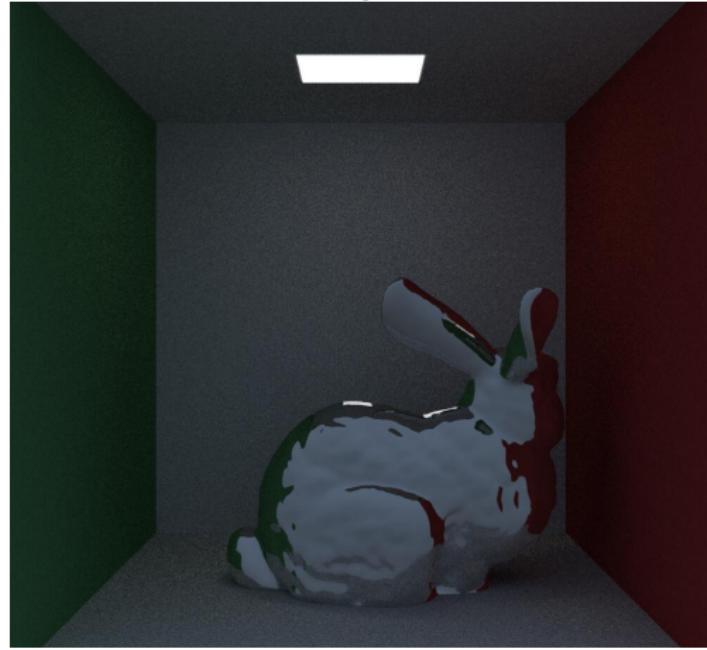
# 需求分析

- ▶ 渲染(Render) 指将三维场景中的模型，按照设定好的环境、灯光、材质及渲染参数，二维投影成数字图像的过程。
- ▶ 光线追踪技术(Ray-Tracing)是由几何光学衍生而来。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。
- ▶ 本项目旨在通过已学的线代和微积分知识，实现一个简易的光追渲染器

## Samples 1

The Cornell Box (data from graphics.cornell.edu)

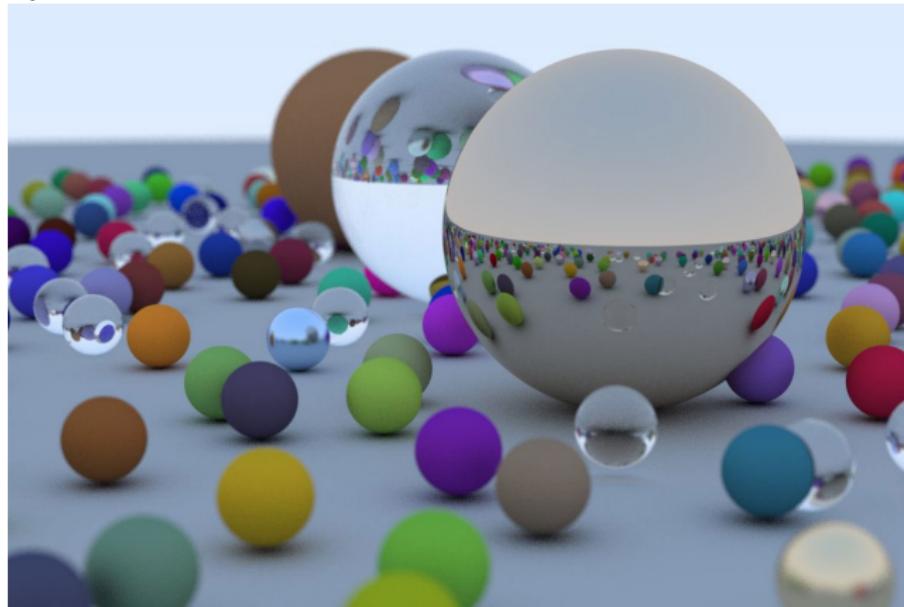
The Stanford Bunny (Source: Stanford University Computer Graphics Laboratory)



(600x600 500 samples per pixel)

## Samples 2

spheres (data from random generator)

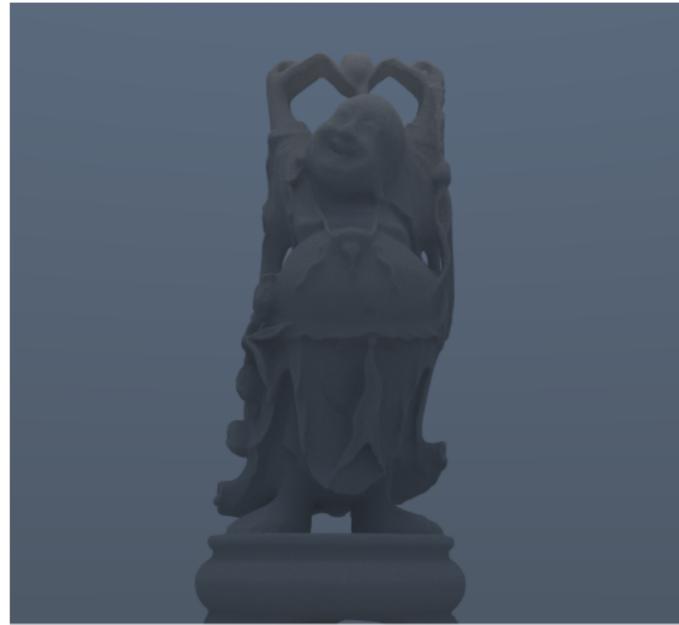


(1200x900 500 samples per pixel)

## Samples 3

Happy Buddha (Source: Stanford University Computer Graphics Laboratory)

Lambertian material



(900x900 1000 samples per pixel)

## Samples 4

The Stanford Bunny (Source: Stanford University Computer Graphics Laboratory)  
Dielectric material



(900x900 300 samples per pixel)

# 模块 1：底层线性代数数学库 (张清安, 肖荣博)

## 主要功能

提供常用线性代数中

数学量(点, 向量, 法线, 矩阵) 的结构体和  
运算方法 (线性运算, 数乘, 内外积,  
矩阵的转置, 求逆, 乘法等) 的函数

- ▶ ./math/vector3.h
- ▶ ./math/matrix3x3.h
- ▶ ./math/transform.h
- ▶ ./math/quaternion.h

## /math/vector3.h

- ▶ 实现了Normal3, Point3, Vector3类
- ▶ 为了方便调用，运算符重载了线性运算

```
BaseVector3 operator + (const BaseVector3 &v) const;  
friend BaseVector3 operator * (const float &s, const BaseVec  
&v);  
...
```

- ▶ 实现了点积，叉积，求长度，求距离，标准化等方法

```
float dot(const BaseVector3 &v) const;  
float length() const;  
float distance(const BaseVector3 &p) const;  
BaseVector3& normalize();  
...
```

## /math/vector3.h

- ▶ SIMD(Single Instruction Multiple Data)
- ▶ 单指令多数据流，能够复制多个操作数，并把它们打包在大型寄存器的一组指令集。
- ▶ 向量每个分量在线性运算时候都是独立的，可以依靠SIMD并行运算
- ▶ 本项目通过intrin.h，实现了X86平台下的SIMD优化
- ▶ ...

```
__m128 vs = _mm_load_ss(&s);
vs = _mm_pshufd_ps(vs, 0x80);
m_val128 = _mm_mul_ps(m_val128, vs);
...
...
```

## /math/matrix3x3.h

- ▶ 实现了Matrix3x3类
- ▶ 用三个vector3表示一个3x3的矩阵
- ▶ 实现了矩阵的常用运算

```
Matrix3x3 transpose() const;  
Matrix3x3 absolute() const;  
Matrix3x3 adjoint() const;  
Matrix3x3 inverse() const;  
...  
...
```

## /math/transform.h

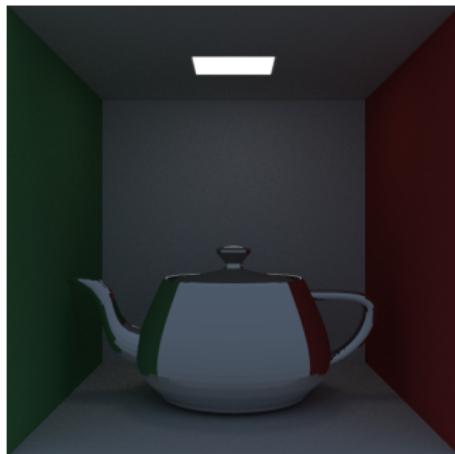
- ▶ 实现了Transform类
- ▶ 封装了一个Matrix3x3和Vector3及其逆过程，表达了空间中的一个变换

[Transform class members:]

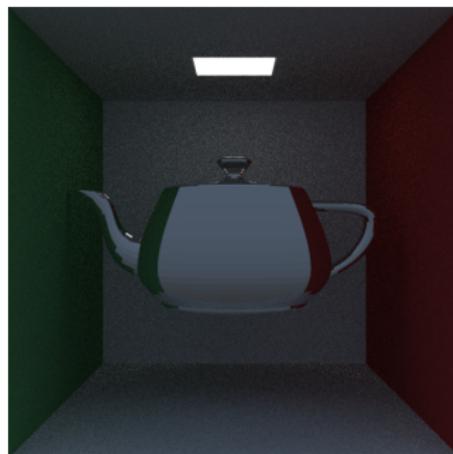
```
Matrix3x3 m_mat, m_inv;  
Vector3 m_trans;
```

## 位移 (Translate)

- ▶  $\text{m\_trans} = (x, y, z)$

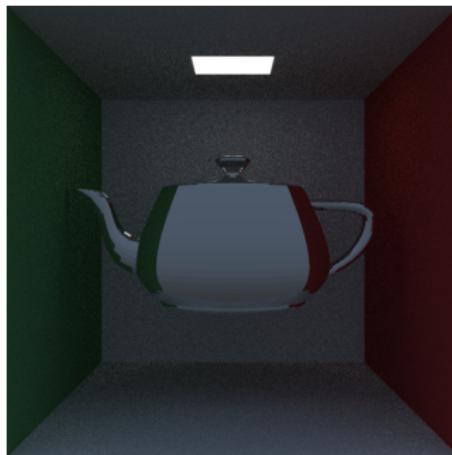


To

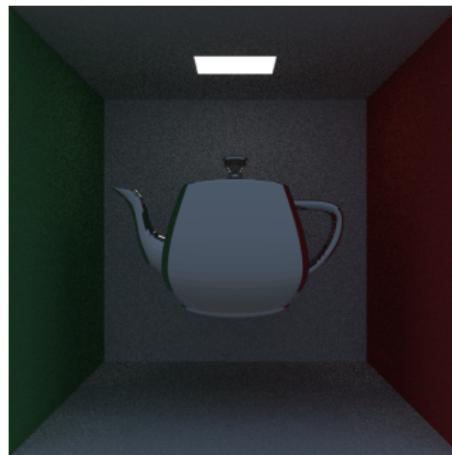


## 缩放 (Scale)

►  $m_{mat} = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix}, m_{mat}^{-1} = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$



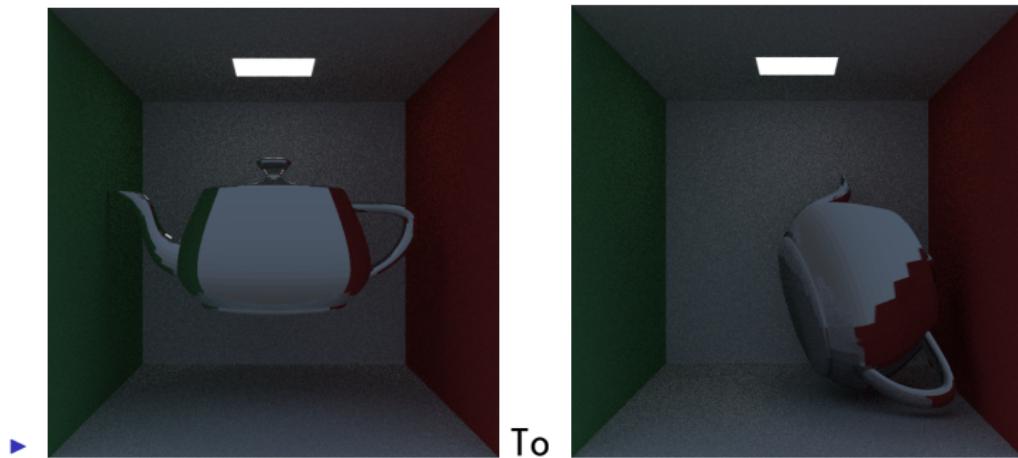
To



## 旋转 (Rotation)

- ▶ 设旋转轴为  $A$ , 旋转角为  $\theta$ ,  $c = \cos\theta, s = \sin\theta$

- ▶  $m_{mat} = \begin{bmatrix} c + (1 - c)A_x^2 & (1 - c)A_xA_y - sA_z & (1 - c)A_xA_z + sA_y \\ (1 - c)A_xA_y & c + (1 - c)A_y^2 & (1 - c)A_xA_y - sA_z \\ (1 - c)A_xA_z - sA_y & (1 - c)A_yA_z + sA_x & c + (1 - c)A_z^2 \end{bmatrix}, m_{mat}^{-1} = m_{mat}^T$



## 模块 2：光追渲染器组件实现（余畅）

### 主要功能

定义场景中的物体，材质，纹理相机等参数，  
并实现光线传播（漫反射，镜面反射，折射，相交）的相应计算

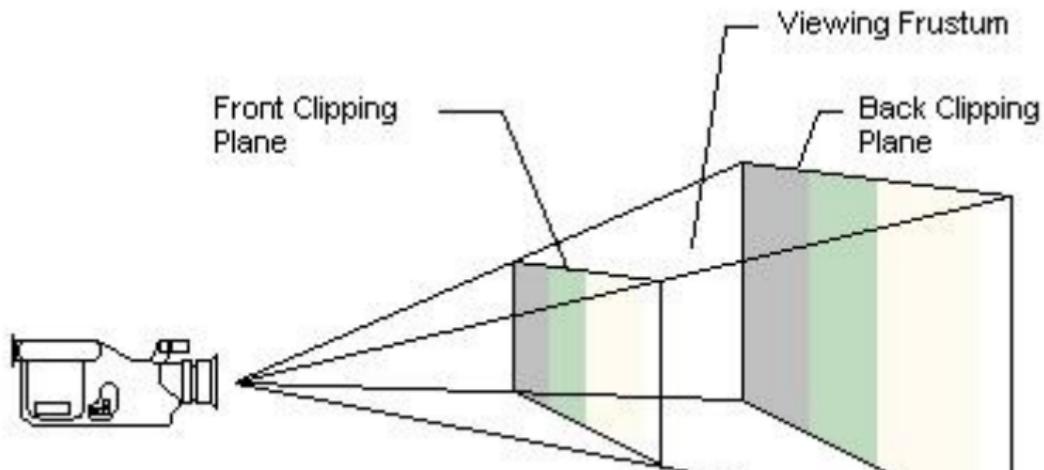
- ▶ ./core/camera.h
- ▶ ./core/integrator.h
- ▶ ./core/material.h
- ▶ ./core/primitive.h, cpp
- ▶ ./core/texture.h
- ▶ ./shapes/rectangle.h, cpp
- ▶ ./shapes/sphere.h, cpp
- ▶ ./shapes/triangle.h, cpp
- ▶ ./accelerators/BVH.h, cpp

## /core/camera.h

- ▶ 实现了 Camera, ProjectiveCamera 类
- ▶ Camera 必须实现的方法

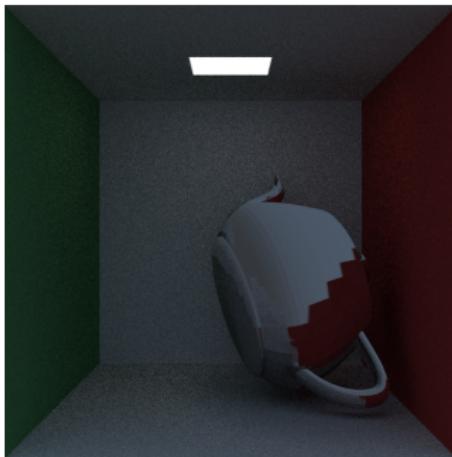
```
virtual Ray getRay(const float &s, const float &t)  
const;
```

- ▶ 投影相机 (ProjectiveCamera类)

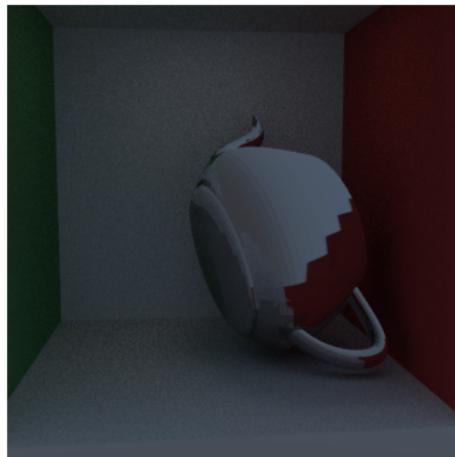


## /core/camera.h

- ▶ const Point3 lookform,  
const Point3 lookat,  
const Vector3 vup
- 确定了摄像机的位置和角度

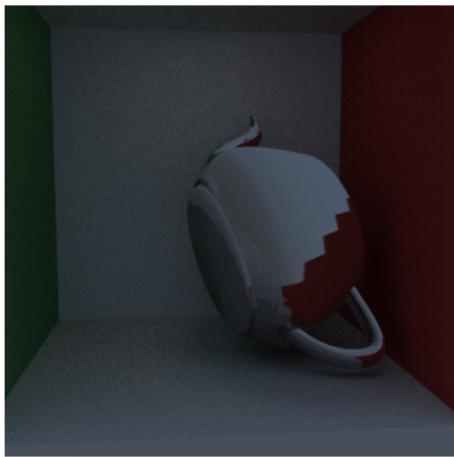


To

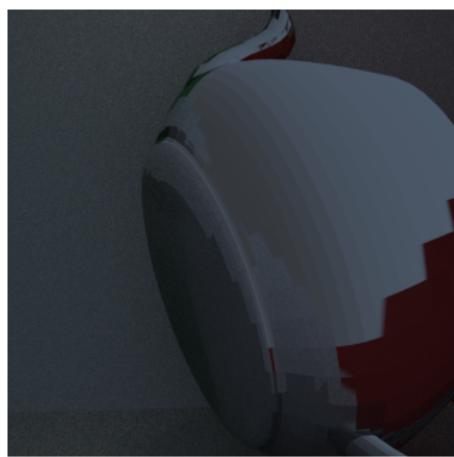


## /core/camera.h

- ▶ float vfov  
确定视角的大小



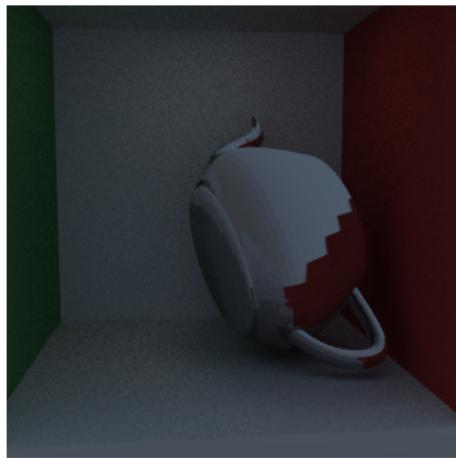
To



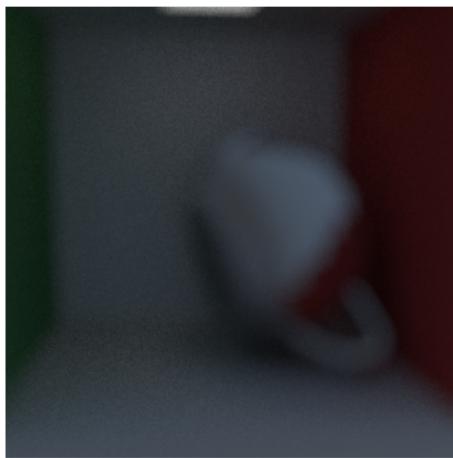
## /core/camera.h

- ▶ float aperture

确定光圈（小孔成像中小孔）的半径



To



## /core/texture.h

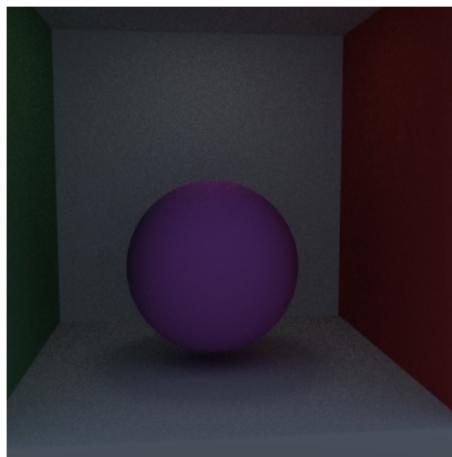
- ▶ 实现了Texture, ConstantTexture, CrossTexture, NoiseTexture类
- ▶ Texture必须实现的方法

```
virtual Spectrum value(float u, float v, const Point3  
&p) const;
```

对通过光线与平面交点及(u,v)坐标对纹理采样颜色

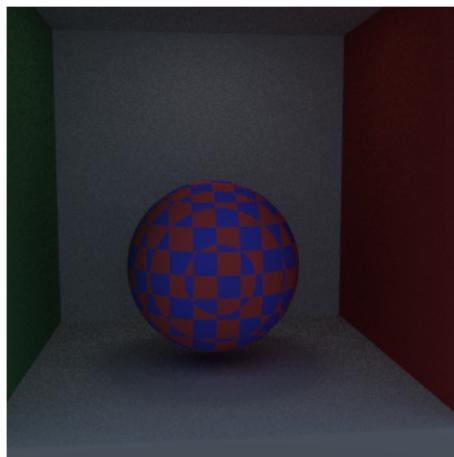
## /core/texture.h

- ▶ ConstantTexture, 返回一个常量颜色的纹理



## /core/texture.h

- ▶ CrossTexture, 返回一个交错的棋盘纹理



## /core/texture.h

- `NoiseTexture`, 返回一个柏林噪音(Perlin Noise)实现的随机纹理

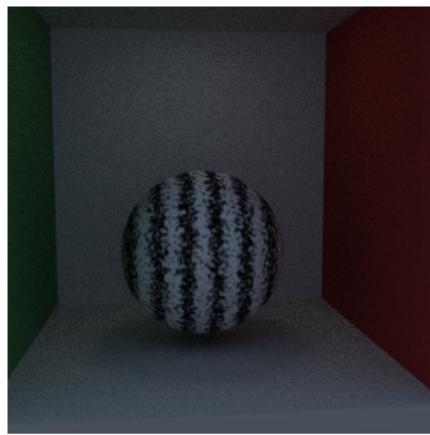


Figure: scale = 0.5

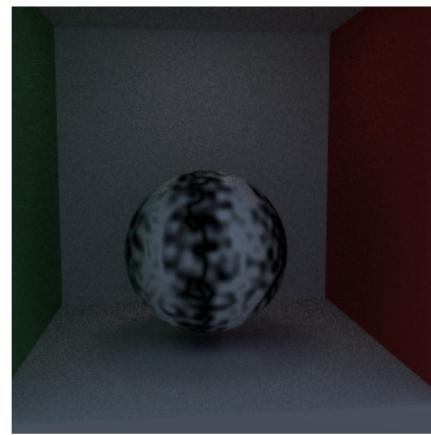


Figure: scale = 0.05

## /core/material.h

- ▶ 实现了**Material**, **MentalMaterial**, **LambertianMaterial**, **DielectricMaterial**类
- ▶ **Material**必须实现的方法

```
virtual bool scatter  
(const Ray &r_in, const SurfaceInteraction &si, Spectrum  
&attenuation, Ray &scattered) const;
```

根据入射光线和表面的信息，返回出射光线和衰减率

- ▶ 

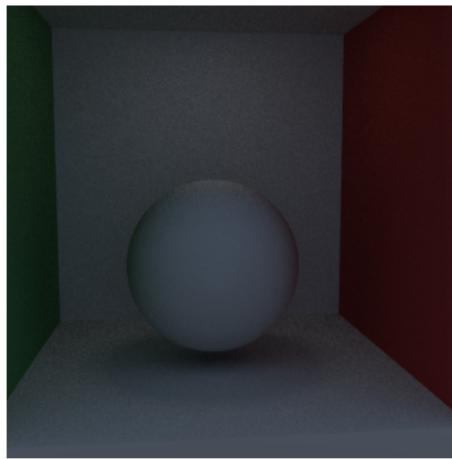
```
virtual Spectrum emitted  
(float u, float v, const Vector3 &p) const;
```

返回物体的自发光

## /core/material.h

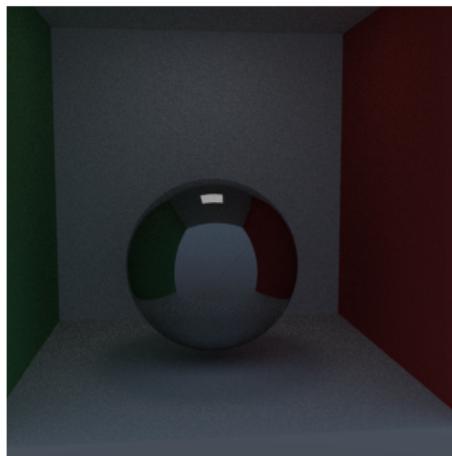
- ▶ **LambertianMaterial**, 漫反射材质
- ▶ 光线射入物体后向各方向随机漫反射

```
Vector3 target = si.p + si.n + rng.randomInUnitSphere();
scattered = Ray(si.p, target - si.p, r_in.m_time);
```



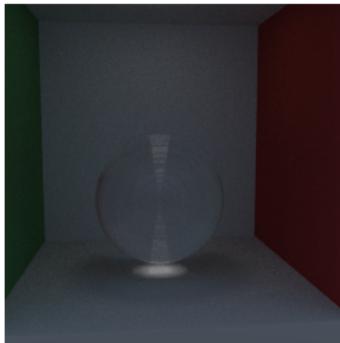
## /core/material.h

- ▶ **MetalMaterial**, 金属材质
- ▶ 入射角和反射角相同, 即按法线反转
- ▶  $\text{out} = \text{in} - (\text{in}, \mathbf{n}) \cdot 2 \cdot \mathbf{n}$
- ▶ **fuzz**参数描述镜面的粗糙程度



## /core/material.h

- ▶ DielectricMaterial, 电介质材质
- ▶ 同时存在镜面反射和折射效应，能量分配通过菲涅耳公式(Fresnel Formula)计算，实际项目中采用schlick公式近似
- ▶  $\frac{\sin\theta_1}{\sin\theta_2} = r$
- ▶ m\_refractive\_idx参数描述介质材料的折射率



## /core/shape.h

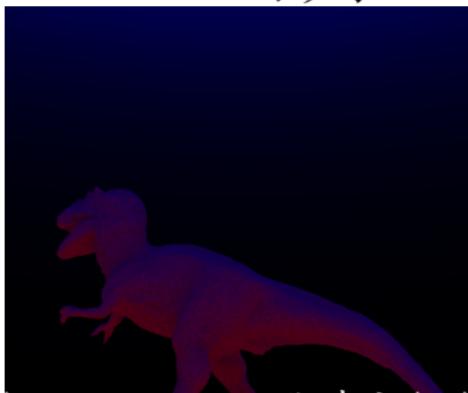
- ▶ 实现了Shape基类
- ▶ Shape必须实现的方法
- ▶ `virtual BBox objectBound() const = 0;`  
返回形状在物体参考系中的包围盒
- ▶ `virtual BBox worldBound() const = 0;`  
返回形状在世界参考系中的包围盒
- ▶ `virtual bool canIntersect() const;`  
判断当前图元是否能相交
- ▶ `virtual void refine(std::vector<SharedPtr<Shape>>&refined) const;`  
细分无法相交的图元
- ▶ `virtual bool intersect(const Ray &ray, float *hit_t, Surface *dg) const;`  
对相应光线执行相交计算

## /core/shape.h

- ▶ ./shapes/triangle.h
- ▶ ./shapes/triangle.cpp
- ▶

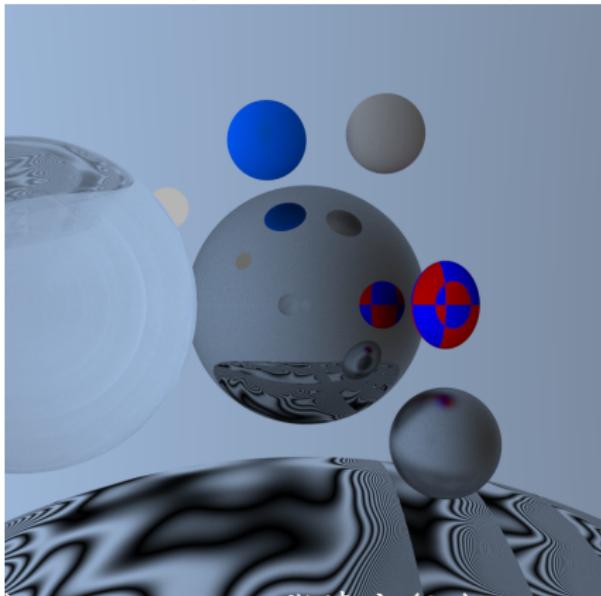
为保证内存的连续性，三角形的顶点，索引等数据统一储存在 TriangleMesh类中，具体计算由细分后的Triangle类实现

- ▶ 相交方法实现以Fast, Minimum Storage Ray-Triangle Intersection为参考



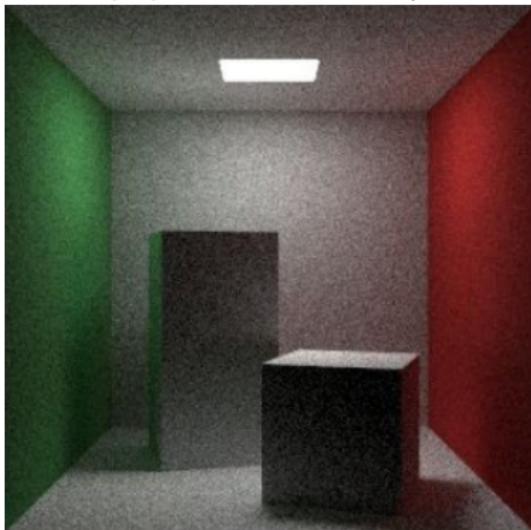
## /core/shape.h

- ▶ ./shapes/sphere.h
- ▶ ./shapes/sphere.cpp
- ▶ 计算相交即解一个关于射线参数t的一元二次方程



## /core/shape.h

- ▶ ./shapes/rectangle.h
- ▶ ./shapes/rectangle.cpp
- ▶ 将射线转化到物体坐标系后六个平面和坐标轴分别对齐，  
分别计算相交后取t的最小值的平面



## /accelerator/BVH.h

- ▶ 如果每一束光线都对场景中的每一个物体进行相交计算，计算开销将会无法接受
- ▶ 本项目对每一个形状构建了包围盒，并使用了数据结构BVH(Bounding Volume Hierarchy)进行加速
- ▶ 本质是一类二叉搜索树，每层对三条坐标轴中的一条进行排序
- ▶ 复杂度可由 $O(n^2)$ 降为 $O(n \log n)$

## /core/parser.h (梧桐雨)

- ▶ 实现了parser类，用以将描述场景的JSON文件和描述模型的OBJ文件解析入Scene结构体的内存中

## /core/parser.h

- ▶ **JSON (JavaScript Object Notation)**  
是一种轻量级的数据交换格式。易于人阅读和编写
- ▶ **JSON**是一个标记符的序列。这套标记符包含六个构造字符、字符串、数字和三个字面名
- ▶ 本项目使用的描述文件是**JSON**的子集，  
即**parser**类仅按照特定语法树和关键词进行解析

## /core/parser.h

### ► 场景文件样例 (part 1.)

```
1  {
2      "config": {
3          "screen_x": 1000,
4          "screen_y": 1000,
5          "sample_times": 500
6      },
7      "camera": {
8          "lookfrom": {
9              "x": 278,
10             "y": 278,
11             "z": -760
12         },
13        "lookat": {
14            "x": 278,
15            "y": 278,
16            "z": 0
17        },
18        "lookup": {
19            "x": 0,
20            "y": 1,
21            "z": 0
22        },
23        "fov": 40.0,
24        "aspect": 1,
25        "aperture": 0.0,
26        "focus_dist": 100000,
27        "t0": 0,
28        "t1": 0
29    },
30    "transforms": [
31        [
32            "matrix": [
33                [
34                    "x": 1.0, "y": 0.0, "z": 0.0, "w": 0.0,
35                    "x": 0.0, "y": 1.0, "z": 0.0, "w": 0.0,
36                    "x": 0.0, "y": 0.0, "z": 1.0, "w": 0.0,
37                    "x": 0.0, "y": 0.0, "z": 0.0, "w": 1.0
38                ],
39                [
40                    "x": 0.0, "y": 0.0, "z": 0.0, "w": 0.0,
41                    "x": 1.0, "y": 0.0, "z": 0.0, "w": 0.0,
42                    "x": 0.0, "y": 1.0, "z": 0.0, "w": 0.0,
43                    "x": 0.0, "y": 0.0, "z": 1.0, "w": 0.0
44                ],
45                [
46                    "x": 0.0, "y": 0.0, "z": 0.0, "w": 0.0,
47                    "x": 0.0, "y": 1.0, "z": 0.0, "w": 0.0,
48                    "x": 0.0, "y": 0.0, "z": 1.0, "w": 0.0,
49                    "x": 1.0, "y": 0.0, "z": 0.0, "w": 0.0
50                ],
51                [
52                    "x": 0.0, "y": 0.0, "z": 0.0, "w": 0.0,
53                    "x": 0.0, "y": 0.0, "z": 1.0, "w": 0.0,
54                    "x": 0.0, "y": 1.0, "z": 0.0, "w": 0.0,
55                    "x": 0.0, "y": 0.0, "z": 1.0, "w": 0.0
56                ]
57            ]
58        ]
59    ]
60 }
```

## /core/parser.h

### ► 场景文件样例 (part 2.)

```
36     },
37     {
38         "translate" : { "x" : 555,
39                         "y" : 277.5,
40                         "z" : 277.5}
41     },
42     {
43         "translate" : { "x" : 277.5,
44                         "y" : 277.5,
45                         "z" : 555}
46     },
47     {
48         "translate" : { "x" : 277.5,
49                         "y" : 555,
50                         "z" : 277.5}
51     },
52     {
53         "translate" : { "x" : 277.5,
54                         "y" : 0,
55                         "z" : 277.5}
56     },
57     {
58         "translate" : { "x" : 278,
59                         "y" : 553.5,
60                         "z" : 279.5}
61     }
62 ],
63 "shapes" :
64 [
65     { "rectangle" : {
66         "transform" : 0,
```

## /core/parser.h

### ► 场景文件样例 (part 3.)

```
102     "materials" :  
103     [  
104         {"mental": {  
105             "spectrum(256)": {"r": 191, "g": 173, "b": 111},  
106             "fuzz": 0.0  
107         }},  
108         {"lambertian": {  
109             "texture": {  
110                 "constant": {  
111                     "spectrum(1)": {"r": 0.73, "g": 0.73, "b": 0.73}  
112                 }},  
113             "lambertian": {  
114                 "texture": {  
115                     "constant": {  
116                         "spectrum(1)": {"r": 0.65, "g": 0.05, "b": 0.05}  
117                     }},  
118             }},  
119             {"lambertian": {  
120                 "texture": {  
121                     "constant": {  
122                         "spectrum(1)": {"r": 0.12, "g": 0.45, "b": 0.15}  
123                     }},  
124             }},  
125             {"diffuse": {  
126                 "texture": {  
127                     "constant": {  
128                         "spectrum(1)": {"r": 1, "g": 1, "b": 1}  
129                     }},  
130             }},  
131         }]  
132     ],
```

## /core/parser.h

- ▶ config对象，定义了场景输出图片的长宽和采样数
- camera对象，定义了场景中的相机信息
- transforms数组，定义了场景中所用到的转移矩阵
- shapes数组，定义了场景中的图形信息
- materials数组，定义了场景中的材质信息
- primitive数组，定义了场景中的图元信息

## /core/parser.h

- ▶ `void load(const char *file);`  
从文件中导入JSON字符串（忽略无意义的空白符(ws)）
- ▶ `void run(Scene *scene);`  
开始解析入场景结构体中

## /core/parser.h

- ▶ READ\_BRACE\_BEGIN();  
READ\_BRACE\_END();  
READ\_COLON();  
READ\_ARRAY\_BEGIN();  
READ\_BRACE\_ELEMENT\_END();  
READ\_ARRAY\_ELEMENT\_END();  
读入构造字符串构造语法树并判断合法性，  
通过assert()函数检查合法性

## /core/parser.h

- ▶ std::string READ\_STRING();
- bool READ\_BOOL();
- int READ\_INT();
- float READ\_FLOAT();
- BaseVector3 READ\_VECTOR();
- Spectrum READ\_SPECTRUM1();
- Spectrum READ\_SPECTRUM256();

读入基本的数据类型（JSON自己基础的字符串，浮点数，整数，布尔值和扩展类型，包括向量和颜色）

## /core/parser.h

- ▶ READ\_SHAPES();  
READ\_SHAPE();  
READ\_RECTANGLE();  
READ\_SPHERE();
- ▶ 进入读取shape的分支后，先用READ\_SHAPES()对数组循环读取
- ▶ 在循环中用READ\_SHAPE()解析其中的每个元素
- ▶ 在解析到其中的type值对后，细化为具体  
READ\_RECTANGLE() 或 READ\_RECTANGLE() 或 READ\_SPHERE() 的方法，  
进而通过读取值对读取其中的元素值
- ▶ 其余分支读取方法相同，在此不再赘述

## /core/parser.h

- ▶ obj文件是3D模型文件格式。适合用于3D软件模型之间的互导
- ▶ 本项目实现了对其的读入支持

## Others (余畅)

- ▶ `./core/config.h` 控制项目的编译选项开关
- ▶ `./core/rng.h` 实现了柏林噪声 (Perlin Noise) 生成器, 梅森旋转 (Mersenne Twister) 算法实现的随机数生成器
- ▶ `./core/memory.h`  
实现了自动释放和拷贝内存的引用计数指针
- ▶ `./math/mathutility.h` 使用 SIMD 重写了部分常用数学函数
- ▶ `./math/bbox.h` 定义了包围盒的结构体
- ▶ `./core/primitive.h, cpp`  
将 `shape` 和 `material` 封装整合为图元
- ▶ `./core/interaction.h` 定义了描述相交表面细节的结构体
- ▶ `./core/scene.h` 定义了描述场景内容的结构体
- ▶ `./core/ray.h` 定义了射线的结构体
- ▶ `./core/integrator.h` 定义了对场景采样的积分器
- ▶ `./core/spectrum.h` 定义了颜色 (简化后辐射度) 的结构体

; )

Thank you for watching!