



3.12截止  
不算分

## 图形绘制技术

Introduction and Prerequisite

对本课程使用的实验框架 Moer-lite 的简单介绍

2023 春季学期

Tab-O

# Abstract

该文档是南京大学计算机系 2023 春季学期《图形绘制技术》课程的实验手册。

Lab0 的内容是要对本课程实验框架 Moer-lite 的简单介绍，Moer-lite 是一个使用 C++ 编写的基于物理的离线蒙特卡洛光线追踪渲染框架，包括相机模型、几何体、加速结构、材质、纹理等基础功能。我们将在该框架的基础上，介绍现代计算机图形学渲染的理论及实现。

这也是该课程第一次使用该框架以及实验安排作为教学工具，因此难免出现许多纰漏与不足。各位同学如果在学习过程中有任何疑问或想法，包括但不限于

- 实验框架的出现的 Bug
- 手册中叙述不完善的部分
- 对实验安排的建议
- 更好的框架设计

主讲老师：过洁 Email : guojie@nju.edu.cn

欢迎联系助教，帮助我们完善课程的实验部分。TAs :

- 陈振宇 QQ : 895761580 Email : chenzy@smail.nju.edu.cn
- 袁军平 QQ : 1924188282 Email : 191250189@smail.nju.edu.cn
- 周辰熙 QQ : 1305845549 Email : 191250210@smail.nju.edu.cn
- 王宸 QQ : 1401520906 Email : chenwang@smail.nju.edu.cn

同时，课程的框架代码以及实验文档也会不断更新改进，请同学们关注助教发布的相关信息。

# 1. Download and Compiling

项目在 github 上的地址 <https://github.com/NJUCG/Moer-lite>

## 1.1 Prerequisite

为了确保项目能够正常下载并成功编译，请确保你的开发环境至少包含如下内容

- git
- CMake 3.15 及以上
- C++ 编译器（支持 C++17）

## 1.2 Download

为了避免中文路径可能导致的问题，建议在全英文路径下下载及编译项目。下载成功后，你在本地的项目结构应当如下

```
Moer-lite
  \externals
  \libs
  \src
  CMakeLists.txt
  README.md
```

### 使用 Git

```
git clone --recursive https://github.com/NJUCG/Moer-lite.git
```

注意，一定要用`--recursive` 将项目的子模块也 clone 到本地，否则将无法成功编译项目。

### 直接下载

如果由于网络问题从 github 上拉取较慢，可以考虑下载我们提供的压缩包

## 1.3 Compiling

### Linux/MacOS

如果你使用的是 Linux 或 MacOS 系统，建议使用以下方法进行编译

```
# 进入项目目录
mkdir build
cd build
cmake ..
make -j4
```

## Windows

假设我们将项目 clone 到 C:/2023spr/Moer-lite

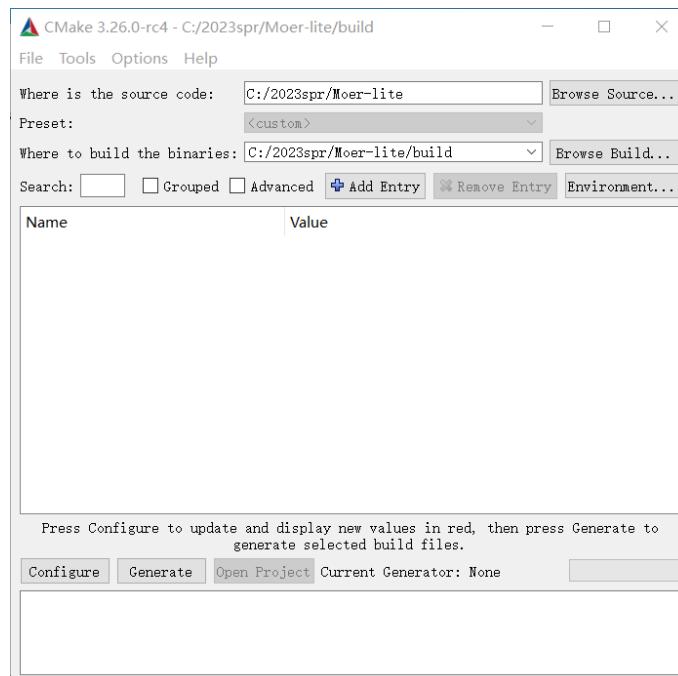


图 1: 设定项目目录以及构建目录, 然后点击 configure

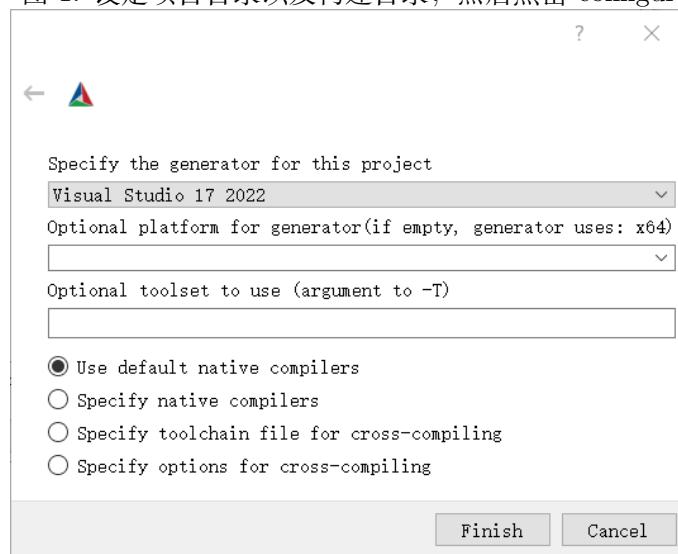


图 2: 选择编译器, 此处为 VS2022

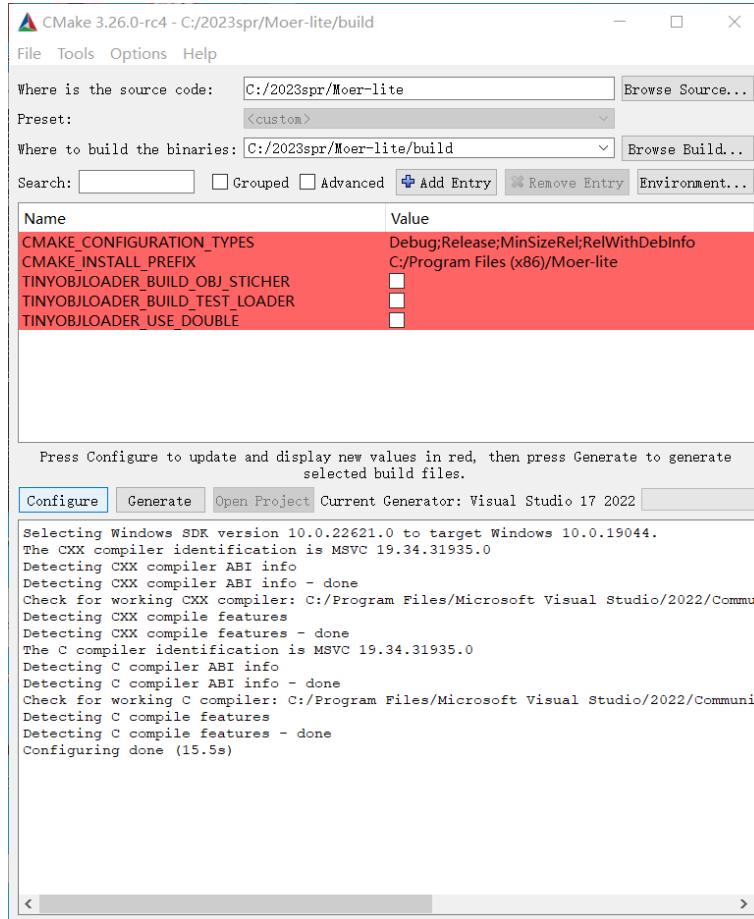


图 3: 配置完成后点击构建 Generate

📁 CMakeFiles	2023/2/26 9:25
📁 externals	2023/2/26 9:25
📄 ALL_BUILD.vcxproj	2023/2/26 9:25
📄 ALL_BUILD.vcxproj.filters	2023/2/26 9:25
📄 cmake_install	2023/2/26 9:25
📄 CMakeCache	2023/2/26 9:25
📄 INSTALL.vcxproj	2023/2/26 9:25
📄 INSTALL.vcxproj.filters	2023/2/26 9:25
📄 Moer.vcxproj	2023/2/26 9:25
📄 Moer.vcxproj.filters	2023/2/26 9:25
📌 Moer-lite.sln	2023/2/26 9:25
📄 ZERO_CHECK.vcxproj	2023/2/26 9:25
📄 ZERO_CHECK.vcxproj.filters	2023/2/26 9:25

图 4: 在 build 文件夹中使用 vs 打开 Moer-lite.sln, 完成 vs 项目构建

## 1.4 Test

完成编译后，你的项目根目录下应当会出现一个 target 文件夹，其结构如下

```
\target
```

```
 \bin
```

```
   Moer
```

```
 \lib
```

第三方依赖编译生成的二进制库

执行 Moer 并传递场景描述文件参数

```
./Moer 场景描述文件的目录
```

等待程序执行完成后，你应该能在 bin 目录下得到一张图片

## 2. Introduction

蒙特卡洛光线追踪是目前最成熟的离线渲染框架，几乎所有的实时感渲染系统都基于光线追踪算法。这些渲染系统的规模有小有大，从只有 99 行 c++ 代码的 [smallpt](#)，到大致千行左右的 [Ray Tracing in One Weekend Series](#)，再到学术界、工业界使用的大规模渲染器，无论其在功能、性能上有多大差异，所有的 ray tracer 基本都由以下几个部分组成 [1]

- Cameras 相机模型
- Ray-object intersections 光线-物体求交
- Light sources 光源模型
- Surface scattering 表面散射
- Light transport simulation 光线传输模拟

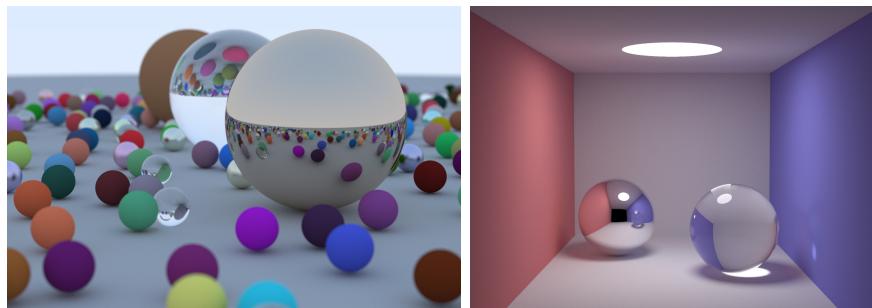


图 5: Ray Tracing in One Weekend (左) 和 smallpt (右)

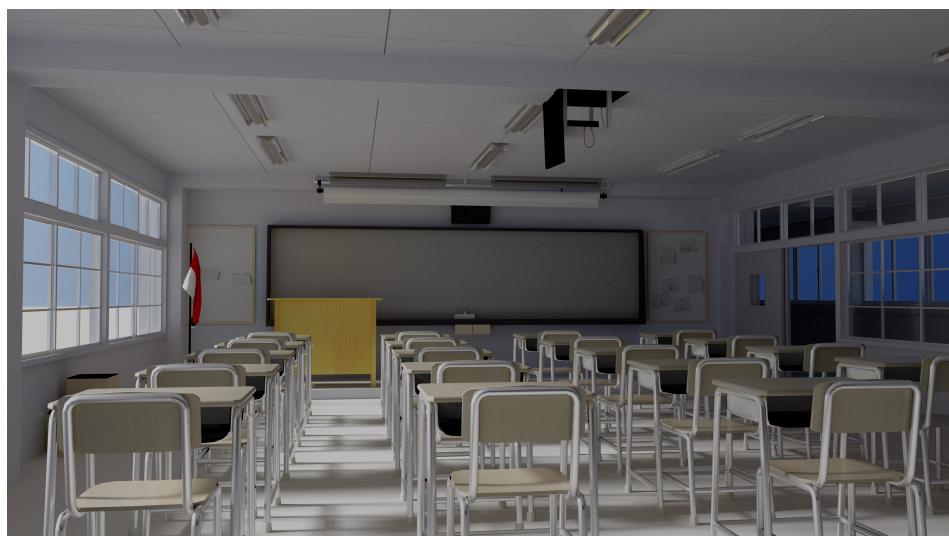


图 6: Moer

## 2.1 Moer-lite System

Moer-lite 的主要渲染功能部分如下

基类	目录	作用
Acceleration	FunctionLayer/Acceleration	加速光线与物体的求交
Camera	FunctionLayer/Camera	模拟相机系统
Film	FunctionLayer/Film	做屏幕空间降噪等操作
Integrator	FunctionLayer/Integrator	求解渲染方程
Light	FunctionLayer/Light	光源模型
Material	FunctionLayer/Material	物体表面材质模型
Sampler	FunctionLayer/Sampler	随机数采样器
Shape	FunctionLayer/Shape	几何体模型
Texture	FunctionLayer/Texture	纹理

Moer-lite 目前的第三方依赖

- [embree3](#) Intel 推出的光线求交引擎
- [FastMath](#) 一个高性能数学库
- [json](#) 用于 Json 的解析
- [stb](#) PNG/JPG/HDR 等格式图片的 IO
- [tinyobjloader](#) 解析 OBJ 模型文件

## 2.2 运行与场景描述

目前 Moer-lite 只有命令行交互模式，GUI 交互界面的开发在计划中，欢迎有兴趣的同学加入我们。一个完整的场景描述建议使用如下结构

```
folder #场景名称
  \scene.json    #场景描述文件
  \models        #存放obj文件
  \images        #存放图片
```

使用 Moer-lite 渲染该场景

./Moer folder完整路径

## 场景格式

```
{  
    "output" : {  
        "filename" : "result.png"      # 输出图片的名称  
    },  
    "sampler" : {  
        "type" : "...",             # 使用的sampler类型  
        ...                          # sampler的其他参数  
    },  
    "camera" : {  
        "type" : "...",             # 相机的类型  
        ...                          # camera的其他参数  
    },  
    "integrator" : {  
        "type" : "...",             # 积分器的类型  
        ...                          # 积分器的其他参数  
    },  
    "scene" : {  
        "acceleration" : "...",     # 加速结构的类型  
        "shapes" : [  
            ...  
        ],                            # 场景中的几何体  
        "lights" : [  
            ...  
        ]                             # 场景中的光源  
    }  
}
```

### 限制

目前支持的图片格式有 PNG/JPG/HDR

目前支持的模型格式只有 obj，并且每个 obj 文件中只能有一个 mesh

### 3. Examples

我们提供了几个简单的示例场景

Two-SpotLights

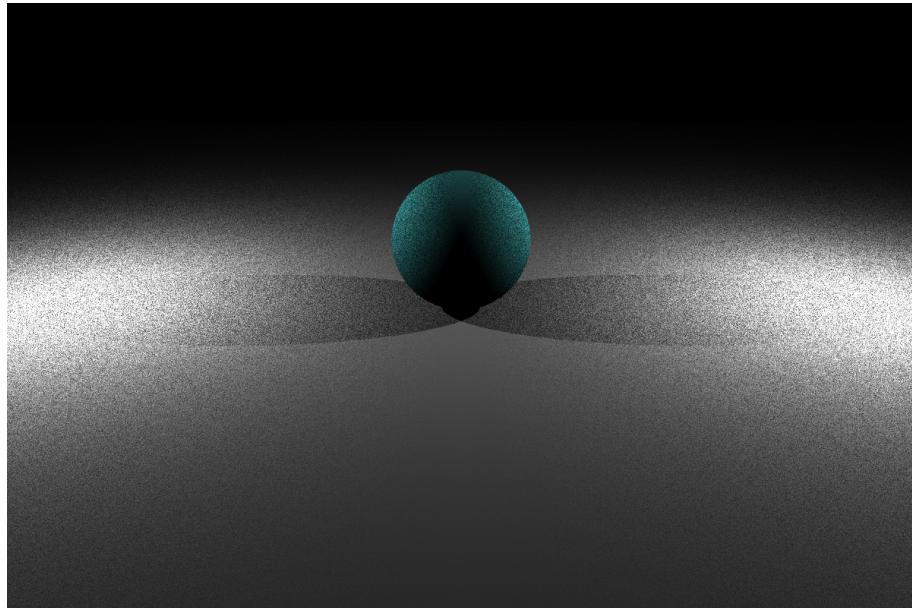


图 7: 场景中左右两侧都有点光源

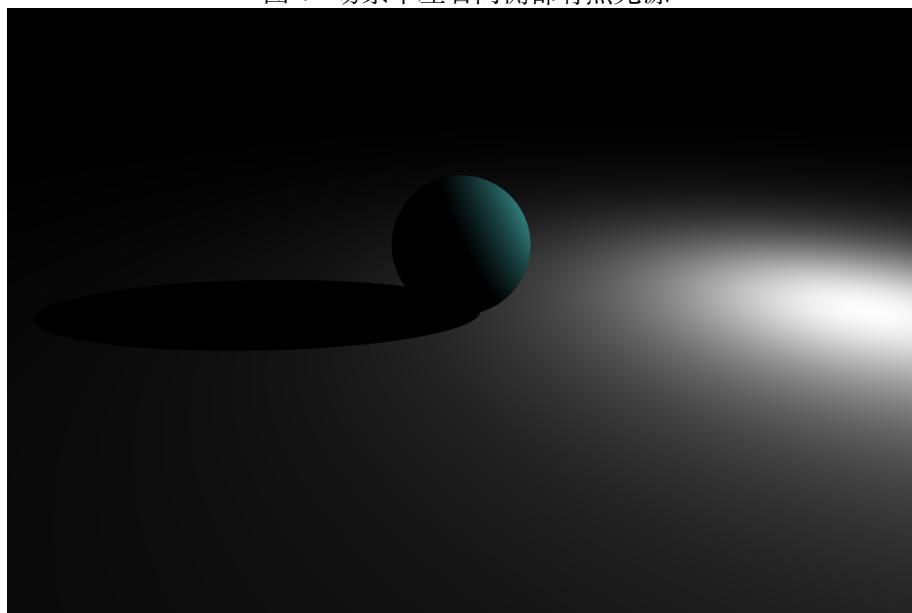


图 8: 场景中只有右侧有点光源

同学们可以思考下为什么图 7 的中间部分噪点较少，而两侧噪点非常明显。

### AreaLight

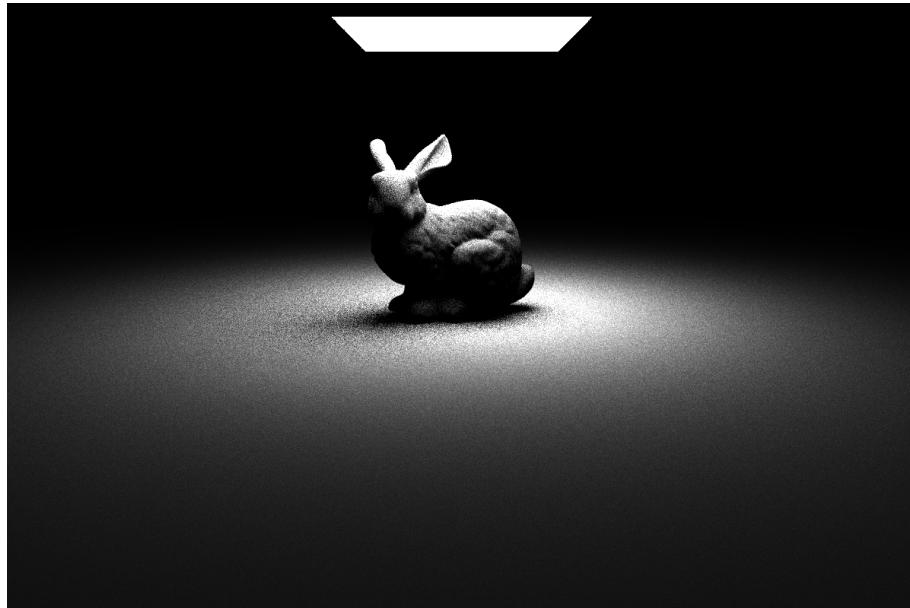


图 9: 场景中的兔子模型由一个面光源照亮

## 4. Camera 相机

Camera 的配置描述应当至少包括一下部分

```
"camera" : {
    "type" : "...",           # 相机的类型
    "tNear" : 0.1,            # 相机产生光线允许发生相交的距离最小值
    "tFar" : 10000,           # 相机产生光线允许发生相交的距离最大值
    "timeStart" : 0,          # 相机快门开始时间
    "timeEnd" : 0,            # 相机快门结束时间
    "film" : {
        "size" : [1200, 800]   # 输出图片的分辨率
    }
}
```

对于光追渲染系统，相机的主要任务是对图片上每一个像素采样一条光线，用于计算该像素的颜色。任何一个相机模型都应当实现以下两个对外接口。

```
1 virtual Ray sampleRay(const CameraSample &sample, Vector2f NDC) const = 0;
2
3 virtual Ray sampleRayDifferentials(const CameraSample &sample, Vector2f NDC) const = 0;
```

NDC 全称为 **Normalized device Coordinate**, 需要注意的是, 此处 NDC 的定义与常见的 NDC 定义稍有不同。在 OpenGL 以及其它图形 api 中, NDC 通常定义为一个三维坐标。此处我们的 NDC 描述的是一个像素在图像上的相对位置。

## CameraSample

CameraSample 是一个包含 5 个 float 的结构体, 其定义如下

```
1 struct CameraSample {
2     Vector2f xy;
3     Vector2f lens;
4     float time;
5 };
```

由于相机产生光线是一个采样过程, 因此我们需要一些随机数作为输入, 得到一个结果 (**此处是光线**)。对于相机的采样过程, 所需要的随机数分为三组。其作用如下

- xy 在一块像素内采样一个位置
- lens 在透镜上采样一个位置
- time 在快门打开的时间段内采样一个时间点

## sampleRay

```
1 virtual Ray sampleRay(const CameraSample &sample, Vector2f NDC) const = 0;
```

该函数接受一个 CameraSample 以及像素在图像上的相对位置, 为该像素采样出一条光线

## sampleRayDifferentials

```
1 virtual Ray sampleRayDifferentials(const CameraSample &sample, Vector2f NDC) const = 0;
```

该函数接受一个 CameraSample 以及像素在图像上的相对位置, 为该像素采样出一条带微分的光线, 微分的作用将在 Texture 纹理部分详细介绍。

### 4.1 PinholeCamera 针孔相机

**针孔相机 PinholeCamera** 是最简单的相机模型, 也是目前框架中唯一提供的相机模型, 在 scene.json 中配置一个针孔相机的描述如下

```
"camera" : {
    "type" : "pinhole",
    "transform" : {
        "position" : [0, 3, 15],
```

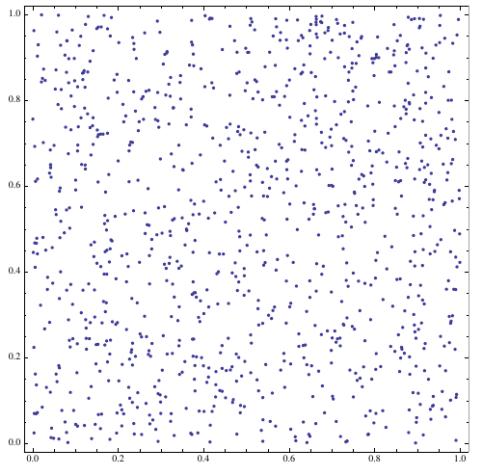
```
"up" : [0, 1, 0],  
"lookAt" : [0, 0, 0]  
, # 透视相机的lookAt描述  
"tNear" : 0.1,  
"tFar" : 10000,  
"verticalFov" : 45, # 相机的垂直视角, 单位角度  
"timeStart" : 0,  
"timeEnd" : 0,  
"film" : {  
    "size" : [1200, 800]  
}  
}
```

transform 中的 position 确定了相机的位置, lookAt 则通过一个点确定了相机指向的方向, up 大致确定了相机向上的方向 (因为当 up 方向与相机指向的方向不垂直时, up 会重新进行计算)。verticalFov 指定了相机在竖直方向上视角的大小。

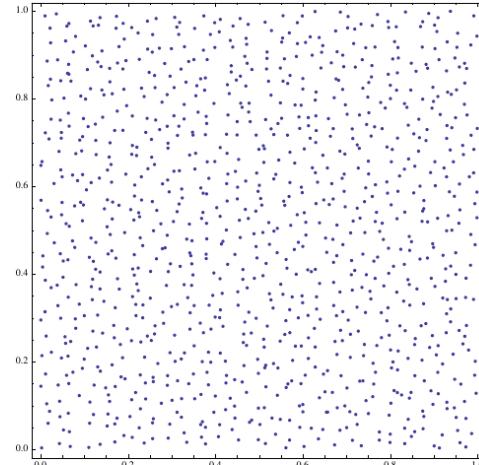
TODO:lookAt 矩阵计算的介绍以及从局部坐标系产生光线变换到世界坐标系

## 5. Sampler 采样器

采样器所完成的工作非常简单也非常底层，sampler 的作用就是提供采样所需要随机数，sampler 产生的随机数都是  $[0, 1]$  范围内的浮点数。虽然采样器完成的任务与渲染相对独立，但是采样器产生的随机数质量将影响最终渲染的质量。



(a) A projection of the first 1024 points onto the first two dimensions. Note the sample clumping.



(a) A projection of the first 1024 points onto the first two dimensions.

图 10: Independent (左) 和 Stratified (右) [2]

在 scene.json 中配置一个 sampler 至少需要以下描述

```
"sampler" : {
    "type" : "...",
    "xSamples" : 3,
    "ySamples" : 3
},
```

这里的 xSamples 和 ySamples 在实现更为高效的 sampler 时将有实际意义，目前来说，每个像素的采样数 spp 是 xSamples 和 ySamples 的乘积。

所有的 sampler 都需要实现一下两个接口

```
1 virtual float next1D() = 0;
2
3 virtual Vector2f next2D() = 0;
```

### IndependentSampler 独立采样器

独立采样器是目前唯一支持的采样器，它的随机采样不使用任何优化策略，它简单地调用 c++ 提供的随机数分布产生  $[0,1]$  内的随机数。

```
1 gen = std::mt19937(rd());
2 dist = std::uniform_real_distribution<float>(.0f, 1.f);
3
4 float IndependentSampler::next1D() {
5     return dist(gen);
6 }
7
8 Vector2f IndependentSampler::next2D() {
9     return Vector2f{dist(gen), dist(gen)};
10 }
```

## 6. Integrator 积分器

在 scene.json 中配置一个积分器如下

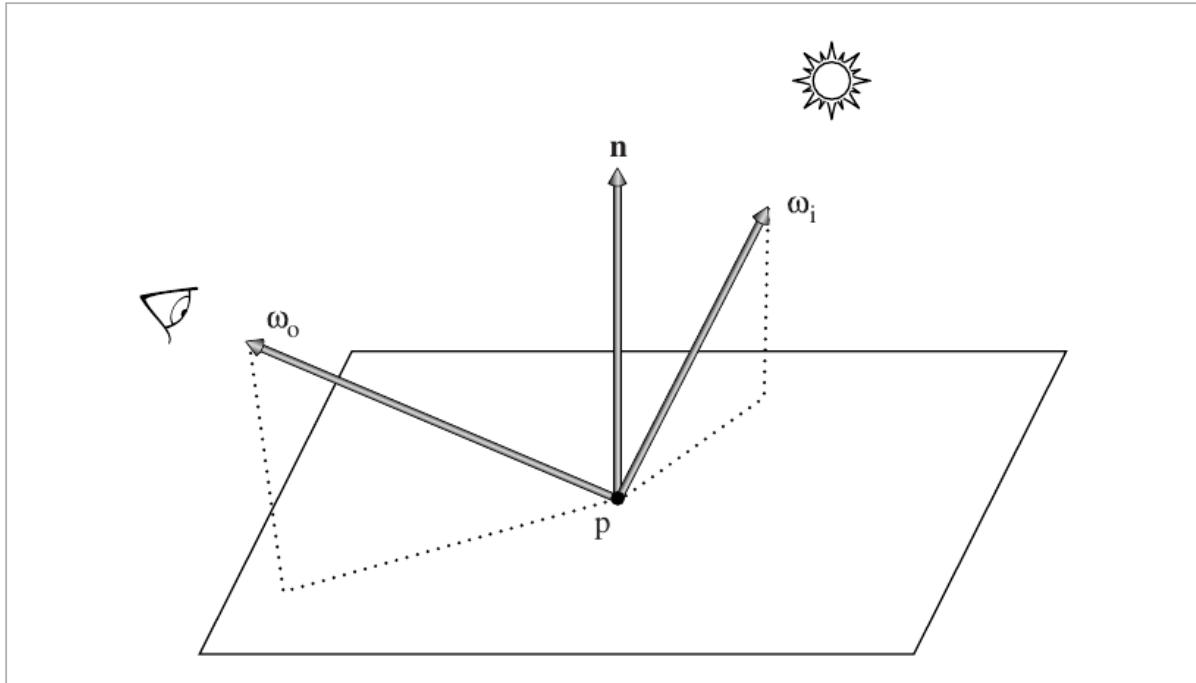
```
"integrator" : {
  "type" : "..."
}
```

目前支持的积分器不需要额外参数，只需要指定类型即可。

积分器 Integrator 完成的任务是整个光追渲染中最重要的、也几乎是最耗时的部分。  
积分器的任务是求解 Rendering Equation[3]。

在不考虑介质以及次表面散射的情况下，积分器主要求解的方程表示如下

$$L_o(p, \vec{\omega}_o) = L_e(p, \vec{\omega}_o) + \int_{\Omega} f_r(p, \vec{\omega}_i, \vec{\omega}_o) L_i(p, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i \quad (1)$$



**Figure 5.18: The BRDF.** The bidirectional reflectance distribution function is a 4D function over pairs of directions  $\omega_i$  and  $\omega_o$  that describes how much incident light along  $\omega_i$  is scattered from the surface in the direction  $\omega_o$ .

图 11: 计算 p 点在出射方向上的 radiance[1]

假设相机采样出一条光线  $R(o, -\vec{\omega}_o)$ , 该光线与场景中的点  $p$  相交, 那么这条光线携带的 radiance 就是  $L_o(p, \vec{\omega}_o)$ 。它由两个部分组成, 首先是  $L_e(p, \vec{\omega}_o)$ , 即点  $p$  自身沿  $\vec{\omega}_o$  方向释放的 radiance (当  $p$  位于光源上时); 另一部分是经过至少一次散射的间接光部分, 是  $\vec{\omega}_i$  方向上的 radiance 在  $p$  处散射至  $\vec{\omega}_o$  方向的部分, 因此第二部分是一个球面积分 (点  $p$  将各个方向上入射的 radiance 散射至出射方向)。

其中  $f_r$  是 BSDF, 即 **Bidirectional Scattering Distribution Function**, 将在材质 Material 部分详细介绍,  $L_e$  是自身发光项, 将在光源 Light 部分介绍。

所有积分器都需要实现以下接口

```
1 virtual Spectrum li(const Ray &ray, const Scene &scene,
2                         std::shared_ptr<Sampler> sampler) const = 0;
```

该函数将计算在给定场景 scene 下, 光线 ray 所携带的 radiance 值, 计算过程需要的所有随机采样由 sampler 提供。

### NormalIntegrator

NormalIntegrator 实际上不计算渲染方程, 而是简单的返回 ray 与 scene 交点处法线的值 (该值会做调整, 以确保没有负数)

```
1 Spectrum NormalIntegrator::li(const Ray &ray, const Scene &scene,
2                                 std::shared_ptr<Sampler> sampler) const {
3     auto intersectionOpt = scene.rayIntersect(ray);
4     if (!intersectionOpt.has_value())
5         return Spectrum(.0f);
6     return Spectrum((intersectionOpt.value().normal + Vector3f(1.f)) * .5f);
7 }
```

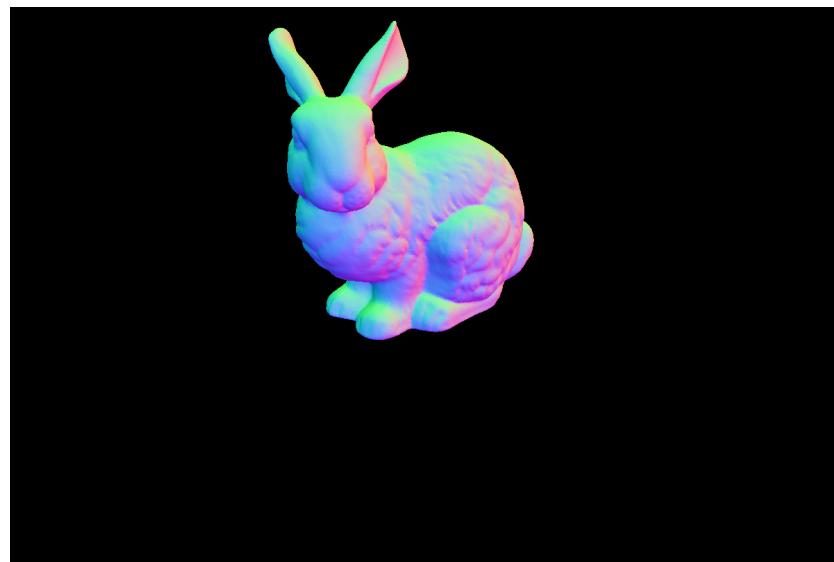


图 12: 用 NormalIntegrator 对兔子模型的法线做可视化

在后续作业调试加速结构、物体求交等功能时推荐使用该积分器。

## DirectIntegrator

DirectIntegrator 积分器实际上只计算了渲染方程的一部分，即交点处自身发光项以及该点处接收到的直接光。在计算过程中，交点处的自身发光是一个确定的值，只取决于光线是否与光源相交。而交点处接收到的直接光则需要求解一个球面积分

$$\int_{\Omega} f_r(p, \vec{\omega}_i, \vec{\omega}_o) L_i(p, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i \quad (2)$$

由于我们只求  $p$  处的直接光照，因此我们只考虑  $p$  在  $\vec{\omega}_i$  上的是否能接受到来自光源的直接光（即  $p$  在  $\vec{\omega}_i$  方向上与光源之间没有遮挡）。所以，我们将 (2) 式改写为如下形式

$$\int_{\Omega} f_r(p, \vec{\omega}_i, \vec{\omega}_o) L_{direct}(p, \vec{\omega}_i) (\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i \quad (3)$$

其中

$$L_{direct}(p, \vec{\omega}_i) = \begin{cases} L_e(p_L, -\vec{\omega}_i), & p \text{ 在 } \vec{\omega}_i \text{ 方向上与光源相交于 } p_L \\ 0, & p \text{ 在 } \vec{\omega}_i \text{ 方向上最近交点不是光源} \end{cases} \quad (4)$$

我们一般使用蒙特卡洛方法对方程 (3) 求解，即

$$\langle \text{方程 (3)} \rangle = \frac{f_r(p, \vec{\omega}, \vec{\omega}_o) L_i(p, \vec{\omega}) (\vec{\omega} \cdot \vec{n})}{pdf} \quad (5)$$

其中  $\vec{\omega}$  是我们采样的到的一个方向， $pdf$  是这个采样对应的概率密度分布函数的值。

根据不同的采样方法，框架中实现了两种直接光照积分器：

DirectIntegratorSampleLight 和 DirectIntegratorSampleBSDF

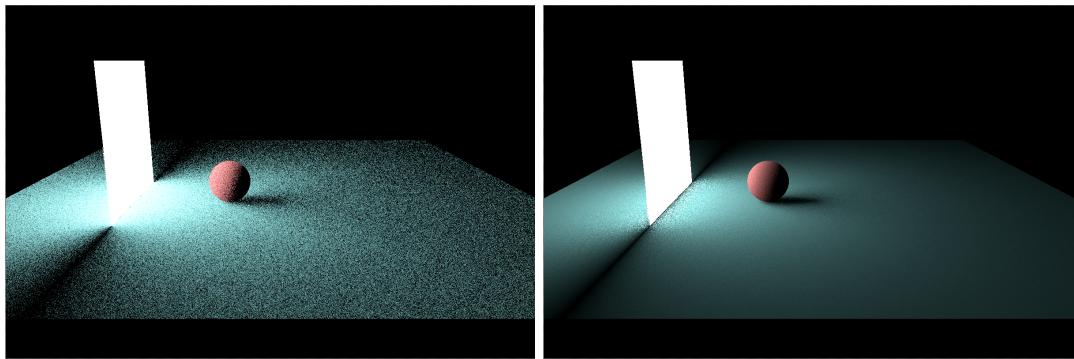


图 13: 采样 BSDF (左) 和采样光源 (右)

可以看到，在场景和采样数均一致的情况下，不同的采样策略生成的图片质量也不相同。这两种方法均有各自的优点和缺陷，具体细节我们将在后续的实验中再详细介绍。

# 7. Shape 几何体

在 scene.json 中配置一个 shape 大致如下

```
{
  "type" : "...",          # shape的类型
  "transform" : {...},    # 对shape做三维空间变换（可选，没有时不做变换）
  "material" : {...},      # shape的材质
  ...                      # shape的其他参数
}
```

## transform 的配置

transform 可以将 shape 的几何信息进行改变，框架支持三种变换

```
"translate" : [1, 2, 3]
```

以上配置在 transform 中添加一个 translate 平移操作，将 shape 的每个点的 x,y,z 分量各平移 1,2,3。

```
"scale" : [1, 2, 3]
```

以上配置在 transform 中添加一个 scale 缩放操作，shape 在 x、y、z 方向上各缩放 1、2、3 倍。

```
"rotate":{
  "axis" : [1, 2, 3],
  "radian" : 0.2
}
```

以上配置在 transform 中添加一个 rotate 旋转操作，将 shape 沿 axis 轴（经过原点）逆时针旋转 radian 弧度。

每个 shape 都需要实现以下三个方法

```
1 virtual bool rayIntersectShape(const Ray &ray, float *distance, int *primID,
2                               float *u, float *v) const = 0;
3
4 virtual void fillIntersection(float distance, int primID, float u, float v,
5                               Intersection *intersection) const = 0;
6
7 virtual void uniformSampleOnSurface(Vector2f sample, Intersection *result,
8                                   float *pdf) const = 0;
```

## rayIntersectShape

```

1 virtual bool rayIntersectShape(const Ray &ray, float *distance, int *primID,
2                               float *u, float *v) const = 0;

```

该方法计算 ray 与当前 shape 的交点，如果存在交点则返回 true，否则返回 false。交点的信息存储在几个以指针形式传递的参数中，distance 存储交点到光线起点的距离，primID、u、v 则共同确定交点在 shape 上的位置。

### fillIntersection

```

1 virtual void fillIntersection(float distance, int primID, float u, float v,
2                               Intersection *intersection) const = 0;

```

该方法对交点处的各种信息进行计算，Intersection 结构如下

```

1 struct Intersection {
2     float distance;           // distance from ray origin to hitpoint
3     Point3f position;        // the position of hitpoint (in world space)
4     Vector3f normal;         // the normal at hitpoint
5     Vector3f tangent, bitangent; // tangent, bitangent and normal form the local tangent space
6     Vector2f texCoord;       // the texture coordinate at hitpoint (optional)
7     const Shape *shape;       // the shape which intersects with ray
8     Vector3f dpdu, dpdv;      // the tangent at hitpoint along u/v direction (optional)
9
10    /* ray differential, which is used for texture filtering, just ignore here
11    float dudx, dvdx, dudy, dvdy;
12    Vector3f dpdx, dpdy;
13 };

```

### uniformSampleOnSurface

```

1 virtual void uniformSampleOnSurface(Vector2f sample, Intersection *result,
2                                     float *pdf) const = 0;

```

该方法接受一个 2 维随机数，随机的在 shape 表面采样一个点，采样的点的信息存在 result 中。这个方法主要在 shape 作为面光源时被调用，详细信息在光源部分的 AreaLight 中介绍。

## 7.1 Sphere 球

在 scene.json 中配置一个球如下

```
{
  "type" : "sphere",
  "center" : [0, 0, 0],
  "radius" : 1
  "transform" : {...},
  "material" : {...}
}
```

目前对于 sphere 的 transform 只有 translate 能够生效，rotate 和 scale 在开发中

上面的配置在场景中添加了一个原点在  $(0, 0, 0)$ ，半径为 1 的球体。

## 7.2 Parallelogram 平行四边形

在 scene.json 中配置一个平行四边形如下

```
{  
    "type" : "parallelogram",  
    "base" : [-10, -10, 0],  
    "edge0" : [20, 0, 0],  
    "edge1" : [0, 20, 0],  
    "transform" :{...},  
    "material" : {...}  
}
```

上面的配置在场景中添加了一个以  $(-10, -10, 0)$  为顶点，两个向量  $(20, 0, 0)$ 、 $(0, 20, 0)$  为边的平行四边形，该平行四边形的法线由 edge0 叉乘 edge1 得到。

## 7.3 Triangle 三角形 Mesh

在 scene.json 中配置一个三角形 Mesh 如下

```
{  
    "type" : "triangle",  
    "file" : ".../model.obj"  
    "transform" :{...},  
    "material" : {...}  
}
```

上面的配置在场景中添加一个三角形 Mesh，可以根据 transform 调整该 shape 的位置。

## 8. Acceleration 加速结构

编写加速结构暂定为第一次实验的内容，因此现在不做过多介绍。目前的框架中只使用了 embree 作为空间加速结构。

所有的加速结构都需要实现以下两个接口

```
1 virtual std::optional<Intersection> rayIntersect(const Ray &ray) const = 0;  
2  
3 virtual void build() = 0;
```

### rayIntersect

rayIntersect 接受一个 Ray 作为参数，负责完成光线与场景的求交，该方法的返回值使用了 C++17 的新特性 std::optional，我们后续会专门用一个章节简单介绍框架中目前使用的 C++11/14/17 特性。语义上来说，std::optional 是一个可选值，就此方法而言，当光线与场景没有交点时，返回值的 *has\_value()* 是 false；而有交点时可以通过 *value()* 方法访问返回的 Intersection 结构。

### build

build 方法负责根据加速结构中已经添加的几何体构建一个空间加速结构，以加速场景与光线求交的过程。

# 9. Light 光源

在 scene.json 中配置一个光源大致如下

```
{
  "type" : "...",      # light的类型
  ...                  # light的其他参数
}
```

对于所有 Light，都应该实现以下两个方法

```
1 virtual Spectrum evaluateEmission(const Intersection &intersection,
2                                     const Vector3f &wo) const = 0;
3 virtual LightSampleResult sample(const Intersection &shadingPoint,
4                                   const Vector2f &sample) const = 0;
```

特别的，当光线不与场景发生相交时，不会生成 Intersection 结构体。但是有些光源仍然会对该光线产生贡献，这些光源可以被理解为无穷远处的光源（例如环境光源）。InfiniteLight 还需要实现以下方法

```
1 virtual Spectrum evaluateEmission(const Ray &ray) const = 0;
```

各类光源的原理，方法的实现将在后续作业中详细介绍。

## 9.1 SpotLight 点光源

配置一个点光源如下

```
{
  "type" : "spotLight",
  "position" : [0, 1, 0],
  "energy" : [10, 10, 10]
}
```

以上配置在场景中 (0, 1, 0) 处添加了一个 RGB 强度为 (10, 10, 10) 的点光源。

## 9.2 AreaLight 面光源

配置一个面光源如下

```
{
  "type" : "areaLight",
  "shape" : {...}      # 此处shape的配置同场景中shape的配置
  "energy" : [10, 10, 10]
}
```

以上配置在场景中添加了一个以 shape 表面为漫反射光源的面光源，其强度为 (10, 10, 10)。目前使用 mesh 表面作为光源的功能还没有提供。

### 9.3 EnvironmentLight 环境光

配置一个环境光源如下

```
{  
    "type" : "environmentLight",  
    "texture" : {...}          # 将一个纹理作为环境贴图使用  
}
```

以上配置在场景中添加了一个以 texture 作为纹理的环境光贴图。**推荐配置 imageTex 图像纹理。**

# 10. Material 材质

材质一般作为 shape 的一个属性进行配置，这也是渲染器功能最丰富的部分之一。  
配置大致如下

```
{  
    "type" : "...",           # 材质的类型  
    ...                      # 材质的其他参数  
}
```

每个材质都应当实现以下接口

```
1 virtual std::shared_ptr<BSDF> computeBSDF(const Intersection &intersection) const = 0;
```

该方法根据 intersection 处的信息返回交点处的 BSDF 描述。

## 10.1 Matte 漫反射材质

配置一个漫反射材质

```
"material" : {  
    "type" : "matte",  
    "albedo" : ...      # albedo 项的值可以是一个 spectrum, 也可以是一个纹理  
}
```

以上配置了一个漫反射材质，其 albedo 由一个纹理或常量 spectrum 决定。

目前只支持 Matte 漫反射材质，其余材质和 BSDF 的描述将后续进行更新

# 11. Texture 纹理

纹理通常作为一个属性被配置，配置一个纹理大致如下

```
{
    "type" : "...",    # texture 的类型
    ...                # texture 的其他属性
}
```

每一种纹理都应当实现以下接口

```
1 virtual TReturn evaluate(const Intersection &intersection) const = 0;
2
3 virtual TReturn evaluate(const TextureCoord &texCoord) const = 0;
```

## evaluate

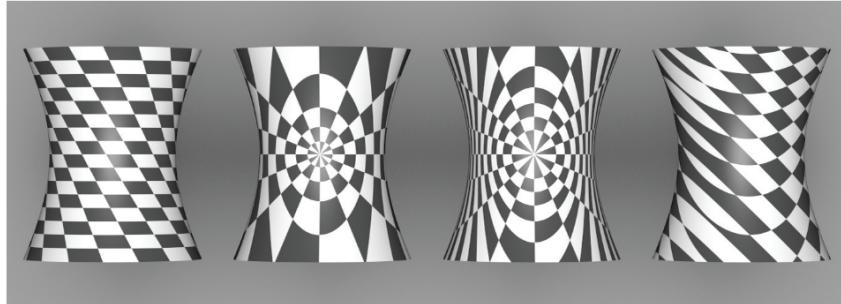
evaluate 方法有两个，一种接受 Intersection 为参数，一种接受 TextureCoord 为参数。对于接受 Intersection 为参数的方法，都可以使用 texture 类的 mapping 成员将一个 intersection 对应的 TextureCoord 计算出来，以进行代码复用。

```
1 auto textureCoord = mapping->map(intersection);
2 TReturn ret = evaluate(textureCoord);
```

evaluate 负责根据参数对象提供的信息，计算纹理对应处的值。

## TextureMapping 纹理映射

对于每个 texture 对象，其都有一个纹理映射成员。纹理映射主要用于将 intersection 交点处信息转换为一个直观的、被用于进行纹理值计算的 TextureCoord 纹理坐标对象。不同的映射方式会产生不同的结果。目前只支持最简单的 UV 坐标对应映射。



**Figure 10.7:** A checkerboard texture, applied to a hyperboloid with different texture coordinate generation techniques. From left to right,  $(u, v)$  mapping, spherical mapping, cylindrical mapping, and planar mapping.

图 14: 不同映射将产生不同结果 [1]

### 11.1 ImageTexture 图像纹理

```
{
  "type" : "imageTex",    # texture 的类型
  "file" : "...",         # 图像的路径
}
```

#### 各向同性纹理过滤

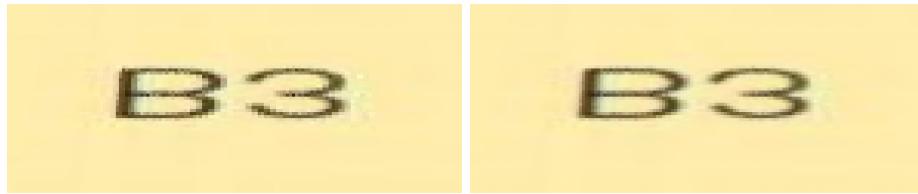


图 15: 不做过滤 (左) 和进行双线性插值过滤 (右)

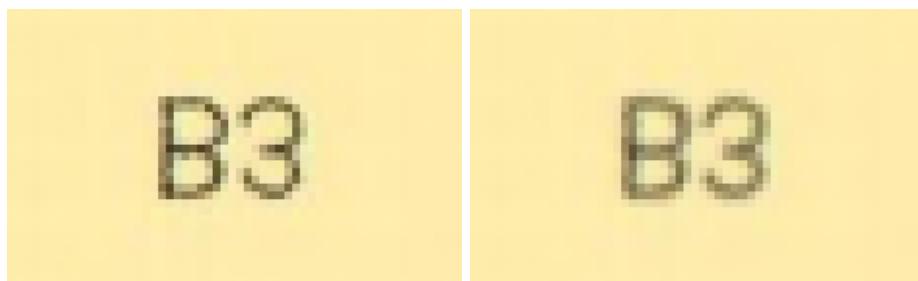


图 16: 双线性插值过滤 (左) 和使用 mipmap 的三线性过滤插值 (右)

#### 各向异性纹理过滤

各向同性过滤在有时候效果很差，这时需要进行各向异性过滤。目前框架还不支持各向异性纹理过滤，我们将在后续完善各向异性过滤的解释与实现。



图 17: 对应处文字为 E3，可以看到各向同性过滤不能很好地处理该情况

### 11.2 NormalTexture 法线贴图

```
"normalmap" : {
  "file" : "...",         # 法线贴图的路径
}
```

法线贴图可以为简单的几何体生成更丰富的细节，作为 material 配置的一部分。

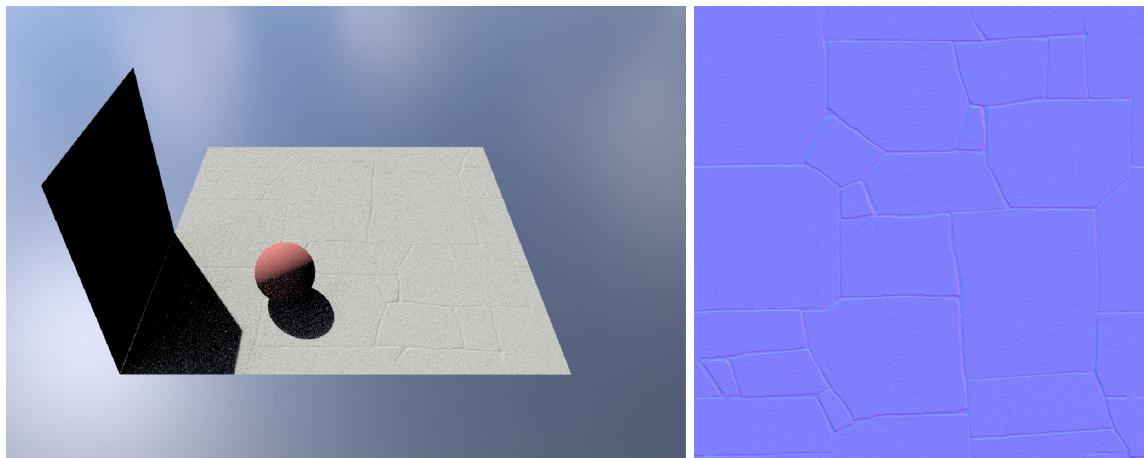


图 18：平行四边形在使用了法线贴图后表面的法线发生了变化

## 12. 作业

1. 编译 Moer-lite
2. 运行提供的测试场景
3. (bonus) 可以自行构建场景并运行测试

## 13. References

### Bibliography

- [1] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. Physically Based Rendering: From Theory to Implementation (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [2] Wenzel Jakob. Mitsuba <https://www.mitsuba-renderer.org/docs.html>
- [3] James T. Kajiya. 1986. The rendering equation. SIGGRAPH Comput. Graph. 20, 4 (Aug. 1986), 143–150. <https://doi.org/10.1145/15886.15902>