PBRT 系列 19-专业理论知识与代码实战-复杂模型的读取接口

Dezeming Family

2021 年 4 月 5 日

因为本书是电子书，所以会不断进行更新和再版（更新频率会很高）。如果您从其他地方得到了这本书，可以从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：学习模型读取库 Assimp 的使用，编写一个模型读取接口，渲染出下面的图像：

前言

以前我们都是自己制作一些模型（比如康奈尔盒，比如箱子模型），或者使用一些简单的 obj 模型。这些模型没有携带纹理信息，所以没法渲染出特别好看的图像。

本书我们将自己实现一个接口，来读取和渲染更好看的模型。首先先使用 Assimp 来构造模型读入系统，然后加载纹理。

一个好的方法是自己写一个文件转换程序，将 Assimp 读取到的文件写成自己喜欢的文件格式并保存，供自己的系统来使用，方法很简单，因此这里不介绍了。

本书的售价是 3 元（电子版），但是并不直接收取费用。如果您免费得到了这本书的电子版，在学习和实现时觉得有用，可以往我们的支付宝账户（17853140351，可备注：PBRT）进行支持，您的赞助将是我们 Dezeming Family 继续创作各种图形学、机器学习、以及数学原理小册子的动力！

目录

一 PBRT 三角形回顾

本章我们回顾一下 PBRT 中的三角形和 TriangleMesh 的读取和加载流程。

1 1 TriagnleMesh

在 Triangle 类中，顶点索引为：

```
v = &mesh->vertexIndices[3 * triNumber];
```

而法向量和纹理坐标的读取坐标都是：

```
//顶点坐标
mesh->p[v[0]]
mesh->p[v[1]]
mesh->p[v[2]]
//纹理坐标
mesh->uv[v[0]];
mesh->uv[v[1]];
mesh->uv[v[2]];
//法向量坐标
mesh->n[v[0]]
mesh->n[v[1]]
mesh->n[v[2]]
```

因此，PBRT 程序中认为，顶点坐标和法向量坐标、纹理坐标都是相同的顺序。但是如果您打开一些 obj 文件，就会看到这样的数据

```
f     51/3/51 52/2/52 48/1/48
```

它表示的分别是顶点坐标索引、法向量坐标索引和纹理坐标索引，这三个索引是不同的，但模型读取文件会帮我们自动修改为相同的索引顺序，即一个顶点索引包含了其他所有的索引方法，因此我们不需要在意这些细节。

1 2 三角形结构的生成

我们回顾一下一个四边形的生成过程：

```
const int nTrianglesWall = 2; //三角形面片数量
int vertexIndicesWall[nTrianglesWall * 3]; //顶点索引
for (int i = 0; i < nTrianglesWall * 3; i++)
    vertexIndicesWall[i] = i;
//顶点数（尽管四边形只需要4个顶点索引值，但我们不考虑重复使用顶点）
const int nVerticesWall = nTrianglesWall * 3;
Point3f P_wall[nVerticesWall] = {
    Point3f(0.f,0.f,1.f),Point3f(1.f,0.f,zlength), Point3f(0.f,0.f,0.f),
    Point3f(1.f,0.f,zlength),Point3f(1.f,0.f,0.f),Point3f(0.f,0.f,0.f)
};
//tri_Object2World是三维变换类
std::shared_ptr<TriangleMesh> meshFloor = std::make_shared<TriangleMesh>
    (tri_Object2World, nTrianglesWall, vertexIndicesWall, nVerticesWall,
       P_wall, nullptr, nullptr, nullptr, nullptr);
```

```
Transform tri_World2Object = Inverse(tri_Object2World);
std::vector<std::shared_ptr<Shape>> trisFloor;
for (int i = 0; i < nTrianglesWall; ++i)
    trisFloor.push_back(std::make_shared<Triangle>(&tri_Object2World, &
        tri_World2Object, false, meshFloor, i, shapeID++));
for (int i = 0; i < trisFloor.size(); ++i)
    prims.push_back(std::make_shared<GeometricPrimitive>(trisFloor[i], mat,
        nullptr));
```

构建三角形的方法非常简单，所以这里就不再多说了。

1 3 三角形构造方法

在 TriangleMesh 类的构造函数参数列表中表示顶点索引的成员变量：

```
//  const int *vertexIndices
// TriangleMesh Public Methods
TriangleMesh(const Transform &ObjectToWorld, int nTriangles,
    const int *vertexIndices, int nVertices, const Point3f *P,
    const Vector3f *S, const Normal3f *N, const Point2f *uv,
    const int *faceIndices);
//成员变量
std::vector<int> vertexIndices;
```

在构造函数中该成员变量赋值：

```
        vertexIndices(vertexIndices, vertexIndices + 3 * nTriangles)
```

在 GetUVs 函数和 Sample 函数，Intersect 函数里都有包括法向量和纹理的索引：

```
    mesh->n[v[i]]
    mesh->uv[v[i]]
```

二 Assimp

assimp[3] 是一模型导入库，可以导入多种模型。因此我们就来学习一下这个库，并写一个接口，通过该库导入模型到 PBRT 系统里。

本章更建议读者自己去动手实现，加深对模型和 PBRT 系统的了解，如果有问题或者不知道如何下手，可以参考下面的章节。

2 1 Assimp 库的初步使用

编译和安装可以参考 [2]，网上也有很多资料，我不想再写一遍了。

关于如何使用 assimp，直接讲解很抽象，我们一边动手做就可以很容易地搞明白了。

我们先建立两个文件，ModelLoad.h 和 ModelLoad.cpp，然后定义类：class ModelLoad。

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
class ModelLoad {
    public:
        /*  模型数据  */
        std::vector<std::shared_ptr<TriangleMesh>> meshes;
        std::string directory;
        /*  函数   */
        void loadModel(std::string path, const Transform &ObjectToWorld);
        void processNode(aiNode *node, const aiScene *scene, const Transform
            &ObjectToWorld);
        std::shared_ptr<TriangleMesh> processMesh(aiMesh *mesh, const
           aiScene *scene, const Transform &ObjectToWorld);
    };
```

其中下面的两个函数参考 [2] 得到（大家可以理解为，模型是一棵树，每个枝干都索引了一些 mesh，每个 mesh 表示模型的一部分，比如人体模型中，手可以作为一个 mesh，头可以作为另一个 mesh，因此需要针对树这种结构采用递归的方法遍历，生成各个 mesh 对象）：

```
void ModelLoad::processNode(aiNode *node, const aiScene *scene, const
   Transform &ObjectToWorld){
        // 处理节点所有的网格（如果有的话）
        for (unsigned int i = 0; i < node->mNumMeshes; i++){
                aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
                meshes.push_back(processMesh(mesh, scene, ObjectToWorld));
        }
        // 接下来对它的子节点重复这一过程
        for (unsigned int i = 0; i < node->mNumChildren; i++)
                processNode(node->mChildren[i], scene, ObjectToWorld);
}
void ModelLoad::loadModel(std::string path, const Transform &ObjectToWorld){
    Assimp::Importer import;
    const aiScene *scene = import.ReadFile(path, aiProcess_Triangulate |
       aiProcess_FlipUVs);
    if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->
       mRootNode)
```

```
        return;
    directory = path.substr(0, path.find_last_of('/'));
    processNode(scene->mRootNode, scene, ObjectToWorld);
}
```

现在的问题是我们怎么定义 processMesh 函数来生成 TriangleMesh。

2 2 没有纹理对象的 mesh 模型

设计模型接口比较困难的地方在于 mesh 是有纹理的，因此怎么把 mesh 绑定纹理并不简单。因此我们先实现一个没有纹理的版本，不过它加载了纹理坐标 uv：

```
TriangleMesh ModelLoad::processMesh(aiMesh *mesh, const aiScene *scene,
    const Transform &ObjectToWorld) {
    size_t nVertices = mesh->mNumVertices;
    size_t nTriangles = mesh->mNumFaces;
    int *vertexIndices = new int[nTriangles * 3];
    Point3f *P = new Point3f[nVertices];
    Vector3f *S = nullptr;// new Vector3f[nVertices];
    Normal3f *N = new Normal3f[nVertices];
    Point2f *uv = new Point2f[nVertices];
    int *faceIndices = nullptr; //第几个面的编号
    for (unsigned int i = 0; i < mesh->mNumVertices; i++) {
        //顶点
        P[i].x = mesh->mVertices[i].x;
        P[i].y = mesh->mVertices[i].y;
        P[i].z = mesh->mVertices[i].z;
        //法向量
        if (mesh->HasNormals()){
            N[i].x = mesh->mNormals[i].x;
            N[i].y = mesh->mNormals[i].y;
            N[i].z = mesh->mNormals[i].z;
        }
        //纹理坐标
        if (mesh->mTextureCoords[0]) {
            uv[i].x = mesh->mTextureCoords[0][i].x;
            uv[i].y = mesh->mTextureCoords[0][i].y;
        }
    }
    for (unsigned int i = 0; i < mesh->mNumFaces; i++){
        aiFace face = mesh->mFaces[i];
        for (unsigned int j = 0; j < face.mNumIndices; j++)
            vertexIndices[3 * i + j] = face.mIndices[j];
    }
    if (!mesh->HasNormals()) {
        delete[] N;
        N = nullptr;
    }
    if (!mesh->mTextureCoords[0]) {
```

```
        delete[] uv;
        uv = nullptr;
    }
    std::shared_ptr<TriangleMesh> a =
    std::make_shared<TriangleMesh>(ObjectToWorld, nTriangles, vertexIndices,
        nVertices,
        vertexIndices, vertexIndices, P, S, N, uv, faceIndices);
    delete[] P;
    delete[] S;
    delete[] N;
    delete[] uv;
    return a;
}
```

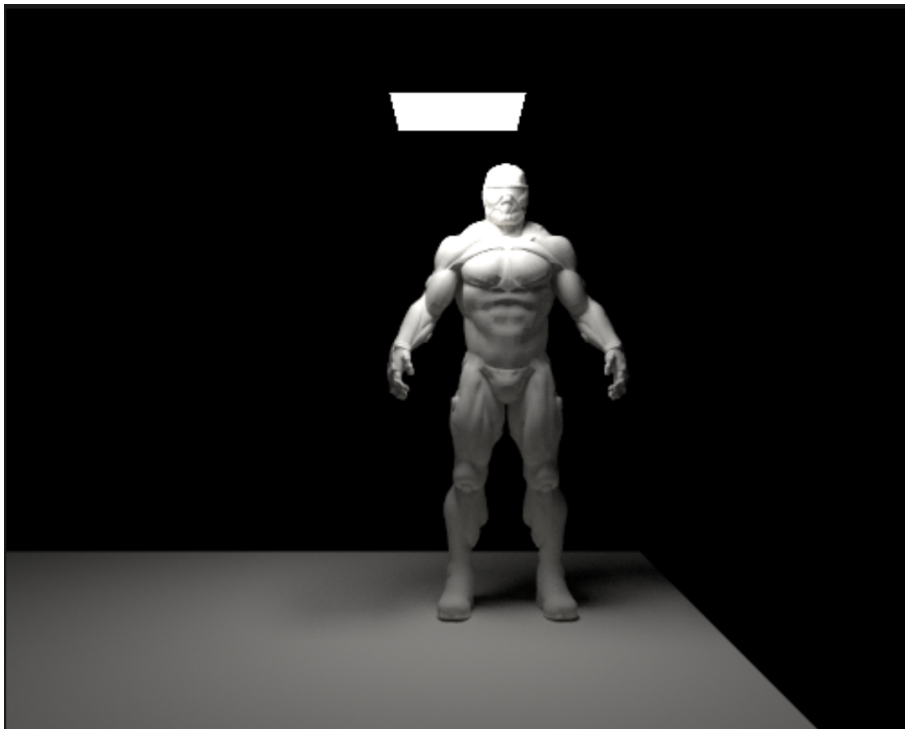
我们还需要一个函数来生成对象：

```
void ModelLoad::buildNoTextureModel(Transform& tri_Object2World,
        std::vector<std::shared_ptr<Primitive>> &prims, std::shared_ptr<
            Material> material) {
        std::vector<std::shared_ptr<Shape>> trisObj;
        Transform tri_World2Object = Inverse(tri_Object2World);
        for (int i = 0;i < meshes.size();i++) {
                for (int j = 0; j < meshes[i]->nTriangles; ++j) {
                        std::shared_ptr<TriangleMesh> meshPtr = meshes[i];
                        trisObj.push_back(std::make_shared<Triangle>(&
                            tri_Object2World, &tri_World2Object, false,
                            meshPtr, j, shapeID++));
                }
        }
        //将物体填充到基元
        for (int i = 0; i < trisObj.size(); ++i)
                prims.push_back(std::make_shared<GeometricPrimitive>(trisObj
                    [i], material, nullptr));
}
```

并用下面几行代码来构造：

```
    Transform tri_Object2WorldModel = Scale(0.3, 0.3, 0.3);
    ModelLoad ML; ML.loadModel("模型路径", tri_Object2WorldModel);
    ML.buildNoTextureModel(tri_Object2WorldModel, prims, floorMaterial);
```

渲染得到如下结果：



三　纹理加载

对于模型来说，纹理加载是一个比较令人头疼的话题。不同的模型，其纹理的种类完全不同，例如 PBR 模型和一般的游戏使用的模型。

纹理是需要加载到材质中才能进行使用的，不同的游戏引擎都会使用不同的模型系统，想做一个比较通用的模型系统并不是很容易，因此我们也不会刻意去处处要求完美。

3 1　有纹理对象的 mesh 模型

在 [2] 的关于模型的章节里，有 nanosuit 模型，该模型文件为 nanosuit.obj，引用了 nanosuit.mtl 文件。

```
1     newmtl Arm
2     Ns 96.078431
3     Ka 0.000000 0.000000 0.000000
4     Kd 0.640000 0.640000 0.640000
5     Ks 0.500000 0.500000 0.500000
6     Ni 1.000000
7     d 1.000000
8     illum 2
9     map_Kd arm_dif.png
10    map_Bump arm_showroom_ddn.png
11    map_Ks arm_showroom_spec.png
```

该文件会对不同照明组件（漫反射、镜面反射）贴图进行定义，还包含了照明系数。map_Kd 表示漫反射贴图，map_Bump 表示凹凸贴图，map_Ks 表示镜面反射贴图。这组模型主要是用于 Phong 照明的，我们在读入的时候需要注意，这里我们就当它是个微表面模型，分为漫反射和镜面反射组件（注意有的基于物理的模型还有其他组件，比如粗糙度等）。

想做一个通用的纹理加载的过程实在是困难（主要是因为大多数模型都与我们需要的不匹配，这也是为什么 PBRT 要做模型转换的工作并使用自己定义的模型接口，因为模型读入和使用的接口设计确实需要依赖于自己系统的实现）。

assimp 可以读取复杂的模型文件，构成一颗 Mesh 树，而我们可以用非递归的方法来读取 Mesh 树，但一个通用的文件读取器难以为每个 Mesh 赋值材质。所以我们还是使用 PBRT 的方法，读取一个 Mesh 并为其赋予材质属性。

首先先定义一个成员变量：

```
1         std::vector<std::string> texName; //纹理序列
```

processMesh 函数中，我们加载纹理：

```
1  //加载纹理
2  aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
3  for (unsigned int i = 0; i < material->GetTextureCount(aiTextureType_DIFFUSE
       ); i++) {
4      aiString str;
5      material->GetTexture(aiTextureType_DIFFUSE, i, &str);
6      std::string filename = str.C_Str();
7      texName.push_back(filename);
8  }
```

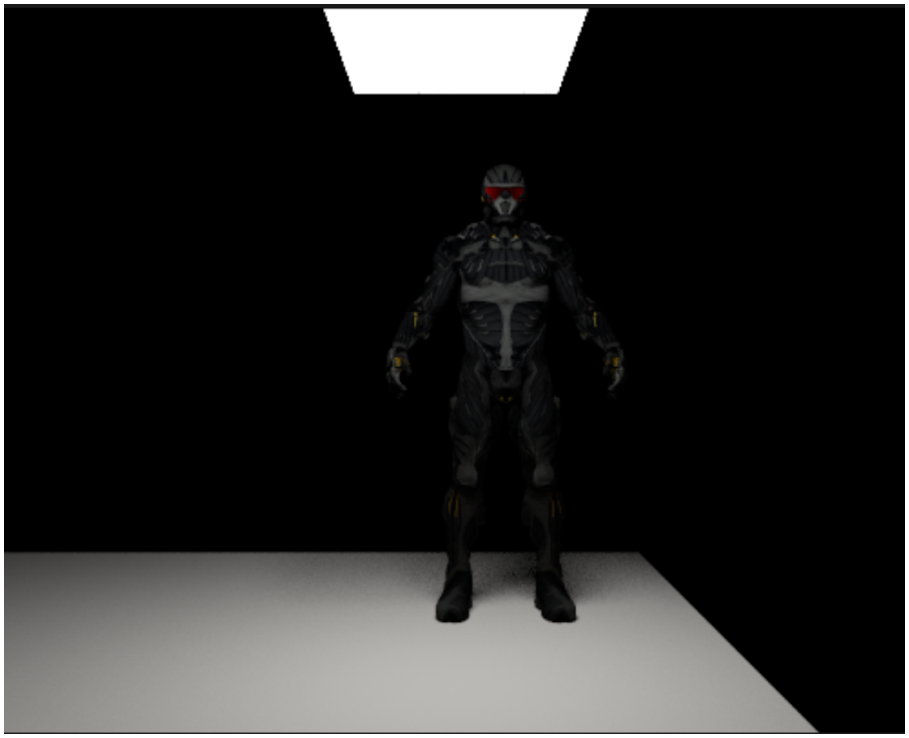
定义一个简单的加载纹理的函数：

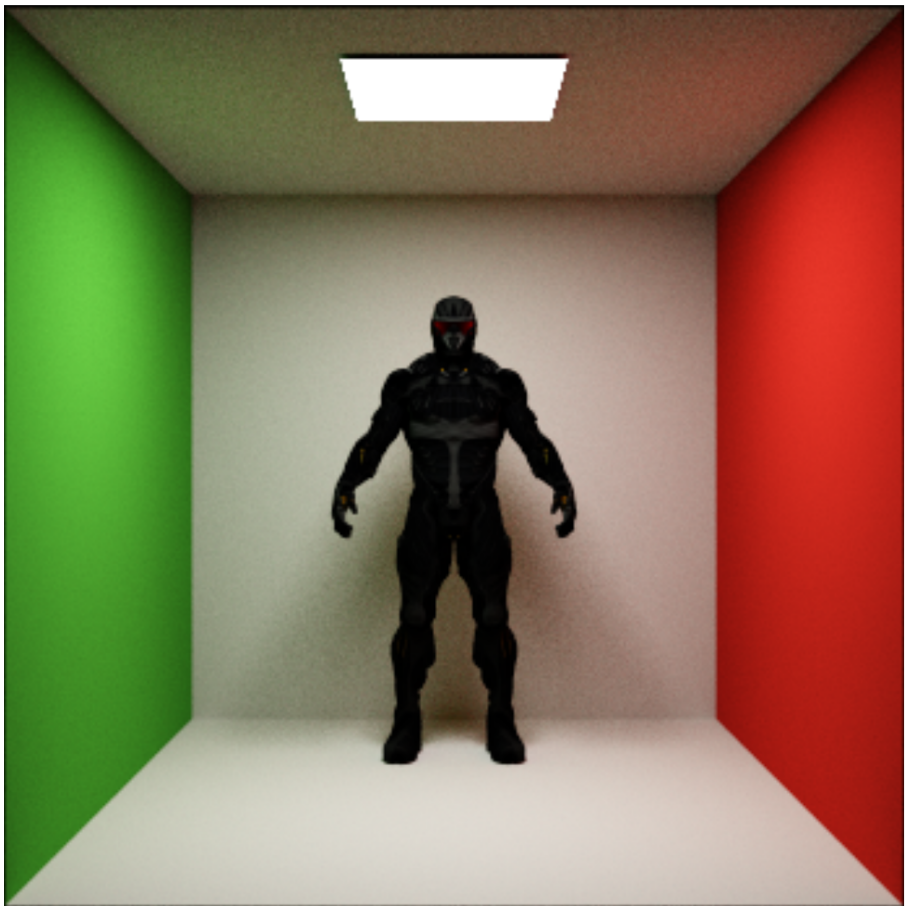
```
inline std::shared_ptr<Material> getDiffuseMaterial(std::string filename) {
    std::unique_ptr<TextureMapping2D> map = std::make_unique<UVMapping2D>(1.
        f, 1.f, 0.f, 0.f);
    ImageWrap wrapMode = ImageWrap::Repeat;
    bool trilerp = false;
    float maxAniso = 8.f;
    float scale = 1.f;
    bool gamma = false;
    std::shared_ptr<Texture<Spectrum>> Kt = std::make_shared<ImageTexture<
        RGBSpectrum, Spectrum>>(std::move(map), filename, trilerp,
         maxAniso, wrapMode, scale, gamma);
    std::shared_ptr<Texture<float>> sigmaRed = std::make_shared<
        ConstantTexture<float>>(0.0f);
    std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
        ConstantTexture<float>>(0.0f);
    return std::make_shared<MatteMaterial>(Kt, sigmaRed, bumpMap);
}
```

然后是包含纹理的模型构造：

```
    void ModelLoad::buildTextureModel(Transform& tri_Object2World,
        std::vector<std::shared_ptr<Primitive>> &prims) {
        std::vector<std::shared_ptr<Shape>> trisObj;
        Transform tri_World2Object = Inverse(tri_Object2World);
        for (int i = 0; i < meshes.size(); i++) {
                std::string filename = directory + "/" + texName[i];
                std::shared_ptr<Material> material = getDiffuseMaterial(
                    filename);
                for (int j = 0; j < meshes[i]->nTriangles; ++j) {
                        std::shared_ptr<TriangleMesh> meshPtr = meshes[i];
                        prims.push_back(std::make_shared<GeometricPrimitive
                            >(
                                std::make_shared<Triangle>(&tri_Object2World
                                    , &tri_World2Object, false, meshPtr, j,
                                    shapeID++),
                                material, nullptr));
                }
        }
}
```

得到渲染结果如下：





3 2 微表面材质

对于 Mesh 来说，还包含其他纹理类型，例如镜面反射纹理，可以与漫反射纹理一起作为材料材质的参数。

在 processMesh 函数中，我们加载纹理，防止可能携带多个同类型纹理，我们没有写程序检测，因此明确只加载且一定加载 1 个：

```
1 //加载纹理，只加载一个
2 aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];
```

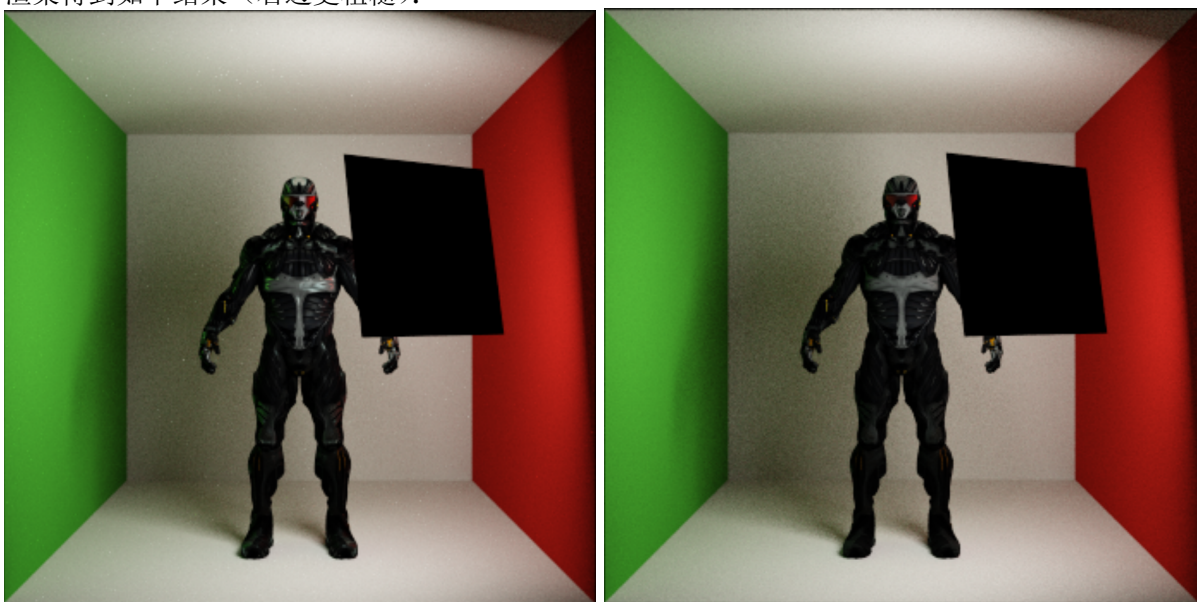
```
int count = material->GetTextureCount(aiTextureType_DIFFUSE);
if (count == 0) {
    diffTexName.push_back("");
}
else {
    for (unsigned int i = 0; i < material->GetTextureCount(
        aiTextureType_DIFFUSE); i++) {
        aiString str;
        material->GetTexture(aiTextureType_DIFFUSE, i, &str);
        std::string filename = str.C_Str();
        diffTexName.push_back(filename);
        break;
    }
}
count = material->GetTextureCount(aiTextureType_SPECULAR);
if (count == 0) {
    specTexName.push_back("");
}
else {
    for (unsigned int i = 0; i < material->GetTextureCount(
        aiTextureType_SPECULAR); i++) {
        aiString str;
        material->GetTexture(aiTextureType_SPECULAR, i, &str);
        std::string filename = str.C_Str();
        specTexName.push_back(filename);
        break;
    }
}
```

获得塑料材质：

```
inline std::shared_ptr<Material> getPlasticMaterial(std::string diffFilename
    , std::string specFilename) {
    std::unique_ptr<TextureMapping2D> map1 = std::make_unique<UVMapping2D
        >(1.f, 1.f, 0.f, 0.f);
    std::unique_ptr<TextureMapping2D> map2 = std::make_unique<UVMapping2D
        >(1.f, 1.f, 0.f, 0.f);
    ImageWrap wrapMode = ImageWrap::Repeat;
    bool trilerp = false;
    float maxAniso = 8.f;
    float scale = 1.f;
    bool gamma = false; //如果是tga和png就是true;
    std::shared_ptr<Texture<Spectrum>> plasticKd = std::make_shared<
        ImageTexture<RGBSpectrum, Spectrum>>(std::move(map1), diffFilename,
        trilerp,
        maxAniso, wrapMode, scale, gamma);
    std::shared_ptr<Texture<Spectrum>> plasticKr = std::make_shared<
        ImageTexture<RGBSpectrum, Spectrum>>(std::move(map2), specFilename,
```

```
        trilerp,
12          maxAniso, wrapMode, scale, gamma);
13      std::shared_ptr<Texture<float>> plasticRoughness = std::make_shared<
            ConstantTexture<float>>(0.1f);
14      std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
            ConstantTexture<float>>(0.0f);
15      return std::make_shared<PlasticMaterial>(plasticKd, plasticKr,
            plasticRoughness, bumpMap, true);
16  }
```

渲染得到如下结果（右边更粗糙）：



这本书虽然特别短，但重在实践。想设计或实现一个完整的模型加载并非易事，很多渲染器都会封装自己的模型类型。这个话题可以说很久，但我们学习的时候不需要搞太复杂，一般都是使用一个固定的模型来渲染。

参考文献

[1] Pharr M, Jakob W, Humphreys G. Physically based rendering: From theory to implementation[M]. Morgan Kaufmann, 2016.

[2] https://learnopengl.com/

[3] http://assimp.org/

[4] https://free3d.com/3d-models/