

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	1
1.1 TriangleMesh	1
1.2 三角形结构的生成	1
1.3 三角形构造方法	2
二 Assimp	3
2.1 Assimp 库的初步使用	3
2.2 没有纹理对象的 mesh 模型	4
三 纹理加载	7
3.1 有纹理对象的 mesh 模型	7
3.2 微表面材质	9
参考文献	12

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 TriangleMesh

在 Triangle 类中，顶点索引为：

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

而法向量和纹理坐标的读取坐标都是：

```
1 // 顶点坐标
2 mesh->p[v[0]]
3 mesh->p[v[1]]
4 mesh->p[v[2]]
5 // 纹理坐标
6 mesh->uv[v[0]];
7 mesh->uv[v[1]];
8 mesh->uv[v[2]];
9 // 法向量坐标
10 mesh->n[v[0]]
11 mesh->n[v[1]]
12 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

```
1 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：

```
1 const int nTrianglesWall = 2; // 三角形面片数量
2 int vertexIndicesWall[nTrianglesWall * 3]; // 顶点索引
3 for (int i = 0; i < nTrianglesWall * 3; i++)
4     vertexIndicesWall[i] = i;
5 // 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）
6 const int nVerticesWall = nTrianglesWall * 3;
7 Point3f P_wall[nVerticesWall] = {
8     Point3f(0.f, 0.f, 1.f), Point3f(1.f, 0.f, zlength), Point3f(0.f, 0.f, 0.f),
9     Point3f(1.f, 0.f, zlength), Point3f(1.f, 0.f, 0.f), Point3f(0.f, 0.f, 0.f)
10 };
11 // tri_Object2World 是三维变换类
12 std::shared_ptr<TriangleMesh> meshFloor = std::make_shared<TriangleMesh>
13     (tri_Object2World, nTrianglesWall, vertexIndicesWall, nVerticesWall,
14     P_wall, nullptr, nullptr, nullptr, nullptr);
```

```

14 Transform tri_World2Object = Inverse(tri_Object2World);
15 std::vector<std::shared_ptr<Shape>> trisFloor;
16 for (int i = 0; i < nTrianglesWall; ++i)
17     trisFloor.push_back(std::make_shared<Triangle>(&tri_Object2World, &
18         tri_World2Object, false, meshFloor, i, shapeID++));
19 for (int i = 0; i < trisFloor.size(); ++i)
20     prims.push_back(std::make_shared<GeometricPrimitive>(trisFloor[i], mat,
21         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：

```

1 // const int *vertexIndices
2 // TriangleMesh Public Methods
3 TriangleMesh(const Transform &ObjectToWorld, int nTriangles,
4     const int *vertexIndices, int nVertices, const Point3f *P,
5     const Vector3f *S, const Normal3f *N, const Point2f *uv,
6     const int *faceIndices);
7 //成员变量
8 std::vector<int> vertexIndices;

```

在构造函数中该成员变量赋值：

```

1     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：

```

1     mesh->n[v[i]]
2     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。

```
1 #include <assimp/Importer.hpp>
2 #include <assimp/scene.h>
3 #include <assimp/postprocess.h>
4 class ModelLoad {
5     public:
6         /* 模型数据 */
7         std::vector<std::shared_ptr<TriangleMesh>> meshes;
8         std::string directory;
9         /* 函数 */
10        void loadModel(std::string path, const Transform &ObjectToWorld);
11        void processNode(aiNode *node, const aiScene *scene, const Transform
            &ObjectToWorld);
12        std::shared_ptr<TriangleMesh> processMesh(aiMesh *mesh, const
            aiScene *scene, const Transform &ObjectToWorld);
13    };
```

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）：

```
1 void ModelLoad::processNode(aiNode *node, const aiScene *scene, const
    Transform &ObjectToWorld){
2     // 处理节点所有的网格（如果有的话）
3     for (unsigned int i = 0; i < node->mNumMeshes; i++){
4         aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
5         meshes.push_back(processMesh(mesh, scene, ObjectToWorld));
6     }
7     // 接下来对它的子节点重复这一过程
8     for (unsigned int i = 0; i < node->mNumChildren; i++)
9         processNode(node->mChildren[i], scene, ObjectToWorld);
10 }
11 void ModelLoad::loadModel(std::string path, const Transform &ObjectToWorld){
12     Assimp::Importer import;
13     const aiScene *scene = import.ReadFile(path, aiProcess_Triangulate |
        aiProcess_FlipUVs);
14     if(!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->
        mRootNode)
```

```

15     return ;
16     directory = path.substr(0, path.find_last_of('/'));
17     processNode(scene->mRootNode, scene, ObjectToWorld);
18 }

```

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：

```

1 TriangleMesh ModelLoad::processMesh(aiMesh *mesh, const aiScene *scene,
2   const Transform &ObjectToWorld) {
3     size_t nVertices = mesh->mNumVertices;
4     size_t nTriangles = mesh->mNumFaces;
5     int *vertexIndices = new int[nTriangles * 3];
6     Point3f *P = new Point3f[nVertices];
7     Vector3f *S = nullptr; // new Vector3f[nVertices];
8     Normal3f *N = new Normal3f[nVertices];
9     Point2f *uv = new Point2f[nVertices];
10    int *faceIndices = nullptr; // 第几个面的编号
11    for (unsigned int i = 0; i < mesh->mNumVertices; i++) {
12        // 顶点
13        P[i].x = mesh->mVertices[i].x;
14        P[i].y = mesh->mVertices[i].y;
15        P[i].z = mesh->mVertices[i].z;
16        // 法向量
17        if (mesh->HasNormals()) {
18            N[i].x = mesh->mNormals[i].x;
19            N[i].y = mesh->mNormals[i].y;
20            N[i].z = mesh->mNormals[i].z;
21        }
22        // 纹理坐标
23        if (mesh->mTextureCoords[0]) {
24            uv[i].x = mesh->mTextureCoords[0][i].x;
25            uv[i].y = mesh->mTextureCoords[0][i].y;
26        }
27    }
28    for (unsigned int i = 0; i < mesh->mNumFaces; i++) {
29        aiFace face = mesh->mFaces[i];
30        for (unsigned int j = 0; j < face.mNumIndices; j++)
31            vertexIndices[3 * i + j] = face.mIndices[j];
32    }
33    if (!mesh->HasNormals()) {
34        delete [] N;
35        N = nullptr;
36    }
37    if (!mesh->mTextureCoords[0]) {

```

```

37     delete [] uv;
38     uv = nullptr;
39 }
40 std::shared_ptr<TriangleMesh> a =
41 std::make_shared<TriangleMesh>(ObjectToWorld, nTriangles, vertexIndices,
42     nVertices,
43     vertexIndices, vertexIndices, P, S, N, uv, faceIndices);
44 delete [] P;
45 delete [] S;
46 delete [] N;
47 delete [] uv;
48 return a;
49 }

```

我们还需要一个函数来生成对象:

```

1 void ModelLoad::buildNoTextureModel(Transform& tri_Object2World,
2     std::vector<std::shared_ptr<Primitive>> &prims, std::shared_ptr<
3     Material> material) {
4     std::vector<std::shared_ptr<Shape>> trisObj;
5     Transform tri_World2Object = Inverse(tri_Object2World);
6     for (int i = 0; i < meshes.size(); i++) {
7         for (int j = 0; j < meshes[i]->nTriangles; ++j) {
8             std::shared_ptr<TriangleMesh> meshPtr = meshes[i];
9             trisObj.push_back(std::make_shared<Triangle>(&
10                 tri_Object2World, &tri_World2Object, false,
11                 meshPtr, j, shapeID++));
12         }
13     }
14     //将物体填充到基元
15     for (int i = 0; i < trisObj.size(); ++i)
16         prims.push_back(std::make_shared<GeometricPrimitive>(trisObj
17             [i], material, nullptr));
18 }

```

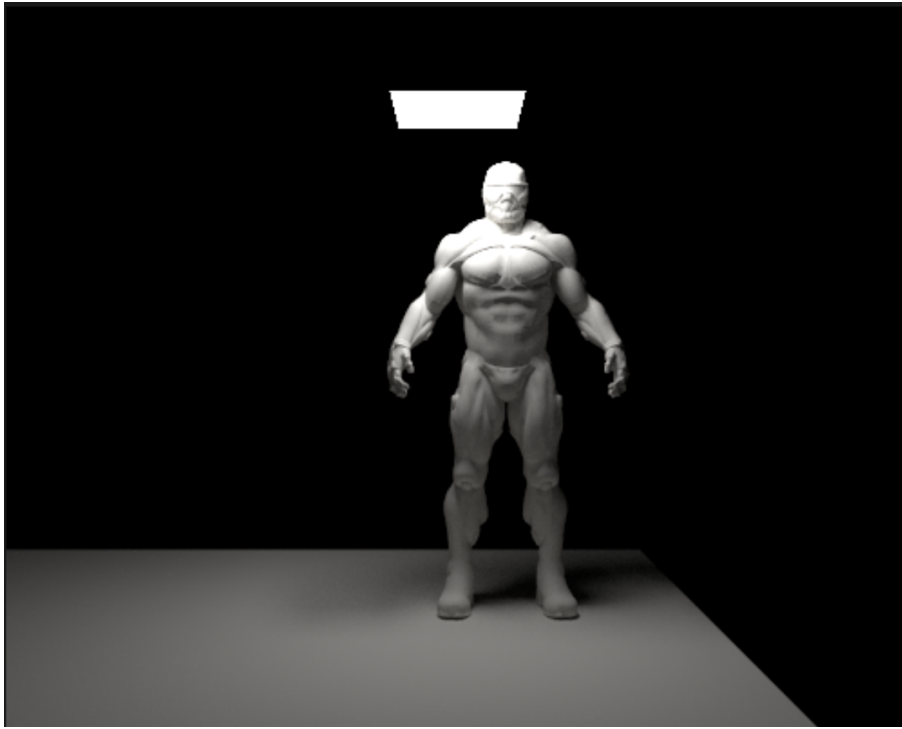
并用下面几行代码来构造:

```

1 Transform tri_Object2WorldModel = Scale(0.3, 0.3, 0.3);
2 ModelLoad ML; ML.loadModel("模型路径", tri_Object2WorldModel);
3 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：

```

1  inline std::shared_ptr<Material> getDiffuseMaterial(std::string filename) {
2      std::unique_ptr<TextureMapping2D> map = std::make_unique<UVMapping2D>(1.
3          f, 1.f, 0.f, 0.f);
4      ImageWrap wrapMode = ImageWrap::Repeat;
5      bool trilerp = false;
6      float maxAniso = 8.f;
7      float scale = 1.f;
8      bool gamma = false;
9      std::shared_ptr<Texture<Spectrum>> Kt = std::make_shared<ImageTexture<
10         RGBSpectrum, Spectrum>>(std::move(map), filename, trilerp,
11         maxAniso, wrapMode, scale, gamma);
12      std::shared_ptr<Texture<float>> sigmaRed = std::make_shared<
13         ConstantTexture<float>>(0.0f);
14      std::shared_ptr<Texture<float>> bumpMap = std::make_shared<
15         ConstantTexture<float>>(0.0f);
16      return std::make_shared<MatteMaterial>(Kt, sigmaRed, bumpMap);
17  }

```

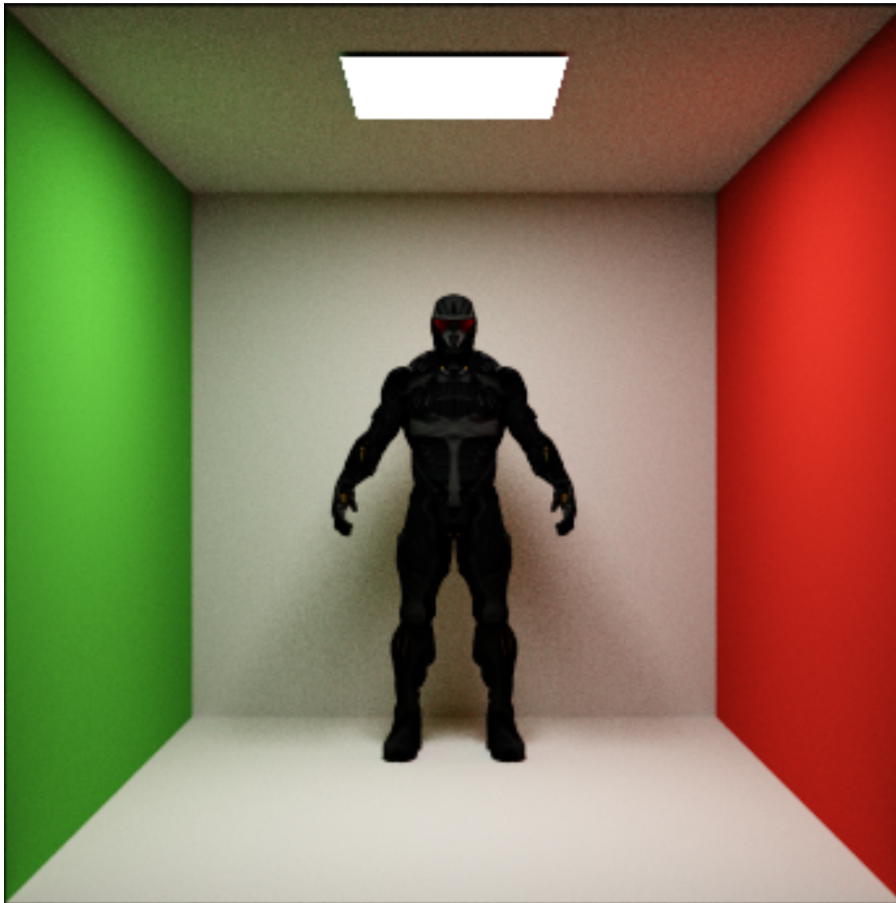
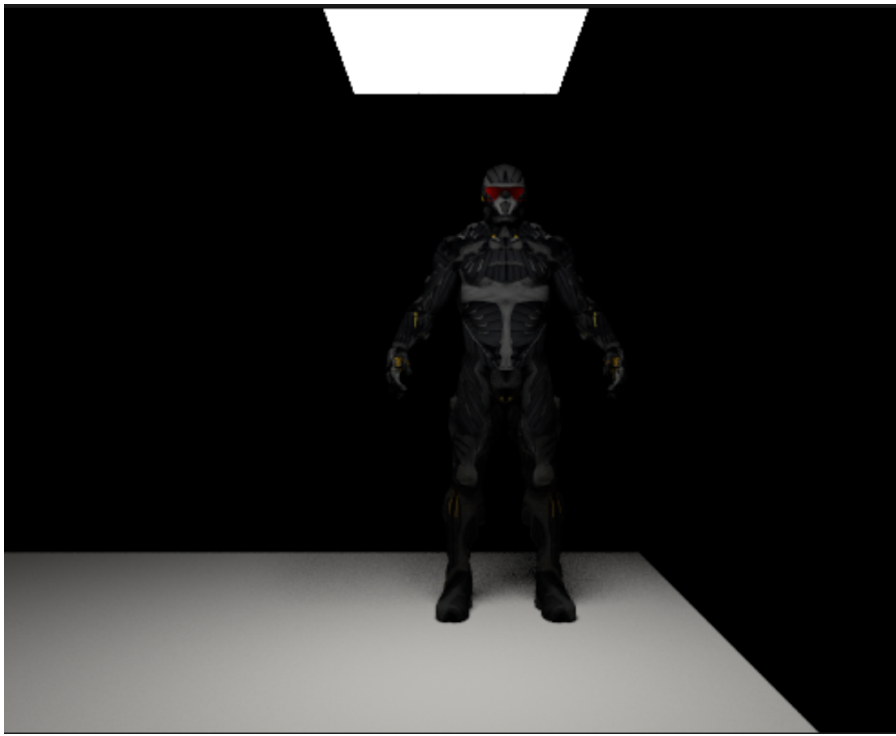
然后是包含纹理的模型构造:

```

1  void ModelLoad::buildTextureModel(Transform& tri_Object2World,
2      std::vector<std::shared_ptr<Primitive>> &prims) {
3      std::vector<std::shared_ptr<Shape>> trisObj;
4      Transform tri_World2Object = Inverse(tri_Object2World);
5      for (int i = 0; i < meshes.size(); i++) {
6          std::string filename = directory + "/" + texName[i];
7          std::shared_ptr<Material> material = getDiffuseMaterial(
8              filename);
9          for (int j = 0; j < meshes[i]->nTriangles; ++j) {
10             std::shared_ptr<TriangleMesh> meshPtr = meshes[i];
11             prims.push_back(std::make_shared<GeometricPrimitive
12                 >(
13                 std::make_shared<Triangle>(&tri_Object2World
14                     , &tri_World2Object, false, meshPtr, j,
15                     shapeID++),
16                 material, nullptr));
17         }
18     }
19 }

```

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];
```

```

3 int count = material->GetTextureCount(aiTextureType_DIFFUSE);
4 if (count == 0) {
5     diffTexName.push_back("");
6 }
7 else {
8     for (unsigned int i = 0; i < material->GetTextureCount(
9         aiTextureType_DIFFUSE); i++) {
10        aiString str;
11        material->GetTexture(aiTextureType_DIFFUSE, i, &str);
12        std::string filename = str.C_Str();
13        diffTexName.push_back(filename);
14        break;
15    }
16 }
17 count = material->GetTextureCount(aiTextureType_SPECULAR);
18 if (count == 0) {
19     specTexName.push_back("");
20 }
21 else {
22     for (unsigned int i = 0; i < material->GetTextureCount(
23         aiTextureType_SPECULAR); i++) {
24        aiString str;
25        material->GetTexture(aiTextureType_SPECULAR, i, &str);
26        std::string filename = str.C_Str();
27        specTexName.push_back(filename);
28        break;
29    }
30 }

```

获得塑料材质:

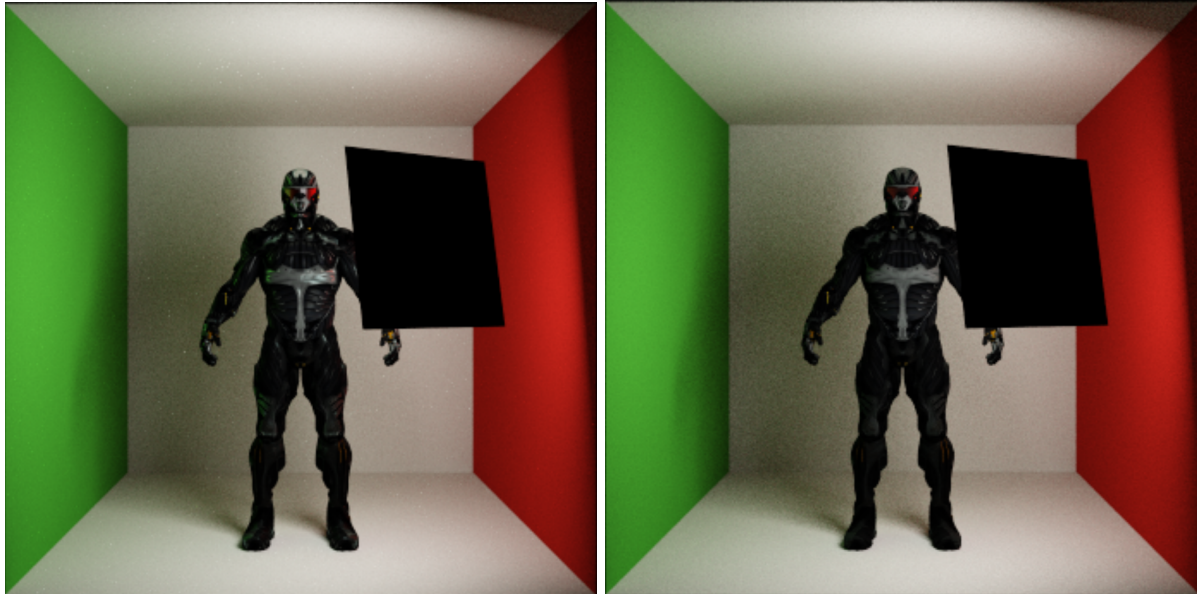
```

1 inline std::shared_ptr<Material> getPlasticMaterial(std::string diffFilename
2     , std::string specFilename) {
3     std::unique_ptr<TextureMapping2D> map1 = std::make_unique<UVMapping2D>
4         >(1.f, 1.f, 0.f, 0.f);
5     std::unique_ptr<TextureMapping2D> map2 = std::make_unique<UVMapping2D>
6         >(1.f, 1.f, 0.f, 0.f);
7     ImageWrap wrapMode = ImageWrap::Repeat;
8     bool trilerp = false;
9     float maxAniso = 8.f;
10    float scale = 1.f;
11    bool gamma = false; //如果是tga和png就是true;
12    std::shared_ptr<Texture<Spectrum>> plasticKd = std::make_shared<
13        ImageTexture<RGB Spectrum, Spectrum>>(std::move(map1), diffFilename,
14        trilerp,
15        maxAniso, wrapMode, scale, gamma);
16    std::shared_ptr<Texture<Spectrum>> plasticKr = std::make_shared<
17        ImageTexture<RGB Spectrum, 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/>