Dezeming Family

PBRT 系列 18-专业知识理论与代码实战-切线空间与凹凸贴图、透明贴图

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列电子书，可以从我们的网站 *[https://dezeming.top/]* 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

0.1 本文前言

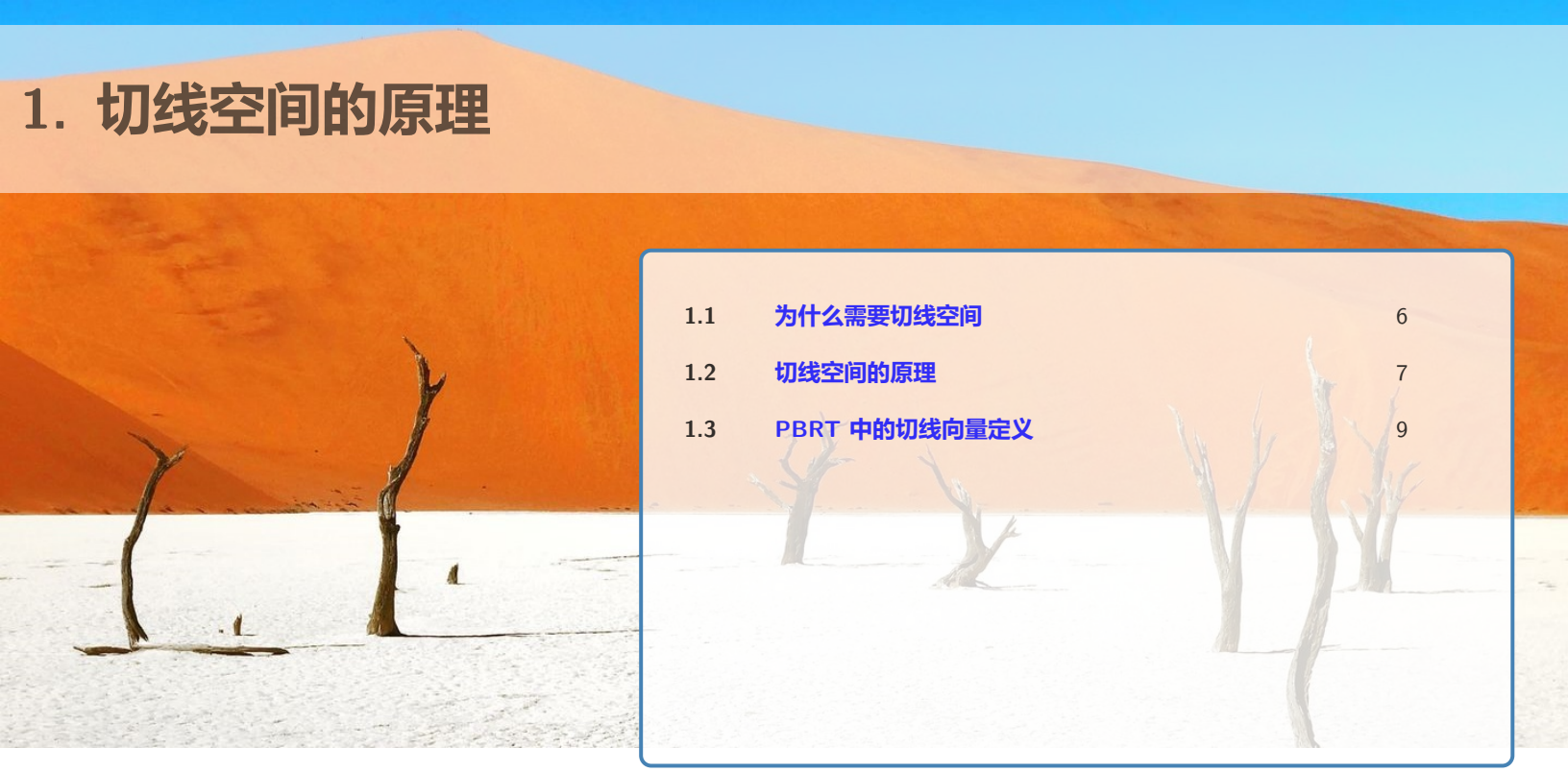
时隔半年，终于决定开始写《专业理论与代码实战》部分了。所幸写代码不像以前做嵌入式或者单片机，没过几年就会淘汰一批产品然后更新换代——对于想从零开始自己实现一个渲染系统的人来说，本系列恰如光线追踪三部曲一样，永远不会过时。尽管 PBRT 会一直更新迭代，但是从零开始写一个渲染器却是任何一个想真正成为高手的人的必经之路。

在我开始写本系列之前，我花了将近一个月的时间将基础理论系列进行了再版，主要是为每本书都提供了实现源码。不然，如果没有源码，根本没法很好地讲解这些系列，尤其是高级渲染积分器部分，有时候大家会忘记前面我们移植了什么、没有移植什么；读者自己动手如果不成功，有些时候容易造成灰心——这也是我们花大力气再版以及提供源码的原因。我们希望这个系列教程可以让更多人在渲染中更上一层楼。

我们的系列书全都是免费公开的，但由于服务器实在是昂贵（一开始我们买的是联机打游戏用的服务器，自从光子映射作为第一本电子书被放在网站上以后，DezemingFamily 才正式开始，后来我们也不想再花费精力换服务器了），所以希望觉得本系列有帮助的读者们能够给予一定的支持。本系列全套售价为 120 元——当然，哪怕是赞助 5 块钱，我们都会很开心——免费公开的内容如果大家愿意支持，说明我们的系列书得到了认可，我们自然是非常开心的。

本书内容紧跟 PBRT 系列 16，PBRT17 是关于物理材质的，没有源码上实现些什么（我们在基础篇已经移植和使用过微表面材质，但是没有详细讲解原理，PBRT17 主要是原理讲解，所以没有新需要实现的代码）；而本书理论部分较少，主要是代码的移植和应用。

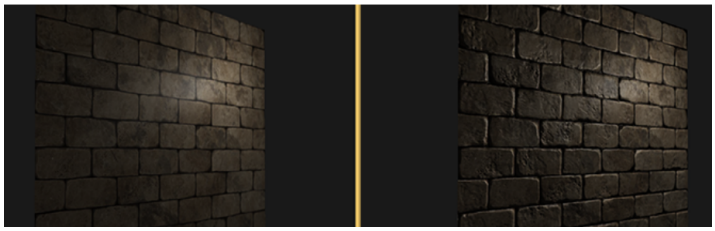
1. 切线空间的原理



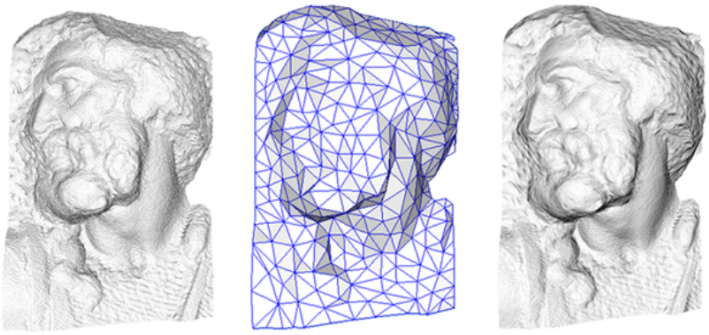
由于 *LearningOpenGL*[2] 有非常好的切线空间的介绍，所以我们按照 *LearningOpenGL* 上的内容来讲解基本原理。

1.1 为什么需要切线空间

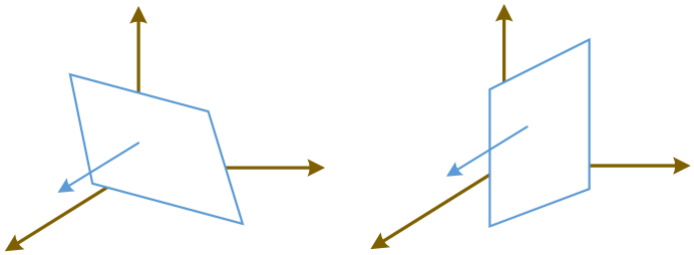
我们怎样显示出一个凹凸不平的墙面？一面光滑的墙可以由两个三角形面组成，我们当然可以构造凹凸不平的墙壁表面，但是这样会多出一大堆三角面。如果我们不构造凹凸不平的表面，而是对墙壁的表面重新定义每个细节位置的法向量（作为一个纹理），就可以得到类似凹凸不平的效果。下图都是采用的两个三角形构造的一面墙，而左边是完全平面的状态，右边是使用了法向量贴图的状态：



由此说明，法向量贴图可以很好地改变表面细节。有时候，我们简化表面结构，再使用法向量贴图，可以得到几乎与原来表面结构相似的呈现效果，比如下图，左边是四百万个三角面构成的模型；中间是 500 个三角形面构成的模型；右边是 500 个三角形面构成的模型，但是使用了法向量贴图：



而怎么定义表面的法向量贴图呢？一张图片的 RGB 三通道正好可以每个点法向量的方向（RGB 中 [0-1] 范围表示法向量的 [-1,1] 范围，取半即可），但有一个潜在问题，模型是会旋转的，如果我们一开始的法向量图中某个点朝向 z 轴，模型在沿着 y 轴转了 90 度以后，本来应该朝向 x 轴的，但贴图不知道你的模型围绕 y 轴转了 90 度，此时就会出现问题：

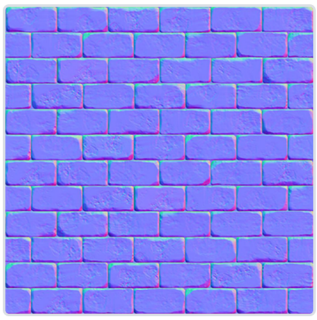


上图中，我们本来希望与 z 轴垂直的平面法向量贴图中的值指向 z 轴，但是平面旋转了 90 度以后，这个法向量贴图表示的法向量仍然指向 z 轴，这是我们不希望的。

当然我们可以这么做：记录模型的起始朝向，然后为这个起始朝向定义一个法向量贴图，然后当物体运动时，对法向量贴图得到的值也做同样的变换——但这样除了必须要记录起始位置，还有一个不好的地方：如果你的物体包含了大量重复的结构，比如一个立方体盒子有六个本来要使用同样纹理的面，那么就要为这六个面分别创建六张朝向不同方向的法向量贴图，岂不是非常浪费资源？因此，切线空间就诞生了。

1.2 切线空间的原理

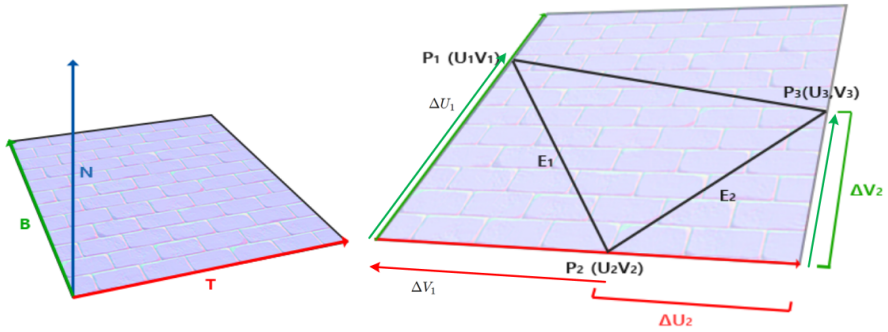
在切线空间中，法向量永远指向 z 轴正方向（会有不同程度的扰动），因此，我们看到的所有法向量贴图都是偏蓝色的：



我们希望能够将定义在切线空间的法向量给变换到模型上去，因为我们知道我们的模型中某

个面片的实际朝向，因此根据面片实际朝向，将切线空间的法向量变换过去，就能得到扰动以后的法向量值了！

我们先告诉读者我们需要什么：我们只需要三角形的三个顶点位置，以及三个顶点的纹理坐标，我们就能够计算出法线贴图如何变换到真实世界空间中去了。变换其实就是一个矩阵，我们称之为 TBN 矩阵（切线 tangent、副切线 bitangent 和 normal 三个向量），关于 TBN 可以参考下图左：



要想根据法向量来计算出切线和副切线并不是非常容易，参考上图右，我们列一个公式：

$$\begin{aligned} E_1 &= \Delta U_1 T + \Delta V_1 B \\ E_2 &= \Delta U_2 T + \Delta V_2 B \end{aligned} \tag{1.2.1}$$

展开以后就可以写为：

$$\begin{aligned} (E_{1x}, E_{1y}, E_{1z}) &= (U_1 - U_2)(T_x, T_y, T_z) + (V_1 - V_2)(B_x, B_y, B_z) \\ (E_{2x}, E_{2y}, E_{2z}) &= (U_3 - U_2)(T_x, T_y, T_z) + (V_3 - V_2)(B_x, B_y, B_z) \end{aligned}$$

可以写为一个矩阵的形式：

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} (U_1 - U_2) & (V_1 - V_2) \\ (U_3 - U_2) & (V_3 - V_2) \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} \tag{1.2.2}$$

通过矩阵求逆，就能得到切线和副切线：

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} (U_1 - U_2) & (V_1 - V_2) \\ (U_3 - U_2) & (V_3 - V_2) \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} \tag{1.2.3}$$

当然，还有一个问题需要思考，对于模型网格，往往有很多共享顶点，顶点的法向量通常是切线向量的平均化，也就是说一个面的三个顶点的法向量不是只想同样的方向，它们会受到相邻表面的法向量的影响，此时的 TBN 矩阵可能很难定义。再者，切线向量有时候也是会在不同的表面中进行平均化，使得它与表面法向量 N 不再垂直，这种情况下我们可以通过施密特正交化 (Gram-Schmidt process) 来重新让切线向量垂直于法向量。我们平时在处理时，只要是根据三角形表面的三个顶点和纹理坐标计算出的切线和副切线，就不会出现上述两种问题。

在使用 assimp 库的时候，可以选择为顶点计算出柔和的切线和副切线：

```
1  const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate
       | aiProcess_FlipUVs | aiProcess_CalcTangentSpace);
```

应用 TBN 矩阵，就是先将 t、n、s 变换到世界空间，然后这三个向量就可以构成一个 3X3 的正交矩阵，也就是 TBN 矩阵：

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} \tag{1.2.4}$$

1.3 PBRT 中的切线向量定义

本节内容看似杂乱，但它本质上只牵扯了两个类：SurfaceInteraction 类和 BSDF 类，这里面都有关于切线的定义。在我们目前移植的程序里，三角形求交中我们会对 SurfaceInteraction 类中的相关变量进行赋值，然后在材质计算中通过 SurfaceInteraction 类中的值去初始化 BSDF 相关变量的值。

如果你还对三角形求交有那么一点点印象，应该还记得这里有两个变量：Normal3f dndu, dndv; 它们就是切线向量和副切线向量。

我们可以看 Triangle::Intersect 中的代码：

```
  // 如果模型自身提供了法向量或者切线向量
if (mesh->n || mesh->s) {
  if (mesh->n){
    // 计算着色法向量 ns
    ......
  }
  if (mesh->s){
    // 计算着色切线向量 ss
    ......
  }
  // 计算ts
  if (mesh->n){
    // 计算三角形的着色几何切线和副切线向量 dndu dndv
    ......
  }
  isect->SetShadingGeometry(ss, ts, dndu, dndv, true);
}
```

在三角形求交中，dpdu 和 dpdv 就是根据前面所述的方法求得的：

```
// 注意2X2矩阵中通过行列式和伴随矩阵的方法求逆会比较快
float determinant = duv02[0] * duv12[1] - duv02[1] * duv12[0];
bool degenerateUV = std::abs(determinant) < 1e-8;
if (!degenerateUV) {
  float invdet = 1 / determinant;
```

```
  dpdu = (duv12[1] * dp02 - duv02[1] * dp12) * invdet;
  dpdv = (-duv12[0] * dp02 + duv02[0] * dp12) * invdet;
}
```

注意后面还有 dndu 和 dndv，这是着色切线向量和着色副切线向量，如果模型提供了顶点法向量，就根据这个法向量进行重新计算得到的：

```
float determinant = duv02[0] * duv12[1] - duv02[1] * duv12[0];
float invDet = 1 / determinant;
dndu = (duv12[1] * dn1 - duv02[1] * dn2) * invDet;
dndv = (-duv12[0] * dn1 + duv02[0] * dn2) * invDet;
```

在三角形求交函数里，如果没有预先定义 Mesh 的切线向量，但是预定义了顶点法向量，那么 ss 就是 isect->dpdu。

注意求交代码里有些长得很像的变量：

```
// 两顶点的差值
Vector3f dp02 = p0 - p2, dp12 = p1 - p2;
// 三角形表面偏微分，用于计算光线微分
Vector3f dpdu, dpdv;
```

BSDF 的 WorldToLocal 函数利用了切线空间，将世界空间的向量（比如光线射入的向量 wi 和射出的向量 wo）变换到切线空间，然后计算采样 BSDF 的方向。

```
Vector3f WorldToLocal(const Vector3f &v) const {
  return Vector3f(Dot(v, ss), Dot(v, ts), Dot(v, ns));
}
```

注意在 BSDF 初始化时，切线和法向量是这么设置的：

```
BSDF(const SurfaceInteraction &si, float eta = 1)
  : eta(eta),
  ns(si.shading.n),
  ng(si.n),
  ss(Normalize(si.shading.dpdu)),
  ts(Cross(ns, ss)) {}
```

SurfaceInteraction 的初始化中，在三角形求交可以看出，dndu、dndv、shading.dndu、shading.dndv 都被初始化为了 0 向量（见下面程序的两个 Normal3f(0, 0, 0)）：

```
// Fill in _SurfaceInteraction_ from triangle hit
*isect = SurfaceInteraction(pHit, pError, uvHit, -ray.d, dpdu, dpdv,
  Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time, this, faceIndex);
```

所以说，如果模型 mesh 里没有预先定义好的法向量或者切线向量，SurfaceInteraction.shading.dndu 和.dndv 都将是 0 向量。在 BSDF 的计算中，显然我们不能直接使用 dndu 和 dndv，因为大部分

模型读取的 Mesh 都没有预设的切线向量，因此，采用 shading.dpdu 来初始化 ss，以及使用 ns 和 ss 的叉积来初始化 ts。

可以看到，关于切线等内容我们目前并没有去使用，只是进行了定义。PBRT 中并没有实现与法向量贴图有关的内容，但我们自己实现一个并不难：我们已经有了切线、副切线，可以轻松地构造一个 TBN 矩阵，然后用这个矩阵去乘以法向量贴图中采样的纹理值，得到新的法向量，在 OpenGL 里的描述是这样：

```
1 normal = texture(normalMap, fs_in.TexCoords).rgb;
2 normal = normalize(normal * 2.0 - 1.0);
3 normal = normalize(fs_in.TBN * normal);
```

为了我们程序不至于太复杂，我们给出的源码里没有自己去实现法向量贴图，大家可以作为一项作业，用来熟悉 PBRT 的编程结构。之所以用一个章节去描述它，是因为我觉得无论是实时应用还是离线渲染，法向量贴图都是非常重要的技术，它可以非常好地提升细节。不过，使用贴图的提升细节的方式并不只有法向量贴图，凹凸贴图也是一项非常重要的技术，但在此之前，我打算先讲解一下透明 Mask 纹理。

2. 透明度 Mask 纹理

本章讲解透明贴图（透明纹理）的意义与 *PBRT* 的实现方法。本章暂时先不将 *PBRT* 的内容移植到我们自己的系统上，因为在下一章，我们会介绍我们自己开发一个小工具，用来把 *assimp* 读取的模型设置为我们自己喜欢的结构，这样就能使得读取和显示变得更灵活了。

2.1 为什么需要透明 Mask 纹理

考虑你有一根上面很多树叶的树枝，像这样：

那么应该怎么表示一片不规则的树叶呢？可能需要大量三角面片来构成。但是对于一棵树，上万片树叶，如果我们需要一些长得不太一样的树叶，岂不是一棵树就要上千万的面片？

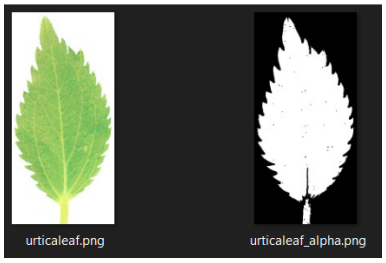
那么现在我们换一种思路，将整个叶片所在的正方形作为一个 Shape，当采样 Ray 打到纹理透明的区域时，就直接穿透过去，这样不但简单，也极大地减少了面片的数量，何乐何不为呢！

2.2 PBRT 中的透明 Mask 的应用细节

我们当前的渲染器加载 Tree 模型以后显示效果为下图右，使用微软自带的 obj 文件阅读器打开显示为下图左：

我们现在也很难分辨模型读取是否有问题（我觉得似乎纹理对应错误），还是要先实现透明度 Mask 再看看模型读取是否正确吧。注意在 PBRT 中，图像使用 MipMap 管理，也就是说透明度 Mask 也是 MipMap 管理的，这样会产生插值，因此有些边缘位置的读取值会介于 (0 – 1) 之间的半透明物体（类似于半透明的薄玻璃），但在透明度判断时，PBRT 把半透明和完全不透明都算作完全不透明。

我们以 [pbrt-v3-scenes/ecosys] 场景为例，该场景中有大量的花草树木：



这里一片树叶的不透明度被单独作为了一张纹理图像，而不是作为一张 RGB 图的第四个通道。再看.pbrt 文件的场景定义：

```
Texture "urtica-2" "color" "imagemap"
        "string filename" [ "textures/urticaleaf.png" ]
Texture "urtica-2-alpha" "float" "imagemap"
        "string filename" [ "textures/urticaleaf_alpha.png" ]
```

我们现在遇到了很多困难——我们确实可以修改一下模型文件读取系统，将额外的 Mask 加载进来，但是这样会使得模型系统变得混乱，而网上大多数模型却都是.obj 或者.blend 的，需要依赖 assimp 进行读取。这也是为什么一开始我在基础篇不得不舍弃透明度 Mask 的原因——它实在是太麻烦了。下一章我们介绍我们自己实现的模型转换工具，我们写一个小程序，将模型自动转换为我们更喜欢的格式，然后再写一个模型读取器，用来读取我们自己定义的模型。

2.3 PBRT 中的透明 Mask 代码流程

在 Triangle 的 Intersect 函数中，我们可以看到与纹理 Mask 有关的内容：

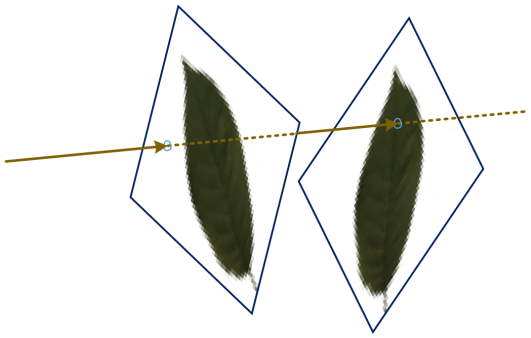
```
// Test intersection against alpha texture, if present
if (testAlphaTexture && mesh->alphaMask) {
  SurfaceInteraction isectLocal(pHit, Vector3f(0, 0, 0), uvHit, -ray.d
      ,
    dpdu, dpdv, Normal3f(0, 0, 0),
    Normal3f(0, 0, 0), ray.time, this);
  if (mesh->alphaMask->Evaluate(isectLocal) == 0) return false;
}
```

在 Triangle 的 IntersectP 函数中，不但会判断 alphaMask，还会判断阴影 Mask：

```
if (testAlphaTexture && (mesh->alphaMask || mesh->shadowAlphaMask)) {
  ......
  if (mesh->alphaMask && mesh->alphaMask->Evaluate(isectLocal) == 0)
    return false;
  if (mesh->shadowAlphaMask && mesh->shadowAlphaMask->Evaluate(
      isectLocal) == 0)
    return false;
}
```

对于阴影 Mask，可能有些时候为了某些特效或者计算加快，就没必要搞得跟 alphaMask 一样精细，但很多时候我们都只有 alphaMask，因此不需要过多关注阴影 Mask。

如果 Ray 与某个三角形相交后打中的位置在 alpha 为 0 的位置时，则判断 Ray 没有与该三角形相交；否则就与该三角形相交了。Ray 可能没有与当前三角形相交，但是与后面的三角形相交：



因此，使用 alphaMask 仅仅会对求交代码有点影响，但并不会影响到渲染程序的其他内容，所以结构还算简单——只是对于一个复杂的模型，我们应该怎么应用到 PBRT 的 alphaMask 中确实是需要考虑的。大型游戏公司都会有自己专门的模型存储与读写工具，PBRT 也有一套自己的模型读写工具，我并不打算使用 PBRT 的工具，而是使用更流行的 assimp 来制作一个模型读写工具，下一章我们介绍我们的简单清晰的模型工具 Rattler。

3. 模型转换小工具 Rattler

本章介绍我们自己实现的一个模型转换和读取小工具——*Rattler*。这是一个基于 *assimp* 实现的模型读取器，用来生成我们更容易操作的模型文件，方便应用于我们的 *PBRT* 系统。

3.1 基础功能

在 Rattler 的 Core 文件夹下是移植的 PBRT 工具类。3rdLib 目录下是第三方工具，里面有 assimp 和 openGL 库，但我们其实目前并没有用到 OpenGL 相关的东西。

Main 目录下，File 和 ImageProcess 是用于图像文件拷贝、将透明纹理提取 Alpha 通道的。

ModelLoad 类就是处理模型的类，调用 loadModel() 函数，设置模型位置以及输出目录，然后调用 processNode() 递归地处理每个节点，在每个节点里，调用 processMesh() 处理节点中的 Mesh。

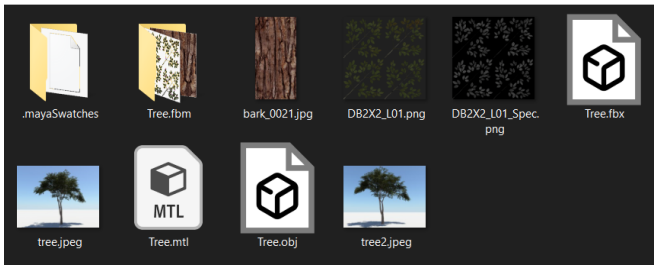
processMesh() 函数的注释写得已经非常详细了，而且代码通俗易懂。注意我们读取了 Mesh 中的最大顶点坐标和最小顶点坐标，这两个值在我们设置模型变换矩阵时会有帮助（比如我们发现这个模型坐标范围特别大或者过于小，就用 Scale 函数将其放缩一下）。

3.2 程序测试

程序执行：

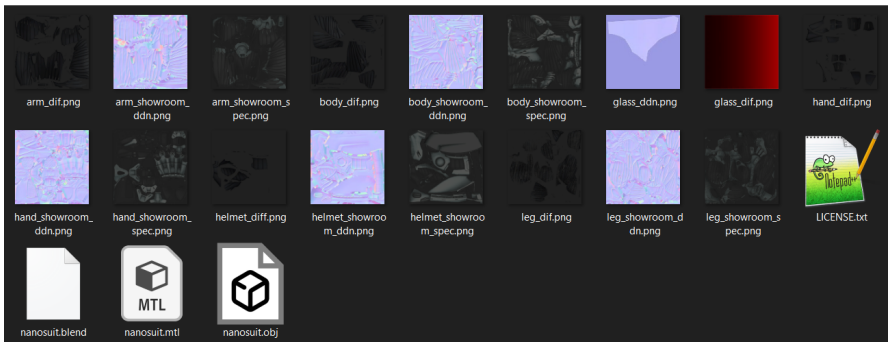
```
1 Rattler::ModelLoad model;
2 model.loadModel("./Resources/Tree/Tree.obj", "./Resources/Output");
```

对于 Tree.obj 来说，我们最初的模型文件是这样：

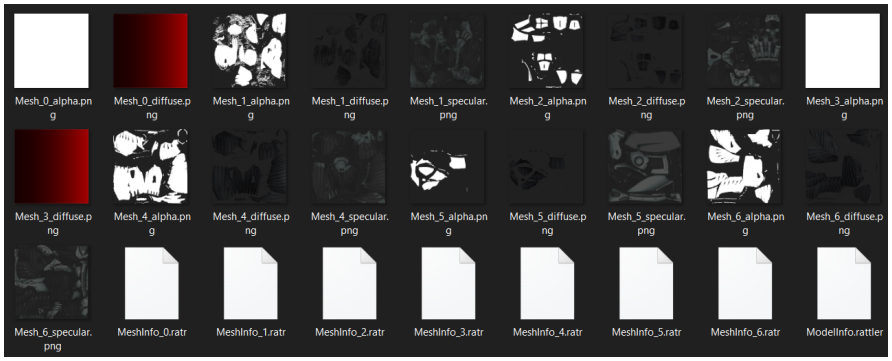


生成的我们的模型文件夹是：

对于 nanosuit 模型，我们最初的模型文件是这样：



生成的我们的模型文件夹是这样：



注意由于 PBRT 中并不支持法向量贴图，所以我们并没有将法向量贴图导出（但是这并不难，可以参考 LearnOpenGL 上关于如何读入模型法向量贴图相关的介绍）。

3.3 应用 Rattler 制作的模型

在 PBRT 工程里，我们在 Shape 文件夹下定义了 RattlerLoad 类，该类输入一个模型的文件夹，然后它会自动解析出整个模型。

简单起见，我们将只有 diffuse map 的模型部分定义为漫反射材质，将包含有 specular map 的模型部分定义为塑料材质。其实大家可以按照自己的意愿去修改模型文件和程序，去定义更多更精美的材质。

RattlerLoad 类的使用很简单：

```
Feimos::Transform tri_Object2WorldModel;
Feimos::RattlerLoad rattler("./Resources/Tree-Rattler/",
    tri_Object2WorldModel, prims, noMedium);
```

我们现在添加 Alpha 纹理，代码里有些需要修改的地方。在 TriangleMesh 类中加入两个变量：

```
std::shared_ptr<Texture<float>> alphaMask, shadowAlphaMask;
```

之后 TriangleMesh 的构造函数也需要相应地修改。另外，Triangle 的 Intersect 和 IntersectP 函数都需要添加一段代码，用于支持 AlphaMask。

我们把前面所有的构造 TriangleMesh 的函数里的倒数第二和倒数第三个参数都设置为 nullptr（表示没有 Alpha 纹理），设置完以后，能编译通过，说明是正确的。

这里强调一句，有的模型在加载纹理时需要翻转图像，在 loadImage 函数中：

```
  stbi_set_flip_vertically_on_load(true);
```

但是有的不需要，则把上面的参数改为 false。这个与模型自身的读取有关，我们要么手动把模型文件修改一下，要么就把程序进行设置。

构建 float 纹理的程序为 getAlphaMaskTexture() 函数。我们在 RattlerLoad 的构造函数中将获取到的 alphaMask 纹理赋给 TriangleMesh 对象，渲染结果为：

这个白边的原因是来自于纹理插值，原图中，透明区域的 RGB 值都是 1.0，而透明区域和非透明区域的边界处经过插值，值可能大于 0.0，因此就会形成一条白边。我们改一下判断是否透明跳过的程序即可：

```
// Test intersection against alpha texture, if present
if (testAlphaTexture && mesh->alphaMask) {
  SurfaceInteraction isectLocal(pHit, Vector3f(0, 0, 0), uvHit, -ray.d
      ,
    dpdu, dpdv, Normal3f(0, 0, 0),
    Normal3f(0, 0, 0), ray.time, this);
  if (mesh->alphaMask->Evaluate(isectLocal) < 0.99) return false;
}
```

alpha 插值以后，小于 0.99 的区域全都认为是透明的。此时渲染结果就正常了：

4. 凹凸贴图原理与实现

本章介绍凹凸贴图的基本原理与 *PBRT* 中的实现方案。

4.1 凹凸贴图的原理

除了法向量贴图，也有很多其他贴图技术来提升效果，例如位移贴图 (Displacement Mapping，也被翻译为置换贴图)。位移贴图有很多种类，比如视差贴图 (Parallax Mapping)，在游戏中有时候也叫“凹凸贴图”(Bump Mapping)，但也有些时候不这么叫，因为位移贴图会改变顶点位置，而有些时候凹凸贴图只是增加一些计算量，但不会改变顶点位置。在游戏领域，这些技术都被广泛使用，不同的游戏引擎也有不同的支持。我们以 PBRT 实现的内容为准。

Bump Mapping 见 PBRT 书 [1] 的 9.3 节。在这本书里，顶点需要计算偏移，计算方式是原来的点的位置加上一个朝向法向量方向的偏差，但是这些工作都是在着色计算中完成的，而不会影响光线的反弹等。

根据贴图位移后的顶点 $p'(u,v)$ 为：

$$p'(u,v) = p(u,v) + d(u,v)\mathbf{n}(u,v) \tag{4.1.1}$$

我们还需要计算偏微分，用于切线空间的着色。偏导为：

$$\frac{\partial p'(u,v)}{\partial u} = \frac{\partial p(u,v)}{\partial u} + \frac{\partial d(u,v)}{\partial u}\mathbf{n}(u,v) + d(u,v)\frac{\partial \mathbf{n}(u,v)}{\partial u}$$
$$\frac{\partial p'(u,v)}{\partial v} = \frac{\partial p(u,v)}{\partial v} + \frac{\partial d(u,v)}{\partial v}\mathbf{n}(u,v) + d(u,v)\frac{\partial \mathbf{n}(u,v)}{\partial v}$$

上式中除了 $\frac{\partial d(u,v)}{\partial u}$ 和 $\frac{\partial d(u,v)}{\partial v}$ 以外，其他都是已知的（dpdu、dpdv、dndu、dndv 都能计算得到，$d(u,v)$ 是从纹理中获取的），这两个未知量可以近似方法计算出：

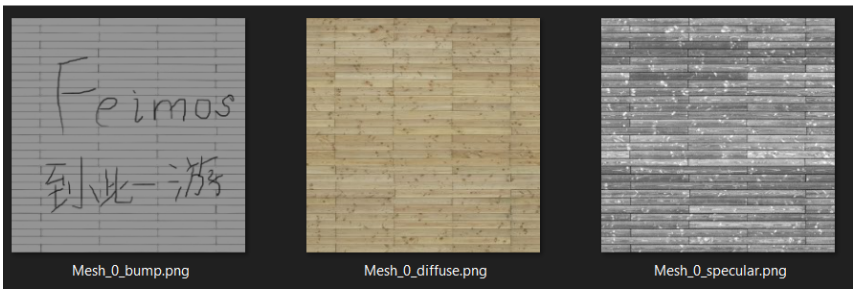
$$\frac{\partial d(u,v)}{\partial u} \approx \frac{d(u+\Delta_u,v) - d(u,v)}{\Delta_u}$$
$$\frac{\partial d(u,v)}{\partial v} \approx \frac{d(u,v+\Delta_v) - d(u,v)}{\Delta_v}$$

4.2 PBRT 中的实现

在大部分材质类中都会检测是否有凹凸贴图，如果有就进行偏移计算：

```
1 // Perform bump mapping with _bumpMap_, if present
2 if (bumpMap) Bump(bumpMap, si);
```

Bump 函数会计算偏移和微分量。目前我手边有包含凹凸纹理材质的模型种类并不多，制作复杂模型又比较麻烦，大家如果有需要可以尝试自己转换或制作一下有凹凸纹理贴图的模型，虽然代码比较容易，但可能要在模型文件的转换或制作中费点功夫。我实现的 Bump 纹理贴图如下：



就是一个板子，我在原有的 Bump 贴图基础上“刻”了几个字（现实世界中一般是不允许这样做的），使用了凹凸贴图的效果如下：

好像在这种情况下并没有明显改观，这是因为 PBRT 的顶点并不会真正根据贴图来唯一，位移只是用于计算着色。

本文完稿于 2022 年 7 月 21 日，总共用了三四天时间，期间还制作了 Rattler 工具。本书内容相比后面其他的书算得上是最简单的一本了，基本没有复杂的理论和概念，都是很直观形象的东西，即使读者对微分量等不太熟悉也没有关系，这些并不影响我们后续内容的学习。下一本书我们的目标就是“种草”，即通过实例化，实现大量重复性物体的快速渲染。同时，我们还将实现运动模糊的效果。

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/