PBRT 文件加载和场景解析

Dezeming Family

2020 年 11 月 19 日

因为本小书是电子书，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，最好从官方网站：https://dezeming.top/ 下载新的版本（免费下载）。

本书目标：学习 PBRT 场景文件的解读，明白代码中场景文件如何加载到 PBRT 系统里并初始化里面的各个类和对象，以及各个类的基本作用。掌握 PBRT 中比较高级的 C++ 语言语法。明白 PBRT 是如何开始启动渲染的。

本文于 2022 年 7 月 3 日进行再版。本文增加了对 curTransform 的详细描述，以及修改了很多内容的叙述方式。注：本文只是基本概念的了解，没有附加代码。

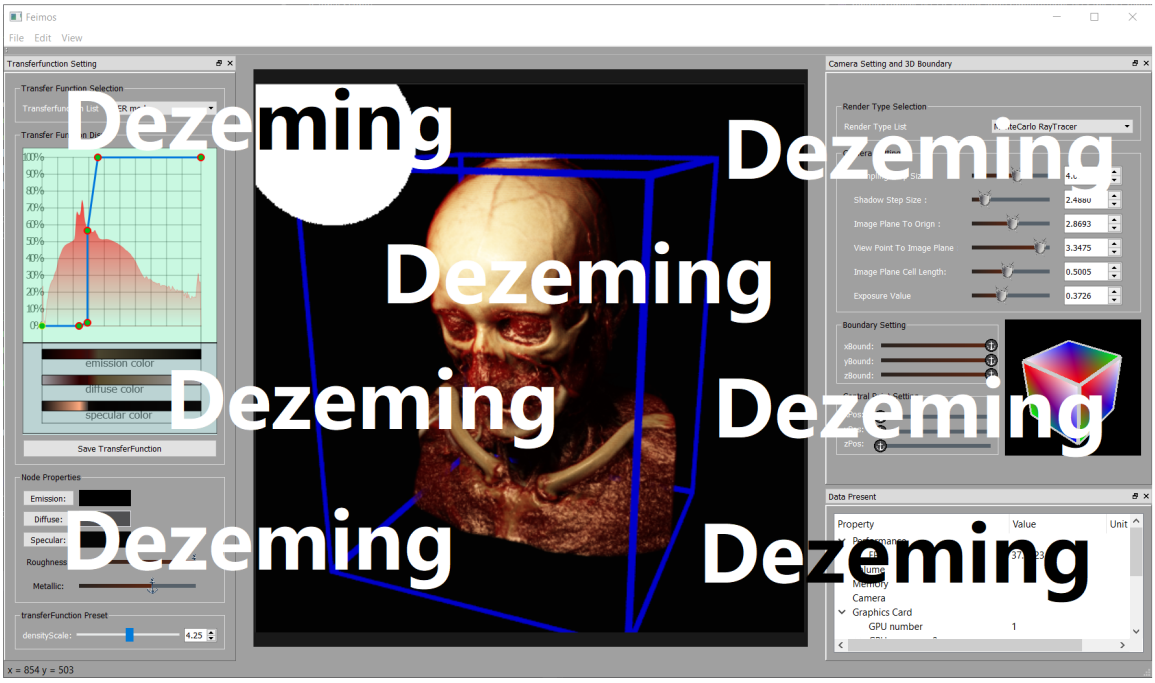
前言

PBRT[4] 作为一个比较完备和轻量级引擎，里面有很多细节和实现步骤我们都很难短时间掌握。因此，我决定写一系列的小书，来从一个简易的光线追踪器到 PBRT 的实现。这个系列的小书会包含对 PBRT 渲染器从细节原理到一步步移植。

光线追踪最好的入门系列之一是光追三部曲 [1][2][3]，这三本书可以让你短时间内就能拥有一个自己编写的光线追踪引擎，但是其完备性和鲁棒性都不够，可以用来渲染的场景也有限。而即使这三本小书学透了，再去学习 PBRT 的源码也会面临诸多困难，而 PBRT[4] 这本书的讲解结构也是非常松散的，不利于马上动手去实现。因此，我们的任务目标是在 [1][2][3] 的基础上，从整体到细节去把握 PBRT，直到把 PBRT 应用到我们自己的光线追踪引擎上去。

对于计算机图形学的过于基础的知识，在虎书《计算机图形学基础》[5] 中有讲解，比如透视投影和三维空间变换，这些内容我不会详细讲解。

之所以选择的高级引擎书籍是 PBRT，因为其代码只有不到五万行，跟我自己写的第一个比较完整的渲染引擎，一个医学数据渲染引擎（如下图）的代码量差不多，因此我感觉能对如何介绍里面的知识相对容易把握一些。



在写这套系列书的时候，我其实是比较忐忑的。我一直在想，对于一个初学者而言，应该怎么去面对一个比较庞大而且复杂的引擎架构，以及他们想看到一本什么样的书呢。为此我咨询了不少初学者，可以说，本书虽然来自于我的笔记，但也是一边整理一边向大家求得反馈而形成的。文章是用 Latex 写的，由于时间关系，我没有使用复杂的模板和排版。在讲解中，语言也会尽量通俗（但会减弱口语化的叙述，因为第一版有些过于口语化，出现很多前言不搭后语的比较矛盾的叙述）。

PBRT 是一个轻量级的系统，但仍然里面有比较复杂的逻辑结构划分，因此，本小书作为 PBRT 解读系列的第一本书，意在能够让读者在有一定基础中快速把握整个 PBRT 系统，知道场景如何加载，以及加载到了各个基类的什么位置，程序如何启动，这样再研究该系统的话就会更轻松。本书目标：一两天就让你弄明白 PBRT 是如何工作的。

本套系列书一共分为两个部分：

第一部分为 PBRT 基本知识《基础理论与代码实战》，一共 15 本书，售价 40 元，其中每本的售价都不同。我们不直接收取任何费用，如果其中某本书对大家学习有帮助，可以往我们的支付宝账户（17853140351，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！

第二部分为专业理论知识部分《专业知识理论与代码实战》，一共 12 本书，暂定售价 120 元，每本价格也各不相同。我于 2022 年 7 月，开始着手写这个系列，但在此之前，我会将第一部分的每本书都配备

源码，然后再开始第二个部分的写作。

目录

一　场景文件解读

这一章介绍场景文件描述的信息的具体含义。

1 1　PBRT-V3 的安装

网上有很多教程，这里简单说一下关键步骤。我们使用的操作系统是 Windows 系统。

使用 GitBash 下载的命令：

```
git clone --recursive https://github.com/mmp/pbrt-v3/
```

看到工程拷贝到的地址（feimos 是我的用户名）：

```
Cloning into 'C:/Users/feimos/pbrt-v3/src/ext/glog'...
```

使用 CMake 编译时，注意 DCMAKE_BUILD_TYPE 参数，默认是 Release 模式。我们不改变里面的任何设置，因为对初学者来说，有些设置会比较陌生。等我们研究完 PBRT 以后，读者就会很清楚地明白不同的设置的含义和作用。

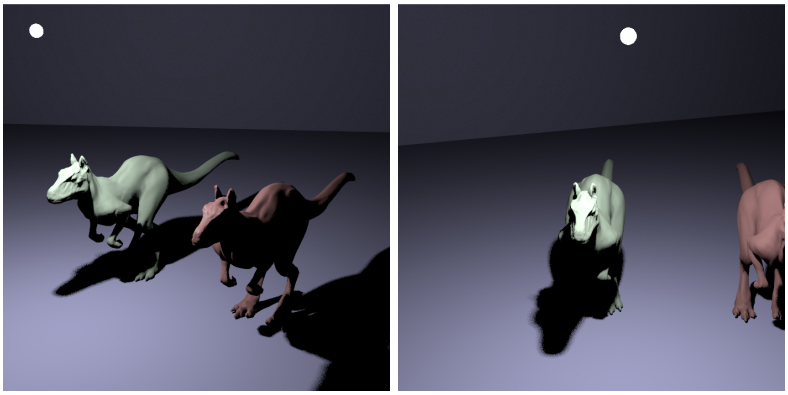
1 2　基本设置

我们解析一下 PBRT3 自带的一个关于恐龙的默认场景，在 killeroo-simple.pbrt 文件里。首先是基本设置的场景表示：

```
    LookAt 400 20 30    0 63 -110    0 0 1
    Rotate -5 0 0 1
    Camera "perspective" "float fov" [39]
    Film "image"
    "integer xresolution" [700] "integer yresolution" [700]
        "string filename" "killeroo-simple.exr"
    # zoom in by feet
    # "integer xresolution" [1500] "integer yresolution" [1500]
    #   "float cropwindow" [ .34 .49  .67 .8 ]
    Sampler "halton" "integer pixelsamples" [8]
    Integrator "path"
```

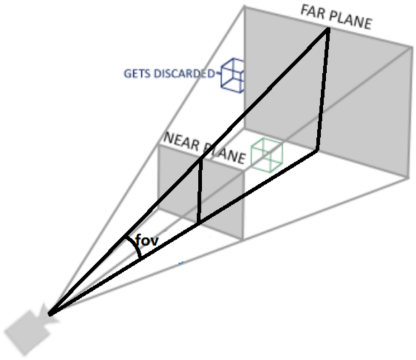
第一行是 lookAt 相机设置（计算机图形学中相机空间与世界空间的关系可以参考 [5]），前三个数构成相机位置，中间三个数构成视线方向向量，最后三个数构成 up 向量。注意这里的 up 向量的设置不是一般我们常用的 (0,1,0) 代指天空方向，而是 (0,0,1)（但很多时候我们渲染的场景都是设置 up 为 (0,1,0)）。提前提一句，一般来说，PBRT 中的渲染计算是物体先从物体坐标系变换到世界坐标系；相机发出的 Ray 从相机坐标系变换到世界坐标系，然后在世界坐标系中计算与物体包围盒的求交。

第二行 Rotate 相当于世界坐标系与相机坐标系之间的转换关系（我们会在第四章的第二节详细介绍），字面意思上就是绕 (0,0,1) 轴旋转-5 度。下图是分别转 0 度和-30 度以后的结果：



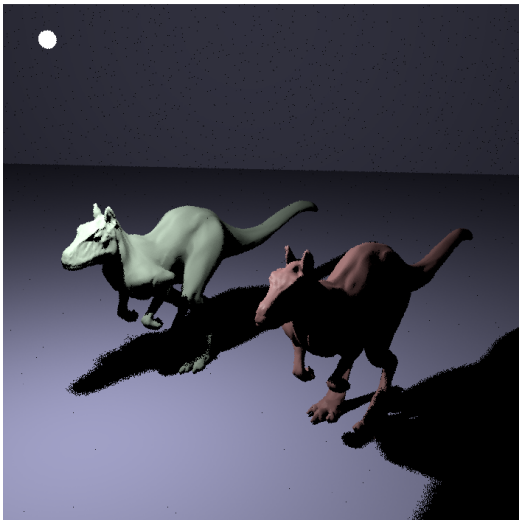
在场景加载中，外层的变换会作用到内层的变换。比如，在 WorldBegin 和 WorldEnd 内，但是在 AttributeBegin 和 AttributeEnd 之外的变换，会作用于所有的 Attribute。

第三行表示相机采用透视投影相机，注意我们可以在 PBRT 中生成多种图像，例如全景相机。相机的 fov（如下图）为 39 度。



第四行到第六行表示生成的图像是 image。注意 Film 的类型只能是 image，否则就会报错（详见 api.cpp 的 MakeFilm 函数）。分辨率是 700*700，输出文件名为 killeroo−simple.exr，这是一种高动态范围 (High-Dynamic Range, HDR) 图像文件格式，我们也可以生成 jpg 或者 png 格式的文件。

第十行表示我们使用 halton 来产生随机数序列，每个像素采样 8 次。上图是采样 8 次的结果，如果只采样一次，结果如下，可以看到影子边缘位置噪点比较多：



第十一行表示我们使用的积分渲染方程的积分器是路径追踪器（PBRT 支持多种积分器，例如随机渐进式光子映射，路径追踪器和双向路径追踪器等）。

1 3　场景设置

首先放上场景的整体概览，每个场景属性块放在了 WorldBegin 和 WorldEnd 之间：

```
1   WorldBegin
2
3   AttributeBegin
4   ......
5   AttributeEnd
6
7   AttributeBegin
8   ......
9   AttributeEnd
10
11  AttributeBegin
```

```
    ......
    AttributeEnd

    WorldEnd
```

我们的恐龙场景有三个属性设置块（Attribute），我们一个一个进行分析。

1 3.1 属性块一

```
AttributeBegin
        Translate 150 120  20
        AreaLightSource "area"  "color␣L" [2000 2000 2000] "integer␣nsamples
           " [8]
        Shape "sphere" "float␣radius" [3]
AttributeEnd
```

AttributeBegin 和 AttributeEnd 来包括这个局部场景的信息。注意属性块里面设置的变换，比如设置了平移和旋转，是不影响其他属性块内的变换的（简单提一句，在源码实现中，这非常类似于 OpenGL 中 GLUT 的变换矩阵入栈和出栈操作）。

Translate 为物体设置平移矩阵。AreaLightSource 和 Shape 设置面光源，颜色（亮度）为 2000 的白光。每次对光采样 8 个点（在一些积分渲染器中这个值不需要，比如 Whitted 光线追踪器，每次渲染只会对每个光源采样一个点）。光源形状为球形，半径为 3。

1 3.2 属性块二

```
AttributeBegin
    Material "matte" "color␣Kd" [.5 .5 .8]
    Translate 0 0 -140
    Shape "trianglemesh" "point␣P" [ -1000 -1000 0 1000 -1000 0 1000 1000 0
       -1000 1000 0 ]
    "float␣uv" [ 0 0 5 0 5 5 0 5 ]
    "integer␣indices" [ 0 1 2 2 3 0]
    Shape "trianglemesh" "point␣P" [ -400 -1000 -1000   -400 1000 -1000
       -400 1000 1000 -400 -1000 1000 ]
    "float␣uv" [ 0 0 5 0 5 5 0 5 ]
    "integer␣indices" [ 0 1 2 2 3 0]
AttributeEnd
```

该属性快描述了地面结构。Material 表示地面的材质和颜色，matte 表示粗糙的漫反射材质。

第四到第六行表示地面的形状，该形状为方形，由两个三角形面片构成，point P 中的数据每三个表示一个顶点，编号为从 0 到 3；uv 表示纹理索引，因为没有加载纹理图片，所以没有用处；integer indices 描述了顶点索引，我们有四个顶点，构成两个三角形，每个三角形三个顶点，所以顶点索引有 6 个数，每三个代表了一个三角形的三个顶点。

1 3.3 属性块三

```
AttributeBegin
    Scale .5 .5 .5
    Rotate -60 0 0 1
```

```
4     Material "plastic" "color Kd" [.4 .2 .2] "color Ks" [.5 .5 .5]
5             "float roughness" [.025]
6     Translate 100 200 -140
7     Include "geometry/killeroo.pbrt"
8     Material "plastic" "color Ks" [.3 .3 .3] "color Kd" [.4 .5 .4]
9             "float roughness" [.15]
10    Translate -200 0 0
11    Include "geometry/killeroo.pbrt"
12 AttributeEnd
```

第二和第三行表示对数据进行缩放以及旋转操作。第四五行和第八九行定义材质和颜色。注意在塑料这种材质中，Kd 表示漫反射组成部分的材质颜色，Ks 表示镜面反射组成部分的材质颜色，以及 roughness 表示粗糙度。第七和第十一行表示包含的模型文件。

1 4　小结

本章只是对场景文件做了介绍，但是文件表示的信息在 PBRT 的渲染系统里如何表示，以及场景中对应的类应该如何进行表示会在后面逐步进行介绍。

二　PBRT 场景加载必备 C++ 基础

本章介绍一下 PBRT 用到的 C++ 语法，默认读者已经掌握了 C++ 的基本语言形式，也知道怎么使用基本的标准库工具（vector, string 等）。这里主要是介绍 C++11 的一些新的语法特性。

源码（PBRT3 parser.cpp）：

```
1  void pbrtParseFile(std::string filename) {
2      if (filename != "-") SetSearchDirectory(DirectoryContaining(filename
           ));
3      auto tokError = [](const char *msg) { Error("%s", msg); exit(1); };
4      std::unique_ptr<Tokenizer> t =
5          Tokenizer::CreateFromFile(filename, tokError);
6      if (!t) return;
7      parse(std::move(t));
8  }
```

2 1　lambdas 表达式

lambdas 表达式可以在函数内定义一个函数：

```
1  auto func = [](const string &a, const string &b)
2  { return a.size() < b.size();};
3  //使用该函数
4  cout << func("as","rs")<< endl;
```

该函数的函数名为 func，有两个 string 类型的参数，a 和 b，该函数返回 a 和 b 的比较值。该函数可以赋值给 std::function 类：

```
1  std::function<bool(const string &a, const string &b)> ds = func;
2  //使用该函数
3      cout << ds("as", "rs") << endl;
```

2 2　uniqut_ptr

uniqut_ptr 是一个智能指针，在 memory 头文件下。一个对象只能被一个 uniqut_ptr 指向，然后被该指针拥有。当 uniqut_ptr 释放时，该对象也被释放。

```
1      std::unique_ptr<double> p1;
2      double *p = new double(12.678);
3      //p1 = p; 不可以，unique_ptr不能被赋值
4      //p1 = new double(1.328); 不可以，unique_ptr不能被赋值
5      std::unique_ptr<double> p2(new double(12.112));//可以
6      std::unique_ptr<double> p3(p);//可以
7      cout << *p2 << endl;
8      cout << *p3 << endl;
```

因为一个对象只能被一个 uniqut_ptr 智能指针指向，因此如果我们要把对象转移给另一个指针时，我们需要使用另一个函数，即 std::move。

```
1      std::unique_ptr<double> p4(std::move(p2)); //可以
2      cout << *p4 << endl;
```

2 3　外部变量访问方式说明符

源码（PBRT3 parser.cpp）：

```
    auto basicParamListEntrypoint = [&](
        SpectrumType spectrumType,
        std::function<void(const std::string &n, ParamSet p)> apiFunc) {
        string_view token = nextToken(TokenRequired);
        string_view dequoted = dequoteString(token);
        std::string n = toString(dequoted);
        ParamSet params =
            parseParams(nextToken, ungetToken, arena, spectrumType);
        apiFunc(n, std::move(params));
    };
```

lambda 表达式 [&] 里面的符号表示“外部变量访问方式说明符”，即如果是“=”则表示不允许定义在外面的变量被中修改，如果是“&”则表示定义在外面的变量允许在中被修改。

```
        float data = 10;
        auto func = [&data](const float &a)
        { return a + data; };
        cout << func(12) << endl;
```

其中，[&data] 构成捕获列表，即方括号里面的变量可以在函数体中使用，以及被更改。

```
        float data = 10;
    auto func = [=](const float &a)
    //不可以，因为data不能在函数体内被修改
        { data *= 2; return a + data; };
        cout << func(12) << endl;
```

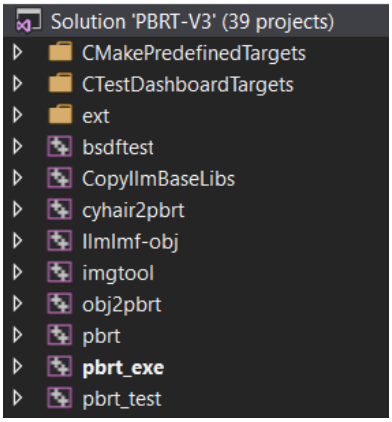
估计很多读者看不懂源码的原因就是在场景加载这里了。lambda 表达式应该是 C++11 的新标准的内容，很多老版的 C++ 书籍都没有涉及。但根据我对整个源码的研究，涉及到的 C++ 新语法的内容大概也就这些，因此读者没有必要重新去学习新版本的 C++。

三　PBRT 场景加载

这一章是理解 PBRT 场景加载系统的关键，建议对照着源码多读几遍，对整体有个比较熟练的把握。之后大家就会发现，这个所谓复杂而又“庞大”的渲染引擎其实也没有那么难懂，大家也可以自己实现一个同样优秀甚至更优秀的引擎。

本章简单的描述了每个类是如何被加载到渲染器中的，以及一些要理解该工程必须要掌握的内容，但本章并没有把所有的类的构成的渲染流程给串起来，而是分开描述，让读者对 PBRT 中的十个基类有一个比较好的了解，为下一章 PBRT 整体的启动流程做一个铺垫。

在 VS2015 中编译生成的工程目录如下：



简单介绍一下各个工程：bsdftest 是用来测试 bsdf 函数的。cyhair2pbrt 是用来把 CyHair 文件转为 PBRT 文件的。IlmImf-obj 是 Industrial Light & Magic 公司的库，里面有一些可供使用的函数。imgtool 是一些用于图像的工具。obj2pbrt 把 obj 文件转为 pbrt 可用的文件。pbrt_exe 工程就是我们可以直接来执行的程序。

pbrt 工程是核心工程，里面包含了上面的十个文件夹，以及一个 core 文件夹，基类都定义在 core 文件夹中，而每个基类的派生类都定义在各自文件夹下，比如 Camera 基类定义在 core 文件夹下，透视相机，全景相机和真实感相机各自定义在 camera 文件夹下。PBRT 的基类表示如下，右边是它们各自所在的文件夹：

class	directory
Shape	shapes/
Aggregate	accelerators/
Camera	cameras/
Sampler	samplers/
Filter	filters/
Material	materials/
Texture	textures/
Medium	media/
Light	lights/
Integrator	integrators/

本章主要内容是对 pbrt 工程进行整体上的分析。

3 1　几何和变换类

几何和变换类定义在了 geometry.h 和 geometry.cpp 文件以及 transform.h 和 transform.cpp 文件中。这里面包含了点、向量和矩阵等表示，也包含了包围盒。功能包括生成相机变换矩阵，正交、透视投影矩阵等。功能和 OpenGL 以及其他光线追踪引擎（包括简单的引擎和复杂的引擎）并没有什么本质的区别。

curTransform 表示当前的变换（api.cpp 文件），对于全局变换的表示，例如上面场景文件的基本设

置中，读到 Rotate 时，会为 curTransform 赋值（curTransform 表示物体坐标与世界坐标之间的变换）。当 parse 到 AttributeBegin 后，curTransform 进栈保存，curTransform 会被乘以其它变换；当 parse 到 AttributeEnd 后，弹栈并恢复全局的 curTransform。因此，每个物体的物体坐标-世界坐标的变换等于全局变换乘以局部变换。比如我们恐龙的场景文件里，面光源就是先平移再旋转的变换（Rotate[−5 0 0 1]*Translate[150 120 20]）。

在《PBRT 实战-形状和加速器》书中，会对变换和坐标系统进行更详细的介绍，这里我只简单提一句，大部分情况下，PBRT 中是把物体变换到世界空间坐标，然后在相机坐标系产生 Ray，把 Ray 变换到世界坐标系，然后在世界坐标系中计算 Ray 与物体的求交。

3 2 参数 ParamSet

场景加载到文件里以后，参数会被存在哪里呢？其实会被存在 ParamSet 类中。在结构 RenderOptions（api.cpp）中，有各种类的参数集，例如相机类的参数集 CameraParams，采样器类的参数集 SamplerParams。在通过 lambda 表达式生成的函数 basicParamListEntrypoint 中，会调用生成参数集 ParamSet 类的函数，因为涉及 parse 文件的语法，我们不做介绍（其实在这个语法结构并不复杂，网上也有不少资料）。这里只是提一下场景文件里的参数是被下面行代码加载到参数集内的：

```
1    ParamSet params = parseParams(nextToken, ungetToken, arena, spectrumType
         );
```

ParamSet 类表示了生成当前目标物（比如一个相机）的所有参数（在下一节相机类中会详细说明）。ParamSet 类并通过一系列的函数来定位和寻找参数，比如：

```
1    Point2f FindOnePoint2f(const std::string &, const Point2f &d) const;
```

FindOnePoint2f 通过第一个参数 string 表示的名称来寻找参数集里的 Point2f 类型的参数，并返回其值。

如上所述，所有的参数都会被最终加载到 RenderOptions 结构中，该结构包含了多个 ParamSet；该结构还有三个重要的场景生成函数：

```
1    Integrator *MakeIntegrator() const;
2    Scene *MakeScene();
3    Camera *MakeCamera() const;
```

这些函数负责使用之前从文件中读入的场景参数进行初始化，构造相机、渲染方程积分器和场景。

3 3 相机类 Camera

当解析场景文件遇到 Camera 时：

```
1 if (tok == "Camera")
2     basicParamListEntrypoint(SpectrumType::Reflectance, pbrtCamera);
```

SpectrumType 一共有两个类别，用来区分光谱类型，光谱分为两种，一种是反光物，另一种是发光物（在一些渲染积分器，比如双向路径追踪中会区分从光源和从相机发出的采样光线），我们暂时先不管它。

```
1    enum class SpectrumType { Reflectance, Illuminant };
```

pbrtCamera 就是初始化相机的程序，它的函数实体在 api.cpp 里：

```
1    void pbrtCamera(const std::string &name, const ParamSet &params)
```

params 是对应于相机参数集，里面加载了 Camera 文件里面的场景参数。basicParamListEntrypoint 函数实体：

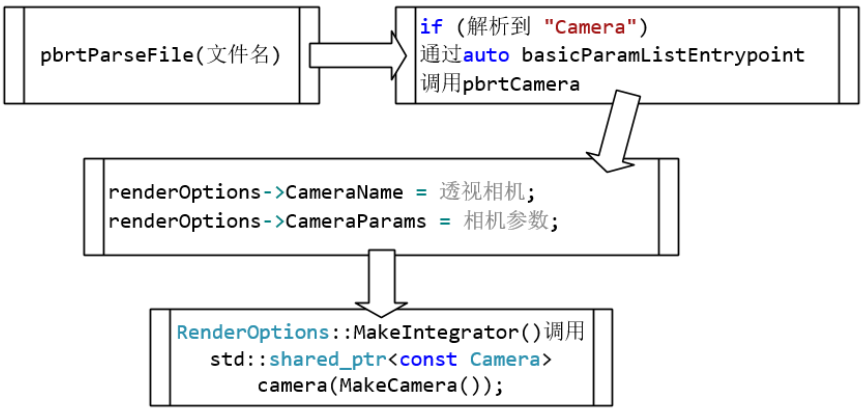
```
auto basicParamListEntrypoint = [&](
    SpectrumType spectrumType,
    std::function<void(const std::string &n, ParamSet p)> apiFunc) {
    string_view token = nextToken(TokenRequired);
    string_view dequoted = dequoteString(token);
    std::string n = toString(dequoted);
    ParamSet params =
        parseParams(nextToken, ungetToken, arena, spectrumType);
    apiFunc(n, std::move(params));
};
```

在调用相机生成时，这里的 apiFunc 就对应了 pbrtCamera，即 pbrtCamera(n, std::move(params))，其中 n 表示参数名，这里是 Camera，params 表示的是生成相机用的参数（因为是 unique 智能指针指向的，防止被不同指针指向，因此需要用 std::move 来传递，具体语法可以参考上一章）。

RenderOptions::MakeCamera()（api.cpp）调用了生成相机类的程序，输入参数为相机名称（透视相机，全景相机等）、相机参数和相机到世界坐标系的变换等信息。然后在 pbrt::MakeCamera() 函数中实现了生成相机，比如，如果是透视相机：

```
if (name == "perspective")
    camera = CreatePerspectiveCamera(paramSet, animatedCam2World, film,
                                     mediumInterface.outside);
```

之后每种相机各自调用属于自己的相机构造程序，使用相机参数来实例化相机。为了清楚起见，用一个流程图结构表示一下：



因为相机已经比较详细的描述了场景的加载，所以后面的类中就不再赘述场景是如何被加载进来的了，而是着重介绍每个基类的作用和结构。

3 4 采样器类 Sampler

在 sampler.h 和 sampler.cpp 文件中定义了采样器，以及 sampling.h 和 sampling.cpp 定义了一些采样方法。

采样器在 pbrtSampler() 函数中加载到渲染器的设置里，并通过 MakeSampler() 来构造，有 halton、sobol 等采样器，这些具体的采样器就定义在 samplers 文件夹下。

sampling.h 和 sampling.cpp 文件里主要是具体的采样方法，比如从 1 维区间内分层抽样，比如使用“接受拒绝法”从圆盘内抽样，比如从单位球面进行均匀采样等。这些函数要么输入是一个随机点，然后映射到所在空间（比如随机点映射到球面），比如下面第一行的函数；要么是一个采样器，生成样本或者样本序列，比如下面第二行的函数。

```
Vector3f UniformSampleHemisphere(const Point2f &u);
```

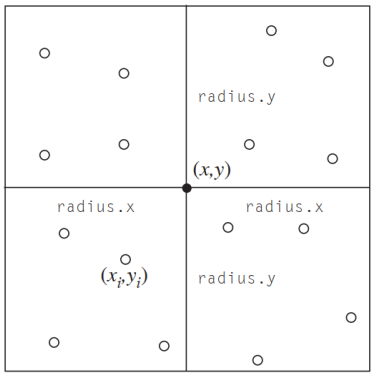
```
2    void StratifiedSample2D(Point2f *samp, int nx, int ny, RNG &rng, bool
         jitter);
```

采样器的作用是产生低差异随机序列，低差异序列的分布更均匀，更有利于渲染。

3 5 滤波器类 Filter

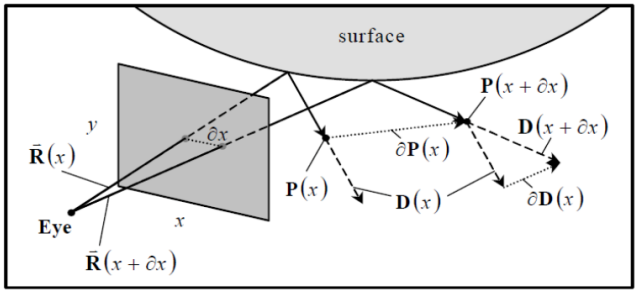
滤波器非常简单，主要就是盒滤波器，三角滤波器，sinc 滤波器等，这些滤波器用来给最终生成的图像进行滤波，以及对采样中的纹理进行滤波。通过 pbrtPixelFilter() 加载到渲染器的设置里，并通过 MakeFilter() 来构造。

图像滤波：我们生成图像时每个像素采样次数有限，因此需要进行滤波（下图来自 [4]）：



根据当前像素中心位置 (x, y) 和每个采样点位置 (x_i, y_i) 进行加权平均，得到最终像素值。不同的滤波器加权是不一样的。

纹理滤波：因为使用了光线微分算法，即每个发射的采样光线会计算一个像素的偏差 [5]，构成一个光锥，光锥与物体相交会构成一个相交面（而不只是一个交点），那么就需要在这个面内进行滤波。



关于光线微分算法这里不展开介绍，后续系列我会专门写一本小书来描述算法原理和 PBRT 实现流程。

3 6 形状类 Shape

只要学习过光线追踪引擎，就知道它里面包含了各种形状。只不过在加载时，需要考虑一些信息。形状类通过 pbrtShape() 函数将文件场景加载到渲染器中，这个函数相比前面其他类的相关加载函数复杂一些。

因为由第一章的场景文件可知，每个形状集都是包含在一对 AttributeBegin 和 AttributeEnd 之间的，因此里面的一些信息用来对形状集进行处理：比如使用的三维变换矩阵将形状变换到世界坐标系中；比如考虑是不是动态物体需要进行动态模糊；比如该形状是不是面光源。如果是运动物体还要为该形状构建运动物体的加速器结构（运动的 BVH 树）。

所有处理完以后的形状集（包括光源）会根据其类型被补充到数组 renderOptions->primitives 中，面光源还会被补充到 renderOptions->lights 中。

3 7 加速器类 Aggregate

用来构建渲染加速结构的类，从场景文件通过 pbrtAccelerator() 加载到渲染器中，并通过 MakeAccelerator() 函数来构造，可选的加速器有 bvhTree 和 kdTree 两种。

```
    if (name == "bvh")
        accel = CreateBVHAccelerator(std::move(prims), paramSet);
    else if (name == "kdtree")
        accel = CreateKdTreeAccelerator(std::move(prims), paramSet);
```

通过不同的 Creat 函数创建两种不同的加速器。

3 8 材料类 Material

默认的渲染恐龙的例子里没有很好的关于材料和纹理方面的表示，这里放另一段场景文件：

```
WorldBegin
        Texture "Texture01" "spectrum" "checkerboard" "float uscale" [
            20.000000 ] "float vscale" [ 20.000000 ] "rgb tex1" [ 0.325000
            0.310000 0.250000 ] "rgb tex2" [ 0.725000 0.710000 0.680000 ]
        MakeNamedMaterial "Floor" "string type" [ "matte" ] "texture Kd" [ "
            Texture01" ]
        NamedMaterial "Floor"
        Shape "trianglemesh" "integer indices" [ 0 1 2 0 2 3 ] "point P" [
            -0.785994 0 3.11108 -4.55196 -4.75246e-007 -0.80933 -0.63155 0
            -4.57529 3.13441 4.75246e-007 -0.654886 ] "normal N" [ 1.2361e
            -007 -1 2.4837e-009 1.2361e-007 -1 2.4837e-009 1.2361e-007 -1
            2.4837e-009 1.2361e-007 -1 2.4837e-009 ] "float uv" [ 0 0 1 0 1 1
             0 1 ]
WorldEnd
```

当场景文件检测到 MakeNamedMaterial 时，材料类在 pbrtMakeNamedMaterial() 函数中从文件加载到渲染器内，并在该函数中将该材料记录到 graphicsState 中。GraphicsState 是保存当前场景中图形状态的类，比如当前图形应该使用什么纹理，使用什么材质等。当场景文件检测到 NamedMaterial 时，就设置 graphicsState 中的当前材料为该材料，比如上面的文件中，读到 NamedMaterial "Floor" 后，下面的 Shape 的材料属性就是 Floor 了。

而当场景文件检测到 Material 时（上面的场景段没有这个标志，其实这个标志的作用就相当于 MakeNamedMaterial 和 NamedMaterial 一起使用），在 pbrtMaterial() 函数中，直接通过 MakeMaterial 来构造，并设置为 graphicsState 的当前材料：

```
    std::shared_ptr<Material> mtl = MakeMaterial(name, mp);
    graphicsState.currentMaterial =
        std::make_shared<MaterialInstance>(name, mtl, params);
```

设置为当前材料以后，在其他材质加载进来改变它之前，后面读入的形状就会被设定为这种材料。

材质有很多种，例如 matte，plastic，它们各自拥有不同的渲染效果。正是因为材质与很多因素都有关，例如 BRDF，BTDF 等，这里不展开介绍（会在系列 8 开始介绍反射与材质）。这本小书的目标也只是讲解场景文件的加载和构建，一次灌输太多内容会变得不好理解。

3 9 纹理类 Texture

纹理类在 pbrtTexture() 函数中被加载到渲染器中，并将信息保存在 graphicsState 里，和前面的材料差不多。只不过纹理会被材料绑定，比如上面的 Floor 材料就绑定了 Texture01 纹理。

3 10 灯光类 Light

面光源在 pbrtAreaLightSource() 被加载到渲染器中，点光源方向光源等通过 pbrtLightSource() 加载，之前提到过，光源的形状会在 pbrtShape 里面进行处理。

3 11 介质类（体渲染）Medium

介质的创建方法和材料类基本上没有太大区别，它是由 pbrtMakeNamedMedium()，pbrtMediumInterface() 来加载的，不再赘述。

3 12 积分器类 Integrator

积分器类是用来积分渲染方程的类，简单来说就是各种渲染方法，例如路径追踪，光子映射，双向光传输，Metropolis 光传输等，该类被 pbrtIntegrator() 加载到渲染器中，然后根据其名称来选择渲染时用到的渲染器。

3 13 小结

综上所述，PBRT 其实结构并不复杂。加载文件时，状态和参数会被保存在 RenderOptions，GraphicsState 等一些属性类里，文件中的场景会依次加载生成各个类的实体，然后启动渲染。

我们已经基本上了解了 PBRT 的场景文件中不同的语句会被如何加载到渲染器并构造场景，但是对于整体的启动流程，我们还不是特别清楚，这也是下一章我们要介绍的。

四　渲染程序启动

有了上一章的铺垫，这一章再讲述就容易很多了。

4 1　程序的开始和结束

定位到 pbrt.cpp 文件中，main 函数作为应用程序的入口。从 pbrtParseFile 开始加载文件场景，调用 parse 来加载。parse 函数做一些初始化工作以后，就在 while 大循环里逐行处理场景文件。

```
    while (true) {
        string_view tok = nextToken(TokenOptional);
        if (tok.empty()) break;
        switch (tok[0]) {
        case 'A':
            break;
        case 'C':
            break;
        /**** 此处省略一些项 ****/
        case 'T':
            break;
        case 'W':
        default:
            syntaxError(tok);
        }
    }
```

针对不同的首字母来执行不同的处理，例如当处理到 Camera 时，首字母是 C，就定位到 case 'C' 上。

当最终检测到"WorldEnd" 的时候，执行 pbrtWorldEnd()，这个函数就开始根据之前的读入和加载的参数来创建场景，然后渲染，渲染完以后再清理内存和 reset 内部设置，伪代码如下：

```
    void pbrtWorldEnd() {
        // (1) 保证场景正确加载。
        ......
        // (2) 创建场景并开始渲染。
        ......
        // (3) 渲染完后，clean 和 reset 渲染组件，内存等。
        ......
    }
```

全部执行完以后，就释放内存，结束程序。

4 2　程序启动整体流程

程序启动过程比较简单，首先先列一个场景文件的大纲：

```
设置相机，采样器，积分器等
WorldBegin
    AttributeBegin
    ......
    AttributeEnd
    AttributeBegin
```

```
    ......
    AttributeEnd
WorldEnd
```

加载场景时，在 WorldBegin 之前，所有的场景信息都会被保存在 RenderOptions 结构里，包含了场景内容以及参数。

当加载到 WorldBegin 里面的内容后，对场景建立一个 GraphicsState，这个 GraphicsState 是整个场景的状态，里面存储了各种渲染用到的资源，比如纹理和材料等。

之前说过，每个 AttributeBegin 和 AttributeEnd 属性对都是独立的场景信息，这时，会给每个属性对里的场景建立一个 GraphicsState，这个 GraphicsState 会在 AttributeBegin 后被压栈到一个 vector 里，当遇到 AttributeEnd 后被弹栈，里面存储了当前场景的属性，比如当前的三维变换，当前的纹理，当前的材料属性等。

前一节说过，当解析完 WorldEnd 后，执行 pbrtWorldEnd() 函数，该函数负责创建场景和启动渲染。pbrtWorldEnd() 函数会首先创建场景：

```
    std::unique_ptr<Integrator> integrator(renderOptions->MakeIntegrator());
    std::unique_ptr<Scene> scene(renderOptions->MakeScene());
```

MakeIntegrator() 函数把存储在 RenderOptions 里面的参数都拿出来进行初始化，该函数首先初始化相机，调用 MakeCamera()；然后初始化采样器，调用 MakeSampler()；然后使用相机和采样器作为参数来初始化积分器，例如 integrator = CreatePathIntegrator(IntegratorParams, sampler, camera)；然后判断如果有参与介质（体渲染物体）在场景中，当前的积分器是否支持渲染参与介质；最后判断灯光是否存在，如果不存在就发出警告。

MakeScene() 更简单了。根据选择的加速结构类型创建加速结构，然后使用加速结构和构造的光源 vector 数组来初始化场景类：Scene *scene = new Scene(accelerator, lights)；然后把容器里的基元（形状）和灯光给清除，释放内存。

创建场景没有问题的话，就开始渲染了：

```
    if (scene && integrator) integrator->Render(*scene);
```

根据所选的积分器来执行对应的渲染程序。

curTransform 的补充

本小节可以忽略，因为即使不了解，也不会影响后面的学习。

前面我们简单说过，curTransform 是一个保存当前变换状态的类，在 parse() 函数中，检测到 WorldBegin 会调用 pbrtWorldBegin() 函数：

```
    if (tok == "WorldBegin")
        pbrtWorldBegin();
```

pbrtWorldBegin() 函数：

```
    for (int i = 0; i < MaxTransforms; ++i) curTransform[i] = Transform();
    activeTransformBits = AllTransformsBits;
```

activeTransformBits 表示“激活的变换”所在的位，即对应的二进制中为 1 的位表示激活的变换，为 0 表示没有激活的变换。关于 AllTransformsBits 的定义可以在 api.cpp 中找到：

```
int MaxTransforms = 2;
int AllTransformsBits = (1 << MaxTransforms) - 1;
```

最大的变换保存数为 2，因此，AllTransformsBits 的最后两个位都是 1，表示全都激活了。

在场景文件中，WorldBegin 后面可能就定义了一些变换，表示 World 中所有的元素都要执行该变换。

pbrtAttributeBegin() 函数中，会将当前的 curTransform 保存入 pushedTransforms 栈中，然后将 curTransform 乘以 Attribute 中定义的数据。等调用 pbrtAttributeEnd() 函数后，pushedTransforms 弹栈。

大家可能会好奇，在 killeroo-simple.pbrt 文件的第二行中有一个 Rotate，它并不在 WorldBegin 和 WorldEnd 之间，那么这个量被保存到哪里了呢？我们再来看 parse() 函数中关于 Camera 的部分：

```
1 if (tok == "Camera")
2     // 注意第二个参数是一个名为pbrtCamera的函数
3     basicParamListEntrypoint(SpectrumType::Reflectance, pbrtCamera);
```

重点是 pbrtCamera() 这个函数：

```
1 renderOptions->CameraToWorld = Inverse(curTransform);
```

现在大家应该就明白了：Rotate 读取后，此时 curTransform 表示的是将世界坐标的内容变换到相机坐标系的变换，因此，取逆就变为了相机坐标到世界坐标系的变换，也就是 CameraToWorld。

现在还有一个问题，为什么要定义最多两个变换（MaxTransforms=2）？其实这是因为 PBRT 支持渲染动态物体。对于运动的物体，很显然运动开始的变换和运动结束的变换是不同的；而对于不动的物体，一开始的变换和结束的变换是相同的。所以：

```
1  void pbrtActiveTransformAll() {
2      activeTransformBits = AllTransformsBits;
3  }
4  void pbrtActiveTransformEndTime() {
5      // EndTransformBits 为 2
6      activeTransformBits = EndTransformBits;
7  }
8  void pbrtActiveTransformStartTime() {
9      // StartTransformBits 为 1
10     activeTransformBits = StartTransformBits;
11 }
```

宏 FOR_ACTIVE_TRANSFORMS 用来将 TransformSet 中激活的 Transform 进行赋值。

4 3 小结

到目前为止，这本书就结束了。从 11 月 19 日开始写，一直写到了 11 月 25 日，总共花了六天的时间。

PBRT 给我的感觉，一直都是短小精悍。这么一个不到五万行的渲染器，可以渲染出这么多种类的精美图片，非常有趣。它的代码组织方式也让我从中受益良多。

接下来，我还会继续写 PBRT 解析的书目，通过这些小书，大家就可以很容易的动手去将 PBRT 的内容移植到自己的系统上，然后测试效果。

我一直比较喜欢一些小书，[1][2][3] 是让我眼前一亮的著作，两三周就能看完一遍的内容，非常适合初学者学习。本 PBRT 系列也会沿用这种风格，而就算以后该系列写得越来越多，我应该也不会考虑将它们合并成一本厚书。对于本书作为系列 1，当你阅读完本书后，如果有人问题，PBRT 是怎么实现的呢？我想你已经有了自己的答案。哪怕你还不清楚各个模块的细节，但是最起码这个架构对你来说已经不再是一个巨无霸难啃的大部头，至于内部细节，渲染引擎还不是都差不多嘛（PBRT 与 [1][2][3] 的差别并没有那么大，很多地方都是非常相似的）。

最后，谢谢大家能够读到本书的最后一章，有问题可以随时批评指正，希望在大家的帮助下，Dezeming Family 可以帮助更多的人，Make rendering easier!

参考文献

[1] Shirley P. Ray Tracing in One Weekend[J]. 2016.

[2] Shirley P. Ray Tracing The Next Week[J]. 2016.

[3] Shirley P. Ray Tracing The Rest Of Your Life[J]. 2016.

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

[5] Igehy, H. Tracing ray differentials.(1999). SIGGRAPH '99 Proceedings

[6] Marschner S , Shirley P . Fundamentals of computer graphics. 4th edition.[J]. World Scientific Publishers Singapore, 2009, 9(1):29-51.