PBRTv4-系列2-代码结构与场景加载

Dezeming Family

2023年8月26日

DezemingFamily系列文章和电子文档**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的系列文章，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。

目录

一　代码结构

1 1　源码结构

1 1.1　ext

ext里是PBRT使用的第三方工具库。

double-conversion：该项目（双转换）为IEEE双精度提供二进制-十进制和十进制-二进制例程。该库由从V8 JavaScript引擎中提取的高效转换例程组成。代码经过重构和改进，可以更容易地在其他项目中使用。

filesystem：这个类只是一个临时的解决方法，可以避免严重的boost依赖，直到将来某个时候将“boost::filesystem”集成到标准模板库中。

filp：FLIP是一种新的算法，可以自动评估交替图像之间的差异，并有助于图形研究。它建立在人类感知原理的基础上，并结合了对观看距离和监视器像素大小的依赖性。

glad：OpenGL的工具，基于官方规范的多语言GL/GLES/EGL/GLX/WGL加载程序生成器。

glfw：GLFW是一个开源、多平台的库，用于OpenGL、OpenGL ES和Vulkan在桌面上的开发。它提供了一个简单的API，用于创建窗口、上下文和表面，接收输入和事件。

gtest：The Google C++ Testing Framework (Google Test)。

libdeflate：是一个基于DEFLATE的快速全缓冲区压缩和解压缩库。

lodepng：LodePNG是一个PNG图像解码器和编码器，集于一体，不需要对zlib或libpng的依赖或链接。它是为C(ISO C90)设计的，并且有一个C++包装器，上面有一个更方便的接口。

openexr：OpenEXR提供了电影行业专业级图像存储格式EXR文件格式的规范和参考实现。

openvdb：OpenVDB是一个获得奥斯卡奖的开源C++库，它包括一个新颖的分层数据结构和一套工具，用于高效存储和操作在三维网格上离散化的稀疏体数据。

ptex：Ptex是由华特迪士尼动画工作室开发的纹理贴图系统，用于制作高质量的渲染：不需要UV指定。

qoi：无损地将图像压缩到与PNG大小相似的大小，同时提供20倍-50倍的编码和3倍-4倍的解码速度。

rply：读写ply文件。ply文件是一种三维mesh模型数据格式，全名为多边形档案（Polygon File Format）或斯坦福三角形档案（Stanford Triangle Format）。

skymodel[6]：关于论文”An Analytic Model for Full Spectral Sky-Dome Radiance”和”Adding a Solar Radiance Function to the Hosek Skylight Model”的实现。

stb：用于公共域中C/C++的单个文件头文件库的集合，主要面向游戏开发人员。它们被设计为易于集成、易于使用和易于发布。

utf8proc：是一个小型、干净的C库，它为UTF-8编码的数据提供Unicode规范化、大小写折叠(case-folding)和其他操作，支持Unicode版本15。它最初是由Jan Behrens和公共软件集团的其他成员开发的，他们应该为这个包获得几乎所有的赞誉。在公共软件集团的支持下，Julia开发人员接管了utf8proc的开发，因为最初的开发人员已经转移到其他项目。

zlib：免费通用的、法律上不受限制的、不受任何专利保护的无损数据压缩库，几乎可以在任何计算机硬件和操作系统上使用。zlib数据格式本身可以跨平台移植。

1 1.2　pbrt

这里面就是pbrt的各个源码了，包括cpu代码和gpu代码。

cmd目录下的文件生成各种命令行可执行程序，比如pbrt.exe/soac.exe等。

cpu目录下都是在CPU上实现的功能（只能在CPU运行），比如各种渲染积分器。

gpu目录下是基于Optix实现的渲染器，因为使用Optix管线，所以不需要再额外定义渲染积分器。

base目录下是一些基类，另外一些派生类的实现放在了pbrt主目录下。这些代码都是可以在CPU/GPU上执行的。

util目录下是utility工具，比如矩阵向量变换/颜色色彩工具/误差界定等。

wavefront是wavefront path tracing方法。在为GPU编程时，简单地将大型CPU程序移植到同样大的GPU内核通常不是一个好方法。由于GPU上的SIMT执行模型，控制流的差异会带来很大的性能损失，高寄存器使用率也会降低GPU的高延迟、高带宽存储系统所必需的延迟隐藏能力。使用wavefront公式在GPU上实现的路径跟踪器可以避免使用评估成本高昂的材料时特别突出的这些陷阱。

1 2　工程结构

1 2.1　编译生成的文件

编译生成的文件有：

- imgtool.lib
- libpbrt.lib
- nanovdb2pbrt.lib
- pbrt.lib
- pbrt_test.lib
- plytool.lib
- pspec.lib
- sky_lib.lib
- cyhair2pbrt.exe
- imgtool.exe
- nanovdb2pbrt.exe
- pbrt_test.exe
- plytool.exe
- pspec.exe
- rgb2spec_opt.exe
- soac.exe

在CMakeLists.txt中搜索OUTPUT_NAME就可以看到工程要输出的库或者exe文件的名称，以及要输出这些文件所依赖的源文件或者库。

大部分库或者函数前面都有对应的介绍，这里把几个之前没有介绍过的功能再介绍一下。

- pspec.exe：计算pbrt采样器使用的各种采样器(sampler)的功率谱。
- rgb2spec_opt.exe：用于RGB光谱和曲线光谱之间的转换。

1 2.2　pbrt_exe工程

该工程就是启动pbrt命令行来渲染的工程。

main()函数分为如下几个步骤：

- 将命令行参数转换为字符串向量，然后处理命令行参数。
- 在终端打印欢迎语句。
- 检查给定参数的合法性。

- 初始化PBRT。
- Parse场景文件并启动渲染

很多命令行参数都可以在场景文件中设置。在下面函数中可以看到参数及其示意：

```
static void usage(const std::string &msg = {})
```

命令行参数在解析后会存放在PBRTOptions对象options中。有几个参数先介绍一下：

- –format：将输入文件的重新格式化版本打印为标准输出，并将所有三角形网格转换为PLY文件。该选项不渲染图像。
- –toply：将输入文件的重新格式化版本打印为标准输出。该选项不渲染图像。
- –quiet：取消显示除错误消息以外的所有文本输出。
- –render-coord-sys：渲染的场景所使用的坐标系。
- –pixelmaterial：打印像素所对应的材质。当使用此项时，会禁用GPU渲染。
- –interactive：提供交互界面，仅支持–gpu和–wavefront渲染积分器。

初始化PBRT时调用的InitPBRT(options)来自于pbrt_lib工程下的pbrt.cpp，与pbrt_exe工程下的pbrt.cpp是不同的。InitPBRT(.)的工作就是初始化环境，比如如果PBRTOptions对象设置为使用GPU渲染，就初始化GPU设备，并且选择编号0的GPU执行渲染。

1 3 Parse场景

ParserTarget有两种派生类，一是BasicSceneBuilder，另一个是FormattingParserTarget。执行加载FormattingParserTarget时并不会启动渲染：

```
if (format || toPly || options.upgrade) {
    FormattingParserTarget formattingTarget(toPly, options.upgrade);
    ParseFiles(&formattingTarget, filenames);
}
```

Parse场景文件使用的是pbrt_lib工程下的parser.cpp文件中的函数ParseFiles(..)。

执行加载BasicSceneBuilder时才会启动渲染：

```
BasicScene scene;
BasicSceneBuilder builder(&scene);
ParseFiles(&builder, filenames);
```

之后启动渲染：

```
// Render the scene
if (Options->useGPU || Options->wavefront)
    RenderWavefront(scene);
else
    RenderCPU(scene);
```

渲染完以后，就清理内存：

```
CleanupPBRT();
```

二 Parse文件

在PBRT v4中，所有的类和变量都不再是派生关系，而都是独立的类，只是在进行对象初始化时使用智能指针TaggedPointer进行管理。

2 1 示例工程

加载过程以下面的.pbrt文件为例，将内容列出来：

```
Integrator "path"
    "integer maxdepth" [ 65 ]
Transform [ -0.264209 0.071763 -0.961792 -0 1.86265e-9 0.997228 0.074407 -0
    -0.964466 -0.019659 0.263477 -0 -0.886691 -1.14097 5.46644 1 ]
Sampler "sobol"
    "integer pixelsamples" [ 1024 ]
PixelFilter "triangle"
    "float xradius" [ 1 ]
    "float yradius" [ 1 ]
Film "rgb"
    "string filename" [ "living-room.png" ]
    "integer yresolution" [ 720 ]
    "integer xresolution" [ 1280 ]
Camera "perspective"
    "float fov" [ 58.715508 ]

WorldBegin

Transform [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 ]

Texture "Texture01" "spectrum" "imagemap"
    "string filter" "trilinear"
    "string filename" [ "textures/cap.tga" ]

MakeNamedMaterial "BottleCap"
    "string type" [ "coateddiffuse" ]
    "bool remaproughness" [ false ]
    "float vroughness" [ 0.15 ]
    "float uroughness" [ 0.15 ]
    "texture reflectance" [ "Texture01" ]

NamedMaterial "BottleCap"
AttributeBegin
    Transform [ 1 0 0 0 0 1 0 0 0 0 1 0 0.634815 0 -0.151441 1 ]
    Shape "plymesh"
        "string filename" [ "models/Mesh001.ply" ]
AttributeEnd
```

注意没有WorldEnd标志。

ParseFiles(..)会调用parse(..)函数来加载场景文件。

2 2 加载基本构件

在WorldBegin之前，主要是加载path，全局变换以及采样器类型，Film类型等。

以采样器为例，假如parse到"Sampler"，就执行下面的语句：

```
if (tok->token == "Sampler")
    basicParamListEntrypoint(&ParserTarget::Sampler, tok->loc);
```

其实相当于调用BasicSceneBuilder::Sampler(...)函数，得到一个SceneEntity对象，以记录该对象的参数。

当全部场景文件加载完以后，基元/光源和材质等场景对象会在RenderCPU(.)或者RenderWavefront(.)函数中被创建。如果是用GPU渲染，那么有些场景文件的设置（比如采样器类型）就不会起作用。

2 3 创建相机/采样器/Film等对象

当parse到"WorldBegin"时，就执行BasicSceneBuilder::WorldBegin(.)函数。该函数调用Scene::SetOptions(...)创建场景对象，例如相机/Film/采样器/渲染积分器。虽然Scene::SetOptions(...)会有加速结构的场景参数，但是并不会创建加速结构，因为毕竟此时基元都还没有parse到。此时只是记录加速结构的类型。

在RenderCPU(.)或者RenderWavefront(.)函数中才会创建基元，此时已经parse完场景文件了。

RenderCPU(.)函数中获得已经创建好的采样器的代码：

```
Sampler sampler = parsedScene.GetSampler();
```

2 4 加载全局变换

我们默认场景的所使用的坐标系，也就是命令行启动时的–render-coord-sys参数，是"cameraworld"，其他参数还有"camera"以及"world"（后面的小节再解释该参数的用处）。

当parse到"Transform"时，就会调用：

```
if (tok->token == "Transform") {
    if (nextToken(TokenRequired)->token != "[")
        syntaxError(*tok);
    Float m[16];
    for (int i = 0; i < 16; ++i)
        m[i] = parseFloat(*nextToken(TokenRequired));
    if (nextToken(TokenRequired)->token != "]")
        syntaxError(*tok);
    target->Transform(m, tok->loc);
}
```

target-¿Transform(..)函数就是调用：

```
graphicsState.ForActiveTransforms([=](auto t) {
    return Transpose(pbrt::Transform(SquareMatrix<4>(pstd::MakeSpan(tr, 16))
        ));
});
```

该函数做了这么一件事情：

```
void ForActiveTransforms(F func) {
    for (int i = 0; i < MaxTransforms; ++i)
        if (activeTransformBits & (1 << i))
```

```
            ctm[i] = func(ctm[i]);
}
```

要理解这件事，我们需要先看这三个常量的二进制值：

```
static constexpr int StartTransformBits = 1 << 0;
static constexpr int EndTransformBits = 1 << 1;
static constexpr int AllTransformsBits = (1 << MaxTransforms) - 1;
```

二进制值（最后八位，其他位都是0）：

$$StartTransformBits: \ 0000\ 0001$$
$$EndTransformBits: \ 0000\ 0010$$
$$AllTransformsBits: \ 0000\ 0011$$

注意前两个都是给运动物体使用的，为了简洁性可以暂时不用在意。

我们可以看到，变换队列最多只有两个值，因为：

```
constexpr int MaxTransforms = 2;
```

parse最开始时，activeTransformBits是AllTransformsBits，因此我们加载得到的"Transform"所表示的变换矩阵会将TransformSet对象ctm的两个元素都初始化（只有两个元素，一个代表运动物体开始时的变换，一个代表运动物体结束时的变换，如果是静止的物体，那么这两个矩阵就是相同的值）。

当parse到"WorldBegin"时，在BasicSceneBuilder::WorldBegin(.)函数会将全部ctm的两个元素都初始化为单位矩阵：

```
for (int i = 0; i < MaxTransforms; ++i)
    graphicsState.ctm[i] = pbrt::Transform();
graphicsState.activeTransformBits = AllTransformsBits;
```

2 5 加载每个物体的变换

因此，对于"WorldBegin"后面的静态物体，当parse到"AttributeBegin"时，就把当前状态graphicsState入栈，当遇到"AttributeEnd"时，就恢复此前保存的状态graphicsState，然后弹栈。

所以，如果我们在"WorldBegin"后直接定义了Transform，它会作用于全部物体；如果只是在"AttributeBegin"和"AttributeEnd"之间定义了Transform，它只会作用于当前属性块内的物体。

为什么"WorldBegin"后直接定义的Transform会作用于全部物体呢？是因为ForActiveTransforms(.)函数中就是将已有变换的结果乘以新parse到的Transform（ctm[i] = func(ctm[i])就是将之前parse到的变换乘到新的变换中）。

2 6 坐标系

再回到"WorldBegin"之前的文件parse中。

Transform类型如果在"Camera"之前，就会影响到Camera的值：

```
TransformSet cameraFromWorld = graphicsState.ctm;
TransformSet worldFromCamera = Inverse(graphicsState.ctm);
```

命令行启动时的–render-coord-sys参数有"cameraworld"、"camera"以及"world"。这些参数会在下面的代码中起作用：

```
CameraTransform cameraTransform(
    AnimatedTransform(worldFromCamera[0], graphicsState.transformStartTime,
        worldFromCamera[1], graphicsState.transformEndTime));
renderFromWorld = cameraTransform.RenderFromWorld();
```

这些代码会计算worldFromRender。

假设都是静态场景，没有运动的物体。如果是"camera"参数，意味着worldFromRender就是worldFromCamera。如果是"cameraworld"参数，意味着worldFromRender是：

```
Translate(Vector3f(worldFromCamera(Point3f(0, 0, 0))))
```

如果是"world"参数，意味着worldFromRender是单位矩阵变换。

在进行渲染发射采样射线时，相机会进行如下操作：

```
CameraRay{RenderFromCamera(ray)};
```

也就是把相机坐标系下的相机采样射线变换到渲染坐标系下。

在静态场景下，进一步可以看出，如果是"camera"参数，意味着RenderFromCamera就是单位矩阵，那么相机坐标系下的相机采样射线就直接在当前空间采样。如果是"world"参数，就相当于将worldFromCamera作用于相机采样射线，把射线从相机空间变换到世界空间。如果是"cameraworld"参数，RenderFromCamera就是：

```
Translate(-Vector3f(worldFromCamera(Point3f(0, 0, 0)))) * worldFromCamera
```

2 7 材质和纹理

parse到"MakeNamedMaterial"参数时，调用BasicSceneBuilder::MakeNamedMaterial(...)函数。

注意材质下的内容（比如用了哪个纹理，）都被parse的时候加载到了ParsedParameterVector对象params中了，用于后续对材质初始化时使用。

```
MakeNamedMaterial "BottleCap"
    "string type" [ "coateddiffuse" ]
    "bool remaproughness" [ false ]
    "float vroughness" [ 0.15 ]
    "float uroughness" [ 0.15 ]
    "texture reflectance" [ "Texture01" ]
```

当parse到"NamedMaterial"参数时，就会调用BasicSceneBuilder::NamedMaterial(..)函数。将当前环境下的材质设置为该参数定义的材质。之后加载的几何体的材质都是该材质。

2 8 Shape的加载

parse到"Shape"时，就会调用BasicSceneBuilder::Shape(...)函数。

renderFromShape变换就是：

```
renderFromWorld * graphicsState.ctm[]
```

此时的graphicsState.ctm就是每个AttributeBegin和AttributeEnd里面定义的Transform。

形状对象也是在RenderCPU(.)函数或者RenderWavefront(.)函数中才初始化的。对于CPU渲染，会调用Shape::Create(...)函数来初始化。

纹理/材质/渲染积分器/采样器等对象也都是在RenderCPU(.)函数或者RenderWavefront(.)函数中进行的。

2 9　渲染的启动

场景创建完成以后，就会在RenderCPU(.)函数或者RenderWavefront(.)函数中进行渲染。

三　小结

本节我们介绍了PBRT的加载流程。但由于PBRT v4包含CPU渲染和GPU渲染，因此后面的讲解我们也会分成两个部分。CPU渲染我们主要注重框架机制，在v3系列文章讲解过的内容会减少篇幅；而GPU渲染，以及包括可交互GPU渲染，我们会介绍PBRT在Optix上的实现流程。

参考文献

[1] https://github.com/mmp/pbrt-v4

[2] https://pbrt.org/

[3] https://developer.nvidia.com/designworks/optix/downloads/legacy

[4] https://pbrt.org/resources

[5] https://www.cnblogs.com/Heskey0/category/2166679.html

[6] https://github.com/ebruneton/clear-sky-models/tree/master/atmosphere/model/hosek

[7] https://www.cnblogs.com/Heskey0/p/15973546.html