Mitsuba 系列 2-Mitsuba 的依赖库

Dezeming Family

2023 年 1 月 13 日

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

目录

一　我们的工程布置方法与初始工程

本节描述一下我们的 Mitsuba v1 系列电子书的配套源码使用方法。介绍如何布置工程，以及介绍初始工程的结构。

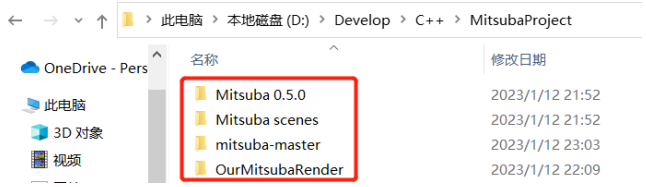
本文可能会穿插着介绍一些移植时遇到的琐碎的内容，这些内容也是比较重要的，但是由于本人已经把需要移植的内容移植完毕，所以大家不用再动手自己移植一遍了，直接使用 CMake 进行编译和使用即可。

1 1　工程布置方法

在 D 盘下面目录中新建一个 MitsubaProject 文件夹：

```
1 D:\Develop\C++\MitsubaProject
```

在该目录下一共放置四个子目录：



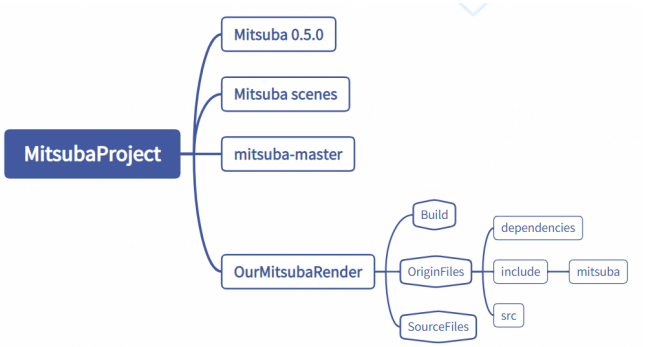
第一个目录是 Mitsuba 0.5.0，是从 [1] 下载的可执行软件的目录。

第二个目录 Mitsuba scenes，用于存放从 [1] 下载的场景文件。

第三个目录 mitsuba-master 是从 [2] 下载的代码文件。

以上这三个目录里的文件都在 [3] 栏目中打包，可以直接在该网页上下载。

OurMitsubaRender 目录是我们自己建立的目录，以后我们移植和实现代码都在这个目录下。该目录的结构是：



SourceFiles 和 Build 目录分别是我们自己的代码目录和 CMake 生成工程的目录，这 SourceFiles 目录下放置着我们每本电子书自己实现的代码。OriginFiles 目录下的 dependencies 即依赖库目录（从 [4] 下载得到的依赖项，解压以后把文件夹重命名为 dependencies；在 [3] 中也有打包的备份）；include 是从 mitsuba-master 中取出的 mitsuba 的头文件目录；src 是从 mitsuba-master 中取出的 src 文件夹（即 mitsuba 的源码）。

这样设置好的好处是后面描述时较为简洁，避免混乱。我们的目标是把 include 和 src 里面的文件逐步移植到自己的系统上。

此外，我在 D 盘安装 Qt5，这里我安装的是 Qt5.7，目录在：

```
1 D:/DevTools/QT5/5.7/msvc2015_64
```

在我们每本电子书的源码的 CMakeLists.txt 文件中，QT_PATH 都是设为了该固定值，大家需要根据自己安装 Qt 的目录去修改。

1 2　基础工程代码

见 SourceFiles/0 - InitialWorks 目录。

在 Build 目录新建一个同名子目录 0 - InitialWorks，使用 CMake 把工程建立到该子目录下。

我们的初始工程只是与 GUI 相关的代码，用于打开一个 Qt 窗口，和 PBRT 系列基本上是一样的。DebugText 是用于调试输出显示的类。

DisplayWidget 是显示区，当点击 Start Rendering 按钮时，就会创建一个 Qt 子线程来运行渲染程序，然后不断地在 DebugText 对象中输出“Rendering”。

二　添加依赖库并测试

本节讲解测试 Mitsuba 项目给出的依赖库的使用。

2 1　库和头文件的包含

见 SourceFiles/1 - dependencies 目录，该目录下相比前一个工程多了一个 Include 文件夹，里面包含了 Mitsuba v1 的全部头文件。

下面解释一下我在 CMakeLists.txt 文件中新增加的一些内容，这些内容是用来包含依赖库头文件、Mitsuba 头文件以及链接依赖库中相应 lib 的。

在 CMakeLists.txt 中，下面几行代码用于定义 dependencies 中的依赖库头文件和库的目录：

```
SET(MITSUBA_DEPENDENCIES_PATH ${CMAKE_CURRENT_SOURCE_DIR}/../../OriginFiles/
    dependencies/)
SET(MITSUBA_DEPENDENCIES_INCLUDE_PATH ${MITSUBA_DEPENDENCIES_PATH}include/)
SET(MITSUBA_DEPENDENCIES_LIB_PATH ${MITSUBA_DEPENDENCIES_PATH}lib/)
```

然后让当前工程能够包含这些头文件目录，并且链接这些库到当前工程中：

```
target_include_directories(MitsubaRender PUBLIC
    ${MITSUBA_DEPENDENCIES_INCLUDE_PATH}
    Include/
)
target_link_libraries(MitsubaRender ...)
```

因为我们的工程中已经包含了 Qt，链接库中不链接 Mitsuba 依赖库里的 Qt 中的 lib；且我们不需要 Python 工具，因此不添加与 Python 有关的库。

以及工程中需要一些预定义宏，参见 CMakeLists.txt：

```
add_compile_definitions(
        SINGLE_PRECISION
        SPECTRUM_SAMPLES=3
)
```

SINGLE_PRECISION 表示 Float 类型是 32 位浮点数，SPECTRUM_SAMPLES=3 表示光谱（颜色）类型是三通道（比如 RGB 或者 XYZ）。

2 2　测试依赖库和 Mitsuba 头文件

还是见根据 SourceFiles/1 - dependenciesCMake 的工程，里面的 IMAGraphicsView 类构造函数中有几行代码用来测试矩阵相乘和求逆函数。运行程序可以得到正确输出，则说明我们的依赖库和头文件大致是移植成功的。

在移植的过程中，我们先尽量不修改源文件，有些情况不得不修改时，我会用序号来记录。

修改内容一：我们在 platform.h 文件中做一些修改，把生成和导入 dll 的宏定义的内容注释掉：

```
#define MTS_EXPORT //__declspec(dllexport)
#define MTS_IMPORT //__declspec(dllimport)
```

使得我们的工程不导出 dll 库，而是直接作为源码使用。

Mitsuba 有四个基本功能支持库，分别是 core（表示基础功能，比如 I/O、数据结构、基本运算功能）、render（表示基本内容的抽象，比如 light、shape、material 等）、hw（即 hardware 加速库，实现了 OpenGL 显示的相关内容，并且调用 VPL 算法来渲染场景的可交互预览）以及 bidir（与实现双向路径追踪与 MLT 算法有关）。

2 3 包含库源文件

本小节代码见 SourceFiles/2 - lib-src 文件夹。

我们前面只包含了 Mitsuba 头文件的内容，但是没有源文件，这些源文件在 OriginFiles/src 的下面几个文件夹里：

```
libbidir
libcore
libhw
librender
```

这几个文件夹被复制到了 SourceFiles/2 - lib-src/Include/mitsuba 目录下。

我们在 CMake 里删除了对 libhw 库的引用，因为该库仅仅只是用于显示渲染结果的。

修改内容二： 由于目录的设置，将 medium.h 的头文件索引修改成了：

```
#include "../../../Src/medium/materials.h"
```

修改内容三： 在 Statistics.h 里的 Statistics 类中定义了一个静态成员 m_instance，然后在 Statistics.cpp 文件中进行了初始化。StatsCounter 是记录数据的，需要在调用其构造函数之前就已经执行对 m_instance 初始化。每种数据（比如执行了多少次镜面反射）都会定义一个全局 StatsCounter 对象，不能保证此时 m_instance 已经被初始化了，所以当把所有的工具都定义在一个工程文件时就会报错（StatsCounter 的构造函数的 assert 会报错）。

于是我们暂时将 m_instance 设置为全局变量，并改名为 Statistics_m_instance_dez，这里的 dez 表示是被 dezeming 修改过的地方。同时，在 Statistics.cpp 文件中的 StatsCounter 的构造函数加上一行，保证在使用前先初始化：

```
if(Statistics::getInstance() == NULL)Statistics_m_instance_dez = new
    Statistics();
```

这样运行代码时就不会因为还没有初始化而报错了。

三 当前代码功能介绍

我们本节分为两部分，第一部分是介绍依赖库的主要功能。第二部分是介绍一下 bidir、core、render 和 hw 这几个目录的里的头文件，以及 libbidir、libcore、librender 和 libhw 这几个源文件它们的功能。由于我们并没有移植和包含 hw 与 libhw，所以暂时先不介绍它们，以后移植的过程中如果需要再说。

大家应当把这些文件都依次打开，先混个眼熟，有助于后面使用时能够进行联想。

3 1 依赖库

zlib：提供数据压缩用的函数库，由 Jean-loup Gailly 与 Mark Adler 所开发。

OpenEXR：解析 exr 格式的库。exr 是一种开放标准的高动态范围图像格式，在计算机图形学里被广泛用于存储图像数据。

libjpeg-turbo：libjpeg-turbo 图像编解码器, 使用了 SIMD 指令 (MMX,SSE2,NEON,AltiVec) 来加速 x86,x86-64,ARM 和 PowerPC 系统上的 JPEG 压缩和解压缩。

libpng：用于读写 png 文件，libpng 实际上使用的是 zlib 的算法。

boost：Boost 是为 C++ 语言标准库提供扩展的一些 C++ 程序库的总称。Boost 库是一个可移植、提供源代码的 C++ 库，作为标准库的后备，是 C++ 标准化进程的开发引擎之一。

xerces-c：Xerces 是由 Apache 组织所推动的一项 XML 文档解析开源项目。

glew：(OpenGL Extension Wrangler Library) OpenGL 扩展库，是个简单的工具，用于帮助 C/C++ 开发者初始化扩展（OpenGL 扩展功能）并书写可移植的应用程序。

half：开源 C++ 库，用于半精度浮点运算。

glext 和 khr：OpenGL 里的 Registry 头文件，其实就是常用的 OpenGL 头文件。

FFTW：FFTW(the Faster Fourier Transform in the West) 是一个快速计算离散傅里叶变换的标准 C 语言程序集。

3 2 core

aabb.h 和 aabb.cpp：定义了轴对齐包围盒有关的类。

appender.h 和 appender.cpp：顾名思义，是负责写记录事件的组件，与 log 搭配使用。

atomic.h：基于 PBRT 实现的用于原子操作的功能。

autodiff.h：利用 Eigen 实现的自动微分功能。

barray.h：定义 2D 块数组，用于有效缓存 2 维数据。

bitmap.h 和 bitmap.cpp：读取图像，支持读取 PNG, JPEG, BMP, TGA 以及 OpenEXR 格式，并支持写入 PNG, JPEG 和 OpenEXR 格式。

brent.h 和 brent.cpp：Brent 的非线性零点探测方法。

bsphere.h：三维绑定球体。

chisquare.h 和 chisquare.cpp：该类执行零假设的卡方拟合优度测试 (chi-square goodness-of-fit test)，即指定的采样过程产生根据提供的密度函数分布的样本。这对于验证 BRDF 和相位函数采样代码的正确性非常有用。目前，它支持 2D 和离散采样方法及其混合。

class.h 和 class.cpp：用于运行时类型信息检查。

cobject.h：泛型可序列化对象，支持从 Properties 实例构造。Mitsuba 中的所有插件都源自可配置对象。此机制允许它们接受外部 XML 文件中指定的参数。此外，它们可以具有子对象，这些子对象对应于 XML 文件中的嵌套实例化请求。我们后面遇到 ConfigurableObject 再详细介绍。

constants.h：定义了一堆常量。

cstream.h 和 cstream.cpp：默认 stdin/stdout 控制台流的流样式接口。

fmtconv.cpp：该文件包含 mitsuba/render/bitmap.h 格式的 FormatConverter 接口的实现。

formatter.h 和 formatter.cpp：用于将 log 信息转化为人们能理解的形式。

frame.h：用于三维坐标系的快速转换。

fresolver.h 和 fresolver.cpp：FileResolver 是一个便捷的类，允许以跨平台兼容的方式在一组可指定的搜索路径中搜索文件（类似于各种操作系统上的 PATH 变量）。

fstream.h 和 fstream.cpp：简单的文件流的实现。在多个 Linux 平台上使用 POSIX 流，在 Windows 上使用本地 API 实现。

fwd.h：声明了所有出现的主要的类名。

getopt.h：用于获得 Option，我们目前加载的库并没有把改头文件加载进来，但是注意这里有对 MTS_EXPORT_CORE 的重新定义，后面如何包含此头文件可能需要修改一些代码，这里只是先知道有这么一回事儿。

half.h 和 half.cpp：这是 OpenEXR 中的 half 的实现，使得我们的依赖库可以不用包含 OpenEXR 支持。

kdtree.h：一个简单的 kd Tree 的实现。

lock.h 和 lock.cpp：递归 boost 线程锁的轻薄封装 (thin warper)。

logger.h 和 logger.cpp：用于输出信息日志。

lrucache.h：通用的 LRU 缓存实现。这个缓存不支持即用的多线程——它需要使用某种形式的锁定机制来保护。

math.h 和 math.cpp：一维函数运算。

matrix.h：通用的固定尺寸的稠密矩阵类，用行主向量存储模式。

mempool.h：内存池操作，用于有效地为对象分配和释放内存。

mmap.h 和 mmap.cpp：内存映射文件（由一个文件到一块内存的映射，类似于虚拟内存）的跨平台抽象层。

mstream.h 和 mstream.cpp：基于内存 buffer 的流，可以自动地进行内存管理。

netobject.h：引用共享网络资源的对象的抽象接口。

normal.h：三维法向量数据结构。

object.h 和 object.cpp：所有 Mitsuba 类的父类，包含与每个对象相关的函数，如引用计数、有限类型内省 (introspection) 和生存期管理。

octree.h：无锁的链接列表数据结构。

platform.h：对不同的平台进行定义。

plugin.h 和 plugin.cpp：抽象插件类——表示可加载的可配置对象和实用程序。

pmf.h：用于将均匀分布转换为我们定义的离散概率密度分布。

point.h：“点”数据结构，有一维到四维的点。

properties.h 和 properties.cpp：用于构造 ConfigurableObject 子类的关联参数映射。

qmc.h 和 qmc.cpp：用于生成准蒙特卡洛随机数序列。

quad.h 和 quad.cpp：非自适应求积分的基本工具（比如勒让德-高斯求积 (Legendre-Gauss quadrature)）。

quat.h：可参数化的四元数数据结构。

random.h 和 random.cpp：面向 SIMD 的快速 Mersenne Twister（SFMT）伪随机数发生器。

ray.h：光线数据结构。

ray_sse.h：用 SSE 指令集加速的光线数据结构。

ref.h：一种类似于智能指针的工具，用于记录对象数等。

rfilter.h 和 rfilter.cpp：可分离图像重建滤波器的通用接口。

sched.h 和 sched.cpp：与工作任务调度等有关。

sched_remote.h 和 sched_remote.cpp：与远程工作调度有关。

serialization.h 和 serialization.cpp：对象的序列化（将内存中保存的对象以二进制数据流的形式进行处理传输）支持。

sfcurve.h：Hilbert space-filling 曲线的二维版本。

shvector.h 和 shvector.cpp：与球谐函数相关功能有关。

simplecache.h：用于缓存昂贵函数调用求值的通用线程本地存储。

spectrum.h 和 spectrum.cpp：与光谱类有关。

spline.h 和 spline.cpp：与样条插值有关。

sse.h：SSE 指令集的头文件。

ssemath.h 和 ssemath.cpp：SSE 加速的数学运算。

ssevector.h：SSE 加速的向量类。

sshstream.h 和 sshstream.cpp：基于加密 SSH 通道的流实现。

sstream.h 和 sstream.cpp：网络 SocketStream 的实现。

statistics.h 和 statistics.cpp：记录统计数据（比如渲染中进行了多少次求交代码）。

stream.h 和 stream.cpp：流对象的基类。

thread.h 和 thread.cpp：跨平台多线程的实现。

timer.h 和 timer.cpp：独立于平台的毫/微/纳秒计时器。

tls.h 和 tls.cpp：与 TLS 管理相关的一些声明，TLS 是 Thread Local Storage(线程局部存储)。

track.h 和 track.cpp：运动追踪。

transform.h 和 transform.cpp：4X4 矩阵描述的线性变换及其逆变换。

triangle.h 和 triangle.cpp：三角形类的实现。

util.h 和 util.cpp：各种功能函数。

vector.h：向量类。

version.h：定义版本号和年份，当前是 0.6.0 版本。

vmf.h 和 vmf.cpp：球上的 Von Mises-Fisher 分布的实现。

warp.h 和 warp.cpp：实现从单位正方形映射到其他域（如球体、半球等）的常用 warp 技术。

zstream.h 和 zstream.cpp：基于 zlib 的压缩/解压流。

3 3 bidir

该目录的内容都与双向路径追踪有关，大家应当先学会 PBRT 系列中的双向路径追踪和 MLT 才能学习这里的函数，但当前我们只是先了解架构，不要求读者掌握这些内容。

common.h 和 common.cpp：一些公共定义。

edge.h 和 edge.cpp：双向路径的边数据结构。

geodist2.h：一种特殊的 clamped two-tailed 几何分布，支持样本生成和概率质量和累积分布函数的估计。

manifold.h 和 manifold.cpp：定义了 SpecularManifold，用于在镜面 manifold 上进行路径扰动。

mempool.h：内存池的创建。

mut_bidir.h 和 mut_bidir.cpp：

mut_caustic.h 和 mut_caustic.cpp：Veach 的 MLT 中的焦散变异。

mut_lens.h 和 mut_lens.cpp：Veach 的 MLT 中的镜头变异。

mut_manifold.h 和 mut_manifold.cpp：镜面 manifold 扰动策略。

mut_mchain.h 和 mut_mchain.cpp：Veach 的多链扰动策略。

mutator.h：Veach 的双向路径变异方法。

path.h 和 path.cpp：双向路径追踪的路径数据结构。

pathsampler.h 和 pathsampler.cpp：双向路径追踪的路径采样器。

rsampler.h 和 rsampler.cpp：用于 MLT 类型算法的专用采样器实现。

util.h 和 util.cpp：双向方法的一些功能函数。

verification.cpp：用于值验证。

vertex.h 和 vertex.cpp：用于记录顶点的数据结构。

3 4 render

bsdf.h 和 bsdf.cpp：定义 BSDF 等相关功能。

common.h 和 common.cpp：公共内容定义。

emitter.h 和 emitter.cpp：用于发射辐射度/重要性的接口。

film.h 和 film.cpp：Film 基类。

fwd.h：类的预声明。

gatherproc.h 和 gatherproc.cpp：并行光子图构建。

gkdtree.h：优化的 KD Tree，GenericKDTree 结构。

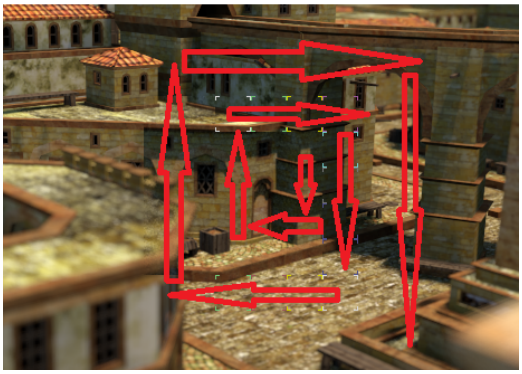
imageblock.h 和 imageblock.cpp：定义图像块。

imageproc.h 和 imageproc.cpp：块图像处理。

integrator.h 和 integrator.cpp：渲染积分器基类。

intersection.cpp：Intersection 表示交点，定义在 render/shape.h 头文件。

irrcache.h 和 irrcache.cpp：辐射度缓存功能。
medium.h 和 medium.cpp：参与介质。
mipmap.h：Mipmap 功能。
noise.h 和 noise.cpp：基于 PBRT 实现的噪声类。
particleproc.h 和 particleproc.cpp：粒子追踪有关。
phase.h 和 phase.cpp：相位函数相关。
photon.h 和 photon.cpp：构建光子图结构。
photonmap.h 和 photonmap.cpp：基于 Henrik Wann Jensen 的光子映射书构建的光子结构。
range.h：工作单元，指定要处理的范围。
rectwu.h 和 rectwu.cpp：工作单元，指定图像中矩形区域。
renderjob.h 和 renderjob.cpp：协调渲染单个图像的过程，多线程执行。
renderproc.h 和 renderproc.cpp：基于采样的积分器的渲染过程。
renderqueue.h 和 renderqueue.cpp：渲染多个场景相关。
sahkdtree2.h、sahkdtree3.h 和 sahkdtree4.h：2/3/4 维的 KD Tree。
sampler.h 和 sampler.cpp：样本生成器的基类。
scene.h 和 scene.cpp：定义场景 Scene 类。
scenehandler.h 和 scenehandler.cpp：任务调度控制相关。
sensor.h 和 sensor.cpp：继承自 AbstractEmitter 的所有传感器的基类，比如投影相机。
shader.h 和 shader.cpp：定义 VPL 风格的渲染器。
shape.h 和 shape.cpp：所有形状类的基类。
skdtree.h 和 skdtree.cpp：SAH KD-tree 加速求交数据结构。
spiral.h：生成要渲染的块螺旋，注意 Mitsuba 渲染时是按照螺旋块的方式进行的：



subsurface.h 和 subsurface.cpp：次表面材质模型的抽象。
testcase.h 和 testcase.cpp：用于测试的函数。
texture.h 和 texture.cpp：定义纹理。
triaccel.h：Ingo Wald 的三角形加速求交方法。
triaccel_sse.h：SSE 加速的三角形相关功能。
trimesh.h 和 trimesh.cpp：三角面片类。
util.h 和 util.cpp：一些功能（比如加载场景）。
volume.h 和 volume.cpp：体数据结构。
vpl.h 和 vpl.cpp：虚拟点光源算法。

四　移植目标

虽然 2 - lib-src/Src 目录下已经包含了 Mitsuba 的全部文件，但是我们还没有加载到自己的工程里，直接添加肯定会出现很多错误。

我们的 Mitsuba 学习过程就是将这些源文件一点一点移植到我们的工程里，而且我们的目标是“能用就用”以及“尽量不删改”，但我们并不会使用 mtsgui 里面的 GUI 显示，我们可能会移植里面的一些内

容。

当前简单做个计划安排：

- 系列 3 我们将学习和使用 film，然后将 film 的输出与我们的 GUI 平台绑定。
- 系列 4 我们将开始移植一个渲染器，从场景加载到渲染结果显示整个流程都跑通。

完成系列 3 和系列 4 以后，就可以说对 Mitsuba 的结构有了足够的了解，之后会介绍各种细节的实现。Mitsuba 里面有大量的与设计模式相关的内容，还有 C++ 的多种复杂特性的实现，因此我们学习 Mitsuba 系列的重点有两个：

- 学习更复杂的渲染算法和材质。
- 学习更高级的 C++ 代码实现。
- 学习更有效的软件设计架构和模式。
- 学习更复杂的代码功能（比如高级的流设计和多线程）。

参考文献

[1] https://www.mitsuba-renderer.org/download.html

[2] https://github.com/mitsuba-renderer/mitsuba

[3] https://dezeming.top/?page_id=1917

[4] https://github.com/mitsuba-renderer/dependencies_win64