Mitsuba 系列 1-初识 Mitsuba

Dezeming Family

2022 年 6 月 17 日

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。

20230113：重新修改了大部分章节，重新确定了 Mitsuba v1 系列的讲解方案。

目录

一　基本介绍

关于 Mitsuba 0.6 的详细介绍可以参考 [1]，这里仅仅说一些比较重要的地方。

Mitsuba 可以说是基于 PBRT 的，但它的实现尤其是可交互性有很多改进。它支持的材质种类远多于 PBRT，而且实现的渲染算法数量也多于 PBRT（比如即时辐射度 (Instant Radiosity)、能量再分配路径追踪 (Energy redistribution path tracer)）。

Mitsuba 目前主要有两个大版本，版本 1 就是常规的渲染算法，版本 2 则是一个可微渲染器，集成了自动微分底层，用于实现渲染中的自动微分。我们本系列文章只考虑研究 Mitsuba v1，而不涉及神经渲染方面的内容。Mitsuba v1 主要在 CPU 上运行（渲染好的图像会借助 OpenGL 显示），Mitsuba v2 可以支持 GPU，但是对 GPU 中的内容支持还尚不全面（而且有些复杂渲染算法无法使用 GPU 渲染）。

Mitsuba v1 的学习基本上是劝退级的，编译过程也时常会遇到各种困难。在这里，我们依然按照研究 PBRT 的思路，一点一点搭建出一个 Mitsuba 渲染器。想研究透一个渲染器，一定是反复研究很多遍，把整个框架体系翻来覆去地看和思考，才能真正理解好、掌握好一个渲染器。

读者在看本系列内容之前，应当先实际操作一下 Mitsuba v1，最好能粗略看一下 [2] 中的 Using Mitsuba 部分（用户手册），至于 Development guide（开发指南）可以先暂时跳过。用户手册的内容主要是教读者怎么编译了解里面的内容，开发指南的内容是教读者怎么去自己实现或者改造一些内容（比如可以自己实现一个改进的渲染器，然后集成到 Mitsuba 中）。

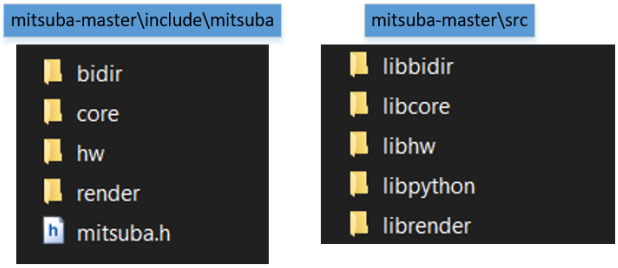
本文意在了解整个 Mitsuba 的结构、场景加载与渲染启动过程、渲染图像的生成和显示以及交互响应。读者需要对 Qt 的信号和槽的机制比较熟悉、对 Qt 常用控件和类有一定的了解（比如 QAction）。同时需要掌握一定的渲染器基础原理，例如需要掌握 PBRT 零基础到吃透系列的《基础知识与代码实战》部分的内容（对 PBRT 的启动过程、结构有一定的了解最好，如果提前学习过文件 parse，即场景加载的方法，那么再学习场景加载会更容易一些）。

在研究 PBRT 时，我们的方法是“能舍就舍（线程并发和内存管理类我们都舍去了）”，而 Mitsuba 中我们采取折中思路，即“好用就用，难用就舍”。如果说 PBRT 系列的目的是为了学习渲染算法的实现和原理，那么 Mitsuba 系列我们的目标更倾向于学习渲染器框架和编程思想。

二　版本下载与初步使用

在 [3] 中可以找到 mitsuba1 的各个版本，我们找到 0.5.0 并下载（这是目前用的最多的一个版本了），这里下载的文件是可执行版本。Mitsuba 官网的链接有时候会失效，我们源码直接从 Github 上下载 Mitsuba 0.6，地址为 [4]。官方文件（目录索引）为 [5]。

源码下载以后，进入 mitsuba-master 目录，可以看到 build、data、doc、include 和 src 这几个文件夹。build 里有编译好的 Visual Studio 2017 项目（没有 CMakeLists 文件，你不能自己编译成 VS2017 工程，但是可以使用别人编译好的工程）；data 里是一些用到的数据；doc 里是解释各个模块的文件，是 Latex 格式的；include 是 Mitsuba 用到的一些 Lib 的头文件，这些 Lib 有些是 Mitsuba 的作者 Wenzel Jakob 团队构建的；src 就是 Mitsuba 的源码，注意源码里面有些文件夹以 lib 开头，这些文件都是与库有关的，与前面 include 是相互对应的：



Mitsuba 使用的 GUI 系统叫做 mtsgui，是基于 Qt 实现的，

在 include 的 core 目录下（mitsuba-master/include/mitsuba/core），有许多最基本的类型，比如 Vector 就定义在了 vector.h 里、Matrix 定义在了 matrix.h 里；还有一些常用的数据结构，比如 kdTree、ocTree

等。注意代码中出现的 MTS 是 Mitsuba 的缩写（没有其他含义）。这里很多代码都用到了 boost 库的功能（一个开源的 C++ 功能库，被称为“准标准库”，编程风格类似于 STL）。

我们自己未必会用到 core 目录下的所有文件功能，例如在 core 目录下，有一个 half.cpp 文件，该文件同名头文件 half.h，这两个文件定义了 16 位浮点数，在头文件里还写了它的精度表示范围。我们一般就是使用 32 位的浮点数，就能满足渲染中绝大部分场景的需求了。

在 [5] 中可以看到各个模块的详细解释，我们用到以后再进行说明，大家可以简单浏览浏览 Modules 底下的各个库中的功能。Mitsuba v1 的支持库被分为五大类，可以参考下表：

表 1: Mitsuba 的 Modules 简介

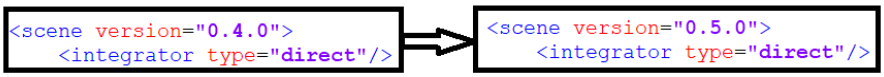
Core library	Mitsuba 中核心功能 API
Render library	Mitsuba 中与渲染有关的 API
Hardware acceleration library	Mitsuba 中与硬件加速有关的 API
Bidirectional support library	Mitsuba 中与渲染的双向方法有关的 API
Python bindings	Mitsuba 中与 Python 绑定有关的类

要想编译 Mitsuba v1，需要 Visual Studio 2013 版本及以下的 Visual Studio 开发环境（虽然 [4] 中有生成好的 VS2017 工程，可以直接使用，但是也需要很多配置，比如一定得需要 Python27 环境。）。这里强烈建议不要自己去 CMake，一开始我认为是自己生成的步骤不太对，但后来我才认识到网上一堆网友都在编译中失败了，最后只能转战 Lunix 系统。在 [7] 中有比较详细的构建过程，有需要可以参考一下。也是因为原版 Mitsuba 确实支持功能太多（比如对 Python 的支持）不好编译，所以本系列中我们尽量去自己移植和构建。

三　Mitsuba 的使用

我们先使用已经编译好的.exe 工程，可以从 [6] 中下载可执行文件（我已经把.zip 文件上传到了网站的本栏目 [9] 下），解压 Mitsuba 0.5.0 64bit.zip 后进入目录。从 [6] 上也可以下载许多场景，比如我们下载一个 medieval.zip 并解压到某目录，在 Mitsuba v1 中选择文件并打开 medieval 中的 xml 文件。

打开文件时会当前场景是 0.4.0 版本的，要不要升级，选择要升级即可。



一开始时默认用 VPL（虚拟点光源技术）渲染的，当我们点 Mitsuba 上的绿色箭头时，就能使用 xml 中定义的渲染算法来渲染了，即 direct（直接光照积分器），它会使用 CPU 多线程渲染，每个线程渲染一个小块。

四　Mitsuba 的场景加载过程

我们的讲解以 Windows 系统为例，Linux 系统和 Mac 除了一些依赖于平台的内容，其他方面都是一样的。

打开 mitsuba-master/src/mtsgui 目录，找到 main.cpp，里面有 main() 函数，这是程序启动加载的函数。这里首先会进行一大堆初始化工作，然后初始化 QtOpenGL 组件（渲染结果显示和交互窗口），之后就创建 Qt 的 MainWindow。等 Qt 界面关闭以后，就清除资源。

```
1    // Initialize the core framework
2    ......
3    // Initialize WINSOCK2 (Windows平台的工具)
4    ......
5    // 建立Qt主界面
```

```
6       mainWindow = new MainWindow();
7           if (mainWindow->initWorkersProcessArgv())
8               retval = app.exec();
9       // Shut down WINSOCK2
10      ......
11      // Shutdown the core framework
12      ......
```

和 PBRT 相似，Mitsuba 的场景也是根据文件来定义的，定义在.xml 格式的文件里，在 [6] 目录下可以看到不少例子，我们以 cbox 场景为参考：

如果我们运行渲染器，点击 File 菜单的 Open，就会调用下面两个函数：

```
1 void MainWindow::on_actionOpen_triggered();
2 void MainWindow::onOpenDialogClose(int reason);
```

第一个函数被触发以后打开文件选择器，文件选择器关闭以后就调用第二个函数，用于加载文件。第二个函数会调用 MainWindow::loadFile() 函数，该函数调用 MainWindow::loadScene() 函数。

MainWindow::loadScene() 函数就是场景加载的函数。里面会调用 SceneLoader 类（这是一个继承自 Thread 类的函数），SceneLoader 类的 SceneLoader::run() 函数（在 sceneLoader.cpp 文件中）会创建 SceneHandler 对象，该对象负责解析 xml 文件。SceneLoader::run() 函数就是启动的场景加载的线程，这里面也包含了文件 parse 并获得场景信息的过程。

在 mtsgui 目录下找到 SceneHandler 类（src/librender/scenehandler.cpp），该类有两个便捷函数 loadScene 和 loadSceneFromString 以不同的方式用于负责 parse 场景（我们可以只参考 SceneHandler::loadScene() 函数）。SceneHandler::loadScene() 会加载场景：

```
1 SAXParser* parser = new SAXParser();
2 parser->parse(filename.c_str());
3 ref<Scene> scene = handler->getScene();
```

parser 是 xercesc（一个开源的 XML 文档解析库）定义的 SAXParser 对象，XML 文件的解析有两种方式，即 XmlPullParser 解析和 SAXParser 解析。SceneHandler 类继承自 xercesc::HandlerBase，通过下面一行来设定 parse 的内容：

```
1     parser->parse(filename.c_str());
```

加载完以后，我们就会得到一个 Scene 对象，这就包含了全部渲染场景（比如相机 (sensors)、光源 (emitters)、物体 (objects)、积分器 (integrator) 等）。

SceneHandler::loadScene() 函数最后会把 Scene 类的实体 scene 返回：

```
1 ref<Scene> scene = handler->getScene();
```

在 MainWindow::loadScene() 函数里，SceneLoader 对象（对象名为 loadingThread）在加载线程结束以后，会返回一个 SceneContext 对象 result：

```
1     result = loadingThread->getResult();
```

SceneContext 对象里存储了很多与渲染结果的需求有关的信息（比如 Scene 对象、分辨率、曝光值等），而且也包括了对 Scene 对象 scene 的指针。

五　Mitsuba 如何启动渲染

在调用完 MainWindow::onOpenDialogClose() 后，场景便被加载。Mitsuba 支持多个场景加载。

在 MainWindow::loadFile() 中，在调用完 MainWindow::loadScene() 函数以后，会更新“最近打开的文件列表”。然后选择新加载的场景：

```
SceneContext *context = loadScene(filename, destFile);
......
addRecentFile(context->fileName);
......
// Select the newly loaded scene
......
updateUI();
```

MainWindow::updateUI() 函数更新 UI 界面。

Mitsuba 可以直接从命令行启动渲染，也可以使用 GUI 交互界面手动选择场景文件，我们分别进行介绍。

直接从命令行启动，即使用 mitsuba.exe。在 main.cpp 的 main 函数里，有这么三行代码：

```
    mainWindow = new MainWindow();
    if (mainWindow->initWorkersProcessArgv())
        retval = app.exec();
```

initWorkersProcessArgv() 函数负责初始化一些参数，如果是从命令行启动渲染，则该函数就会根据命令行接收到的信息来启动渲染，调用 MainWindow::on_actionRender_triggered() 函数，这个函数是用来负责执行渲染的函数，它的最后一行用于调用渲染的线程启动：

```
    context->renderJob->start();
```

renderJob 是 RenderJob 类的对象，继承自 Thread 类，RenderJob 的 run 函数会调用 Scene::render() 函数，Scene::render() 调用积分器类的 render() 函数。

无论 initWorkersProcessArgv() 函数有没有执行渲染，只要参数没有问题，该函数都会返回 true，然后进入 Qt 界面的等待响应程序 app.exec()。

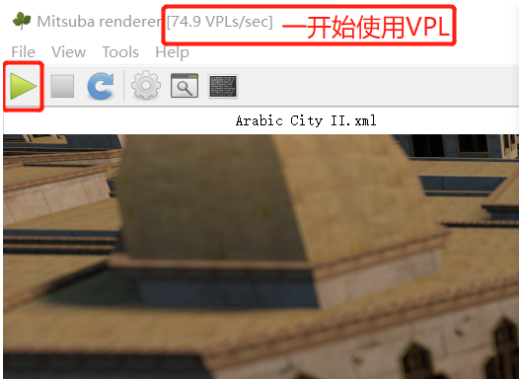
从 GUI 界面手动选择场景渲染,即使用 mtsgui.exe。MainWindow 类中定义了一个对象 ui->glView，这是在 Qt 的.ui 文件里定义的 GLWidget 对象（该类见 glwidget.h）。在 MainWindow 的构造函数里将 GLWidget 中开始渲染的信号与 MainWindow 的启动渲染的槽进行了绑定：

```
    connect(ui->glView, SIGNAL(beginRendering()), this, SLOT(
        on_actionRender_triggered()));
```

当我们加载文件以后，界面是采用 VPL（虚拟点光源算法）方式渲染的（为了查看大致的结果），当我们按下按键 R 或者点击 Mitsuba 界面的绿色箭头，则会调用.xml 场景文件中指定的渲染器来渲染结果：

```
void GLWidget::keyPressEvent(QKeyEvent *event) {
    ......
    if (event->key() == Qt::Key_R) {
        emit beginRendering();
    }
    ......
}
```

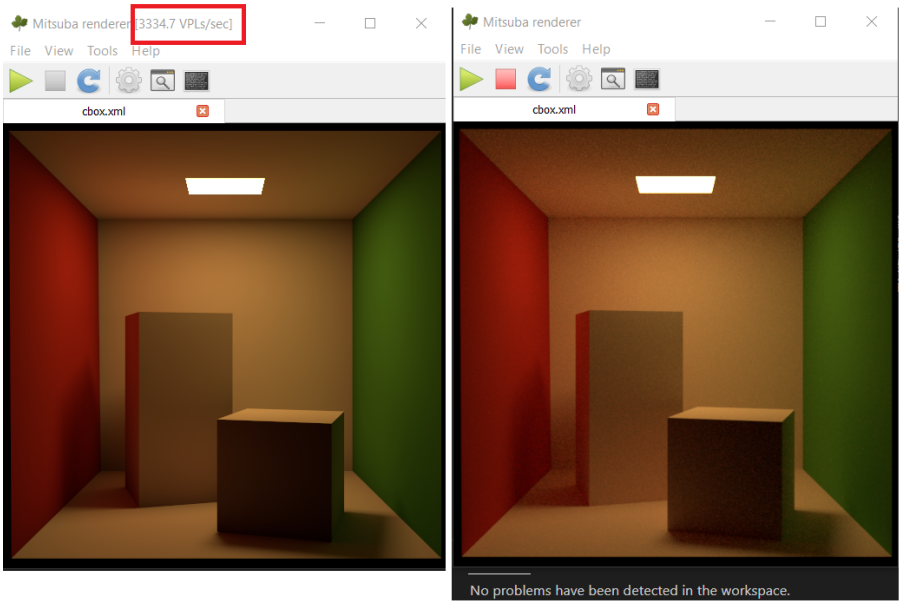
下图中，可以看到一开始是用 VPL 渲染的，当点击绿色箭头以后就用我们定义的渲染器来渲染：



至此，程序就开始渲染场景了。

关于从 GUI 界面调用 VPL 算法渲染。最后，补充一下 mtsgui.exe 启动时，预渲染使用的 VPL 算法的启动流程。调用 VPL 进行渲染的程序在 GLWidget 里，由于我们选择场景文件、更新 GUI（调用 updateUI() 函数）后就自动使用 VPL 渲染，因此可以看出用 VPL 进行渲染的程序应该在一些会自动执行的函数中。我们找到 GLWidget::timerImpulse() 函数，该函数表示每隔一定的时间片就执行一次，该函数最后一行调用 resetPreview() 函数。

GLWidget::resetPreview() 函数内会调用 m_preview->setSceneContext() 函数。m_preview 是 PreviewThread 类的对象，启动 VPL 渲染线程。



上图左是通过加载完场景后直接使用 VPL 渲染的，上图右是通过.xml 里定义的路径追踪渲染器渲染的。

六 Mitsuba 的 GUI 界面交互

有了上一节的基础，Mitsuba 的 GUI 交互看起来就相对比较清晰了。

我们以鼠标按下然后移动为例。见：

```
1 void GLWidget::mouseMoveEvent(QMouseEvent *event);
```

这里包含了当鼠标按下然后移动时渲染器的对变换坐标和相机的调整。比如鼠标中键按下：

```
1     Transform invView = getWorldTransform();
2     ......
3     // 如果中间按键按下
4     if (event->buttons() & Qt::MidButton) {
5         setWorldTransform(invView *
6             Transform::translate(Vector((Float) rel.x(), (Float) rel.y(), 0)
```

```
7                 * m_mouseSensitivity * .6f * m_context->movementScale));
8         didMove = true;
9     }
```

七 Mitsuba 的渲染图像生成与保存

该过程比较复杂，我们只了解下面描述的大致过程即可，以后移植时会再详细介绍需要了解的内容。

我们来看 Scene::render() 函数，该函数调用积分器的 render() 函数。我们以 SamplingIntegrator 积分器为例，SamplingIntegrator::render() 函数建立并行渲染线程：

```
1     ref<ParallelProcess> proc = new BlockedRenderProcess(job, queue, scene->
          getBlockSize());
2     ......
3     sched->schedule(proc);
```

job 是指向 RenderJob 类型的指针。sched（Scheduler 类的对象）通过调用 schedule() 函数来启动渲染过程。

Li 是积分器里渲染全局光照的函数，该函数在 renderBlock() 函数中调用。renderBlock() 是积分器类的一个函数，BlockRenderer::process() 会调用这个函数：

```
1     ImageBlock *block = static_cast<ImageBlock *>(workResult);
2     m_integrator->renderBlock(m_scene, m_sensor, m_sampler, block, stop,
          m_hilbertCurve.getPoints());
```

BlockRenderer 类会被上面的 BlockedRenderProcess 类创建。

渲染结束以后，Scheduler::releaseWork() 函数调用 BlockedRenderProcess::processResult() 函数，将 ImageBlock 类型的对象赋给 Film 对象：

```
1     const ImageBlock *block = static_cast<const ImageBlock *>(result);
2     m_film->put(block);
```

Film 是胶片的意思，意为渲染图像的输出。BlockedRenderProcess::bindResource() 函数中，m_film 与 sensorResID 索引的传感器类（也就是相机类）绑定。

RenderJob::run() 函数在调用完 Scene::render() 函数后，就会调用后处理函数 Scene::postprocess()，该函数会调用 Sensor::getFilm()->develop() 函数，用于保存渲染结果。我们以 HDRFilm::develop() 函数为例，该函数将 m_storage（ImageBlock 类的对象指针）转为 Bitmap 类的对象指针，然后再保存输出为各种我们设定的目标图像格式。

显示到屏幕上的函数在 GLWidget::paintGL() 里由 Qt 的 updateGL() 函数自动调用，来将渲染结果输出到屏幕上。有两种模式：

```
1 EPreview：预观察模式（使用VPL来渲染）
2 ERender：开始渲染模式（使用自己定义的渲染器来渲染）
```

这两种模式将 buffer（GPUTexture 类的对象）赋予不同的内存来显示。然后 OpenGL 将 buffer 的内容显示到屏幕上。

八 小结

当我们要去学习 Mitsuba 渲染器时，我们完全不需要去把各种 GUI 和 Buffer 的转换之类的内容全都搞懂，因为交互系统在修修补补中增加了很多编程模式和组织方法，变得比较庞大，也更稳定，但是也更

复杂。本文的目的在于了解 Mitsuba 渲染器的基本结构和流程，和《PBRT 系列 1-文件加载和设定》一样，使得用户能够更好地把握渲染器的结构。

本系列的意义在于使用户将 Mitsuba 中复杂的内容层层剥离出来，得到一个相对小巧但是完全符合 Mitsuba 功能的渲染器。在研究 PBRT 时，我们的方法是“能舍就舍（线程并发和内存管理类我们都舍去了）”，而 Mitsuba 我们采取折中思路，“好用就用，难用就舍”，旨在更好地学习 Mitsuba v1 的渲染器框架和编程思想。

本系列每本小书都会附带源码，而且源码除了给出的依赖库和 Qt 以外不会需要过多其它库（我们会在下一本电子书介绍）。

参考文献

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

[2] https://www.mitsuba-renderer.org/docs.html

[3] https://www.mitsuba-renderer.org/releases/

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

[5] http://mitsuba-renderer.org/api/index.html

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

[7] https://banbao991.github.io/2021/04/26/CG/mitsuba/mitsuba-0-6-installation/

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

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