Mitsuba 系列 4-Mitsuba 的文件解析

Dezeming Family

2023 年 1 月 25 日

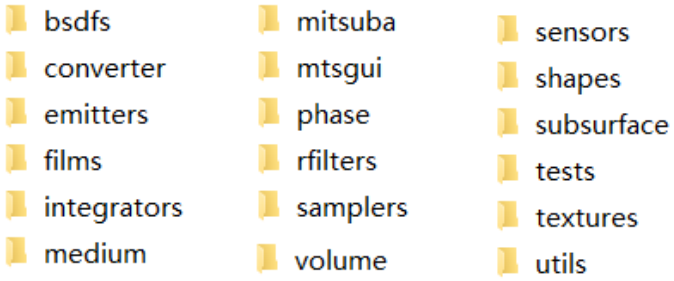
DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列书，可以从我们的网站 [https://dezeming.top/] 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

本文我们学习和移植 Mitsuba 的 XML 场景文件解析。

目录

一　Src 目录下的文件功能

Src 目录下有很多子目录：



我们本节介绍一下这些子目录中文件的功能，我们按照不同目录的类别分类介绍。

表面材质与参与介质

bsdfs 目录与各种不同的表面散射材质相关。
subsurface 与次表面散射和体渲染有关。
phase 里有 medium 中会用到的多种相位函数。
medium 表示参与介质需要用到的功能。
volume 中表示参与介质。

物体的形状与纹理

shapes 与形状有关，比如球体，三角形。
textures 里包含了各种纹理类型。

光源

emitters 表示光源。

渲染与重建滤波器

films 目录下是胶片类，我们已经移植过了。
rfilters 里面是用于胶片的滤波器，我们也已经移植过了。
integrators 表示渲染积分器类。
sensors 与相机有关。

随机数采样器

samplers 与低差异序列采样器有关，比如 halton 和 sobol。

其他功能

除了 mtsgui.exe 以外，Mitsuba 工程还能生成四个可执行程序：mitsuba.exe、mtsimport.exe、mtssrv.exe 和 mtsutil.exe。

converter 用于将其他格式的资源（比如.obj 格式的资源）转换为 Mitsuba 格式。该目录下的文件生成 mtsimport.exe。

mitsuba 目录生成 mitsuba.exe、mtssrv.exe（与分布式集群运行有关）和 mtsutil.exe，使得程序可以从命令行执行。

mtsgui 目录生成 mtsgui.exe。

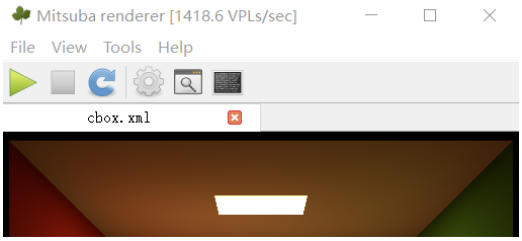
tests 目录包含了一些类和功能的测试。

utils 目录下包含了一些实用功能。

二　mtsgui 程序结构

如果尝试运行 mitsuba 0.5 的可执行程序，就会发现当加载完场景以后，一开始是使用的 VPL 算法进行的渲染，得到一个初步的外观。

当点击 Mitsuba GUI 界面上的绿色箭头（文件加载成功前是灰色的）时，就会开始使用我们的场景文件中定义的渲染器来进行渲染。



在 main.cpp 中进行了一些初始化工作，然后创建 Qt 的 MainWindow，在 MainWindow 的构造函数中，这个地方都很重要：

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

调用 on_actionRender_triggered() 槽程序。

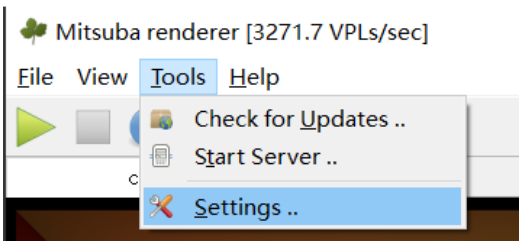
注意有很多设置都是在 ui 文件里的，因此无法直接从源码中找到，比如 GUI 界面上的绿色箭头。该箭头定义在 mainwindow.ui 里，是显示在 ToolBar 中的 QAction，其作用也是点击后响应 on_actionRender_triggered() 槽程序：

```
<action name="actionRender">
 <property name="icon">
  <iconset resource="resources.qrc">
   <normaloff>:/resources/play.png</normaloff>:/resources/play.png</
       iconset>
 </property>
 <property name="text">
  <string>Render</string>
 </property>
 <property name="toolTip">
  <string>Start rendering</string>
 </property>
</action>
```

on_actionRender_triggered() 槽程序会创建 renderJob 然后启动 renderJob。

MainWindow::updateUI() 是用来更新 GUI 界面的，比如 GUI 一启动时由于还没有加载场景，所以很多功能键都是灰白的，当加载好场景以后，根据是否加载成功来使得界面上的功能键变色，如果开始使用用户设定的渲染器进行渲染时，则该绿色箭头又变为灰色（不能重复点击）。

点击下面的设置就会生成 RenderSettingsDialog，这里可以修改渲染中的一些参数，并根据新的设置来生成场景：



三　xml 文件 parser

MainWindow::loadFile() 用于加载场景 xml 文件，该函数内会调用 MainWindow::loadScene() 函数以加载场景到 SceneContext 对象中。SceneContext 是定义在 mtsgui 目录下的 common.h 头文件中的一个类，用来记录场景的信息。

MainWindow::loadFile() 加载完文件以后，会默认使用 VPL（虚拟点光源照明算法）来预渲染。加载文件后，会将 context->renderJob 设置为 Null，表示目前没有正在渲染的线程（预渲染不算）。注意该函数中会调用 MainWindow::updateUI() 更新 GUI 界面。

我们暂不介绍加载文件与调用 VPL 之间的信号关系（也是因为我以前都是直接使用和修改核心代码，没有了解过这些内容，但它们大致是通过某些信号机制进行连接的，比如加载完场景就会触发 GLWidget 渲染 VPL 的机制）。

先把 sceneloader.h 和 sceneloader.cpp 以及 common.h 加载到工程里，此时编译会有一些找不到的符号链接，因为有几个 SceneContext 的函数定义在了 mainwindow.cpp 文件中，我们并不使用 Mitsuba 的 mainwindow，所以我们把这个文件里的几个函数都拷贝到了 sceneloader.cpp 文件里。

MainWindow::loadFile() 调用 MainWindow::loadScene() 函数，MainWindow::loadScene() 函数会调用 loadingThread = new SceneLoader()，创建加载场景的线程。SceneLoader::run() 有两种功能，它既可以支持打开显示一张图像（我也不知道为什么非要支持这种功能），也可以支持加载场景：

```
1 if (suffix == "exr" || suffix == "png"  || suffix == "jpg" || suffix == "
    jpeg" ||
2 suffix == "hdr" || suffix == "rgbe" || suffix == "pfm" || suffix == "ppm"){
3     // 表示我们当前打开的文件是一张图像
4     ......
5 } else{
6     // 加载场景
7     ......
8 }
```

对于加载场景，首先需要加载一个固定设置的文件：

```
1 fs::path schemaPath = m_resolver->resolveAbsolute("data/schema/scene.xsd");
```

然后调用加载场景的代码，主要代码如下：

```
1 parser->parse(filename.c_str());
2 ref<Scene> scene = handler->getScene();
3 scene->setSourceFile(filename);
4 scene->setDestinationFile(m_destFile.empty() ? (filePath / baseName) :
    m_destFile);
5 scene->initialize();
6 //开始加载每个组件，比如渲染积分器、光源等
7 ......
```

SceneHandler 继承自 xercesc::HandlerBase，与 SAXParser* parser 配合使用，作为场景加载器：

```
1 parser->setDoNamespaces(true);
2 parser->setDocumentHandler(handler);
3 parser->setErrorHandler(handler);
```

场景加载完以后，赋值给 SceneContext* m_result：

```
1 m_result->scene = scene;
```

```
2 m_result->sceneResID = Scheduler::getInstance()->registerResource(scene);
```

然后回到 MainWindow::loadScene() 函数：

```
1 // 场景文件加载结束以后，赋值给SceneContext *result
2 result = loadingThread->getResult();
```

四　根据场景文件构建场景

我们已经了解了那些代码用来加载场景，那么本节就介绍如何根据加载出的场景参数来构建场景实例。这会对我们移植程序非常有帮助。

Scene 类继承自 NetworkedObject，该类里面有指向各个所需场景组件的指针，比如：

```
1 ref<ShapeKDTree> m_kdtree;
2 ref<Sensor> m_sensor;
3 ref<Integrator> m_integrator;
4 ref<Sampler> m_sampler;
5 ref<Emitter> m_environmentEmitter;
6 ref_vector<Shape> m_shapes;
7 ref_vector<Shape> m_specialShapes;
8 ref_vector<Sensor> m_sensors;
9 ref_vector<Emitter> m_emitters;
```

我们注意到在 SceneLoader::run() 中有加载场景文件后得到的 Scene：

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

在 SceneHandler::loadScene() 函数中也是会使用这个代码来加载场景：

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

只不过代码里（据我了解）没有通过 SceneHandler::loadScene() 来加载场景。

SceneHandler 覆盖了很多函数，这些函数会与 SAXParser 中的功能相匹配来解析文件，下面几个函数很重要：

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

parse() 函数就是用来加载场景文件，并初始化场景的。

4 1　文件加载基本概念

关于文件 parse 的过程，Xml 解析用到了 Xerces-C++，可以参考我们的《Xerces-C++ 使用指南》，里面有详细的过程，这里只简单说一些比较重要的地方。

这两句代码用于初始化和终止 XML 解析库 Xerces，可以在 SceneHandler 类的函数中找到对它们的调用：

```
1 XMLPlatformUtils::Initialize();
2 XMLPlatformUtils::Terminate();
```

Xerces 有三个重要的模块，分别是 DOM API、SAX2 API 和 SAX API，SAX 可以解析很大的 XML 文件，但是没有写入功能。因此，在场景文件版本较低时，需要升级（upgrade），升级时是使用 Qt 的 DOM 模块进行改写的。Xerces 的 SAX 负责读取和解析场景文件。

SceneHandler::startElement() 和 SceneHandler::endElement() 两个函数中，startElement 是当 parse 到某个元素时执行的，而 endElement() 是当这个元素被 parse 完时执行（此时会在这个函数里创建对象）。比如对于下面的场景文件：

```
<?xml version="1.0" encoding="utf-8"?>
<scene version="0.5.0">
    <integrator type="path"/>
    <sensor type="perspective">
    </sensor>
</scene>
```

当加载到 <integrator type="path"/> 时，就初始化渲染积分器为路径积分器；当加载到 </scene> 就说明整个场景被加载完了，那么就初始化这个场景。

4 2 检测当前对象类型

当 parse 检测到一些 mitsuba 特定对象（插件对象，比如'path'、"）时，需要执行创建对象的工作，这里的工作是在 SceneHandler::endElement() 里进行的，因为当运行到 SceneHandler::endElement() 时说明这个对象相关的属性都已经加载完了，所以就可以创建对象了。

当 parse 执行 endElement() 时，执行下面几句代码：

```
std::string name = transcode(xmlName);
ParseContext &context = m_context.top();
// 插件类型
std::string type = boost::to_lower_copy(context.attributes["type"]);
context.properties.setPluginName(type);
// 如果属性中找到了id，就存储id
if (context.attributes.find("id") != context.attributes.end())
    context.properties.setID(context.attributes["id"]);
// 找到对应的Tag
TagMap::const_iterator it = m_tags.find(name);
if (it == m_tags.end()) {
    XMLLog(EError, "Unhandled tag \"%s\" encountered!", name.c_str());
}
```

这里的插件类型，就是比如'perspective' 表示透视相机，'path' 表示路径追踪器。Tag 就是 scenehandler.h 中的枚举 enum ETag，表示场景文件中所有可能遇到的标志。比如 EIntegrator, EEmitter, ESensor 这种渲染中使用的插件对象，或者是 EInteger, EFloat 这种数据对象。

如果属性中找到了属性'id'，意味着 context.attributes.find("id") 返回的结果不是末尾（返回末尾索引表示没有找到）。

4 3 Parse 栈

ParseContext 是一个栈结构。当 Parse 到一个对象时，就使其进栈。SceneHandler::startElement() 函数中有该过程：

```
    m_context.push(context);
```

当执行完 SceneHandler::endElement()，表示栈顶的元素已经被使用完了，因此弹栈：

```
1     m_context.pop();
```

比如对下面的内容进行 Parse：

```
1  <shape type="obj">
2      <string name="filename" value="meshes/cbox_luminaire.obj"/>
3      <transform name="toWorld">
4          <translate x="0" y="-0.5" z="0"/>
5      </transform>
6
7      <ref id="light"/>
8
9      <emitter type="area">
10         <spectrum name="radiance" value="400:0, 500:8, 600:15.6, 700:18.4"/>
11     </emitter>
12 </shape>
```

Parse 到第一行时，shape 属性进 m_context 栈，位于栈顶。

Parse 到第二行时，在 string 属性在 startElement() 中进 m_context 栈，在 endElement() 中出 m_context 栈。

Parse 到第三行时，transform 属性进栈，此时它位于 m_context 栈顶。然后当 Parse 到第五行时，transform 属性出 m_context 栈。

在 endElement() 函数中操作的栈顶即是当前要处理的属性。处理完以后在 endElement() 结尾弹栈。

4 4 对象 ID 与对象创建

有些插件对象是有名称的，即 id，比如下面的这个 diffuse 的 id 就是'light'：

```
1  <bsdf type="diffuse" id="light">
2      <spectrum name="reflectance" value="400:0.78, 500:0.78, 600:0.78, 700:0
           .78"/>
3  </bsdf>
```

这是为了方便其他数据共享该对象，比如场景中同时有两个这种类型的光源（漫反射光源，非黑体光源），那么它们就都可以引用该属性。引用方式是：

```
1  <shape type="obj">
2      <string name="filename" value="meshes/cbox_luminaire.obj"/>
3      <transform name="toWorld">
4          <translate x="0" y="-0.5" z="0"/>
5      </transform>
6      <ref id="light"/>
7      <emitter type="area">
8          <spectrum name="radiance" value="400:0, 500:8, 600:15.6, 700:18.4"/>
9      </emitter>
10 </shape>
```

注意漫反射光源既可以是照明物 (emitter)，也可以是反射物 (reflectance)，因为更强的光照射到漫反射光源时，会使得漫反射光源更亮（黑体光源则会吸收所有照射到上面的光）。

SceneHandler::m_namedObjects 里存储了所有对象的'id' 属性，用于其他对象索引到对应的对象。在 SceneHandler::endElement() 最后一行里，将新出现的有'id' 的属性添加到 m_namedObjects 里：

```
if (id != "" && name != "ref") {
    if (m_namedObjects->find(id) != m_namedObjects->end())
        XMLLog(EError, "Duplicate ID '%s' used in scene description!", id.
            c_str());
    (*m_namedObjects)[id] = object;
    if (object)
        object->incRef();
}
```

五　我们给出的本文代码结构

在我们的实现中（代码【4 - scene-parser】中），Mitsuba 中的 MainWindow::loadScene() 函数被我们实现在了 IMAGraphicsView::loadScene() 中。

我们的代码由于没有插件，所以很多对象没法创建，如果直接使用原来的代码就会报错，所以我们屏蔽了一些内容，屏蔽掉的内容是用下面的标志注释掉的：

```
    // Dezeming Cancel
    /* ..... */
```

有一句代码由于如果不初始化 object 就不会执行，因此我们修改为了 if(1)：

```
// Dezeming Cancel
//if (object != NULL || name == "null")
if(1) {
    ......
}
```

运行该代码，会输出 cbox.xml 里面全部的对象，并对引用'id' 对象的对象输出所有的引用'id' 名。

如果在运行 Debug 模式时出现 QString 报错等相关问题，这是由于我们的调试显示类 DebugText 和 Mitsuba 的一些输出格式不兼容导致的。在下一本小书中我们把打印调试的功能都转移到 cout 输出到控制台上。

六　本文小结

本文从 1 月 25 号断断续续地写到 4 月，期间因为投论文耽误了一段时间，现在总算又多出来一点儿空闲时间，于是把本文进行了完结。应该算是把 Mitsuba 的文件加载过程讲得比较清楚明白了。

下一本小书我们就会把里面的多个插件源文件加载到工程里，修改成我们的接口，然后实现一个光线跟踪器，这样，Mitsuba 的平台就算搭建好了。把这个平台能自己搭建一遍，对掌握其结构、自己实现渲染器都是非常有好处的。而且我搭建这个代码也是为了解决 Mitsuba 对用户设置开发环境不太友好这个问题。

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