Mitsuba 系列 3-film 类的移植

Dezeming Family

2023 年 1 月 17 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 Mitsuba 的代码结构

我们参考 MitsubaProject/Mitsuba 0.5.0 中的可执行程序所在目录，有一个子目录 plugins，里面有一堆 dll，这些都是 Mitsuba 库生成的插件。由此可知，Src 的目录下几乎每个 cpp 文件都会生成对应的插件。

1 1 编译 boost 库

由于给定的 boost 库有一些问题（mitsuba 工程给出的附带的 boost 库不全，没有 debug 模式的库），我们需要自己编译 boost 库，在 [2] 下载 boost，我下载的是 boost 1.77。解压后运行 bootstrap.bat，得到 b2.exe，在当前目录下打开 cmd 命令行，执行：

```
b2 install --toolset=msvc-14.1 --build-type=complete --prefix="D:/DevTools/boost-generate" link=shared runtime-link=shared runtime-link=static threading=multi
```

其中，"D:/DevTools/boost-generate" 是我们的生成目录。编译 boost 时的参数选项（摘录自 [4]）：

```
-without/with：选择不编译/编译哪些库。默认是全部编译
bjam --show-libraries 查看boost包含库
-toolset：指定编译器
link：生成动态链接库/静态链接库 static只会生成lib文件 shared会生成lib文件和dll文件
runtime-link：动态/静态链接C/C++运行时库。同样有shared和static两种方式
threading：单/多线程编译。现在基本都是multi方式了
address-model：64位平台还是32位平台，不填就两种平台的库都会编译。
debug/release：debug版本，release版本，不填就两种版本的库都会编译
```

之后把下面的几个 lib 和 dll 文件放到之前的 OriginFiles/dependencies/lib 目录下（下面的名称对于相应的 lib 和 dll 文件都是相同的）：

```
    boost_chrono-vc141-mt-x64-1_77
    boost_chrono-vc141-mt-gd-x64-1_77
    boost_filesystem-vc141-mt-x64-1_77
    boost_filesystem-vc141-mt-gd-x64-1_77
    boost_system-vc141-mt-x64-1_77
    boost_system-vc141-mt-gd-x64-1_77
    boost_thread-vc141-mt-x64-1_77
    boost_thread-vc141-mt-gd-x64-1_77
```

并且在 CMakeLists 上修改过来。

由于一些版本问题，Mitsuba 的 core/platform.h 需要将下面两行注释掉：

```
//#define snprintf _snprintf
//#define vsnprintf _vsnprintf
```

否则会显示在命名空间 std 中没有名为 _vsnprintf 的成员 [6]。

还需要在 CMakeLists.txt 的预定义宏上加：

```
BOOST_ALL_NO_LIB
```

这样就保证链接的库名称为 boost_xxxxx 而不是 libboost_xxxxx（参考自 [5]）。

1 2 生成类

Mitsuba 的类和功能通过 MTS_EXPORT_PLUGIN 定义输出 dll 接口，会输出这种形式的接口（以 HDRFilm 为例）：

```
void *CreateInstance(const Properties &props) {
    return new HDRFilm(props);
}
const char *GetDescription() {
    return "High dynamic range film";
}
```

使用 MTS_IMPLEMENT_CLASS_S 宏生成的类：

```
Object *__HDRFilm_unSer(Stream *stream, InstanceManager *manager) {
    return new HDRFilm(stream, manager);
}
// false表示非抽象类
Class *HDRFilm::m_theClass = new Class(HDRFilm, false, Film, NULL, (void *)
   &__HDRFilm_unSer);
const Class *HDRFilm::getClass() const {
    return m_theClass;
}
```

由于我们意图将所有文件都作为目标工程的一部分，因此这样会造成每个类都定义了同名同参数的 CreateInstance 函数，故我们在源码中将类似下面的语句给注释掉：

```
//MTS_EXPORT_PLUGIN(TiledHDRFilm, "Tiled high dynamic range film");
```

并且新建 annotations.cpp 文件，将 annotations.h 中的定义转移到 annotations.cpp 中，这也是避免编译时的重复定义。

1 3 基类与子类之间的关系

Object 是所有 Mitsuba 类的基类。

SerializableObject 和 InstanceManager 继承自 Object，用于对象的序列化。关于这两个基类的功能我们本小节末尾有介绍。

Random、WorkProcessor、IrradianceCache、PhotonMap 等类都是继承自 SerializableObject。这些类的特点是既可以通过常规方法来创建，也可以用序列化的方式来用二进制流数据赋值。

ConfigurableObject 继承自 SerializableObject，支持从 Properties 实例中构建。Properties 也是一个基类，用于构造 ConfigurationObject 子类的关联参数映射（换言之，就是存放数据的类）。

Film、NetworkedObject、ReconstructionFilter、BSDF、PhaseFunction、Sampler、Shape、AbstractEmitter 等都是继承自 ConfigurableObject。这些类的特点是不但可以用序列化的方式赋值，也可以预先指定相关参数，比如指定 Film 的长宽、指定 BSDF 是透射还是反射类型等，预先指定的信息存放在 Properties 里。

HWResource 是一个基类，实现提供了对也能够在 GPU 上运行的功能的支持。默认情况下，方法“createShader”只返回 NULL，这意味着 BSDF/光源/纹理/等尚未移植到基于 GPU 的渲染器中。

Stream、HemisphereSampler、ParallelProcess、Scheduler、WorkUnit、WorkResult、Mutator、SpecularManifold、PathSampler、Appender、Bitmap、ChiSquare、Formatter、FileResolver、Mutex、WaitFlag 等直接继承自 Object。这些类的特点是不需要序列化操作，也不用提前指定什么信息，因此直接继承自基类即可。

我们本文和下一本电子书都不会涉及太多类之间的关系和结构，仅仅只是能够使用即可。

序列化

对象是类的实例 (instance)，程序在使用对象的过程是先加载类来创建类的一个实例（即对象），然后初始化，之后就可以调用对象的属性和方法了，所有的这一切都是在内存中进行的，也就是说，每次使用对象都是重新创建。

序列化把一个对象的状态写入一个字节流，这样就能把对象存储在磁盘上，使用时可以直接加载到内存中。比如我们已经构建了一个场景 KD-Tree，我们不希望再使用时又重新构建，因此我们就可以使用序列化操作。

对于 InstanceManager 类，当将复杂的对象图 (object graph) 序列化为二进制数据流时，实例管理器会对数据流进行注释，以避免将对象序列化两次或陷入循环依赖关系。这允许序列化任意连接的对象图。（想进一步了解原因的人可以搜索“如何将对象图序列化”。）

二 Film 类

Film 类继承自 ConfigurableObject，一共有四个派生类：LDRFilm，HDRFilm，MFilm，以及 TiledHDRFilm。

我们本节介绍 Film 基类的各个函数，以及与一些其他类之间的联系，下一节再分别介绍四种派生类的功能。

2 1 Film 基类的函数

Film 类中，getSize() 返回传感器分辨率。但可能需要显示的图像仅仅是全部分辨率中的一个区域，因此 getCropSize() 返回要显示/输出的图像区域（即要渲染的图像区域）。getCropOffset() 返回其在传感器分辨率上的偏置。

下面几个函数名所代表的函数都是纯虚函数，需要在子类实现：

```
1  clear
2  put
3  setBitmap
4  addBitmap
5  setDestinationFile
6  develop
7  destinationExists
```

addBitmap() 函数仅仅在双向路径追踪方法时才会用到（采样相机顶点时得到的 Light map 与其他部分进行合并），故我们本文暂时不做讲解。在介绍派生类的实现时我们再介绍这些函数的功能。

下面几个函数都跟序列化操作等有关，我们暂时不需要了解：

```
1  addChild
2  configure
3  serialize
```

下面几个函数与重建滤波器值相关：

```
1  hasHighQualityEdges
2  hasAlpha
3  getReconstructionFilter
```

hasHighQualityEdges() 函数表示对于边缘区域而言，是否会采样图像外部使得边缘图像质量更高一些。不过对于 box 滤波器而言是没有什么区别的。

2 2 ImageBlock 类

ImageBlock 类表示图像块，它能够给 Film 中的存储区赋值。Film（Film 的派生类）中的数据也是用 ImageBlock 来保存的。

ImageBlock 继承自 WorkResult，WorkResult 继承自 Object。WorkResult 也可以从流中初始化，但是它跟 SerializableObject 类的区别在于它实现的接口是 load 和 save，表示从流中加载数据/将数据输出到流。

ImageBlock 内的数据是存储在 Bitmap 对象中的，Bitmap 直接继承自 Object。Bitmap 就是按每个像素来存储图像的类，可以存储各种类型的图像，像素类型由 EComponentFormat 枚举来表示，比如 EUInt8、EFloat32 等。

Bitmap 主要可以从流中赋值，其重载函数虽然包含了从路径读取，但是读取的文件应当是存储流的文件，而不是一些图像格式（比如 PNG 图像）。要想使用里面的 PNG 和 OpenEXR 等读写功能，需要预定义下面几个宏，定义在 CMakeLists.txt 中：

```
1    MTS_HAS_LIBPNG
2    MTS_HAS_LIBJPEG
3    MTS_HAS_FFTW
```

我们没有定义 MTS_HAS_OPENEXR，是因为 Mitsuba 给出的 OpenEXR 依赖库不全，我也暂时不太想再编译一遍，所以干脆不添加 OpenEXR 支持。

三 Film 类的基类

本节介绍四个 Film 类的派生类，关于 hdr 和 ldr 之间的色调映射转换等内容我们放在下一节再介绍。

3 1 LDRFilm

LDRFilm 类最简单，大部分函数的作用都非常清晰明了，我们只介绍一下两个重载的 develop 函数的功能。先看第一个：

```
1 bool develop(const Point2i &sourceOffset, const Vector2i &size, const
    Point2i &targetOffset, Bitmap *target) const
```

该函数将 film 中的一块子区域（坐标从 sourceOffset 开始）赋值到 target 的目标区域（坐标从 targetOffset 开始）上。

第二个函数把 Film 根据先前定义好的文件名直接写入到文件中：

```
1 void develop(const Scene *scene, Float renderTime);
```

LDRFilm 和 HDRFilm 都可以在这个 develop 函数里使用 annotate() 函数将 metadata 标注到图像文件中（相当于图像文件的一些额外备注信息，比如这张图是何时进行渲染的）。

3 2 HDRFilm

HDRFilm 比 LDRFile 多了两个 vector：

```
1 std::vector<Bitmap::EPixelFormat> m_pixelFormats;
2 std::vector<std::string> m_channelNames;
```

EPixelFormat 表示像素实际意义，比如是表示单通道 luminance，还是表示 3 通道 XYZ，又或者是 4 通道 RGBA 等。这点要区别于描述像素每个通道的数值类型 EComponentFormat，比如 8 位无符号整数、32 位浮点数。

在 HDRFilm 的构造函数中，根据 tokenize() 函数（根据逗号来分割为多个字符串）从 props 中获取 pixelFormats 数组。channelNames 也是从 props 中获取的每个通道的名字。之后，利用 pixelFormats 和 channelNames 分别对 m_pixelFormats 和 m_channelNames 进行赋值（m_channelNames 相当于每种格式的每个通道都给一个名称，比如假设 pixelFormats[2] 为 ERGB，channelNames[2] 为"ccc"，那么像素的四个通道名就会命名为"ccc.R","ccc.G","ccc.B"）。

HDRFilm 的输出格式 m_fileFormat 有三种可能，分别是 OpenEXR（默认格式）、RGBE () 和 PFM（Portable Float Map format）。

参考 Sensor::configure() 函数可知，传感器中的 m_film 会默认设定为 HDRFilm 类型。

3 3 MFilm

MFilm 用于将 Spectrum、RGB、XYZ 或 luminance 值作为矩阵导出到 MATLAB 或 Mathematica ASCII 文件或 NumPy 二进制文件。在一些计算机视觉应用中，这样会比较方便将 Mitsuba 的渲染结果用于其他程序。

该类目前不太常用，因此我们暂不介绍该类的具体功能。

3 4 TiledHDRFilm

TiledHDRFilm 非常类似于 HDRFilm，但是它会直接将渲染结果存储到磁盘里，而不是存储到内存，因此它可以支持渲染超大的图像，比如 $100K \times 100K$ 的大图。

该类目前不太常用，因此我们暂不介绍该类的具体功能。

四 格式之间的转换

格式之间的转换涉及比如色调映射之类的内容，稍微有点复杂，之前我也是花了一段时间才弄明白它们之间的关系。

4 1 色调映射

先看 LDRFilm 类。色调映射有两种，见 ETonemapMethod 枚举，分别是 Gamma 色调映射（曝光和 Gamma 校正）和 Reinhard 色调映射（全局自动色调映射方案）。曝光值 m_exposure 只有在进行 Gamma 色调映射时才有用。m_gamma 为-1 时表示 sRGB，比如在读取图像时，如果默认图像时 sRGB 格式存储的，则读取时就把 m_gamma 设为-1。

Bitmap 中也存储了一个 gamma 值，在调用 convert 时，如果传入的 m_gamma 值和 Bitmap 对象的 m_gamma 值相同，则无需进行转换：

```
1 bitmap = bitmap->convert(m_pixelFormat, Bitmap::EUInt8, m_gamma, multiplier)
    ;
```

Bitmap::convert() 会调用 FormatConverter::convert() 函数。

4 2 HDR 数据转换为 LDR 并显示

我们的显示器是 LDR 显示器，但光传输算法计算得到的值是 HDR 值，所以需要转换为 LDR 值再显示到屏幕上。

如果我们定义更严格的相机，那么可以设定相机响应曲线，将自然界高动态范围的光经过相机曲线映射为相机获取值，该值可以是 HDR 值，也可以是 LDR 值。

SamplingIntegrator::renderBlock() 中，put 函数将渲染得到的 Li() 值保存到 ImageBlock 类的 bitmap 中：

```
1 block->put(samplePos, spec, rRec.alpha);
```

绘制时，见 GLWidget::paintGL() 函数：

```
Bitmap *source = m_context->framebuffer;
float *sourceData = source->getFloat32Data();
uint8_t *targetData = (uint8_t *) m_fallbackBitmap->getData();
for (int y=0; y<source->getHeight(); ++y) {
    for (int x=0; x<source->getWidth(); ++x) {
        ......
    }
}
```

在省略号中的代码里可以看到，如果转换为 8 位浮点数时的值太高，就会赋值为 255：

```
const float invGammaValue = 0.45455f;
*targetData++ = (uint8_t) std::max(std::min(std::pow(*sourceData++,
   invGammaValue) * 255.0f, 255.0f), 0.0f);
```

五 bitmap 类的使用

5 1 加载数据到 bitmap 中

在 GLWidget::initializeGL() 函数可以看到把图片加载到 Bitmap 中的流程：

```
QResource res("/resources/logo.png");
SAssert(res.isValid());
ref<Stream> mStream = new MemoryStream(res.size());
mStream->write(res.data(), res.size());
mStream->seek(0);
ref<Bitmap> bitmap = new Bitmap(Bitmap::EPNG, mStream);
```

需要 Qt 加载资源再转化到流中。

为了方便，我们不使用流来创建，而是直接读取和生成。使用 stb 库（见 3 - film_test/Include/stb-lib 目录）读取文件，然后直接转化为 bitmap 对象。见我们的 MainGUI/IMAGraphicsView.cpp 文件：

```
// 使用stb-lib来读取图像
int width_img, height_img, nrChannels_img;
unsigned char *data = stbi_load("./Icons/cat-1.png", &width_img, &height_img
    , &nrChannels_img, 0);
if (nrChannels_img == 4) {
    using mitsuba::Bitmap;
    using mitsuba::Vector2i;
    using mitsuba::FileStream;
    ref<FileStream> fsstream;
    Vector2i size(width_img, height_img);
    // 加载到bitmap流
    mitsuba::ref<mitsuba::Bitmap> bitmap = new mitsuba::Bitmap(Bitmap::ERGBA
        , Bitmap::EUInt8, size, nrChannels_img, data);
}
```

5 2 保存图像

若要保存图像，必须使用流的功能，但要注意 Mitsuba 里很多类是必须要在使用前先静态初始化的，否则就会报错。在 MainGUI/IMAGraphicsView.cpp 文件中可以看到我们的初始化项：

```
// mitsuba 静态组件初始化
Class::staticInitialization();
Object::staticInitialization();
PluginManager::staticInitialization();
Statistics::staticInitialization();
Thread::staticInitialization();
Logger::staticInitialization();
FileStream::staticInitialization();
Spectrum::staticInitialization();
Bitmap::staticInitialization();
Scheduler::staticInitialization();
SHVector::staticInitialization();
SceneHandler::staticInitialization();
```

很多人在一开始构建 Mitsuba 程序时都很容易忽略这些静态初始化，导致 Mitsuba 程序无法正常运行。

之后就可以使用 write 函数了：

```
// 使用boost的文件系统
fs::path cat_image_path("./Icons/cat-2.png", boost::filesystem::native);
bitmap->write(cat_image_path);
```

六 Film 类的使用

由于代码中的注释已经写的足够详细了，所以这里我们不再做过多介绍。代码主要是在 IMAGraphicsView.cpp 文件中完成的。但是也修改了一些其他的地方，我们会详细列出。

6 1 修改的 Mitsuba 文件

由于 Mitsuba 中的具体类（比如 LDRFilm）都是用插件的形式完成的，所以我们需要修改一下以便整个工程能直接支持。此外 Film 中还需要滤波器，因此我们需要将 rFilters 文件夹中的滤波器都移植到系统中。

新建两个头文件：getFilmInstance.h 和 getFiltersInstance.h，在里面实现新建对象的接口，大家可以在源码中参考这两个头文件中的内容，非常简单。

Film::configure() 中会使用插件的方式新建类，我们修改一下：

```
    Properties prop;
    m_filter = getGaussianFilterInstance(prop);
```

6 2 初始化 Properties

Properties 中的一些函数，比如 setFloat()、getInteger() 等函数，都是根据 DEFINE_PROPERTY_ACCESSOR 宏来定义的。

scenehandler.h 头文件中定义了 ParseContext 结构，该结构中含有一个 Properties 对象，我们不使用 ParseContext 结构，而是直接定义一个 Properties 对象：

```
Properties prop_film;
prop_film.setInteger("width", width_img);
prop_film.setInteger("height", height_img);
```

6 3 直接用 bitmap 赋值给 Film

见 IMAGraphicsView.cpp 文件：

```
    Properties prop_film;
    prop_film.setInteger("width", width_img);
    prop_film.setInteger("height", height_img);
    ref<Film> m_film = mitsuba::getLDRFilm_Instance(prop_film);
    m_film->configure();
    m_film->setBitmap(bitmap);

    m_film->savePNG_files("./Icons/cat-3.png", 0.0f);
```

运行完就能看到输出的 cat-3.png。

6 4 使用 ImageBlock 来进行赋值

见 IMAGraphicsView.cpp 文件，由于源码中的注释非常详细，所以这里不再赘述。

注意 put() 函数是在原值的基础上增加值，而不是赋值操作。

七 本文小结

本文写作和代码调试花了整整一周的时间，期间解决了很多 bug，比如一开始不小心将 Debug 和 Release 模式的 boost 依赖库都导入到了依赖项中，结果导致 boost::filesystem 的文件读取一直报错。这些问题虽然都看似简单，但是一旦遇到也着实需要费不少功夫才能解决。

本文到目前位置需要读者去配置的内容仅有 boost 库和 Qt 库，后面应该也不再需要额外需要配置的依赖环境了，因此还算比较简洁。注意我们生成的 Visual Studio 工程需要是 Debug 模式 x64 位的。

参考文献

[1] http://mitsuba-renderer.org/api/classmitsuba__1__1__film.html

[2] https://www.boost.org/users/news/

[3] https://download.qt.io/archive/qt/

[4] https://blog.csdn.net/u012516419/article/details/112978228

[5] https://www.lmlphp.com/user/220345/article/item/3793612/

[6] https://blog.csdn.net/jacke121/article/details/88695522