最长的一帧

王锐（array）

这是一篇有关 OpenSceneGraph 源代码的拙劣教程，没有任何能赏心悦目的小例子，也不会贡献出什么企业级的绝密的商业代码，标题也只是个噱头（坏了，没人看了^_^）。

本文写作的目的说来很简单，无非就是想要深入地了解一下，OSG 在一帧时间，也就是仿真循环的一个画面当中都做了什么。

对 OSG 有所了解之后，我们也许可以很快地回答这个问题，正如下面的代码所示：

```
while (!viewer.done())
  viewer.frame();
```

就这样，用一个循环结构来反复地执行 frame()函数，直到 done()函数的返回值为 true 为止。每一次执行 frame()函数就相当于完成了 OSG 场景渲染的一帧，配置较好的计算机可以达到每秒钟一二百帧的速率，而通常仿真程序顺利运行的最低帧速在 15~25 帧/秒即可。

很好，看来笔者的机器运行 frame()函数通常只需要 8~10ms 左右，比一眨眼的工夫都要短。那么本文就到此结束吗？

答案当然是否定的，恰恰相反，这篇繁琐且可能错误百出的文字，其目的正是要深入 frame()函数，再深入函数中调用的函数……一直挖掘下去，直到我们期待的瑰宝出现；当然也可能是一无所获，只是乐在其中。

这样的探索要到什么时候结束呢？从这短短的 10 毫秒中引申出来的，无比冗长的一帧，又是多么丰富抑或无聊的内容呢？现在笔者也不知道，也许直到最后也不会明了，不过相信深入源代码的过程就是一种享受，希望读者您也可以同我一起享受这份辛苦与快乐。

源代码版本：OpenSceneGraph 2.6.0；操作系统环境假设为 Win32 平台。为了保证教程的篇幅不致被过多程序代码所占据，文中会适当地改写和缩编所列出的代码，仅保证其执行效果不变，因此可能与实际源文件的内容有所区别。

由于作者水平和精力所限，本文暂时仅对单视景器（即使用 osgViewer::Viewer 类）的情形作出介绍。

本文在写作过程中将会用到一些专有名词，它们可能与读者阅读的其它文章中所述有所差异，现列举如下：

场景图形-SceneGraph；场景子树-Subgraph；节点-Node；摄像机-Camera；渲染器-Renderer；窗口-Window；视口-Viewport；场景-Scene；视图-View；视景器-Viewer；漫游器-Manipulator；访问器-Visitor；回调-Callback；事件-Event；更新-Update；筛选-Cull；绘制-Draw。

第一日

好了，在开始第一天的行程之前，请先打开您最惯用的编程工具吧：VisualStudio？CodeBlocks？UltraEdit？SourceInsight？Emacs？Vim？或者只是附件里那个制作低劣的记事本……总之请打开它们，打开 OpenSceneGraph-2.6.0 的源代码文件夹，打开

osgViewer/ViewerBase.cpp 这个文件……同样的话就不再重复了。但是如果您没有这样做，而仅仅是一边聊着 QQ，一边在电话里和女朋友发着牢骚，一边还要应付突然冲进来的老板，一边打开这篇教程的话——对不起，我想您会在 1 分钟以内就对其感到厌烦，因为它就像天书一样，或者就像一坨乱七八糟的毛线团，只有家里那只打着哈欠的小猫可能会过来捅上一下。

不过不用担心，如果您已经耐着性子读到了这里，并且已经看到了 ViewerBase.cpp 中总共 752 行规规矩矩的代码，那么我会尽全力协助您继续走完漫长的一帧的旅程，并在其中把自己所知所得（不仅仅局限于那个该死的 frame 函数，而是遍及 OSG 的方方面面）全数灌输给您。当然也希望您尽全力给我以打压、批评和指正，我只是一个普普通通的梦想着先找个女朋友成家立业的毛头小子而已，如果我的话全都是对的，那么那些真正的图形学权威们一定都住在寒风刺骨的珠穆朗玛峰顶上。

好了，废话不再多说。我们这就开始。

当前位置：osgViewer/ViewerBase.cpp 第 571 行，osgViewer::ViewerBase::frame()

frame 函数的内容本身几乎一眼就可以看完。不过要注意的是，这个函数是 ViewerBase 类的成员函数，而非 Viewer 类。因此，无论对于单视景器的 Viewer 类，还是多视景器的 CompositeViewer 类，frame 函数的内容都是相同的（因为它们都没有再重写这个函数的内容）。

该函数所执行的主要工作如下：

1、如果这是仿真系统启动后的第一帧，则执行 viewerInit()；此时如果还没有执行 realize() 函数，则执行它。

2、执行 advance 函数。

3、执行 eventTraversal 函数，顾名思义，这个函数将负责处理系统产生的各种事件，诸如鼠标的移动，点击，键盘的响应，窗口的关闭等等，以及摄像机与场景图形的事件回调（EventCallback）。

4、执行 updateTraversal 函数，这个函数负责遍历所有的更新回调（UpdateCallback）；除此之外，它的另一个重要任务就是负责更新 DatabasePager 与 ImagePager 这两个重要的分页数据处理组件。

5、执行 renderingTraversals 函数，这里将使用较为复杂的线程处理方法，完成场景的筛选（cull）和绘制（draw）工作。

下面我们就按照 1~5 的顺序，开始我们的源代码解读之旅。

当前位置：osgViewer/View.cpp 第 227 行，osgViewer::View::init()

Viewer::viewerInit 函数只做了一件事，就是调用 View::init()函数，而这个 init 函数的工作似乎也是一目了然的：无非就是完成视景器的初始化工作而已。

不过在我们离开这个函数，继续我们的旅程之前，还是仔细探究一下，这个初始化工作到底包含了什么？

阅读某个函数的源代码过程中，如果能够大致知道这个函数的主要工作，并了解其中用到的变量的功能，那么即使只有很少的注释内容，应该也可以顺利地读完所有代码。如果对一些命名晦涩的变量不甚理解，或者根本不知道这个函数于运行流程中有何用途，那么理解源代码的过程就会麻烦很多。

View::init 函数中出现了两个重要的类成员变量：_eventQueue 和_cameraManipulator，并且还将一个 osgGA::GUIEventAdapter 的实例传入后者的初始化函数。

代码如下：

```
osg::ref_ptr<osgGA::GUIEventAdapter> initEvent = _eventQueue->createEvent();
```

initEvent->setEventType(osgGA::GUIEventAdapter::FRAME);

if (_cameraManipulator.valid())

_cameraManipulator->init(*initEvent, *this);

从变量的名称可以猜测出_eventQueue 的功能，它用于储存该视景器的事件队列。OSG 中代表事件的类是 osgGA::GUIEventAdapter，它可以用于表达各种类型的鼠标、键盘、触压笔和窗口事件。在用户程序中，我们往往通过继承 osgGA::GUIEventHandler 类，并重写 handle 函数的方法，获取实时的鼠标/键盘输入，并进而实现相应的用户代码（参见 osgkeyboardmouse）。

_eventQueue 除了保存一个 GUIEventAdapter 的链表之外，还提供了一系列对链表及其元素的操作函数，这其中，createEvent 函数的作用是分配和返回一个新的 GUIEventAdapter 事件的指针。

随后，这个新事件的类型被指定为 FRAME 事件，即每帧都会触发的一个事件。

那么，_cameraManipulator 呢？没错，它就是视景器中所用的场景漫游器的实例。通常我们都会使用 setCameraManipulator 来设置这个变量的内容，例如轨迹球漫游器（TrackballManipulator）可以使用鼠标拖动来观察场景，而驾驶漫游器（DriveManipulator）则使用类似于汽车驾驶的效果来实现场景的漫游。

上面的代码将新创建的 FRAME 事件和 Viewer 对象本身传递给_cameraManipulator 的 init 函数，不同的漫游器（如 TrackballManipulator、DriveManipulator）会重写各自的 init 函数，实现自己所需的初始化工作。如果读者希望自己编写一个场景的漫游器，那么覆写并使用 osgGA::MatrixManipulator::init 就可以灵活地初始化自定义漫游器的功能了，它的调用时机就在这里。

那么，回到 viewerInit 函数……很好，这次似乎没有更多的内容了。没想到一个短短的函数竟然包含了那么多的信息，看来草率地阅读还真是使不得。

解读成果：

osgGA::EventQueue::createEvent，osgGA::MatrixManipulator::init，osgViewer::View::init，osgViewer::Viewer::viewerInit。

悬疑列表：

无。

第二日

当前位置：osgViewer/Viewer.cpp 第 385 行，osgViewer::Viewer::realize()

Viewer::realize 函数是我们十分熟悉的另一个函数，从 OSG 问世以来，我们就习惯于在进入仿真循环之前调用它（现在的 OSG 会自动调用这个函数，如果我们忘记的话），以完成窗口和场景的“设置”工作。那么，什么叫做“设置”，这句简单的场景设置又包含了多少内容呢？艰辛的旅程就此开始吧。

首先是一行：setCameraWithFocus(0)，其内容无非是设置类变量_cameraWithFocus 指向的内容为 NULL。至于这个“带有焦点的摄像机”是什么意思，我们似乎明白，似乎又不明白，就先放入一个“悬疑列表”（Todo List）中好了。

下面遇到的函数就比较重要了，因为我们将会在很多地方遇到它：

Contexts contexts;

getContexts(contexts);

变量 contexts 是一个保存了 osg::GraphicsContext 指针的向量组，而 Viewer::getContexts

函数的作用是获取所有的图形上下文，并保存到这个向量组中来。

对于需要将 OSG 嵌合到各式各样的 GUI 系统（如 MFC，Qt，wxWidgets 等）的朋友来说，osg::GraphicsContext 类是经常要打交道的对象之一。一种常用的嵌入方式也许是这样实现的：

```
osg::ref_ptr<osg::GraphicsContext::Traits> traits = new osg::GraphicsContext::Traits;
osg::ref_ptr<osg::Referenced> windata =
   new osgViewer::GraphicsWindowWin32::WindowData(hWnd);
traits->x = 0;
traits->y = 0;
……
traits->inheritedWindowData = windata;
osg::GraphicsContext* gc = osg::GraphicsContext::createGraphicsContext(traits.get());
Camera* camera = viewer.getCamera();
camera->setGraphicsContext(gc);
……
viewer.setCamera(camera);
```

这个过程虽然比较繁杂，但是顺序还是十分清楚的：首先设置嵌入窗口的特性（Traits），例如 X、Y 位置，宽度和高度，以及父窗口的句柄（inheritedWindowData）；然后根据特性的设置创建一个新的图形设备上下文（GraphicsContext），将其赋予场景所用的摄像机。而我们在 getContexts 函数中所要获取的，也许就包括这样一个用户建立的 GraphicsContext 设备。

当前位置：osgViewer/Viewer.cpp 第 1061 行，osgViewer::Viewer::getContexts()

在这个函数之中，首先判断场景的主摄像机_camera 是否包含了一个有效的 GraphicsContext 设备，然后再遍历所有的从摄像机_slaves（一个视景器可以包含一个主摄像级和多个从摄像机），将所有找到的 GraphicsContext 图形上下文设备记录下来。

随后，将这些 GraphicsContext 的指针追加到传入参数（contexts 向量组）中，并使用 std::sort 执行了一步排序的工作，所谓的排序是按照这样的原则来进行的：

1、屏幕数量较少的 GraphicsContext 设备排列靠前；

2、窗口 X 坐标较小的设备排列靠前；

3、窗口 Y 坐标较小的设备排列靠前。

如果希望观察自己的程序中所有的图形设备上下文，不妨使用这个函数来收集一下。简单的情形下，我们的程序中只有一个主摄像机，也就只有一个 GraphicsContext 设备，它表达了一个全屏幕的图形窗口；而 osgcamera 这个例子程序可以创建六个从摄像机，因此可以得到六个图形上下文设备，且各个图形窗口的 X 坐标各不相同，这也正是这个例子想要表达的。

可是，主摄像机的 GraphicsContext 呢？为什么 osgcamera 中不是七个 GraphicsContext 设备呢？答案很简单，主摄像机没有创建图形上下文，因此也就得不到设备的指针。为了理解这个现象的原因，我们不妨先回到 realize 函数中。

当前位置：osgViewer/Viewer.cpp 第 394 行，osgViewer::Viewer::realize()

有一个显而易见的事实是：当程序还没有进入仿真循环，且对于 osgViewer::Viewer 还没有任何的操作之时，系统是不会存在任何图形上下文的；创建一个新的 osg::Camera 对象也不会为其自动分配图形上下文。但是，图形上下文 GraphicsContext 却是场景显示的唯一

平台，系统有必要在开始渲染之前完成其创建工作。

假设用户已经在进入仿真循环之前，自行创建了新的 Camera 摄像机对象，为其分配了自定义的 GraphicsContext 设备，并将 Camera 对象传递给视景器，就像 osgviewerMFC 和 osgcamera 例子，以及我们在编写与 GUI 系统嵌合的仿真程序时常做的那样。此时，系统已经不必为图形上下文的创建作任何多余的工作，因为用户不需要更多的窗口来显示自己的场景了。所以就算主摄像机_camera 还没有分配 GraphicsContext，只要系统中已经存在图形上下文，即可以开始执行仿真程序了。

但是，如果 getContexts 没有得到任何图形上下文的话，就说明仿真系统还没有合适的显示平台，此时就需要尝试创建一个缺省的 GraphicsContext 设备，并再次执行 getContexts，如果还是没能得到任何图形上下文的话，那么就不得不退出程序了。

创建缺省 GraphicsContext 设备的方法有以下几种：

1、读取 OSG_CONFIG_FILE 环境变量的内容：如果用户在这个环境变量中定义了一个文件路径的话，那么系统会尝试用 osgDB::readObjectFile 函数读入这个文件，使用 cfg 插件进行解析；如果成功的话，则调用 osgViewer::Viewer::take 函数，使用配置信息设置当前的视景器。这些工作在 osgViewer::Viewer::readConfiguration 函数中实现。

2、读取 OSG_WINDOW 环境变量的内容：如果用户以“x y w h”的格式在其中定义了窗口的左上角坐标（x，y）和尺寸（w，h）的话（注意要以空格为分隔符），系统会尝试使用 osgViewer::View::setUpViewInWindow 函数来创建设备。

3、读取 OSG_SCREEN 环境变量的内容：如果用户在其中定义了所用屏幕的数量的话，系统会尝试用 osgViewer::View::setUpViewOnSingleScreen 函数，为每一个显示屏创建一个全屏幕的图形窗口；如果同时还设置了 OSG_WINDOW，那么这两个环境变量都可以起到作用，此时将调用 setUpViewInWindow 函数。

4、如果上述环境变量都没有设置的话（事实上这也是最常见的情况），那么系统将调用 osgViewer::View::setUpViewAcrossAllScreens 函数，尝试创建一个全屏显示的图形设备。

那么，下文就从这几种图形设备建立的方法开始。至于后面的路，果然遥遥无期呢。

解读成果：

osgViewer::Viewer::getContexts，osgViewer::Viewer::readConfiguration。

悬疑列表：

类变量_cameraWithFocus 的意义是什么？

第三日

当前位置：osgViewer/View.cpp 第 575 行，osgViewer::View::setUpViewInWindow()

这个函数有五个传入参数：窗口左上角坐标 x，y，宽度 width，高度 height，以及屏幕数 screenNum。它的作用顾名思义是根据给定的窗口参数来创建一个图形设备。

首先函数将尝试获取 osg::DisplaySettings 的指针，这个类在 OSG 的窗口显示中扮演了重要的地位：它保存了 OSG 目前用到的，与图形显示，尤其是立体显示有关的所有信息，主要包括：

_displayType：显示器类型，默认为 MONITOR（监视器），此外还支持 POWERWALL（威力墙），REALITY_CENTER（虚拟实境中心）和 HEAD_MOUNTED_DISPLAY（头盔显示器）。

_stereoMode：立体显示模式，默认为 ANAGLYPHIC（互补色），此外还支持

QUAD_BUFFER（四方体缓冲），HORIZONTAL_SPLIT（水平分割），VERTICAL_SPLIT（垂直分割），LEFT_EYE（左眼用），RIGHT_EYE（右眼用），HORIZONTAL_INTERLACE（水平交错），VERTICAL_INTERLACE（垂直交错），CHECKERBOARD（棋盘式交错，用于 DLP 显示器）。

_eyeSeparation：双眼的物理距离，默认为 0.05。

_screenWidth，_screenHeight：屏幕的实际宽度和高度，分别默认设置为 0.325 和 0.26，目前它们影响的仅仅是视图采用透视投影时的宽高比。

_screenDistance：人眼到屏幕的距离，默认为 0.5。

_splitStereoHorizontalEyeMapping：默认为 LEFT_EYE_LEFT_VIEWPORT（左眼渲染左视口），也可设为 LEFT_EYE_RIGHT_VIEWPORT（左眼渲染右视口）。

_splitStereoHorizontalSeparation：左视口和右视口之间的距离（像素数），默认为 0。

_splitStereoVerticalEyeMapping：默认为 LEFT_EYE_TOP_VIEWPORT（左眼渲染顶视口），也可设为 LEFT_EYE_BOTTOM_VIEWPORT（左眼渲染底视口）。

_splitStereoVerticalSeparation：顶视口和底视口之间的距离（像素数），默认为 0。

_splitStereoAutoAdjustAspectRatio：默认为 true，用于屏幕分割之后对其宽高比进行补偿。

_maxNumOfGraphicsContexts：用户程序中最多可用的 GraphicsContext（图形设备上下文）数目，默认为 32 个。

_numMultiSamples：多重采样的子像素样本数，默认为 0。如果显示卡支持的话，打开多重采样可以大幅改善反走样（anti-aliasing）的效果。

此外还有很多可以设置的类变量，如_minimumNumberStencilBits（模板缓存的最小位数）等，其默认设置均在 osg::DisplaySettings::setDefaults 函数中完成，其中有些变量可能还没有作用。要注意的是，DisplaySettings 的作用仅仅是保存所有可能在系统显示中用到的数据，这个类本身并不会据此改变任何系统设置和渲染方式。

值得称道的是，DisplaySettings 可以很方便地从系统环境变量或者命令行参数中获取用户对显示设备的设置，详细的调用方法可以参阅 DisplaySettings::readEnvironmentalVariables 和 DisplaySettings::readCommandLine 两个函数的内容，十分通俗易懂。

如果希望在用户程序中更改 DisplaySettings 中的显示设置，请务必在执行视景器的 realize 函数之前，当然也就是仿真循环开始之前。这一点也是要切记的。

不知不觉中，似乎完全跑题了，那么我们还是先设法回到主题上来……

当前位置：osgViewer/View.cpp 第 579 行，osgViewer::View::setUpViewInWindow()

代码解读的工作完全没有进展，看来需要加快进度了。获取系统显示设备的设置参数之后，下面我们要开始创建新的 GraphicsContext 设备了，回忆“第二日”的内容中所介绍的 OSG 与 GUI 窗口嵌合的流程，第一步是新建一个显示设备特性实例：

```
osg::ref_ptr<osg::GraphicsContext::Traits> traits = new osg::GraphicsContext::Traits;
```

设置图形窗口的特性值。注意这里用到了一个函数 ScreenIdentifier::readDISPLAY，它的工作仅仅是尝试检查系统环境变量 DISPLAY，并调用 ScreenIdentifier::setScreenIdentifier 函数，将其中的内容按照“hostName:0.0”的格式解析为系统的主机名称（hostName），显示数（在 Win32 下必须为 0）和屏幕数（0 或者其它数字）。

根据立体显示模式的不同，窗口特性中的模板位数等参量也会有所区分。

下一步，创建新的 GraphicsContext，并将其设置给视景器的主摄像机：

```
osg::ref_ptr<osg::GraphicsContext> gc =
    osg::GraphicsContext::createGraphicsContext(traits.get());
```

_camera->setGraphicsContext(gc.get());

千万不要简单地使用 new 来创建新的 GraphicsContext 指针，因为相比起来，createGraphicsContext 还完成了这样一些工作：

1、获取窗口系统 API 接口，即 GraphicsContext::WindowingSystemInterface 的实例；

2、执行 setUndefinedScreenDetailsToDefaultScreen 函数，如果用户没有设置屏幕数，则自动设置为缺省值 0；

3、返回 WindowingSystemInterface::createGraphicsContext 的值。

看似一切顺利，但是稍一深究就会发现，这里面存在了一个重要但是不好理解的问题：WindowingSystemInterface::createGraphicsContext 可是一个纯虚函数，它怎么可能返回新建立的图形设备上下文呢？事实上，这个看似简单的 WindowingSystemInterface 结构体也是另有玄机的，注意这个函数：

void GraphicsContext::setWindowingSystemInterface
(WindowingSystemInterface* callback);

它的作用是指定操作平台所使用的视窗 API 接口，也就是在特定的系统平台上创建图形窗口的时候，将会使用到哪些本地 API 函数。当然，Windows 系统要使用 Win32 API，而 Linux 系统要使用 X11 API，Apple 系统则使用 Carbon。

一切有关视窗 API 接口的工作都是由 GraphicsWindowWin32，GraphicsWindowX11 和 GraphicsWindowCarbon 这三个类及其协作类来完成；而指定使用哪一个窗口系统 API 接口的关键，就在于源文件 osgViewer/GraphicsWindowWin32.cpp 中定义的结构体 RegisterWindowingSystemInterfaceProxy 了，仔细研读一下这个结构体和刚才所述的 setWindowingSystemInterface 函数的关系，还有注意那个紧跟着结构体的全局变量（GraphicsWindowWin32.cpp，2367 行），相信您一定会大呼巧妙的。

什么，GraphicsWindowX11.cpp 中也有这个结构体？那么请仔细检查一下 CMake 自动生成的 osgViewer.vcproj 工程，看看有没有包含这个多余的文件（对于 Windows 系统来说）——这也许就是使用 CMake 来实现跨平台编译的好处之一了。

至于 WindowingSystemInterface::createGraphicsContext 函数是如何使用 Win32 API 来实现图形设备的创建的，鉴于本文并不想追赶《资本论》的宏伟规模，就不再深究了，读者不妨自行刨根问底。

回来吧，回来吧。还是让我们回到 setUpViewInWindow 函数中来。

这个函数剩下的内容并不是很多，也不难理解，主要的工作有：

1、调用 osgGA::GUIEventAdapter::setWindowRectangle 记录新建立的窗口设备的大小，因而这个设备上产生的键盘和鼠标事件可以以此为依据。

2、设置主摄像机_camera 的透视投影参数，并设置新的 Viewport 视口。

3、执行 osg::Camera::setDrawBuffer 和执行 osg::Camera:: setReadBuffer 函数，这实质上相当于在渲染的过程中执行 glDrawBuffer 和 glReadBuffer，从而自动设置此摄像机想要绘制和读取的缓存。

就这样。不过这回真是一次又一次地离题万里……希望我们还是从中得到了一些收获和启迪的，对吗？

解读成果：

osg::DisplaySettings::setDefaults，osg::GraphicsContext::createGraphicsContext，osgViewer::View:: setUpViewInWindow。

悬疑列表：

类变量_cameraWithFocus 的意义是什么？

第四日

当前位置：osgViewer/Viewer.cpp 第 426 行，osgViewer::Viewer::realize()

setUpViewOnSingleScreen 和 setUpViewAcrossAllScreens 函数的实现流程与上一日介绍的 setUpViewInWindow 区别不是很大。值得注意的是，setUpViewAcrossAllScreens 函数中调用 GraphicsContext::getWindowingSystemInterface 函数取得了与平台相关的视窗 API 接口类（其中的原理请参看上一日的文字），并进而使用 WindowingSystemInterface::getNumScreens 函数取得了当前系统的显示屏幕数。

事实上，如果我们需要在自己的程序中获取屏幕分辨率，或者设置屏幕刷新率的话，也可以使用同样的方法，调用 getScreenResolution，setScreenResolution 和 setScreenRefreshRate 等相关函数即可。具体的实现方法可以参见 GraphicsWindowWin32.cpp 的源代码。

setUpViewAcrossAllScreens 函数可以自行判断屏幕的数量，并且使用多个从摄像机来对应多个屏幕的显示（或者使用主摄像机_camera 来对应单一屏幕）。此外它还针对水平分割显示（HORIZONTAL_SPLIT）的情况，对摄像机的左/右眼设置自动做了处理，有兴趣的读者不妨仔细研究一下。

最后，本函数还执行了一个重要的工作，即 View::assignSceneDataToCameras，这其中包括以下几项工作：

1、对于场景漫游器_cameraManipulator，执行其 setNode 函数和 home 函数，也就是设置漫游器对应于场景图形根节点，并回到其原点位置。不过在我们使用 setCameraManipulator 函数时也会自动执行同样的操作。

2、将场景图形赋予主摄像机_camera，同时设置它对应的渲染器（Renderer）的相关函数。这里的渲染器起到了什么作用？还是先放到悬疑列表中吧，不过依照我们的解读速度，这个问题可能会悬疑很久。

3、同样将场景图形赋予所有的从摄像机_slaves，并设置每个从摄像机的渲染器。

终于可以回到 realize 函数的正轨了，还记得下一步要做什么吧？对，在尝试设置了缺省的 GraphicsContext 设备之后，我们需要再次使用 getContexts 来获取设备，如果还是不成功的话，则 OSG 不得不退出运行了（连图形窗口都建立不起来，还玩什么）。

当前位置：osgViewer/Viewer.cpp 第 446 行，osgViewer::Viewer::realize()

现在我们遍历所得的所有 GraphicsContext 设备（通常情况下，其实只有一个而已）。对于每个 GraphicsContext 指针 gc，依次执行：

```
gc->realize();
if (_realizeOperation.valid() && gc->valid())
{
   gc->makeCurrent();
   (*_realizeOperation)(gc);
   gc->releaseContext();
}
```

一头雾水，但是决不能轻言放弃。仔细研究一下吧，首先是 GraphicsContext::realize 函数，实际上也就是 GraphicsContext::realizeImplementation 函数。

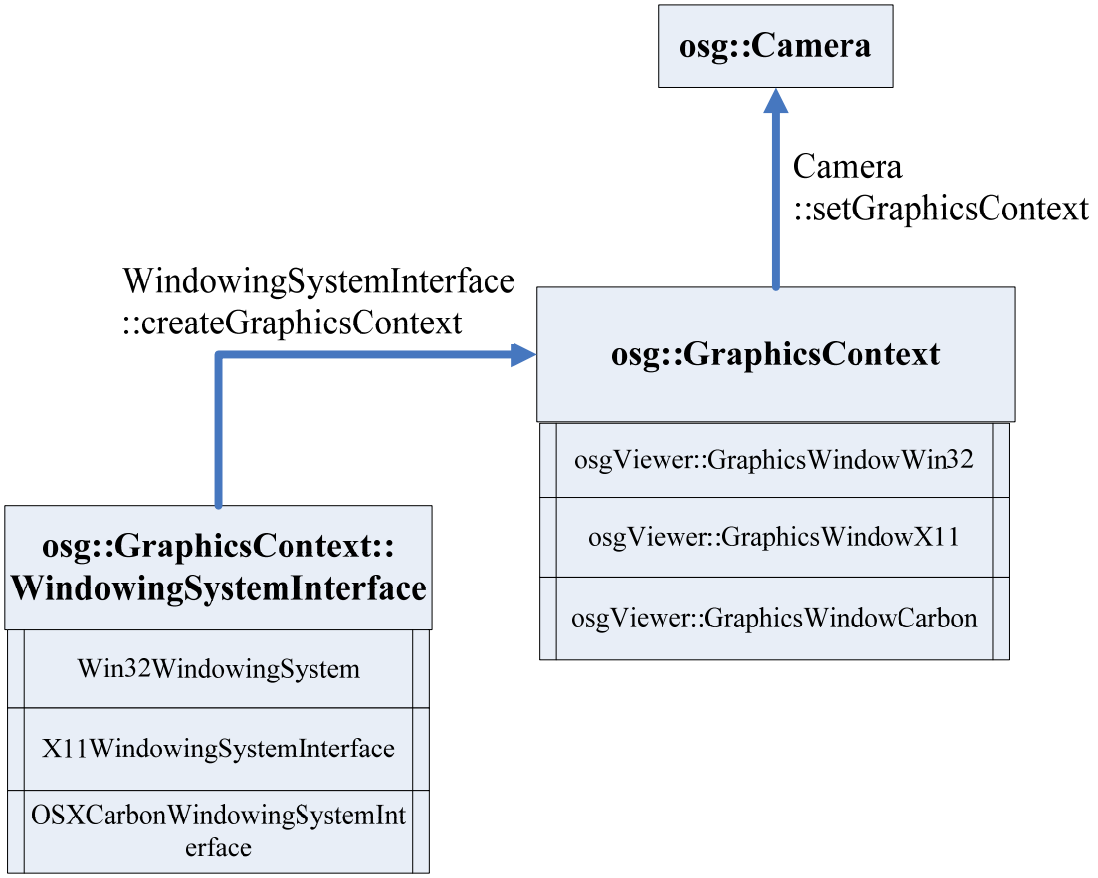
realizeImplementation 是纯虚函数吗？没错，回想一下第三日的内容，当我们尝试使用 createGraphicsContext 来创建一个图形设备上下文时，系统返回的实际上是这个函数的值：

```
ref_ptr<GraphicsContext::WindowingSystemInterface> &wsref =
    windowingSystemInterfaceRef();
return wsref->createGraphicsContext(traits);
```

而正如我们历经千辛万苦所分析的那样，wsref 所指向的是平台相关的 API 接口类，也就是 Win32 API 的接口，也就是 GraphicsWindowWin32.cpp 中对应类的实例。换句话说，此时 WindowingSystemInterface:: createGraphicsContext 函数返回的值，也应当是派生自 GraphicsContext 的具体类的实例！

正确，对于 Windows 用户来说，这个函数返回的恰恰是 GraphicsWindowWin32 的实例，而前文的 realizeImplementation 函数，正是 GraphicsWindowWin32::realizeImplementation。

晕头转向了……那么，用一张图也许能解决一点问题吧：



1、视景器 Viewer 的主/从摄像机均需要使用 setGraphicsContext 设置对应的图形设备上下文，实际上也就是对应的显示窗口；

2、GraphicsContext 的创建由平台相关的抽象接口类 WindowingSystemInterface 负责，对于 Win32 平台而言，这个类是由 GraphicsWindowWin32.cpp 的 Win32WindowingSystem 类具体实现的，它创建的显示窗口设备即 osgViewer::GraphicsWindowWin32 的实例。

3、进一步深究的话，如果窗口特性（Traits）中开启了 pbuffer 选项，则 OSG 将尝试创建 osgViewer::PixelBufferWin32 设备，以实现离屏渲染（Offscreen Render），纹理烘焙（Render-To-Texture）等工作；否则只建立通常的 OpenGL 窗口。

真是令人兴奋！没错，GraphicsContext::makeCurrent 和 GraphicsContext:: releaseContext 函数也是用相同的方法来实现多态性的，而它们的工作也就是 OpenGL 开发者使用函数 wglMakeCurrent 完成的工作，将渲染上下文 RC 对应到正确的窗口绘制句柄上。

如果您还想要深究具体的实现方法的话，就好好地阅读 GraphicsWindowWin32.cpp 中的相关内容吧，不过我们的旅程要继续了。

等等，刚才那段程序里面，_realizeOperation 是什么，它又执行了什么？嗯，简单说来，这个变量是通过 ViewerBase::setRealizeOperation 来设置的，其主要作用是在执行 realize 函数时，顺便完成用户指定的一些工作。您自己的工作内容可以通过继承 osg::Operation 类，并重载 operator()操作符来添加。osgcatch 这个妙趣横生的例子（一个傻娃娃接玩具的小游戏）中就使用了 setRealizeOperation，主要的作用是为场景中的 Drawable 几何对象立即编译显示列表（Display List）。有兴趣的话不妨细细把玩一下。

解读成果：

osgViewer::View::setUpViewOnSingleScreen，osgViewer::View::assignSceneDataToCameras，osgViewer::View::setUpViewAcrossAllScreens。

悬疑列表：

类变量_cameraWithFocus 的意义是什么？渲染器（Renderer）类起到什么作用？

第五日

当前位置：osgViewer/Viewer.cpp 第 463 行，osgViewer::Viewer::realize()

下面我们再次遍历所有 GraphicsContext 设备，对于每个 GraphicsContext 指针 gc，判断它是否为 GraphicsWindow 对象，并执行 GraphicsWindow::grabFocusIfPointerInWindow 函数。阅读 GraphicsWindowWin32 类（即 GraphicsContext 的具体实现者）的同名函数可以发现，这个函数不过是负责把鼠标焦点转到当前窗口上而已。

下一步工作的代码如下：

```
osg::Timer::instance()->setStartTick();
setStartTick(osg::Timer::instance()->getStartTick());
setUpThreading();
```

首先调用 osg::Timer::setStartTick 函数，启动 OSG 内部定时器并开始计时。

随后，Viewer::setStartTick 函数的工作是找到当前视景器和所有 GraphicsContext 设备的事件队列_eventQueue，并设定它们的启动时刻为当前时间。

下一行是调用 ViewerBase::setUpThreading 函数……设置线程，对于一向以多线程渲染而闻名的 OSG 而言，这一定是个值得深究的话题。

当前位置：osgViewer/ViewerBase.cpp 第 122 行，osgViewer::ViewerBase::setUpThreading()

OSG 的视景器包括四种线程模型，可以使用 setThreadingModel 进行设置，不同的线程模型在仿真循环运行时将表现出不同的渲染效率和线程控制特性。通常而言，这四种线程的特性如下：

SingleThreaded：单线程模型。OSG 不会创建任何新线程来完成场景的筛选和渲染，因而也不会对渲染效率的提高有任何助益。它适合任何配置下使用。

CullDrawThreadPerContext：OSG 将为每一个图形设备上下文（GraphicsContext）创建一个图形线程，以实现并行的渲染工作。如果有多个 CPU 的话，那么系统将尝试把线程分别放在不同的 CPU 上运行，不过每一帧结束前都会强制同步所有的线程。

DrawThreadPerContext：这一线程模型同样会为每个 GraphicsContext 创建线程，并分配到不同的 CPU 上。十分值得注意的是，这种模式会在当前帧的所有线程完成工作之前，开始下一帧。

CullThreadPerCameraDrawThreadPerContext：这一线程模型将为每个 GraphicsContext 和每个摄像机创建线程，这种模式同样不会等待前一次的渲染结束，而是返回仿真循环并再次开始执行 frame 函数。如果您使用四核甚至更高的系统配置，那么使用这一线程模型将最大限度地发挥多 CPU 的处理能力。

与 DrawThreadPerContext 和 CullThreadPerCameraDrawThreadPerContext 这两种同样可以用于多 CPU 系统，且相对更有效率的线程模型相比，CullDrawThreadPerContext 的应用范围比较有限；而 SingleThreaded 模式在单核以及配置较低的系统上运行稳定。

这些话长篇大论地说出来似乎令人满腹疑窦：OSG 为什么要为每个 GraphicsContext 设备配置一个线程？为什么又要为每个摄像机配置一个线程？线程的同步是怎么实现的？线程与 CPU 的关系又是怎么处理的？OSG 入门书籍中常说的更新（Update）/筛选（Cull）/绘制（Draw）三线程又是在那里体现的？为什么……

天哪，这么多问题我们都要解读吗？是的，绝对要解读，不管花费多少时间！OSG 的学习是为了实际的应用，但是只有真正理解了它的运行机制，才能够最有效地把这个愈加著名的实时场景渲染软件用好。

但是有些事情是急不来的，从 frame 函数的源代码中可以大致推测出来，场景的筛选和绘制工作是由 ViewerBase::renderingTraversals 函数来完成的。相应的，很多线程的调度和同步工作也是在这个函数中完成的，那么就让我们把问题留到那个时候吧。不过不妨先透露一点信息：第四日中我们提到的渲染器（Renderer）类，事实上也是与 OSG 的渲染线程密切相关的，因为筛选和绘制的工作就是由它来具体负责！好的，遗留的问题可以说暂时得到了解答，不过新的问题又出现了，而且任务看起来更为艰巨，继续努力好了。

线程相关的问题留待后面解决，不过还是让我们先通读一下 setUpThreading 函数的代码也无妨。它的工作主要是处理单线程（SingleThreaded）模式下，多处理器系统的数据线程分配方式。

听起来很深奥，不过实际上这里没有多么复杂。在现阶段，如果采用单线程模式的话，OSG 系统将使用 CPU0 来处理用户更新、筛选和渲染等一切事务，而使用 CPU1 来处理场景的两个分页数据库（DatabasePager）线程（它们分别用于处理本地和网络上的场景数据）。

这里还出现了一个 Viewer::getScenes 函数（osgViewer/Viewer.cpp，141 行），它的作用是获取当前视景器对应的 osgViewer::Scene 对象，也就是场景。一个场景包括了唯一的场景图形根节点，分页数据库（DatabasePager），以及分页图像库（ImagePager）。Viewer 视景器对象通常只包括一个 Scene 场景，而 CompositeViewer 复合视景器则可能包括多个场景对象。

如果系统采用了 SingleThreaded 之外的其它线程模型，那么 setUpThreading 函数将自动执行 ViewerBase::startThreading——多线程渲染的最重要函数之一，这个函数将在我们追踪到 renderingTraversals 函数的时候重新进行解析。

当前位置：osgViewer/Viewer.cpp 第 486 行，osgViewer::Viewer::realize()

好了，如果您还没有忘记我们来自何方的话，请回到 realize 函数，现在这个函数的执行已经接近了尾声，不过我们又遇到了一个问题：编译上下文（也就是 Compile Contexts，暂时就这样翻译吧）？如果要启用它的话并不困难，只需要在调用 realize 之前执行：

```
osg::DisplaySettings::instance()->setCompileContextsHint(true);
```

随后，正如您在 realize 函数的 491-503 行之间看到的，系统将设法遍历所有可能的 GraphicsContext 设备，针对它们分别再各自添加一个新的 GraphicsContext 设备（也就是说，如果系统中已经有了数个图形上下文，那么现在又将新增同样数量的图形上下文与之对应），所用的函数为 GraphicsContext::getOrCreateCompileContext。这之后，分别执行了创建图形线程，设置 CPU 依赖性，以及启动图形线程的工作，具体的实现内容可以暂时忽略。

观察 getOrCreateCompileContext 函数的内容，很快我们就可以发现其中的重点：这些新增的 GraphicsContext 对象使用了 pBuffer 的特性，并与对应的已有对象共享同一个图形上下文（Traits::sharedContext 特性）。事实上，这是 OSG 利用 OpenGL 的像素缓存（Pixel Buffer）技术，为图形上下文的后台编译提供的一种新的解决方案。这样不仅可以提高图形刷新的速度，还可以方便用户为某一特定的 GraphicsContext 设备添加特殊的处理动作，方法是使用 osg::GraphicsContext::getCompileContext 获取后台图形上下文，再使用 GraphicsContext::add 函数向其中追加 osg::Operation 对象，类似的例子可以参看 osgterrain。

对了，在结束这一日的旅途之前，还要提示一句：“编译上下文”这一功能在 Windows 的实现尚有问题，目前可能会造成系统的崩溃（不要大失所望呀^_^）。

解读成果：

线程模型，osgViewer::Viewer::realize。

悬疑列表：

类变量_cameraWithFocus 的意义是什么？如何调度和实现 OSG 的多线程机制？

第六日

当前位置：osgViewer/Viewer.cpp 第 510 行，osgViewer:: Viewer::advance()

好的，现在我们终于正式进入仿真循环当中了，之前的 realize 函数虽然十分重要，但它实际上是循环运行前的准备工作。而从这一日开始介绍的 advance，eventTraversal，updateTraversal 和 renderingTraversals 函数，才是真正的一帧的组成部分。

advance 函数的工作内容如下：

1、获取上一次记录的参考时间（Reference Time）；

2、根据当前时刻，重新记录参考时间，并因此得到两次记录之间的差值，即一帧经历的时间；

3、记录已经经过的帧数；

4、有的时候我们需要将帧速率，参考时间等内容予以记录并显示给用户，此时需要通过 ViewerBase::getStats 函数获得 osg::Stats 对象，用以进行帧状态的保存和显示；

5、如果需要的话，使用 Referenced::getDeleteHandler()来处理 osg::Referenced 对象被弃用之后的删除工作。

仿真循环运行的参考时间，总时间和总帧数都是由 osg::FrameStamp 变量_frameStamp 来处理的，如果用户程序需要获取这些信息的话，也可以通过读取这个变量的成员函数来实现。当然，使用 Viewer 中的 osg::Stats 变量_stats 也是可以的，缺省情况下，这个变量会忠实地记录当前帧以及之前的 24 帧的每帧用时，事件遍历用时，更新遍历用时，以及渲染遍历用时信息。如果我们想获得更多的历史数据，抑或对于频繁的记录操作感到厌烦，可以在开始仿真循环之前执行 ViewerBase::setStats 函数，重新设置这个记录器的参数，或者简单地将其置为 NULL。

一切顺利！不过在结束短暂的 advance 之旅之前，我们还有一个问题需要解决，就是那个看起来有点让人不解的 getDeleteHandler：

```
osg::Referenced::getDeleteHandler()->flush();
osg::Referenced::getDeleteHandler()->setFrameNumber(_frameStamp->getFrameNumber())
;
```

简单说来，它的工作是收集所有已经弃用的 OSG 场景对象，并在需要的时候（例如

advance 函数代码的相应部分）执行 osg::DeleteHandler::flush，将它们统一删除。

这里所说的“弃用”，与我们非常熟悉的 osg::ref_ptr 智能指针是密切相关的。我们已经知道，ref_ptr 采用内存引用计数的方式，当一个场景对象（通常是 Node 节点）链接到根节点或者其他节点时，它的引用计数加一，这一动作是通过 ref_ptr::ref()函数实现的；如果它被剔除出节点，那么它的引用计数减一，执行这一工作的函数是 ref_ptr::unref()。unref 函数的另一个重要任务是检查对象的引用计数值是否到达零，如果已经没有被其它对象所引用，那么称这个对象被“弃用”，它应当被立即删除，以释放相应的内存空间，避免泄露。

C++中最通用的删除对象的方法是 delete，OSG 的智能指针也是采用这种方式来释放对象的，不过由于OSG采用多线程更新/渲染的方式（这一点我们会在后面的日子中详细介绍），这样做可能带来会某些隐患，想象这样一种情况：

1、场景某个的节点负责显示某种图形，它的工作一直很正常；

2、我们采用 DrawThreadPerContext 或者 CullThreadPerCameraDrawThreadPerContext 线程模型，根据前一日中我们所知的，这两种模式中存在“上次的渲染工作与下次的更新工作交叠”这一情形。

3、假设我们在更新工作中立即将这个节点删除，而上次渲染工作可能正要将这个节点中的数据送往 OpenGL 图形渲染管线，那么灾难就发生了……

看到这里，读者您一定已经想到了一种解决方案。对，就是在渲染后台也使用 ref_ptr 来引用（ref）图形节点，然后在渲染结束取消引用（unref），这样不就可以避免无谓的牺牲了吗？也省却用户的很多麻烦。

说得有道理，不过这其中恐怕忽视了一个核心的问题：渲染效率。是的，假设我们要渲染成千上万个这样的几何体节点（这对您来说也许简直是家常便饭），如果每个节点的渲染都要多执行一次 ref/unref 的话，效率的损失将是无法被忽略的。事实上经过测算，CPU 时间的流失大概可以达到 6%，对于一个实时渲染系统来说，这的确值得斟酌。

因此，OSG 的新版本中提出了 DeleteHandler 的概念，也就是“垃圾收集”，把那些引用计数已经为零的对象统一收集起来，确保它们不会再被渲染线程用到之后，再在适当的地方予以释放。DeleteHandler 有一个重要的参数_numFramesToRetainObjects，它的意义是，垃圾对象被收集之后，再经过多少帧（默认设置是 2），方予以释放。因此，OSG 的垃圾收集器同样需要使用 DeleteHandler::setFrameNumber 来记录当前的帧数。

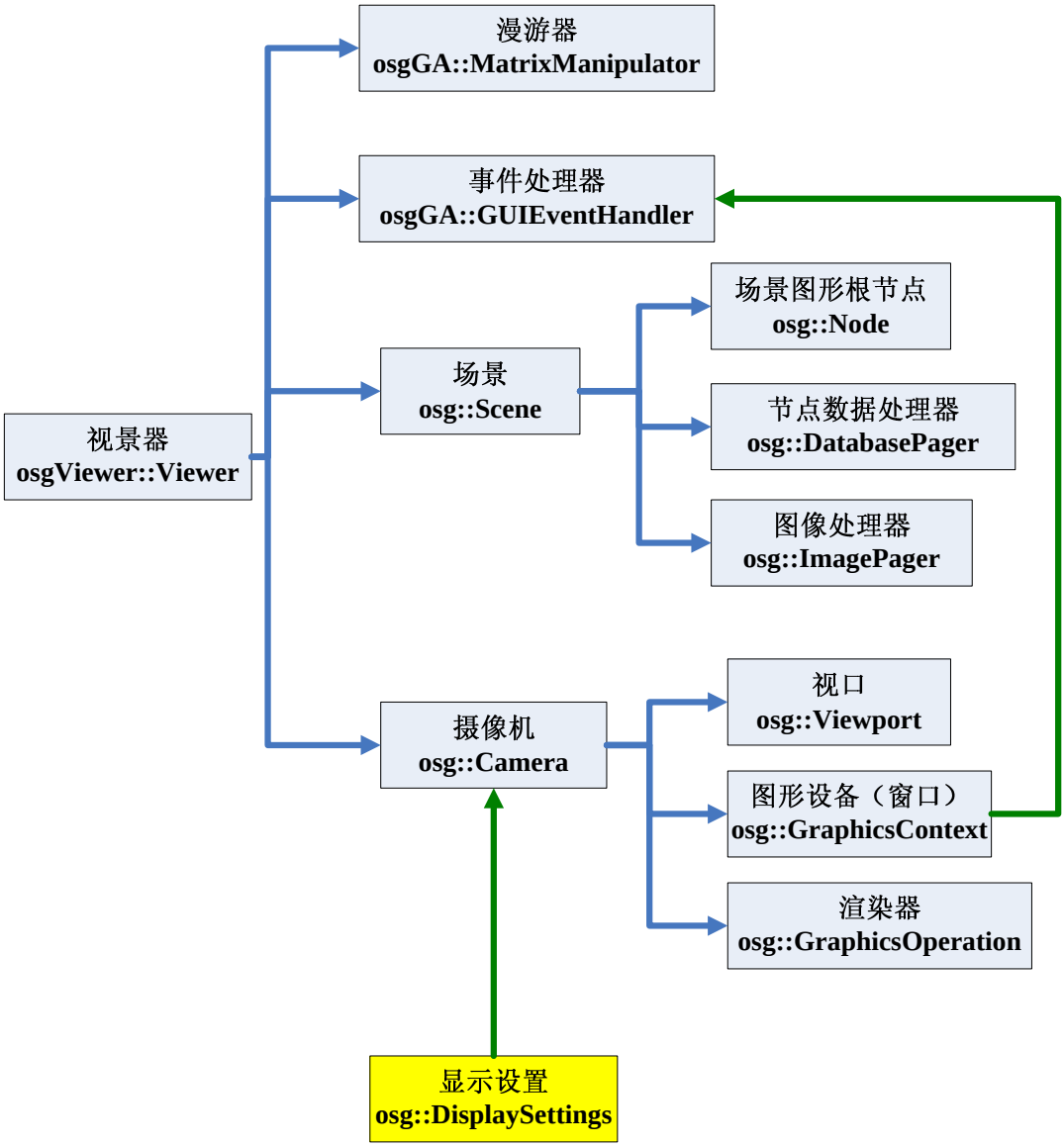
这个概念提出的时间并不长，也许还需要一段时间的测试，也许会有更好的方案来替代它。目前，OSG 的发行版本仍然采用第一种方式，也就是渲染后台采用 ref_ptr 引用计数的方式来避免删除对象造成的问题；如果您想要尝试使用和帮助调试 DeleteHandler 的话，可以在自己的程序中（main 函数之前）加入：

#undef OSGUTIL_RENDERBACKEND_USE_REF_PTR

以请求使用 DeleteHandler。

当前位置：osgViewer/Viewer.cpp 第 549 行，osgViewer::Viewer::eventTraversal()

在开始了解 OSG 事件遍历的过程之前，也许有必要先总结一下 OSG 视景器、摄像机与场景的关系。如下图所示：



视景器包括几个最主要的组件：漫游器_cameraManipulator，用于实现交互式的场景漫游；事件处理器组_eventHandlers，负责处理视景器的事件队列_eventQueue，主要是键盘/鼠标等事件的处理；场景_scene，它包括视景器所对应的场景图形根节点，以及用于提高节点和图像数据处理速度的两个分页数据库；摄像机_camera 和_slaves，前者为场景的主摄像机，后者为从摄像机组，不过 OSG 并没有规定一定要使用主摄像机来显示场景，它的更重要的作用是为 OSG 世界矩阵的计算提供依据。

摄像机是 OSG 视图显示的核心器件，没有摄像机就没有办法将场景图形的实景展现给用户。它包括：

1、视口（Viewport），它指示了摄像机显示窗口的位置和尺寸。

2、图形上下文（GraphicsContext），通常这也就是平台相关的图形显示窗口（即 GraphicsWindow，对于 Win32 系统而言，它实际上是通过 CreateWindowEx 这个熟悉的 API 来创建的），不过也可能是离屏渲染的设备（例如 PixelBufferWin32）。

图形窗口的另一个任务是及时把系统和用户交互的事件反馈到事件处理器组中去，观察 Win32 平台下的窗口设备 GraphicsWindowWin32 中的 handleNativeWindowingEvent 函数和它的传入参数，hwnd，msg，lParam，wParam……没错，相信您已经找到熟悉的感觉了。OSG 所处理的事件正是来源于 Win32 SDK 编程中常见的窗口消息。如果您正好有些新的想法想

要实践（例如，捕捉 WM_HOTKEY 系统热键消息，并传递给 OSG 的事件队列），不妨试一试修改相关的内容，并记得把好的建议发送到 osg-users。

3、渲染器（GraphicsOperation，更多时候是 osgViewer::Renderer），这是整个 OSG 筛选（CULL）和绘制（DRAW）的关键，它的功能我们会在后面的日子里慢慢展开。

此外，OSG 的显示设置工具 DisplaySettings 也会直接对摄像机的处理工作负责，大部分设置选项都可以传递到摄像机对应的窗口特性（GraphicsContext::Traits）中，并在渲染过程中发挥作用。

好了，闲话少说，下一日，让我们继续旅程。

解读成果：

osg::DeleteHandler，osgViewer:: Viewer::advance。

悬疑列表：

类变量_cameraWithFocus 的意义是什么？如何调度和实现 OSG 的多线程机制？

第七日

当前位置：osgViewer/Viewer.cpp 第 549 行，osgViewer::Viewer::eventTraversal()

OSG 的事件遍历与更新遍历是两个不同的顺序执行的过程，我们常说的更新回调（UpdateCallback）就是在更新遍历函数 updateTraversal 中被调用的。而现在我们将要介绍的事件遍历，除了可以执行用户设置的事件回调（EventCallback）之外，更重要的工作是为所有的用户交互和系统事件提供一个响应的机制。

上一日我们已经介绍过，视景器中保存有一个事件处理器组_eventHandlers，它负责处理由图形窗口设备传递到事件队列_eventQueue 的各种事件。根据我们已有的知识，新的事件处理器可以通过 View::addEventHandler 添加，除了 RecordCameraPathHandler，StatsHandler 等常见的处理器工具之外，通过继承事件处理器的基类 osgGA::GUIEventHandler，并重写 handle 函数，我们还可以实现自定义的交互事件响应流程，这在 osgviewer，osgkeyboardmouse 等例子中均有详细的演示。

那么，eventTraversal 函数的主要任务也就明了了：它必须在每一帧的仿真过程中，取出已经发生的所有事件，摒弃那些对场景不会有助益的（例如，在视口以外发生的鼠标移动事件和胡乱点击），依次交付给各个事件处理器，最后清空现有的事件队列，等待下一帧的到来。

eventTraversal 函数的前几行负责记录这个函数的起始时刻，相应的，在最后部分会记录函数的结束时刻，并将函数的总共运行时间送入_stats 记录器。这有助于我们了解每一帧当中事件遍历，更新遍历和渲染遍历运行所占用的时间比例。

在记录了起始时刻之后，OSG 系统执行了以下几项工作：

1、取得事件队列的状态事件（EventQueue::getCurrentEventState）；

2、取得主摄像机的视口范围（如果它存在的话，正如我们在前面所论述的，主摄像机并不一定存在 Viewport 视口，也不一定存在 GraphicsContext 图形设备），并设置为事件队列的“响应范围”（EventQueue::setInputRange）；

3、计算主摄像机的 VPW 矩阵。

这其中有一些名词需要加以解释，首先是“状态事件”。OSG 的事件队列实际上可以理解为一个由 GUIEventAdapter 事件组成的链表，诸如鼠标的移动，键盘上的按键被按下，窗口的尺寸被改变等动作，都会作为一个新的 GUIEventAdapter 对象插入到链表中，插入事件

的方法是由图形窗口 GraphicsWindow 执行 EventQueue 类的成员函数 mouseMotion，keyPress 和 windowResize，并间接地调用 EventQueue::addEvent 函数。而这些事件之间可能共通的参数和状态就从“状态事件”中读取。

举例来说，鼠标移动的事件之后，再触发键盘按键的事件（这种操作在《反恐精英》等游戏中司空见惯），则前者将负责更新“状态事件”中的鼠标 X，Y 坐标参数；而后者就从中取得此坐标，并再次更新“状态事件”中按键相关的信息。

因此，当我们在处理有关按键的 GUIEventAdapter 事件时，同样也可以使用成员函数 GUIEventAdapter::getX 和 getY 取得当前的鼠标位置，而不必担心键盘事件与鼠标的操作无关。

既然事件队列中存在一个公用的“状态事件”，那么存在公用的“响应范围”就不难理解了，EventQueue::setInputRange 函数的主要工作是设置鼠标活动的最大和最小范围，如果同时还开启了鼠标范围的限定标志（EventQueue::setUseFixedMouseInputRange），那么鼠标移动的范围将自动限制在这个范围之内（不过此选项默认是关闭的）。

至于 VPW 矩阵，对世界坐标和窗口坐标变换有所研究的朋友可能有所了解，现介绍如下：

1、V 表示摄像机的观察矩阵（View Matrix），它的作用是把对象从世界坐标系变换到摄像机坐标系。因此，对于世界坐标系下的坐标值 worldCoord(x0, y0, z0)，如果希望使用观察矩阵 VM 将其变换为摄像机相对坐标系下的坐标值 localCoord(x’, y’, z’)，则有：

localCoord = worldCoord * VM

此外，观察矩阵可以理解为“摄像机在世界坐标系下的变换矩阵的逆矩阵”，因此 Camera 类也专门提供了 getInverseViewMatrix 这样一个函数，它的实际意义是表示摄像机在世界坐标系下的位置。

2、P 表示投影矩阵（Projection Matrix），当我们使用 setProjectionMatrixAsPerspective 之类的函数设置摄像机的投影矩阵时，我们相当于创建了一个视截锥体，并尝试把包含在其中的场景对象投影到镜头平面上来。如果投影矩阵为 PM，而得到的投影坐标为 projCoord(x”, y”, 0)的话，那么：

projCoord = localCoord * PM

3、W 表示视口矩阵（Window Matrix），它负责把投影坐标变换到指定的二维视口中去，对于视口矩阵 WM，通过下面的公式可以得到最终的窗口坐标 windowCoord(x, y, 0)：

windowCoord = projCoord * WM

将所有的公式整合之后，得到：

windowCoord = worldCoord * VM * PM * WM

而这个所谓的窗口坐标 windowCoord，实际上也就是世界坐标系下的坐标值 worldCoord 在指定的摄像机视口中（也就是我们的屏幕上）对应的平面位置。怎么样，不知不觉中，我们已经实现了 gluProject 函数所完成的功能了，而反转这三个步骤就可以得到视口中指定位置所对应的世界坐标了（也就是 gluUnProject 的工作）。

上面那么一大段论述对于精通数学的您来说可能是废话，不过能够了解 OpenGL 函数 gluProject 和 gluUnProject 的原理（它们其实就是这样实现的，信不信由你），相信我们还是有所斩获的。好了，折腾了这么久，总算明白了 VPW 矩阵的概念，那么代码中这一段也就很好理解了：

```
osg::Matrix masterCameraVPW =
   getCamera()->getViewMatrix() * getCamera()->getProjectionMatrix();
if (getCamera()->getViewport())
{
```

```
    osg::Viewport* viewport = getCamera()->getViewport();
    masterCameraVPW *= viewport->computeWindowMatrix();
    ……
}
```

继续开拔之前，今日让我们先休息一下。

解读成果：

osgGA::EventQueue，VPW 矩阵。

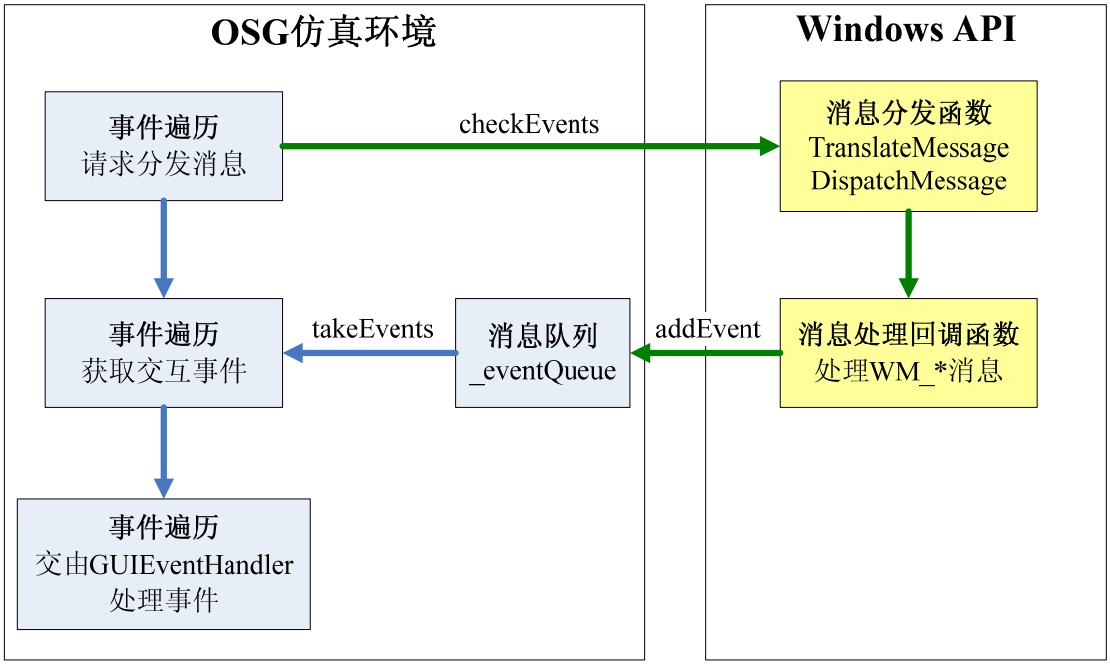
悬疑列表：

类变量_cameraWithFocus 的意义是什么？如何调度和实现 OSG 的多线程机制？

第八日

当前位置：osgViewer/Viewer.cpp 第 577 行，osgViewer::Viewer::eventTraversal()

新的工作开展了，首先我们要使用 Viewer::getContexts 函数找到视景器中所有已有的 GraphicsWindow 图形窗口，然后执行 GraphicsWindowWin32::checkEvents 函数（为什么直接指向了 GraphicsWindowWin32 的实例，请参阅第四日的内容，还有我们假设是在 Win32 平台下进行分析），看看这个函数的内容……熟悉 Win32 SDK 编程的朋友一定又大喜过望了，TranslateMessage，DispatchMessage！多么熟悉的消息传递函数啊，而它们的工作当然也很明确：通知 Windows 执行窗口的消息回调函数，进而执行用户交互和系统消息的检查函数 GraphicsWindowWin32::handleNativeWindowingEvent。而这个函数的作用在第六日就已经提过，它负责把 WM_*消息转化并传递给 osgGA::EventQueue 消息队列。怎么样，其实 OSG 与操作系统 API 还真是“千里姻缘一线牵”呢，如下图所示：



之后，使用 EventQueue::takeEvents 函数，把当前 GraphicsWindow 图形窗口对象 gw 的事件队列保存到指定的变量 gw_events 中，代码如下：

```
osgGA::EventQueue::Events gw_events;
gw->getEventQueue()->takeEvents(gw_events);
```

注意这个 takeEvents 函数除了将所有的事件取出之外，还负责清空事件队列，这样下一帧到来后我们就不必再重复处理旧的事件了。

下一步，遍历刚刚所得的所有事件（天啊，到底要做什么），对于每一个 GUIEventAdapter 事件对象 event：

1、首先处理 Y 轴的方向问题，通常的 GUI 窗口系统都会将屏幕左上角定义为(0, 0)，右下角定义为(Xmax, Ymax)，即 GUIEventAdapter::Y_INCREASING_DOWNWARDS 宏的含义，此时由 GUI 系统传回到 OSG 的坐标位置也是符合这一坐标系的；但是 OSG 的视口坐标系定义为左下角(0, 0)，右上角(Xmax, Ymax)，也就是 Y_INCREASING_DOWNWARDS。此时，有必要对每个 event 对象的鼠标坐标值做一步转换。

2、对于符合下列条件的摄像机，设置为“焦点摄像机”（Camera with Focus），即第二日悬疑列表中的_cameraWithFocus：

（1）鼠标移动（而非拖动）时，鼠标进入了摄像机的视口范围之内；

（2）摄像机是对应该视口的，允许接收事件（Camera::getAllowEventFocus），且“渲染目标实现方式”（Render Target Implementation）为 Camera::FRAME_BUFFER……

又有新问题了，什么是“渲染目标实现方式”？先放到这里，之后我们再深究。

现在我们终于有了“焦点摄像机”，也就是鼠标当前位置所激活的摄像机窗口。不过这并未打消我们的疑问：这个焦点摄像机可以用来做什么呢？继续看下面的代码：

```
osg::Viewport* viewport = getCameraWithFocus()->getViewport();
osg::Matrix localCameraVPW = getCameraWithFocus()->getViewMatrix() *
  getCameraWithFocus()->getProjectionMatrix();
if (viewport) localCameraVPW *= viewport->computeWindowMatrix();
osg::Matrix matrix( osg::Matrix::inverse(localCameraVPW) * masterCameraVPW );
osg::Vec3d new_coord = osg::Vec3d(x,y,0.0) * matrix;
x = new_coord.x();
y = new_coord.y();
……
```

有了上一日的经验，现在我们不再惧怕那个 VPW 矩阵了。毫无疑问，这段代码中首先得到了焦点摄像机的 VPW 矩阵 localCameraVPW。由于已知鼠标在屏幕坐标中的位置(x, y)，又已知主摄像机的 masterCameraVPW 和焦点摄像机的 localCameraVPW，根据：

视口坐标 = 世界坐标 * VPW

并结合上面的代码可以判断出，new_coord 实际上是焦点摄像机中的(x, y)经过转换之后，在主摄像机视口中的坐标位置，而这个位置被重新写入所有事件的 setX，setY 函数。也就是说，当我们在重写后的 GUIEventHandler::handle 函数中使用 ea.getX 和 ea.getY 获取鼠标位置时，这个位置很可能已经经过了一次换算，并非实际的像素位置！

感兴趣的读者不妨改写一下 osgcamera 例子，派生一个新的 GUIEventHandler 事件处理器并在其中读取鼠标坐标值。当我们使用以下参数启动 osgcamera 时：

```
osgcamera.exe -1
```

可以颇为惊异地发现，所得到的坐标范围在(-1, -1)到(1, 1)之间！因为 osgcamera 有可能不建立主摄像机视口（参见第二日的内容），这无疑造成了我们不希望看到的结果。也许这个让人有些困惑（也可能对您来说正好提供了方便）的转换会在以后得到更合理的改进；当然，您也可以自己即刻手动重写 eventTraversal 函数，获取您真正想要的坐标值。

当前位置：osgViewer/Viewer.cpp 第 697 行，osgViewer::Viewer::eventTraversal()

在结束一日的征程之前，我们再看一看 OSG 针对单一窗口的关闭做了什么处理：

```
bool wasThreading = areThreadsRunning();
if (wasThreading) stopThreading();
gw->close();
if (wasThreading) startThreading();
```

当我们选择关闭一个 GraphicsWindow 窗口 gw 时，OSG 系统必须首先尝试终止所有的渲染线程（包括各个图形设备开启的线程和渲染器开启的线程，参见第五日），，关闭窗口之后再打开所有的渲染线程。事实上，当我们试图在运行时开启一个新的 OSG 图形窗口时，也必须使用相同的线程控制步骤，即，关闭线程，创建新渲染窗口，开启线程。否则很可能造成系统的崩溃（这同样涉及到 OSG 的多线程机制，目前它还在悬疑列表中沉睡）。

在这个大型循环体（577-724 行）的最后，所有来自图形窗口和视景器的事件都被添加到一个 std::list 链表当中，下一步我们可以统一处理这些交互事件了。

解读成果：

osgViewer::GraphicsWindowWin32::checkEvents，osgGA::EventQueue::takeEvents。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？

第九日

当前位置：osgViewer/Viewer.cpp 第 782 行，osgViewer::Viewer::eventTraversal()

现在我们遇到的这一段代码专门用来处理 OSG 的退出事件，对于 osgviewer 等例子而言，有一个专门的退出键 Esc，按下后即可直接退出 OSG。

对于很多成型的仿真软件来说，按下 Esc 就退出程序的做法显然不太现实，此时我们可以使用 ViewerBase::setQuitEventSetsDone 设置是否允许按下某个键之后直接退出这种做法，同时还可以使用另一个函数 ViewerBase::setKeyEventSetsDone 来设置自定义的退出键（缺省是 GUIEventAdapter::KEY_Escape）。

ViewerBase::setDone 函数可以随时帮助我们结束仿真程序，因此不必担心退出键被屏蔽之后会带来什么后果。

下一步我们再次遍历所有得到的事件，这里面可能包括用户的鼠标与键盘操作事件，也可能由系统产生的窗口尺寸变化事件，或者程序的关闭事件。对于遍历过程中得到的每一个 GUIEventAdapter 事件对象 event，把它传递给每一个注册的 GUIEventHandler 事件处理器：

```
for(EventHandlers::iterator hitr = _eventHandlers.begin();
    hitr != _eventHandlers.end();
    ++hitr)
{
  (*hitr)->handleWithCheckAgainstIgnoreHandledEventsMask( *event, *this, 0, 0);
}
```

不必关心那个名字又长又拗口的执行函数，它的实质内容其实一看便知。

当前位置：osgGA/GUIEventHandler 第 73 行，osgGA::GUIEventHandler::handleWithCheckAgainstIgnoreHandledEventsMask()

这个函数的主要工作是执行 GUIEventHandler::handle 函数，后者在用户自定义的事件处理器中是必须被重写的。当 handle 函数返回值为 true 时，表示此事件已经处理完毕，其它

事件处理器不会再理会这个事件；返回 false 则继续交由后继的 GUIEventHandler 对象处理。没有特殊需要的话，请不要重写 handleWithCheckAgainstIgnoreHandledEventsMask 函数。

在新的 OSG 版本中，如果希望某个事件处理器暂时不要处理某一事件，可以使用 GUIEventHandler::setIgnoreHandledEventsMask 函数，传入 GUIEventAdapter::EventType 类型的参数（或者通过“或”运算传入多个 EventType 类型），就可以禁止处理器处理此类事件了。

在事件处理器组_eventHandlers 处理过事件之后，场景漫游器_cameraManipulator 将再次考察同一事件，TrackballManipulator 等漫游器就是在这个时候执行场景的平移、旋转和缩放动作的。因此，对于鼠标/键盘事件，请不要轻易设置 handle 函数的返回值为 true，因为那样将屏蔽漫游器对这些事件的处理。

当前位置：osgViewer/Viewer.cpp 第 827 行，osgViewer::Viewer::eventTraversal()

通过阅读 osgcallback 例子以及各种相关的教程，我们早已知道 OSG 中存在着各种各样的回调（Callback），其中以事件回调（EventCallback）和更新回调（UpdateCallback）最为常见。对于场景图形中的任何节点，以及 Geode 叶节点所包含的任何 Drawable 几何体，我们都可以使用 setEventCallback 和 setUpdateCallback 函数设置它的事件回调与更新回调对象。这些回调对象的调用时机就在 eventTraversal 函数中。

场景节点的回调对象必须继承自 osg::NodeCallback，并重写 NodeCallback::operator()函数以实现回调的具体内容。由此有所不同的是，Drawable 对象的事件回调必须继承自 Drawable::EventCallback，并具现 EventCallback::event 函数的内容；其更新回调则必须继承 Drawable::UpdateCallback 并具现 UpdateCallback::update 函数。

为了正确地遍历场景的节点和几何体对象，并执行所有可能的事件回调和更新回调，OSG 使用访问器（Visitor）机制来处理场景图形的访问工作。这其中，_eventVisitor 就是负责管理事件回调的遍历工作的（下一日介绍的_updateVisitor 负责更新遍历）。这个变量的值会自动初始化，当然我们也可以用 ViewerBase::setEventVisitor 加载我们自定义的事件访问器，前提是这个访问器要继承自 osgGA::EventVisitor 类。

在事件回调的处理函数中（operator()或者 event），我们可以通过读取第二个传入参数，并调用 EventVisitor::getEvents 函数来获取当前发生的事件。所有的交互和系统事件都会一次又一次地触发事件回调，因此编写这个回调的内容时请一定要慎重，不然会大幅度地降低系统的性能。

在遍历场景节点并执行其事件回调之后，OSG 还要转至主摄像机_camera 和从摄像机组_slaves，再次执行它们的事件回调对象，注意这里的实现代码：

```
osg::NodeVisitor::TraversalMode tm = _eventVisitor->getTraversalMode();
_eventVisitor->setTraversalMode(osg::NodeVisitor::TRAVERSE_NONE);
if (_camera.valid() && _camera->getEventCallback()) _camera->accept(*_eventVisitor);
for(unsigned int i=0; i<getNumSlaves(); ++i)
{
  osg::Camera* camera = getSlave(i)._camera.get();
  if (camera && camera->getEventCallback()) camera->accept(*_eventVisitor);
}
_eventVisitor->setTraversalMode(tm);
```

依然使用访问器，但是设置访问器不要向下遍历节点（因为 Camera 同样可以作为场景的一个中间节点），在访问过所有摄像机之后再恢复访问器的原有值。

最后，计算事件遍历的结束时间，将相关的时刻信息保存到记录器中，休息一下，准备

向下一个函数 updateTraversal 迈进。

解读成果：

osgGA::GUIEventHandler，osgViewer::Viewer::eventTraversal()。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？

第十日

当前位置：osgViewer/Viewer.cpp 第 873 行，osgViewer::Viewer::updateTraversal()

OSG 更新回调的作用与事件回调有类似之处：由专门的访问器对象_updateVisitor 的负责场景图形更新遍历；所有的节点和 Drawable 几何体对象都可以使用 setUpdateCallback 设置更新回调；通过具现 NodeCallback::operator()或者 Drawable::UpdateCallback::update 函数，可以在回调对象中添加自定义的工作。

但是，更新回调与事件回调最大的不同在于：每当一个用户交互或系统事件产生时，每一个节点（以及 Drawable 对象）的事件回调都会被调用一次；而节点（以及 Drawable 对象）的更新回调只会在每帧中被调用一次。这一区别决定了我们应当在什么时候使用事件回调，以及在什么时候使用更新回调。

OSG 的更新遍历函数 updateTraversal 在系统每帧的执行过程中有着重要的地位，除了处理用户的更新回调对象之外，还要负责更新摄像机的位置，并且更新分页数据库 DatabasePager 和图像库 ImagePager 的内容。这里我们首先对它的流程做一个概述：

1、获取函数的起始时刻。

2、使用预设的更新访问器_updateVisitor，访问场景图形的根节点并遍历其子节点，实现各个节点和 Drawable 对象的更新回调。

3、使用 DatabasePager::updateSceneGraph 函数以及 ImagePager::updateSceneGraph 函数，分别更新场景的分页数据库和分页图像库。

4、处理用户定义的更新工作队列_updateOperations。

5、执行主摄像机_camera 以及从摄像机组_slaves 的更新回调（但是不会遍历到它们的子节点），注意摄像机回调的执行时机与场景节点还是有所区别的。

6、根据漫游器_cameraManipulator 的位置姿态矩阵，更新主摄像机_camera 的观察矩阵。注意这里使用的函数是 MatrixManipulator::getInverseMatrix，根据第七日介绍的内容，摄像机在世界坐标系中的位置姿态矩阵，即是其观察矩阵的逆矩阵，由是可得。

7、使用 View::updateSlaves 函数更新从摄像机组_slaves 中所有摄像机的投影矩阵，观察矩阵和场景筛选设置（CullSettings）。

8、获取函数的结束时刻，将相关的时刻信息保存到记录器中。

不要以为我们对这个函数的介绍就这么草草地完结了。事实上，在浏览更新遍历的执行流程时，我们已经遇到了一大堆之前没有提及的概念，如果不从中好好地发掘一番，“最长的一帧”这个标题恐怕就徒有虚名了（其实本来就徒有虚名）。

首先是更新工作队列_updateOperations 的问题。第四日中我们曾介绍过可以使用 ViewerBase::setRealizeOperation 来自定义的场景预处理工作，更新工作的作用及实现方法与之相似，均需要继承 osg::Operation 并重写 operator()操作符，以实现一些特定的用户操作。不过在这里，用户可以使用 ViewerBase::addUpdateOperation 写入多组更新处理器 Operation 对象（或者使用对应的成员函数 removeUpdateOperation 来移除），因此可以实现的操作也更

加丰富。

注意，习惯上 Operation 对象可以用来处理与图形设备以及 OpenGL 底层 API 相关的工作，不过用户层级的工作在这里实现通常也没有问题。

当前位置：osgViewer/View.cpp 第 148 行，osgViewer::View::updateSlave()

其次是从摄像机组_slaves 的更新。这里我们可以看出从摄像机组与主摄像机的关系：从摄像机组从本质上继承了主摄像机的投影矩阵，观察矩阵和场景筛选设置，但是可以在使用 View::addSlave 添加从摄像机时，设置投影矩阵与观察矩阵的偏置值，还可以使用 CullSettings::setInheritanceMask 设置 CullSettings 的继承掩码（注意 osg::Camera 类继承自 osg::CullSettings 类）。

那么，CullSettings 是什么呢？顾名思义就是场景筛选（Cull，国内通常称为“场景裁减”）的设置选项了。这里所谓的场景筛选大家恐怕都早已耳熟能详，简单来说的话，就是在大规模的场景当中，略去那些不会被观察者所见到的几何体，从而大幅度提升渲染的效率。

我们能想到的最简单的筛选方法是背面筛选（back-face culling，背面裁减），也就是忽略所有几何体背向观察者的一面，场景的多边形复杂度最多将降低 1/2，这种筛选方式已经由 OpenGL 实现了（GL_CULLFACE），可以使用 OSG 渲染属性 osg::CullFace 开启。

另一种值得一提的筛选方式是视锥体筛选（view frustum culling），将场景中每个物体的包围盒与视锥进行比较，并剔除视锥之外的物体，这种筛选方式有时可以将场景的多边形复杂度降低为原来的 1/4。

当然还有剪切平面（clip plane）的方法，不过 OpenGL 中已经使用 glClipPlane 实现了这种场景筛选的功能，而 OSG 可以使用渲染属性 osg::ClipPlane 来设置相应的参数。

对了，相信大家还想到了另一种高效的筛选方法：遮挡筛选（occlusion culling）。也就是将那些被其它物体所遮挡的对象从场景中剔除。不过这是一种比较复杂的场景筛选手段，如果用在整个场景中反而会增大计算的开销，因此往往作为一个预处理步骤出现。在 OSG 2.3.x 及以后的版本中，开始提供遮挡筛选的算法。

OSG 目前所支持的场景筛选方式（即 CullSettings::CullingModeValues 枚举量）包括：

NO_CULLING：也就是不进行场景筛选，如果您希望自己实现场景筛选的功能，或者由于某种特殊原因不希望执行筛选工作，可以选择这一项。

NEAR_PLANE_CULLING：近平面筛选，将超出近平面范围以外的对象剔除。

FAR_PLANE_CULLING：远平面筛选，将超出远平面范围以外的对象剔除。

VIEW_FRUSTUM_CULLING：视锥体筛选，它保证场景中只有视锥体范围内（包括视锥，近平面和远平面）的一小部分节点和 Drawable 对象是可见的，因而加速场景的绘制速度。

VIEW_FRUSTUM_SIDES_CULLING：视锥体侧面筛选，它不执行近平面/远平面的筛选工作，除此之外与 VIEW_FRUSTUM_CULLING 没有区别。

SMALL_FEATURE_CULLING：细节筛选，场景中某些物体对于观察者而言可能是十分微小的，足以忽略不计，此时可以用细节筛选特性将它们剔除。判断对象是否足够微细的阈值由 CullSettings::setSmallFeatureCullingPixelSize 设定。注意这种筛选可能会剔除一些必要的信息（比如用户在屏幕上绘制了一些点，却发现它们统统被吞噬掉了），此时可以强制设置几何体对象的包围盒大小（Drawable::setInitialBound），或者关闭细节筛选特性。

SHADOW_OCCLUSION_CULLING：遮挡筛选，虽然需要用户手动去设置诸多的几何信息，但是 OSG 的确出色地实现并继续完善本影遮挡筛选（即完全遮挡）的算法。在 osgoccluder 例子中，您可以仔细揣摩其中有关 ConvexPlanarOccluder 和 OccluderNode 的用法。

CLUSTER_CULLING：聚集筛选，这是一种类似于背面筛选的场景筛选方法，但是它可以将多个对象组合起来并进行统一的背面筛选，OSG 中，目前可以使用筛选回调 ClusterCullingCallback 来实现节点的聚集筛选（对节点使用 Node::setCullCallback），这在地球地理信息的裁减时尤为适用。

DEFAULT_CULLING：缺省方式下会自动打开视锥体侧面筛选，细节筛选，遮挡筛选和聚集筛选的选项，不过后两者还需要编写额外的代码以实现功能。

ENABLE_ALL_CULLING：开启全部筛选方式，包括视锥体筛选，细节筛选，遮挡筛选和聚集筛选。

设置一种或多种筛选方式可以使用 CullSettings::setCullingMode 函数（采用“或”运算来指定多种筛选方式），如果希望暂时屏蔽某一种筛选方式，可以如下编写代码：

```
camera->setCullingMode(camera->getCullingMode() &
  ~osg::CullSettings::SMALL_FEATURE_CULLING);
```

有关各种筛选方式的详细算法，可以参考 CullingSet::isCulled 函数，有关遮挡筛选和聚集筛选的算法请单独参阅 OccluderNode，ConvexPlanarOccluder 和 ClusterCullingCallback 等相关类的实现代码。

解读成果：

View::updateSlaves，场景筛选方式。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？

第十一日

当前位置：osgViewer/Viewer.cpp 第 889 行，osgViewer::Viewer::updateTraversal()

上一日的“解读成果”中并没有标明 updateTraversal 函数，原因很简单：因为有关这个函数的解读还没有结束。至于是哪些问题还没有得到解答，相信您也心知肚明了，对，就是 DatabasePager！它的工作原理，流程，以及在场景图形中起到的作用。也许您早已对此烂熟于胸，也许正好也想把这个有名的 OSG 工具剖析剖析，那么机会来了：就让我们一起用上两到三日的时间，仔细地解读一下这个所谓“分页数据库”到底是什么来头。

“悬疑列表”中的那些数据好像沉积了很久了……不过还是再忍耐一下，不要担心会不会变质或者无法兑现的问题（^_^）。

在解读 DatabasePager 之前，我们或许需要一点预备知识：也就是 OpenSceneGraph 中的线程处理库 OpenThreads。

面向对象的跨平台线程库 OpenThreads 原本是独立的开源工程，OSG 2.x 以后的版本将其纳入了自己的体系结构当中，成为 OSG 基本库的一份子，目前最新的版本为 2.3。

OpenThreads 库包含了以下几个最主要的线程处理类：

Thread 类：线程实现类。它是一个面向对象的线程实现接口，每定义一个 Thread 类，就相当于定义了一个共享进程资源，但是可以独立调度的线程。通过重写 run()和 cancel()这两个成员函数，即可实现线程运行时和取消时的操作；通过调用 start()和 cancel()，可以启动或中止已经定义的进程对象。

Mutex 类：互斥体接口类。如同 pthread 等常用的线程库那样，OpenThreads 也提供了互斥体操作的机制，它有效地避免了各个线程对同一资源的相互竞争，即，某一线程欲操作某一共享资源时，首先使用互斥体成员的 lock()函数加锁，操作完成之后再使用 unlock 函数解

锁。一个线程类中可以存在多个 Mutex 成员，用于在不同的地点或情形下为共享区域加锁；但是一定要在适当的时候解锁，以免造成线程的共享数据无法再访问。

Condition 类：条件量接口类。它依赖于某个 Mutex 互斥体，互斥体加锁时阻塞所在的线程，解锁或者超过时限则释放此线程，允许其继续运行。

这里涉及了几个线程操作中重要的概念：同步，阻塞以及条件变量。线程同步，简单来说就是使同一进程的多个线程可以协调工作，例如让它们都在指定的执行点等待对方，直到全员到期之后才开始同步运行；拥塞，即强制一个线程在某个执行点上等待，直到满足继续运行的条件为止。例如其它的线程到达同一执行点，某个变量初始化完成等等，可以通过条件变量来设计各种条件。

Block 类：阻塞器类。顾名思义，这个类的作用就是阻塞线程的执行，使用 block()阻塞执行它的线程（注意，不一定是定义它的 Thread 线程，而是当前执行了 block 函数的线程，包括系统主进程），并使用 release()释放之前被阻塞的线程。

下图所示的代码实现了一个最简单的线程，并演示了 Block 类的使用方法。运行程序后可以发现，Block::block()函数将首先阻塞主进程，被释放后再次阻塞的是 TestThread 线程，这与它是谁的成员变量并无关系。

```
class TestThread : public OpenThreads::Thread
{
public:
    TestThread() : _done(false), _count(0) {}
    ~TestThread() { cancel(); }
    void block() { _operator.block(); }
    virtual int cancel()
    {
        _operator.release();
        _done = true;
        while( isRunning() )
            OpenThreads::Thread::YieldCurrentThread();
        return 0;
    }
    virtual void run()
    {
        do
        {
            std::cout << "(" << _count << ")";
            ++_count;

            if ( _count==10 )
            {
                _operator.release();
                _operator.reset();
                _operator.block();
            }
            microSleep( 150000L );
        } while( !_done );
    }
protected:
    bool _done;
    unsigned long _count;
    OpenThreads::Block _operator;
};
```

```
int main(int argc, char** argv)
{
    TestThread t;
    t.start();
    t.block();
    std::cout << "(Main)" << std::endl;
    getchar();
    t.cancel();
    return 0;
}
```

BlockCount 类：计数阻塞器类。它与阻塞器类的使用方法基本相同：block()阻塞线程，release()释放线程；不过除此之外，BlockCount 的构造函数还可以设置一个阻塞计数值。计数的作用是：每当阻塞器对象的 completed()函数被执行一次，计数器就减一，直至减到零就释放被阻塞的线程。

Barrier 类：线程栅栏类。这是一个对于线程同步颇为重要的阻塞器接口，它的构造函数与 BlockCount 类似，可以设置一个整数值，我们可以把这个值理解成栅栏的“强度”。每个执行了 Barrier::block()函数的线程都将被阻塞；当被阻塞在栅栏处的线程达到指定的数目

时，就好比栅栏无法支撑那么大的强度一样，栅栏将被冲开，所有的线程将被释放。重要的是，这些线程是几乎同时释放的，也就保证了线程执行的同步性。

注意 BlockCount 与 Barrier 的区别，前者是由其它任意线程执行指定次数的 completed() 函数，即可释放被阻塞的线程；而后者则是必须阻塞指定个数的线程之后，所有的线程才会同时被释放。

ScopedLock 模板：这个模板是与 Mutex 配合出现的，它的作用域之内将对共享资源进行加锁，作用域之外则自动解锁，代码格式如下：

```
{
  OpenThreads::ScopedLock<OpenThreads::Mutex> lock(_mutex);
  ……
}
……
```

在大括号范围内，进程的共享资源被当前进程锁定，超过范围则自动解锁。

当前位置：include/osgDB/ DatabasePager 第 250 行，
osgDB:: DatabasePager::updateSceneGraph()

updateSceneGraph 函数的工作是更新分页数据库的内容，它的内容简单到只包含了两个执行函数的内容：

1、DatabasePager::removeExpiredSubgraphs：用于去除已经过期的场景子树；

2、DatabasePager::addLoadedDataToSceneGraph：用于向场景图形中添加新载入的数据。

有关这两个函数的内容介绍我们先放在一边，因为现在即使通读它们的内容，恐怕也很难找出什么有用的线索来。DatabasePager 类的那些繁多的成员变量就已经让我们应接不暇了，有必要分出主次，放弃阅读那些没有对主要功能直接做出贡献的变量和函数内容，才能够顺利完成这次的任务。

那么，我们首先来了解一下 DatabasePager 的概念。

数据库的分页技术在很多领域都十分常见。例如网络上的海量数据库搜索，往往会采取分页的方式，只搜索数据库的一部分内容；当用户浏览到后面的页面之后，再继续搜索对应的内容。

在三维场景的浏览中同样会面对这个问题，如果用户需要浏览的数据量很大，比如地形模拟，虚拟小区和城市，甚至是虚拟地球的工程中，都不可避免地要使用到场景数据库的分页技术，否则将对计算机系统产生极大的负担。

在 OSG 中， osgDB::DatabasePager 类执行的就是这一工作：每一帧的更新遍历执行到 updateSceneGraph 函数时，都会自动将“一段时间之内始终不在当前页面上”的场景子树去除，并将“新载入到当前页面”的场景子树加入渲染，这里所说的“页面”往往指的就是用户的视野范围。这些分页和节点管理的工作如果由渲染循环来完成的话，恐怕是费时又费力的，对于场景的显示速度有较大的影响，因此，DatabasePager 中内置了专用于相关工作处理的 DatabaseThread 线程，也就是我们下一日将要讲解的主要内容。

解读成果：

OpenThreads 库。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？

第十二日

当前位置：osgDB/DatabasePager.cpp 第 407 行，
osgDB:: DatabasePager::DatabaseThread::run ()

在讲解 DatabaseThread 线程之前，我们理应先仔细考虑一下，OSG 的分页数据库应该使用单独的线程来处理什么：

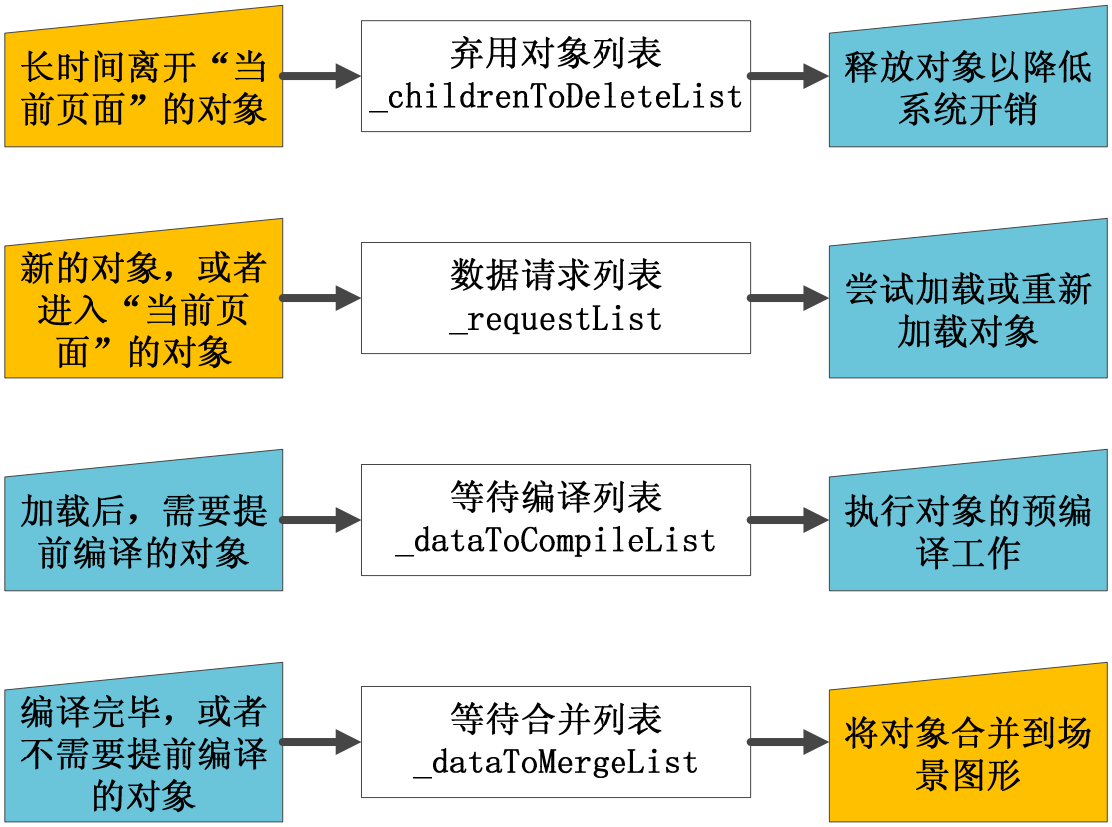
1、删除过期的场景数据：这一步工作当然也可以在仿真循环中进行，但是这样做很可能会造成场景渲染的延迟，我们采用线程来处理场景数据的理由也正是因为如此。过期对象的统一删除工作在这里完成，而更新遍历则负责将检索到的对象送入相应的过期对象列表。

2、获取新的数据加载请求：请求加载的可能是新的数据信息，也可能是已有的场景数据（曾经从“当前页面”中去除，更新又回到“当前页面”中）；可能是本地的数据文件，也可能来自网络，并需要把下载的数据缓存在本地磁盘上。这些都需要在线程中一一加以判断。

3、编译加载的数据：有些数据如果提前进行编译可以有效地提升效率，例如为几何体数据创建显示列表（Display List），以及将纹理对象提前加载到纹理内存；虽然 OSG 同样可以在渲染时根据用户需要执行这些工作，但是那样势必会造成帧的延迟，对于大型场景的加载来说这种延迟将更为严重。因此预编译加载的数据是很有必要的。在数据处理线程执行预编译工作当然不为过，但是如果系统配置足够高级的话，也可以选择由图形设备线程（GraphicsContext::getGraphicsThread）来完成这些原属于它们的工作。

4、将加载的数据合并至场景图形：直接由线程来完成这一工作显然是不合适的，因为我们不知道当 DatabaseThread 线程试图操作场景中的节点时，OSG 的渲染器在做些什么。最好的方法是将读入的数据先保存在一个列表中，并且由仿真循环负责获取和执行合并新节点的操作。

那么，我们就得到了一个也许可行的数据流图，如下所示：



左侧的图框表示数据的检索和输入，中间的白色图框表示用于数据存储的内存空间，而右边的图框表示存储数据的输出。此外，蓝绿色图框表示可以在 DatabaseThread 线程中完成的工作，而橙色图框表示由线程之外的函数完成的工作。

这幅图中事实上已经标示出了 DatabasePager 中的几个重要成员变量。不过在认识它们之前，我们还需要了解一下 DatabasePager 类所定义的各种数据结构：

1、DatabasePager::DatabaseThread 类：这是分页数据库的核心处理线程，它负责实现场景元素的定期清理，加载以及合并工作；但是让它一直处于检查各个数据列表的循环状态，这未免太过耗费系统资源。因此，这个线程在平常状态下应当被阻塞，需要时再予以唤醒。

2、DatabasePager::DatabaseRequest 结构体：这个结构体保存了用户的单个数据请求，包括数据文件名，请求时间，数据加载后存入的节点，以及要进行合并的父节点等；除此之外还有一个重要的编译映射表_dataToCompileMap，这个映射表负责保存图形设备 ID 与编译对象（几何体显示列表，纹理等）的映射关系。

3、DatabasePager::RequestQueue 结构体：它负责保存和管理一个“数据请求列表”_requestList，也就是由 DatabaseRequest 对象组成的向量组，除此之外还负责对列表中的数据按请求时间排序。上图中所示的_dataToCompileList 和_dataToMergeList 实际上都是 RequestQueue 类型的对象，不过它们所保存的“请求列表”事实上是已经完成加载的“待编译/待合并列表”了。

4、DatabasePager::ReadQueue 结构体：这个结构体继承自 RequestQueue，不过还增加了一个“弃用对象列表”_childrenToDeleteList，也就是 osg::Object 对象组成的向量组。它是数据处理线程中最重要的对象之一，除了可以随时向两个列表里追加数据请求和弃用对象之外，这个结构体还包括了一个 updateBlock 函数，负责阻塞或者放行 DatabaseThread 线程，其根据是：列表中是否存在新的数据请求或弃用对象需要处理，以及用户是否通过函数设置暂时不要启用线程（DatabasePager ::setDatabasePagerThreadPause）。

OSG 2.6 版本的 DatabasePager 中缺省创建了两个数据处理线程，均保存在线程列表_databaseThreads 中，这两个线程分别负责处理来自本地文件的数据队列_fileRequestQueue 和来自 HTTP 站点的网络数据队列_httpRequestQueue（这两个队列均为 ReadQueue 对象）。来自 DatabaseThread 线程之外的文件加载请求被本地文件处理线程所接收；如果从文件名（DatabaseRequest::_fileName）判断出是来自网络的文件的话，则转交给网络数据处理线程。

来自网络的数据也可以被缓存到本地，方法是设置系统变量 OSG_FILE_CACHE 为本地路径。

当前位置：osgDB/DatabasePager.cpp 第 451 行，
osgDB:: DatabasePager::DatabaseThread::run ()

现在我们可以进入线程循环体内部浏览了。每次循环开始时，数据处理线程都被自动阻塞，避免无谓的系统消耗；直到 updateBlock 函数在外部被执行才会放行，继续下面的代码。

updateBlock 函数可能在以下几种情形下被执行：

1、ReadQueue 对象中的“数据请求列表”被修改，例如新的数据加载请求被传入，请求被取出，列表被重置。

2、ReadQueue 对象中的“弃用对象列表” 被修改，例如有新的过期对象被送入，对象被删除，列表被重置。

3、执行了 DatabasePager ::setDatabasePagerThreadPause 函数，当线程被重新启动时，会自动检查线程是否应当被唤醒。

这之后是过期数据的删除工作，即取出_childrenToDeleteList 中的所有对象，并安全地将它们析构（462-471 行）。

随后，使用 DatabasePager::ReadQueue::takeFirst 函数，从当前线程对应的 ReadQueue 对象（_fileRequestQueue 或_httpRequestQueue）的队列中取出并清除第一个数据加载请求（DatabaseRequest），而下一步的工作就是处理这个请求了。

解读成果：

DatabasePager 数据流，重要变量。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？

第十三日

当前位置：osgDB/DatabasePager.cpp 第 479 行，
osgDB::DatabasePager::DatabaseThread::run ()

这里首先要注意一点，OSG 2.6 的版本会自动判断数据加载请求是否已经过期，即，请求从外部程序发出的时间是否为当前帧或者上一帧，如果这个 DatabaseRequest 请求是很多帧以前就发出的，那么 DatabaseThread 就不会再处理它了。这种判断方式显然是有利也有弊的，因为必须仔细考虑数据处理线程与主进程的同步问题。更新版本的 OSG 对此已经有了改动，不再严格地按照“发出时间是否为当前帧或者上一帧”的条件来进行处理。

下一步，如果判断请求加载的文件来自 HTTP 网络，且已经设置了系统变量 OSG_FILE_CACHE，那么“本地文件处理线程”将尝试建立同名的文件缓存路径，并将请求转交给 “网络文件处理线程”的_httpRequestQueue 进行处理（516-547 行）。

使用 DatabaseThread::dpReadRefNodeFile 加载文件，我们判定 OSG 的 reaNodeFile 函数是线程安全的，因此 dpReadRefNodeFile 函数的实现与 osgDB:: reaNodeFile 没有太大差异（585-589 行）。

加载的过程可长可短，由于是在数据处理线程中完成的，因此不会影响到主场景的渲染和遍历。加载完毕之后，网络数据还将使用 osgDB::writeNodeFile 缓存到本地文件（591-615 行）。

然后我们尝试获取 DatabaseRequest::_groupForAddingLoadedSubgraph 的值，对于已经加载的模型而言，这就是把它合并到当前场景时它的父节点。

下一步的工作大致分为下面几个步骤：

1、获取从这个父节点到场景根节点的一条路径，这实际上可以理解为场景遍历时，从根节点一直到新加载的节点之前的遍历路径，其代码为：

```
osg::NodePath nodePath;
osg::NodePathList nodePathList =
   groupForAddingLoadedSubgraph->getParentalNodePaths();
if (!nodePathList.empty()) nodePath = nodePathList.front();
nodePath.push_back(groupForAddingLoadedSubgraph.get());
```

2、创建场景对象的预编译访问器 DatabasePager::FindCompileableGLObjectsVisitor，这个访问器的主要工作是找到场景树中所有的 StateSet 渲染属性集以及 Drawable 几何体对象，并将它们记录到上一日所述的编译映射表 DatabaseRequest::_dataToCompileMap 中。这样我们就可以在需要的时候调用映射表中的内容，来执行显示列表创建和纹理绑定这两项重要的 OpenGL 预处理工作。

3、将第一步得到的节点路径压入到访问器中，这样相当于从场景的根节点开始执行遍

历。这里这么做的主要目的是，测试新节点合并到场景树之后场景可能产生的变化。这里涉及到一个较少用到的函数 NodeVisitor::pushOntoNodePath，代码如下：

```
for(osg::NodePath::iterator nitr = nodePath.begin(); nitr != nodePath.end(); ++nitr)
  frov.pushOntoNodePath(*nitr);
```

4、一切顺利的话，现在所有需要预编译的对象（StateSet 与 Drawable）都已经记录到“图形设备-编译对象”的映射表_dataToCompileMap 中了。这里 OSG 将需要编译的对象复制到每个 GraphicsContext 图形设备的对应映射像中。由于映射表的每个像值实质上是相同的，建立这个映射表的意义似乎不大；不过它对于以后的扩展（例如，对于不同的 GC 设备，预编译的对象有区别时）还是有很大用途的。

5、还需要注意的是，无论是否使用了预编译访问器 FindCompileableGLObjectsVisitor（可以通过 DatabasePager::setDoPreCompile 设置是否执行对象的预编译），系统都会自动为每个对象创建 k-Dop Tree 包围体（前提是开启了文件加载时的 KdTree 设置，参见 osgkdtree 例子），它的主要作用是为场景的射线交集检测提供一种更加精确的测试工具。

6、最后，将已经加载完成，不过还在等待预编译的数据加载请求（DatabaseRequest）送入待编译列表_dataToCompileList（或者待合并列表_dataToMergeList，如果不需要预编译的话）。

当前位置：osgDB/DatabasePager.cpp 第 729 行，
osgDB::DatabasePager::DatabaseThread::run ()

对于 DatabaseThread 线程的解读似乎逐渐到了尾声，那么我们先加快速度，了解一些还有哪些剩余的工作（根据上一日给出的数据流图来看，似乎不剩什么工作了），并且给我们闲置已久的“悬疑列表”再添加一点新的内容（明明还有很多问题沉积在里面）。

首先从待编译列表_dataToCompileList 中筛选并清除所有已经过期的数据请求，即那些早在当前帧或上一帧之前就已经发出的请求（729-755 行）。

随后，遍历所有的图形窗口设备，检查它们是否注册了线程（除了 SingleThreaded 单线程模型之外，其他几种线程模型均会为每个图形设备创建一个线程）。如果线程有效的话，则向其中加入一个 DatabasePager::CompileOperation 对象；否则按照下面的代码自行处理渲染状态集和几何体对象的编译工作：

```
gc->makeCurrent();
_pager->compileAllGLObjects(*(gc->getState()));
gc->releaseContext();
```

熟悉图形编程的读者一定知道，当我们执行任何 OpenGL 操作之前，都需要先指定一个渲染上下文（Render context）。Win32 环境下我们使用 wglMakeCurrent 来完成这一工作，X11 下则是 glXMakeCurrent。由于 OSG 很好地将这些图形设备和渲染上下文操作指令封装在 GraphicsContext 的子类当中，因此每当我们试图实现指定或释放渲染 RC 的操作时，只需要执行相应图形窗口的 makeCurrent()和 releaseContext()函数即可。

至于 DatabasePager::compileAllGLObjects 函数，通过阅读其中的内容我们可以很快发现，它的工作无非是取出映射表_dataToCompileMap 中的所有 StateSet 和 Drawable 对象，并依次执行 StateSet::compileGLObjects 和 Drawable::compileGLObjects 函数。

有关 compileGLObjects 函数，以及它的传入参数 osg::State 的意义，我们会在阅读渲染线程的过程中再做讲解。

而有关 DatabasePager::CompileOperation 类，根据已有的经验（第四日，第十日）可以知道，它继承自 osg::Operation，并重写了 operator()操作符，以完成指定的图形操作。但是，有关它被追加到图形设备线程之后，又是如何被执行的，这一点同样会在介绍渲染线程时涉

及到。

“待编译列表”中的对象在预编译完成后会转存到“待合并列表”中。

解读成果：

DatabasePager::DatabaseThread::run。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？如何使用 compileGLObjects 完成预编译工作？Operation 对象在线程中的应用时机是什么？

第十四日

当前位置：osgDB/DatabasePager 第 250 行，osgDB::DatabasePager::updateSceneGraph ()

好了，请回到第十一日的故事中来，毕竟我们主要的任务还远未结束。现在我们需要重新审视 DatabasePager::updateSceneGraph 的工作了。

1、DatabasePager::removeExpiredSubgraphs：用于去除已经过期的场景子树。

我们首先遍历 DatabasePager::_pagedLODList 这个成员变量，并执行其中每个 PagedLOD 对象的 removeExpiredChildren 函数，取得其中已经过期的子节点并记录到一个列表里。

将这些过期节点标记为“可删除”，并传递给_fileRequestQueue->_childrenToDeleteList 成员，也就是前文所述的“待删除列表”，同时唤醒 DatabaseThread 线程。

下一步，将过期节点从_pagedLODList 中删除，由于它们已经被传递到“待删除列表”当中，因此 ref_ptr 引用计数不会减到零，也就不会在主仿真循环中触发内存释放（delete）动作。

最后还要执行 SharedStateManager::prune 函数。这里的 osgDB::SharedStateManager 指的是一个渲染状态共享管理器，它负责记录分页数据库中各个节点的渲染属性（StateAttribute），并判断节点之间是否共享了同一个渲染属性，从而节省加载和预编译的时间。prune 函数的工作是从 SharedStateManager 中剔除那些没有被共享的渲染属性。

如果希望启用 SharedStateManager（默认是关闭的，其性能目前可能没有想象的那么好），需要在进入仿真循环之前执行：

```
osgDB::Registry::instance()->getOrCreateSharedStateManager();
```

2、DatabasePager::addLoadedDataToSceneGraph：用于向场景图形中添加新载入的数据。

这里首先取得“待合并列表”_dataToMergeList，并遍历其中每一个 DatabaseRequest 对象。

遍历过程中，首先执行 SharedStateManager::share 函数，将新加载节点_loadedModel 的渲染属性保存到 SharedStateManager 管理器中。

随后执行 DatabasePager::registerPagedLODs，在加载的节点及其子树中搜索 PagedLOD 节点，并添加到刚刚提到的_pagedLODList 列表中。

最后，判断 DatabaseRequest::_groupForAddingLoadedSubgraph 对象（也就是新加载节点在场景中的父节点）是否合法，并将 DatabaseRequest::_loadedModel 添加为它的子节点。

这里反复提到的 PagedLOD 节点，实际上揭开了关于 DatabasePager 的又一个关键问题的帷幕，那就是：什么情形下我们才会用到 DatabasePager？

答案是使用 osg::PagedLOD 和 osg::ProxyNode 节点的时候。下面我们就分别阐述一下这两种类型的节点的功能和用法。

首先是内容较为简单的 ProxyNode 节点，从名字我们大概可以猜测出，它的功能是“代理节点”，就像网络上的代理服务器那样，使用代理节点可以指向某个已经存在的模型文件，并在需要的时候加载它。

事实正是如此，当我们希望在场景仿真循环开始之后才加载某个模型文件时，可以使用 ProxyNode 节点来指定要加载的文件名，并在场景筛选（Cull）的过程中加载模型，加载后的新节点将作为 ProxyNode 节点的子节点。示例代码如下：

```
osg::ProxyNode* proxyNode = new osg:: ProxyNode;
proxyNode->setFileName(0, “nodefile.osg”);
```

这里 setFileName 的两个参数分别是新载入的子节点在 ProxyNode 下的位置，以及对应模型文件的名称。我们还可以使用 ProxyNode::setLoadingExternalReferenceMode 来设置加载的时机，例如首先设置为不自动加载（NO_AUTOMATIC_LOADING），在适当的时候再设置为立即加载（LOAD_IMMEDIATELY）。

然后是 PagedLOD 节点。

有关场景的 LOD（细节层次，Level of Detail）技术，以及 osg::LOD 节点的使用，本教程不会做过多的阐述。osg::PagedLOD 类事实上继承自 osg::LOD，它的工作同样是按照用户的可视范围，将多个子节点作为同一场景的多个细节层次。这样可以在视点靠近物体时呈现较多的物体细节，而在远离时则仅仅呈现一个简化的模型，从而降低了运算和绘制的负担。但是，对于 LOD 节点而言，由于其子树的规模往往是十分复杂而宏伟的，因此在加载时很可能耗费过长的时间，且占用了庞大的系统资源，不利于其它工作的开展。因此 OSG 中为用户提供了 PagedLOD 节点：它运用了分页数据库的功能，将多个模型数据分批加载到场景图形中（作为 PagedLOD 的子节点）；并根据用户当前的可视范围，将那些一段时间内均无法被看到的 PagedLOD 子节点剔除出场景图形，以节约系统资源；当然，如果用户移动了视点之后，被剔除的节点又重新进入视野，那么 OSG 的分页数据库线程将重新加载它。

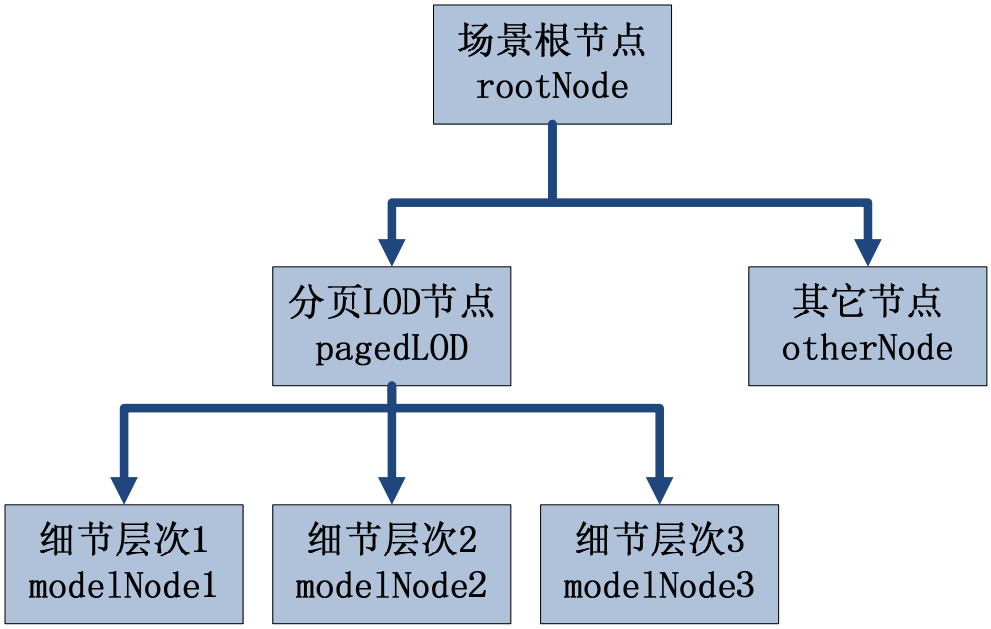
以上所述就是我们使用 DatabasePager 的时机。假设场景中需要四级 Node 节点及其子树来表达某个场景（例如一座城市，一片地形图，或者一架设计精良的战斗机）在不同视距下的细节层次，且这些模型已经保存为诸如“node_1.osg”的文件形式。那么可以采用这样的代码来加载场景：

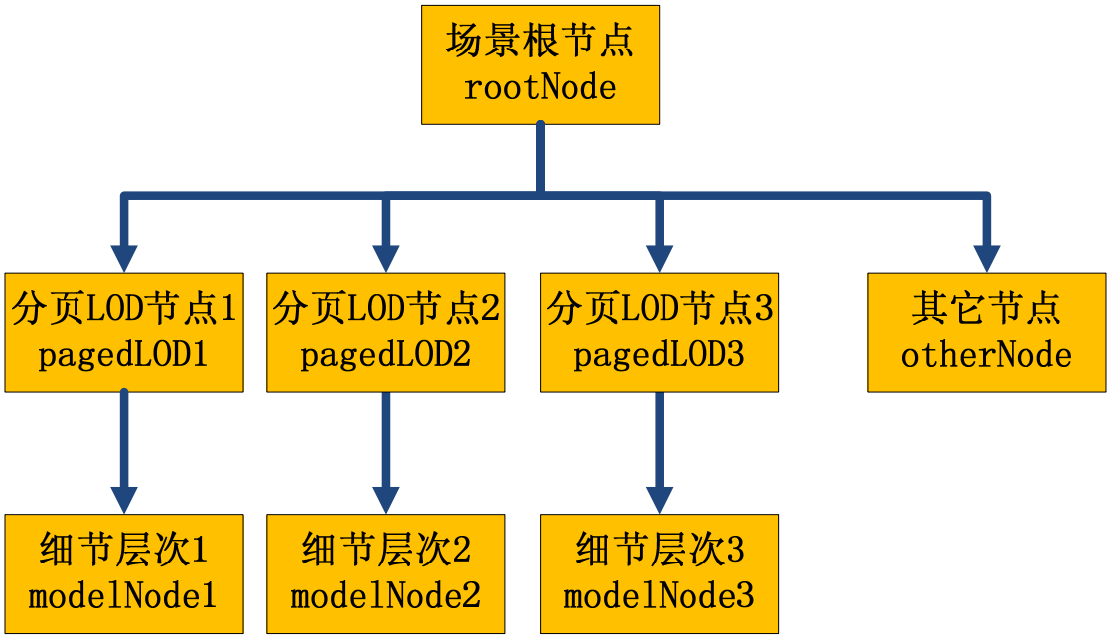
```
osg::PagedLOD* pagedLOD = new osg::PagedLOD;
pagedLOD->setFileName(0, “node_1.osg”);
pagedLOD->setRange(0, minRange1, maxRange1);
pagedLOD->setFileName(1, “node_2.osg”);
pagedLOD->setRange(1, minRange2, maxRange2);
……
```

上述代码的含义是：指定 PagedLOD 的各级子节点，0 级为 node_1.osg 的内容，其可视范围为（minRange1，maxRange1）；1 级-4 级以此类推。此时 OSG 将自动使用 DatabasePager 的处理能力，分批次加载场景数据，并根据用户当前视野和过期时间来决定是否剔除或再次加载场景数据。

DatabasePager::setExpiryDelay 用于决定视野外对象的过期时间。此外，我们还可以运用 PagedLOD::setPriorityOffset 来设置子节点的优先级，setCenter 来设置中心点等等。

注意下图中两种不同的场景结构。第一种结构是前面有关 PagedLOD 的代码所呈现的，也是适合于分页数据库运作的；而后一种结构虽然也可以达到同样的目的（通过 setRange 实现），但它也可能给程序带来不必要的运算负担。





最后，我们讨论一下 ProxyNode 和 PagedLOD 的区别：ProxyNode 的功能主要是在运行时加载一个或多个模型文件作为子节点；而 PagedLOD 虽然可以实现相同的功能，但它还有另外一项重要的工作，那就是根据用户的视点范围来实现场景树的“修剪”——剔除对场景长期没有助益的节点，加载用户可见的节点。这也是这几日以来我们一直强调的“分页”的精髓所在了吧。

解读成果：

DatabasePager::updateSceneGraph，ProxyNode 与 PagedLOD 节点。

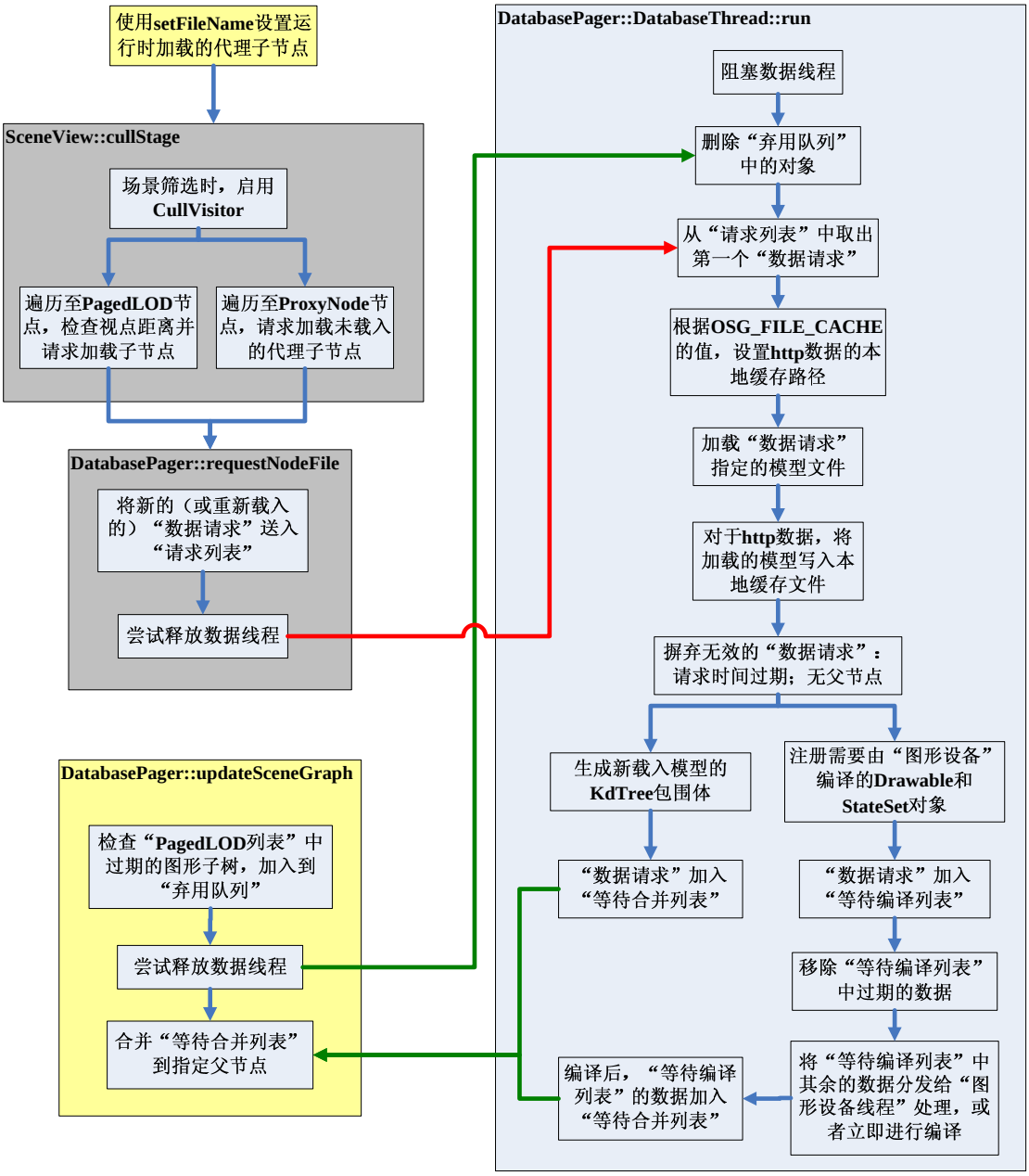
悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？如何使用 compileGLObjects 完成预编译工作？Operation 对象在线程中的应用时机是什么？

第十五日

当前位置：osgViewer/Viewer.cpp 第 892 行，osgViewer::Viewer::updateTraversal()

在完成有关 DatabasePager 的漫长讨论之前，我们还是来总结一下此行的收获吧，如下图所示：



1、首先，osg::PagedLOD 节点或者 osg::ProxyNode 节点使用 setFileName 函数，请求运行时加载模型文件为子节点。

2、在场景的筛选（Cull）过程中，OSG 将自动取出 PagedLOD 和 ProxyNode 中保存的文件名数据，并使用 DatabasePager::requestNodeFile 函数将其保存到“数据请求列表”中（RequestQueue::_requestList）。

3、DatabasePager 内置了两个数据处理线程（DatabaseThread），分别用于处理本地文件和 HTTP 数据，线程的主要工作是删除“已弃用队列”（RequestQueue::_childrenToDeleteList）

中的对象，并从“数据请求列表”中获取新的请求。

4、线程中如果取得新的数据请求，则尝试加载新的模型，判断是否需要预编译模型，并送入“等待编译列表”（_dataToCompileList）。预编译的含义是执行显示列表的创建，纹理绑定，GLSL 数据绑定等 OpenGL 动作，通常情况下预编译模型可以避免它在显示时出现帧延迟。

5、对于编译完成或者无需编译的数据请求，首先创建其 KDTree 包围体（用于 K-Dop Tree 碰撞检测计算），然后送入“等待合并列表”（_dataToMergeList），线程让出控制权。

6、场景的每次更新遍历均会执行 updateSceneGraph 函数，于其中将一段时间内没有进入用户视野的节点送入“已弃用队列”（注意这一工作只限于 PagedLOD 节点的子节点），并将“等待合并列表”中的新数据使用 addChild 送入当前的场景图形。

好了，不管您是否真的理解了我所说的一切，我们都要继续前进了。不必拘泥于我所述的东西，您自己钻研和实验所得的才是最重要的。

当前位置：osgViewer/Viewer.cpp 第 895 行，osgViewer::Viewer::updateTraversal()

有了前面的基础，现在我们可以轻松地阅读 ImagePager 的相关内容了。这个类的工作性质与 DatabasePager 没什么大的区别，它主要负责的是纹理图片文件的运行时加载工作。

OSG 2.6 的版本中，只有 osg::ImageSequence 可以使用图片数据库来执行工作，通过调用 ImageSequence::addImageFile 函数，指定运行时加载的图片文件名称。文件将被逐步加载并送入这个 ImageSequence 对象中。ImageSequence 类的主要功能是使用自身包含的 Image 图片对象序列，实现一种动画纹理的效果。

与 DatabasePager 相同，ImagePager 也内置了一个处理线程（ImagePager::ImageThread），其中随时读取“图片加载请求”（ImagePager::ImageRequest）的内容，并根据其中的文件名使用 osgDB::readImageFile 加载数据（osg::Image 对象）。加载之后的图片数据将被加入到申请它的 ImageSequence 对象中。

ImagePager 本身暂时不具备“分页”的功能，换句话说，在目前的版本中，它不会负责将长时间不用的图片删除。

那么，现在我们宣布——漫长的 updateTraversal 函数之旅可以在今日划上句号了。好好休息一下，更艰巨的任务还在后面。

解读成果：

osgDB::DatabasePager，osgDB::ImagePager，Viewer::updateTraversal。

悬疑列表：

如何调度和实现 OSG 的多线程机制？什么是“渲染目标实现方式”？如何使用 compileGLObjects 完成预编译工作？Operation 对象在线程中的应用时机是什么？

第十六日

当前位置：osgViewer/ViewerBase.cpp 第 596 行，osgViewer:: ViewerBase::renderingTraversals()

欢迎来到“最长的一帧”最后一个环节：也就是 frame()函数中置于最后的“渲染遍历”操作。但是这最后一个环节并不意味着我们马上就要完结这篇冗长的故事，恰恰相反，就好比电脑游戏中最后关底的 BOSS 那样，我们即将深入去了解的这个“渲染遍历”，正是 OSG 中最为精华也最为复杂的组成部分，也是本教程中最艰辛的一段旅程。

首先我们还是回忆一下，OSG 传统教程中，有关渲染机制的一些概念。

OSG 的场景渲染过程可以简单地分为三个阶段：用户（APP）阶段，更新用户数据，负责场景对象的运动和管理等等；筛选（CULL）阶段，负责对场景中的对象进行筛选裁减，略过那些不会被用户所见（因而不必渲染）的物体，并根据渲染状态的相似性对即将进入渲染管线的对象排序（从而避免 OpenGL 状态量的频繁切换）；绘制（DRAW）阶段，执行各种 OpenGL 操作，将数据送入 OpenGL 渲染管线及显示系统处理。

如果有多个图形设备（渲染窗口）时，需要分别为每个窗口的每个摄像机执行相应的筛选和绘制工作，因为各个摄像机的投影矩阵和观察矩阵均可能不同；不过用户（APP）阶段不需要被执行多次，因为用户数据应当是被各个图形设备所共享的。

对于单线程运行的系统来说，用户/筛选/绘制这三个阶段在每一帧当中都应当是顺序执行的；而对于多线程运行，以至多 CPU 的系统来说，则可以将前后两帧的工作稍微有所交叠。用户更新（APP）和场景筛选（CULL），以及场景筛选和绘制（DRAW）的工作互相不能重叠；但是我们可以允许在上一帧的绘制没有结束之前，就开始下一帧的用户数据更新工作；我们还可以允许由不同的 CPU 来执行不同图形设备的筛选和绘制工作，从而提高整体渲染的效率，实现实时渲染的目的。

那么，我们再来重新审视一下 frame()函数的基本内容：

```
advance(simulationTime);   // 记录场景的帧数，帧速率信息
eventTraversal();          // 处理场景的交互事件及其回调
updateTraversal();         // 处理场景的更新回调，以及分页数据的更新
renderingTraversals();     // 场景的渲染遍历工作
```

细心的您一定已经发现了，在我们前面十六日所介绍的内容当中，并没有涉及到有关场景筛选或绘制的源代码，而是着重于有关用户更新（APP）操作的各类内容，而这些代码中也几乎没有涉及到线程与多核处理的内容（分页数据库的处理线程是个例外，它并非是与场景渲染相关的内容）。那么，从今天开始，一切都会与以前有所区别：我们将针对 OSG 目前提供的四种线程模型的特点，对 OSG 的渲染机制做尽量细致的讲解，希望能对您的学习，您得创作，甚至今后实现您自己的渲染引擎有所助益。

请您先回到第五日的内容，也许您还记得我们所叙述过的 ViewerBase::setUpThreading 函数，以及留在悬疑列表中至今没有介绍的 ViewerBase::startThreading 的函数，那么我们就从这个“线程启动”函数开始。

当前位置：osgViewer/ViewerBase.cpp 第 243 行，
osgViewer:: ViewerBase::startThreading ()

startThreading 函数一开始，先要完成这样几个工作：

1、执行 ViewerBase::releaseContext，首先释放渲染上下文；

2、如果用户没有设置线程模型，则使用 ViewerBase::suggestBestThreadingModel 自动进行判断；

3、使用 Viewer::getContexts 函数获取当前所有的图形设备（GraphicsContext）；

4、使用 Viewer::getCameras 函数获取当前所有的摄像机（主摄像机和所有从摄像机）。

这其中，suggestBestThreadingModel 函数可以帮助用户决定采用什么样的线程模型，基本的判断原则如下：

1、如果定义了环境变量 OSG_THREADING，且其中内容为四个字符串中的一个，则采用对应的模型，“SingleThreaded”，“CullDrawThreadPerContext”，“DrawThreadPerContext”，“CullThreadPerCameraDrawThreadPerContext”。

2、如果当前不存在图形设备或者摄像机，则采用 SingleThreaded 模型。

3、如果当前只存在一个图形窗口（这是最常见的情况），则采用 SingleThreaded 模型（单核）或 DrawThreadPerContext 模型（多核）。

4、如果系统 CPU 数超过当前图形设备和摄像机数总合，为了充分发挥每个 CPU 的能力，将采用 CullThreadPerCameraDrawThreadPerContext 模型。

5、以上情形均不符合时，采用 DrawThreadPerContext 模型。

可见，对于目前流行的多核系统而言，CullThreadPerCameraDrawThreadPerContext 和 DrawThreadPerContext 是最好的选择，而 SingleThreaded 则适用于配置较低的系统，剩下的 CullDrawThreadPerContext 模型目前并不建议用户使用。

那么，这几种线程模型在系统中的执行有什么区别呢？下面我们首先讨论单线程 SingleThreaded 的情形。

对于单线程模型的执行流程，startThreading 函数没有更多的工作要做，因此我们直接跳到 renderingTraversals 函数的执行过程，并注意略去其中关于线程控制的部分（因为单线程模型中没有为图形设备或摄像机创建更多的执行线程）。

当前位置：osgViewer/ViewerBase.cpp 第 596 行，
osgViewer:: ViewerBase::renderingTraversals()

对于单线程模型来说，以下一些变量是无效的（没有经过定义）：

ViewerBase::_startRenderingBarrier：可以理解为渲染启动的一个栅栏标志，用于同步开始所有的图形设备的线程操作。

ViewerBase::_endRenderingDispatchBarrier：渲染结束的一个栅栏标志，用于同步结束所有的图形设备的线程操作。

ViewerBase::_endDynamicDrawBlock：用于同步结束所有的动态对象绘制操作，这里所谓的动态对象，指得是使用 Object::setDataVariance 设置为 DYNAMIC 的场景对象。

有关它们的详细介绍将在后文展开，现在我们先不必关心这些变量的操作。

单线程模式下，renderingTraversals 函数的基本执行步骤如下：

1、首先使用 ViewerBase::checkWindowStatus 检查是否存在有效的图形设备，不存在的话，需要使用 ViewerBase::stopThreading 停止线程运行。

2、记录渲染遍历开始的时间。

3、遍历视景器对应的所有 Scene 场景（Viewer 单视景器只存在一个场景），记录分页数据库的更新启动帧（使用 DatabasePager::signalBeginFrame，这将决定 DatabasePager 中的数据请求是否过期），并计算场景节点的边界球。

4、获取当前所有的图形设备（GraphicsContext）和摄像机。

5、遍历所有摄像机的渲染器（Renderer），执行 Renderer::cull 场景筛选的操作！

6、遍历所有的图形设备，设置渲染上下文（使用 ViewerBase::makeCurrent）并执行 GraphicsContext::runOperations，实现场景绘制的操作！

7、再次遍历所有的图形设备，执行双缓存交换操作（GraphicsContext::swapBuffers），熟悉图形编程的朋友一看便知，这是避免动态绘图时产生闪烁的重要步骤。

8、遍历视景器中的场景，告知分页数据库更新已经结束（DatabasePager::signalEndFrame，目前这个函数没有作用）。

9、释放当前的渲染上下文（ViewerBase::releaseContext）。

10、记录渲染遍历结束的时间，并保存到记录器当中（ViewerBase::getStats）。

无疑，第 5 步和第 6 步的操作是整个函数，以致整个 OSG 系统的重点，下面我们将花费大量的篇幅来深入探讨它们的执行过程。

解读成果：

ViewerBase::suggestBestThreadingModel，ViewerBase::renderingTraversals 基本流程。

悬疑列表：

什么是“渲染目标实现方式”？如何使用 compileGLObjects 完成预编译工作？Operation 对象在线程中的应用时机是什么？

第十七日

当前位置：osgViewer/Renderer.cpp 第 596 行，osgViewer::Renderer::cull ()

在了解摄像机渲染器如何进行场景筛选之前，我们先来了解一下什么是渲染器，以及 osgViewer::Renderer 类与摄像机和场景的关系。

osgViewer::Renderer 为摄像机渲染场景的工作提供了一个公有接口。当我们向视景器（Viewer）添加一个新的摄像机（Camera）时，一个与摄像机相关联的渲染器（Renderer）也会被自动创建。而当我们准备渲染场景时，与特定图形设备（GraphicsContext）相关联的摄像机也会自动调用其渲染器的相应函数，执行场景筛选与绘制等工作。

获取摄像机附带渲染器的代码为：

```
osgViewer::Renderer* renderer =
   dynamic_cast<osgViewer::Renderer*>(camera->getRenderer());
```

注意这里我们用了 dynamic_cast 动态类型转换，因为 Camera::getRenderer 函数返回的是 osg::GraphicsOperation 的指针。之所以没有直接返回 Renderer 指针，其原因是 Renderer 类属于 osgViewer 命名空间，因而依赖于 Camera 所属的 osg 命名空间，不能直接用于 Camera 的成员函数。

这里还将涉及到一个新的概念：场景视图，也就是 osgUtil::SceneView 类，在 OSG 早期版本中这个类时常需要由用户调用来完成各类功能；但是自从 2.x 版本发布之后，这个类开始退居幕后，仅仅由 OSG 系统内部加以调用，而负责调用场景视图（SceneView）各种功能的，就是新增加的这个渲染器（Renderer）了。

每个渲染器当中都会自动创建两个 SceneView 对象（Renderer::_sceneView[2]），从而实现了渲染后台双缓存的支持，不过 SingleThreaded 和 CullDrawThreadPerContext 环境下只使用到第一个场景视图（SceneView）。

那么我们首先看一下 Renderer::cull 函数主要都完成了什么工作：

1、首先从_availableQueue 队列中获取一个可用的场景视图（SceneView）。这个队列中通常会保存有两个 SceneView 对象，以实现我们刚刚提到的渲染后台双缓存支持。

2、执行 Renderer::updateSceneView 函数，更新这个场景视图的全局渲染状态（根据场景主摄像机的 StateSet 渲染状态集，更新成员变量 SceneView::_globalStateSet），状态量（osg::State），显示设置（osg::DisplaySettings）。

这里的全局渲染状态_globalStateSet，将在后文中我们介绍渲染状态树时发挥它的作用，现在还是先把它放到“悬疑列表”当中吧。

此外这里还涉及到一个 OSG 内部经常使用的类，osg::State。简单来说，这个类是 OpenGL 状态机在 OSG 中的具体实现。它封装了几乎所有的 OpenGL 状态量，属性参数，以及顶点数组的设置值。我们编程时常见的对 StateSet，Geometry 等类的操作，实质上最终都交由 State 类来保存和执行。它提供了对 OpenGL 状态堆栈的处理机制（因此我们不必像 OpenGL 开发者那样反复考虑堆栈处理的问题），对即将进入渲染管线的数据进行优化（执行渲染状数据的排序，减少 OpenGL 状态的变化频率），同时还允许用户直接查询各种 OpenGL 状态

的当前值（直接执行 State::captureCurrentState，而不必再使用 glGet*系列函数）。

3、更新场景视图（SceneView）的融合距离（Fusion Distance）和筛选设置（CullSettings）。所谓融合距离，指得是双眼所在平面到视线汇聚点的距离，可以通过 View::setFusionDistance 函数传递给 SceneView，通常应用于立体显示的场合。

4、执行 SceneView::cull 函数，这才是真正的场景筛选（裁减）工作的所在！

5、记录场景筛选所耗费的时间，并保存到统计器（osg::Stats）中。

6、最后，将这个渲染视图添加到绘制队列_drawQueue 中。这个队列中保存的对象将在场景绘制时用到。

可以看出，如果要深入学习场景筛选的具体流程，我们下一步探索的重点应该是 SceneView::cull 的执行过程，这里还将涉及有关 StateGraph，RenderLeaf，RenderStage 和 RenderBin 的相关内容，以及场景渲染后台的状态树和渲染树的工作原理。不过在这之前，我们有必要先了解一下 renderingTraversals 函数中所述的另一个重要步骤，即执行图形设备的 GraphicsContext::runOperations 函数，以完成场景的绘制工作。这将有助于后面我们对 SceneView 的深入学习。

当前位置：osg/GraphicsContext.cpp 第 675 行，osg::GraphicsContext::runOperations()

这个函数的执行过程如下：

1、获取场景中所有注册的摄像机（包括主摄像机和从摄像机组），对它们执行排序，排序的原则根据摄像机的渲染顺序而定，可以通过 Camera::setRenderOrder 进行设置。设置为 PRE_RENDER 级别的摄像机排序在最前，而 POST_RENDER 级别的摄像机排序在最后；同一级别的摄像机根据 setRenderOrder 函数中传入的整数设置先后顺序，排序数较小的摄像机在前。

2、依次遍历排序过的各个摄像机，执行其渲染器 Renderer 的 operator()操作，它有一个传入参数，即当前的 GraphicsContext 图形设备。这个重载的操作符实质上执行了场景在该图形设备中的绘制工作，因此前面的排序工作将决定哪个摄像机的内容先被绘制出来。

3、遍历 GraphicsContext::_operations 队列中的各个 Operation 对象，执行其 operator()操作。这里的 osg::Operation 类已经在之前的文字中被提及多次，我们可以重写自己的 Operation 派生对象，并通过 GraphicsContext::add 将其添加到图形设备的执行队列中，从而实现自己定义的 OpenGL 绘图功能（由于在执行 runOperations 函数之前已经执行了图形设备的 makeCurrent 函数，因此这里不必考虑渲染上下文的设置问题）。

Renderer 类成员函数 operator()的工作仅仅是判断是否使用图形线程来执行场景的筛选（根据 Renderer::_graphicsThreadDoesCull 变量的值）。对于单线程模型（SingleThreaded）来说，它将转向到 Renderer::draw 函数，因为场景筛选的工作已经由前面的代码完成了；对于线程模型（CullDrawThreadPerContext）来说，它将转向 Renderer::cull_draw 函数；而对于另外两种线程模型而言，DrawThreadPerContext 同样使用 Renderer::cull 和 Renderer::draw 来执行场景筛选与绘制的工作，而 CullThreadPerCameraDrawThreadPerContext 则为每个摄像机创建线程来完成筛选工作，场景的绘制仍然由下文将要叙述的 Renderer::draw 来完成。

解读成果：

Renderer::cull，GraphicsContext::runOperations。

悬疑列表：

什么是“渲染目标实现方式”？如何使用 compileGLObjects 完成预编译工作？Operation 对象在线程中的应用时机是什么？全局渲染状态_globalStateSet 有什么作用？

第十八日

当前位置：osgViewer/Renderer.cpp 第 334 行，osgViewer::Renderer::draw ()

这个函数的基本执行流程如下：

1、从绘制队列_drawQueue 中取出一个场景图形（SceneView）对象。

2、执行 Renderer::compile，这个函数目前还没有作用。

3、执行 Renderer::initialize，初始化 Renderer 绘制所需的基本变量。

4、下一步的工作是执行 SceneView::getDynamicObjectCount 函数判断场景视图中动态对象（设置为 DYNAMIC）的个数，并执行其回调类（此回调类派生自线程阻塞器 BlockCount，此处为 State::getDynamicObjectRenderingCompletedCallback）的 completed 函数。

如果您还记得我们在十一日中介绍的内容的话，也许能理解这里 completed 函数的作用，也就是将阻塞器的引用计数减一，减到零时自动释放所阻塞的线程。不过，对于单线程模型来说，这一步骤基本上没有用处。它实质上是为了在多线程工作时保证动态（DYNAMIC）对象的更改不会影响到渲染管线而实现的：正如 OSG 基础教程中所强调的那样，只有设置为 setDataVariance(DYNAMIC)的对象才可以在仿真循环中被随时更改。我们将在讲解多线程的工作流程时再对此作详细介绍。

5、执行 OpenGLQuerySupport::checkQuery 函数，判断是否可以使用 OpenGL 查询对象（query objects）。

6、执行 OpenGLQuerySupport::beginQuery 函数，创建或者获取一个查询对象，其工作主要是获取并统计 GPU 计算的时间。

7、执行 SceneView::draw 函数，果然，场景的绘制工作最后也是在 SceneView 函数中完成的！看来代码解读的重心无疑要偏移到这个历史悠久的“场景视图类”当中了。

8、将已经结束绘制的场景视图对象再次追加到_availableQueue 队列中，这样可以保证该队列始终保存有两个 SceneView 对象，以正确实现场景的筛选和渲染工作。

9、执行 Renderer::flushAndCompile 函数。它的作用是执行场景中所有过弃对象的删除工作，单线程模型下将直接执行 osg::flushDeletedGLObjects 函数；而对于多线程模型，由于每个图形设备（GraphicsContext）都分配了一个操作线程（见第五日的内容），因此将向线程追加一个新的 osg::FlushDeletedGLObjectsOperation 操作，它所调用的函数实质上与单线程模式下相同。FlushDeletedGLObjectsOperation 继承自 Operation 类，有关它的调用时机的问题……也许我们还需要让它在“悬疑列表”中再等待一段时间。

需要删除的对象包括过弃的显示列表，纹理，着色器等，具体的方法可以参考 releaseGLObjects 函数，它在多个类当中都有实现。

flushAndCompile 还负责分页数据库的预编译工作（即 DatabasePager::compileGLObjects 函数的内容）。在“悬疑列表”中我们搁置了对于 compileGLObjects 函数的介绍，现在可以兑现了。

这个函数的主要工作其实并不复杂：首先，从 DatabasePager 的“待编译列表”中取出一个数据，并从它的 DataToCompileMap 映射表中取出待编译的几何体（Drawable）队列和渲染状态（StateSet）队列。

对于 Drawable 对象，执行 Drawable::compileGLObjects，创建几何体的显示列表（使用我们熟悉的 glNewList）；对于 StateSet 对象，执行 StateSet::compileGLObjects，进而执行各种渲染属性的预编译命令（例如 Texture 对象要使用 glTexImage2D 等函数执行纹理数据的加载，而 Program 对象要执行 GLSL 代码的载入和编译）。

最后，从“待编译列表”中剔除已经编译的数据。注意每一帧的过程中没有必要编译所

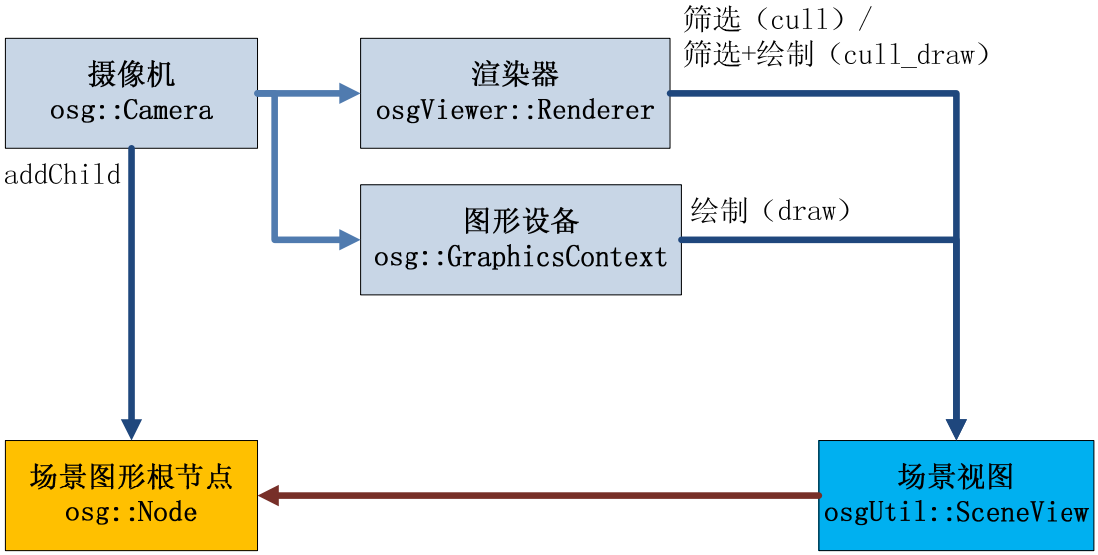
有的对象，使用 DatabasePager::setMaximumNumOfObjectsToCompilePerFrame 可以设置每一帧从“待编译列表”中取出多少数据执行编译工作，缺省值为 4。

10、执行 OpenGLQuerySupport::endQuery，结束查询对象的工作。

11、计算场景绘制所需的时间，并传递给记录器（ViewerBase::getStats）。

当前位置：osgViewer/Renderer.cpp 第 452 行，osgViewer::Renderer::draw ()

好了，现在我们已经大体上明确了单线程模型（SingleThreaded）下 OSG 渲染遍历的工作流程。事实上无论是场景的筛选还是绘制工作，最后都要归结到场景视图（SceneView）的相应实现函数中去完成，渲染器类 Renderer 只是一个更为方便和直观的公用接口而已。下图中演示了单线程运行时，OSG 系统的场景图形，摄像机，图形设备，渲染器和场景视图的关系：



OSG 视景器的摄像机（包括主摄像机_camera 和从摄像机组_slaves）均包括了与其对应的渲染器（Renderer）和图形设备（GraphicsContext）；同时，当我们使用 setSceneData 将场景图形的根节点关联到视景器时，这个根节点实质上被添加为此 Viewer 对象中每个主/从摄像机的子节点（使用 View::assignSceneDataToCameras 函数），因而我们可以通过改变摄像机的观察矩阵来改变我们观察整个场景的视角。

场景的筛选（CULL）和绘制（DRAW）工作实质上都是由内部类 osgUtil::SceneView 来完成的，但是 OSG 也为场景渲染的工作提供了良好的公用接口，就是“渲染器”。渲染器 Renderer 负责将场景绘制所需的各种数据（OpenGL 状态值，显示设置，筛选设置等）传递给 SceneView 对象，并调用 SceneView::cull 和 SceneView::draw 函数，以完成场景的筛选/绘制工作。

摄像机所对应的图形设备（GraphicsContext）同样也可能负责调用 SceneView::draw 函数，这与我们选择的线程模型有关。事实上，由于 OSG 的多线程模型将为每一个图形设备创建一个专门的工作线程（使用 GraphicsContext::createGraphicsThread 函数），并在其中处理与场景绘制相关的诸多工作，因此 GraphicsContext 类在某种意义上也可以视作 SceneView 的一个公有实现接口（不过它更重要的意义在于，它是 OSG 与特定操作系统平台 API 的接口，参见第四-六日的内容）。

场景视图的工作过程中将遍历场景图形的根节点，此时只要获取对应摄像机的子节点就可以了。

下面我们就正式进入 SceneView 的内部，看看 OSG 那高效的渲染后台，到底是个什么

样子。

解读成果：

Renderer::draw，摄像机/渲染器/场景视图的关系。

悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？全局渲染状态_globalStateSet 有什么作用？

第十九日

当前位置：osgUtil/SceneView.cpp 第 670 行，osgUtil:: SceneView::cull ()

我们在进行用户程序的开发时，最常用到的场景管理方式是“场景节点树”的结构，场景树顶端的叶节点（osg::Geode）包含了各种需要渲染的几何体的顶点和渲染状态信息；而组节点（osg::Group）及其派生出的各种特殊功能节点则作为场景树的各个枝节节点，它们也可以拥有不同的渲染状态；有且只有一个节点可以直接作为整个场景的根节点，使用 setSceneData 将其设置给场景的视景器系统，即等同于将整个场景树传递给 OSG 的渲染和显示系统。

而保存节点和几何体的各种渲染属性（osg::StateAttribute，例如纹理，雾效，材质，Alpha 校验等）和模式开关，则使用节电所附带的渲染状态集（osg::StateSet）。一个状态集中可以包含多种不同的渲染属性和开关，处于场景树顶端的节点将继承并综合各级父节点的渲染状态，实现几何形状的正确渲染。

OSG 渲染后台的主体是场景视图（SceneView），它同样实现了“树状结构”的管理方式，并据此实现了多个专用于渲染工作的内部类。那么在深入介绍场景视图之前，我们先来认识一下 OSG 渲染后台的几个“幕后英雄”：

osgUtil::CullVisitor：“筛选访问器”。虽然同样是继承自 osg::NodeVisitor，不过这个访问器在整个 OSG 系统中可是起了举足轻重的作用。当我们使用它遍历场景图形的各个节点时，CullVisitor 将会对每一个遇到的节点执行场景筛选的工作，判断它是否会超出视截锥体范围，过于渺小，或者被遮挡节点（OccluderNode）挡住，从而将无助益于场景浏览的物体筛选并剔除，降低场景绘制的资源消耗。我们甚至可以使用 SceneView::setCullVisitor 来构建和指定使用自己设计的筛选访问器，不过在系统渲染后台之外的环境使用 CullVisitor 通常并无用处。

osg::RenderInfo：“渲染信息”管理器。这个类负责保存和管理与场景绘制息息相关的几个重要数据：当前场景的视景器，当前场景对应的所有摄像机，以及当前所有 OpenGL 渲染状态和顶点数据（使用第十七日所述的 osg::State 类保存）。这些数据将在场景筛选和渲染时为 OSG 系统后台的工作提供重要依据。

osgUtil:: StateGraph：“状态节点”。我们可以对比场景树的组节点（Group），将 StateGraph 理解为 OSG 渲染后台的组节点。它的组织结构与场景图形的节点结构类似，但是状态树的构建主要以节点的渲染状态集（StateSet）为依据：设置了 StateSet 的场景节点，其渲染状态会被记录到“状态节点”中，并保持它在原场景树中的相对位置；状态节点采用映射表 std::map 来组织它的子节点，同一层次的子节点如果渲染状态相同，则合并到同一个“状态节点”中。

osgUtil::RenderLeaf：“渲染叶”。我们可以把 RenderLeaf 理解为 OSG 渲染后台状态树的叶节点。但是，状态树的叶节点绝非等同于场景树的 Geode 节点；事实上，“渲染叶”的工

作主要是记录场景树中存在的各种 Drawable 对象（以及与之相关的投影矩阵，模型视点矩阵等信息）。

每个“状态节点”中都包含了一个渲染叶的列表（StateGraph::_leaves），不过只有最末端的“状态节点”会负责记录场景中的“渲染叶”。

osgUtil::RenderStage：“渲染台”。OSG 的渲染后台除了使用“状态树”来组织和优化节点的渲染状态之外，还有另外一种用于场景实际渲染的组织结构，我们称之为“渲染树”，“渲染树”的根节点就是“渲染台”。

通常来说，由于 OSG 后台只有一个渲染树结构，因此应当也只有一个“渲染台”存在；不过 OSG 还提供了“设置摄像机渲染顺序”的功能，即 Camera::setRenderOrder。设置为 PRE_RENDER 的摄像机子树将在主摄像机之前执行渲染，通常我们可以由此实现诸如“纹理烘焙”（Render To Texture）的高级功能（参见 osgprerender 例子）；设置为 POST_RENDER 的摄像机子树将在主摄像机之后执行渲染，一些必须在最后进行渲染的场景对象，例如 HUD 显示牌，可以置为这类摄像机的子节点。

注意，把视景器（Viewer）的主/从摄像机设置为 PRE_RENDER 或 POST_RENDER 往往是没有意义的。所谓从摄像机组（View::_slaves），其功能主要是实现同一场景的分窗口以及分屏幕显示（参见 osgcamera 例子）。如果您希望实现诸如 HUD 显示，简单鹰眼图等功能时，应当向场景树中添加新的摄像机节点，并设置与主摄像机不同的观察矩阵和投影矩阵。

osgUtil::RenderBin：“渲染元”。它是 OSG 渲染树的分支节点，不过对于没有特殊要求的场景渲染来说，更多的渲染树分支也许并不需要：场景中需要渲染的元素及其渲染属性被保存到各个“状态节点”和“渲染叶”当中；渲染树只要按照遍历的顺序，把这些数据记录到作为根节点的“渲染台”当中（即分别保存到 std::vector 成员量 RenderBin::_stateGraphList 和 RenderBin::_renderLeafList 当中，注意 RenderStage 派生自 RenderBin），就可以执行场景的绘制工作了。

但是，很多时候我们需要某些几何体在其它对象之前被绘制，比如天空总是要被任何飞过的物体所遮挡；很多时候我们也需要在大部分对象绘制完成之后才绘制某个几何体的数据（例如 HUD 文字总是显示在所有对象之上）。这种情况下，就有必要对“渲染台”中的数据进行排序，甚至为其创建新的分支“渲染元”，以实现这种复杂的渲染顺序处理。

在用户程序中，渲染顺序通过 StateSet::setRenderBinDetails 实现设置。这个函数有两个传入参数，整型数表示渲染的顺序，以 0 为标准，小于 0 的渲染状态集（亦即包含了这个 StateSet 的 StateGraph 状态节点）将排列在前，大于 0 的则排列在后；字符串参数“RenderBin”或者“DepthSortedBin”作为名称时有特殊含义，其中“RenderBin”表示在渲染树中新建分支进行渲染，“DepthSortedBin”表示新建分支，并且所有要渲染的数据将按照深度值降序进行排序。

注意，当字符串参数不为“RenderBin”或“DepthSortedBin”时，渲染顺序的设定也是无效的；当字符串参数和整型参数均有效时，OSG 系统将尝试寻找同类型的渲染元节点并将 StateSet 记录到此“渲染元”中，或者创建新的“渲染元”节点。相关的示例代码如下：

```
// 缺省渲染方式，渲染顺序 0，此时状态节点直接置入“渲染台”
stateSet->setRenderBinDetails( 0, “” );
// 渲染顺序-1（先渲染），此时渲染树中将新建一个“渲染元”节点
stateSet->setRenderBinDetails( -1, “RenderBin” );
// 渲染顺序 10，此时将新建一个“渲染元”，并按深度值降序排序各元素
stateSet->setRenderBinDetails( 10, “DepthSortedBin” );
```

为了简化操作，用户程序还可以使用 StateSet::setRenderingHint 来设置渲染的顺序，这个函数的传入参数可以为枚举量 OPAQUE_BIN 或 TRANSPARENT_BIN。前者可以指定该

渲染状态用于不透明物体的渲染，后者则指定该渲染状态用于透明物体的渲染，此时 OSG 自动将其渲染顺序置后，并设置它所管理的“状态节点”和“渲染叶”数据按照深度值降序进行排序。

关于 setRenderBinDetails 与 setRenderingHint 的关系，也可以这样解释：

stateSet->setRenderingHint (OPAQUE_BIN);

stateSet->setRenderingHint (TRANSPARENT_BIN);

分别等价于：

stateSet->setRenderBinDetails(0, “RenderBin”);

stateSet->setRenderBinDetails(10, “DepthSortedBin”);

状态树，渲染树，还有那些名字看起来都差不多的类……一口气出现这么多新事物，您是不是有点晕头转向了？没关系，也许下一日的一个场景实例就可以稍微帮您理清思路：

解读成果：

Camera::setRenderOrder，StateSet::setRenderBinDetail，StateSet::setRenderingHint。

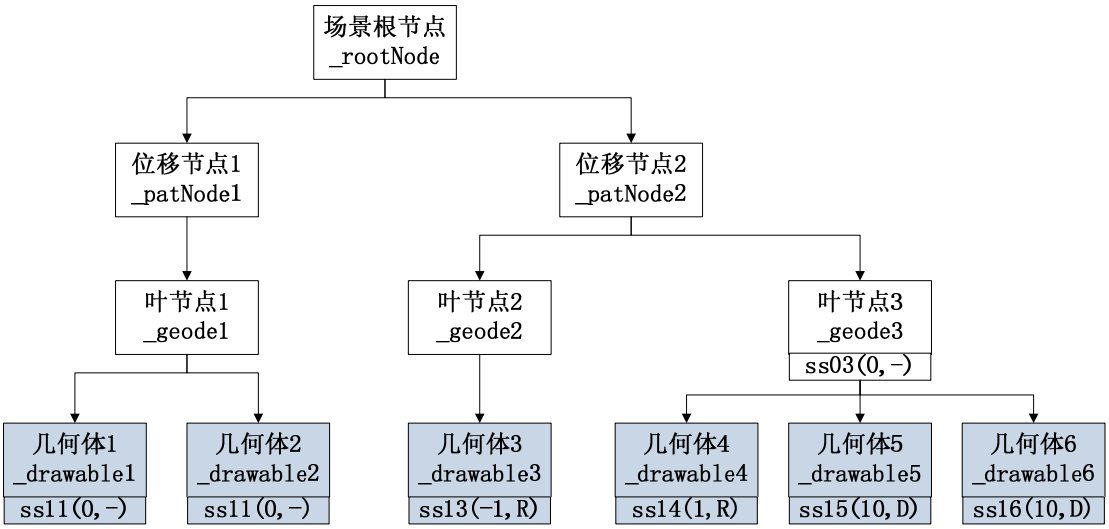
悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？全局渲染状态_globalStateSet 有什么作用？

第二十日

当前位置：osgUtil/SceneView.cpp 第 670 行，osgUtil:: SceneView::cull ()

首先我们来看一个场景构建的实例，并希望借此机会了解一下“状态节点”StateGraph 和“渲染叶”RenderLeaf 所构成的状态树，“渲染台”RenderStage 及“渲染元”RenderBin 所构成的渲染树，这两棵树之间错综复杂的关系，以及它们与场景节点树之间更为错综复杂的关系（真是让人冒汗）。



上面的场景结构图中，叶节点_geode3，以及所有六个几何对象均设置了关联的渲染状态集（StateSet），且几何体 1 和几何体 2 共享了同一个 StateSet。图中用“ss”加上数字代号来标识这些 StateSet 对象，后面括号中的两个参数分别表示 setRenderBinDetails 的两个设置项（“-”表示空字串，“R”表示“RenderBin”，“D”表示“DepthSortedBin”）。用代码来表示的话，即是：

ss03->setRenderBinDetails(0, “”);

ss11->setRenderBinDetails(0, “”);

ss13->setRenderBinDetails(-1, “RenderBin”);

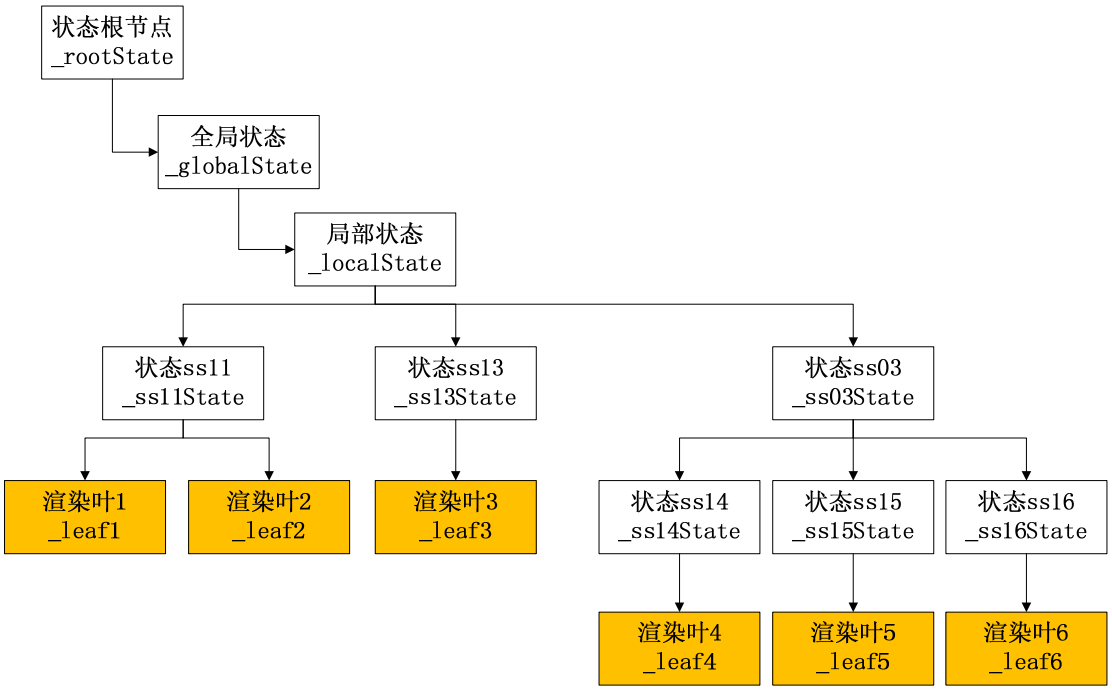
ss14->setRenderBinDetails(1, “RenderBin”);

ss15->setRenderBinDetails(10, “DepthSortedBin”);

ss16->setRenderBinDetails(10, “DepthSortedBin”);

注意，“setRenderBinDetails(0, “”)”是缺省的设置，这里写出来是为了便于区别；此外，OSG 中所有的 Drawable 几何体对象都会自动关联一个 StateSet 对象，无论用户是否在自己的程序中作了设置。

进入渲染后台之后，OSG 将为这个场景生成“状态树”，它是由“状态节点”StateGraph 和“渲染叶”RenderLeaf 所组成的：



图中的“状态根节点”和“局部状态节点”都是由状态树自动生成的，其中后者的主要工作是保存和维护一些渲染后台自动创建的渲染属性；而“全局状态节点”则保存了一个名为_globalStateSet 的渲染状态集对象。没错，这就是我们在第十七日提及的那个“全局渲染状态”，它的取值是场景主摄像机的 StateSet，换句话说，任何对状态树的遍历都将首先及至场景主摄像机的渲染状态，然后才是各个节点的渲染状态，这就是_globalStateSet 的功用所在了。

而整个状态树的构建过程则可以参考上面的场景树结构图，其规则为：

1、状态树是根据渲染状态（StateSet）来生成的，那些没有设置 StateSet 的场景节点将不会影响状态树的构架；

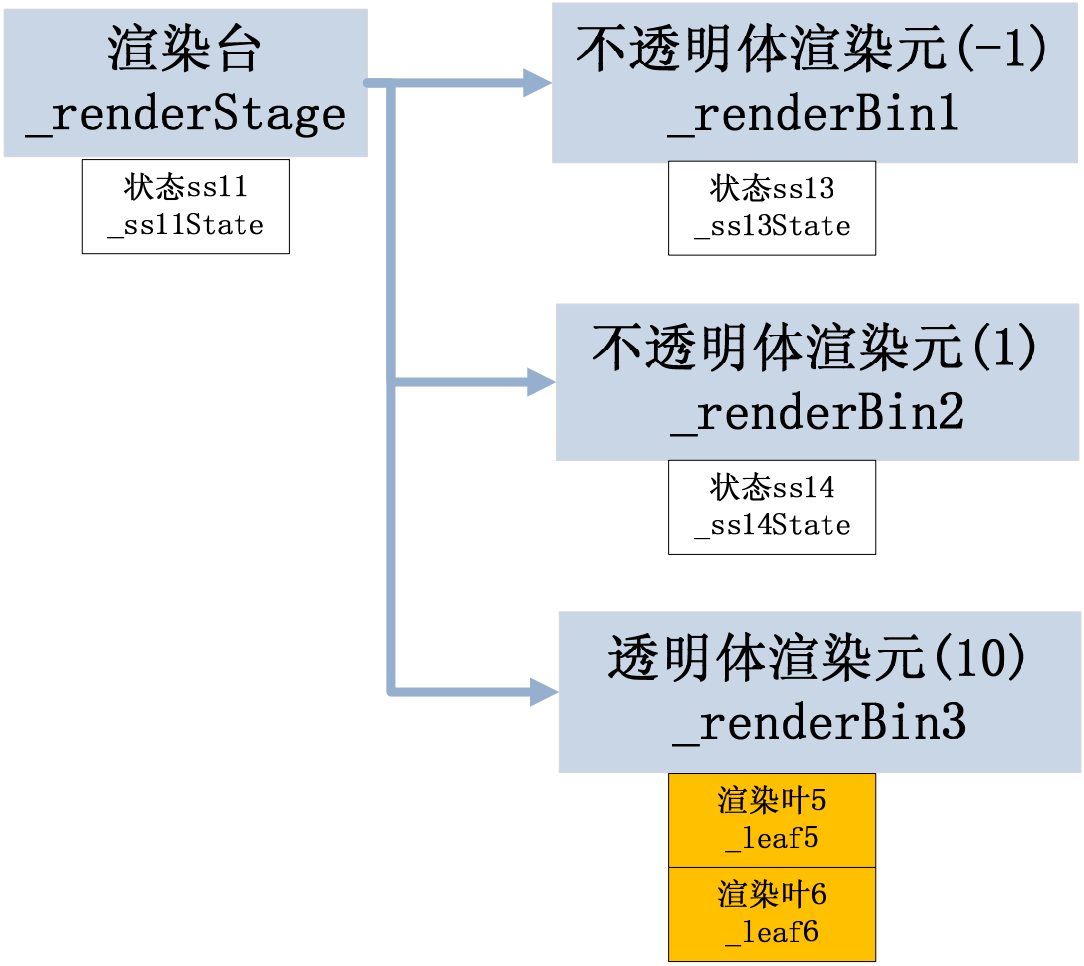
2、场景中的 Drawable 对象在状态树中被置入分别的渲染叶（RenderLeaf）中，而一个或多个渲染叶必然被一个状态树末端的节点（StateGraph）所拥有；

3、共享同一个渲染状态的 Drawable 对象（图中的_drawable1 和_drawable2）在状态树中将置入同一个末端节点。

生成状态树的同时，OSG 渲染后台还将生成对应的“渲染树”，其组成为一个 RenderStage 对象和多个 RenderBind 对象。如果我们不使用 setRenderBinDetails 设置 StateSet 的渲染细节的话，那么所有状态树中的末端节点（其中必然包含了一个或多个“渲染叶”）都会按遍历

顺序保存到渲染树根节点（渲染台）中，渲染树的构建也就到此结束。

但是，如果我们对于场景中部件的渲染顺序有特殊要求的话，那么渲染树也会因而变得复杂，上面的场景示例最后可能得到如下的一株“渲染树”：



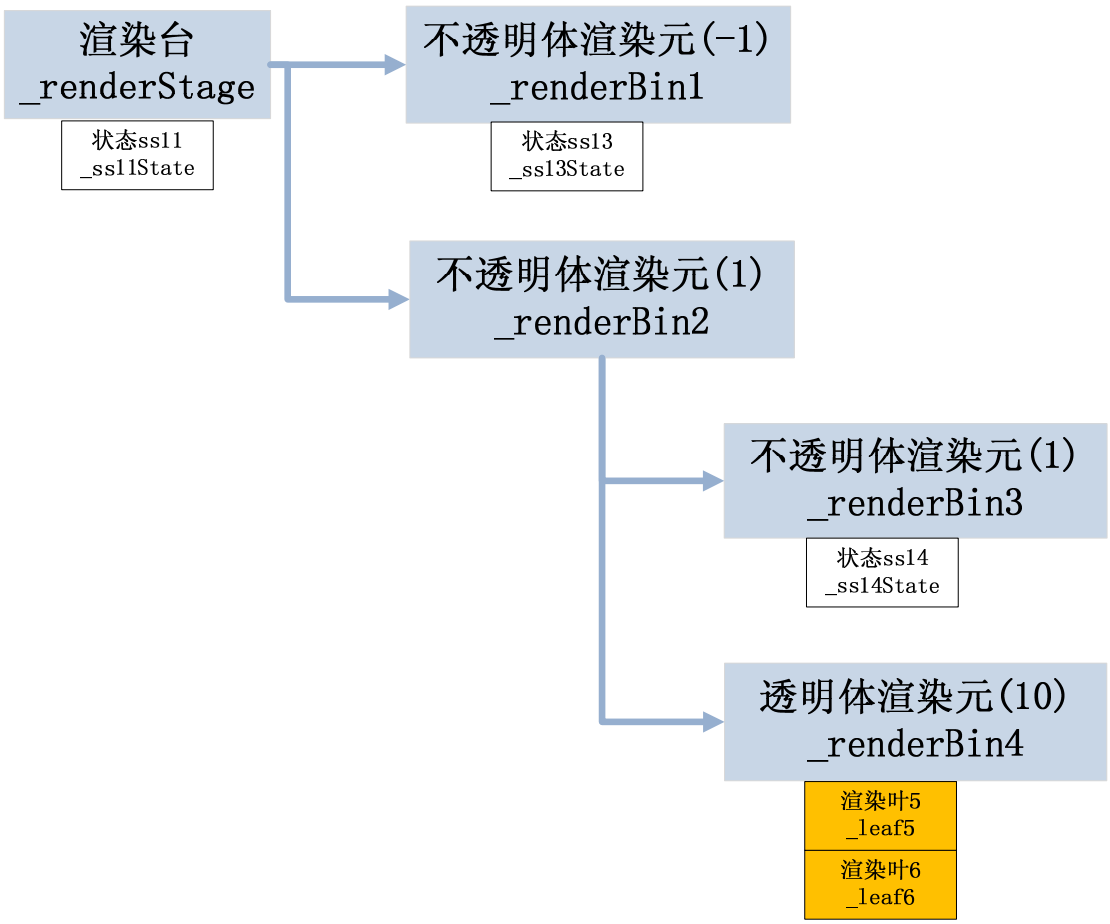
根据渲染顺序的不同，渲染树生出了三个分支。相应的状态节点置入各个渲染元（RenderBin）分枝中，其中渲染细节设置为“RenderBin”的状态节点（StateGraph）所处的渲染元也可称为“不透明体渲染元”；而设置为“DepthSortedBin”的状态节点则将其附带的渲染叶（RenderLeaf）送入“透明体渲染元”，于其中采用按深度值降序的方式排序绘制，以获得正确的透明体渲染结果；未设置渲染细节的状态节点则直接由根节点（渲染台，RenderStage）负责维护。

一个渲染元中可以保存一个或多个状态节点（或渲染叶）；一个状态节点（或渲染叶）只能置入一个渲染元中。

上面的场景结构中，我们并没有设置 Geode 节点_geode3 的渲染细节。如果设置它也采用“RenderBin”或者“DepthSortedBin”方式，按照指定的渲染顺序号来绘制，那么在渲染树中_geode3 节点及其附带的几何体将构成更复杂的结构形式。例如：

ss03->setRenderBinDetails(1, “RenderBin”);

其它均不变。此时虽然 ss03 和 ss14 的渲染细节设置完全一样，但是由于关联 ss03 和 ss14 的节点之间是父子的关系，因此渲染树的结构将变为：



注意两个渲染状态（StateSet）的渲染顺序号相同时，它们不一定放入同一个“渲染元”中，这还取决于两个 StateSet 对象在状态树中所处的层次。有关渲染细节设置的例子，可以参考 osghangglide。

最后，我们分别用一句话来总结“状态树”与“渲染树”的这几个组成类。之所以选择在经历了如此冗长的篇幅之后再作定义，也是为了便于读者进行归纳和总结，或者在阅读和实践的过程中提出自己的见解。

osgUtil::StateGraph：状态树的分枝节点（状态节点），负责管理场景树中的一个渲染状态（StateSet）对象，末端的 StateGraph 节点还负责维护一个 “渲染叶”（RenderLeaf）的列表。

osgUtil::RenderLeaf：状态树的叶节点（渲染叶），负责管理和绘制场景树末端的一个几何体（Drawable）对象。

osgUtil::RenderStage：渲染树的根节点（渲染台），负责管理默认渲染顺序的所有末端 StateGraph 节点(附带“渲染叶”),并保存了“前序渲染”(pre-render)和“后序渲染”(post-render)的渲染台指针的列表。

osgUtil::RenderBin：渲染树的分枝节点（渲染元），负责管理自定义渲染顺序的末端 StateGraph 节点（附带“渲染叶”）；渲染树的根节点和分枝节点最多只能有“RenderBin”和“DepthSortedBin”两类子节点，但可以根据不同的渲染顺序号衍生出多个子节点，它们在渲染时将按照顺序号升序的次序执行绘制。

解读成果：

状态树（StateGraph/ RenderLeaf），渲染树（RenderStage/ RenderBin）。

悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？

第二十一日

当前位置：osgUtil/SceneView.cpp 第 670 行，osgUtil:: SceneView::cull ()

有了前面两日的基础之后，我们从今日开始阅读场景视图 SceneView 的源代码。

场景的筛选函数 cull 主要完成了以下几个工作：

1、初始化必要的 SceneView 类成员变量，包括该视图的渲染信息（_renderInfo），筛选访问器（_cullVisitor），状态树根节点（_stateGraph）和渲染树根节点（_renderStage）。此外还有局部渲染状态_localStateSet 的更新（SceneView::updateUniforms），更新的主要内容是一些内设的 osg::Uniform 着色器变量（osg_FrameNumber，osg_FrameTime，osg_DeltaFrameTime，osg_SimulationTime，osg_ViewMatrix，osg_ViewMatrixInverse）。我们可以在编写 GLSL 程序时调用这些变量获取 OSG 提供的一些场景和时间方面的信息。

2、如果之前我们设置了立体显示的选项（参见第三日的相关内容），那么此时 OSG 会针对左/右眼（osg::DisplaySettings::LEFT_EYE/RIGHT_EYE）以其它各种设置做出适当的处理，相关的函数包括 SceneView 类成员 computeLeftEyeProjection，computeLeftEyeView，computeRightEyeProjection，computeRightEyeView 等，这里不再做深入的研究。

3、执行 SceneView::cullStage 函数，它也是场景视图筛选工作的核心函数。

4、执行 CullVisitor::clampProjectionMatrix，根据远/近平面的取值，重新设定场景视图的投影矩阵。由于远/近平面是由筛选访问器计算出来的，有的时候我们可能不希望按照它的计算值来进行视图的处理，此时可以使用 setClampProjectionMatrixCallback 设置 SceneView 的投影矩阵计算回调，自己编写相关的处理函数。

当前位置：osgUtil/SceneView.cpp 第 805 行，osgUtil:: SceneView::cullStage ()

这个函数的工作流程如下：

1、首先统计场景中的遮挡节点（OccluderNode），并使用 CollectOccludersVisitor 访问器遍历场景中的所有节点。代码如下：

```
for(unsigned int i=0; i< _camera->getNumChildren(); ++i)
  _camera->getChild(i)->accept(*_collectOccludersVisitor);
```

注意，SceneView::_camera 的值取自视景器的主摄像机，视景器的 setSceneData 函数中，会自动场景根节点设置为各个主/从摄像机的子节点（使用 View::assignSceneDataToCameras 函数），因此这里访问器的 accept 函数将遍历场景中所有的节点（815-856 行）。

2、将筛选所需的数据送入筛选访问器（CullVisitor），包括筛选设置（CullSettings），状态树根节点（StateGraph），渲染树根节点（RenderStage），渲染信息（RenderInfo）。注意此时状态树和渲染树还没有生成，我们还要设置渲染树构建所需的各种信息（860-910 行）。

3、下一步，首先将上一日提到的“全局状态节点”和“局部状态节点”追加到状态树中。这里用到了两个重要的函数 CullVisitor::pushStateSet 和 popStateSet。马上我们就对它们加以介绍。

4、使用筛选访问器遍历场景中的节点，在遍历过程中将筛选出那些无法被用户看到的对象，并将它们裁减掉，从而提高场景绘制的效率。因此，下面这段代码可以说是整个场景筛选过程的核心了：

```
for(unsigned int childNo=0; childNo<_camera->getNumChildren(); ++childNo)
```

_camera->getChild(childNo)->accept(*cullVisitor);

对它的深入研究也将随后进行。

5、依次执行 RenderStage::sort 和 StateGraph::prune 函数，对筛选访问器执行之后得到的渲染树内容进行排序和精简（构建过程中可能有些空节点需要剔除）。

6、最后，计算出场景中动态对象（DYNAMIC）的数目，并保存到 SceneView 的成员变量_dynamicObjectCount 中。如果您还记得我们在第十八日中讨论的内容的话就会知道，这个变量将随后被 SceneView::getDynamicObjectCount 函数获取，并用于多线程模式下渲染线程与场景更新的协调控制。

当前位置：osgUtil/CullVisitor 第 103 行，osgUtil::CullVisitor::pushStateSet()

状态树与渲染树的构建都是在 pushStateSet 和 popStateSet 函数中完成的。而 CullVisitor::apply 函数（在遍历节点时调用）则负责根据不同的节点类型，在不同的时机调用这两个函数。那么下面我们就看一下 pushStateSet 函数的实现过程。

pushStateSet 完成的主要工作有（传入参数 StateSet* ss）：

1、状态树的构建。判断传入的渲染状态 ss 是否已经存在于某个状态节点中，并将状态树的当前位置（CullVisitor::_currentStateGraph）转到那个节点或者新建一个包含了 ss 的状态节点（StateGraph::find_or_insert 的工作），即：

_currentStateGraph = _currentStateGraph->find_or_insert(ss);

2、渲染树的构建。创建新的渲染树节点（渲染元）有三个条件：一是渲染状态没有采用覆盖渲染细节（OVERRIDE_RENDERBIN_DETAILS）的方式（由 setRenderBinMode 函数设置），二是使用 setRenderBinDetails 设置了渲染细节，三是渲染细节的字符串名称不为空（事实上也不能随意写，只能为“RenderBin”或“DepthSortedBin”）。如果不满足这些条件的话，渲染树的当前位置（CullVisitor::_currentRenderBin）就不会发生变化；否则将尝试转到指定的节点或者新建一个渲染元（RenderBin::find_or_insert 的工作），并使用堆栈记录上一次在渲染树中的位置，即：

_renderBinStack.push_back(_currentRenderBin);

_currentRenderBin =

_currentRenderBin->find_or_insert(ss->getBinNumber(),ss->getBinName());

此外，渲染树的构建过程中只生成空的渲染元（RenderBin）节点，向其中纳入状态节点和渲染叶的任务将在后面的工作中完成。

popStateSet 的任务正好与 pushStateSet 相反：

1、从堆栈中取出上一次渲染树中所处的渲染元节点，并跳转到这一位置，即：

_currentRenderBin = _renderBinStack.back();

_renderBinStack.pop_back();

2、状态树从当前位置跳转到其父节点，即：

_currentStateGraph = _currentStateGraph->_parent;

想象一下，如果我们在遍历场景节点树时，使用 pushStateSet 将某个节点的渲染状态置入，然后再将它的子节点的渲染状态置入，如此反复……结束这个子树的遍历时，则依次使用 popStateSet 弹出_currentRenderBin 和_currentStateGraph，直到返回初始位置为止。如此即可在遍历节点子树的过程中构建起渲染后台的状态树和渲染树；并且，假如在筛选（CULL）过程中我们判断某个节点（及其子树）应当被剔除掉时，只要跳过 pushStateSet 和 popStateSet 的步骤，直接返回，就不会在渲染时留下节点的任何蛛丝马迹。

没错，这就是 CullVisitor 的工作了，也是我们下一日将要介绍的内容。

解读成果：

SceneView::cullStage，CullVisitor::pushStateSet，CullVisitor::popStateSet。

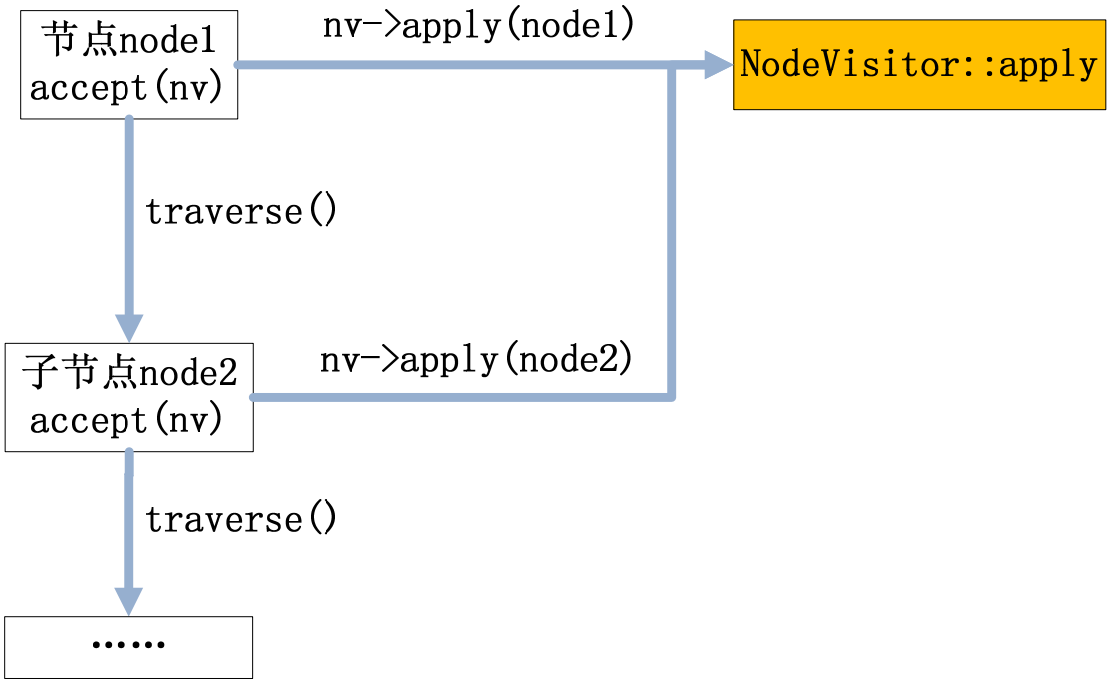
悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？

第二十二日

当前位置：osgUtil/CullVisitor.cpp 第 738 行，osgUtil::CullVisitor::apply(Node&)

在深入 CullVisitor 的源代码之前，我们先来回忆一下节点访问器（NodeVisitor）的工作原理，如下图所示：



当我们执行节点的 accept(NodeVisitor* nv)函数时，当前节点自动调用 NodeVisitor::apply 方法，将自身的信息传递给节点访问器 nv，由它负责执行相应的处理工作；然后节点将自动执行 Node::traverse 函数，调用所有子节点的 accept 函数，从而实现了节点树的遍历。在遍历的过程中每个节点都会调用 NodeVisitor::apply 将自身的指针传递给访问器，因此 NodeVisitor 的每个派生类都会重载针对各个节点的 apply 函数，以实现针对不同类型节点的访问操作。

筛选访问器(CullVisitor)的 apply 重载函数针对 Geode，Billboard，LightSource，ClipNode，TexGenNode，Group，Transform，Projection，Switch，LOD，ClearNode，Camera，OccluderNode，OcclusionQueryNode 以及通用的 Node 类型节点执行了相应的筛选工作。由于本文的篇幅已经足够长了，因此这里仅仅对比较常见的 Transform，Geode 和 Camera 三种节点的 apply 操作进行介绍，其它类型节点的访问和筛选代码不妨以此类推。

当前位置：osgUtil/CullVisitor.cpp 第 1008 行，osgUtil::CullVisitor::apply(Transform&)

首先执行的是 isCulled 函数。虽然只是个内联函数，但它却是 OSG 场景筛选的主要工具：如果这个函数的返回值为 true，说明当前节点（及其子树）应当被裁减出场景图形；不过我们也可以使用 Node::setCullingActive 设置某个节点始终不会被剔除。追踪 isCulled 的源

代码可以发现，它最终将执行到 CullingSet::isCulled 函数，并依次执行第十日所述的视锥体筛选，细节筛选和遮挡筛选这几种类型的筛选工作，判断节点是否可以被显示。

下一步执行的是 pushCurrentMask 函数，它的工作是记录当前节点视锥体筛选计算的结果（即，视锥体的哪几个面与节点的包围球有交集），并将这个结果压入堆栈，以便为下一次的计算提供方便。读者不妨自行阅读 osg::Polytope::contains 系列函数以了解其实现过程。

随后，尝试获取节点的渲染状态（StateSet），如果存在的话，使用上一日所述的函数 pushStateSet，将这个 StateSet 对象置入状态树和渲染树中，添加到对应的状态节点/渲染元中，或者为其新建一个相关的节点。

然后的工作是计算并储存 Transform 节点的位置姿态矩阵。这一步可以说是 Transform 节点在节点树中主要作用的体现。CullVisitor 将尝试为矩阵变换节点提供一个存储矩阵（使用 CullStack::createOrReuseMatrix），并使用 CullStack::pushModelViewMatrix 将计算得到的世界矩阵（Transform::computeLocalToWorldMatrix）压入堆栈，供后面的场景绘制和相应的用户回调使用。

执行 CullVisitor::handle_cull_callbacks_and_traverse 函数，它的任务正如它的名字一样：处理用户自定义的节点筛选回调（Node::setCullCallback），并使用 traverse 将访问器对象传递给所有的子节点。

之后的工作可以说是前几步的“逆操作”，即先后“弹出”模型视点矩阵（所用函数为 popModelViewMatrix，事实上只是弹出堆栈中的临时数据，计算结果仍然保留，下同），渲染状态（使用 popStateSet）和筛选结果掩码（popCurrentMask）。

当前位置：osgUtil/CullVisitor.cpp 第 759 行，osgUtil::CullVisitor::apply(Geode&)

主要的工作流程如下，一些与前文重复的函数和操作这里不再赘述：

1、执行 isCulled 函数，实现叶节点的筛选。

2、执行 pushStateSet 函数，根据 Geode 的渲染状态构建状态树和渲染树。

3、执行 handle_cull_callbacks_and_traverse 函数，处理筛选回调并传递到子节点。

4、遍历 Geode 节点储存的所有几何体对象（Drawable），执行几何体的筛选。这里可以选择使用用户自定义的几何体筛选回调（Drawable::setCullCallback）来进行处理，OSG 则使用 isCulled 函数和 CullVisitor::updateCalculatedNearFar 函数（计算几何体的边界是否超出远/近平面）来执行几何体对象的筛选工作。

5、执行 pushStateSet 函数，根据 Drawable 对象的渲染状态构建状态树和渲染树。

6、使用 CullVisitor::addDrawableAndDepth 函数，将几何体对象及其深度值置入状态树和渲染树。这一步在渲染后台树状结构的构建上有着举足轻重的作用，当前叶节点附带的所有 Drawable 几何体对象将被追加到第二十一日中所提及的当前状态节点（_currentStateGraph，使用 StateGraph::addLeaf）和当前渲染元（_currentRenderBin，使用 RenderBin::addStateGraph）当中，从而真正为状态树和渲染树添加了实质性的可绘制内容。

7、执行多次 popStateSet 函数，将_currentStateGraph 和_currentRenderBin 指针跳回到原先的位置，保证在遍历场景其余节点时状态树和渲染树的位置正确。

当前位置：osgUtil/CullVisitor.cpp 第 1170 行，osgUtil::CullVisitor::apply(Camera&)

当场景树中出现一个摄像机节点时，它以下的场景子树将按照这个摄像机的筛选、视口、观察矩阵和投影矩阵设置进行显示。我们也可以使用此摄像机指向另一个图形设备（窗口），Camera 节点的特性使得 HUD 文字，鹰眼图等效果都可以在 OSG 的场景中轻松实现。

与前文所述的步骤类似，Camera 节点总体上要完成这样一些工作：

1、加载当前 Camera 的筛选设置（setCullSettings），并保存之前的设置。

2、加载当前 Camera 的遍历掩码（setTraversalMask），这里的遍历掩码往往是用户使用 CullSettings::setCullMask 函数设置的，节点掩码（setNodeMask）与遍历掩码“与”操作之后为 0 的节点将不会在当前摄像机中显示。

3、得到当摄像机的视口，投影矩阵和模型视点矩阵，并依次压入堆栈（pushViewport，pushProjectionMatrix 与 pushModelViewMatrix 函数）。

4、这一步工作应当说是摄像机节特有的。当我们使用 Camera::setRenderOrder 设置了摄像机渲染的顺序时，这里将针对采用 PRE_RENDER 和 POST_RENDER 方式的摄像机新建一个“渲染台”（RenderStage），并使用摄像机的相关参数来初始化这个渲染台。

此时 Camera 节点的子树将全部追加到新建的“渲染台”当中（并根据渲染细节的设置生成渲染树），最后使用 addPreRenderStage 或者 addPostRenderStage 函数将新建渲染台追加到当前 RenderStage 对象的相应列表当中。在渲染过程中，各个摄像机将按照渲染台的顺序执行渲染工作。

对于设置为 NESTED_RENDER 的摄像机（默认设置），不存在前序渲染/后序渲染这一说法，因此直接执行前文所述的 handle_cull_callbacks_and_traverse 函数，继续向子节点遍历。

5、后面的工作无非是从堆栈中依次弹出模型视点矩阵，投影矩阵和摄像机视口的临时计算量，以及恢复遍历掩码和筛选设置的原先值。从而回到上级摄像机的控制当中，继续场景图形的遍历工作。

解读成果：

CullVisitor::apply（针对 Transform，Geode 和 Camera 节点）。

悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？

第二十三日

当前位置：osgUtil/RenderStage.cpp 第 146 行，osgUtil::RenderStage:: sort()

通过 CullVisitor 的使用，我们已经成功地构建了 OSG 系统的状态树和渲染树，并在这一过程中使用 isCulled 函数剔除了场景中对渲染没有助益的对象（在下一次遍历时，还会重新审核所有的节点，上一次被剔除的节点可能在新的循环中将被显示出来）。

根据第二十一日中的表述，在状态树和渲染树构建完毕之后，我们将依次执行 RenderStage::sort 和 StateGraph::prune 函数，以完成对渲染树中数据的排序和优化。

RenderStage::sort 函数的执行是按照前序渲染台，当前渲染台，后序渲染台的顺序进行的，其中前序渲染台（RenderStage::_preRenderList）和后序渲染台（_postRenderList）是通过上一日所述的 Camera::setRenderOrder 实现的，它们保存了指定摄像机及其场景子树的渲染树结构。

渲染树及其各个分支中数据的排序工作事实上是通过 RenderBin::sortImplementation 函数实现的，如果我们希望实现自定义的渲染树排序动作，通过 RenderBin::setSortCallback 函数为根节点渲染台（可以从摄像机的 SceneView 对象中取得）设置新的排序回调即可，当然前提是您知道自己应该做什么（^_^）。

注意需要排序的对象仅仅是渲染树中各个渲染元（RenderBin）中保存的状态节点（StateGraph）或者渲染叶（RenderLeaf），渲染元之间不需要进行排序（那样会打乱实际的绘制顺序）。

前文中我们反复说过，StateSet 对象的渲染细节可以设置为“RenderBin”（不透明体）

或者“DepthSortedBin”（透明体，按深度排序）。对于设置为“RenderBin”或者缺省形式的渲染状态来说，再次进行排序的意义实际上不大；因此 OSG 事实上仅针对“DepthSortedBin”渲染元中的各个渲染叶（RenderLeaf）进行排序，排序函数为 RenderBin::sortBackToFront，其中按照深度值降序的原则使用 std::sort 执行所有元素的排序动作。这里面的深度值是在 CullVisitor::apply(Geode&)函数中计算出来的（CullVisitor.cpp，820 行）。

除了上文介绍的“RenderBin”所使用的排序方式（SORT_BY_STATE，但事实上是不排序）和“DepthSortedBin”所使用的方式（SORT_BACK_TO_FRONT）之外，OSG 还内置了另外两种排序的方式，可以使用 RenderBin::setSortMode 加以指定：

SORT_BY_STATE_THEN_FRONT_TO_BACK：首先获取当前渲染元所保存的所有头状态节点（StateGraph），将每个节点中所有的渲染叶对象按深度升序排序；然后将各个状态节点按最小深度值升序排序（即，保存有深度值最小的渲染叶的节点排在最前）。

SORT_FRONT_TO_BACK：与 SORT_BACK_TO_FRONT 方式正相反，采用深度值升序的原则执行所有元素的排序。

结束了渲染树的排序之后，StateGraph::prune 函数的工作仅仅是查找状态树（StateGraph）中有没有无效的状态节点，并将它们删除。

当前位置：osgUtil/SceneView.cpp 第 947 行，osgUtil:: SceneView::cullStage ()

根据第二十一日中所述，场景筛选的最后一步是统计出场景中动态对象（DYNAMIC）的数目，并保存到 SceneView 的成员变量_dynamicObjectCount 中，供线程同步时使用。

负责统计的函数是 RenderBin::computeNumberOfDynamicRenderLeaves，它负责统计所有 RenderLeaf::_dynamic 设置为 true 的渲染叶的数目。而这个_dynamic 变量则是由对应的 Drawable 对象或者包含此渲染叶的 StateGraph 节点的数据变度值所决定的（状态节点的状态变度则由其中的 StateSet 对象决定）。

在用户程序中，设置数据变度的方式众所周知：

obj->setDataVariance(osg::Object::DYNAMIC);

场景筛选（CULL）的流程就介绍到这里，如果您咬着牙或者悠然地读完了这些看似错综复杂的源代码，或许会认为这其实也并不太过复杂。没错，也许您的脑海中已经有了更好的场景筛选算法，那么还等什么，快些将其付诸实践吧。

下面我们将开始场景绘制源代码的阅读，相关的函数是 SceneView::draw。请记住我们现在还假设处于单线程（SingleThreaded）的运行模式下，并着力于解释渲染后台的筛选和绘制流程。后面我们将重点介绍 OSG 的多线程运行机制，并将眼光放在多个渲染线程的同步性实现上，不过现在还需要忍耐一段时间。

当前位置：osgUtil/SceneView.cpp 第 983 行，osgUtil:: SceneView::draw ()

场景的绘制工作依然由 OSG 内部的场景视图类（SceneView）负责引领，而真正的绘制工作则通过渲染树的遍历，分散到各个 Drawable 类当中执行。

SceneView::draw 函数的第一个工作是初始化 osg::State 类的 GL 库函数。State 类在之前已经提到过，它保存了所有的 OpenGL 状态和属性参数；除此之外，State 类还负责从当前系统平台的 OpenGL 链接库中获取函数的地址，这也是我们第一次执行场景绘制之前的必备工作，所用函数为 State::initializeExtensionProcs。

之后，如果用户设置了场景视图初始化访问器（SceneView::setInitVisitor），那么 draw 函数第一次执行时将使用这个访问器遍历场景树。相关代码位于 SceneView::init 函数中。

然后将所有已经标记为要删除的节点或者 Drawable 对象统一从场景和内存中删除，执行 flushDeletedGLObjects 函数（参见第十八日）。每一帧绘制之前都会执行这一删除操作；

而多线程渲染的过程中，每个图形线程也都可能执行这一删除操作。这样可能会出现多个线程同时申请使用 flushDeletedGLObjects 删除对象的情况，为此，SceneView 类提供了一个成员变量_requiresFlush，用以避免多个图形线程同时执行对象的清理工作。

下一步就是场景绘制的核心工作了，OSG 为立体显示提供的支持也在这里体现出来（1010-1475 行）。针对不同的立体显示设置（DisplaySettings::getStereoMode），此处均提供了详尽的处理流程（譬如 ANAGLYPHIC 互补色显示，OSG 将负责使用红色掩码渲染左眼视图，使用补色青色掩码渲染右眼视图），感兴趣的朋友不妨在这里细细品味，甚至把您所设计的立体显示方案于其中实现。

现在我们仅针对非立体显示的情形进行介绍（1477-1510 行）：

1、首先是设置渲染台（RenderStage）的读/写缓存（通常包括 GL_NONE，GL_FRONT_LEFT，GL_FRONT_RIGHT，GL_BACK_LEFT，GL_BACK_RIGHT，GL_FRONT，GL_BACK，GL_LEFT，GL_RIGHT，GL_FRONT_AND_BACK 以及 GL_AUX 辅助缓存），其中的值是根据摄像机的 setDrawBuffer 和 setReadBuffer 函数来设定的。

2、确保颜色掩码的每个颜色通道都是被激活的（使用 osg::ColorMask）。

3、执行“前序渲染”渲染台的绘制（RenderStage::drawPreRenderStages）。

4、执行当前渲染台（即渲染树的根节点）的绘制（RenderStage::draw），无疑这是场景绘制的核心部分。

在结束了渲染树的绘制之后，SceneView::draw 函数还负责恢复所有的 OpenGL 状态（使用 State::popAllStateSets 函数），判断是否在绘制过程中出现了 OpenGL 绘图命令错误，并将错误信息打印出来。

下面我们转入 RenderStage::draw 的战场。

当前位置：osgUtil/RenderStage.cpp 第 993 行，osgUtil::RenderStage::draw ()

首先，简要地分析一下 RenderStage::draw 函数的执行流程：

1、执行摄像机的初始化回调（Camera::setInitialDrawCallback）。

2、运行摄像机设置（RenderStage::runCameraSetUp），详细的步骤我们将在下一日中讲解。

3、为了保证各个图形处理线程之间不会产生冲突，这里对当前调用的图形设备指针（GraphicsContext）做了一个检查。如果发现正在运行的图形设备与渲染台所记录的当前设备（RenderStage::_graphicsContext）不同的话，则转换到当前设备，避免指定渲染上下文时（GraphicsContext::makeCurrent）出错。

4、执行摄像机的绘制前回调（Camera::setPreDrawCallback）。

5、下一步就是实际的场景绘制工作了。对于多线程模型来说，这里将向图形设备线程（GraphicsContext::getGraphicsThread）添加一个新的 Operation 对象 DrawInnerOperation，专用于绘制工作；以及一个阻塞器 BlockAndFlushOperation（同为 Operation 对象），它强制在绘制结束之后方能继续执行线程的其它 Operation 对象。有关图形线程与 Operation 对象的关系，虽然已经日渐明朗，不过目前它还是“悬疑列表”的一部分。

6、对于单线程模型，这里将直接执行 RenderStage::drawInner 函数。后面会着重对这个函数进行介绍。

7、如果设定了摄像机的 RTT（纹理烘焙）方式，则执行 RenderStage::copyTexture 函数，将场景拷贝到用户指定的纹理对象中。注意这与摄像机的“渲染目标实现方式”有关。还记得吗？第八日中我们曾经提及过 Camera::getRenderTargetImplementation 并将其遗忘在“悬疑列表”中很久了，那么，也许很快就可以让它重现天日。

8、执行摄像机的绘制后回调（Camera::setPostDrawCallback）。

9、对于单线程模型来说，这个时候应当使用 glFlush 刷新所有 OpenGL 管道中的命令，并释放当前渲染上下文（GraphicsContext::releaseContext）。

10、执行“后序渲染”渲染台的绘制（RenderStage::drawPostRenderStages）。

11、执行摄像机的绘制结束回调（Camera::setFinalDrawCallback）。可见场景绘制时总共会执行五种不同时机下调用的摄像机回调（尤其注意回调时机与渲染上下文的关系），根据我们的实际需要，可以选择在某个回调中执行 OpenGL 函数（初始化与结束回调时不能执行）或者自定义代码，完成所需的操作。

更多的内容，待下一日分解。

解读成果：

SceneView::cull，摄像机的五种回调。

悬疑列表：

什么是“渲染目标实现方式”？Operation 对象在线程中的应用时机是什么？

第二十四日

当前位置：osgUtil/RenderStage.cpp 第 1014 行，osgUtil::RenderStage::draw ()

首先我们要解决的问题是 RenderStage::runCameraSetUp 和 RenderStage::copyTexture 这两个函数的工作内容。事实上也就是对“悬疑列表”中“什么是渲染目标实现方式”这个搁置了十数日的话题作一番讨论。

我们第一次接触到 Camera::getRenderTargetImplementation 函数是在“第八日”中，讲解 Viewer::eventTraversal 函数时（Viewer.cpp，623 行），不过事件遍历的代码中并没有体现出“渲染目标”或者“Render Target”的任何作用来。事实上，这个函数帮助我们实现了一个场景渲染过程中可能非常重要的功能，即纹理烘焙（Render To Texture，RTT），或者称之为“渲染到纹理”。RTT 技术意味着我们可以将后台实时绘制得到的场景图像直接作为另一个场景中对象的纹理，从而实现更加丰富的场景表达效果。

在 OpenGL 的较早版本中，RTT 技术的实现主要是通过从帧缓存（Frame Buffer）中取得数据并传递给纹理对象来实现的；而随着硬件水平的发展，现在我们有了帧缓存对象（Frame Buffer Object），像素缓存（Pixel Buffer）等多种绘制平台的选择。

RTT 实现的基本步骤为：（1）首先创建一个“渲染纹理”（Render Texture），例如 FBO 对象，像素缓存对象等；（2）设置它为图形设备的渲染目标（Render Target）；（3）将“渲染纹理”绑定到一个纹理或图片对象上；（4）此时图形设备的渲染将在后台进行，其结果将直接体现在所绑定的纹理对象上。

“渲染目标”的设定即通过 Camera::getRenderTargetImplementation 函数实现。其可用的传入参数中，FRAME_BUFFER 表示帧缓存，可以适用于较广泛的硬件平台上；而 FRAME_BUFFER_OBJECT 表示 FBO 对象，它可以用来实现离屏渲染（Offscreen Rendering）的工作，其渲染结果不会体现在图形窗口中。

设置渲染目标和绑定纹理的方法十分简单，例如：

```
osg::Texture2D* texture = new osg::Texture2D;
camera->setRenderTargetImplementation( osg::Camera::FRAME_BUFFER );
camera->attach( osg::Camera::COLOR_BUFFER, texture );
```

即可将纹理对象 texture 与场景绘制的帧缓存绑定在一起。我们既可以将 texture 的内容保存成图片，作为场景的截图；也可以将纹理绑定到某个物体上，实现纹理烘焙的效果。这

里 osgprerender 是一个很好的例子，使用附加参数--fb，--fbo，--pbuffer，--window 等可以充分了解不同渲染目标实现的过程及其差异。

绑定到摄像机的实际纹理或者图片，在 Camera 类中均使用 Camera::Attachment 结构体来保存。而 RenderStage::runCameraSetUp 则反复遍历名为 Camera::BufferAttachmentMap 的映射表，检索并设置那些与颜色缓存（COLOR_BUFFER），深度缓存（DEPTH_BUFFER）等相对应的 Attachment 对象；RenderStage::copyTexture 则负责针对 FRAME_BUFFER 渲染目标，拷贝场景图像到 Attachment 对象中。

下面我们就深入 RenderStage::drawInner 函数，它正是整个场景绘制的重点部分。

当前位置：osgUtil/RenderStage.cpp 第 780 行，osgUtil::RenderStage::drawInner ()

此函数的工作首先是 FBO 对象的初始化，这里将使用 FBOExtensions::isSupported 和 FrameBufferObject::hasMultipleRenderingTargets 函数来判断显示卡是否支持 FBO 以及 MRT（多重渲染目标）扩展，并使用 FrameBufferObject::apply 来调用实际的 FBO 执行函数。OpenGL 为多重渲染目标的支持提供了多达十六个颜色缓存，在 OSG 中它们均表示为 Camera::COLOR_BUFFERi，最后一个 i 取值为 0 到 15。

如果没有启用 FBO 支持或者没有使用 MRT 的话，此时作为渲染树根节点的渲染台（RenderStage）还将负责使用 glDrawBuffer 和 glReadBuffer 分别设置场景绘制缓存和读取缓存的值(用户层次上则使用 Camera 类的成员函数 setDrawBuffer 和 setReadBuffer 来实现)。当两个缓存的值均设置为 GL_BACK 时，场景的绘制将在后台缓存完成，并可以使用 SwapBuffer 动作交换前后双缓存的数据，避免场景绘制是产生闪烁。这也是 OSG 为场景缺省摄像机自动设置的特性（通过指定 GraphicsContext::Traits::doubleBuffer 的值，我们也可以在创建新的图形窗口时设置是否使用双缓存的特性）。

第二步是我们马上要重点研究的，RenderBin::draw 函数。它负责从根节点开始遍历渲染树，并执行各个渲染叶（RenderLeaf）以及上层状态节点（StateGraph）所包含的内容。

完成场景的实际绘制工作之后，OSG 将检测并显示出场景绘制当中遇到的错误。有的时候我们会在控制台看到“Warning: detected OpenGL error … after RenderBin::draw()”的字样，这事实上就是在 RenderBin::draw 函数绘制时产生的错误。通常是因为显示卡对 OpenGL 高版本的某些函数或枚举量不支持而造成的。

后面依然是有关 FBO 的操作，包括使用 glBlitFramebufferEXT 进行解算，将结果复制到关联的纹理以及图片对象中，并结束 FBO 的调用。

需要特别注意的是：如果希望使用 FBO 来实现纹理烘焙或者场景截图的话，不可以将场景主摄像机的 setRenderTargetImplementation 直接设置为相应的枚举量，那样将无法正常地看到场景（因为主摄像机对应的渲染台已经将场景绘制的结果绑定到 FBO 上了）。正确的作法是在场景树中增加一个 Camera 节点，设置“渲染目标实现方式”为 FBO 方式；并通过 Camera::setRenderOrder 设定它的渲染顺序，设置为 PRE_RENDER 可以保证这个摄像机在主场景之前执行绘制（它创建了一个“前序渲染台”，存入 RenderStage::_preRenderList 列表），从而实现“渲染到纹理”的效果。参见 osgprerender 例子以及第二十二日所述 CullVisitor::apply(Camera&)函数的内容。

当前位置：osgUtil/RenderBin.cpp 第 387 行，osgUtil::RenderBin::drawImplementation ()

RenderBin::draw 函数的工作就是调用 RenderBin::drawImplementation 函数，当然用户也可用自定义的绘制回调（RenderBin::setDrawCallback）代替 drawImplementation 来完成这一绘制工作，当然前提是我们知道在场景绘制的过程中都需要完成什么。

我们曾经在第十七日的内容中列出了 osg::State 类的几点重要功能：（1）保存 OpenGL 的所有状态、模式、属性参数、顶点和索引数据；（2）提供了对 OpenGL 状态堆栈的处理机制，对即将进入渲染管线的数据进行优化；（3）允许用户直接查询各种 OpenGL 状态的当前值。

这里所述的第二点，对于 OpenGL 渲染状态堆栈的处理，实际上就是对于 OSG 状态树（StateGraph）的遍历处理。而各种 OpenGL 模式的开关设定（也就是我们熟悉的 glEnable 和 glDisable）实际上是通过 State::applyMode 函数完成；顶点坐标，法线坐标以及各种顶点和索引数组的设置（即 glVertexPointer，glNormalPointer 等）也是由 State 类的相关函数，如 setVertexPointer 等实现的；各种渲染属性的 OpenGL 处理函数繁多而复杂，此时 State 类将使用 applyAttribute 函数，进而调用不同渲染属性对象的 StateAttribute::apply(State&)函数，实现多种多样的渲染特性。

由此可见，osg::State 类是 OSG 与 OpenGL 的主要接口，场景状态树的遍历者和整合者，也是各种渲染状态，以及顶点值的处理途径。但是我们早已知道，OSG 的顶点坐标和索引信息是由 osg::Geometry 类负责保存的，那么负责将 Geometry 对象的数据传递给 State 对象的，就是渲染树的叶节点 RenderLeaf 了。它通过执行自己所包含的 Drawable 几何体对象的 Drawable::draw 函数，实现几何体的实际绘制；而在 Geometry 类的绘制过程中，则将自己记录的数据信息传递给 State 对象，由它负责完成顶点的载入和处理工作。

而渲染树在其中的作用，就是抽取每个渲染树节点（RenderBin）中的渲染叶（RenderLeaf）对象，交由 osg::State 整合它在状态树中继承的全部渲染状态，并将几何体数据传递给 OpenGL 管线，完成绘制的工作。

也许您对于这堆突如其来的结论颇为糊涂，没有关系，具体的代码我们将在下一日加以解析。

解读成果：

RenderStage::draw，RenderStage::drawInner。

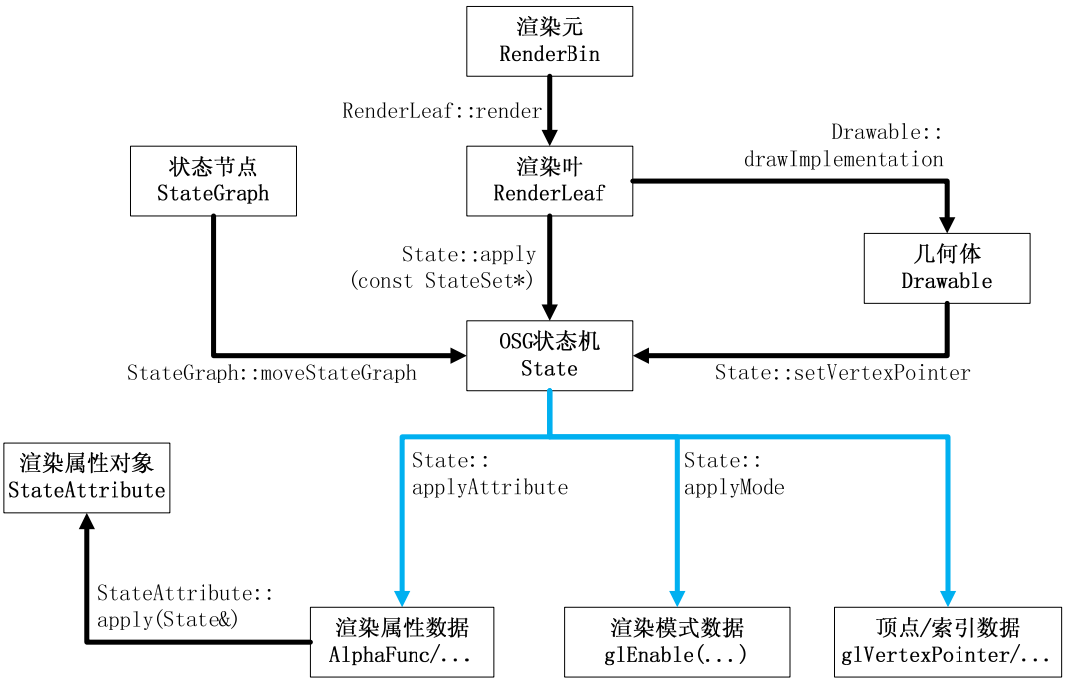
悬疑列表：

Operation 对象在线程中的应用时机是什么？

第二十五日

当前位置：osgUtil/RenderBin.cpp 第 387 行，osgUtil::RenderBin::drawImplementation ()

根据上一日所述，我们首先得到 OSG 渲染后台中渲染树（RenderStage/RenderBin），场景树（StateGraph/RenderLeaf），状态机（State），渲染属性（StateAttribute 的诸多派生类）和几何体（Drawable）之间的关系图，如下所示：



图中浅蓝色的箭头表示状态机对象中保存的各种 OpenGL 状态，即渲染属性的数据（例如 Alpha 检测，纹理，雾效等），模式数据（种种使用 glEnable/glDisable 开启或关闭的模式），以及顶点坐标、法线坐标、颜色坐标、纹理坐标，以及数据索引的数据。这些 OpenGL 编程中经常用到的概念在 OSG 中被良好地封装起来，而 osg::State 类就是它们的具体实现者。

如果我们需要自己创建新的派生自 Drawable 的对象（就像 osgText 中所实现的），或者自己创建一种新的渲染属性（派生自 StateAttribute），那么图中同样介绍了一些值得注意和借鉴的地方：Drawable 几何体对象的具体实现在于 drawImplementation 函数（事实上是通过 draw 函数间接调用的）；而渲染属性的具现函数为 StateAttribute::apply(State&)，所有的渲染属性都重写了这一函数，以实现自己的功能。

了解了这些概念之后，事实上我们已经对 OSG 的渲染流程有了大体的认识，即：

1、渲染树的作用是遍历各个渲染元（RenderBin），并按照指定的顺序执行其中各个渲染叶的渲染函数（RenderLeaf::render）。

2、状态树保存了从根节点到当前渲染叶的路径，遍历这条路径并收集所有的渲染属性数据（StateGraph/moveStateGraph），即可获得当前渲染叶渲染所需的所有 OpenGL 状态数据。

3、渲染叶的渲染函数负责向状态机（osg::State）传递渲染状态数据，进而由渲染属性类本身完成参数在 OpenGL 中的注册和加载工作；渲染叶还负责调用几何体（Drawable）的绘制函数，传递顶点和索引数据并完成场景的绘制工作。

下面我们很快地浏览一遍代码，并即将骄傲地宣布对于 OSG 场景绘制模块的解读告一段落。

当前位置：osgUtil/RenderBin.cpp 第 387 行，osgUtil::RenderBin::drawImplementation ()

执行流程如下：

1、首先判断当前 RenderBin 在渲染树中的位置，并在此位置临时插入一个新的渲染状态 RenderBin::_stateset。对于透明渲染元（TRANSPARENT_BIN），此渲染状态会自动设置一个 Alpha 检测属性（osg::AlphaFunc），以便自动剔除绘制结果中颜色 Alpha 分量为 0 的像素。因此，我们可以直接指定某个几何体的 StateSet 为 TRANSPARENT_BIN，从而自动实

现背景透明的效果（如果纹理或者颜色的 Alpha 值设置正确的话）。

2、遍历所有的子渲染元（RenderBin::_bins），其中渲染顺序号小于 0 的渲染元将在这里执行它们的 RenderBin::draw 函数，由于 draw 函数内部调用了 drawImplementation，因此这构成了一个递归调用，直至渲染树遍历至末端节点。在用户程序中，渲染顺序号的设置使用 StateSet::setRenderBinDetails 函数。

3、遍历当前 RenderBin 所保存的所有渲染叶（RenderBin::_renderLeafList），执行 RenderLeaf::render 函数，实现场景的绘制。通常只有被设置为“DepthSortedBin”的渲染元会选择保存渲染叶而非状态节点（StateGraph），因为这样便于按照深度值排序对象。

4、遍历当前 RenderBin 所保存的所有状态节点（RenderBin::_stateGraphList），获取其中保存的 RenderLeaf 对象（保存为 StateGraph::_leaves），并执行其 render 函数。

5、遍历所有的子渲染元（RenderBin::_bins），其中渲染顺序号大于 0 的渲染元此时才执行它们的 RenderBin::draw 函数。

由此可知，渲染树中最先被绘制的将是那些顺序号小于 0 的末端 RenderBin 节点，其次则依次是顺序号等于 0 的末端节点，大于 0 的末端节点，小于 0 的倒数第二级节点……而作为渲染树根节点的 RenderStage 中保存的数据将最后被渲染。

如果在渲染树的同一层中顺序号小于 0（或大于 0）的渲染元不止一个，那么它们会按照顺序号从小到大的顺序依次被渲染，这是由于 RenderBin::_bins 变量是 std::map 的类型，在顺序遍历时会自动进行数据的排列。

渲染树同一层中不可能存在渲染顺序号相同的渲染元，因为使用 setRenderBinDetails 设置了相同数字参量的 StateSet 对象被构建成状态节点（StateGraph）之后，将插入到同一个 RenderBin 中。

当前位置：osgUtil/RenderLeaf.cpp 第 20 行，osgUtil::RenderLeaf::render ()

前文中已经反复提到，渲染叶 RenderLeaf 是 OSG 渲染后台中几何体（Drawable）对象的唯一管理者；而节点树的构建，开关、变换和 LOD 等节点类型的应用，渲染状态的设置等工作，最终都要归结到几何体的渲染上来。而这里的 render 函数主要负责获取之前保存的 Drawable 指针，投影矩阵，模型视点矩阵，深度值等信息（传递这些信息的是第二十二日中提到的 CullVisitor::addDrawableAndDepth 函数），并将它们传递给负责渲染状态处理的 State 类，以及执行 Drawable::draw 函数。

它的工作流程概括如下：

1、使用 State::applyProjectionMatrix 传递投影矩阵。

2、使用 State::applyModelViewMatrix 传递模型视点矩阵。

3、如果当前渲染叶与上一次处理的渲染叶父节点不同，则需要遍历状态树中相应的路径，并更新 State 状态机中保存的渲染状态数据（采用 std::map 类型，分别名为_modeMap 和_attributeMap）。用于更新的函数为 StateGraph::moveStateGraph，它负责清除上一次使用的各种渲染状态，再沿着状态树中的路径，依次添加当前渲染叶所需的数据。最后执行函数 State::apply(const StateSet*)，由 OSG 状态机处理并执行相应的 OpenGL 指令。

4、如果当前渲染叶与上一次处理的渲染叶有相同的父节点（StateGraph 对象），则不改变传入 State 状态机的状态数据，直接执行 State::apply 函数。

5、执行此渲染叶所保存的 Drawable 对象的 draw 函数，完成几何体的绘制。Geometry 对象将在这一函数中（实际上是 Drawable::drawImplementation）向状态机传递顶点和索引数据，并交由状态机对象来完成几何数据的绘制。

解读成果：

RenderLeaf::render，RenderBin::drawImplementation。

悬疑列表：

Operation 对象在线程中的应用时机是什么？

第二十六日

当前位置：osg/State.cpp 第 376 行，osg::State::apply(const StateSet*)

在结束对 OSG 渲染后台的研究之前，我们还是看一下 OSG 状态机是如何处理传入的渲染状态（StateSet）数据的，即 State::apply 函数的工作流程：

1、使用 State::applyModeList 函数，接收渲染状态中的模式数据（使用 StateSet::setMode 设置），并通过 applyMode 函数（State 头文件，1132 行）予以执行，真正的执行代码可以浓缩为以下两行：

```
if (enabled) glEnable(mode);
else glDisable(mode);
```

这里 enabled 变量是由是否设置了 StateAttribute::ON 决定的，而 mode 就是我们通常使用 setMode 函数设置的 OpenGL 枚举量，例如 GL_NORMALIZE。

2、使用 State::applyAttributeList 函数，接收渲染状态中的属性数据（即 StateAttribute 的派生类对象，使用 StateSet::setAttribute 设置），并通过 applyAttribute 函数（State 头文件，1151 行）予以执行，它的执行代码事实上甚至可以浓缩为一行：

```
attribute->apply(*this);
```

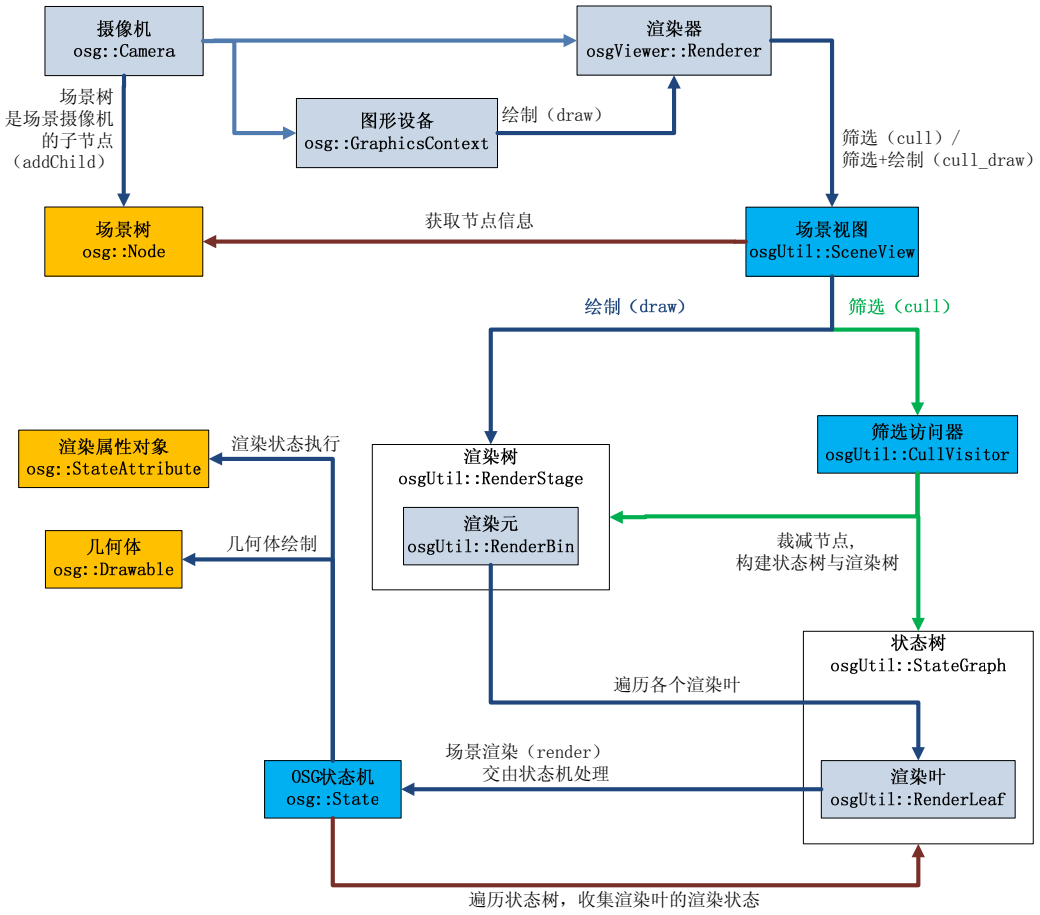
变量 attribute 就是我们设置的 StateAttribute 对象，而 this 指的是 State 类对象。换句话说，此时 State 类将执行的工作移交给了各个 StateAttribute 派生类来完成。因此，如果我们希望自己编写一个新的渲染属性类（例如，它可以同时完成雾效和图像融合的工作），只要将虚函数 StateAttribute::apply(State&)重写就可以实现它与 OSG 渲染后台的接口了，十分简单。

State 类保存了两个映射表_modeMap 和_attributeMap，用于记录当前渲染状态的执行情况，场景绘制的过程中会通过 applyModeList 和 applyAttributeList 不断更新这两个映射表，以标识那些需要被更新的以及更新完毕的渲染状态。

3、随后，针对纹理相关的渲染属性和模式进行与上两步相似的处理，之所以将纹理（Texture 及其派生类）与通常的渲染状态区分开，是因为一个 StateSet 中可以设置多个同类型的纹理对象（例如多个 Texture2D 对象），它们分别加载到不同的纹理单元上，以实现多重纹理的效果；而普通的渲染属性在 StateSet 中不具备这种特性。

4、使用 State::applyUniformList 将着色器所使用的 Uniform 变量传递下去（事实上是传递给 OSG 内部的 GLSL 预编译器 Program::PerContextProgram 处理了），这是实现 GLSL 与 OSG 系统交互的重要途径，它的实现代码就在这里。

好了，还记得我们在第十八日给出的“场景图形，摄像机，图形设备，渲染器和场景视图的关系图”吗，现在我们也许可以将它完善起来，形成 OSG 的渲染后台全景了。



OSG 渲染后台与用户层的接口是摄像机类（Camera）。场景中至少有一个主摄像机，它关联了一个图形设备（GraphicsContext，通常是窗口），以及一个渲染器（Renderer）；我们可以在场景树中（或者别的视图 View 中，对于复合视景器而言）添加更多的摄像机，它们可以关联相同的或者其它的图形设备，但都会配有单独的渲染器，用以保存该摄像机的筛选设置、显示器设置等信息。

场景筛选和绘制的工作由渲染器来完成，而图形设备 GraphicsContext 则负责根据不同时机的选择，调用渲染器的相关函数。例如在单线程模式中，ViewerBase::renderingTraversals 函数依次执行 Renderer::cull 和 Renderer::draw 函数(后者通过 GraphicsContext::runOperations 调用)，而在多线程模型中调用者的关系将更加错综复杂。

OSG 渲染后台的调度中心是场景视图（SceneView），它负责保存和执行筛选访问器（CullVisitor）。CullVisitor 负责遍历并裁减场景，同时在遍历过程中构建对于场景绘制至关重要的渲染树和状态树；生成的状态树以 StateGraph 为根节点和各级子节点（其中保存场景树的渲染状态 StateSet 数据），以 RenderLeaf 为末端叶节点的内容（其中保存场景树中的几何体 Drawable 对象）；渲染树则以 RenderStage 为根节点，RenderBin 为各级子节点，根据渲染顺序和方法的设定，状态树中的节点和渲染叶（RenderLeaf）被记录到 RenderStage 和各级 RenderBin 中；SceneView 负责保存和维护状态树和渲染树。

绘制场景时，渲染树中的各级节点将取出保存的渲染叶数据，传递给 OSG 状态机(State)。后者是 OpenGL 状态机制的封装和实现，也是场景绘制的核心元件。状态机取得渲染叶中的几何数据之后，再向根部遍历状态树，取得该几何体绘制相关的所有渲染状态设置，并亲自或者交由 StateAttribute 派生类完成渲染状态的实际设定，以及场景元素的实际绘制工作。

此时用到的就已经是我们耳熟能详的各种 OpenGL 函数了。

那么，有关 OSG 场景绘制部分的介绍，就此告终吧。希望这洋洋洒洒的汉字，在您解读 OSG 源代码时能起到些许的帮助。

当前位置：osgViewer/ViewerBase.cpp 第 243 行，
osgViewer:: ViewerBase::startThreading ()

上一次我们讨论 startThreading 函数还是在第十六日的时候，随后我们在假定使用单线程方式运行的前提下，深入到 OSG 的渲染后台中，一起浏览和讨论了场景筛选的流程和执行函数，状态树与渲染树的构建，遍历渲染树实现场景绘制的过程，遍历状态树收集对象渲染状态的过程，OSG 状态机及其与 OpenGL 状态机的关系，渲染后台与 Drawable 几何体和 StateAttribute 渲染属性的关系。这其中，场景视图类 SceneView 可谓是整个渲染后台的指挥部分，而摄像机的渲染器（Renderer）和图形设备（GraphicsContext）分别负责执行场景筛选（CULL）和绘制（DRAW）的操作。

虽然我们还有三种不同的线程处理方式要讨论(它们分别是 CullDrawThreadPerContext，DrawThreadPerContext 和 CullThreadPerCameraDrawThreadPerContext)，但总体上讲，渲染后台的处理函数和处理流程不会再有大的变化，线程模型的选择也许可以概括为一个决定谁来调用 SceneView::cull，以及谁来调用 SceneView::draw 的过程，但是其流程却远没有这么简单，完成的效果也是大相径庭。

那么首先摘录第十六日的一部分文字。

startThreading 函数一开始，先要完成这样几个工作：

1、执行 ViewerBase::releaseContext，首先释放渲染上下文；

2、如果用户没有设置线程模型，则使用 ViewerBase::suggestBestThreadingModel 自动进行判断；

3、使用 Viewer::getContexts 函数获取当前所有的图形设备（GraphicsContext）；

4、使用 Viewer::getCameras 函数获取当前所有的摄像机（主摄像机和所有从摄像机）。

这之后，对于单线程模式（SingleThreaded）来说，将直接退出函数的执行，而对于其它三种多线程的运行模式而言，一切才刚刚开始。

解读成果：

State::apply，SceneView::draw。

悬疑列表：

Operation 对象在线程中的应用时机是什么？

第二十七日

当前位置：osgViewer/ViewerBase.cpp 第 264 行，
osgViewer:: ViewerBase::startThreading ()

我们再摘录一段第十六日中叙述的文字，这是对几个多线程渲染中重要的成员变量的解释说明：

ViewerBase::_startRenderingBarrier：可以理解为渲染启动的一个栅栏标志，用于同步开始所有的图形设备的线程操作。

ViewerBase::_endRenderingDispatchBarrier：渲染结束的一个栅栏标志，用于同步结束所有的图形设备的线程操作。

ViewerBase::_endDynamicDrawBlock：用于同步结束所有的动态对象绘制操作，这里所谓的动态对象，指得是使用 Object::setDataVariance 设置为 DYNAMIC 的场景对象。

这里提到的栅栏（OpenThreads::Barrier），我们在第十一日中对它以及它的“孪生兄弟”Block（阻塞器）有过介绍。那么，为了方便起见，这里还是把那一段文字摘录过来：

BlockCount 类：计数阻塞器类。它与阻塞器类的使用方法基本相同：block()阻塞线程，release()释放线程；不过除此之外，BlockCount 的构造函数还可以设置一个阻塞计数值。计数的作用是：每当阻塞器对象的 completed()函数被执行一次，计数器就减一，直至减到零就释放被阻塞的线程。

Barrier 类：线程栅栏类。这是一个对于线程同步颇为重要的阻塞器接口，它的构造函数与 BlockCount 类似，可以设置一个整数值，我们可以把这个值理解成栅栏的“强度”。每个执行了 Barrier::block()函数的线程都将被阻塞；当被阻塞在栅栏处的线程达到指定的数目时，就好比栅栏无法支撑那么大的强度一样，栅栏将被冲开，所有的线程将被释放。重要的是，这些线程是几乎同时释放的，也就保证了线程执行的同步性。

注意 BlockCount 与 Barrier 的区别，前者是由其它任意线程执行指定次数的 completed()函数，即可释放被阻塞的线程；而后者则是必须阻塞指定个数的线程之后，所有的线程才会同时被释放。

为什么要长篇累牍地摘录这些内容呢？事实上，它对于我们后面的阅读有着至关重要的作用。_startRenderingBarrier 和_endRenderingDispatchBarrier 变量虽然是 osg::BarrierOperation 类型，其实均派生自 OpenThreads::Barrier 类；而另一个成员变量_endDynamicDrawBlock 是 osg::EndOfDynamicDrawBlock 类型，亦即 BlockCount 的派生成员。因此，后文中对于这三个变量的操作，实质上也就是对于线程阻塞与同步的操作。而这正是 OSG 多线程渲染实现的关键所在。

osg::BarrierOperation 类在定义时会传入两个参数，整型参数定义了这个栅栏可阻塞的线程数，或者说它的强度（达到这一数值时将自动释放所有线程），另一个则定义是否需要在阻塞前执行固定的操作（通常是不用的，即 BarrierOperation::NO_OPERATION）。

osg::EndOfDynamicDrawBlock 在定义时会传入一个参数，表示阻塞的最大计数值，当阻塞器对象的 completed()函数执行次数达到这一数值时，才会释放被阻塞的线程。

下面我们就来看一下，多线程模式中，这些成员量是如何被处理的。

当前位置：osgViewer/ViewerBase.cpp 第 264 行，
osgViewer:: ViewerBase::startThreading ()

这里的 switch-case 条件语句段针对几种线程模型，给出了不同的预设参数，即：渲染开启栅栏_startRenderingBarrier 的最大强度 numThreadsOnStartBarrier，以及渲染结束栅栏_endRenderingDispatchBarrier 的最大强度 numThreadsOnEndBarrier。除了单线程模型之外，其余三种线程模型的设置如下。

CullDrawThreadPerContext：这一模式下，系统将为每个图形设备（GraphicsContext）创建一个线程（GraphicsContext::createGraphicsThread）。每一帧结束前都会强制同步所有的线程。栅栏的设置为：

numThreadsOnStartBarrier = contexts.size()+1;

numThreadsOnEndBarrier = contexts.size()+1;

DrawThreadPerContext：系统将为每个图形设备创建一个线程。并且在当前帧的所有线程完成工作之前，开始下一帧。栅栏的设置为：

numThreadsOnStartBarrier = 1;

numThreadsOnEndBarrier = 1;

CullThreadPerCameraDrawThreadPerContext：系统将为每个图形设备和每个摄像机创建线程（Camera::createCameraThread）。并且在当前帧的所有线程完成工作之前，开始下一帧。栅栏的设置为：

numThreadsOnStartBarrier = cameras.size()+1;

numThreadsOnEndBarrier = 1;

以 CullDrawThreadPerContext 模式为例，如果它将为各个图形设备启动共 n 个图形线程的话，那么“渲染启动栅栏”和“渲染结束栅栏”的强度均为 n+1。这就意味着：如果每个线程在完成工作以后均使用 block()函数将自己阻塞（即栅栏承受的强度将达到 n），那么只要我们在主进程中再执行一次 block()，就会成功地冲开栅栏，释放所有被阻塞的线程。

这就为场景渲染线程的同步提供了一种可能。那就让我们转入 renderingTraversals 函数探个究竟。

当前位置：osgViewer/ViewerBase.cpp 第 662 行，
osgViewer::ViewerBase::renderingTraversals()

这里将首次同步各个图形线程：

if (_startRenderingBarrier.valid()) _startRenderingBarrier->block();

它意味着我们选择将主进程予以阻塞，此时后面的代码都不会被执行，直到这个栅栏被冲开为止。而冲开的条件就是所有图形设备（GraphicsContext，以下简称 GC）的线程都执行了_startRenderingBarrier->block()函数，并因此阻塞了运行。因此，当且仅当所有 GC 线程执行到 block 函数并被阻塞时，栅栏恰好被冲开，主进程以及各个 GC 线程继续执行各自后继的代码。

而在后面不远处，又有一处类似的代码段（701 行）：

if (_endRenderingDispatchBarrier.valid()) _endRenderingDispatchBarrier->block();

其作用与上一段本质上是相同的。而根据 OSG 中的定义，_startRenderingBarrier 用于在线程渲染工作开始之前同步各个线程，而_endRenderingDispatchBarrier 则在渲染工作结束之后执行同步。

注意在这两段代码之间的段落是我们之前介绍过的，用于在单线程方式下按顺序执行各个图形设备的筛选和绘制工作。概括起来就是

```
if (!renderer->getGraphicsThreadDoesCull() && !(camera->getCameraThread()))
        renderer->cull();
if (!(gc->getGraphicsThread()) && gc->valid())
{
        makeCurrent(gc);
        gc->runOperations();
}
```

在 CullDrawThreadPerContext 以及 DrawThreadPerContext 方式下，由于上面代码段中的“gc->getGraphicsThread()”这一条件为真，因此不再执行 GraphicsContext::runOperations，也就是不再从主进程的 renderingTraversals 函数中执行场景绘制；而在更加高效也更加复杂的 CullThreadPerCameraDrawThreadPerContext 方式下，不但场景绘制不再由主进程来完成，由于“camera->getCameraThread()”这个条件也为真，因而筛选动作也是由摄像机线程来完成的。

解读成果：

渲染栅栏（Barrier）的作用。

悬疑列表：

Operation 对象在线程中的应用时机是什么？

第二十八日

当前位置：osgViewer/ViewerBase.cpp 第 730 行，osgViewer::ViewerBase::renderingTraversals()

这里我们还落下了一个十分重要的变量_endDynamicDrawBlock，在“渲染开启栅栏”和“渲染结束栅栏”的同步代码段之间，有这样一行代码（730 行）：

```
if (_endDynamicDrawBlock.valid()) _endDynamicDrawBlock->block();
```

很显然，由于_endDynamicDrawBlock 是一个计数阻塞器，因此当且仅当我们执行了指定次数的 completed()函数之后，主进程才会被释放，后面的代码才会被继续执行。

那么 completed 函数是在哪里执行的呢？我们需要结合多处代码来进行分析。首先是 ViewerBase::startThreading 函数中（337 行）：

```
if (_threadingModel==DrawThreadPerContext ||
    _threadingModel==CullThreadPerCameraDrawThreadPerContext)
{
    ……
    _endDynamicDrawBlock = new osg::EndOfDynamicDrawBlock(
        numViewerDoubleBufferedRenderingOperation);
    ……
}
if (numThreadsOnStartBarrier>1)
{
    _startRenderingBarrier = new osg::BarrierOperation(
        numThreadsOnStartBarrier, osg::BarrierOperation::NO_OPERATION);
}
if (numThreadsOnEndBarrier>1)
{
    _endRenderingDispatchBarrier = new osg::BarrierOperation(
        numThreadsOnEndBarrier, osg::BarrierOperation::NO_OPERATION);
}
……
for(citr = contexts.begin(); citr != contexts.end(); ++citr, ++processNum)
{
    ……
    osg::GraphicsContext* gc = (*citr);
    gc->getState()->setDynamicObjectRenderingCompletedCallback(
        _endDynamicDrawBlock.get());
    ……
}
```

这里列出了相当长的一段代码，不过其中却透露了很多信息。例如_startRenderingBarrier 和_endRenderingDispatchBarrier 的初始化就是在这里完成的，不过我们更希望了解的是动态

绘制阻塞器_endDynamicDrawBlock 的初始化过程。注意，_endDynamicDrawBlock 只有在 DrawThreadPerContext 和 CullThreadPerCameraDrawThreadPerContext 两种模式下会被初始化，这是因为只有这两种模式下才会出现上一帧渲染未结束而下一帧开始的情况。而正如我们在以往的教程中所了解的，如果一个对象需要在运行时被修改，那么需要设置它为动态对象，即：

```
object->setDataVariance( DYNAMIC );
```

否则将可能出现渲染中的数据被新一帧的用户回调更改的情形，并因此造成系统崩溃或者各种无法预知的错误。由于单线程和 CullDrawThreadPerContext 模型不会出现渲染过程与用户更新过程交叠的情形，因此不必考虑这一操作。

动态绘制阻塞器_endDynamicDrawBlock 的作用就是避免 DYNAMIC 对象的运行时更改影响后台渲染的工具。在初始化时我们需要设置它的最大计数值，当 completed 函数的执行次数达到这一数值时，将释放被阻塞的线程，也就是我们的主进程。这一最大计数值正是场景中摄像机渲染器（Renderer）的个数，为什么呢？让我们转到 RenderLeaf::render 函数，其中有这样一段代码（osgUtil/RenderLeaf.cpp，77 行）：

```
if (_dynamic)
    state.decrementDynamicObjectCount();
```

而 State::decrementDynamicObjectCount 函数的内容为（osg/State，1013 行）：

```
{
    --_dynamicObjectCount;
    if (_dynamicObjectCount==0 && _completeDynamicObjectRenderingCallback.valid())
        _completeDynamicObjectRenderingCallback->completed(this);
}
```

这里 State ::_dynamicObjectCount 变量表示当前状态机中仍在等待渲染的所有动态对象的数目，它是通过 RenderStage::computeNumberOfDynamicRenderLeaves 函数计算的。

由此可见，每当一个被设置为“动态”的渲染叶（“动态”的属性事实上是从 Node、Drawable 或者 StateSet 对象继承来的）都会在渲染结束时执行动态对象计数的“减一”操作；而每当 OSG 状态机中的动态对象计数为 0 时，相应的动态对象渲染回调（这个参数实际上就指向_endDynamicDrawBlock）将执行 completed 操作。由于每个摄像机的渲染器都会根据自己的实际情况重新刷新状态机 State 的内容，并因而在全部动态对象的渲染结束时执行一次 completed 函数。因此，最终所有 n 个摄像机的内容都渲染完毕时，completed 函数也将被执行 n 次，并进而解开了计数阻塞器的阻塞，释放主进程，使之继续后面的代码，开始下一帧的运行。

因此，设置场景中的节点、几何体或者渲染状态为 DYNAMIC 对象时，将保证这些对象的渲染结束之前，不会开始下一帧的用户更新工作，也就保证用户对数据的修改操作不会对本次的渲染结果和系统的稳定性产生影响。

当前位置：osgViewer/ViewerBase.cpp 第 361 行，
osgViewer::ViewerBase::startThreading()

解决了动态对象渲染阻塞器的问题，现在我们可以转手去面对_startRenderingBarrier 和 _endRenderingDispatchBarrier 这对孪生兄弟，还有多线程渲染处理的问题了。虽然我们已经解释了有关渲染栅栏的原理以及它们的初始化代码。但是悬而未决的问题实质上还有很多：

1、图形设备线程和摄像机线程，它们是如何工作的呢？预编译、筛选、绘制，这些事情有的线程要做，有的则不必做，如何区分它们呢？在“悬疑列表”中搁置的问题“Operation 对象在线程中的应用时机”，是否就是线程工作的关键所在呢？

2、渲染开启栅栏和结束栅栏都已经有了，可是它们该如何去阻塞各个图形线程呢？况且我们怎样定义何时算是渲染开始，何时算是结束呢？

3、除了对动态对象渲染的处理之外，CullDrawThreadPerContext，DrawThreadPerContext 和 CullThreadPerCameraDrawThreadPerContext 三种线程模型之间，还有没有别的不同点？又如何去评判它们的优劣呢？

那么就让我们在下一日尝试去揭开 OSG 图形渲染线程的面纱吧。

解读成果：

动态对象渲染机理。

悬疑列表：

Operation 对象在线程中的应用时机是什么？

第二十九日

当前位置：osg/ OperationThread.cpp 第 382 行，osg:: OperationThread::run ()

首先要注意的是，当我们使用 GraphicsContext::createGraphicsThread 创建线程时，得到的是一个 osg::GraphicsThread 线程对象；而使用 Camera::createCameraThread 创建线程时，得到的是 osg::OperationThread 对象，它是 GraphicsThread 的父类。

这一微小的区别使得这两类线程之间不会存在太大的差异。事实上，图形设备所用的 GraphicsThread 线程只是在每次运行时（即 GraphicsThread::run 函数）保证设备的渲染上下文 RC 设置正确，即，在恰当的时机使用 GraphicsContext::makeCurrent 和 releaseContext 函数操作 RC 设备，并在 RC 设备正确关联之后执行 OperationThread::run 函数。

那么我们现在就来看一下 OperationThread 线程的执行内容，它事实上也是 GC 线程和摄像机线程在启动以后要反复完成的工作。流程如下：

1、取得任务队列（OperationQueue），注意这里要使用 Mutex 互斥锁，避免用户追加任务时与线程的执行产生冲突。

2、获取任务队列中的一个任务（OperationQueue::getNextOperation）。这个函数看似简单，只要从 std::list 列表中取出一个 osg::Operation 对象就可以了。但是其中还是有诸多的注意事项：

首先，如果任务列表是空的，渲染线程将选择暂时阻塞自己（使用 block 函数），直到有新的 Operation 操作加入到队列中为止。

其次，我们有一个任务列表迭代器_currentOperationIterator，如果这个迭代器已经到达列表的末尾，则自动将其转至列表首部，这样就可以在线程中循环执行任务列表中的内容。

如果迭代器取得了一个 Operation 操作任务，那么我们需要判断这个任务是否将被反复执行，即，迭代器转至任务列表首部之后，是否还可以取得这个任务。判断所用的函数是 Operation::getKeep，这个函数返回 true 时，任务将允许反复执行（例如场景筛选和绘制的任务），否则任务将被随即从列表中移除，我们也不会再取得这个 Operation 对象（除非再次将其加入列表）。

3、现在我们得到了一个 Operation 对象，那么“悬疑列表”中的问题也就迎刃而解了：Operation 对象在线程中的应用时机就在此处了。线程运行中将执行 Operation::operator()操作，并将当前的 GC 设备或者摄像机作为传入参数传入。

那么下面我们将关注这样一个问题：到底有哪些操作会被传入渲染线程的任务列表呢？这些传入的 Operation 操作又分别起到什么作用呢？

当前位置：osgViewer/ViewerBase.cpp 第 361 行，
osgViewer::ViewerBase::startThreading()

向任务列表传入新的 Operation 任务的函数为 OperationThread::add，如果需要的话，我们也可以向 GC 线程或者摄像机线程传递自己定义的任务，只是对于并不熟悉线程编程的开发者而言，这一操作极具危险性。

在第十三日我们曾提到过 DatabasePager::CompileOperation 这个操作任务，它是在数据分页线程的执行过程中，从 DatabaseThread::run 传递给对应的 GC 线程的。而它的执行函数 operator()的内容也很简单：无非是执行 DatabasePager::compileAllGLObjects 函数，对当前 GC 设备中待编译的对象执行预编译（使用 compileGLObjects 函数），第十八日中曾对这一过程作出过介绍，此处不再重复。

注意这个 CompileOperation 的 getKeep 属性为 false，因此执行完一次之后，它将被剔除出 GC 线程的任务列表

除此之外，GC 线程的主要工作任务设置都是在 startThreading 这个函数中完成的。而这个 startThreading 将在 Viewer::realize 函数中执行，因此我们尽量不要在执行 realize 之后再作多余的工作，因为此时渲染线程已经启动了。

在 startThread 函数中，根据线程模型的不同，以下几种任务对象将可能被添加到 GC 线程或者摄像机线程中：

osg::BarrierOperation：事实上也就是前文中反复提到的启动栅栏_startRenderingBarrier 和结束栅栏_endRenderingDispatchBarrier，它们同时也可以作为任务对象被添加到线程中，这就使得线程的同步控制变得十分方便：只要任务队列执行到启动栅栏或者结束栅栏，就自动使用 block 阻塞线程，直到栅栏被冲开（也就是全部线程都被阻塞的那一刻），才会继续执行后面的任务。

osg::RunOperations：这个任务将负责执行 GraphicsContext::runOperations 函数，而就像我们在第十七日中介绍的那样，runOperations 函数将通过执行渲染器的 Renderer::operator() 操作，完成场景的绘制（或者筛选加上绘制，Renderer::cull_draw）工作。

osg::SwapBuffersOperation：这个任务将负责执行双缓存交换的动作，以实现场景的平滑浏览，相关函数为 GraphicsContext::swapBuffersImplementation。

osgViewer::Renderer：没错，渲染器本身也是可以作为一个任务存在的，它将根据相应的设置直接执行场景的绘制（Renderer::draw）或者筛选加绘制（Renderer::cull_draw）操作。

那么，这几种线程模型，在任务的添加和处理顺序上各自有什么要求呢？那么就从 CullDrawThreadPerContext 开始：

CullDrawThreadPerContext 模式的渲染开始栅栏和结束栅栏的强度都设为 contexts()+1，即 GC 线程的数目加一。而任务列表中任务的顺序为：

1、_startRenderingBarrier 任务（startThreading 函数，380 行）。由于 getKeep()设置为 true，这个任务不会从列表中删除。

2、osg::RunOperations 任务（393 行），它将同时负责场景的筛选和绘制（即最终执行 Renderer::cull_draw 函数）。该任务不会从列表中删除。

3、swapReadyBarrier 任务（401 行），它实质上是一个 BarrierOperation 栅栏对象，强度等于 GC 线程的数目。它的作用是在交换双缓存之前对所有 GC 线程执行一次同步。该任务不会从列表中删除。

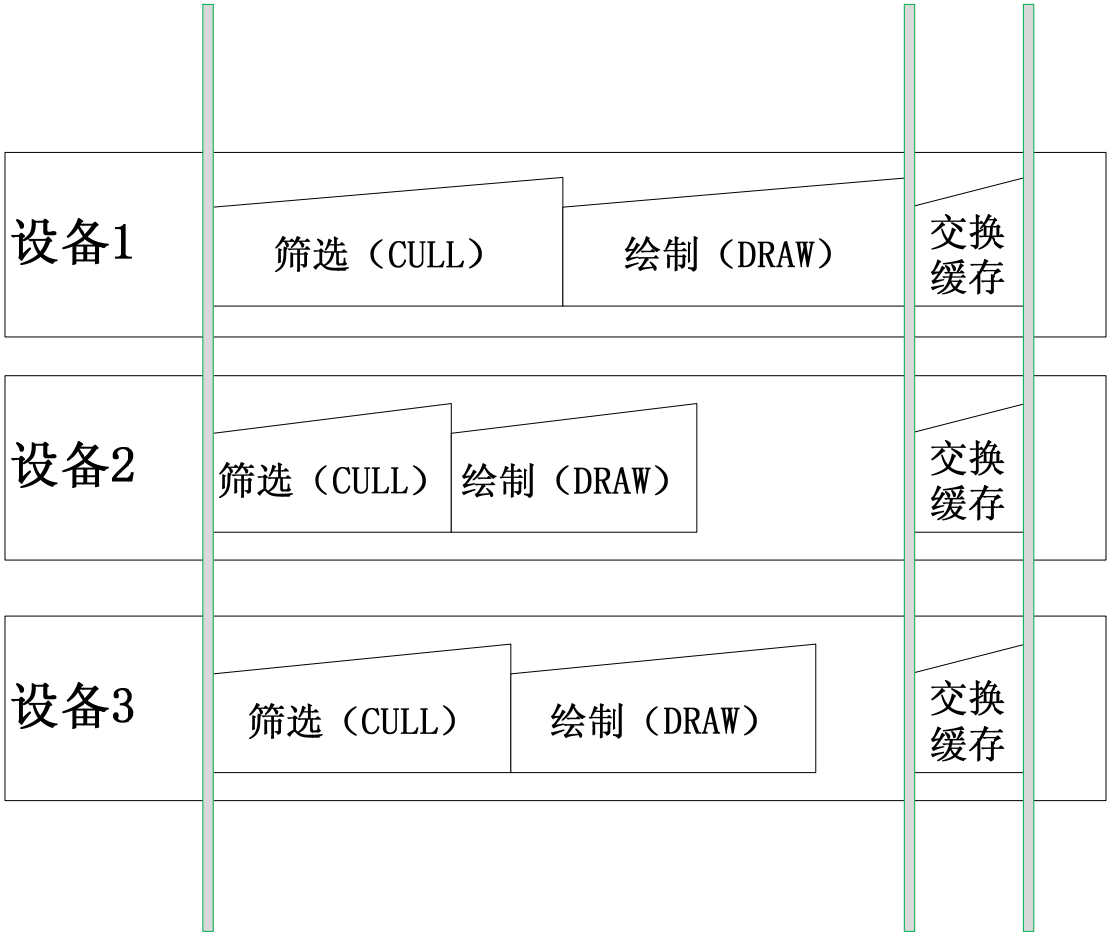
4、swapOp 任务（404 行），它是一个 SwapBuffersOperation 对象，用于执行绘制后的双缓存交换操作。该任务不会从列表中删除。

5、_endRenderingDispatchBarrier 任务（409 行），显然这就是渲染结束栅栏的位置。该

任务不会从列表中删除。

由此我们可以判断出，对于 CullDrawThreadPerContext 模式：它的渲染过程中将三次对各个 GC 线程进行同步，分别是场景筛选和绘制开始前，场景筛选和绘制完毕后，以及双缓存交换完毕后；其它时候则交由各个 GC 线程自由完成针对各个 GC 设备的场景渲染工作。

下面的示例图中，三个 GC 线程分别绘制到三个不同的图形设备，由于场景筛选所需时间以及渲染数据量等种种差异，三个线程完成筛选（CULL）和绘制（DRAW）的时间都不相同，但是 CullDrawThreadPerContext 将保证它们在运行时的同步性。



图中三个用于同步线程的栅栏，依次分别是_startRenderingBarrier，swapReadyBarrier 和_endRenderingDispatchBarrier（从左至右）。

下一日我们将接着分析另外两种线程模型的任务列表及其执行流程。

解读成果：

OperationThread，CullDrawThreadPerContext 模式。

悬疑列表：

无。

第三十日

当前位置：osgViewer/ViewerBase.cpp 第 361 行，osgViewer::ViewerBase::startThreading()

DrawThreadPerContext 模式事实上是默认的一种线程模式，如果 ViewerBase 类的成员函数 suggestBestThreadingModel 没有找到适合当前计算机系统的线程模式的话，将自动采用这一模式来完成渲染的工作。不过值得注意的是，DrawThreadPerContext 模式没有设置渲染启动栅栏_startRenderingBarrier 和结束栅栏_endRenderingDispatchBarrier。因此这一渲染模式下将不会在每一帧对场景的筛选和绘制工作进行同步，且用户的更新动作将有可能在某些线程的渲染动作还未结束时即开始运行。

DrawThreadPerContext 模式下会根据场景中摄相机的数量设置_endDynamicDrawBlock 变量的值，这个阻塞器用于在每个渲染器都渲染完毕之前阻塞主进程的运行，以免用户对数据的更新动作与动态对象的渲染动作产生冲突。

DrawThreadPerContext 需要执行的任务列表如下：

1、osg::RunOperations 任务（393 行），它将负责场景的绘制（即最终执行 Renderer::draw 函数）。该任务不会从列表中删除。

2、swapReadyBarrier 任务（401 行），它实质上是一个 BarrierOperation 栅栏对象，强度等于 GC 线程的数目。它的作用是在交换双缓存之前对所有 GC 线程执行一次同步。该任务不会从列表中删除。

3、swapOp 任务（404 行），它是一个 SwapBuffersOperation 对象，用于执行绘制后的双缓存交换操作。该任务不会从列表中删除。

这三个任务将在线程运行的过程中反复被执行。

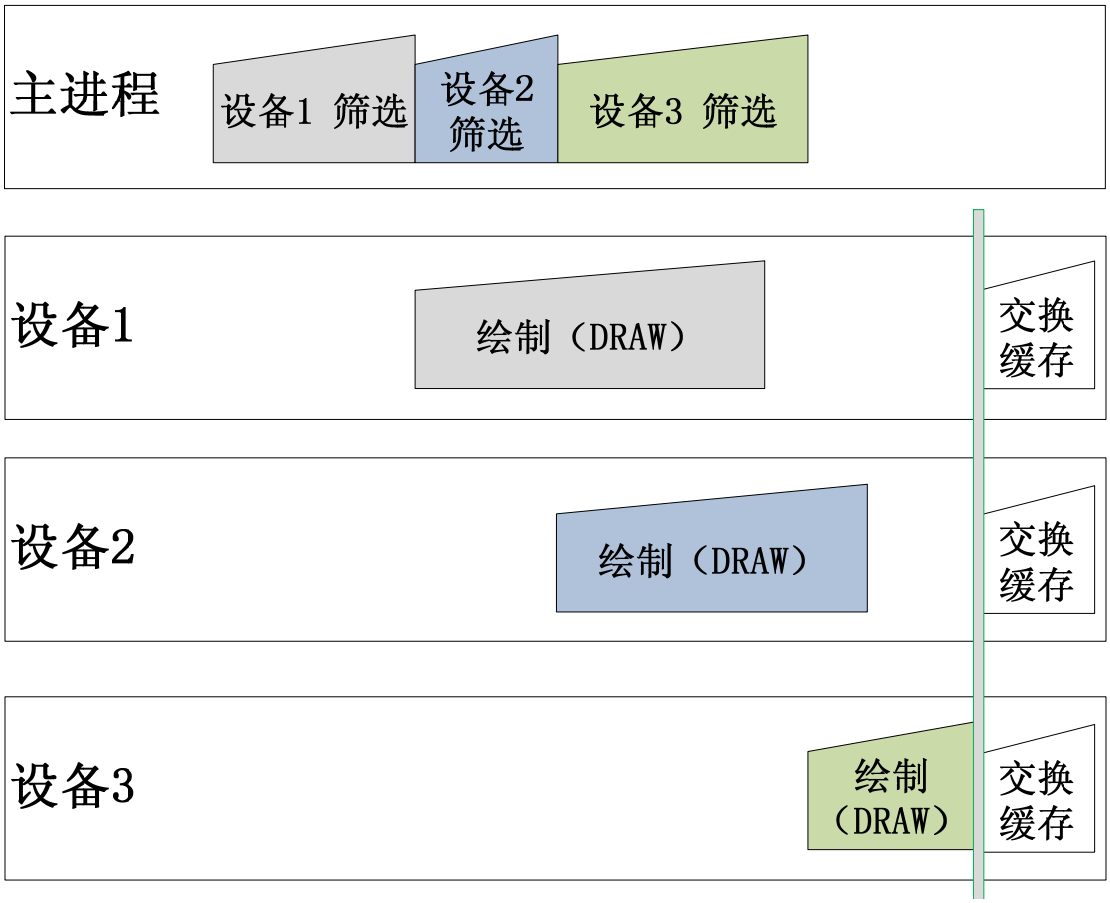
那么，场景的筛选（CULL）工作呢？线程中不负责场景元素的裁剪的话，整个场景渲染的过程中不就缺少了重要的环节吗？不过不必心急，就像单线程模式中所实现的那样，场景筛选的工作是由 ViewerBase::renderingTraversals 函数完成的，正如下面的代码所示（ViewerBase.cpp，680 行）：

```
if (!renderer->getGraphicsThreadDoesCull() && !(camera->getCameraThread()))
    renderer->cull();
```

Renderer::getGraphicsThreadDoesCull 函数只有在 CullDrawThreadPerContext 或者单线程模式下为 true，而摄像机线程 getCameraThread 只有最后一种多线程渲染模式中才会被创建（CullThreadPerCameraDrawThreadPerContext）。因此此处的 Renderer::cull 函数将被执行。

这样一来，场景的筛选工作将在每一帧当中仅执行一次，而绘制工作则交给 GC 线程来完成。是否会因此造成筛选和绘制数据之间的冲突呢？答案当然是否定的，还记得我们在第十七日和十八日中所介绍的吗：Renderer::cull 最后将向绘制队列_drawQueue 中添加一个已完成筛选的场景视图对象（SceneView），而 Renderer::draw 函数一开始就会尝试从这个队列中取出一个数据（takeFront 函数），并清空它在队列中的位置。因此，如果新的线程绘制工作提前到来的话，由于场景筛选的函数还没有把 SceneView 传入到_drawQueue 队列中，因此这次多余的绘制动作将自动宣告结束（事实上也会暂时阻塞这个“多事”的线程）。

那么，我们依然假设有三个条件不一的 GC 渲染线程，并得到相应的流程示意图如下。注意主进程中三个设备的筛选工作与各个设备线程中的绘制工作是首尾相连的，即，主进程中的筛选一旦结束，线程所控制的绘制就会马上开始；而由于有 swapReadyBarrier 交换缓存栅栏的存在，各个图形设备的交换缓存工作依然是统一执行的。



图中还应注意的是，由于 DrawThreadPerContext 模式不存在渲染启动和结束栅栏，因此主进程执行完场景的筛选之后就可以继续执行，进而开始新一帧的用户数据更新工作。而此时 GC 线程的绘制工作很有可能还没有完成，如果此时场景中某些需要绘制的数据有了改变，将会造成无法预料的事情发生，最严重的当然就是系统崩溃了。

幸好我们还有动态对象阻塞器_endDynamicDrawBlock 这个保护伞。在使用这一线程模式以及后面马上要介绍的 CullThreadPerCameraDrawThreadPerContext 模式时，不要忘记为场景中的变度对象设置 setDataVariance(Object::DYNAMIC)。

当前位置：osgViewer/ViewerBase.cpp 第 414 行，osgViewer::ViewerBase::startThreading()

下面我们即将介绍的 CullThreadPerCameraDrawThreadPerContext 模式是 OSG 目前所采用的四种线程模型的最后一种，目前来看它也是实现效率最高的。

CullThreadPerCameraDrawThreadPerContext 模式正如它的名字所说的那样：建立了多个摄像机线程，用于场景的筛选；同时建立了多个 GC 线程，用于场景的绘制。这种模式也提供了动态对象阻塞器_endDynamicDrawBlock，以免用户更新动作影响到场景的渲染工作。其中，各个 GC 线程的任务列表的建立，与 DrawThreadPerContext 模式并没有差异：

1、osg::RunOperations 任务负责场景的绘制。

2、swapReadyBarrier 任务的作用是在交换双缓存之前对所有 GC 线程执行一次同步。

3、swapOp 任务（404 行）用于执行绘制后的双缓存交换操作。

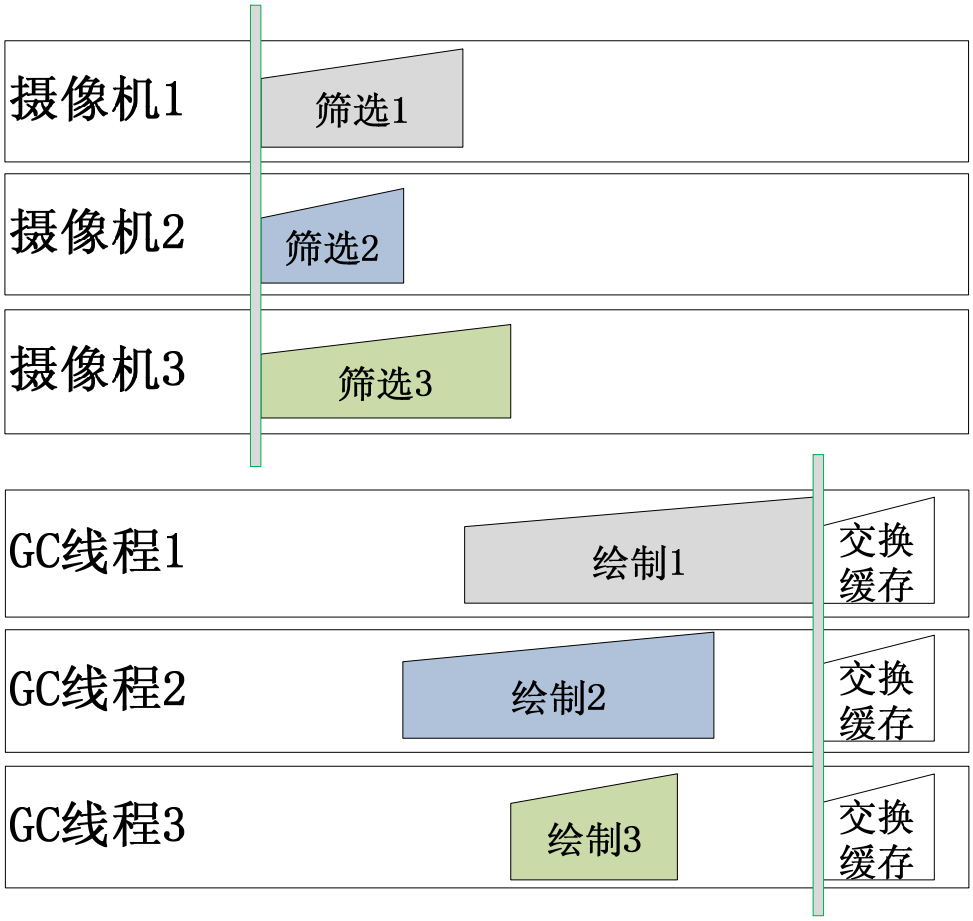
而建立摄像机线程的数目与场景中摄像机的数目相同。其任务列表如下：

1、_startRenderingBarrier，渲染启动栅栏在这里又起到了应有的作用，不过注意这种模

式下它应当改个名字，叫做“筛选启动栅栏”。其强度是场景摄像机的数目加一，因此在 renderingTraversals 函数中（ViewerBase.cpp，662 行）将会通过阻塞主线程，对场景的筛选线程（即各个摄像机）执行一次同步。

2、osgViewer::Renderer，它也可以作为一个操作任务存在的，当线程中执行到这个任务时，会转而执行 Renderer::operator()(osg::Object* object)函数，并在其中执行 Renderer::cull 函数，完成场景的筛选。

以上所有的任务都会在线程的运行过程中循环执行。那么我们假设有三个条件不一的 GC 绘制线程和三个摄像机筛选线程，并得到流程示意图如下。注意这里摄像机线程的筛选工作与 GC 设备线程中的绘制工作同样是首尾相连的，即，摄像机线程的筛选一旦结束，GC 线程的绘制就会马上开始。



这里我们还要简单地介绍一下 OSG 分配 CPU 的策略。对于各个 GC 线程（除了单线程之外的三种模式），将使用 OpenThreads::Thread::setProcessorAffinity 函数平均地安排到每个 CPU 上；如果还有摄像机线程的话，则按照与 GC 线程相同的做法，安置到最后一个 GC 线程之后的 CPU 上。

下图演示了一个四核 CPU 环境中，将三个 GC 线程与三个摄像机线程分配到各个 CPU 的分配方案：

当前位置：osgViewer/ViewerBase.cpp 第 593 行，osgViewer::ViewerBase::frame ()

本节内容事实上也是这篇教程的最后一个部分，完成了对于 startThreading 和 renderingTraversals 函数的解析之后，frame 函数也揭开了它的最后一缕面纱。不过随着仿真循环的执行，它依然会周而复始地循环下去……OSG 的强大，也许就是如此简单，但我们却不得不佩服其创造者丰富的知识水平和良好的系统架构，惊叹于其中种种的优化策略和线程操作技巧，并庆幸自己能够在这个日渐庞大的软件系统仍在飞速发展的时期加入到它的开发者的行列。

那么，幸运的我们又该做些什么呢？是日复一日地翘首盼望新的功能？是抱怨 OSG 又出现了这样以及那样的毛病？是“跪求”他人贡献出各式各样的中英文的教程和代码？我想答案应当都是否定的。我们理应致力于 OSG 在国内的发展与成熟，致力于它的传播，致力于扩展它的功能和文档，以及做出各种各样自己力所能及的贡献。

国内的虚拟仿真行业依然处于创立和发展的阶段，由于它的市场和利润丰厚，技术含量较高，希望投身其中的企业和个人自然不在少数，有志于成就事业，甚至梦想着成为 VR 行业的比尔·盖茨的朋友也是大有人在。但是，坦白来说，国内的技术水平及相关产品目前还缺少足够的说服力；学校等相关研究机构似乎也鲜有高水平的研究成果问世（至少我在清华大学所知所见的寥寥）；暂时更不存在如 OpenSceneGraph，Orge，Vega，Virtools 等广泛应用于实践，功能和稳定性都能够让人信服的中间件产品；各种高水平的中间件在求学者中间的普及程度恐怕也十分有限（OpenGL 类的读物倒是海量，可惜大多内容重复，粗制滥造者比比皆是）。这也许可以说是巨大的机遇；但如果十年之后，我们所见所处的依然是这种环境，依然眼巴巴地望着国际级游戏大作的精妙效果，用着好不容易盗版得来的“洋软件”的话，就不得不说是一种永远难以逾越的差距，是一种悲哀。

不要以为我是在做一番“愤世青年”的感慨。作为一名曾经的机械行业的学生，我曾亲耳聆听一位名声赫赫的老教师的感慨：中国的液压技术永远达不到国际水平，从业者已经彻底放弃对它的研究，而是专心地为了购买欧美的过时产品而费尽口舌，忍受着高昂价格和肆意的限制。大飞机的研究也濒临这种命运，还有数控机床，还有伺服电动机……是否不知几时，还会多了虚拟仿真业呢？

就像我在这篇《最长的一帧》的序言中所说的那样：我们的任务是一直挖掘下去，找出期待的抑或不曾期待的瑰宝。很高兴我们能一直走到这一步，也很高兴 OSG 的方方面面还未有个尽头。笔者在艰难地写出这篇晦涩文章的同时，一直为自己的大大小小的发现而惊喜着，甚至有了以后开发自己的渲染引擎的念头和信心，相信读者您也如此（或者已经走在我的前面了 ^_^）。是的，我们的所得也许并非多么渺小，恰恰相反，它可能是一个新的成功作品的开端，而它的作者，也许就是您。

MiniGUI 的成功曾经让我们中国的开源软件名扬世界（虽然如今的发展策略让它日渐闭塞起来）。那么，什么时候我们能拥有名扬世界的国产 3D 渲染引擎呢？最好就在明日，但愿就在不远的将来。

（全文完）

解读成果：

ViewerBase::frame。

悬疑列表：

无。