一、

任何程序都是有生命的，是生命就需要呼吸。例如普通的windows程序，当运行完main()函数后，就需要进入消息循环，来监听用户的各种操作，以便做出及时的回应。这样的每次循环就像生命的每次呼吸，来维持生命体征。

osg的程序不仅仅需要消息循环来监听用户的鼠标、键盘等操作，同时也得具备了渲染循环。当然随着我们的对osg的深入了解会发现，osg的事件监听和渲染循环是串行的。但是当我们把osg与MFC（QT）等结合时，相应UI上的鼠标，键盘事件的同时也要兼顾可能发生在osg中的效果，所以一般的osg程序起码需要两个并行的线程（例如osg与qt结合使用，为了保持足够灵敏的相应速度就需要把QTUI和osg渲染看成两种生命，分为两个线程）来维持它的正常运行。我们今天就是要解读osg程序赖以生存的每次呼吸。

首先我们得找到osg是用什么呼吸的，就想地球上的一般生物都是用鼻子呼吸，我们又大概得知道鼻子长在生物的那个位置。这样我们才可以开始我们的研究。当然我们肯定得有osg的源码，就像我们研究生物的呼吸先得有这种生物的身体。有了身体我们还得持续的观察一个有生命的生物，所以我们最先得到osg的可运行的程序就是example中的各种程序。其中大部分的程序main()函数的最后部分都是调用一下viewer.run()。所以我们可以有一个模糊的判断这一类的通过调用viewer类的run函数的程序，他的呼吸系统可能是通过run完成的。但是run()函数是一个单独的一行，按说他执行完毕以后程序就会结束了，所以我们有了新的判断osg的每一帧的调用的入口是在run()函数中的。这是osg程序存在的一种形式（或者叫独立运行模式）。Osg还有另一种存在形式，就是和各种UI混合使用，例如qt与osg结合使用，MFC与osg结合使用等等。我们可以从examples/osgviewerQt 的例子，可以根据上一个的思路，呼吸不是一次性的动作，是只要存活就会一直存在的。所以从osgviewerQt.cpp中根据以前的经验定位到timer (计时器),他每次timeout触发时调用的函数update()中一定包含了osg的每一帧的调用的入口。

根据上面两种osg的存活形式，可以进行进一步的确认，究竟哪里才是维持osg生命体征的位置。Viewer.run()函数(OSG Core/osgViewer/Viewer.cpp)最后会继续调用ViewerBase::run()函数(OSG Core/osgViewer/ViewerBase.cpp)，我们在ViewerBase::run()中继续耐心的寻找就会发现有一个特殊的函数frame()，为什么特殊呢？因为frame的英文的意思就是’帧’，而我们学渲染都知道’帧’代表屏幕上一幅画，这和osg库的本质就联系在了一起。Osg就是一个库，一个在计算机屏幕上作画的库。所以ViewerBase::frame()就是我们要找的osg中会呼吸的地方。同样我们在examples/osgviewerQt中也会发现，timer每到设定事件就会调用update()函数，而qt的update()函数在内部就会调用paintEvent()函数，我们在osgviewerQt.cpp的paintEvent()函数中也会发现osg::CompositeViewer的update函数，而osg::CompositeViewer继承自ViewerBase，所以最后也会定位到ViewerBase::frame()。这样我们就可以确定osg这类生物的呼吸的入口是ViewerBase::frame()函数。终于我们打开了通往新世界的大门，下一步就是经历轮回，看看osg这类生物是怎么生存的。

二

我们把ViewerBase::frame()比作osg这类生物的肺，首先我们先来大概的看一下‘肺’长什么样子，有哪几部分组成。在这之前得对一些固定的零件进行说明，例如\_done代表osg的viewer是否被删除释放内存；\_firstFrame代表是否是第一次进入frame函数。那么接下来我们会发现frame函数表面上组成结构非常简单，逻辑上也非常的清晰---先判断当前的viewer是否被删除，也就是判断是否died，如果已经died，那么肺的功能就不会进行。然后判断这个osg小孩是否刚刚出生，是的话就要执行一些初始化工作---嚎啕大哭。最后剩下的四个函数就是一个正常的osg生物的肺周而复始进行的工作。

我们一步步来，首先看看刚刚生下来的osg宝宝为什么会哭，对osg整体产生了什么样的影响。进入第一个函数viewerInit(),ViewerBase::ViewerInit是纯虚函数，代表他的实现由他的子类完成，上一节我们研究了osg生物两种存在形式时，就已经定义了目前正在进行工作的viewer是哪一个，所以我们直接到osgViewer::Viewer（是ViewerBase的子类）下看看他的ViewerInit函数。经过复杂的拆解工作，终于发现ViewerInit是在osgViewer::Viewer的头文件中定义并且实现的。并且它的功能非常简单就是简单的调用init()方法。而这个init方法又是在那个类里面定义的呢。Viewer类分别继承了osgViewer::ViewerBase以及osgViewer::View。Init既然在osgViewer::ViewerBase和osgViewer::Viewer中都没有记录，那么肯定是在osgViewer::View中定义和实现的。我们这里就有一个疑问Viewer和View到底有什么区别。从名字上我们有一个大概的区分Viewer视景器，View视图风景。简单的理解Viewer就是View的操作器，osgViewer::View就是持有场景中的一张景色。仔细的你们一定会发现osgViewer::View继承自osg::View以及osgGA::GUIActionAdapter. osgGA::GUIActionAdapter我们先不进行详解要不就跑偏太远了，先介绍osg::View. osg::View我们看作是场景中一系列相机的掌管者。好了。我们这些就先说到这里，赶紧趁着刚刚打开的osg的肺还有失去动力之前回去继续看看。

进入osgViewer::View::init()函数，这是一个新的身体零件，和研究frame()函数的思路一样，我们先认识它内部的固定零件:\_eventQueue代表空的osg内的事件队列—这个我们以后再介绍，\_cameraManipulator代表相机的操作方式—这个我们都会在进入frame()函数之前进行单独的指定。所以我们大体上也就大体了解了osgViewer::View初始化工作，就是创建一个名叫osgGA::GUIEventAdapter::FRAME的事件并放到\_eventQueue中，以及对\_cameraManipulator相机操作方式的初始化。不同的摄像机操作模式就会有不同的init函数，我们就对CameraManipulator::init()函数不进行介绍了--- 很简单。

这样我们就完成了对ViewerBase::frame()函数中第一个功能介绍完毕，总结一下。Viewer::viewerInit()函数就是完成了osg中事件队列的初始化以及相机操作器的初始化工作。当然这些工作之前，也就是进入frame函数之前必须先完成viewer以及相机的定义。但是osg库非常强大，即是你没有定义viewer以及camera他就会默认的按照约定的模式创建一组他们出来。

三、

那我们回到ViewerBase::frame函数中来，继续看看为什么osg生命刚刚出生的时候会大哭，除了初始化了eventQuene和cameraManipulator之外还对那些器官进行了初始化。在这之前我们先介绍一下上一节说到的osg的肢体或者器官但是没有展开介绍的。

osgGA::GUIEventAdapter，GUI事件适配器。它就是对所有平台windows linux mac平台上的鼠标、键盘、以及其他的窗口事件进行了封装，目的是使接口统一，用户在使用osg库的时候不用再自己区分平台，直接调用GUIEventAdapter就可以得到平台发过来的事件信息。特别是我们处理自定义的EventHandler事件处理器的时候重写handler (const osgGA::GUIActionAdapter &ea ,osgGA::GUIActionAdapter &aa)的时候一定会用到。

osgGA::GUIActionAdapter，这个内脏器官的用处比较多。我们慢慢的说。作用：定义GUIEventHandlers可以请求使用GUI操作系统的动作的抽象接口类。这些动作请求应该遵守用户所使用平台的GUI工具包的规范。例如当一个GUIEventHandler处理一个鼠标事件的时候，它希望去请求GUI,例如当一个模型被’抛出’---（使用TrackballManipulator，按住鼠标右键快速移动，然后松开右键类似抛出物体的动作）时，这个TrackballManipulator可能希望启动计时器，并重复调用，以持续刷新摄像机的位置和方向。但是，它无法做到这一点，因为它对它运行的窗口系统一无所知。得不到任何支持。相反，GUIEventHandler可以通过GUIActionAdapter发出自己的请求，然后使用这个GUI系统的viewer应该遵循这些请求。当然GUIActionAdapter的功能不只是这些，再例如我们上一节也看到了osg::Viewer::View的其中一个基类就是GUIActionAdapter，而osgViewer::View又是osgViewer::Viewer的基类，所以我们可以GUIActionAdapter强转成osgViewer::Viewer.这种使用方式一般也是在处理自定义的EventHandler事件处理器的时候重写handler (const osgGA::GUIActionAdapter &ea ,osgGA::GUIActionAdapter &aa)的时候会用到。

说了两个概念性的东西，有可能比较枯燥，但是技术类工作就是这样，每每在学习一种新的技术或者了解他背后的原理的时候，我们要耐下性子，一步一步来。只有把每一步都搞清楚弄明白我们在能继续前进不至于迷失了方向。

我们再此回到我们的解剖课中继续探究osg类物种，osgViewer::ViewerBase::frame()中的下一步就是判读viewer是否已经被实现了，如果没有那么就要实现这个viewer。所以我们要到osgViewer::Viewer::isRealized（）函数。主要功能就是这个从这个viewer中得到渲染的上下文，并且判断他们是否被实现。渲染上下文我们应该都很了解，如果您用过qt+osg一定会有这么一个过程，创建一个camera的过程需要指定操作系统的窗口属性，这个渲染的上下文就在这里指定。那我们就去osgViewer::Viewer::getContexts()下看看怎么来得到所有的GraphicsContext()。

osgViewer::Viewer::getContexts()。我们还是老规矩，看看这个新的动作中涉及到那些新的内脏，osg::GraphicsContext。就是图形设备对应的载体，或者说GraphicsContext是任意图形子系统的抽象接口，它提供了统一的图形设备处理函数，用来实现渲染结果和底层设备的交互。图形设备对象的主要工作是提供场景渲染结果的载体，这个载体可以显示缓存，进而绘制到一个图像窗口中，也可以是其他特殊的缓存对象，从而实现复杂的渲染和图像多次曝光等功能，创建一个图像设备不能简单地使用new运算符，因为GraphicContext类是一个不能被实例化的抽象类（这个体现在valid()等一大批纯虚函数上）；通常应当使用createContext()静态函数，自动根据当前的用户环境和特性参数traits，构建一个平台相关的图形设备对象。然后绑定到摄相应的像机上。

osg::ref\_ptr<osg::GraphicsContext> gc = osg::GraphicsContext::creteGraphicsContext(traits)

camera-> setGraphicContext(gc);

所以在osgViewer::Viewer::getContexts()中通过\_camera->getGraphicsContext()来得到图形设备对象。

osgViewer::Viewer::getContexts()。我们又遇到了新的不认识的小零件，osg::Slave代表了主相机下的一个跟随相机，以及视图矩阵。我们在前面已经介绍过了osg::View中定义了所有的相机，当场景中只有一个主相机时，那么它即是主导也是场景的渲染器，但是有了从相机，那么主相机将视图控制到场景，而从属相机负责实现场景的渲染。所以osg::Slave中也会包含osg::GraphicsContext，也要中所有的从相机getSlaves()中得到所有的GraphicsContext。并判断他们是否可用GraphicsContexts->vaild()。这样就介绍完了viewer->getContexts().

回到Viewer::isRealized()函数中，再往下就是确定这些得到的所有的图形设备是否已经初始化完成准备被使用。Osg::GraphicsContext::isRealized() const { return isRealizedImplementation();};也就继承自Osg::GraphicsContext的类或者说Osg::GraphicsContext封装的底层的设备接口类中，这些类一般都在osg::Viewer::api下定义。这些类的仔细介绍我们会稍后进行讲解。这样我们就完成了ViewerBase::frame()一呼一吸动作中的第二个拆解动作isReallized()介绍。

四、

当判断到viewer中没有一个graphicContext可用时，osg就会默认的进行一次对viewer的实现操作，这样可以保证osg以后可以安心的在屏幕上进行作画。那我们就来看看这个osgViewer::Viewer::realize()函数到底具备什么样神奇的功能。

osgViewer::Viewer::realize()的最要作用可以总结为激活设置窗口以及初始化关联线程。Viewer::getContexts()上一节以及进行了详细的介绍，就是得到所有相机上关联的图形设备器。当contexts为空时就代表不存在一个窗口set up，所以osg开启默认一个的视图。读取环境变量OSG\_CONFIG\_FILE，如果设置了OSG\_CONFIG\_FILE指向一个.view后缀的文件，可以创建该文件描述的窗口，格式如下：（在openscenegraph-data目录中的configuration目录下有.view类型的文件）

osgViewer::Viewer

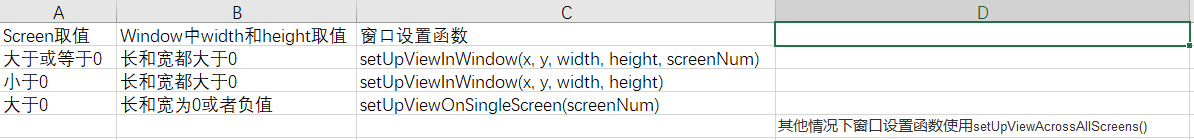
{

setUpViewInWindow 100 200 600 400 0

}

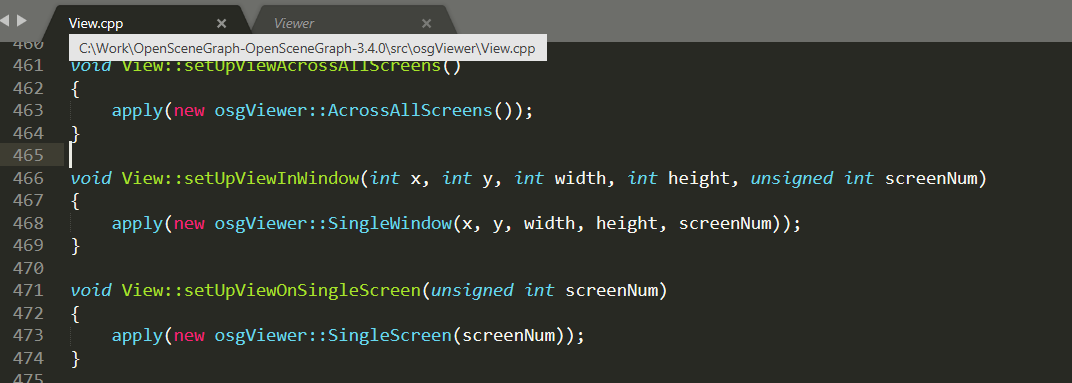
如果设置了OSG\_CONFIG\_FILE环境变量，但是环境变量所对应的文件解析有错，那么整个程序就会退出。

如果没有设置OSG\_CONFIG\_FILE环境变量，那么程序会检查 OSG\_SCREEN和OSG\_WINDOW这两个环境变量的值，OSG\_SCREEN对应窗口的个数（值是一个整型数）OSG\_WINDOW对应窗口的大小和位置，格式是（x, y, w, h) 分别是窗口左上角点坐标(x, y)以及窗口的长和宽(w, h)，具体来说是以下情况：

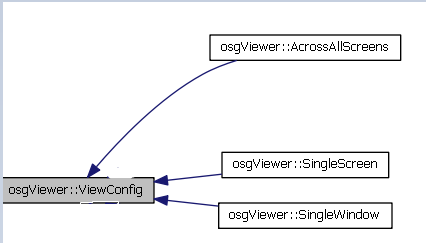


当窗口设置完成之后，osg会再次调用getContexts(contexts) 搜集目前可用的设备渲染上下文，如果这时候还没有窗口产生，也就是说设置窗口的各种方式都失败了，那么程序就会退出，如果成功，那么会针对已经常见的窗口进行一些设置，完成后续的操作

接下来我们先简单的介绍一个表格中三个函数



这些函数调用了osgViewer命名空间中的三个窗口配置类，它们的关系如下图所示：



基类osgViewer::ViewConfig提供了一个虚函数 virtual void configure (osgViewer::View &) const，在子类中通过实现该函数来配置渲染的窗口大小和位置。

这里我们只对一个类的configure函数进行讲解，其他两个请大家自行了解。

AcrossAllScreens::configure(osgViewer::View& view)函数，首先调用osg::GraphicsContext的静态函数getWindowSystemInterface用于获得系统的api接口，至于这个api接口是怎么被创建的，我们需要在src/osg/GraphicsContext.cpp中找到setWindowSytemInterface函数，这里就是进行系统api的设置，他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。由于现在实在ubuntu16.04系统上，所以cmake只会把GraphicsWindowX11.cpp进行编译，所以想知道怎么指定的系统api接口在GraphicsContext中，我们需要进入GraphicsWindowX11.cpp的2136行的WindowingSystemInterface 结构体，这个结构体在初始化的时候就会调用。用于设置系统api接口。还 有 注 意 那 个 紧 跟 着 结 构 体 的 全 局 变 量(GraphicsWindowX11.cpp,2159 行),这就是osg设置系统api的基本流程，如果想深入了解，请自行查看。

回到AcrossAllScreens::configure(osgViewer::View& view)函数

我们得到了系统api的指针，然后将尝试获取osg::DisplaySettings的指针，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。

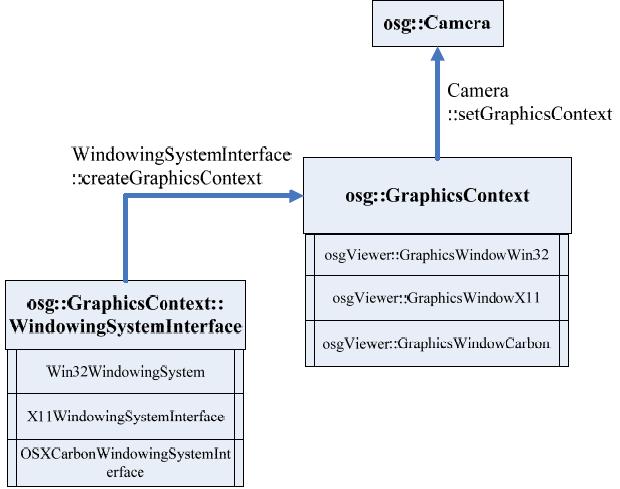
\_numMultiSamples:多重采样的子像素样本数,默认为 0。如果显示卡支持的话,打开多重采样可以大幅改善反走样(anti-aliasing)的效果。此外还有很多可以设置的类变量,如\_minimumNumberStencilBits(模板缓存的最小位数)等,其默认设置均在 osg::DisplaySettings::setDefaults 函数中完成,其中有些变量可能还没有作用。要注意的是,DisplaySettings 的作用仅仅是保存所有可能在系统显示中用到的数据,这个类本身并不会据此改变任何系统设置和渲染方式。

再此回到AcrossAllScreens::configure(osgViewer::View& view)函数，我们上一节总结了一下osg::DisplaySettings的作用，我们继续看看配置一个osg内置的screen需要哪些设置。通过相机得到视椎体的一些信息(包含:fovy俯仰角，aspectRatio纵横比，zNear近平面, zFar远平面，其中这里用到的属性是aspectRatio，主要是用来确定当osg默认的与图形设备中设置的screen数目的相同的从相机中每一个从相机的视口的大小)。然后再通过ScreenIdentifier平面初始化器得到screenNum,displayNum,hostName的值。其中如果screenNum或displayNum未定义则设定为-1，这三个变量的最主要的作用是给GraphicsContext中的某些属性赋值。当displayNum的值是1，也就相当于创建一个SingleScreen，所以直接可以调用昨天提到的SingleScreen的configure函数，创建一个单独的屏幕来渲染三维世界就可以了，但是当displayNum的值大于1时，就需要用到分屏技术，通过创建多个从相机，来得到场景中的某一部分的视图，最后拼接成一个大的三维场景的方式。所以else(src\osgViewer\config\AcrossAllScreens.cpp\ AcrossAllScreens::configure 函数的第50行之后都是在创建多个GraphicsContext并付给相应的从相机。)

五

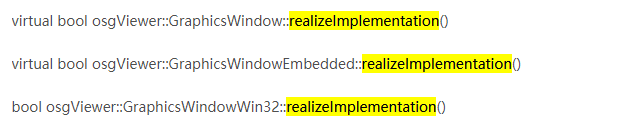
上一节我们对完成了对osg生物内部非常重要器官graphicsContext的初始化工作。这样就可保证我们场景中至少有一个graphicContext存在，不至于刚出生就面临夭折。我们根据上一节中osg代码的研究也就知道了，在我们正常使用osg时，是怎么完成对camera以及graphicContext的创建的了。

回到Viewer::realize()中我们继续向下看，现在我们对osg::DisplaySettings以及osg::GraphicsContext::WindowingSystemInterface，有了新的认识，我在这里再补充一张camera,graphicContext以及windowingSystemInterface的关系图有利于大家进一步了解osg的内部组成。



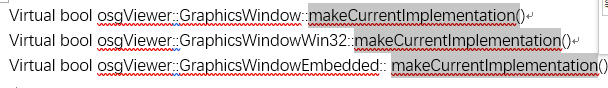
先遍历所有的GraphicsContext，然后判断是否设置了同步交换缓冲区（这一般是渲染的最后一步），这是osg提供的多机同步swapbuffer机制，他会默认调用内置的swapbuffer的回调函数（osg::SyncSwapBuffersCallback中，作用主要是等待client端的同步锁，实现多机同步执行swapbuffer）。如果developer想干预的话 可以调用 osg::GraphicsContext::setSwapCallback(SwapCallback\* rc)来设置自定义的缓存交换回调。自定义的回调必须调用GraphicsContext::swapBuffersImplementation()函数.

再根据所依据的平台(windows,linux,mac等)默认制定的，或者用户后期修改的最大纹理池和最大对象缓冲池的大小，进一步对各个graphicsContext的相应属性值进行设置。然后正式完成对graphicsContext的初始化定义，下一步就是通过调用gc.realize()来使graphicsContext处于可用的状态。Osg::GraphicsContext::realize()函数的实现都是在它的继承类中通过realizeImplementation()函数完成的。

特别是windows平台的实现，使用windows+opengl的同学对这一段GraphicsWindowWin32::realizeImplementation（）函数肯定非常了解，因为这里会涉及到很多windows平台特有的一些属性，就不做过多的介绍了，以后有机会我会再写一份opengl的入门教程，其中肯定会提到函数中涉及的东西。敬请期待。

当GraphicsContext可用了，就需要更新上下文gc->makeCurrent()。那我们就看看GraphicsContext::makeCurrent()完成了什么工作。

GraphicsContext::makeCurrent()首先判断opengl与osg是否是同一个线程，（使用qt5+osg3的同学一定遇到过osg的threadmode只能设置singlethread。其他三种threadmode都会报一个同样的错误，错误的原因就是这里。至于怎么完美的结合qt5与osg3，请移步到我的github下 <https://github.com/JimmieKJ/osgQTWidget> 有具体的实现细节。）。其实GraphicsContext::makeCurrent()的根本是通过调用子类的makeCurrentImplementation（）实现。



当我们移步到bool GraphicsWindowWin32::makeCurrentImplementation()同样会发现，这里和使用opengl的程序有很大的相同之处，其实就是把dc和rc进行绑定。当makeCurrentImplementation返回true的时候，就代表graphicsContext更新成功。然后就是opengl的思路，需要解绑hc和rc防止资源浪费，这就需要调用gc->releaseContext()，其实就是调用子类的releaseContextImplementation（）函数。

再次聚焦到realize函数上（/src/osgViewer/Viewer.cpp::realize()函数），\_incrementalCompileOperation，用于预编译GraphicContext，主要作用是，想在程序运行开始时就加在一个资源文件但是又不想或者没有到显示到界面的时机，则会用到这个预加载操作。具体的用法如下：

//从Viewer获取 osgUtil::IncrementalCompileOperation的指针：

osgUtil::IncrementalCompileOperation\* pIcompOperation = viewer.getIncrementalCompileOperation();//从Viewer获取 osgUtil::IncrementalCompileOperation的指针：

// 创建compileSet:

osg::ref\_ptr<osgUtil::IncrementalCompileOperation::CompileSet> compileSet = osgUtil::IncrementalCompileOperation::CompileSet(NODE,true);

//从CompileCompletedCallback派生新类，然后重写Completed函数，在内部隐藏节点：

//将 派生类 绑定到 compileSet。

compileSet->compileSet->\_compileCompletedCallback = newCompileCompletedCallback;

//设置 IncrementalCompileOperation 过期策略

pIcompOperation->setCompileAllTillFrameNumber(50);

再往下就是使鼠标聚焦到osg的绘制窗口上这个一个功能。

// initialize the global timer to be relative to the current time.

osg::Timer::instance()->setStartTick();

// pass on the start tick to all the associated event queues

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

// configure threading.

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（这个多线程问题我们以后再深入讨论）

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,)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 最后一个for循环看到的,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。

六、

我们用了两节的内容才堪堪讲解完ViewerBase::frame()函数中调用的realize()---Viewer:: realize()函数。我们简单的总结就是Viewer:: realize()主要是使GraphicsContext处于可用状态，并且启动相关的图形线程。

ViewerBase::frame()函数解读到这里，我们完成了osg生物第一次尝试呼吸所需要的所有器官的初始化工作。下面就真正的开始进入osg呼吸动作的研究了。也就意味着我们真是进入osg的仿真循环的研究当中。那我们就来看看osg呼吸的第一个动作advance()。

osgViewer::advance()函数的功能算是比较简单的，老规矩先介绍一下这个函数中遇到的新的成员变量。\_frameStamp：(osg::FrameStamp)是用来记录osg的帧数以及时钟校准，计数所用到的内置器官，这样可以精确的掌握osg的运行时间，有利于开发人员进行调优工作。在这里(advance())首先获得osg运行的上一帧时间(是在osg内部记录的时间不是真实世界的时间)，以及获得已经运行了多少帧了，并使记录加1(也就是记录目前所处的帧数)，再设置现在的相对运行的时间(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)。记录这些的目的就是有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。

上一段内容基本对advance介绍完成了，只剩下最后一个if (osg::Referenced::getDeleteHandler())判断。它的作用是用来将已经收集得到的所有的osg弃用的对象删除(osg::DeleteHandler::flush())。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。

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，你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。

六、

事件循环和渲染循环

终于到了我们嘴里经常念叨的事件循环、更新循环以及渲染循环了。首先我们来区分一下事件循环和渲染循环，他们两个首先是两个不同顺序执行的过程，我们有时候会用到任意node的updateCallback函数，这个就是在更新循环的时候遍历所有的node来调用updateCallback函数的；而事件循环是与用户操作和操作系统事件想关联的，以及调用我们设置的事件回调(EventCallback)函数。而事件循环函数(viewer::eventTraversal())是我们现在要探究的内容。

代码

进入这个函数，我们发现前几行都是我们以前介绍过的osg器官。首先记录了事件循环的开始时间，这样做的目的是：与这个函数最后记录的时间进行比较，然后记录在\_stats记录器中，这样可以帮助开发者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，以便对整个程序进行调优工作。然后得到所有的GraphicsContext，保存到contexts中，当contexts为空时，意味着没有最终的画布，osg会结束运行，通过设置\_done=true;

然后的主要工作是：事件循环会得到已经发生的所有事件，并进行一定的筛选工作，最后全部都传给各自的事件处理器。所以我们首先对其中一些新成员进行简单的介绍：

1. eventState事件队列的目前的状态事件，eventState的设置是通过osgGA::EventQuene::setCurrentEventState函数进行设置的。
2. \_eventSources 实在osgViewer::View下的成员变量，通过View::addDevice()函数来添加新的设备，他的主要作用就是在每一个帧的事件循环中便利所有的设备，然后得到通过Device :: getEventQueue收集生成的所有的事件。

搬掉了几块绊脚石，那么现在我们就可以继续前行了。osgViewer::viewer::eventTraversal()中第一个for循环的目的就是遍历所有的设备所发生的事件，并保存到viewer::events中。这些设备就包含鼠标，键盘等发生的事件。

然后再一个for循环，得到所有的GraphicsContext中的event并插入到eventQuene链表中，也就是诸如鼠标的移动，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。然后我们再会对窗口上发生的点击，释放，拖拽，双击和移动事件中的鼠标坐标进行统一的投影变换，使鼠标坐标重新投影到当前视图的坐标系中。

现在我们就主要来讲解一下鼠标坐标到视图坐标系的转换。

当鼠标只是进行单点操作，或者当然的事件的GraphicsContext不是主GraphicsContext时，需要调用generatePointerData函数来对鼠标的坐标进行转换。Viewer::generatePointerData()函数中，在这里我们要普及一点知识osg或者说opengl中屏幕坐标的原点在左下角，而windows的坐标原点在右上角，所以在这个函数中我们首先需要把判断我们所使用的平台的原点和osg的原点是否相同，如果不同则需要把鼠标坐标的y取反一下(gw->getTraits()->height - y)。然后把新的到的坐标点设置回事件信息中，并把Y轴模式改为向上增长(Y\_INCREASING\_UPWARDS).然后我们得到此时这个GraphicsContext下的所有的正在使用到的相机，并选出目前鼠标事件中的x,y所处在那几个相机的视口中，得到这几个相机作为活动相机。然后根据相机的先后渲染顺序进行排序，因为我们最后渲染的肯定会覆盖先前的，所以我们只需把鼠标坐标投影到最后渲染的相机的视图上就可以了。因为视口的坐标都是以0到1间的数字来表示的，所以鼠标的坐标通过一定的线性变换就可以变换到视口坐标系内

event.addPointerData(new osgGA::PointerData(camera, (x-viewport->x())/viewport->width()\*2.0f-1.0f, -1.0, 1.0,

(y-viewport->y())/viewport->height()\*2.0f-1.0f, -1.0, 1.0));

当然上面所说的适口坐标肯定是主摄像机的视口坐标，如果是目前鼠标是在从相机中移动的，那么再转换到主摄像机坐标系中。这个过程大概可以理解成这样，我们首先要把鼠标的坐标按照从相机的MVP矩阵转换到世界坐标系中，再根据主相机的MVP矩阵把刚刚得到的世界坐标转换到主摄像机的视口中，最后完成了从相机到主相机的坐标转换。

七、

VPM矩阵

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）。

CheckEvent与takeEvents

上一节我们遗漏了GraphicsWindowWin32::checkEvents和osgGA::EventQueue::takeEvents的关系。我们现在来讲解一下。先看一下checkEvents函数，这个函数的内容对于熟悉 Win32 SDK 编程的朋友一定非常熟悉，其中的TranslateMessage，DispatchMessage都是windows的消息传递函数，而它们的工作就是：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。而这个函数的作用是把Win32 SDK 编程中常见的窗口消息(WM\_\*)转化并传递给osgGA::EventQueue 消息队列。而osgGA::EventQueue 消息队列通过takeEvents得到所有的windows窗口消息，并进行处理，以及清空EventQueue。

回到osgViewer:: Viewer::eventTraversal()中，我们继续向下else也就是事件中的鼠标位置多于两个就会调用reprojectPointerData函数，它也是用来把鼠标从window屏幕坐标转换到主相机视口内坐标，和上一节内容基本相同。大家可以参照上一节内容进行理解。

模模糊糊朦朦胧胧，我们也算是跳出了处理所有事件中鼠标坐标的for循环。我们只能继续向下前行。我们又遇到了一个for循环，这个for循环简单来说就是处理当窗口关闭消息osgGA::GUIEventAdapter::CLOSE\_WINDOW发生时，osg会做什么样的工作，使其更加体面的离开。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，然后关闭窗口，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。

再往下我们也要针对目前帧的状态新建一个帧事件(也就是每一帧都会调用的事件)，并添加到事件队列\_evnetQuene中，然后同样得把这个帧事件中的鼠标坐标转化到主相机的视口坐标。再遍历一遍windows消息事件，添加到events中，并清空eventQuene队列。这样我们的events中就把所有来自图形窗口和视景器的事件都添加到一个 std::list 链表(event)当中, 下一步我们可以统一处理这些交互事件了.

八、

那我们就开始处理这些事件中得到的所有的交互事件，首先我们要判断这些事件是否包含osg的退出事件，那什么情况下会触发这个退出事件呢？如果您运行过osg中example中的小例子的，聪明的你一定就会发现当按下esc时就会退出osg。所以osg中默认的退出事件就是由esc触发的。当然我们也可以通过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。

我们还是先来看一下其中一些新认识的成员。\_eventVisitor:, 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。在事件循环中主要是的访问器主要是通过setEventCallback来设置Drawable对象中的事件回调的。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。

那我们回到osgViewer::eventTraversal函数中，首先我们需要设置当前的处于那一帧中，并进行统计，下一步就是我们要遍历所有事件队列中的事件，并放到eventVisitor中。，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，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。

然后在遍历场景节点并执行其事件回调之后，在遍历所有的相机操作中的handler函数。这样就基本完成了事件循环的遍历。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。

九、

前几天我们大体上介绍完成了osg的事件循环的介绍，总结一下osg的时间循环主要就是得到平台(windows)的所有消息，并遍历所有的node的eventCallback，并对他们进行处理。接下来我们就要进入osg的另一个维持生命的循环---更新循环。

OSG 更新循环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不同之处在于：事件循环是在当一个用户交互动作或系统事件产生时，每个节点（以及 Drawable 对象）的事件回调才会被调用一次；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。

那我们就开始进入osgViewer::Viewer::updateTraversal()，updateTraversal和eventTraversal一样首先都要定义目前处在的时间以及帧数，并进行记录，这样有利于进行统计分析。下面我们就要进入osgViewer::Viewer::updateTraversal()里最重要的函数osgViewer::Scene::updateSceneGraph()函数。

这个函数中我们先介绍一下它的主体功能，再去介绍这里遇到的一些新的概念。主要功能： 1、使用DatabasePager::updateSceneGraph函数更新场景的分页数据库，异步处理在分页数据库处理线程中。

2、ImagePager::updateSceneGraph函数， 更新场景的分页图像库，异步处理在分页数据库处理线程中。

3、设置图片请求的处理器。

我们先介绍一下DatabasePager和ImagePager

DatabasePager：分页数据库。在大型三维场景中采用数据分页的方式进行动态调度。这里“分页”的意思是随着视口范围的变化，场景只加载和渲染当前视口范围内数据，并将离开视口范围内的数据清除内存（可以设定不同的数据卸载策略），不再渲染。保证内存中只有有限的数据量，场景的每一帧也只有有限的数据被送到图形渲染管道，从而提高渲染性能。

ImagePager: 分页图像库。查看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。

DatabasePager和ImagePager都会用到独立的线程进行他们自己的工作。我们想要进入读懂他们代码的内容，首先我们得具备openThread的基本知识。

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。 OpenThreads 库包含了以下几个主要的线程处理类： 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。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。

八、

在前几节中，我自己觉得讲的比较粗糙，因为实在是时间上不是很充足，今天我想弥补一下，希望不是亡羊补牢。我们在osgViewer::Viewer::eventTraversal()函数中经常看到这么两个函数：camera->accept(\*\_eventVisitor);以及getSceneData()->accept(\*\_eventVisitor)；今天我们第一步就是要介绍一下这两个函数。首先我们通过查找camera和getSceneData()的定义可以发现其实他们都是继承自osg::Node类，所以我们就找到了accept函数的定义。

Node::accept(NodeVisitor& nv)

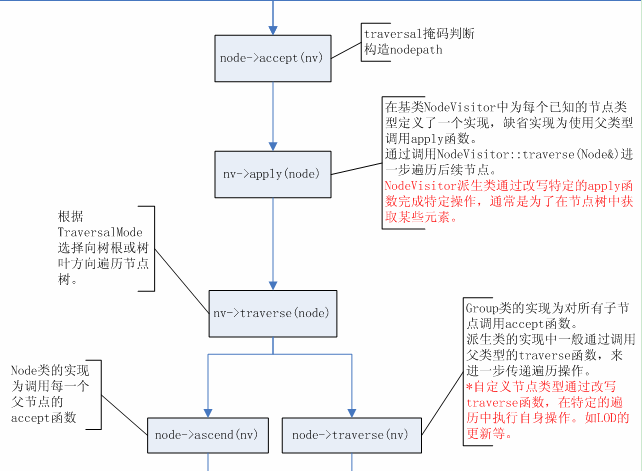
在介绍这两个函数之前我们需要先了解NodeVisitor的作用：作为访问者在osg :: Nodes上进行类型安全的操作。 基于GOF的访客模式。 NodeVisitor对于保护场景图中的节点类型安全操作非常有用（根据访问者模式），并添加了对可选场景图遍历的支持，以允许操作立即应用于整个场景。 Visitor模式使用双重调度技术作为调用NodeVisitor的相应apply（..）方法的机制。 要使用此功能，必须使用在每个Node子类中扩展的Node :: accept（NodeVisitor），而不是直接应用NodeVisitor。 所以使用root-> accept（myVisitor）; 而不是myVisitor.apply（\* root）。 后一种方法将绕过双重调度，并且有可能调用错误的NodeVisitor :: apply（..）方法。

这个函数看上去很简单，就是一个if判断，一共4句代码。但是其内部的乾坤非常的庞大。我们一步一步的分解他。首先我们进入

第一个函数if语句中的NodeVisitor:: validNodeMask()：主要作用是用来判断是否对这个node及其他的子节点进行操作。getTraversalMask()函数是用来得到NodeVisitor :: \_ traversalMask，而 getNodeMaskOverride()得到\_nodeMaskOverride和node.getNodeMask()得到Node：：\_nodeMask。也就是当如果NodeVisitor :: \_ traversalMask为0则将关闭所有节点的所有操作。将\_traversalMask和\_nodeMaskOverride设置为0xffffffff将允许访问者在所有节点上工作，而不管他们自己的Node :: \_ nodeMask状态。

第二个函数void NodeVisitor::pushOntoNodePath(Node\* node)，作用是用来把node节点添加到nodevisitor可访问的nodepath的队列的尾部，当\_traversalMode!=TRAVERSE\_PARENTS。而node不是父节点(也就是叶子节点时)就把node节点添加到nodevisitor可访问的nodepath的队列的头部。

第三个函数nv.apply(\*this);apply函数在nodeVisitor中是一个纯虚函数，他分别在CollectCompileCosts，CollectDrawCosts以及CollectParentPaths三个子类中实现。虽然三个子类前期的处理工作有所不同但是最后都调用了NodeVisitor::traverse(node)方法。在这个方法中会判断node是否在场景中作为父节点出现，如果是叶子节点则调用子节点中实现的traverse方法,来访问自定义的方法。如果是父节点则调用Node:: ascend方法，继续循环这个父节点下的所有子节点，直到是叶子节点调用traverse方法,来访问自定义的方法。



九、

DatabasePager类，也就是常说的数据库分页技术，简单来说，就是在进行数据库查找时，有可能满足条件的数据很多，为了提高相应速度我们进行数据查找时进行分页查找与显示，当点击下一页时才会进行下一次的查找动作。Osg就是把这个原理应用在了三维大场景中，当我们需要加载一个城市级数据时，因为数据量非常大，而我们感兴趣的范围在莫一时刻一般都是很小的一部分，所以我们也可以应用数据分页的技术，对大数据进行”分页”查找并显示。

在osg中DatabasePager执行的动作只有一步，就是在每一帧的更新循环中使用updateSceneGraph函数，把过期的数据卸载掉。并将新加入的数据载入到当前scene中。这些数据都是在用户视野中显示的。所以我们就把用户的视野比作”页面”。我们都知道数据的加载都是非常耗时的操作，更何况是三维模型这种大体量的数据，所以osg把DatabasePager的操作放到了另一个独立于渲染线程之外的线程上进行。

那我们就进入osgDB::DatabasePager中查看这个特殊的线程都进行了什么工作。

DatabasePager::DatabaseThread::run()。我们在上面一节中介绍过openThread的用法。在进入DatabaseThread::run()之前我们先对DatabasePager中的一些成员变量进行简单的介绍：

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）。

好，那我们就可以进入DatabasePager::DatabaseThread::run()内部进行解剖了。首先osg会对读取数据的模式按照本地数据和网络数据进行划分。划分的目的是osg会根据不同类型创建不同的线程进行处理，并且网络数据还可以设置缓存路劲(OSG\_FILE\_CACHE)缓存到本地。

然后会对线程安全的队列read\_queue上锁，以便在判断过期的数据(childrenToDeleteList)进行清空操作。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）。在DatabasePager::RequestQueue::takeFirst函数中遍历所有的请求列表(\_requestList)所有数据并判断要加载的数据是否过期，这样就可以得到要加载的数据。

再往下看，当databaseRequest->\_loadOptions是url数据时并且指定了OSG\_FILE\_CACHE的路径，那么osg就需要进行缓存工作，然后把databaseRequest 保存到将要合并的列表中(\_pager->\_dataToMergeList->addNoLock(databaseRequest.get());),再剩下的步骤就是对数据进行编译以及合并到scene中。

十、

我们接着昨天的继续，昨天主要是讲解了DatabasePager类中的特定的成员变量以及run函数的第一部分，对所要请求加载的数据按照是否是网络数据进行分类加载模式。今天我们就看看数据是怎们加载到osg环境中的。

还是在DatabasePager::DatabaseThread::run()函数中，首先我们保证了databaseRequest是线程安全的，因为run()函数是创建一个唯一的写入\_loadedModel指针的线程。首先还是先依据前面的数据来源是否为网络来判断readFromFileCache是否为true来区分加载方式。我们进入从缓存中读取文件的方法ReaderWriter::ReadResult FileCache::readNode()，就会发现这个方法就是判断当前文件是为缓存文件，然后调用Registry::instance()->readNode()。而这个Registry::instance()->readNode()函数也就是当readFromFileCache为false(不是网络文件)时在DatabasePager::DatabaseThread::run()函数中调用的是同一个函数。所以osg加载数据文件的原理就在这。那我们就深入探究一下osg到底是怎么加载模型文件的。

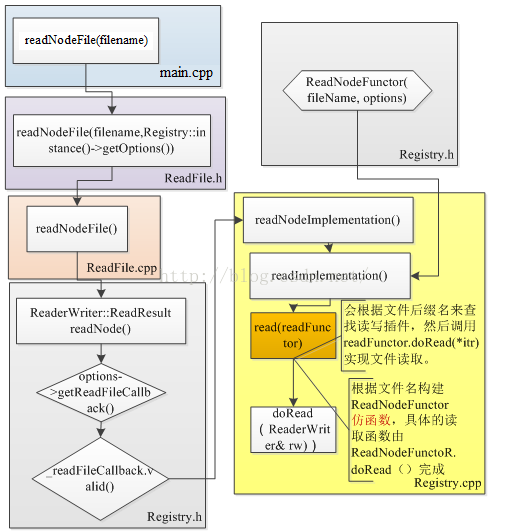
Registry

首先我们先介绍一下Registry::instance()(单例模式—以后会统一的讲解一下osg中的设计模式)。这个Registry即是osg的插件注册的管理器。也就是说我们使用osg加载数据文件的时候，osg是通过某一种插件来进行加载的。我们通过观察osgDB::Registry类的构造函数就会发现osg是默认自动会加载一部分插件到环境中，为什么不全部加载呢，因为osg的插件非常的庞大大概有121种，如果osg启动的时候就要预加载这么多的插件就会导致osg的启动速度非常的缓慢，所以osg采用了职责链的设计模式以加载尽量少的插件。我们就来介绍一个osg是如果查找适合目前功能的插件的。也就是Registry::LoadStatus Registry::loadLibrary(const std::string& fileName)

1. osg会先搜索已经注册的插件列表\_dlList，判断插件是否已经被加载到环境中，如果osg找到了符合次格式的插件，getLibraryItr(fileName)
2. 如果osg没有在\_ dlList中找到所需插件，那么osg就会根据文件的名称来去plugins文件夹下寻找(osgDB\_+此文件的后缀 )同名的插件，load到环境中。DynamicLibrary\* DynamicLibrary::loadLibrary(const std::string& libraryName)
3. 如果还是没有找到相应的插件，就代表此文件I/O的操作再次失败，OSG将返回失败信息。

注:通过查看 Registry ::addFileExtensionAlias成员变量，就可以知道我们平时加载的模型文件对应的插件名称。他们会被osg保存成一个map(\_extAliasMap。)

而我们的osg读取文件的操作就是要先要进行插件的选择预加载，然后通过插件类集成的load函数来进行文件的读取加载操作。我们可以把读取过程总结为下面一张图片，请大家仔细理解



好那我们返回到DatabasePager::DatabaseThread::run()函数中，读取完文件后，osg会根据是否需要缓存，以进行得到的loadedModel缓存到内存中。然后下一步就是找到所有可编译的渲染对象，进行编译。

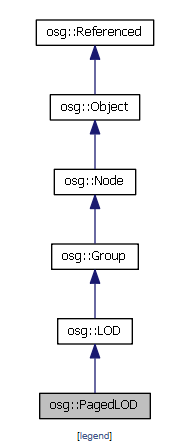
十一、

我们用了前面4节才刚刚算是完成对DatabasePager::DatabaseThread::run()函数的探究，也就是了解了osg究竟是怎么完成对数据的加载的。那么我们现在要回到DatabasePager::updateSceneGraph的工作中，它是在osgViewer::Viewer:: updateTraversal（）函数中遇到的\_scene->updateSceneGraph(\*\_updateVisitor);中被调用的。

1. DatabasePager::removeExpiredSubgraphs()函数的主要作用是通过目前正在使用的PagedLOD节点移除到指定定时间时间以来未访问过(过期的)的子节点。 注意，这段操作应该只从更新循环中调用。探究这个DatabasePager::removeExpiredSubgraphs()函数的内容，发现函数的开头还是对这个删除过程的记录，最后保存到osg的log中。遇到第一个if语句就是跳过第一帧，因为这个时候没有任何过期的数据。记录一下当前\_ activePagedLODList的大小numPagedLODs，当numPagedLODs没有达到最大lod数(\_targetMaximumNumberOfPageLOD)就不用进行清理工作。我们要记录超过的我们约定的最大的\_targetMaximumNumberOfPageLOD，的数量，这样我们就知道删除多少过期的lodpage数量（numToPrune），然后我们就是\_activePagedLODList->removeExpiredChildren()来便利现在所有的pagelod，从存在时间最长的也就是\_activePagedLODList最先加入的成员开始，删除子节点，直到numToPrune为0为止。这样我们就完成了对过期的pageLod的删除工作。
2. DatabasePager::addLoadedDataToSceneGraph: 向场景图中添加新的数据。还记得我们在DatabasePager::run函数中介绍到\_dataToMergeList吗？这个时候就要排上用场了。函数的开头依然是要对下面的添加数据的操作进行记录。把dataToMergeList中的数据转移到localFileLoadedList中，遍历localFileLoadedList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。然后用来判断node是否为osg::PagedLOD或者osg::ProxyNode，也就是说DatabasePager只会在指定为osg::PagedLOD或者osg::ProxyNode才会启用。然后就是得到DatabaseRequest对象使用getDatabaseRequest(plod->getNumChildren())或者是 proxyNode->getDatabaseRequest(proxyNode->getNumChildren())函数，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。

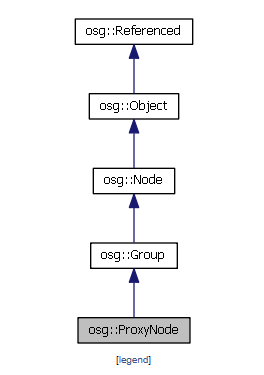
根据上面我们提到的，DatabasePager只有在使用 osg::PagedLOD 和 osg::ProxyNode 节点的时候才会有用。所以我们需要了解一下osg::PagedLOD和 osg::ProxyNode。

osg::PagedLOD我们先看一下类结构



Osg::Lod(level of detail)的意思就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osg::Lod节点可以其内部包含的子节点内容很多，这个时候无论加载哪一个层级都会消耗大量的计算机资源，为了解决这个问题，osg提出了pageLod这么一个类，目的就是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。

Osg::ProxyNode我们还是先看看他的继承情况，再功能分析



ProxyNode就是代理节点的意思，也就是他可以看作一个node的另一个名称，这样我们在特定的时候加载模型时，就不需要进行查找工作，直接对ProxyNode进行操作就可以了。例如：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：

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）。

这样我们就大概讲述完成了DataPager的使用时机，以及完成了对\_scene->updateSceneGraph(\*\_updateVisitor)；函数的主要功能的讲解。

十二、

当然细心的你会发现，\_scene->updateSceneGraph(\*\_updateVisitor)中还有一个imagePager::UpdateSceneGraph()还没有进行讲解，这是因为imagePager和DatabasePager是可以对比这理解的，这里imagePager的主要功能就是加载纹理图片文件。但是imagePager只是负责在另一个线程中加载图片，而没有databasePager的分页功能以及去掉过期数据。这样我们就真的可以重新回到Viewer::updateTraversal()中，继续向下进行了。

1. 大家应该还记得前面提到的SharedStateManager，用于node间储存共享的属性或状态。我们将要遇到的下一个if就是要遍历这个SharedStateManager所有储存的共享状态，删除其中不再被引用到的。
2. 更新又一次被外部引用的在缓存中的对象的时间戳，以保证他不会被清除掉。然后就是清除掉已经过期的在缓存中的对象
3. \_updateOperations代表更新任务的队列，然后取得任务队列（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。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）
4. 把已经编译好的子图进行合并。然后更新所有从相机自己的子图。调用任何相机更新回调，但只遍历那个回调，不要遍历其子图，将其留给场景更新遍历。
5. 使用相机操作器更新相机的位置。然后就是调用所有从相机的更新回调函数。并记录更新循环中的所有的信息。

现在我们总有算是完成了漫长的 updateTraversal 函数之旅。我们明天开始更艰巨的任务osgViewer:: ViewerBase::renderingTraversals()。

十三

前面我们基本上已经完成对ViewerBase::frame()函数的探究，只剩下renderingTraversals()渲染遍历的探究，虽然就剩下了一个函数，但是这却是最重要的，不可少的一个步骤。他主要是完成对场景的筛选和绘制工作，以及很多线程的调度和同步工作也是在这个函数中完成的。前面的几天我们主要是讲解了完成三个主要的遍历函数，我们来总结一下。

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

所以我们总结上面的三个动作主要是，更新用户数据， 负责场景对象的运动和管理等等；而这里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）。那么今天我们主要是讲解renderingTraversals()函数。

因为这里我们会涉及到OSG 目前 提供的四种线程模型。所以我们先来简单的介绍一下他们。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 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。

今天只是对renderingTraversals(); // 场景的渲染遍历工作，开一个小头，因为它的内容比较的丰富，所以我需要一些时间梳理一下思路。

十四

那么今天我们就正式进入osg整个呼吸动作之中最复杂的一个动作，ViewerBase::renderingTraversals()，我们先介绍renderingTraversals的开头的简单的几步操作。

1. 这个函数中先遍历了所有的view中的相机节点分别取得他们的位置Translation以及姿态Rotation,并保存到osg内置的log系统中。
2. 得到所有的渲染上下文contexts，然后使用 ViewerBase::checkWindowStatus 检查是否存在有效的渲染上下文，没有的话，需要使用 ViewerBase::stopThreading 释放每个camera对应的渲染器以及停止所有的图形线程和相机线程的运行。
3. 记录渲染循环开始的时间，最后会相应的进行统计，此次渲染循环进行了多久，方便开发人员调试。
4. 通过getViewerFrameStamp()得到记录了仿真循环运行的参考时间，总时间和总帧数的类osg::FrameStamp中的变量\_frameStamp。所以你们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，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，或不需要对于记录这些信息，简单地 将其置为 NULL。
5. 然后就是遍历所有的view，通过他们来得到当然场景的根节点root以及视图的状态信息stats。然后通过statsVisitor类来遍历场景中所有的节点信息，并区分他们按照osg中节点的类型来区分和统计他们的信息保存到stats中。
6. 再就是得到所有的scene，并且遍历他们，分别得到这个场景所管理的dataPager以及imagerPager类，向数据库线程发出信号，表示已为新帧开始更新，剔除和绘制。 注意，这是由应用程序调用的，这样当CPU忙于主渲染线程时，数据库寻呼机可以进入休眠状态。并且计算得到整个场景的bounding box。
7. 然后判断ViewerBase::\_endDynamicDrawBlock是否已经结束所有的动态对象绘制操作，，并重置endDynamicDrawBlock。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。
8. 阻塞渲染线程。

十五、

我们继续renderingTraversals()的探究。我们接着上一节的”阻塞渲染线程”后就要遍历所有摄像机的渲染器（Renderer），执行 Renderer::cull 场景筛选的操作。

我们在renderingTraversals()中总是遇到这个Renderer类，我们就先补充一下他的介绍。

osgViewer::Renderer类为摄像机渲染场景的工作提供了一个公有接口。通常是在View::setCamera(osg::Camera\* camera)函数中进行默认绑定的。

代码

我们既然找到了设置Renderer的时机，那么我们来进入Renderer类，先看看他的初始化做了些什么事情。

我们先要介绍一下这里遇到的新的osg内部变量：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；但是这个类已经不建议使用了，现在仅仅由 OSG 系统内部加以调用，也就是这个渲染器（Renderer）负责调用场景视图（SceneView）各种功能。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。再往下就是得到camera中的viewer，显示设置(ds)，光照信息，以及（自增式合并渲染操作）osgUtil::IncrementalCompileOperation等分别设置到两个 SceneView 对象（Renderer::\_sceneView[2]）中。已经对两个 SceneView 对象设定远近裁剪遍历。

我们介绍完了Renderer类的初始化，就可以开始对Renderer::cull函数的探究了。

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）。

3、更新场景视图（SceneView）的融合距离（Fusion Distance）。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、就是从state中得到FramStep的对象，然后就可以记录这一次cull裁剪的开始时间和所在的帧数。以及更新筛选设置（CullSettings）。

5、开始cull操作。

6、记录场景筛选所耗费的时间，并保存到统计器（osg::Stats）中。

7、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

具体cull裁剪的操作介绍，我们得先对SceneView 的深入学习。根据指导书籍最长一帧的介绍，我们先对void ViewerBase::renderingTraversals()的下一个步骤，遍历所有的GraphicsContext，然后执行GraphicsContext::runOperations函数。所以下一节我们先研究一下GraphicsContext::runOperations函数。

十六、

osg::GraphicsContext::runOperations()。我们先来看一下这个函数的执行过程。

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。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）。
3. 遍历 GraphicsContext::\_operations 队列中的各个 Operation 对象，判断operation对象是否会在后续的应用操作中进行使用，如果没有指定其在后续的应用操作中进行使用则在执行其 operator()操作后从\_operations 队列中清空。这里的 osg::Operation 类就是我们上一章讲到的osg::Renderer类，osg::Renderer继承自osg::GraphicsOperation，所以这里就是执行osg::Renderer的operator()操作，如果还定义了其他的继承自osg::GraphicsOperation的类，那么他的operator()操作也是在这里被调用的。
4. 我们到void Renderer::operator () (osg::GraphicsContext\* /\*context\*/)函数下看看这里到底进行了什么操作。（根据 Renderer::\_graphicsThreadDoesCull 变量的值）来区分，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。

总结一下osg::GraphicsContext::runOperations()，其实就是一个在调用osg::Operation 类的operation()之前进行的一次筛选工作。下一步我们就是来介绍一下Renderer::draw()进行了什么样的操作。

十七、

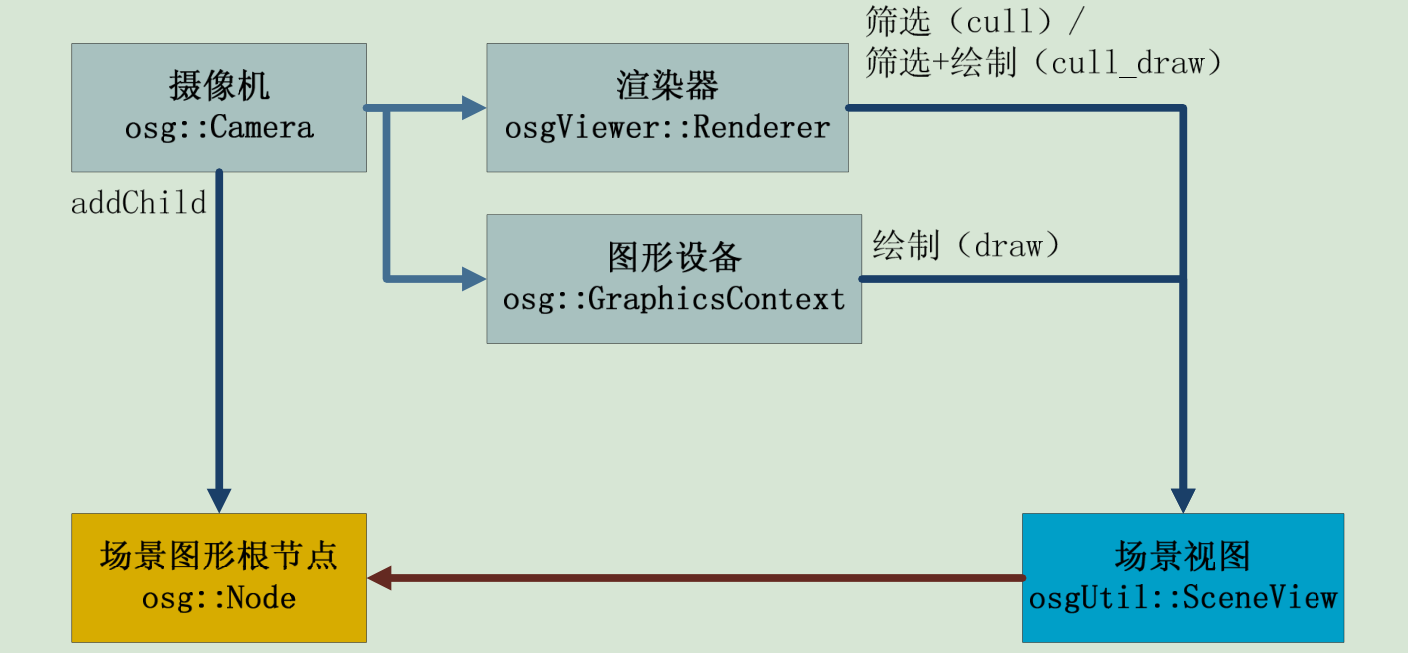
我们今天进入上一节的遗留问题Renderer::draw()的探究。

1. 从\_drawQueue中取出其中一个sceneView对象。SceneView是对scene和view类的封装，通过他可以方便的访问到scene或者view中的成员，以及最主要的作用是封装了更新，挑选，和绘制遍历，但是并不启用DatabasePager。
2. sceneView->collateReferencesToDependentCameras();得到所有的sceneView依赖的相机的引用，因为我们正在运行与主线程并行的绘制线程，所以取消引用的camera有可能还在被此渲染线程使用，所以为了防止这种情况，我们将引用所有这些Camera并且一旦我们清除了这些引用那么就完成了整个渲染调度。
3. \_compileOnNextDraw，它代表接下来的渲染是否已经进行编译，如果为true，则进入Renderer::compile()函数，首先把compileOnNextDraw设置为false，然后我们从sceneView中得到scene的根节点sceneView->getSceneData()，遍历所有的场景中的节点，把GLObjects的节点的状态保存到状态树中。
4. 执行 Renderer::initialize，初始化 Renderer 绘制所需的基本变量。
5. 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，这个类的作用就是阻塞线程的执行，并且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）的 completed 函数。在这里completed函数的主要作用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。
6. 执行 OpenGLQuerySupport::checkQuery 函数，判断是否可以使用 OpenGL 查询对象（query objects）。
7. 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
8. 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！虽然osg在osg::Util::SceneView的头文件中指出，这个sceneView类已经被废除了，存在的意义就是为了向前兼容。但是经过我们的探究发现，sceneView才是进行场景的筛选以及绘制工作的主要类，而渲染器类 Renderer 只是一个更为方便和直观的公用接口而已。SceneView::draw 函数就是用来绘制经过cull裁剪以后得到的”可绘制盒”内的所有物体。而我们还没有进行cull函数的介绍，所以这个draw函数我们也得暂放一段时间。
9. 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
10. \_querySupport->endQuery(state);得到GPU 计算的时间，并记录Draw traversal统计的所有时间
11. 最后要取消所有的camera的引用。

十八、

好了，现在我们经过三节的介绍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render还是绘制cull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。

我们总结一下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）

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。

十九、

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，

a场景树底端的叶节点（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；

b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；

c整个场景的根节点有且只有一个节点可以直接作为，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

d节点所附带的渲染状态集（osg::StateSet）用来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，一个状态集中可以包含多种不同的渲染属性和开关，处于场景树底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

通过上面几节关于renderingTraversals的介绍，我们大概了解到OSG 渲染后台的主体是场景视图（SceneView），而场景视图（SceneView）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。所以我们先介绍一下场景视图（SceneView）中重要的集中内部类。

1、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，从而将这些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CullVisitor只能在系统渲染后台的环境中使用。

2、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。

3、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 。

4、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”。

5、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。

6、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。

// 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 );

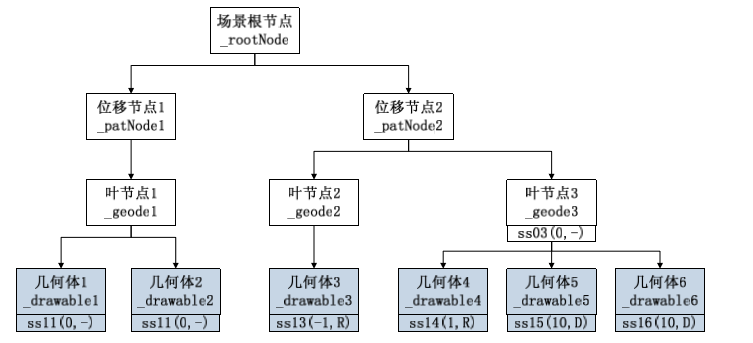
分别等价于：

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

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

二十

首先我们来看一个场景构建的实例，并通过它来了解一下“状态节点”StateGraph 和“渲染叶”RenderLeaf 所构成的状态树，“渲染台”RenderStage 和“渲染元”RenderBin 所构成的渲染树，进一步了解这两棵树之间错综复杂的关系，以及理解它们与场景节点树之间更加复杂的关系。



上面是一个虚构的场景结构图，其中叶节点\_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（ss11(0,-)）。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”）。也就是

ss03->setRenderBinDetails( 0, “” );

ss11->setRenderBinDetails( 0, “” );

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

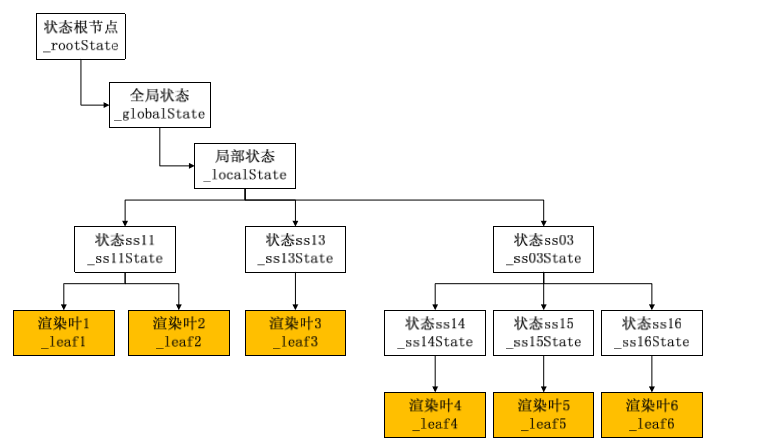
ss14->setRenderBinDetails( 1, “RenderBin” );

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

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

注意1、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2、 “setRenderBinDetails(0, “”)”是缺省的设置，这里写出来是为了便于区别；我们还要知道 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“局部状态节点”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。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：

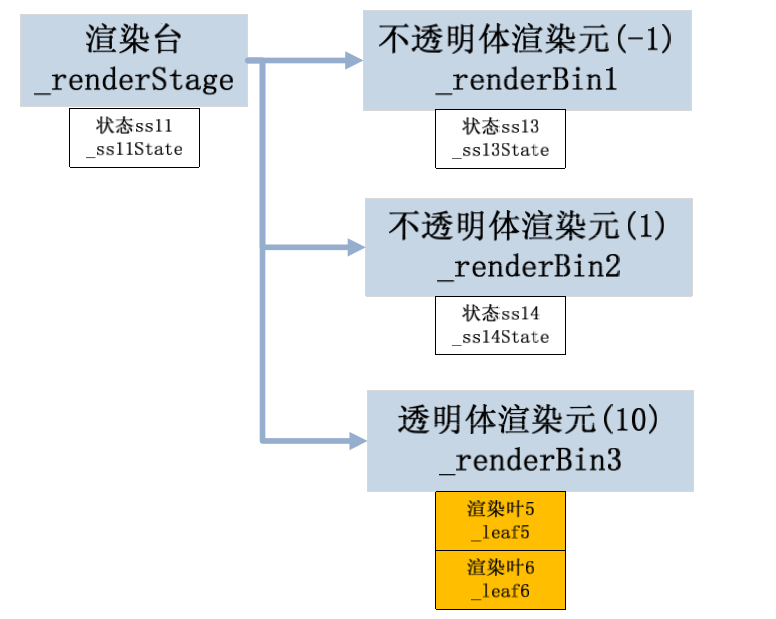
1、状态树是根据渲染状态（StateSet）来生成的，那些没有设置 StateSet 的场景节点将 不会影响状态树的构架；

2、场景中的 Drawable 对象在状态树中被分别置入各自的渲染叶（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。

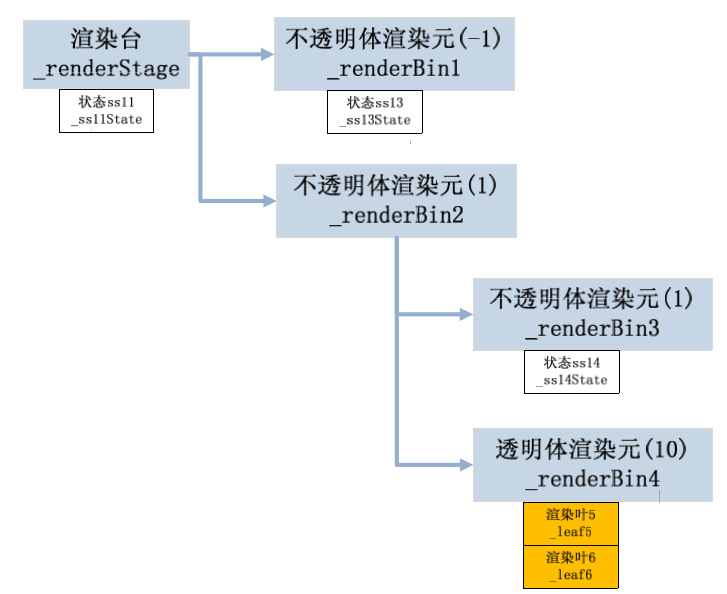
但是，如果我们对于场景中部件的渲染顺序有特殊要求的话，那么渲染树也会因而变得 复杂，上面的场景示例后可能得到如下的一株“渲染树” ：



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。

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。

二十一、

上面我们用了四节课的内容，讲解了一些osg概念性的内部原理。希望大家可以再看今天的讲解之前先再仔细的研究一下前四节的内容。这样你就会对整个osg的渲染过程有一个更加清晰的认知，有助于理解下面两个函数cull()和draw()。

好，相信大家已经又复习了上面几节的内容，那我们就进行sceneView的第一个重要功能osg::Util::SceneView::cull()函数。场景的筛选函数 cull 主要完成了以下几个工作。

1、通过\_camera->getNodeMask()==0来判断主相机下的根节点是否可见，如果不可见则不用进行cull操作。

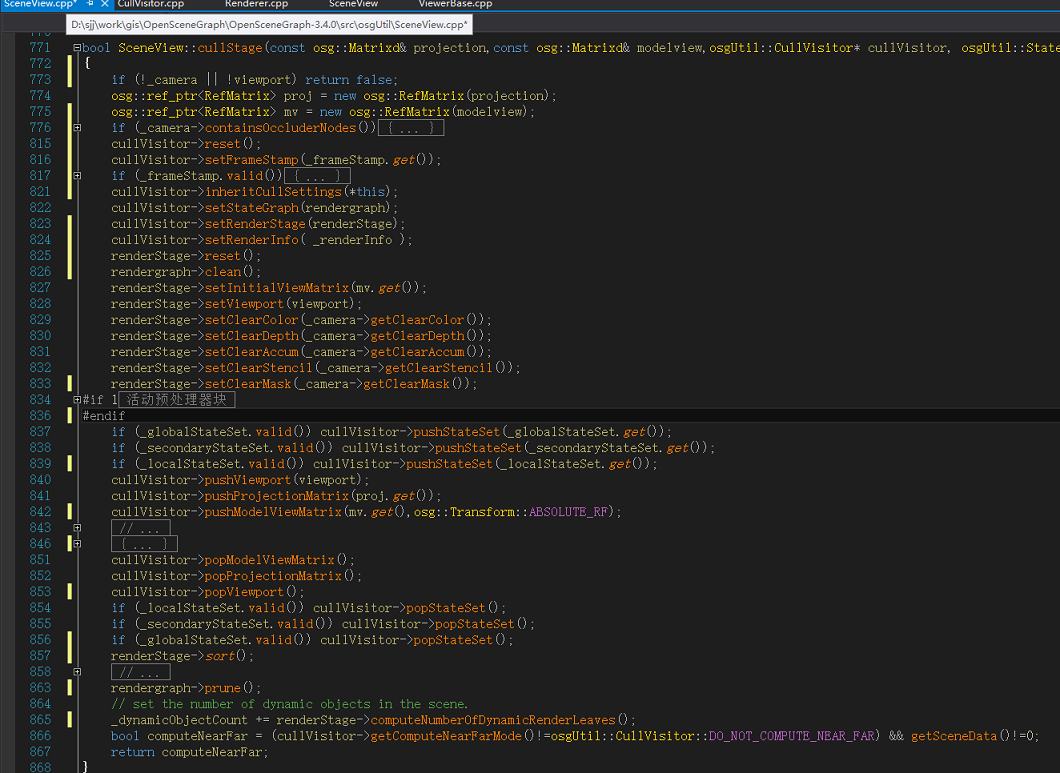
2、进行视图的渲染信息（\_renderInfo）的初始化

3、通过updateUniforms();更新一些osg内设的 osg::Uniform 着色器变量（osg\_FrameNumber，osg\_FrameTime等）。因为opengl中Uniform表示这个变量可以被所有的shader访问到，所以我们在编写 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。

4、对筛选访问器（\_cullVisitor），状态树根节点（\_stateGraph）和渲染树根节点（\_renderStage）进行初始化操作。

5、通过\_displaySettings->getStereo()来判断是否进行立体渲染(例如VR等)，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。这些就等我们研究到osg与vr(ar)结合的时候我们再仔细的研究。

6、(不是立体渲染)进行普通渲染，通过cullStage进行cull操作。我们进行SceneView::cullStage内。



A、首先判断camera中是否存在遮挡节点(OccluderNodes)如果存在，就会通过\_collectOccludersVisitor->traverse(\*\_camera)，根据投影矩阵（projection）、视图模型矩阵（modelview）以及视口矩阵（modelview）来对场景图进行遍历，找到发生遮挡的位置(也就是得到被遮挡的节点)。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）

B、设置状态树以及渲染树构建所需的各种信息。例如viewport、ClearColor、ClearDepth、ClearAccum、ClearStencil、ClearMask、camera等，还要对全局状态节点以及局部状态节点进行默认设置(上一节说到过)。

C、如果camera中设置了CullCallback回调函数，那么就要对摄像机下的所有节点都要进行遍历，并且执行回调函数。如果没有则使用筛选访问器cullVisitor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（被遮挡的，以及超出视椎体的），并将它们裁减掉，从而提高场景绘制的效率。

7、进行完了cull操作后。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）、投影矩阵、视口矩阵，渲染状态（使用 popStateSet））

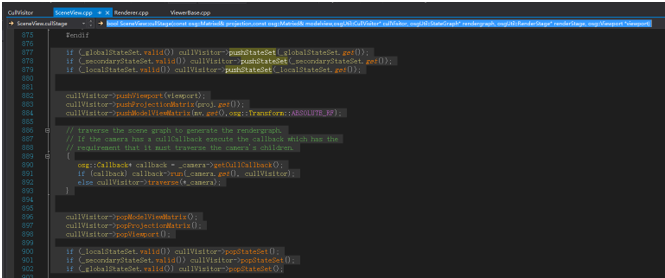
8、依次对筛选访问器执行之后得到的渲染树内容进行排序RenderStage::sort和精简StateGraph::prune（构建过程中可能有些空节点需要剔除）

9、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

10、返回结果computeNearFar。

接下来我们将要深入osgUtil:: CullVisitor类，进一步研究裁剪操作。

二十二、



上一节我们大致上过了一遍sceneView::cull()函数，通过研究，我们发现上图中的这一部分的代码才是整个cull过程的核心部分。所以今天我们来仔细的研究一下这一部分。

1、cullVisitor->pushStateSet函数：

A、\_currentStateGraph = \_currentStateGraph->find\_or\_insert(ss); 判断传入的渲染状态 ss 是否已经存在于这个状态节点中，如果存在则返回找到的那个ss，如果不存在新建一个包含了 ss 的状态节点。

B、创建渲染树，这个过程必须得满足三个条件。1）设置了渲染细节useRenderBinDetails()并且满足BinName不为空(只能为“RenderBin”或“DepthSortedBin”)；2）渲染状态没有采用覆盖渲染细节（OVERRIDE\_RENDERBIN\_DETAILS）的方式(\_numberOfEncloseOverrideRenderBinDetails==0);3）使用 setRenderBinDetails 设置了渲染细节，如果满足这三个条件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。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。

2、向cullVisitor中设置MVP矩阵，然后通过cullVisitor->traverse(\*\_camera); （实际上就是调用CullVisitor::apply 函数函数（在遍历节点时调用））来负责根据不同的节点类型，在不同的时机调用cullVisitor->pushStateSet函数以及cullVisitor-> popStateSet函数。（下一节将会进行展开说明）

3、cullVisitor-> popStateSet函数：

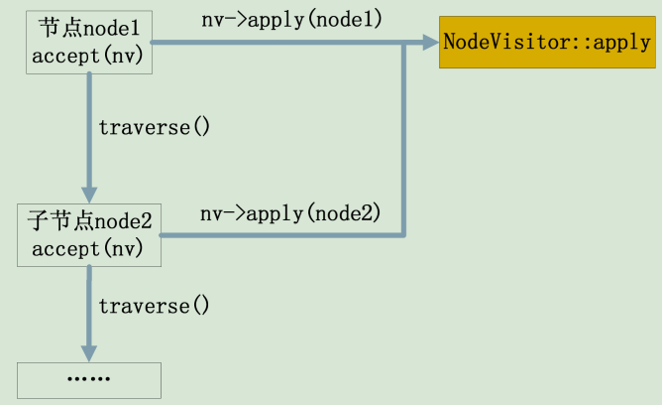
A、\_numberOfEncloseOverrideRenderBinDetails减一，然后还是cullVisitor->pushStateSet函数介绍的三种情况，判断是否使用了RenderBinDetails。然后还是从堆栈中取出上一次渲染树中所处的渲染元节点，并跳转到这一位置。

B、状态树从当前位置跳转到其父节点，即：

\_currentStateGraph = \_currentStateGraph->\_parent;

通过简单的对cullVisitor->pushStateSet函数以及cullVisitor-> popStateSet函数的介绍，可以了解到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，直到遍历完成。当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。这么整个过程就是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，就不会在进行渲染操作。这就是我们下一步要讨论的CullVisitor 的工作。

CullVisitor根本上就是NodeVisitor的子类，所以他的执行过程可以按照节点访问器（NodeVisitor）的工作原理来理解。



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。

下一节我们就要对CullVisitor的 apply 重载函数进行探究。

二十三、

osgUtil::CullVisitor中，我们发现apply函数的重载中，有CullVisitor::apply(Group& node)，CullVisitor::apply(Switch& node)， CullVisitor::apply(LOD& node)，CullVisitor::apply(Geode& node)，CullVisitor::apply(Node& node)是一样的函数内容。所以这五个函数我们就挑出CullVisitor::apply(Node& node)进行探究。

CullVisitor::apply(Node& node)

1、执行isCulled函数(osg::CullStack中)，

inline bool isCulled(const osg::Node& node)

{

if (node.isCullingActive())

{

return getCurrentCullingSet().isCulled(node.getBound());

}

else

{

getCurrentCullingSet().resetCullingMask();

return false;

}

}

通过node.isCullingActive()来判断，node是否开启了被cull的开关（osg::Node::isCullingActive方法在满足1）没有任意的孩子节点设置为不被剔除；2）自身的\_cullingActive属性也得为true；3）包围盒可用。三个条件下才返回true），当为true时，通过osg::CullingSet下的isCulled判断是否满足1）在视椎体内；2）是否为小物体；3）是否被遮挡节点遮挡。满足其中一个则返回true，代表这个节点要被剔除。如果node.isCullingActive()为false则意味着不对这个节点进行剔除操作。也就是说执行的isCulled 函数，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。

2、node满足不被cull的条件后，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。我们可以到 osg::Polytope::contains 的重载函数中认识这个过程。

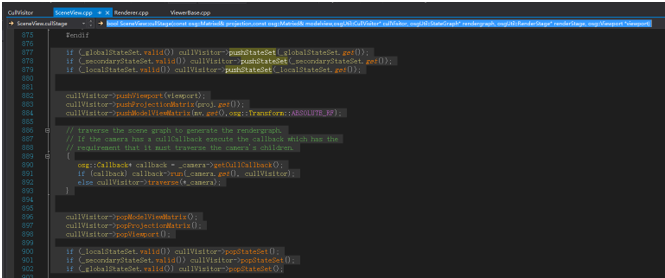
3、这一步就是我们上一节说到的状态树和渲染树的创建。获取节点的渲染状态（StateSet），如果存在的话，使用pushStateSet函数，将这个 StateSet 对象置入到当前的状态树和渲染树中，并添加到对应的状态节点/渲染元中，或者为其新建一个相关的节点。

4、如果设定了裁剪回调函数，那么它的调用时机就是在这里。当然如果没有设置回调函数，那么就要在这里遍历这个node下的所有的子节点，进行cull操作。

5、后面就是是从堆栈中依次弹出模型的StateSet，以及恢复遍历掩码和筛选设置的原先值。

这就是整个cull在遇到node节点时发生的动作。当然cullVisitor的apply的重载有很多，我们可以试着自己进行分析一下。

这样我们就完成了，对场景中所有节点的裁剪操作以及构建完成了状态树和渲染树，我们进行完成了下图的内容



这时我们还要回到SceneView::CullStage()函数中，就会看到我们在前面提到过但是没有深入讲解的两个函数

renderStage->sort();//渲染台排序

rendergraph->prune();//状态树的优化

为了方便大家的理解，请大家一定要走一遍osgUtil::CullVisitor::apply(Camera&)函数。因为RenderStage::sort 函数的排序是按照前序渲染台，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 osgUtil::CullVisitor::apply(Camera&)实现的，所以osgUtil::CullVisitor::apply(Camera&)函数一定要研究透。

二十四

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）。

在上一节我们提到，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。

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）。

在前前几节中我们反复说过，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。

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：

1、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）。

2、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。

现在我们又要回到osgUtil:: SceneView::cullStage ()。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，设置数据变动的方式众所周知：obj->setDataVariance( osg::Object::DYNAMIC );

好，那我们就要正式结束sceneView::cull 函数的讲解了。下一步就是要探究render函数了。

二十五

我们用了四天的时间才探究了一番viewerBase::frame()函数中的renderingTraversals中的场景节点裁剪工作，我们总结一下前面所介绍的内容。首先osg::ViewerBase::frame函数在整个程序的第一帧的时候对一些重要的内脏器官进行初始化操作例如viewer,camera,scene等。然后就是记录上一帧的经历的时间使用advance函数。再之后就是分别执行事件遍历和更新遍历的操作，事件遍历就是对键盘、鼠标等输入事件进行处理，以及用户自定义的一些事件操作；更新遍历主要就是数据分页(设置了pagelod或者设置了proxyNode)以及用户自定义的一些节点的更新操作。上面的这四步以及renderingTraversals的裁剪操作都是为接下来的绘制操作进行数据处理。

所以渲染渲染遍历中最主要的还是渲染，所以我们现在开始进入上面提到的sceneView类中的draw函数的探究。再进入osgUtil::SceneView::draw()函数，之前首先介绍一下前面经常提到的state类。

Osg::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。Osg::State 类还有以下几点重要功能：

（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； 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。

（3）允许用户直接查询各种 OpenGL 状态的当前 值。

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。

二十六

我们今天今天真是进入SceneView::draw()函数。

1、判断主相机是否为隐藏状态也就是(camera->getNodeMask()==0)，隐藏状态就会退出这个draw函数。

2、开始初始化osg::state类，上一节我们对osg::state类进行了简单的介绍。这里就是设置了state的framestamp以及displaySetting的设定。再就是初始化opengl的扩展功能工作通过state->initializeExtensionProcs()函数进行。

3、初始化一个文理对象管理器，主要功能就是维护了一个map，保存纹理，生成新的纹理对象以及删除纹理对象等。以及初始化一个glBuffer的管理对象（在osg::BufferObject中的GLBufferObjectManager类）。

4、判断这个draw函数是否为第一次进入，通过\_initCalled来判断。如果是第一次进入则就需要进行对所有的场景图形中的对象进行初始化操作，通过nodevisitor进行。如果我们在渲染过程成加载未初始化的数据，这需要调用update和cull自动调用。

5、为了支持多管道系统，将推迟删除OpenGL显示列表和纹理对象，直到OpenGL上下文是最初创建对象时的正确上下文。 在这里，我们知道我们所处的上下文，以便刷新适当的缓存。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。

6、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 。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。立体渲染我们会在以后的osg+hololens中介绍。现在我们仅针对非立体显示的情形进行介绍

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。

这里我们就会发现draw和cull的套路是一样的，我们需要进入RenderStage::draw中继续探究。

二十七

上一节我们介绍到SceneView::draw()函数的核心的两个绘制函数

\_renderStage->drawPreRenderStages(\_renderInfo,previous);

\_renderStage->draw(\_renderInfo,previous);

我们首先进入中RenderStage::drawPreRenderStages可以看到其实前置渲染也是调用了RenderStage::draw函数所以我们开始对它的探究。

我们首先大概的介绍一下这个函数的主要功能：

1、通过\_stageDrawnThisFrame判断渲染树是否在已经渲染完成在这一帧中。

2、设置视图矩阵以及渲染相机到renderInfo中，有助于在draw函数执行过程中检索到camera。

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

4、其实drawPreRenderStages函数已经在SceneView::draw()中调用了，所以这里只是以防万一的作用。

5、运行摄像机设置（RenderStage::runCameraSetUp），这里主要是在RTT(渲染到纹理)时会用到。我们先大致的介绍一下这个函数(RenderStage::runCameraSetUp)的流程

1）获取AttachmentMapModified计数

2）获取渲染目标。

3）获取渲染目标后备。

4）获取BufferAttachmentMap，用于配置帧缓冲区对象(pbuffer)和纹理读取

5）计算pbuffer所需的尺寸

6）绘制完渲染台后需要复制的图像。

7）如果存在一个RenderStage的附加图像。就会设置到\_bufferAttachmentMap中

8）对RenderStage的附加纹理进行大小设置。

9）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

6、通过osg的图形上下文得到state以及判断是否为多线程。

7、执行前置渲染的callback函数

8、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。

9、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

10、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。

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

12、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）。

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

14、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。

二十八

通过上一节的介绍，我们发现绘制的核心函数是drawInner( useRenderInfo, previous, doCopyTexture);今天我们就开始探究这个osg的核心动作。

osgUtil::RenderStage::drawInner函数。

1、定义了一个SubFunc的结构体主要是从FBO中读取纹理信息。

2、通过state对象得到osg是否支持fbo以及多纹理目标渲染的方式，如果则在没有提供OSG\_GLES1\_AVAILABLE和OSG\_GLES2\_AVAILABLE宏的基础上绑定glDrawBuffer到\_drawBuffer以及glReadBuffer到\_readBuffer上。

3、把fbo通过ext->glBindFramebuffer(target, 0);绑定到opengl的状态集中。

4、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(下一节进行介绍)

5、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。关于osg中遇到的opengl错误的排查我建议大家使用过gdebugger或者是AMD-CodeXL（gdebugger的升级版本）具体的使用方法我会在接下来的内容中进行介绍。

6、对fbo进行解析(使用 glBlitFramebufferEXT 进行解算).根据GLbitfield从所有的framebuffer中找到那个类型(GL\_DEPTH\_BUFFER\_BIT 、GL\_STENCIL\_BUFFER\_BIT、GL\_DEPTH\_BUFFER\_BIT | GL\_STENCIL\_BUFFER\_BIT、GL\_COLOR\_BUFFER\_BIT)的buffer需要被复制。然后通过ext->glBlitFramebuffer函数得到可以读取的fbo的内容。

7、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。

所以我们就大概了解了drawInner函数主要是完成绘制到FBO上。但是如果我们想要使用 FBO 来实现纹理烘焙(RTT)或者场景截图的话，特别要注意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。

RTT(纹理烘焙|渲染到纹理)简介.

目的是：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。

Osg上的设置渲染目标和绑定纹理的方法十分简单：

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。

我们上一节讲到了RenderStage::innerDraw函数。这个函数主要是在fbo上完成绘制工作。其中核心的函数是RenderBin::draw()函数。这里仔细的你一定会很疑惑，draw函数在RenderBin中不是静态的成员函数也不是全局函数怎么可以用域名调用呢？其实原因很简单RenderStage为了方便管理所有的RenderBin对象，就直接继承了RenderBin类。解决了这个很小的问题，下一步我们就要进入RenderBin::Draw函数中一探究竟。

osgUtil::RenderBin::draw()的主要作用就是从渲染树的顶端的渲染台开始遍历，完成对所有的RenderBin的绘制操作。

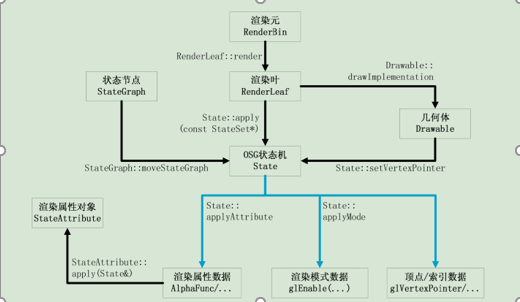
1、把当前的RenderBin对象设置到renderInfo中进行记录。

2、判断如果这个RenderBin对象设置了自定义的绘制回调通过RenderBin::setDrawCallback。那么则意味着这个回调函数就会替换RenderBin:: drawImplementation函数来完成这个RenderBin的绘制工作。所以我们在设置RenderBin的DrawCallback函数的时候一定要清楚的知道自己想要作什么。

3、RenderBin的draw函数最终就是调用RenderBin::drawImplementation函数。所以我们下一步就是进入RenderBin::drawImplementation函数中进行探究。

RenderBin::drawImplementation

在这里我们将要结合前面所讲到的osg::state类的功能(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)以及节点树、状态树和渲染树之间的关系(节点数负责管理场景中的所有的可见或不可见的对象，状态树是负责整理场景中所有的osg::state对象，渲染树是根据指定的渲染顺序依次从状态树中取出继承了所有的渲染状态---osg::state的几何体信息)。我们把这里所有涉及到的信息总结成了下图的图表



上图的理解：

1、RenderBin的主要作用就是遍历其中所有的RenderLeaf执行其render()函数，

2、在执行render函数的时候需要用到osg::state中保存的opengl状态数据，

3、这些数据依据状态树上的所有父节点的渲染状态从而集合得到所需的opengl的状态。

4、当然在进入执行渲染之前还需要从Drawable中得到顶点、索引等数据。其实这些数据也是被保存在osg::state中。

5、汇总完成所有的渲染状态和渲染数据以后就可以进入openg的渲染管线进行渲染。

注：上图的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。

如果理解了上面所总结的内容，那我们我们就算是完成了对OSG 场景绘制模块的探究。但是我们下一节还要继续去看看这些内容在代码中是怎么实现的。

osgUtil::RenderBin::drawImplementation

1. 从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osg::State对象。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)中取出跟踪此图形上下文的当前OpenGL状态的State对象

2、确定当前的state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。这个渲染状态在初始化RendBin时就会默认设置好透明渲染元（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

3、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，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

4、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

5、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

6、遍历所有的子渲染元（RenderBin::\_bins），在这里执行它们的 RenderBin::draw 函数，构成了一个递归调用。

7、把上面插入的渲染状态 RenderBin::\_stateset移除。

这样我们就明确了RendBin的draw函数其实就是按照指定的StateSet::setRenderBinDetails的值(<0的先进行渲染)来调用其中的renderLeaf的render函数。

osgUtil:: RenderLeaf::render()

1、判断这个渲染叶中的渲染状态是否设置了终止渲染标识，如果设置了那么就会跳过这个渲染叶的draw函数(也就是如果想要尽快的结束渲染可以直接直接设置\_abortRenderingPtr)

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

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

4、StateGraph::moveStateGraph(state,NULL,\_parent->\_parent);的作用是判断当前渲染叶与上一次处理的渲染叶父节点是否相同，如果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）。我们进入这个函数就可以发现他的具体的工作是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。

5、执行函数State::apply(const StateSet\*)，由 OSG 状态机处理并执行相应的 OpenGL 指令

6、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

总结

我们都了解了渲染叶RenderLeaf是几何体（Drawable）对象的唯一管理者。Draw函数主要是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（传递这些信息的在 CullVisitor::addDrawableAndDepth 函数中进行的），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。

我们还有最后遗留的两个函数osg::State::apply(const StateSet\*)以及Drawable::draw(RenderInfo& renderInfo)。那我们就继续探究这两个函数。

osg::State::apply(const StateSet\*)

1、把StateSet放入到\_stateStateStack状态栈，这个状态栈会在StateAttribute中查询使用。

2、清空当前的uniform list。排除上一个renderbin中state的影响。

3、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。得到所有texture有关的modelist以及attributelist。然后遍历所有的modemap，调用applyModeOnTexUnit函数把mode通过glEnable设置到对应的texture上。然后调用applyAttributeOnTexUnit函数把调用attribute.apply的函数，设置到对应的texture上。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

4、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。

5、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：

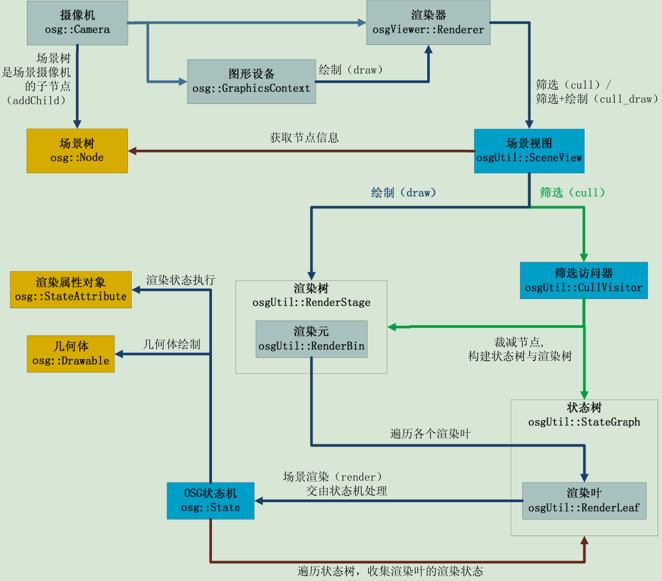
attribute->apply(\*this);

6、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。

7、标识已经更新完毕的渲染状态。

Drawable::draw(RenderInfo& renderInfo) 进行几何体的实际绘制，他的核心就是调用每个继承自Drawable的几何体的drawImplementation函数。所以我们想要在自己的程序中实现一个新的drawable子类最主要的就是重写drawImplementation函数。

好。到这里我们就算是完成了对osg::frame的整体介绍(针对单线程的)，下面我们就来整体的总结一下fame函数的整个流程。



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。

Osg中最初使用的多线程的地方是在osgViewer::Viewer::realize()的时候调用的setUpThreading()函数，而这个函数就是osgViewer::ViewerBase::setUpThreading()。而这个函数中如果是单线程(SingleThreaded)模式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）。

我们先介绍一下多线程模式会遇到的一些成员变量

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。

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。

\_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

如果是多线程则调用osgViewer::ViewrBase::startThreading()函数。所以我们进入这个函数中查看一下。

1、调用releaseContext()函数来释放\_hdc（在win32中）。

2、得到所有的contexts以及cameras。

3、根据线程模式判断开始的线程栅栏以及结束栅栏的数量。并且从这个switch函数我们可以看出各个线程的区别。numThreadsOnStartBarrier和numThreadsOnEndBarrier用于多个GraphicsThreads相互同步

**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

4、得到所有的scene，确保使用线程安全ref / unref分配现有场景图形对象，并更新每个场景图，使其具有足够的GL对象缓冲区内存，用于将使用它的图形上下文。

5、判断是否为CullDrawThreadPerContext模式，如果是则遍历所有的相机设置筛选加绘制（Renderer::cull\_draw）操作。

6、再根据ThreadMode初始化\_startRenderingBarrier、\_endRenderingDispatchBarrier以及\_endDynamicDrawBlock。

7、numThreadsOnStartBarrier大于1就初始化一个开始栅栏，栅栏的强度就是numThreadsOnStartBarrier，当numThreadsOnEndBarrier大于1就初始化一个结束栅栏，栅栏强度就是numThreadsOnEndBarrier。

而开始栅栏(\_startRenderingBarrier)以及结束栅栏（\_endRenderingDispatchBarrier）的具体用法要到渲染循环中进行查看。如果您还对我们前面介绍的渲染遍历有印象那么可以直接跳转到osgViewer::ViewerBase:: renderingTraversals函数下进行查看。如果您不清楚osg的渲染遍历请移步到上一节进行仔细的阅读。

ViewerBase:: renderingTraversals中，我们可以找到这样一句话

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

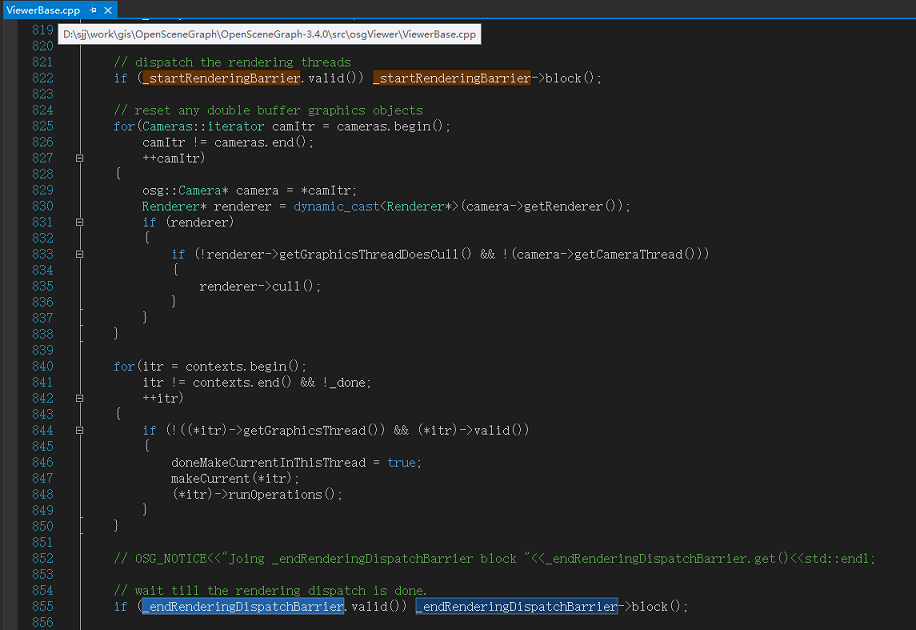
它意味着我们选择将主进程予以阻塞，此时后面的代码都不会被执行，直到这个栅栏被冲开为止。而冲开的条件就是所有图形设备（GraphicsContext，以下简称 GC）的线程都执行了\_startRenderingBarrier->block()函数，并因此阻塞了运行。因此，当且仅当所有 GC 线程执行到 block 函数并被阻塞时，栅栏恰好被冲开，主进程以及各个 GC 线程继续执行各自后继的代码。

我们在结束渲染的时候又会遇到这样一句话

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。

而开始栅栏和结束栅栏之间就是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



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。

我们继续在ViewerBase::startThreading()中往下看，当ThreadMode是DrawThreadPerContext和CullThreadPerCameraDrawThreadPerContext两种模式下\_endDynamicDrawBlock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。他用于确定所有的设定为object->setDataVariance( DYNAMIC );的动态对象全部被执行完毕后，才会开启下一帧的渲染。这样可以避免正在渲染流程中的数据被下一帧的用户回调所更改产生未知的错误，甚至直接导致系统崩溃。

那么什么时候才会触发释放这个锁呢。先给出这个函数completed().。当 completed 函数的执行次数达到我们设定的最大值时，将释放被阻塞的线程，也就是我们的主进程。在osgViewer::ViewerBase::startThread()中的上下文我们会发现这里的最大值正是场景中摄像机渲染器（Renderer）的个数。现在我们就要深入到RenderLeaf::render 函数中看一下具有动态修改属性的物体是怎么个渲染流程。

osgUtil::RenderLeaf:: render()的最后三行我们可以看到一下内容

if (\_dynamic)

{

state.decrementDynamicObjectCount();

}

osg::State的decrementDynamicObjectCount的函数内容是

inline void decrementDynamicObjectCount()

{

--\_dynamicObjectCount;

if (\_dynamicObjectCount==0 && \_completeDynamicObjectRenderingCallback.valid())

{

\_completeDynamicObjectRenderingCallback->completed(this);

}

}

我们可以了解到这个函数就是用来减少渲染此帧的动态对象的数量，一旦计数变为零，则调用DynamicObjectRenderingCompletedCallback以通知完成。

1、这里的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 函数计算得到。

2、completeDynamicObjectRenderingCallback回调函数实际上就指向\_endDynamicDrawBlock，就是在这里要执行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。

当然这个endDynamicDrawBlock也在osgViewer::ViewerBase:: renderingTraversals函数中进行对渲染主线程的阻塞操作，位于上一节的在“渲染开启栅栏”和“渲染结束栅栏”的同步代码段之后。请大家自己的查看相应内容。

我们继续回到osgViewer::ViewerBase::startThreading()函数中，继续向下探究。

下一步就是要遍历所有的GraphicContexts，为每个context创建单独的线程。因为除了singleThread其余的三种ThreadMode(CullDrawThreadPerContext, DrawThreadPerContext, CullThreadPerCameraDrawThreadPerContext)都有为每个Context分配一个单独的线程。

1、指定这个gc的动态物体绘制完成的callback函数。

2、gc->createGraphicsThread();创建单独的图形线程为这个Context。

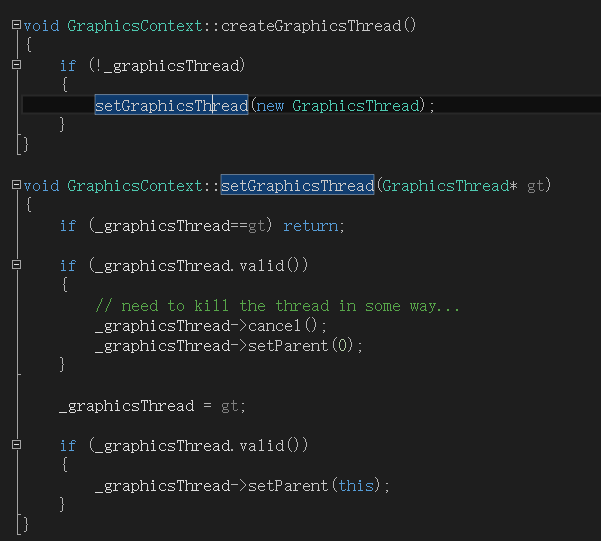
3、把创建好的线程随机分配到一个cpu上。

4、当threadMode为CullDrawThreadPerContext时，指定这个图形线程的开始渲染栅栏

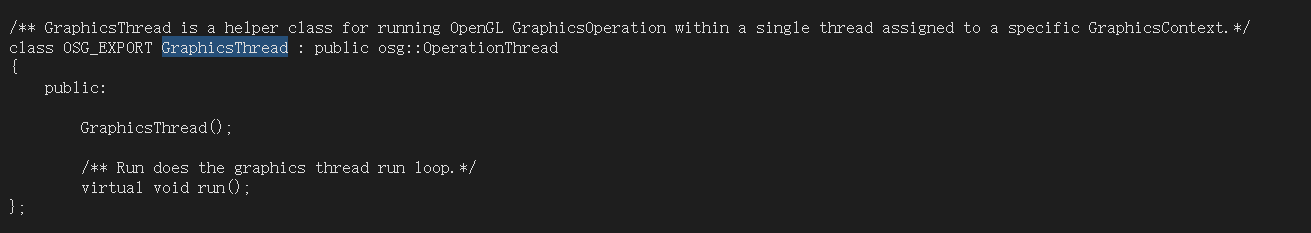
5、根据指定的swapReadyBarrier( 任务的作用是在交换双缓存之前对所有 GC 线程执行一次同步)的位置，来指定这个图形线程的结束栅栏。

那我们就来探究一下图形设备线程的创建以及工作流程。gc->createGraphicsThread();

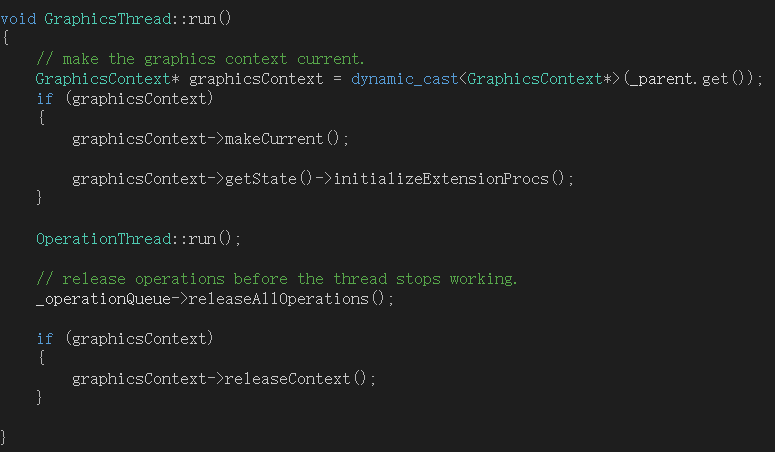
GraphicsContext::createGraphicsThread函数就会发现其实很简单就是创建一个线程对象，然后绑定到这个context上。



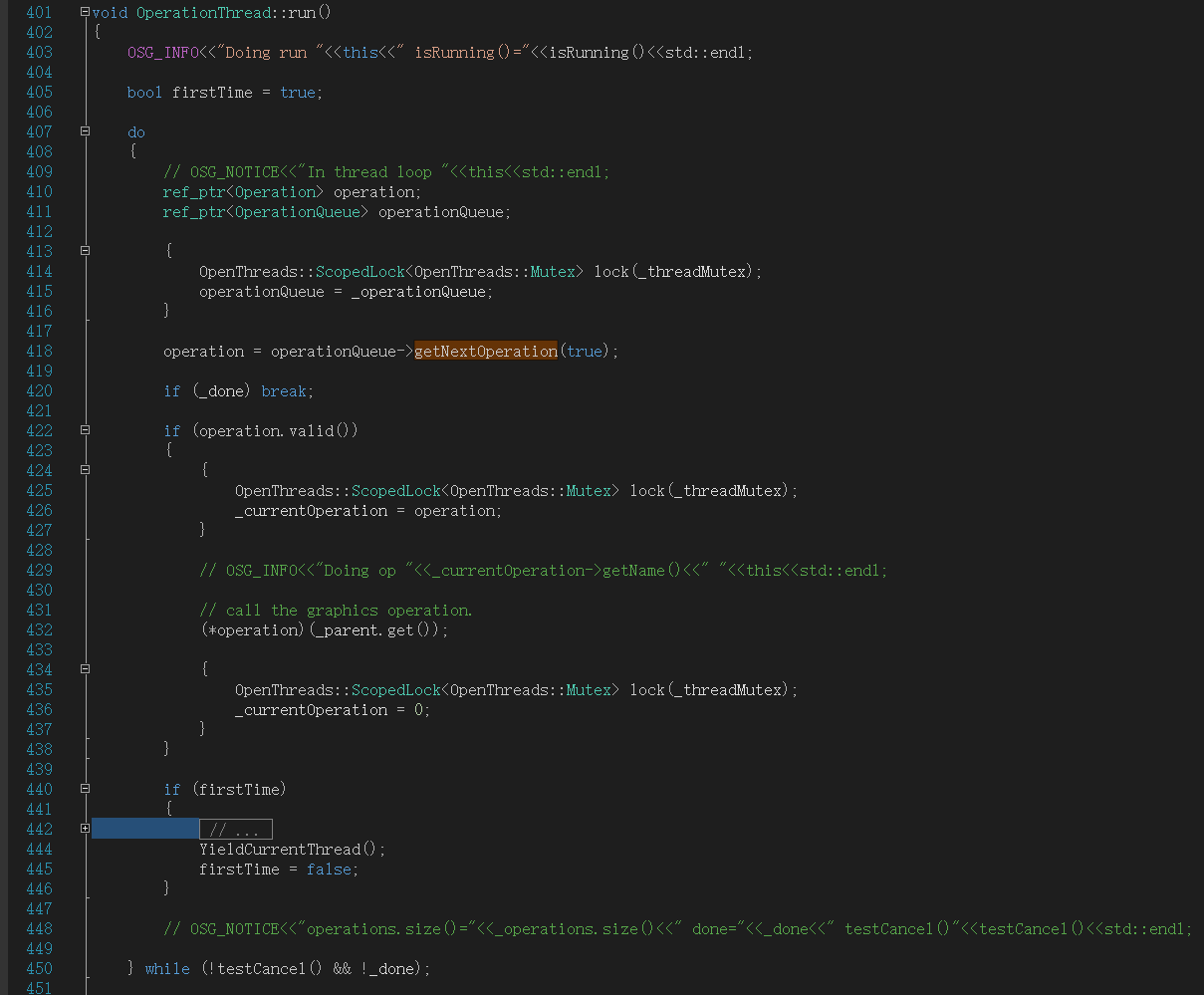
当我们使用 GraphicsContext::createGraphicsThread 创建线程时，得到的是一个 osg::GraphicsThread 线程对象（这个与我们下一步要研究的使用 Camera::createCameraThread 创建线程osg::OperationThread 对象不相同，但是osg::OperationThread是osg::GraphicsThread的父类。）



实际上，就是重写了OperationThread的run函数。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。



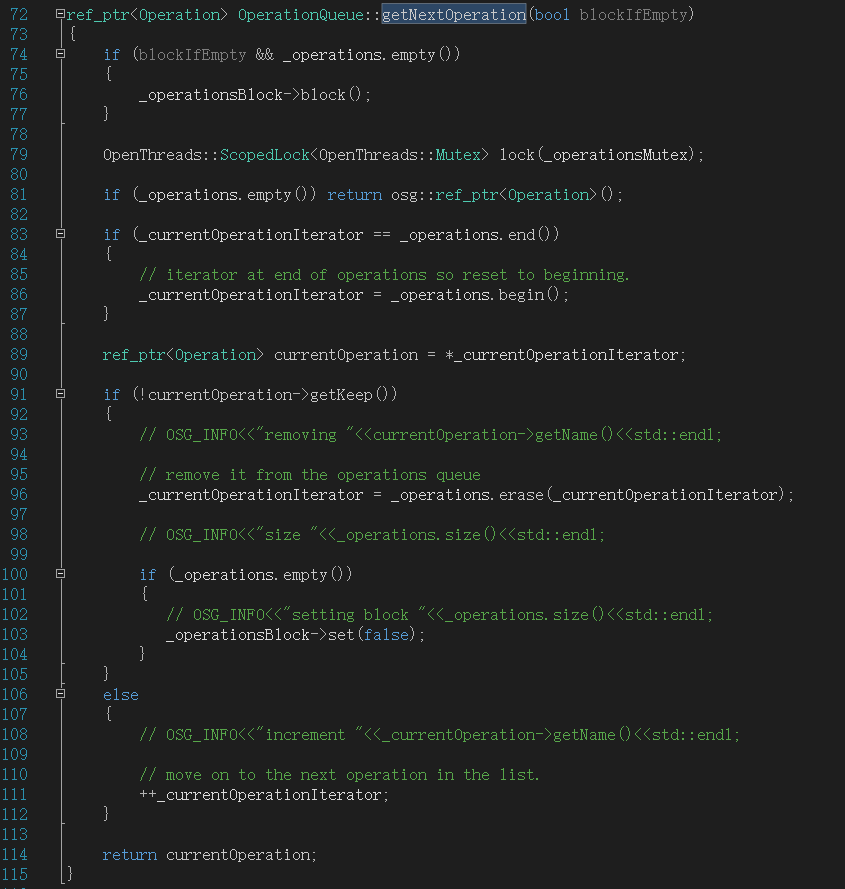
我们现在就来看看OperationThread 线程的执行内容(run函数)。这样我们不仅可以了解GC 线程启动以后要执行的工作，也可以窥探到摄像机线程要反复完成的工作。



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）。只要从 std::list 列表中取出一个 osg::Operation 对象就可以了。

我们进入这这个函数探究一下这个看似简单的函数其中到底有什么需要注意的地方



a、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

b、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

c、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， 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。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。

我们来简单的介绍一种Operation的对象。

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。

到这里我们就完成了对startThreading函数的探究工作，我们要注意这个 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。下一节我们就来介绍总体上研究一下这几种线程模型，在任务的添加和处理顺序上各自有什么要求呢？