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

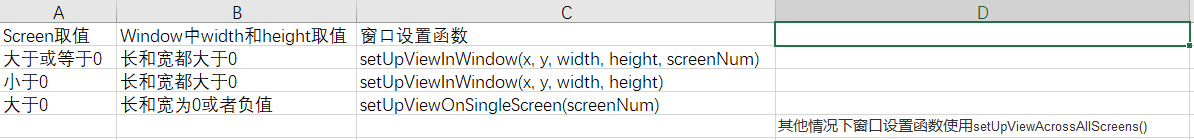
{

setUpViewInWindow 100 200 600 400 0

}

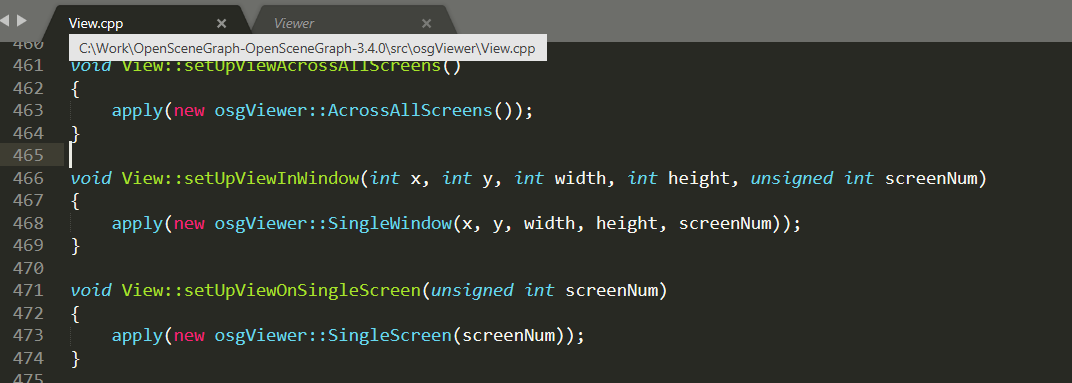
如果设置了OSG\_CONFIG\_FILE环境变量，但是环境变量所对应的文件解析有错，那么整个程序就会退出。

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：

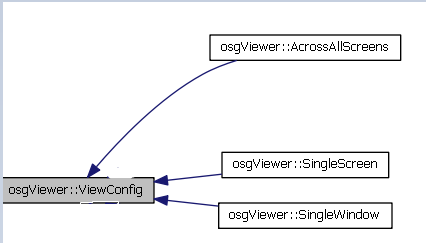


当窗口设置完成之后，osg会再次调用getContexts(contexts) 搜集目前可用的设备渲染上下文，如果这时候还没有窗口产生，也就是说设置窗口的各种方式都失败了，那么程序就会退出，如果成功，那么会针对已经常见的窗口进行一些设置，完成后续的操作

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。

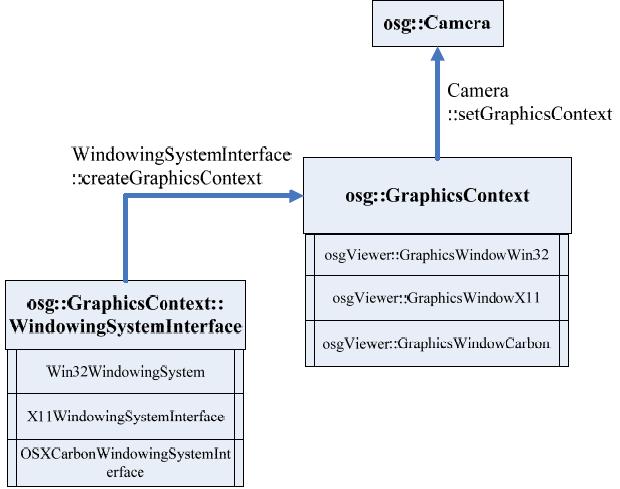
\_numMultiSamples:多重采样的子像素样本数,默认为 0。如果显示卡支持的话,打开多重采样可以大幅改善反走样(anti-aliasing)的效果。此外还有很多可以设置的类变量,如\_minimumNumberStencilBits(模板缓存的最小位数)等,其默认设置均在 osg::DisplaySettings::setDefaults 函数中完成,其中有些变量可能还没有作用。要注意的是,DisplaySettings 的作用仅仅是保存所有可能在系统显示中用到的数据,这个类本身并不会据此改变任何系统设置和渲染方式。

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

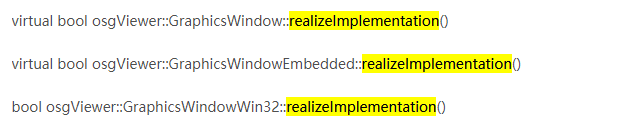
上一节我们对完成了对osg生物内部非常重要器官graphicsContext的初始化工作。这样就可保证我们场景中至少有一个graphicContext存在，不至于刚出生就面临夭折。我们根据上一节中osg代码的研究也就知道了，在我们正常使用osg时，是怎么完成对camera以及graphicContext的创建的了。

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。



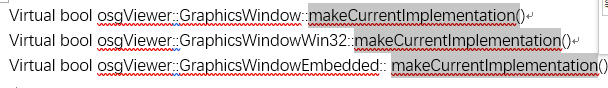
先遍历所有的GraphicsContext，然后判断是否设置了同步交换缓冲区（这一般是渲染的最后一步），这是osg提供的多机同步swapbuffer机制，他会默认调用内置的swapbuffer的回调函数（osg::SyncSwapBuffersCallback中，作用主要是等待client端的同步锁，实现多机同步执行swapbuffer）。如果developer想干预的话 可以调用 osg::GraphicsContext::setSwapCallback(SwapCallback\* rc)来设置自定义的缓存交换回调。自定义的回调必须调用GraphicsContext::swapBuffersImplementation()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

