C++与OSG学习

秦声

主要内容

一、编译链接C++程序

二、3D空间坐标到2D屏幕坐标

三、场景图数据结构

四、OSG的渲染过程

一、编译链接C++程序

- 写程序

1、新建解决方案

2、在解决方案中添加项目

3、在项目中添加头文件和源文件

4、编写代码

- 解决方案与项目

1、同一个解决方案中可以有很多项目，这些项目应该为解决一个问题而共同努力。

2、每个项目都生成一个可执行程序，或动态链接库（.dll）。

3、项目之间可以共享资源。比如可以共享源代码；再比如，项目甲的生成的链接库可以被项目乙利用（项目依赖、项目生成顺序）。

- 程序写好了

✓代码写好之后，有很多头文件（.h）和源文件（.cpp），需要经过编译、链接两个步骤才能生成可执行的程序。

✓编译链接的单位是项目。

- 编译

1、在编译过程中，编译器会检查程序语法，并生成一些目标文件（.obj），或者是静态链接库文件（.lib），这些目标文件只是一些中间文件。

2、每个源文件产生一个目标文件或静态链接库文件。

3、编译的真正对象其实是源文件，而不是头文件。

- 为什么头文件不是编译的对象

1、头文件与包含指令（#include）

✓那些没有被项目中任何源文件包含的头文件，编译器是不去理会它的，不管它有没有语法错误，也不管它是否已添加到项目中。

2、包含指令的执行

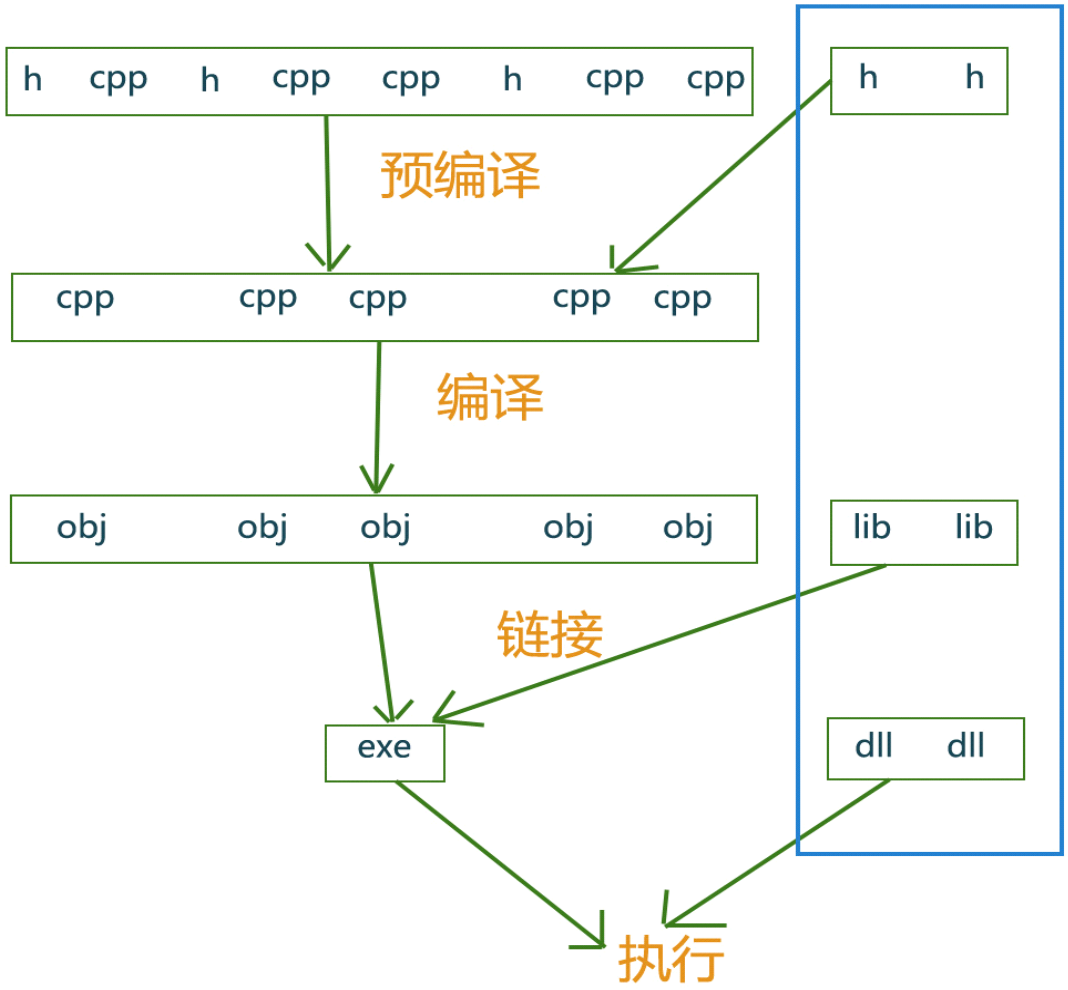
✓包含指令是一种预编译指令，它的执行就像宏替换一样。先找到头文件，然后把头文件中定义的所有符号在包含指令所在源文件中也定义一下。

- 链接

1、链接器的作用是将编译器生成的目标文件，链接在一起，形成最终的可执行程序（或是动态链接库）。

2、链接的对象是目标文件和静态链接库（.lib文件）。

- 编译链接过程



看完这张图，我们可以用它解决很多问题，解释很多道理。

- 解释：包含指令中的尖括号与双引号

✓尖括号与双引号的区别你一定没有忘记，这与头文件的查找目录有关。

✓OSG的配置中，我们要配置包含目录，原因就是让编译器能找到我们所包含的OSG头文件。

✓当然，如果不配置这个包含目录也行，写OSG程序的时候就要在包含指令的双引号中指定头文件的路径啦。

- 链接错误：无法解析的外部符号

✓这是写OSG程序时，经常出现的错误，常常是由于没有添加必要的静态链接库的问题。

✓配置OSG时我们还要配置库文件目录，这是要告诉链接器在哪里找到这些静态链接库。

- 链接错误：一个或多个多重定义的符号

✓同一个头文件被多个源文件包含，那么预编译后的每个源文件都会有头文件的拷贝，头文件中定义的全局变量就会在每个源文件中定义一次，在相应的目标文件中也会有定义，链接时就会出现问题了。

二、3D空间坐标到2D屏幕坐标

- View Matrix

✓View Matrix用来将局部坐标变换到世界坐标，或者说是将模型坐标变换到用户坐标。

✓View Matrix可以由三个分量决定

1、Eye：视点的位置

2、Center：视线上的一参考点

3、Up：向上的方向

- 设置View Matrix

✓OpenGL

glTranslate()、glRotate()、gluLookAt();

在实施变换前，要记得调用
glMatrixMode(GL_MODELVIEW)

✓OSG

osg::Camera::setViewMatrix ()、

osg::Camera::setViewMatrixAsLookAt()。

- Projection Matrix

✓Projection Matrix是投影矩阵，向二维空间投影。

✓有两种投影方式

1、平行投影，或者叫正投影

2、透视投影，构建一个视锥体（平截头体）

- 设置Projection Matrix

✓OpenGL

glOrtho()、gluOrtho2D()、glFrustum()、gluPerspective();

在实施变换前，要记得调用glMatrixMode(GL_PROJECTION)

✓OSG

osg::Camera::setProjectionMatrix()、

osg::Camera::setProjectionMatrixAsOrtho()

…AsOrtho2D、…AsFrustum、…AsPerspective

- Window Matrix

✓绘图区域并不一定是整个窗口，通过视口来指定绘图区域。

- 设置视口

✓OpenGL

glViewport()

✓OSG

osg::Camera::setViewport()

- 从3D空间坐标到2D屏幕坐标，只需要乘以MVPW就可以了。

✓MVPW = ViewMatrix * ProjectionMatrix * WindowMatrix

- 清楚了这三个矩阵，我们可以利用它做很多事情。

- 应用举例一、漫游

✓实时修改View Matrix的三个分量，可以实现在三维场景中漫游。

✓例如要在场景中移动，可以修改视点位置Eye；要拐弯了，修改参考点center；飞机飞得不平稳左右摇晃，修改Up；要加速，增加修改Eye的幅度；等等。

- 应用举例二、拾取

✓MVPM的逆变换；OSG中有一条直线与场景求交的函数，拾取就不用我们自己进行矩阵变换了。

- 应用举例三、HUD

✓在场景中，加一个平行投影的Camera；

✓把这个Camera放在最后渲染和绘制，并禁用掉深度检测。

- 应用举例四、在同一窗口，显示多个视图

✓将这多个视图安排到不同的视口中就OK了。

- 应用举例五、多通道

✓多渲染通道/显示通道是一个很大的研究课题。

✓OSG中的一个例子osgCluster简单的实现了多通道；它的原理是给每个显示通道不同的View Matrix；具体的讲就是每个通道的Eye和Up都一样，center却不一样；这样每个通道拼接起来就会有更宽广的视角，可以看到更广阔的区域。

三、场景图数据结构

- Scene Graph场景图

✓场景图是一种描述三维场景的数据结构。它是一个有向无循环图。

✓OSG中不仅定义了场景图的数据结构，还提供了对这种图数据结构的各种访问方式，或者说是管理方法，如渲染。

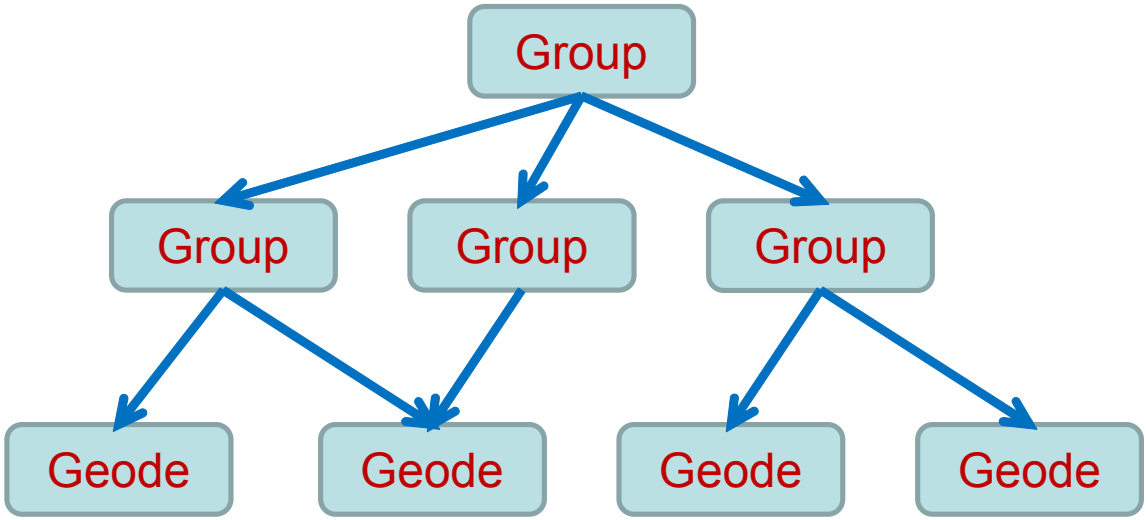
✓本节我们讨论OSG中的场景图数据结构，下一小节讨论渲染过程。

- OSG中的场景图结点osg::Node

✓组结点：osg::Group，有一个或多个子结点。

✓叶子结点：osg::Geode，没有子结点。

✓组结点和叶子结点都是由osg::Node继承而来。



- osg::Group

✓根据不同的用途，有各种不同的组结点，它们都继承自osg::Group。

✓如osg::LOD，可以根据距离远近等因素选择不同的子结点渲染。

✓如osg::Switch，可以在两个子结点中任选其一。

✓如osg::Sequence，可以构建序列动画。

✓如osg::Transform

- 一种组结点：osg::Transform

✓改变其所有子结点相对于场景中其它结点的坐标，可以是旋转、平移或缩放等。我们常用osg::MatrixTransform和osg::PositionAttitudeTransform。

✓它与View Matrix是同一层次的变换，在OpenGL中，同样使用glTranslate、glRotate和glScale完成，变换前同样要用glMatrixMode(GL_MODELVIEW)；它与View Matrix的区别在于它们考虑的角度不同；Tranform变换的是模型，VM变换的是观察者。

✓其实osg::Camera就是从osg::Transform继承来。

- osg::Geode

✓osg::Geode的实质是一组图元，下面有若干个osg::Drawable。

✓其中，osg::Drawable是一个简单的或复杂的图元，可以是一个正方形，也可以是栅格化的字符串。

✓osg::Geode典型示例：osg::Billboard标志牌。

- 包围盒与裁剪

✓场景图的层次特性，使得其裁剪操作很方便与高效。如果父结点被裁剪掉，子结点就不会再去考虑。

✓osg::Node得到的包围盒信息的形式是osg::BoundingSphere，而osg::Drawable得到的包围盒是osg::BoundingBox。

- 模型文件与场景图

✓模型文件中，保存了场景图的层次组织结构；例如文本格式的osg模型文件。

✓OSG从文件中加载模型时，会构建这样的一个场景图，并返回根节点的指针。

四、OSG的渲染过程

- 从最简单的osg程序看起：

```
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
int main(int argc, char** argv)
{
    osgViewer::Viewer viewer;
    viewer.setSceneData(osgDB::readNodeFile("…"));
    return viewer.run();
}
```

✓程序很容易看懂，main()的前两行很简单，先定义一个变量viewer，然后给其设置要渲染的模型数据。

✓看来所有的渲染过程全在viewer.run()上了。

- viewer.run()的实现

```
while(!done())
{
   frame();
}
```

更具体地展开就是：

```
while(!done())
{
   advance();
   eventTraversal();
   updateTraversal();
   renderingTraversals();
}
```

- advance()和eventTraversal()

✓advance()的作用是计时，增加帧计数。

✓eventTraversal()响应用户操作。

✓eventTraversal()遍历的是事件队列，而不是场景图；而且这个事件队列是OSG自己的Event Queue，并非操作系统的事件队列；

✓eventTraversal()与andvance()一起为更新遍历提供依据。

- 渲染过程中对场景图的三个遍历：

1、更新遍历

在updateTraversal()中实现，修改场景图，如渲染状态、结点参数等，以实现动态场景；更新摄像机。

2、裁剪遍历

在renderingTraversals()中实现，剔除不可见结点，将可见结点置入render graph中。

3、绘制遍历

在renderingTraversals()中实现，遍历render graph，向图形卡发送绘制命令；在线程中实现，可能没有执行完，主循环已进入下一轮。

- 渲染流程与回调函数callback

✓渲染过程的几个遍历，特别是事件遍历和更新遍历，我们要想按照我们的意图去响应事件和更新场景图，就要用到回调函数callback来参与到这些遍历中去。

✓其实这些遍历大部分工作就是执行回调函数的过程；即使是我们很难甚至无法参与的绘制函数，也会在绘制前和绘制后给我们执行回调函数的机会。

- 回调函数

✓osg::Node::setEventCallback

如从GUIEventHandler派生来的包括拾取在内的各种事件处理。

✓osg::Node::setUpdateCallback

如AnimationPathCallback，对应的结点按照一定的路线改变位置。

✓osg::Node::setCullCallback

✓osg::Camera::SetPreDrawCallback和 osg::Camera::SetPostDrawCallback

其中PostDrawCallback可能在下一轮的主循环中执行。

- osgGA::MatrixManipulator

✓这是一种专门用来控制摄像机的Callback；因为每个对每个漫游系统来说，摄像机控制器是必不可少的。

✓OSG为我们实现了一个轨迹球操作器，即osgGA:: TrackballManipulator；它像一般的event callback一样处理鼠标事件；只是它把鼠标左键理解为转动模型（或者说是摄像机在模型周围环绕观察），把鼠标右键理解为模型的缩放，把中键理解为模型的平移。

✓修改它，能实现我们自己的漫游方式（如方向盘控制、如CS游戏中的控制方式等等）。

- Node Visitor

✓基于访问者的设计模式，用来对相应的结点以各种方式进行遍历（遍历方式有只访问当前结点、遍历父结点、遍历子结点、遍历活动子结点等）。

✓最常见的最经典的NodeVisitor就是找到特定名字结点的那个Visitor了。

✓Visitor有很多种，如osgGA::EventVisitor、osgUtil::UpdateVisitor和osgUtil::CullVisitor，利用它们你可以控制渲染过程中的相应遍历；在你所设置的相应回调函数中，也将可以利用它们进行遍历。

谢谢大家！