# OSG学习笔记

## 1.初识osg,编写第一个程序

编译配置CMakeLists.txt

cmake\_minimum\_required(VERSION 3.14)

project(Study\_Osg)

set(CMAKE\_CXX\_STANDARD 14)

//找包

find\_package(OpenSceneGraph 3.2 REQUIRED COMPONENTS osgGA osgViewer osgDB osgUtil osg osgQt osgFX)

//查找当前目录下的所有源文件：命名为 OSGWidgets\_SRCS

aux\_source\_directory(. OSGWidgets\_SRCS)

//指定所有源文件生成动态链接库 OSGWidgets

add\_library(OSGWidgets SHARED ${OSGWidgets\_SRCS})

//配置库的头文件到链接库 OSGWidgets中

target\_include\_directories(OSGWidgets PUBLIC ${OPENSCENEGRAPH\_INCLUDE\_DIRS} )

//设置main.cpp为项目 Study\_Osg 的入口

add\_executable(Study\_Osg main.cpp)

//将库文件链接到项目中

target\_link\_libraries(Study\_Osg ${OPENSCENEGRAPH\_LIBRARIES})

编写main.cpp

//申请一个观察器（对象）该观察器可查看模型

osgViewer::Viewer viewer;

//设置观察器viewer中的数据，此处是：打开glider.osg的模型

viewer.setSceneData(osgDB::readNodeFile("glider.osg"));

//进行渲染前的最后一步，会检查和设置各种图像上下文、屏幕等信息。

viewer.realize();

//开始渲染工作

viewer.run();

## 2.改进程序

示例一：添加状态

上面只是简单的显示了osg的渲染界面，但是未添加任何功能，下面我们为这个程序添加如下功能：首先点击 s 键会显示帧速,点击 W 键会显示网格,点击 L 键灯光会开启等等以前都具有的功能。

编辑main.cpp如下：

osgViewer::Viewer viewer;

viewer.setSceneData(osgDB::readNodeFile("glider.osg"));

//添加事件句柄：即添加一个事件句柄，可理解为添加一个响应，鼠标或者是键///键盘。

viewer.addEventHandler(new osgGA::StateSetManipulator(viewer.getCamera()->getOrCreateStateSet()) );

//窗口大小变化事件，按f实现窗口大小的最小化和最大化。

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

//添加一些常用状态设置：响应 S 键,W 键等等

viewer.addEventHandler(new osgViewer::StatsHandler);

viewer.realize();

viewer.run();

实现功能：s 键可显示帧数 f键可切换窗口大小，但是w键和l键功能没有实现？？？

示例二：设置操作器

上面讲述了如何在一个 viewer 当中添加状态,但是显然这个操作器还是不那么如人意,我们需要多一些操作器,而且在早期的版本当中,会有点击 Z 记录路径再播放什么的,而且点击 H 键会显示出来帮助文档,这里我们再把代码修改一下:

osgViewer::Viewer viewer;

viewer.setSceneData(osgDB::readNodeFile("glider.osg"));

viewer.addEventHandler(new osgGA::StateSetManipulator(viewer.getCamera()->getOrCreateStateSet()));

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.addEventHandler(new osgViewer::StatsHandler);

//添加操作器

{

//申请一个使用按键来换操作器的类

osg::ref\_ptr<osgGA::KeySwitchMatrixManipulator> keyswitchManipulator = new osgGA::KeySwitchMatrixManipulator;

//向类中添加操作器

keyswitchManipulator->addMatrixManipulator( '1', "Trackball", new osgGA::TrackballManipulator());

keyswitchManipulator->addMatrixManipulator( '2', "Flight", new osgGA::FlightManipulator());

keyswitchManipulator->addMatrixManipulator( '3', "Drive", new osgGA::DriveManipulator());

keyswitchManipulator->addMatrixManipulator( '4', "Terrain", new osgGA::TerrainManipulator());

//把路径添加到viewer中，类似于状态

viewer.setCameraManipulator(keyswitchManipulator.get() );

}

//添加路径记录

viewer.addEventHandler(new osgViewer::RecordCameraPathHandler);

viewer.realize();

viewer.run();

//运行后,比之以前键下 2 或者 3 键可以发现可以换操作器了,先按 Z 再换过来按小写 Z,发现可以记录路径

//了,而且在文件夹里还为我们生成了一个路径文件:saved\_animation.path 真是非常非常的方便。注意我们

//添加了一些头文件。

## 3.读取模型

### 1.添加模型

示例三：添加模型（在场景中显示多个模型）

在 OSG 当中模型是使用 osg::Group 和 osg::Node 来装载在一起的,比如同时需要加入两个模型,模型 A 了模型 B, AB 各自是一个 NODE,那么可以使用以下语句来做到,首先使用一个 Group,然后 Group ->addChild(A),同样,之后要 Group->addChild(B)。然后再把 Group 添加到 viewer 当中就可以了。

osg::ref\_ptr<osgViewer::Viewer> viewer=new

osgViewer::Viewer();

//申请一个指向根结点的智能指针

osg::ref\_ptr<osg::Group> root= new osg::Group();

//设置两个node结点

osg::ref\_ptr<osg::Node>node1=osgDB::readNodeFile("glider.osg");

osg::ref\_ptr<osg::Node>node2=osgDB::readNodeFile("osgcool.osgt");

//将两个node结点设置为root的子节结点

root->addChild(node1.get());

root→addChild(node2.get());

//设置场景数据

viewer->setSceneData(root.get());

viewer->realize();

viewer->run();

下面还介绍一种简化写法：

osgViewer::Viewer viewer;

//申请一个root结点

osg::ref\_ptr<osg::Group> root=new osg::Group();

//向结点中加入两个子结点

root->addChild(osgDB::readNodeFile("glider.osg"));

root->addChild(osgDB::readNodeFile("osgcool.osgt"));

//设置场景数据

viewer.setSceneData(root);

viewer.realize();

viewer.run();

### 2.删除结点

如果我们不需要某个结点了,比如我们看那个小飞机很不爽,我们想把它从场景中删除掉。不知道出于某种目的,反正现在要删除掉,可能是开始想看见它现在不想看见它了。可以通过 removeChild 方法,删除多个孩子也可以通过 removeChildren 方法,里面的参数有些需要索引值,有些需要结点本身的指针。

这里要提醒大家的是,如果要删除一个结点,那么该结点下的所有结点都会被删除。如果一个结点被加入到一个组中两次,那么这两次是分别存在的,删除一次还有另一次。删除操作不能说不是个危险的操作,有些时候,尤其在有移动结点等等混在一起时,删除操作有时候会发生一些比较奇怪的现象。

在内存映象当中,如果一个模型被读取一次,而用了多次,那么所占用的空间是不会改变的。

### 3.隐藏模型

如果说我们只是想让模型暂是的消失,我想它出来它还要出来,如果删除则它会消失在内存中,再调出来

会花费很长的时间,这样就得不偿失了,于是我们想起了用隐藏的方法。

隐藏模型其实模型仍在渲染当中,因此损耗并未减少,只不过隐藏了而已,隐藏的确不是个什么好操作，但是有时候对小模型确实也很实用。node ->setNodeMask 可以设置隐藏与显示。

示例四：隐藏指定模型

osg::ref\_ptr<osgViewer::Viewer> viewer=new osgViewer::Viewer();

osg::ref\_ptr<osg::Group> root= new osg::Group();

osg::ref\_ptr<osg::Node> node1=osgDB::readNodeFile("glider.osg");

osg::ref\_ptr<osg::Node> node2=osgDB::readNodeFile("osgcool.osgt");

root->addChild(node1.get());

root→addChild(node2.get());

//隐藏node结点的osg模型 隐藏：0x0 显示：1

node2->setNodeMask(0x0);

viewer->setSceneData(root.get());

viewer->realize();

viewer->run();

如果我们不需要某个结点了,比如我们想把飞机从场景中去掉,应该怎么做呢?

第一种方式:隐藏结点

第二种方式：删除结点

第三种方式：在 OSG 当中,专门有一个类来负责打开与关闭结点,该类名为 osg::Switch,里面有相应的方法来控制它所管理的结点的打开与关闭。

总的来说,这后面两种方法都能控制模型的显示和隐藏,区别在于隐藏模型方法不会让模型在内存中消失,这样对于小的物体频繁的调用会节省一些时间,而对于有些大的模块在用一次以后可能很久再用第二次,这个时候用节点开关可以将模型销毁,再次使用再调入内存,以防止占用更多的资源。

### 4.结点开关

示例五：结点开关

功能:打开或关闭一个/些结点,而在关闭时这些结点占用的内存会被释放掉。

osgViewer::Viewer viewer;

osg::Group \*root=new osg::Group();

//申请一个指向Switch类的指针

osg::Switch \*sw=new osg::Switch();

osg::Node \*node=osgDB::readNodeFile("osgcool.osgt");

//向该结点中添加两个子结点，其中false表示关闭 开关 true表示打开 开关

sw->addChild(node,false);

sw→addChild(osgDB::readNodeFile("glider.osg"));

//把开关添加到根结点中

root->addChild(sw);

viewer.setSceneData(root);

viewer.realize();

viewer.run();

## 4.超级指针

在程序中创建对象的一般方法可能如下:

osg::Node\* anotherNode = new osg::Node();

group->addChild(anotherNode);

这种方法存在问题,比如如果在创建对象之后发生了错误,抛出了异常,谁来释放 Node所申请的资源呢?又或者节点从场景中被去掉后,这部分的资源何时被正确释放掉呢?

在 osg 中,创建一个 Node 的一般方法如下:

osg::ref\_ptr<osg::Node> aNode = new osg::Node();

group->addChild(aNode.get());

模板类 osg::ref\_ptr 智能指针其实就是引用一个计数器,这个计数器会计算这个节点被引用的次数,被别人引用一次这个计数器增加一,别人不用一次,即:释放一次,则计数器减一。当减至 0 时,内存放掉不用。new osg::Node()时我们申请了一个 Node 的资源,这时在堆内引用该 Node的计算器会被置 1。在 group->addChild(aNode.get())时又引用了一次,会再加 1。而在这两次引用都结束时,Node 的资源就会被自动释放。

## 5.移动/旋转/缩放模型

对模型最常用的操作莫过于移动了,有很多人不知道如何操作来移动一个模型。现在我们要明确一个事例,也就是说 OSG 把加入的模型默认都放在中点了,这非常的不好受。其实有些时候并不建议在场景中移动模型,替代的是模型在未加入场景之前位置就是完好的。

移动/旋转/缩放其实都是对矩阵进行操作,在 OSG 当中,矩阵可以当作一个特殊的结点加入到 root 当中,而矩阵下也可以另入结点,而加入的结点就会被这个矩阵处理过,比如移动过/旋转过/缩放过。在 OSG 中控制矩阵的类可以为 osg::MatrixTransform。

而相应的方法就是:

移动——osg::Matrix::translate

旋转——osg::Matrix::rotate

缩放——osg::Matrix::scale

示例六：移动/旋转/缩放模型

功能：移动/旋转/缩放模型,这里加入了四个 coolosg,一个是默认加入在最中间,一个向上移 2 单位,一个是向下移 2 单位且缩放 0.5 倍,另一个是向右 4 单位,缩放 0.5 且平躺 45 度。

osgViewer::Viewer viewer;

osg::ref\_ptr<osg::Group> root=new osg::Group();

osg::ref\_ptr<osg::Node> osgcool=osgDB::readNodeFile("/home/zhihui/OSGlib/OSG/OpenSceneGraph-Data-master/osgcool.osgt");

//移动模型（向上移动2个单位）三个参数：x轴、向里的y轴、向上的z轴

osg::ref\_ptr<osg::MatrixTransform> trans=new osg::MatrixTransform;

trans->setMatrix(osg::Matrix::translate(0,0,2));

trans->addChild(osgcool.get());

//缩放模型（向下移动2个单位 并且 缩放0.5倍）三个参数：x轴、向里的y轴、向上的z轴

osg::ref\_ptr<osg::MatrixTransform> scale=new osg::MatrixTransform;

scale->setMatrix(osg::Matrix::scale(0.5,0.5,0.5)\*osg::Matrix::translate(0,0,-2));

scale->addChild(osgcool.get());

//旋转模型 （向右移动4个单位、向下移动2个单位。缩放0.5倍。平躺45°）四个参数：？？？？

osg::ref\_ptr<osg::MatrixTransform> rot=new osg::MatrixTransform;

rot->setMatrix(osg::Matrix::rotate(osg::DegreesToRadians(45.0),1,0,0)\*osg::Matrix::scale(0.5,0.5,0.5)\*osg::Matrix::translate(4,0,-2));

rot->addChild(osgcool.get());

root->addChild(osgcool.get());

root->addChild(trans.get());

root->addChild(scale.get());

root->addChild(rot.get());

//为了方便看，我们在场景中加入了窗口缩放

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root.get());

viewer.realize();

viewer.run();

注：多个设置用\*号连接，但是它们不满足交换律（不同的位置摆放 会有不同的效果）。

## 6.基本几何图元

接下来我们介绍一下如何在 Osg 中绘制一些我们自定义的对象,并且将他们加载到节

点树中去,显示在合适的位置。

在 OSG 当中所有的加入到场景的数据都会加入到一个 Group 当中,而几何图元做为一种需要绘制的对象而非模型也有一个类专门组织,这个类就是 osg::Geode,该类的作用就是组织一些绘制的对象,然后传给 Group。

所有基本的绘制功能完成后都会把结点汇集在一个 osg::Geode当中,然后该 osg::Geode 再自己加入到 root 中完成所有子结点的绘制。

### 1.基本绘制方法

下面我们要绘制一个正方形,绘制有色彩,未贴图。首先我们必须要申请一个 osg::Geometry,把这个 Geometry 加入到 Geode 就可以了。在这个 Geometry 中要设置一些元素,最基本的是顶点 Vertex,颜色 color,以及顶点的关联方式和法线 normal.就可以了。

示例七：绘制正方形

main函数编辑如下：

osgViewer::Viewer viewer;

osg::Group \*root= new osg::Group;

root->addChild(createQuad().get());

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root);

viewer.realize();

viewer.run();

createQuad函数编辑如下：

//绘制该正方体的代码都在这个函数内，该函数返回一个osg::Node类型的结点

osg::ref\_ptr <osg::Node> createQuad()

{

//申请一个几何体结点

osg::ref\_ptr<osg::Geometry> geom=new osg::Geometry;

//定义顶点序列（存放4个顶点）

//申请一个顶点序列，顶点放在vector容器中

osg::ref\_ptr<osg::Vec3Array> v=new osg::Vec3Array;

//把顶点序列加入到几何体中

geom->setVertexArray(v.get());

//定义4个顶点加入到顶点序列中

v->push\_back(osg::Vec3(-1.f, 0.f, -1.f));

v->push\_back(osg::Vec3(1.f, 0.f, -1.f ));

v->push\_back(osg::Vec3(1.f, 0.f, 1.f ));

v->push\_back(osg::Vec3(-1.f, 0.f, 1.f ));

//定义颜色数组

//申请一个颜色序列数组，并且与几何体中进行关联,设置绑定方式为逐点绑定。颜色数组的四个参数分别为 RGBA。

osg::ref\_ptr <osg::Vec4Array> c=new osg::Vec4Array;

geom->setColorArray(c.get());

geom->setColorBinding(osg::Geometry::*BIND\_PER\_VERTEX*);

c->push\_back(osg::Vec4(1.f, 0.f, 0.f, 1.f ));

c->push\_back(osg::Vec4(0.f, 1.f, 0.f, 1.f ));

c->push\_back(osg::Vec4(0.f, 0.f, 1.f, 1.f ));

c->push\_back(osg::Vec4(1.f, 1.f, 1.f, 1.f ));

//定义法线

//申请一个法线绑定到该四方体中。法线为指向 Y 轴负半轴。法线不同会影响光照的效果。

osg::ref\_ptr<osg::Vec3Array> n=new osg::Vec3Array;

geom->setNormalArray(n.get());

geom->setNormalBinding(osg::Geometry::*BIND\_OVERALL*);

n->push\_back(osg::Vec3(0.f,-1.f,0.f));

//设置顶点关联方式

geom->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::*QUADS*,0,4));

//几何组结点

//申请一个几何组结点，并把几何体结点加入到几何组结点当中

osg::ref\_ptr<osg::Geode> geode=new osg::Geode;

geode->addDrawable(geom.get());

//返回该组结点，可以被加入到Group中进行渲染

return geode.get();

}

### 2.所有可绘制图元

下面例举所有可绘制的几何图元:POINTS[点],LINES[线],LINE\_STRIP[线带],LINE\_LOOP[闭合线段],TRIANGLES[三角形],TRIANGLE\_STRIP[三角带],TRIANGLE\_FAN[三角扇],QUADS[四方块],QUAD\_STRIP[四方块带],POLYGON[多边形],在这里我们要以封闭的直线为例来说明如何设置线宽属性。所有的图元可能都有它的特殊性,但是大都与此种设置方法类似。

在 OSG 中设置直线线宽的专门有一个函数来管理,叫做 LineWidth,它本身属于状态与属性类别中的类。事实上也是从那里派生而来。所有设置状态的操作都与此类似。

示例八：绘制线宽

功能：控制线的宽度

main函数编辑如下：

osgViewer::Viewer viewer;

osg::Group \*root= new osg::Group;

root->addChild(createLine().get());

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root);

viewer.realize();

viewer.run();

createLine函数编辑如下：

osg::ref\_ptr<osg::Node> createLine()

{

osg::ref\_ptr<osg::Geometry> geom=new osg::Geometry;

//限制线的宽度线宽是一种属性和状态,在 OSG 当中有专门的属性和状态类来管理这些。

//所有启动属性和状态的操作使用都与此类似。

osg::ref\_ptr<osg::LineWidth> LineSize=new osg::LineWidth;

LineSize->setWidth(10.0);

geom->getOrCreateStateSet()->setAttributeAndModes(LineSize.get(),osg::StateAttribute::*ON*);

//首先定义四个点

osg::ref\_ptr<osg::Vec3Array> v = new osg::Vec3Array;

geom->setVertexArray(v.get() );

v->push\_back(osg::Vec3( -1.f, 0.f, -1.f ) );

v->push\_back(osg::Vec3( 1.f, 0.f, -1.f ) );

v->push\_back(osg::Vec3( 1.f, 0.f, 1.f ) );

v->push\_back(osg::Vec3( -1.f, 0.f, 1.f ) );

//定义颜色数组

osg::ref\_ptr<osg::Vec4Array> c = new osg::Vec4Array;

geom->setColorArray(c.get() );

geom->setColorBinding( osg::Geometry::*BIND\_PER\_VERTEX* );

c->push\_back(osg::Vec4( 1.f, 0.f, 0.f, 1.f ) );

c->push\_back(osg::Vec4( 0.f, 1.f, 0.f, 1.f ) );

c->push\_back(osg::Vec4( 0.f, 0.f, 1.f, 1.f ) );

c->push\_back(osg::Vec4( 1.f, 1.f, 1.f, 1.f ) );

//定义法线

osg::ref\_ptr<osg::Vec3Array> n = new osg::Vec3Array;

geom->setNormalArray( n.get() );

geom->setNormalBinding( osg::Geometry::*BIND\_OVERALL* );

n->push\_back(osg::Vec3( 0.f, -1.f, 0.f ) );

//设置顶点关联方式,这里把顶点关联方式改成了：　LINE\_LOOP

geom->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::*LINE\_LOOP*, 0, 4 ) );

//几何组结点

osg::ref\_ptr<osg::Geode> geode = new osg::Geode;

geode->addDrawable( geom.get() );

return geode.get();

}

### 3.内置几何类型

如同 OpenGL 一样,OSG 同样有一套内置几何类型,这些几何类型都在类 osg::Shape 中,这些 shape 本身都可以本当成一个 Draw 结点加入到 geode 中,然后再扔到geode 中添加到 root 里进行渲染。

形状共有九种,分别为:osg::Box[盒子],osg::Capsule[胶囊形],osg::CompositeShape[组合型],osg::Cone[圆锥形],osg::Cylinder[圆柱形],osg::HeightField[高程形],osg::InfinitePlane[有限面],osg::Sphere[球形],osg::TriangleMesh[三角蒙皮]。

一般的形状态都有特定的因素,比如 Box 有长宽,圆有半径,以及各个图形所画的精细度都需要指明,这些精细度在球这样的形状上意义还是十分巨大的。在 OSG 中有专门指明精细度的类,名为:osg::TessellationHints。以球为例,只需要规定,圆心,半径和精细度就可以画出该球。

示例九：内置几何类型

功能：绘制基本几何类型

main函数编辑如下：

osgViewer::Viewer viewer;

osg::Group \*root= new osg::Group;

root->addChild(createShape().get());

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root);

viewer.realize();

viewer.run();

createShape函数编辑如下：

osg::ref\_ptr<osg::Node> createShape()

{

//申请一个几何组结点　来装载几何体

osg::ref\_ptr<osg::Geode> geode=new osg::Geode;

//申请两个变量表示几何体要使用的半径和高度

float radius=0.08f;

float height=0.1f;

//表示模型的粗细度，值越大精细度就越大。

osg::TessellationHints \*hints=new osg::TessellationHints;

hints->setDetailRatio(0.5f);

//分别是申请球结点:规定圆心、半径、精细度;

//盒子结点:规定盒心、长、精细度;

//圆锥结点:规定底面圆心、半径、高度、精细度;

//圆柱结点:规定底面圆心、半径、高度、精细度;

//胶囊体结点:规定半径、圆心、高度、精细度。

geode->addDrawable(new osg::ShapeDrawable(new osg::Sphere(osg::Vec3(0.0f,0.0f,0.0f),radius),hints));

geode->addDrawable(new osg::ShapeDrawable(new osg::Box(osg::Vec3(0.2f,0.0f,0.0f),2\*radius ),hints));

geode->addDrawable(new osg::ShapeDrawable(new osg::Cone(osg::Vec3(0.4f,0.0f,0.0f),radius ,height),hints));

geode->addDrawable(new osg::ShapeDrawable(new osg::Cylinder(osg::Vec3(0.60f,0.0f,0.0f),radius ,height),hints));

geode->addDrawable(new osg::ShapeDrawable(new osg::Capsule(osg::Vec3(0.80f,0.0f,0.0f),radius ,height),hints));

return geode.get();

}

## ７.交互

介绍如何响应键盘上的一些键,以及响应鼠标的一些操作,至于响应它做什么则不是我们主要考虑的事情。

### 1.交互过程

在这里首先介绍一下交互的原理。我们在平时操作的主要是场景的数据,而场景中的数据主要由类 viewer 负责进行渲染。要得到数据可以使用 viewer ->getSceneData()就可以了。当然也可以把数据指针设置成全局的。

交互主要是针对鼠标与键盘的响应来对应数据的改变。viewer 的主要的功能是控制场景,它是场景的核心类,如果能响应键盘时得到 viewer,那么也可以从键盘的响应中控制整个场景。

viewer 中有一个方法,名为 addEventHandler 就是专门做这件事情的。他会加入一个事件处理器。于是我们就想,一定要自己写一个事件处理器才行,这就必须要了解事件处理器的格式,只要有一个接口就可以了解它的格式,这个接口就是: osgGA::GUIEventHandler,于是我们可以写一个类 A 从该类公有派生出来,即:class A : public osgGA::GUIEventHandler, 在 里 面 处 理 好 各 种 操 作 然 后 加 入 到 viewer 当中 , 即 :viewer.addEventHadler(new A(里面可以有参数)) ;这样就可以完成操作。

viewer

addEvent(A)

Class A:public osgGA

getEventType

判断事件类型

getViewer

操作

事件类型有很多,一个 viewer 也可以响应 ABCD 等等多个事件。一个事件也可以拦截事件,也就是 A 处理完后可以让 BCD 处理,也要以拦截住不让他们多事。

另一个就是,也可以给事件操作器 A 传递参数,不必非要使用 viewer 进行处理。使用 viewer 只是让大家了解一下,可以从里面得到 viewer,这样就无所不能了。也可以把 viewer 当参数传进去,方法很多。

### 2.响应鼠标键盘事件

示例十：响应键盘鼠标  
功能：鼠标右键单击时牛和飞机都隐藏,鼠标左键双击时牛和飞机都显示,按键盘上面的 LEFT 键,显示牛,

按键盘上面的 RIGHT 键显示飞机。其中显示与隐藏结点我们使用的是 setNodeMask(bool)。

申明一个类 UseEventHandler

// 申明一个类　UseEventHandler　它是由　osgGA::GUIEventHandler　派生而来的拥有处理事件的能力,其中这种能力体现在一个虚函数 handle 上。一切的处理都在 handle 当中。

class UseEventHandler:public osgGA::GUIEventHandler

{

public:

//两个极其重要的参数,一个是 const osgGA::GUIEventAdapter,注意有不少人在

//写时少写了个 const 结果就不灵了,因为是虚函数,参数必须与父类中虚函数相应一致。该参数是用来

//识别各种事件类型的参数。第二个参数也很重要:osgGA::GUIActionAdapter,它是控制显示的参数,其

//实最重要的是它是 Viewer 的祖父类,由它可以得到 viewer,在操作中有体现。

virtual bool handle(const osgGA::GUIEventAdapter& ea, osgGA::GUIActionAdapter& aa) {

//由参数二 osgGA::GUIActionAdapter 得到了 viewer。以下这两句非常常用

osgViewer::Viewer\* viewer = dynamic\_cast<osgViewer::Viewer\*>(&aa);

//viewer未得到就返回false

if (!viewer) return false;

//判断事件的类型　　Child0　是飞机 Child１　是牛

// setNodeMask(0) 是隐藏　setNodeMask(1)是显示

switch (ea.getEventType())

{

//如果是有键按下

case osgGA::GUIEventAdapter::*KEYDOWN*:

{

//如果是键盘左键　显示　Child１　隐藏　Child0

if (ea .getKey()== 0xFF51)

{

viewer->getSceneData()->asGroup()->getChild(0)->setNodeMask(1) ;

viewer->getSceneData()->asGroup()->getChild(1)->setNodeMask(0) ;

}

//如果是键盘右键　显示　Child0　隐藏　Child１

if(ea.getKey()== 0xFF53)

{

viewer ->getSceneData() ->asGroup() ->getChild(0) ->setNodeMask(0) ;

viewer ->getSceneData() ->asGroup() ->getChild(1) ->setNodeMask(1) ;

}

}

break;

//如果是鼠标push如果事件发生在鼠标上就需要判断是哪个键,

//一般视 1 代表左键,2 代表中键,4 代表右键。

case osgGA::GUIEventAdapter ::*PUSH*:

//如果是鼠标右键单击　则隐藏　　Child0 和 Child1

if(ea.getButton () == 4)

{

viewer ->getSceneData() ->asGroup() ->getChild(0) ->setNodeMask(0) ;

viewer ->getSceneData() ->asGroup() ->getChild(1) ->setNodeMask(0) ;

}

break;

//没实现？？？

case osgGA::GUIEventAdapter::*DOUBLECLICK*:

if(ea.getButton() == 1)

{

viewer ->getSceneData() ->asGroup() ->getChild(0) ->setNodeMask(1) ;

viewer ->getSceneData() ->asGroup() ->getChild(1) ->setNodeMask(1) ;

}

break;

default:

break;

}

return false;

}

main函数编辑如下：

osgViewer::Viewer viewer;

osg::ref\_ptr<osg::Group> root = new osg::Group() ;

root ->addChild(osgDB::readNodeFile("cow.osg")) ;

root ->addChild(osgDB::readNodeFile("glider.osg"));

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root.get());

//添加一个 UseEventHandler类事件

viewer.addEventHandler(new UseEventHandler);

viewer.realize();

viewer.run();

### 3.pick

pick 主要是通过鼠标的点击来拾取一些物体,或者判断鼠标所点击的位置在哪里。

在 OSG 中 pick 首先肯定需要响应鼠标单击事件,在上一节中可以放在case　osgGA::GUIEventAdapter ::*PUSH*:下,如果是左键单击[左手鼠标的可能不同],那么直接传入函数 pick 当中,pick 本身并不是一个 OSG 的库函数。pick 有两个非常重要的参数,即:点击时的屏幕位置,在 pick 中由这个屏幕上的点发射线到场景中与场景中的物体相交,可能与多个物体相交,然后逐一判断与哪些物体相交。可以得到这些物体的结点,然后另以区别它们。

在开始加入物体结点时区分可以使用 setName 等等方法,如果类型本身不同则不需要这么麻烦直接指针就可以判断出来。

鼠标单击

handle

判断类型为push

获得屏幕上单击位置传入

pick(x,y)中

射线与场景中物体相交

可以判断是哪个物体，也可以得到坐标

需要解决的问题：

１.如何判断射线与 viewer 中物体相交?

2.如何判断相交结点为我想要的那个结点?

解决方法：

１.发出射线并相交。在 OSG 中有库函数,osgViewer::View::computeIntersections 他共有三个参

数:第一个是 x 屏幕坐标,第二个是 Y 屏幕坐标,第三个是存放被交的结点以及相交的坐标结点路径等等相关信息。

2.由于有存放相交射线与场景的结果集,那么只需要判断该结果集中有没有要用的结点就可以了。

示例十一：pick

功能:场景是一飞机和一头加白边的牛。点击牛后牛白边会消失,为了简单起见没有做成单击消失,

单击再出来这种效果。给物体加白边用的是 osgFX::Scribe 类,它是专门做这个的。要加边只要把结点加入到它下面,然后它再加到原先该结点要加的结点下面就加边了,看示例代码就清楚了。

main函数编辑如下：

osgViewer::Viewer viewer;

osg::ref\_ptr<osg::Group> root = new osg::Group() ;

root ->addChild(osgDB::readNodeFile("cessna.osg"));

osg::ref\_ptr<osg::Node> cow=osgDB::readNodeFile("cow.osg");

//添加一个 scribe 结点,该结点下的模型会被加白描线高亮显示。

osg::ref\_ptr<osgFX::Scribe> sc=new osgFX::Scribe;

//将 cow 模型添加到 sc 结点下

sc->addChild(cow.get());

root->addChild(cow.get());

root->addChild(sc.get());

viewer.addEventHandler(new osgViewer::WindowSizeHandler);

viewer.setSceneData(root.get());

viewer.addEventHandler(new CPickHandler(&viewer));

viewer.realize();

viewer.run();

CpickHandler 类编辑如下：

class CPickHandler: public osgGA::GUIEventHandler

{

public:

//构造函数中有参数　osgViewer::Viewer \*　,这样就可以手动把viewer传入类中了

CPickHandler(osgViewer::Viewer \*viewer):mViewer(viewer){}

virtual bool handle(const osgGA::GUIEventAdapter& ea, osgGA::GUIActionAdapter& aa)

{

switch (ea.getEventType())

{

//判断事件为左键单击

case osgGA::GUIEventAdapter::*PUSH*:

if(ea.getButton()==1)

{

//把点击屏幕时的X，Y坐标，传入到Pick函数中去

Pick(ea.getX(),ea.getY());

}

return true;

}

return false;

}

protected:

void Pick(float x,float y)

{

//申请一个相交测试的结果集，判断屏幕与场景相交后，把得出的结果集放入其中

osgUtil::LineSegmentIntersector::Intersections intersections;

//测试屏幕与场景是否相交，相交则放入

if(mViewer->computeIntersections(x, y, intersections))

{

//遍历该结果集

for(osgUtil::LineSegmentIntersector::Intersections::iterator hitr=intersections.begin();hitr!=intersections.end();++hitr)

{

// 遍历器的结果不为空 而且 nodePath 也不为空

if (!hitr->nodePath.empty() && !(hitr->nodePath.back()->getName().empty()))

{

//对 NodePath 进行遍历 找到我们需要的结点

osg::NodePath np=hitr->nodePath;

for(int i=np.size()-1;i>0;--i)

{

//如果结果集中有所需要的结点,则设置隐藏该结点。其中有一个动态转换,如果可以转换成功则左值不为 NULL,否则为 NULL。

osgFX::Scribe \*sc= dynamic\_cast<osgFX::Scribe \*> (np[i]);

if(sc!=NULL)

{

if (sc->getNodeMask()!=0)

{

sc->setNodeMask(0);

}

}

}

}

}

}

}

osgViewer::Viewer \*mViewer;

};

## 查漏补缺

### 1.dynamic\_cast

功能：将一个基类对象指针（或引用）cast到继承类指针，dynamic\_cast会根据基类指针是否真正指向继承类指针来做相应处理.

用法：dynamic\_cast <type-id> ([expression](https://baike.baidu.com/item/expression))

该运算符把expression转换成type-id类型的对象。Type-id 必须是类的[指针](https://baike.baidu.com/item/指针)、类的引用或者[void](https://baike.baidu.com/item/void)\*；

如果 type-id 是类指针类型，那么expression也必须是一个指针，如果 type-id 是一个引用，那么 expression 也必须是一个引用。

dynamic\_cast[运算符](https://baike.baidu.com/item/运算符)可以在执行期决定真正的类型。如果 downcast 是安全的（也就说，如果基类指针或者引用确实指向一个[派生类](https://baike.baidu.com/item/派生类)对象）这个运算符会传回适当转型过的指针。如果 downcast 不安全，这个运算符会传回空[指针](https://baike.baidu.com/item/指针)（也就是说，基类指针或者引用没有指向一个派生类对象）。

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

### 2.osgViewer::CompositeViewer

osgViewer::CompositeViewer是一个osgViewer::View序列，包含多个osgViewer::View。

### 3.osg::Geode

osg中的结点之一，Geode作为叶节点，用于保存几何信息以便渲染。

### 4.osg::Vec3Array

三元向量数组，继承自STL的vector 类，我们可以使用 push\_back 方法来添加数组成员

### 5.osg::Switch

Switch是一个管理类，它下面可以有很多子类，它就像是一个开关可以控制子类的显示与隐藏，而这种隐藏是不耗费内存的，

与SetNodeMask对比有明显的优势，SetNodeMask事实上只是隐藏了结点，但结点仍被绘制。

该类是从Node 和Group 派生而来，故可以被加入组结点以及asGroup 等等，是一个非常常用的类。