OSG 开源教程

整理：荣明、王伟

北 京

2008 年 4 月

序

第一次接触 OSG 是在 2001 年，当时开源社区刚刚兴起，还没有现在这么火。下载了 OSG 源码，但是在看了几个 Demo 之后，感觉没有什么特别之处。时隔七年之后，我再次将目光投向 OSG，发现 OSG 确实有其独到之处，很多 3D 效果已经不弱于甚至超过商业软件，有感于开源力量的巨大。但是，与当前主流 3D 商业软件如 Vega、VegaPrime、VTree、Performer 等相比，开源软件的缺点也很明显，其中文档缺乏可谓其致命弱点之一。开发者只能从浩瀚的源码中，进行编程的学习，效率不高。

OSG 组织也意识到这一点，不断推出一些官方教程，网络上有很多 OSG 爱好者发布的心得和教程。我收集、整理了许多学习资料，其中美国海军研究生院发布的 OSG 教程非常好，可作为 OSG 的官方教程的一个很好补充。它共有十一个专题，结合例子程序，一步步教你如何进行 OSG 的开发。我将其编辑成一个较为完善的教材，供大家学习。在教材整理过程中，王伟调试了源程序，对该书编辑和修订做了大量的工作；实验室的学生杨宇蒙、冯锐、章仕锋、李永川和李益强阅读了本书，并提出了宝贵的修改意见，在此一并表示感谢。

希望这本小册子能对大家学习 OSG 编程起到一定的帮助作用。

谢谢大家阅读本书！

荣　明

二 00 八年四月于崔春园

Email：rong_ming@sina.com

（本书的资料全部来自互联网，仅供个人学习交流使用。感谢竹林小舍、array 翻译了 Navy 的教程。）

目录

1.使用 Open Scene Graph 几何

本节涵盖了生成基本几何形状的一些方法。生成几何物体的方法有这么几种：在最底层对 OpenGL 基本几何进行松散的包装，中级是使用 Open Scene Graph 的基本形状，以及更高级一些的从文件读取。这篇教程涵盖的是最低层的。这种方法弹性最大但最费力。通常在 Scene Graph 级别，几何形状是从文件加载的。文件加载器完成了跟踪顶点的大部分工作。

1.1 背景

对一下几个类的简单解释：

Geode 类：

geode 类继承自 node 类。在一个 Scene Graph 中，node（当然包含 geode）可以作为叶子节点。Geode 实例可以有多个相关的 drawable。

Drawable 类层次：

基类 drawable 是一个有六个具体子类的抽象类。

geometry 类可以直接有 vertex 和 vertex 数据，或者任意个 primitiveSet 实例。

vertex 和 vertex 属性数据（颜色、法线、纹理坐标）存放在数组中。既然多个顶点可以共享相同的颜色、法线或纹理坐标，那么数组索引就可以用来将顶点数组映射到颜色、法线、或纹理坐标数组。

PrimitiveSet 类：

这个类松散的包装了 OpenGL 的基本图形－POINTS,LINES,LINE_STRIP,LINE_LOOP,...,POLYGON.

1.2 代码

以下这节代码安装了一个 viewer 来观察我们创建的场景,一个‘group’实例作为 scene graph 的根节点，一个几何节点（geode）来收集 drawable，和一个 geometry 实例来关联顶点和顶点数据。（这个例子中渲染的形状是一个四面体）

```
...
int main()
{
...
osgProducer::Viewer viewer;
osg::Group* root = new osg::Group();
osg::Geode* pyramidGeode = new osg::Geode();
osg::Geometry* pyramidGeometry = new osg::Geometry();
```

下一步，需要将锥体 geometry 和锥体 geode 关联起来，并将 pyramid geode 加到 scene graph 的根节点上。

```
pyramidGeode->addDrawable(pyramidGeometry);
root->addChild(pyramidGeode);
```

声明一个顶点数组。每个顶点由一个三元组表示——vec3 类的实例。这些三元组用 osg::Vec3Array 类的实例存贮。既然 osg::Vec3Array 继承自 STL 的 vector 类，那么我们就可以使用 push_back 方法来添加数组成员。push_back 将元素加到向量的尾端，因此第一个元素的索引是 0，第二个是 1，依此类推。

使用‘z’轴向上的右手坐标系系统，下面的 0...4 数组元素代表着产生一个简单锥体所需的 5 个点。

```
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
pyramidVertices->push_back( osg::Vec3( 0, 0, 0) ); // front left
pyramidVertices->push_back( osg::Vec3(10, 0, 0) ); // front right
pyramidVertices->push_back( osg::Vec3(10,10, 0) ); // back right
pyramidVertices->push_back( osg::Vec3( 0,10, 0) ); // back left
pyramidVertices->push_back( osg::Vec3( 5, 5,10) ); // peak
```

将这个顶点集合和与我们加到场景中的 geode 相关的 geometry 关联起来。

```
pyramidGeometry->setVertexArray( pyramidVertices );
```

下一步，产生一个基本集合并将其加入到 pyramid geometry 中。使用 pyramid 的前四个点通过 DrawElementsUint 类的实例来定义基座。这个类也继承自 STL 的 vector，所以 push_back 方法会顺序添加元素。为了保证合适的背面剔除，顶点的顺序应当是逆时针方向的。构造器的参数是基本的枚举类型（和 opengl 的基本枚举类型一致），和起始的顶点数组索引。

```
osg::DrawElementsUInt* pyramidBase =
new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
pyramidBase->push_back(3);
pyramidBase->push_back(2);
pyramidBase->push_back(1);
pyramidBase->push_back(0);
pyramidGeometry->addPrimitiveSet(pyramidBase);
```

对每个面重复相同的动作。顶点仍要按逆时针方向指定。

```
osg::DrawElementsUInt* pyramidFaceOne =
new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceOne->push_back(0);
pyramidFaceOne->push_back(1);
pyramidFaceOne->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceOne);
osg::DrawElementsUInt* pyramidFaceTwo =
new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceTwo->push_back(1);
pyramidFaceTwo->push_back(2);
pyramidFaceTwo->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceTwo);
osg::DrawElementsUInt* pyramidFaceThree =
new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceThree->push_back(2);
pyramidFaceThree->push_back(3);
```

```
pyramidFaceThree->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceThree);
osg::DrawElementsUInt* pyramidFaceFour =
new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
pyramidFaceFour->push_back(3);
pyramidFaceFour->push_back(0);
pyramidFaceFour->push_back(4);
pyramidGeometry->addPrimitiveSet(pyramidFaceFour)
```

声明并加载一个 vec4 为元素的数组来存储颜色。

```
osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //index 0 red
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //index 1 green
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //index 2 blue
colors->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); //index 3 white
```

声明的这个变量可以将顶点数组元素和颜色数组元素匹配起来。这个容器的元素数应当和顶点数一致。这个容器是顶点数组和颜色数组的连接。这个索引数组中的条目就对应着顶点数组中的元素。他们的值就是颜色数组中的索引。顶点数组元素与 normal 和纹理坐标数组的匹配也是遵循这种模式。

注意，这种情况下，我们将 5 个顶点指定 4 种颜色。顶点数组的 0 和 4 元素都被指定为颜色数组的 0 元素。

```
osg::TemplateIndexArray
<unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,4>;
colorIndexArray->push_back(0); // vertex 0 assigned color array element 0
colorIndexArray->push_back(1); // vertex 1 assigned color array element 1
colorIndexArray->push_back(2); // vertex 2 assigned color array element 2
colorIndexArray->push_back(3); // vertex 3 assigned color array element 3
colorIndexArray->push_back(0); // vertex 4 assigned color array element 0
```

下一步，将颜色数组和 geometry 关联起来，将上面产生的颜色索引指定给 geometry，设定绑定模式为_PER_VERTEX。

```
pyramidGeometry->setColorArray(colors);
pyramidGeometry->setColorIndices(colorIndexArray);
pyramidGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
osg::Vec2Array* texcoords = new osg::Vec2Array(5);
(*texcoords)[0].set(0.00f,0.0f);
(*texcoords)[1].set(0.25f,0.0f);
(*texcoords)[2].set(0.50f,0.0f);
(*texcoords)[3].set(0.75f,0.0f);
(*texcoords)[4].set(0.50f,1.0f);
pyramidGeometry->setTexCoordArray(0,texcoords);
注：一下部分可能错误。
```

```
// Declare and initialize a transform node.
osg::PositionAttitudeTransform* pyramidTwoXForm =
new osg::PositionAttitudeTransform();
// Use the 'addChild' method of the osg::Group class to
// add the transform as a child of the root node and the
// pyramid node as a child of the transform.
root->addChild(pyramidTwoXForm);
pyramidTwoXForm->addChild(pyramidGeode);
// Declare and initialize a Vec3 instance to change the
// position of the tank model in the scene
osg::Vec3 pyramidTwoPosition(15,0,0);
pyramidTwoXForm->setPosition( pyramidTwoPosition );
```

既然我们生成了一个 geometry 节点并将它加到了场景中，我们就可以重用这个 geometry。例如，如果我们想让另一个 pyramid 在第一个的右侧 15 个单位处，我们就可以在我们的 scene graph 中将这个 geode 加到 transform 节点的子节点上。

最后一步，建立并进入一个仿真循环。

```
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
viewer.sync();
viewer.update();
viewer.frame();
}
```

2.使用 StateSet 产生有纹理的几何体

2.1 本章目标

为教程 1 中介绍的由 OpenGL 基本绘制单位定义的几何体添加纹理。

2.2 背景

前一节教程介绍了包含由 OpenGL 基本单位产生的基本形状的视景。本节讲解如何为这些形状添加纹理。为了使代码更方便使用，我们将 pyramid 的代码放到一个函数中，产生 geode 并返回它的指针。下面的代码来自教程 1。

```
osg::Geode* createPyramid()
{
```

```
osg::Geode* pyramidGeode = new osg::Geode();
osg::Geometry* pyramidGeometry = new osg::Geometry();
pyramidGeode->addDrawable(pyramidGeometry);
// Specify the vertices:
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
pyramidVertices->push_back( osg::Vec3(0, 0, 0) ); // front left
pyramidVertices->push_back( osg::Vec3(2, 0, 0) ); // front right
pyramidVertices->push_back( osg::Vec3(2, 2, 0) ); // back right
pyramidVertices->push_back( osg::Vec3( 0,2, 0) ); // back left
pyramidVertices->push_back( osg::Vec3( 1, 1,2) ); // peak
// Associate this set of vertices with the geometry associated with the
// geode we added to the scene.
pyramidGeometry->setVertexArray( pyramidVertices );
// Create a QUAD primitive for the base by specifying the
// vertices from our vertex list that make up this QUAD:
osg::DrawElementsUInt* pyramidBase =
new osg::DrawElementsUInt(osg::PrimitiveSet::QUADS, 0);
pyramidBase->push_back(3);
pyramidBase->push_back(2);
pyramidBase->push_back(1);
pyramidBase->push_back(0);
//Add this primitive to the geometry:
pyramidGeometry->addPrimitiveSet(pyramidBase);
// code to create other faces goes here!
// (removed to save space, see tutorial two)
osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //index 0 red
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //index 1 green
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //index 2 blue
colors->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); //index 3 white
osg::TemplateIndexArray
<unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,4>;
colorIndexArray->push_back(0); // vertex 0 assigned color array element 0
```

```
colorIndexArray->push_back(1); // vertex 1 assigned color array element 1
colorIndexArray->push_back(2); // vertex 2 assigned color array element 2
colorIndexArray->push_back(3); // vertex 3 assigned color array element 3
colorIndexArray->push_back(0); // vertex 4 assigned color array element 0
pyramidGeometry->setColorArray(colors);
pyramidGeometry->setColorIndices(colorIndexArray);
pyramidGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
// Since the mapping from vertices to texture coordinates is 1:1,
// we don't need to use an index array to map vertices to texture
// coordinates. We can do it directly with the 'setTexCoordArray'
// method of the Geometry class.
// This method takes a variable that is an array of two dimensional
// vectors (osg::Vec2). This variable needs to have the same
// number of elements as our Geometry has vertices. Each array element
// defines the texture coordinate for the cooresponding vertex in the
// vertex array.
osg::Vec2Array* texcoords = new osg::Vec2Array(5);
(*texcoords)[0].set(0.00f,0.0f); // tex coord for vertex 0
(*texcoords)[1].set(0.25f,0.0f); // tex coord for vertex 1
(*texcoords)[2].set(0.50f,0.0f); // ""
(*texcoords)[3].set(0.75f,0.0f); // ""
(*texcoords)[4].set(0.50f,1.0f); // ""
pyramidGeometry->setTexCoordArray(0,texcoords);
return pyramidGeode;
}
```

2.3 加载纹理，生成状态集合并将他们附加到节点上

渲染基本单位的方法是使用 StateSet。这节代码演示了怎样从文件中加载纹理，产生此纹理起作用的一个 StateSet，并将这个 StateSet 附加到场景中的一个节点上。前面开始的代码和上一节教程中的一样，初始化一个 viewer 并建立有一个 pyramid 的场景。

```
    int main()
{
osgProducer::Viewer viewer;
// Declare a group to act as root node of a scene:
osg::Group* root = new osg::Group();
```

```
osg::Geode* pyramidGeode = createPyramid();
root->addChild(pyramidGeode);
```

现在，准备加纹理。这里我们会声明一个纹理实例并将它的数据不一致性设为‘DYNAMIC’。（如果不把纹理声明为 dynamic，osg 的一些优化程序会删除它。）这个 texture 类包装了 OpenGL 纹理模式（wrap,filter,等等）和一个 osg::Image。下面的代码说明了如何从文件里读取 osg::Image 实例并把这个图像和纹理关联起来。

```
    osg::Texture2D* KLN89FaceTexture = new osg::Texture2D;
// protect from being optimized away as static state:
KLN89FaceTexture->setDataVariance(osg::Object::DYNAMIC);
// load an image by reading a file:
osg::Image* klnFace = osgDB::readImageFile("KLN89FaceB.tga");
if (!klnFace)
{
std::cout << " couldn't find texture, quiting." << std::endl;
return -1;
}
// Assign the texture to the image we read from file:
KLN89FaceTexture->setImage(klnFace);
```

纹理可以和渲染 StateSet 关联起来。下一步就产生一个 StateSet，关联并启动我们的纹理，并将这个 StateSet 附加到我们的 geometry 上。

```
    // Create a new StateSet with default settings:
osg::StateSet* stateOne = new osg::StateSet();
// Assign texture unit 0 of our new StateSet to the texture
// we just created and enable the texture.
stateOne->setTextureAttributeAndModes(0,KLN89FaceTexture,
    osg::StateAttribute::ON);
// Associate this state set with the Geode that contains
// the pyramid:
pyramidGeode->setStateSet(stateOne);
```

最后一步是仿真循环：

```
    //The final step is to set up and enter a simulation loop.
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
viewer.sync();
viewer.update();
viewer.frame();
}
return 0;
}
```

3.使用 Shape，改变 state

3.1 本章目标

用 osg::Shape 实例构建场景。使用 osg::StateSet 控制 shape 的渲染。

3.2 使用 Shape 类

Shape 类是所有形状类别的基类。Shape 既可用于剪裁和碰撞检测也可用于定义程序性地产生几何体的那些基本形状。下面的类继承自 Shape 类:

TriangleMesh
Sphere
InfinitePlane
HeightField
Cylinder
Cone
CompositeShape
Box

为了使这些形状可以被渲染，我们需要把他们和 Drawable 类的实例关联起来。ShapeDrawable 类提供了这样的功能。这个类继承自 Drawable 并允许我们把 Shape 实例附加到可以被渲染的东西上。既然 ShapeDrawable 类继承自 Drawable，ShapDrawable 实例就可以被加到 Geode 类实例上。下面的步骤演示了将一个单位立方体加到空场景中时是如何做到这些的。

```
// Declare a group to act as root node of a scene:
osg::Group* root = new osg::Group();
// Declare a box class (derived from shape class) instance
// This constructor takes an osg::Vec3 to define the center
// and a float to define the height, width and depth.
// (an overloaded constructor allows you to specify unique
// height, width and height values.)
osg::Box* unitCube = new osg::Box( osg::Vec3(0,0,0), 1.0f);
// Declare an instance of the shape drawable class and initialize
// it with the unitCube shape we created above.
// This class is derived from 'drawable' so instances of this
// class can be added to Geode instances.
osg::ShapeDrawable* unitCubeDrawable = new osg::ShapeDrawable(unitCube);
// Declare a instance of the geode class:
osg::Geode* basicShapesGeode = new osg::Geode();
// Add the unit cube drawable to the geode:
basicShapesGeode->addDrawable(unitCubeDrawable);
// Add the goede to the scene:
root->addChild(basicShapesGeode);
```

产生一个球体和上面的代码基本相似。没有太多的注释的代码看起来是这个样子：

```
// Create a sphere centered at the origin, unit radius:
osg::Sphere* unitSphere = new osg::Sphere( osg::Vec3(0,0,0), 1.0);
osg::ShapeDrawable* unitSphereDrawable=new osg::ShapeDrawable(unitSphere);
```

现在，我们可以使用 transform 节点将这个球体加到场景中，以便让它离开原点。unitSphereDrawable 不能直接添加到场景中（因为它不是继承自 node 类），所以我们需要一个新的 geode 以便添加它。

```
osg::PositionAttitudeTransform* sphereXForm =
new osg::PositionAttitudeTransform();
sphereXForm->setPosition(osg::Vec3(2.5,0,0));
osg::Geode* unitSphereGeode = new osg::Geode();
root->addChild(sphereXForm);
sphereXForm->addChild(unitSphereGeode);
unitSphereGeode->addDrawable(unitSphereDrawable);
```

3.3 设置状态

前面的教程讲解了如何生成纹理，将其指定为从文件加载的图像，生成一个带纹理的 StateSet。下面的代码建立了两个状态集合——一个是 BLEND 纹理模式，另一个是 DECAL 纹理模式。BLEND 模式：

```
// Declare a state set for 'BLEND' texture mode
osg::StateSet* blendStateSet = new osg::StateSet();
// Declare a TexEnv instance, set the mode to 'BLEND'
osg::TexEnv* blendTexEnv = new osg::TexEnv;
blendTexEnv->setMode(osg::TexEnv::BLEND);
// Turn the attribute of texture 0 - the texture we loaded above - 'ON'
blendStateSet->setTextureAttributeAndModes(0,KLN89FaceTexture,
                                           osg::StateAttribute::ON);
// Set the texture texture environment for texture 0 to the
// texture envirnoment we declared above:
blendStateSet->setTextureAttribute(0,blendTexEnv);
```

重复这些步骤，产生 DECAL 纹理模式的状态集合。

```
osg::StateSet* decalStateSet = new osg::StateSet();
osg::TexEnv* decalTexEnv = new osg::TexEnv();
decalTexEnv->setMode(osg::TexEnv::DECAL);
decalStateSet->setTextureAttributeAndModes(0,KLN89FaceTexture,
                                           osg::StateAttribute::ON);
decalStateSet->setTextureAttribute(0,decalTexEnv);
```

产生了状态集合后我们就可以把它们应用在场景中的节点上。在 scene graph 的绘制遍历(root->leaf)中状态是积累的。除非这个节点有一个它自己的状态，否则它会继承其父节点的状态。（如果一个节点有一个以上的父节点，它会使用一个以上的状态渲染。）

```
root->setStateSet(blendStateSet);
unitSphereGeode->setStateSet(decalStateSet);
```

最后一步是进入仿真循环。

```
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
viewer.sync();
viewer.update();
viewer.frame();
}
return 0;
```

4.更多的 StateSet

4.1StateSet 如何工作

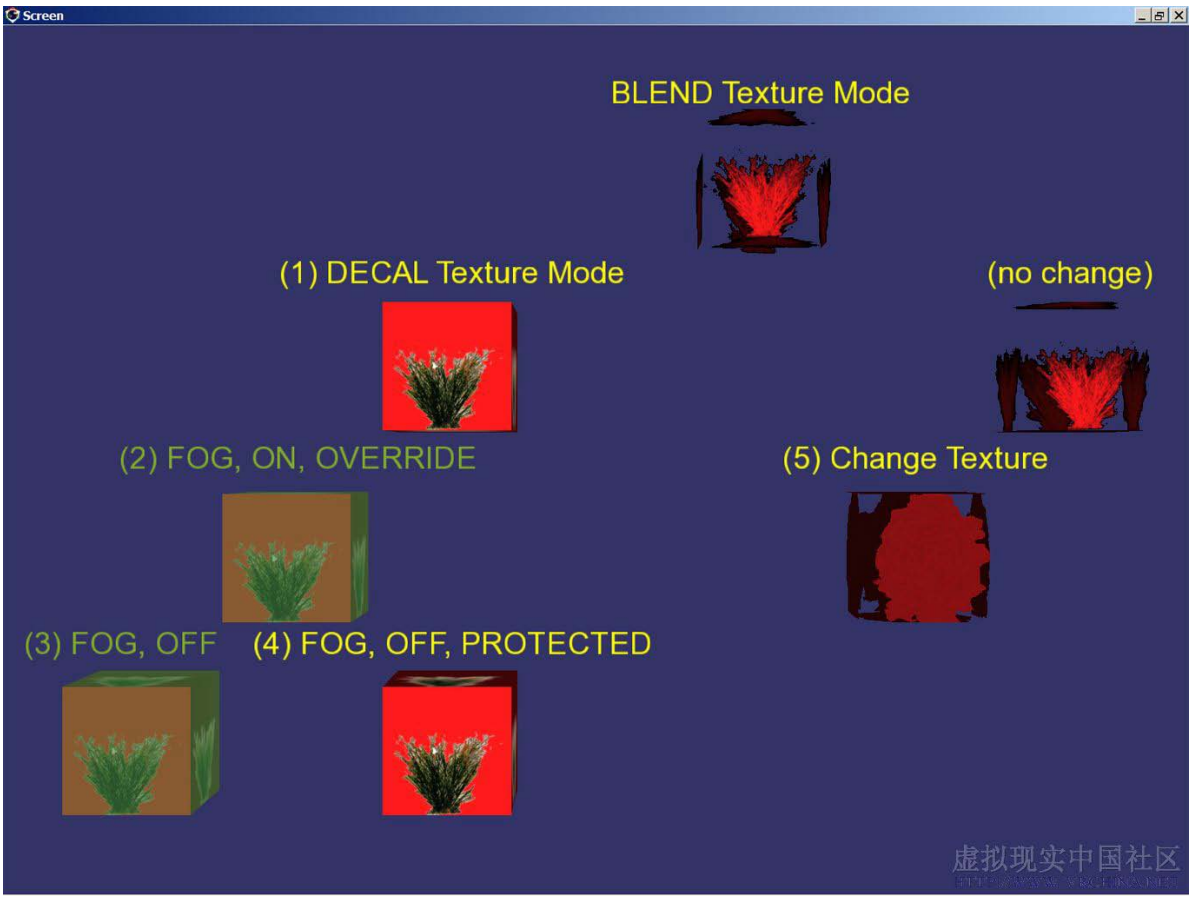
场景图管理器遍历 scene graph，决定哪些几何体需要送到图形管道渲染。在遍历的过程中，场景图管理器也搜集几何体如何被渲染的信息。这些信息存在 osg::StateSet 实例中。StateSet 包含 OpenGL 的属性/数值对列表。这些 StateSet 可以和 scenegraph 中的节点关联起来。在预渲染的遍历中，StateSet 从根节点到叶子节点是积累的。一个节点的没有变化的状态属性也是简单的遗传自父节点。

几个附加的特性允许更多的控制和弹性。一个状态的属性值可以被设为 OVERRIDE。这意味着这个节点的所有孩子节点——不管其属性值是什么——会遗传其父节点的属性值。OVERRIDE 意味着可以被覆盖。如果一个孩子节点把那个属性设为 PROTECTED，那么它就可以设置自己的属性值，而不用理会父节点的设置。

4.2 例子及代码

下面的场景示范了 state 如何影响 scene graph。根节点有一个 BLEND 模式纹理的基本状态。这个基本状态会被所有子节点继承，除非这个状态的参数改变。根节点的右子树就是这样的。右孩子没有被指定任何状态，所以它使用和根节点一样的状态来渲染。对于节点 6，纹理的混合模式没有改变但是使用了一个新纹理。

根节点的左子树的纹理模式被设为 DECAL。其他的参数是一样的，所以它们从根节点继承。在节点 3 中，FOG 被打开了并设置为 OVERRIDE。这个节点——节点 3——的左孩子将 FOG 设为了 OFF。既然它没有设置 PROTECTED 并且它父节点设置了 OVERRIDE，所以就像父节点一样 FOG 仍然是 ON。右孩子 4 设置 FOG 属性值为 PROTECTED，因此可以覆盖其父节点的设置。



下面是操作状态配置并用节点将这些状态关联起来的代码。

```
// Set an osg::TexEnv instance's mode to BLEND,
// make this TexEnv current for texture unit 0 and assign
// a valid texture to texture unit 0
blendTexEnv->setMode(osg::TexEnv::BLEND);
stateRootBlend->setTextureAttribute(0,blendTexEnv,osg::StateAttribute::ON);
stateRootBlend->setTextureAttributeAndModes(0,ocotilloTexture,
                                            osg::StateAttribute::ON);
// For state five, change the texture associated with texture unit 0
//all other attributes will remain unchanged as inherited from above.
// (texture mode will still be BLEND)
stateFiveDustTexture->setTextureAttributeAndModes(0,dustTexture,
                                                  osg::StateAttribute::ON);
// Set the mode of an osg::TexEnv instance to DECAL
//Use this mode for stateOneDecal.
decalTexEnv->setMode(osg::TexEnv::DECAL);
stateOneDecal->setTextureAttribute(0,decalTexEnv,osg::StateAttribute::ON);
```

```
// For stateTwo, turn FOG OFF and set to OVERRIDE.
//Descendants in this sub-tree will not be able to change FOG unless
//they set the FOG attribute value to PROTECTED
stateTwoFogON_OVRD->setAttribute(fog, osg::StateAttribute::ON);
stateTwoFogON_OVRD->setMode(GL_FOG,
osg::StateAttribute::ON | osg::StateAttribute::OVERRIDE);
// For stateThree, try to turn FOG OFF.
Since the attribute is not PROTECTED, and
// the parents set this attribute value to OVERRIDE, the parent's value will be used.
// (i.e. FOG will remain ON.)
stateThreeFogOFF->setMode(GL_FOG, osg::StateAttribute::OFF);
// For stateFour, set the mode to PROTECTED, thus overriding the parent setting
stateFourFogOFF_PROT->setMode(GL_FOG,
osg::StateAttribute::OFF | osg::StateAttribute::PROTECTED);
// apply the StateSets above to appropriates nodes in the scene graph.
root->setStateSet(stateRootBlend);
mtOne->setStateSet(stateOneDecal);
mtTwo->setStateSet(stateTwoFogON_OVRD);
mtThree->setStateSet(stateThreeFogOFF);
mtSix->setStateSet(stateFiveDustTexture);
mtFour->setStateSet(stateFourFogOFF_PROT);
```

5.从文件中加载模型并放入到场景中

5.1 本章目标

加载几何模型并加入到场景中，调整其中一个模型在场景中的位置并通过安装仿真循环观察场景。

5.2 加载几何模型并加入到场景中

如果你下载了当前版本的 Open Scene Graph，那么你就可以将在有相应插件的任何文件格式。包括以下的几何文件格式：3dc,3ds,flt,geo,iv,ive,lwo,md2,obj,osg 和以下这些图像文件格式：bmp,gif,jpeg,rgb,tga,tif。

Open Scene Graph 安装包里包含了很多 open scene graph 格式（.osg）的几何文件。我们会加载其中一个，还有一个 MPI Open Flight（.flt）文件。为了便于找到模型，建立一个 models 文

件夹，并用 OSG_DATA_PATH 系统变量指向它。（通常为 C:\Projects\OpenSceneGraph\OpenSceneGraph-Data\）。解压此文件到那个文件夹下。
几何模型使用 scene graph 的节点表示。因此，为了加载并操作一个几何模型文件，我们需要声明一个句柄（或指针）指向 osg::Node 类型实例。（在一些要求的#include 后）。

```
#include <osg/Node>
#include <osgDB/ReadFile>
...
osg::Node* cessnaNode = NULL;
osg::Node* tankNode = NULL;
...
cessnaNode = osgDB::readNodeFile("cessna.osg");
tankNode = osgDB::readNodeFile("Models/T72-tank/t72-tank_des.flt");
```

这就是加载数据库需要做的事。下一步我们把它作为 scene graph 的一部分加入。将模型加载到 transform 节点的子节点上，这样我们就可以重新定位它了。

```
// Declare a node which will serve as the root node
// for the scene graph. Since we will be adding nodes
// as 'children' of this node we need to make it a 'group'
// instance.
// The 'node' class represents the most generic version of nodes.
// This includes nodes that do not have children (leaf nodes.)
// The 'group' class is a specialized version of the node class.
// It adds functions associated with adding and manipulating
// children.
osg::Group* root = new osg::Group();
root->addChild(cessnaNode);
// Declare transform, initialize with defaults.
osg::PositionAttitudeTransform* tankXform =
new osg::PositionAttitudeTransform();
// Use the 'addChild' method of the osg::Group class to
// add the transform as a child of the root node and the
// tank node as a child of the transform.
root->addChild(tankXform);
tankXform->addChild(tankNode);
// Declare and initialize a Vec3 instance to change the
```

```
// position of the tank model in the scene
osg::Vec3 tankPosit(5,0,0);
tankXform->setPosition( tankPosit );
```

现在，我们的 scene graph 由一个根节点和两个子节点组成。一个是 cessna 的几何模型，另一个是一个右子树，由一个仅有一个 tank 的几何模型的 transform 节点组成。为了观察场景，需要建立一个 viewer 和一个仿真循环。就像这样做的：

```
#include <osgProducer/Viewer>
// Declare a 'viewer'
osgProducer::Viewer viewer;
// For now, we can initialize with 'standard settings'
// Standard settings include a standard keyboard mouse
// interface as well as default drive, fly and trackball
// motion models for updating the scene.
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
// Next we will need to assign the scene graph we created
// above to this viewer:
viewer.setSceneData( root );
// create the windows and start the required threads.
viewer.realize();
// Enter the simulation loop. viewer.done() returns false
// until the user presses the 'esc' key.
// (This can be changed by adding your own keyboard/mouse
// event handler or by changing the settings of the default
// keyboard/mouse event handler)
while( !viewer.done() )
{
// wait for all cull and draw threads to complete.
viewer.sync();
// Initiate scene graph traversal to update nodes.
// Animation nodes will require update. Additionally,
// any node for which an 'update' callback has been
// set up will also be updated. More information on
// settting up callbacks to follow.
viewer.update();
```

```
// initiate the cull and draw traversals of the scene.
viewer.frame();
}
```

你应当能编译并执行上面的代码（保证调用的顺序是正确的，已经添加了 main 等等）。执行代码的时候，按 h 键会弹出一个菜单选项（似乎没有这个功能——译者）。按‘esc’退出。

6.osg Text、HUD、RenderBins

6.1 本章目标

添加文本到场景中——包括 HUD 风格的文本和作为场景一部分的文本。

6.2 摘要

本类继承自 Drawable 类。也就是说文本实例可以加到 Geode 类实例上并且可以和其它几何体一样被渲染。与文本类相关的方法的全部列在*这里*。‘osgExample Text’工程示范了许多方法。这个教程提供了文本类的几个有限的函数。绘制一个 HUD 牵扯到下面两个概念：

1、生成一个子树，它的根节点有合适的投影及观察矩阵...

2、将 HUD 子树中的几何体指定到合适的 RenderBin 上，这样 HUD 几何体就会在场景的其他部分之后按正确地状态设置绘制。

渲染 HUD 的子树涉及到一个投影矩阵和一个观察矩阵。对于投影矩阵，我们会使用相当于屏幕维数水平和垂直扩展的正投影。根据这种模式，坐标相当于象素坐标。为了简单起见，观察矩阵使用单位矩阵。

为了渲染 HUD，我们把它里面的几何体附加到一个指定的 RenderBin 上。RenderBin 允许用户在几何体绘制过程中指定顺序。这对于 HUD 几何体需要最后绘制来说很有用。

6.3 代码

首先，声明我们需要的变量－osg::Text 和 osg::Projection。

```
osg::Group* root = NULL;
osg::Node* tankNode = NULL;
osg::Node* terrainNode = NULL;
osg::PositionAttitudeTransform* tankXform;
osgProducer::Viewer viewer;
// A geometry node for our HUD:
osg::Geode* HUDGeode = new osg::Geode();
```

```
// Text instance that wil show up in the HUD:
osgText::Text* textOne = new osgText::Text();
// Text instance for a label that will follow the tank:
osgText::Text* tankLabel = new osgText::Text();
// Projection node for defining view frustrum for HUD:
osg::Projection* HUDProjectionMatrix = new osg::Projection;
```

从文件里加载模型，和前面的教程一样建立 scene graph（这里没什么新东东）。

```
// Initialize root of scene:
root = new osg::Group();
osgDB::FilePathList pathList = osgDB::getDataFilePathList();
pathList.push_back
("C:\\Projects\\OpenSceneGraph\\OpenSceneGraph-Data\\NPSData\\Models\\T72-Tank\\");
pathList.push_back
("C:\\Projects\\OpenSceneGraph\\OpenSceneGraph-Data\\NPSData\\Models\\JoeDirt\\");
pathList.push_back
("C:\\Projects\\OpenSceneGraph\\OpenSceneGraph-Data\\NPSData\\Textures\\");
osgDB::setDataFilePathList(pathList);
// Load models from files and assign to nodes:
tankNode = osgDB::readNodeFile("t72-tank_des.flt");
terrainNode = osgDB::readNodeFile("JoeDirt.flt");
// Initialize transform to be used for positioning the tank
tankXform = new osg::PositionAttitudeTransform());
tankXform->setPosition( osg::Vec3d(5,5,8) );
// Build the scene - add the terrain node directly to the root,
// connect the tank node to the root via the transform node:
root->addChild(terrainNode);
root->addChild(tankXform);
tankXform->addChild(tankNode);
```

下一步，建立场景来显示 HUD 组件。添加一个子树，它的根节点有一个投影和观察矩阵。

```
// Initialize the projection matrix for viewing everything we
// will add as descendants of this node. Use screen coordinates
// to define the horizontal and vertical extent of the projection
```

```
// matrix. Positions described under this node will equate to
// pixel coordinates.
HUDProjectionMatrix->setMatrix(osg::Matrix::ortho2D(0,1024,0,768));
// For the HUD model view matrix use an identity matrix:
osg::MatrixTransform* HUDModelViewMatrix = new osg::MatrixTransform;
HUDModelViewMatrix->setMatrix(osg::Matrix::identity());
// Make sure the model view matrix is not affected by any transforms
// above it in the scene graph:
HUDModelViewMatrix->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
// Add the HUD projection matrix as a child of the root node
// and the HUD model view matrix as a child of the projection matrix
// Anything under this node will be viewed using this projection matrix
// and positioned with this model view matrix.
root->addChild(HUDProjectionMatrix);
HUDProjectionMatrix->addChild(HUDModelViewMatrix);
```

现在建立几何体。我们根据屏幕坐标建立一个四边形，并设置颜色和纹理坐标。

```
// Add the Geometry node to contain HUD geometry as a child of the
// HUD model view matrix.
HUDModelViewMatrix->addChild( HUDGeode );
// Set up geometry for the HUD and add it to the HUD
osg::Geometry* HUDBackgroundGeometry = new osg::Geometry();
osg::Vec3Array* HUDBackgroundVertices = new osg::Vec3Array;
HUDBackgroundVertices->push_back( osg::Vec3( 0,0,-1) );
HUDBackgroundVertices->push_back( osg::Vec3(1024,0,-1) );
HUDBackgroundVertices->push_back( osg::Vec3(1024,200,-1) );
HUDBackgroundVertices->push_back( osg::Vec3(0,200,-1) );
osg::DrawElementsUInt* HUDBackgroundIndices =
new osg::DrawElementsUInt(osg::PrimitiveSet::POLYGON, 0);
HUDBackgroundIndices->push_back(0);
HUDBackgroundIndices->push_back(1);
HUDBackgroundIndices->push_back(2);
HUDBackgroundIndices->push_back(3);
osg::Vec4Array* HUDcolors = new osg::Vec4Array;
HUDcolors->push_back(osg::Vec4(0.8f,0.8f,0.8f,0.8f));
```

```
osg::Vec2Array* texcoords = new osg::Vec2Array(4);
(*texcoords)[0].set(0.0f,0.0f);
(*texcoords)[1].set(1.0f,0.0f);
(*texcoords)[2].set(1.0f,1.0f);
(*texcoords)[3].set(0.0f,1.0f);
HUDBackgroundGeometry->setTexCoordArray(0,texcoords);
osg::Texture2D* HUDTexture = new osg::Texture2D;
HUDTexture->setDataVariance(osg::Object::DYNAMIC);
osg::Image* hudImage;
hudImage = osgDB::readImageFile("HUDBack2.tga");
HUDTexture->setImage(hudImage);
osg::Vec3Array* HUDnormals = new osg::Vec3Array;
HUDnormals->push_back(osg::Vec3(0.0f,0.0f,1.0f));
HUDBackgroundGeometry->setNormalArray(HUDnormals);
HUDBackgroundGeometry->setNormalBinding(osg::Geometry::BIND_OVERALL);
HUDBackgroundGeometry->addPrimitiveSet(HUDBackgroundIndices);
HUDBackgroundGeometry->setVertexArray(HUDBackgroundVertices);
HUDBackgroundGeometry->setColorArray(HUDcolors);
HUDBackgroundGeometry->setColorBinding(osg::Geometry::BIND_OVERALL);
HUDGeode->addDrawable(HUDBackgroundGeometry);
```

为了正确的渲染 HUD，我们建立带有深度检测和透明度混合的 osg::stateSet。我们也要保证 HUD 几何体最后绘制。几何体在裁剪遍历时通过指定一个已编号的渲染箱可以控制渲染顺序。最后一行演示了这些：

```
// Create and set up a state set using the texture from above:
osg::StateSet* HUDStateSet = new osg::StateSet();
HUDGeode->setStateSet(HUDStateSet);
HUDStateSet->
setTextureAttributeAndModes(0,HUDTexture,osg::StateAttribute::ON);
// For this state set, turn blending on (so alpha texture looks right)
HUDStateSet->setMode(GL_BLEND,osg::StateAttribute::ON);
// Disable depth testing so geometry is draw regardless of depth values
// of geometry already draw.
HUDStateSet->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
HUDStateSet->setRenderingHint( osg::StateSet::TRANSPARENT_BIN );
```

```
// Need to make sure this geometry is draw last. RenderBins are handled
// in numerical order so set bin number to 11
HUDStateSet->setRenderBinDetails( 11, "RenderBin");
```

最后，使用文本时，由于 osg::Text 是继承自 osg::Drawable 的，osg::Text 实例可以作为孩子加到 osg::Geode 类实例上。

```
// Add the text (Text class is derived from drawable) to the geode:
HUDGeode->addDrawable( textOne );
// Set up the parameters for the text we'll add to the HUD:
textOne->setCharacterSize(25);
textOne->setFont("C:/WINDOWS/Fonts/impact.ttf");
textOne->setText("Not so good");
textOne->setAxisAlignment(osgText::Text::SCREEN);
textOne->setPosition( osg::Vec3(360,165,-1.5) );
textOne->setColor( osg::Vec4(199, 77, 15, 1) );
// Declare a geode to contain the tank's text label:
osg::Geode* tankLabelGeode = new osg::Geode();
// Add the tank label to the scene:
tankLabelGeode->addDrawable(tankLabel);
tankXform->addChild(tankLabelGeode);
// Set up the parameters for the text label for the tank
// align text with tank's SCREEN.
// (for Onder: use XZ_PLANE to align text with tank's XZ plane.)
tankLabel->setCharacterSize(5);
tankLabel->setFont("/fonts/arial.ttf");
tankLabel->setText("Tank #1");
tankLabel->setAxisAlignment(osgText::Text::XZ_PLANE);
// Set the text to render with alignment anchor and bounding box around it:
tankLabel->setDrawMode(osgText::Text::TEXT |
osgText::Text::ALIGNMENT |
osgText::Text::BOUNDINGBOX);
tankLabel->setAlignment(osgText::Text::CENTER_TOP);
tankLabel->setPosition( osg::Vec3(0,0,8) );
tankLabel->setColor( osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) );
```

最后，建立 viewer 并进入仿真循环：

```
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
viewer.sync();
viewer.update();
viewer.frame();
}
```

7.搜索并控制开关和 DOF(自由度)节点

7.1 搜索场景图形中的一个有名节点

模型文件可能包含了各种不同的节点类型，用户通过对这些节点的使用来更新和表达模型的各个部分。使用 osgSim::MultiSwitch 多重节点可以在多个模型渲染状态间进行选择。例如，对坦克模型使用多重节点，用户即可自行选择与完整的或者损坏的坦克相关联的几何体以及渲染状态。模型中还可以包含 DOF 节点，以便清晰表达坦克的某个部分。例如炮塔节点可以旋转，机枪节点可以升高。炮塔旋转时，炮塔体（包括机枪）的航向角（heading）与坦克的航向角相关联，而机枪抬升时，机枪的俯仰角（pitch）与炮塔的俯仰角相关联。

对这些节点进行更新时，我们需要一个指向节点的指针。而我们首先要获取节点的名字，才能得到该节点的指针。而获取节点的名称，主要有这样一些方法：咨询建模人员；使用其它文件浏览器（对于.flt 文件，可以使用 Creator 或者 Vega）浏览模型；或者使用 OpenSceneGraph。用户可以根据自己的需要自由运用 OSG 的功能。例如在场景图形中载入 flt 文件，并且在仿真过程中将整个场景保存成.osg 文件。osg 文件使用 ASCII 格式保存，因此用户可以使用各种文本处理软件（写字板，记事本）对其进行编辑。在坦克模型文件中，你可以发现一个名为“sw1”的开关节点，它有两个子节点“good”和“bad”，分别指向坦克未损坏和损坏的状态。坦克模型的.osg 文件可以从这里下载：

http://www.nps.navy.mil/cs/sullivan/osgTutorials/Download/T72Tank.osg

现在我们已经获得了需要控制的开关节点的名称（sw1），亦可获取其指针对象。获取节点指针的方法有两种：一是编写代码遍历整个场景图形；二是使用后面将会介绍的访问器（visitor）。在以前的教程中，我们已经知道如何加载 flight 文件，将其添加到场景并进入仿真循环的方法。

```
#include <osg/PositionAttitudeTransform>
#include <osg/Group>
#include <osg/Node>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <osgGA/TrackballManipulator>
int main()
```

```
{
osg::Node* tankNode = NULL;
osg::Group* root = NULL;
osgViewer::Viewer viewer;
osg::Vec3 tankPosit;
osg::PositionAttitudeTransform* tankXform;
tankNode = osgDB::readNodeFile("../NPS_Data/Models/t72-tank/t72-tank_des.flt");
root = new osg::Group();
tankXform = new osg::PositionAttitudeTransform();
root->addChild(tankXform);
tankXform->addChild(tankNode);
tankPosit.set(5,0,0);
tankXform->setPosition( tankPosit );
viewer.setCameraManipulator(new osgGA::TrackballManipulator());
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
    {
       viewer.frame();
    }
}
```

现在我们需要修改上述代码，以添加查找节点的函数。下面的递归函数有两个参数值：用于搜索的字符串，以及用于指定搜索开始位置的节点。函数的返回值是指定节点子树中，第一个与输入字符串名称相符的节点实例。如果没有找到这样的节点，函数将返回 NULL。特别要注意的是，使用访问器将提供更为灵活的节点访问方式。而下面的代码只用于展示如何手动编写场景图形的遍历代码。

```
osg::Node* findNamedNode(const std::string& searchName,
osg::Node* currNode)
{
osg::Group* currGroup;
osg::Node* foundNode;
// 检查输入的节点是否是合法的，
// 如果输入节点为 NULL，则直接返回 NULL。
if ( !currNode)
    { return NULL; }
// 如果输入节点合法，那么先检查该节点是否就是我们想要的结果。
// 如果确为所求，那么直接返回输入节点。
if (currNode->getName() == searchName)
    { return currNode; }
// 如果输入节点并非所求，那么检查它的子节点（不包括叶节点）情况。
// 如果子节点存在，则使用递归调用来检查每个子节点。
// 如果某一次递归的返回值非空，说明已经找到所求的节点，返回其指针。
// 如果所有的节点都已经遍历过，那么说明不存在所求节点，返回 NULL。
currGroup = currNode->asGroup();
```

```
if ( currGroup )
{
for (unsigned int i = 0 ; i < currGroup->getNumChildren(); i ++)
{
foundNode = findNamedNode(searchName, currGroup->getChild(i));
if (foundNode)
return foundNode; // 找到所求节点。
}
return NULL; // 遍历结束，不存在所求节点。
}
else
{
return NULL; // 该节点不是组节点，返回 NULL。
}
}
```

现在我们可以在代码中添加这个函数，用于查找场景中指定名称的节点并获取其指针。注意这是一种深度优先的算法，它返回第一个符合的节点指针。

我们将在设置场景之后，进入仿真循环之前调用该函数。函数返回的开关节点指针可以用于更新开关的状态。下面的代码用于模型载入后，执行查找节点的工作。

```
osg::Switch* tankStateSwitch = NULL;
osg::Node* foundNode = NULL;
foundNode = findNamedNode("sw1",root);
tankStateSwitch = (osg::Switch*) foundNode;
if ( !tankStateSwitch)
{
std::cout << "tank state switch node not found, quitting." << std::endl;
return -1;
}
```

7.2 按照“访问器”模式搜索有名节点（Finding a named node using the Visitor pattern）

“访问器”的设计允许用户将某个特定节点的指定函数，应用到当前场景遍历的所有此类节点中。遍历的类型包括 NODE_VISITOR，UPDATE_VISITOR，COLLECT_OCCLUDER_VISITOR 和 CULL_VISITOR。由于我们还没有讨论场景更新(updating)，封闭节点（occluder node）和裁选（culling）的有关内容，因此这里首先介绍 NODE_VISITOR（节点访问器）遍历类型。“访问器”同样允许用户指定遍历的模式，可选项包括 TRAVERSE_NONE，TRAVERSE_PARENTS，TRAVERSE_ALL_CHILDREN 和 TRAVERSE_ACTIVE_CHILDREN。这里我们将选择 TRAVERSE_ALL_CHILDREN（遍历所有子节点）的模式。

然后，我们需要定义应用到每个节点的函数。这里我们将会针对用户自定义的节点名称进行字符串比较。如果某个节点的名称与指定字符串相符，该节点将被添加到一个节点列表中。遍历过程结束后，列表中将包含所有符合指定的搜索字符串的节点。

为了能够充分利用“访问器”，我们可以从基类 osg::NodeVisitor 派生一个特定的节点访问器（命名为

findNodeVisitor)。这个类需要两个新的数据成员：一个 std::string 变量，用于和我们搜索的有名节点进行字符串比较；以及一个节点列表变量（std::vector<osg::Node*>），用于保存符合搜索字符串的所有节点。为了实现上述的操作，我们需要重载“apply”方法。基类的“apply”方法已经针对所有类型的节点（所有派生自 osg::Node 的节点）作了定义。用户可以重载 apply 方法来操作特定类型的节点。如果我们希望针对所有的节点进行同样的操作，那么可以重载针对 osg::Node 类型的 apply 方法。findNodeVisitor 的头文件内容在下表中列出，相关的源代码可以在这里下载：

http://www.nps.navy.mil/cs/sullivan/osgTutorials/Download/findNodeVisitor.zip

```
#include <osg/NodeVisitor>
#include <osg/Node>
#include <osgSim/DOFTransform>
#include <iostream>
#include <vector>
class findNodeVisitor : public osg::NodeVisitor {
public:
findNodeVisitor();
findNodeVisitor(const std::string &searchName) ;
virtual void apply(osg::Node &searchNode);
virtual void apply(osg::Transform &searchNode);
void setNameToFind(const std::string &searchName);
osg::Node* getFirst();
typedef std::vector<osg::Node*> nodeListType;
nodeListType& getNodeList() { return foundNodeList; }
private:
std::string searchForName;
nodeListType foundNodeList;
};
```

现在，我们创建的类可以做到：启动一次节点访问遍历，访问指定场景子树的每个子节点，将节点的名称与用户指定的字符串作比较，并建立一个列表用于保存名字与搜索字符串相同的节点。那么如何启动这个过程呢？我们可以使用 osg::Node 的“accept”方法来实现节点访问器的启动。选择某个执行 accept 方法的节点，我们就可以控制遍历开始的位置。（遍历的方向是通过选择遍历模式来决定的，而节点类型的区分则是通过重载相应的 apply 方法来实现）“accpet”方法将响应某一类的遍历请求，并执行用户指定节点的所有子类节点的 apply 方法。在这里我们将重载一般节点的 apply 方法，并选择 TRAVERSE_ALL_CHILDREN 的遍历模式，因此，触发 accept 方法的场景子树中所有的节点，均会执行这一 apply 方法。

在这个例子中，我们将读入三种不同状态的坦克。第一个模型没有任何变化，第二个模型将使用多重开关（multSwitch）来关联损坏状态，而第三个模型中，坦克的炮塔将旋转不同的角度，同时枪管也会升高。

下面的代码实现了从文件中读入三个坦克模型并将其添加到场景的过程。其中两个坦克将作为变换节点（PositionAttitudeTransform）的子节点载入，以便将其位置设置到坐标原点之外。

```
// 定义场景树的根节点，以及三个独立的坦克模型节点
osg::Group* root = new osg::Group();
osg::Group* tankOneGroup = NULL;
osg::Group* tankTwoGroup = NULL;
osg::Group* tankThreeGroup = NULL;
```

```
// 从文件中读入坦克模型
tankOneGroup = dynamic_cast<osg::Group*>
(osgDB::readNodeFile("t72-tank/t72-tank_des.flt"));
tankTwoGroup = dynamic_cast<osg::Group*>
(osgDB::readNodeFile("t72-tank/t72-tank_des.flt"));
tankThreeGroup = dynamic_cast<osg::Group*>
(osgDB::readNodeFile("t72-tank/t72-tank_des.flt"));

// 将第一个坦克作为根节点的子节点载入
root->addChild(tankOneGroup);

// 为第二个坦克定义一个位置变换
osg::PositionAttitudeTransform* tankTwoPAT =
new osg::PositionAttitudeTransform();
// 将第二个坦克向右移动 5 个单位，向前移动 5 个单位
tankTwoPAT->setPosition( osg::Vec3(5,5,0) );
// 将第二个坦克作为变换节点的子节点载入场景
root->addChild(tankTwoPAT);
tankTwoPAT->addChild(tankTwoGroup);

// 为第三个坦克定义一个位置变换
osg::PositionAttitudeTransform* tankThreePAT =
new osg::PositionAttitudeTransform();
// 将第二个坦克向右移动 10 个单位
tankThreePAT->setPosition( osg::Vec3(10,0,0) );
// 将坦克模型向左旋转 22.5 度（为此，炮塔的旋转应当与坦克的头部关联）
tankThreePAT->setAttitude( osg::Quat(3.14159/8.0, osg::Vec3(0,0,1) ));
// 将第三个坦克作为变换节点的子节点载入场景
root->addChild(tankThreePAT);
tankThreePAT->addChild(tankThreeGroup);
```

我们准备将第二个模型设置为损坏的状态，因此我们使用 findNodeVisitor 类获取控制状态的多重开关（multiSwitch）的句柄。这个节点访问器需要从包含了第二个坦克的组节点开始执行。下面的代码演示了声明和初始化一个 findNodeVisitor 实例并执行场景遍历的方法。遍历完成之后，我们即可得到节点列表中符合搜索字符串的第一个节点的句柄。这也就是我们准备使用 multiSwitch 来进行控制的节点句柄。

```
// 声明一个 findNodeVisitor 类的实例，设置搜索字符串为“sw1”
findNodeVisitor findNode("sw1");

// 开始执行访问器实例的遍历过程，起点是 tankTwoGroup，搜索它所有的子节点
// 并创建一个列表，用于保存所有符合搜索条件的节点
tankTwoGroup->accept(findNode);
```

```
// 声明一个开关类型，并将其关联给搜索结果列表中的第一个节点。
osgSim::MultiSwitch* tankSwitch = NULL;
tankSwitch = dynamic_cast <osgSim::MultiSwitch*> (findNode.getFirst());
```

更新开关节点

当我们获取了一个合法的开关节点句柄后，下一步就是从一个模型状态变换到另一个状态。我们可以使用 setSingleChildOn 方法来实现这个操作。setSingleChildOn()方法包括两个参数：第一个无符号整型量相当于多重开关组（switchSet）的索引号；第二个无符号整型量相当于开关的位置。在这个例子中，我们只有一个多重开关，其值可以设置为未损坏状态或者损坏状态，如下所示：

```
// 首先确认节点是否合法，然后设置其中的第一个（也是唯一的）多重开关
if (tankSwitch)
{
//tankSwitch->setSingleChildOn(0,false); // 未损坏的模型
tankSwitch->setSingleChildOn(0,true); // 损坏的模型
}
```

更新 DOF 节点

坦克模型还包括了两个 DOF（自由度）节点“turret”和“gun”。这两个节点的句柄也可以使用上文所述的 findNodeVisitor 来获取。（此时，访问器的场景遍历应当从包含第三个模型的组节点处开始执行）一旦我们获取了某个 DOF 节点的合法句柄之后，即可使用 setCurrentHPR 方法来更新与这些节点相关的变换矩阵。setCurrentHPR 方法只有一个参数：这个 osg::Vec3 量相当于三个欧拉角 heading，pitch 和 roll 的弧度值。（如果要使用角度来描述这个值，可以使用 osg::DegreesToRadians 方法）

```
// 声明一个 findNodeVisitor 实例，设置搜索字符串为“turret”
findNodeVisitor findTurretNode("turret");
// 遍历将从包含第三个坦克模型的组节点处开始执行
tankThreeGroup->accept(findTurretNode);

// 确认我们找到了正确类型的节点
osgSim::DOFTransform * turretDOF =
dynamic_cast<osgSim::DOFTransform *> (findTurretNode.getFirst()) ;

// 如果节点句柄合法，则设置炮塔的航向角为向右 22.5 度。
if (turretDOF)
{
turretDOF->setCurrentHPR( osg::Vec3(-3.14159/4.0,0.0,0.0) );
}
```

同理，机枪的自由度也可以如下设置：

```
// 声明一个 findNodeVisitor 实例，设置搜索字符串为“gun”
findNodeVisitor findGunNode("gun");
```

```
// 遍历将从包含第三个坦克模型的组节点处开始执行
tankThreeGroup->accept(findGunNode);

// 确认我们找到了正确类型的节点
osgSim::DOFTransform * gunDOF =
dynamic_cast<osgSim::DOFTransform *> (findGunNode.getFirst()) ;

// 如果节点句柄合法，则设置机枪的俯仰角为向上 22.5 度。
if (gunDOF)
{
gunDOF->setCurrentHPR( osg::Vec3(0.0,3.14159/8.0,0.0) );
}
```

8. 使用更新回调来更改模型

8.1 本章目标

使用回调类实现对场景图形节点的更新。前一个教程介绍了在进入主仿真循环之前，更新 DOF 和开关节点的方法。本节将讲解如何使用回调来实现在每帧的更新遍历（update traversal）中进行节点的更新。

8.2 回调概览

用户可以使用回调来实现与场景图形的交互。回调可以被理解成是一种用户自定义的函数，根据遍历方式的不同（更新 update，拣选 cull，绘制 draw），回调函数将自动地执行。回调可以与个别的节点或者选定类型（及子类型）的节点相关联。在场景图形的各次遍历中，如果遇到的某个节点已经与用户定义的回调类和函数相关联，则这个节点的回调将被执行。如果希望了解有关遍历和回调的更多信息，请参阅 David Eberly 所著的《3D Game Engine Design》第四章，以及 SGI 的《Performer Programmer's Guide》第四章。相关的示例请参见 osgCallback 例子。

8.3 创建一个更新回调

更新回调将在场景图形每一次运行更新遍历时被执行。与更新回调相关的代码可以在每一帧被执行，且实现过程是在拣选回调之前，因此回调相关的代码可以插入到主仿真循环的 viewer.update()和 viewer.frame()函数之间。而 OSG 的回调也提供了维护更为方便的接口来实现上述的功能。善于使用回调的程序代码也可以在多线程的工作中更加高效地运行。

从前一个教程展开来说，如果我们需要自动更新与坦克模型的炮塔航向角和机枪倾角相关联的 DOF（自由度）节点，我们可以采取多种方式来完成这一任务。譬如，针对我们将要操作的各个节点编写相应的回调函数：包括一个与机枪节点相关联的回调，一个与炮塔节点相关联的回调，等等。这种方法的缺陷是，与不同模型相关联的函数无法被集中化，因此增加了代码阅读、维护和更新的复杂性。另一种（极端的）方法是，只编写一个更新回调函数，来完成整个场景的节点操作。本质

上来说，这种方法和上一种具有同样的问题，因为所有的代码都会集中到仿真循环当中。当仿真的复杂程度不断增加时，这个唯一的更新回调函数也会变得愈发难以阅读、维护和修改。关于编写场景中节点/子树回调函数的方法，并没有一定之规。在本例中我们将创建单一的坦克节点回调，这个回调函数将负责更新炮塔和机枪的自由度节点。

为了实现这一回调，我们需要在节点类原有的基础上添加新的数据。我们需要获得与炮塔和机枪相关联的 DOF 节点的句柄，以更新炮塔旋转和机枪俯仰的角度值。角度值的变化要建立在上一次变化的基础上。因为回调是作为场景遍历的一部分进行初始化的，我们所需的参数通常只有两个：一个是与回调相关联的节点指针，一个是用于执行遍历的节点访问器指针。为了获得更多的参数数据（炮塔和机枪 DOF 的句柄，旋转和俯仰角度值），我们可以使用节点类的 userData 数据成员。userData 是一个指向用户定义类的指针，其中包含了关联某个特定节点时所需的一切数据集。而对于用户自定义类，只有一个条件是必需的，即，它必须继承自 osg::Referenced 类。Referenced 类提供了智能指针的功能，用于协助用户管理内存分配。智能指针记录了分配给一个类的实例的引用计数值。这个类的实例只有在引用计数值到达 0 的时候才会被删除。有关 osg::Referenced 的更详细叙述，请参阅本章后面的部分。基于上述的需求，我们向坦克节点添加如下的代码：

```
class tankDataType : public osg::Referenced
{
public:
// 公有成员……
protected:
osgSim::DOFTransform* tankTurretNode;
osgSim::DOFTransform* tankGunNode;
double rotation;
double elevation;
};
```

为了正确实现 tankData 类，我们需要获取 DOF 节点的句柄。这一工作可以在类的构造函数中使用前一教程所述的 findNodeVisitor 类完成。findNodeVisitor 将从一个起始节点开始遍历。本例中我们将从表示坦克的子树的根节点开始执行遍历，因此我们需要向 tankDataType 的构造函数传递坦克节点的指针。因此，tankDataType 类的构造函数代码应当编写为：（向特定节点分配用户数据的步骤将随后给出）

```
tankDataType::tankDataType(osg::Node* n)
{
rotation = 0;
elevation = 0;

findNodeVisitor findTurret("turret");
n->accept(findTurret);
tankTurretNode =
dynamic_cast <osgSim::DOFTransform*> (findTurret.getFirst());

findNodeVisitor findGun("gun");
n->accept(findGun);
tankGunNode =
dynamic_cast< osgSim::DOFTransform*> (findGun.getFirst());
}
```

我们也可以在 tankDataType 类中定义更新炮塔旋转和机枪俯仰的方法。现在我们只需要简单地让炮塔和机枪角度每帧改变一个固定值即可。对于机枪的俯仰角，我们需要判断它是否超过了实际情况的限制值。如果达到限制值，则重置仰角为 0。炮塔的旋转可以在一个圆周内自由进行。

```
void tankDataType::updateTurretRotation()
{
rotation += 0.01;
tankTurretNode->setCurrentHPR( osg::Vec3(rotation,0,0) );
}

void tankDataType::updateGunElevation()
{
elevation += 0.01;
tankGunNode->setCurrentHPR( osg::Vec3(0,elevation,0) );
if (elevation > .5)
elevation = 0.0;
}
```

将上述代码添加到类的内容后，我们新定义的类如下所示：

```
class tankDataType : public osg::Referenced
{
public:
tankDataType(osg::Node*n);
void updateTurretRotation();
void updateGunElevation();
protected:
osgSim::DOFTransform* tankTurretNode;
osgSim::DOFTransform* tankGunNode;
double rotation; //（弧度值）
double elevation; //（弧度值）
};
```

下一个步骤是创建回调，并将其关联到坦克节点上。为了创建这个回调，我们需要重载“()”操作符，它包括两个参数：节点的指针和节点访问器的指针。在这个函数中我们将执行 DOF 节点的更新。因此，我们需要执行 tankData 实例的更新方法，其中 tankData 实例使用坦克节点的 userData 成员与坦克节点相关联。坦克节点的指针可以通过使用 getUserData 方法来获取。由于这个方法的返回值是一个 osg::Referenced 基类的指针，因此需要将其安全地转换为 tankDataType 类的指针。为了保证用户数据的引用计数值是正确的，我们使用模板类型 osg::ref_ptr<tankDataType>指向用户数据。整个类的定义如下：

```
class tankNodeCallback : public osg::NodeCallback
{
public:
virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
{
osg::ref_ptr<tankDataType> tankData =
dynamic_cast<tankDataType*> (node->getUserData() );
if(tankData)
```

```
{
tankData->updateTurretRotation();
tankData->updateGunElevation();
}
traverse(node, nv);
}
};
```

下一步的工作是“安装”回调：将其关联给我们要修改的坦克节点，以实现每帧的更新函数执行。因此，我们首先要保证坦克节点的用户数据（tankDataType 类的实例）是正确的。然后，我们使用 osg::Node 类的 setUpdateCallback 方法将回调与正确的节点相关联。代码如下所示：

```
// 初始化变量和模型，建立场景……
tankDataType* tankData = new tankDataType(tankNode);

tankNode->setUserData( tankData );
tankNode->setUpdateCallback(new tankNodeCallback);
```

创建了回调之后，我们进入仿真循环。仿真循环的代码不用加以改变。当我们调用视口类实例的 frame()方法时，我们即进入一个更新遍历。当更新遍历及至坦克节点时，将触发 tankNodeCallback 类的操作符“()”函数。

```
// 视口的初始化，等等……
while( !viewer.done() )
{
viewer.frame();
}
return 0;
}
```

9.处理键盘输入

9.1 本章目标

使程序具备将键盘事件与特定函数相关联的能力。在前面的教程中我们已经可以使用更新回调来控制炮塔的旋转。本章中我们将添加一个键盘接口类，实现通过用户的键盘输入来更新炮塔的转角。

9.2 GUI（图形用户接口）事件处理器：

GUI 事件适配器 GUIEventAdapter 和 GUI 动作适配器 GUIActionAdapter。

GUIEventHandler 类向开发者提供了窗体系统的 GUI 事件接口。这一事件处理器使用 GUIEventAdapter 实例来接收更新。事件处理器还可以使用 GUIActionAdapter 实例向 GUI 系统发送请求，以实现一些特定的操作。

GUIEventAdapter 实例包括了各种事件类型(PUSH,RELEASE,DOUBLECLICK,DRAG,MOVE,KEYDOWN,KEYUP，FRAME，RESIZE，SCROLLUP，SCROLLDOWN，SCROLLLEFT）。依据 GUIEventAdapter 事件类型

的不同，其实例可能还有更多的相关属性。例如 X，Y 坐标与鼠标事件相关。KEYUP 和 KEYDOWN 事件则与一个按键值（例如“a”，“F1”）相关联。

GUIEventHandler 使用 GUIActionAdapter 来请求 GUI 系统执行动作。这些动作包括重绘请求 requestRedraw()，多次更新请求 requestContinuousUpdate()，光标位置重置请求 requestWarpPointer(x,y)。

GUIEventHandler 类主要通过 handle 方法来实现与 GUI 的交互。handle 方法有两个参数：一个 GUIEventAdapter 实例用于接收 GUI 的更新，以及一个 GUIActionAdapter 用于向 GUI 发送请求。handle 方法用于检查 GUIEventAdapter 的动作类型和值，执行指定的操作，并使用 GUIActionAdapter 向 GUI 系统发送请求。如果事件已经被正确处理，则 handle 方法返回的布尔值为 true，否则为 false。

一个 GUI 系统可能与多个 GUIEventAdapter 相关联（GUIEventAdapter 的顺序保存在视口类的 eventHandlerList 中），因此这个方法的返回值可以用于控制单个键盘事件的多次执行。如果一个 GUIEventHandler 返回 false，下一个 GUIEventHandler 将继续响应同一个键盘事件。

后面的例子将演示 GUIEventHandler 与 GUI 系统交互的方法：TrackballManipulator 类（继承自 GUIEventHandler）以 GUIEventAdapter 实例的形式接收鼠标事件的更新。鼠标事件的解析由 TrackballManipulator 类完成，并可以实现“抛出”的操作（所谓抛出，指的是用户按下键拖动模型并突然松开，以实现模型的持续旋转或移动）。解析事件时，TrackBallManipulator 将发送请求到 GUI 系统（使用 GUIActionAdapter），启动定时器并使自己被重复调用，以计算新的模型方向或者位置数据。

9.3 简单的键盘接口类

以下主要介绍如何创建一个用于将键盘输入关联到特定函数的键盘接口类。当用户将按键注册到接口类并设定相应的 C++响应函数之后，即可建立相应的表格条目。该表格用于保存键值（“a”，“F1”等等），按键状态（按下，松开）以及 C++响应函数。本质上讲，用户可以由此实现形同“按下 f 键，即执行 functionOne”的交互操作。由于新的类将继承自 GUIEventHandler 类，因此每当 GUI 系统捕获到一个 GUI 事件时，这些类的 handle 方法都会被触发。而 handle 方法触发后，GUI 事件的键值和按键状态（例如，松开 a 键）将与表格中的条目作比较，如果发现相符的条目，则执行与此键值和状态相关联的函数

用户通过 addFunction 方法可以注册按键条目。这个函数有两种形式。第一种把键值和响应函数作为输入值。这个函数主要用于用户仅处理 KEY_DOWN 事件的情形。例如，用户可以将“a”键的按下事件与一个反锯齿效果的操作函数相关联。但是用户不能用这个函数来处理按键松开的动作。

另一个情形下，用户可能需要区分由单个按键的“按下”和“松开”事件产生的不同动作。例如控制第一人称视角的射击者动作。按下 w 键使模型加速向前。松开 w 键之后，运动模型逐渐停止。一种可行的设计方法是，为按下按键和松开按键分别设计不同的响应函数。两者中的一个用来实现按下按键的动作。

```
#ifndef KEYBOARD_HANDLER_H
#define KEYBOARD_HANDLER_H
#include <osgGA/GUIEventHandler>
class keyboardEventHandler : public osgGA::GUIEventHandler
{
public:
typedef void (*functionType) ();
enum keyStatusType
```

```
{
KEY_UP, KEY_DOWN
};
// 用于保存当前按键状态和执行函数的结构体。
// 记下当前按键状态的信息以避免重复的调用。
// （如果已经按下按键，则不必重复调用相应的方法）
struct functionStatusType
{
functionStatusType() {keyState = KEY_UP; keyFunction = NULL;}
functionType keyFunction;
keyStatusType keyState;
};
// 这个函数用于关联键值和响应函数。如果键值在之前没有注册过，它和
// 它的响应函数都会被添加到“按下按键”事件的映射中，并返回 true。
// 否则，不进行操作并返回 false。
bool addFunction(int whatKey, functionType newFunction);
// 重载函数，允许用户指定函数是否与 KEY_UP 或者 KEY_DOWN 事件关联。
bool addFunction(int whatKey, keyStatusType keyPressStatus,
functionType newFunction);

// 此方法将比较当前按下按键的状态以及注册键/状态的列表。
// 如果条目吻合且事件较新（即，按键还未按下），则执行响应函数。
virtual bool handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter&);

// 重载函数，用于实现 GUI 事件处理访问器的功能。
virtual void accept(osgGA::GUIEventHandlerVisitor& v)
{ v.visit(*this); };

protected:
// 定义用于保存已注册键值，响应函数和按键状态的数据类型。
typedef std::map<int, functionStatusType > keyFunctionMap;
// 保存已注册的“按下按键”方法及其键值。
keyFunctionMap keyFuncMap;
// 保存已注册的“松开按键”方法及其键值。
keyFunctionMap keyUPFuncMap;
};
#endif
```

9.4 使用键盘接口类

下面的代码用于演示如何使用上面定义的类：

```
// 建立场景和视口。
// ……
```

```
// 声明响应函数：
// startAction(), stopAction(), toggleSomething()
// ……
// 声明并初始化键盘事件处理器的实例。
keyboardEventHandler* keh = new keyboardEventHandler();

// 将事件处理器添加到视口的事件处理器列表。
// 如果使用 push_front 且列表第一项的 handle 方法返回 true，则其它处理器
// 将不会再响应 GUI 同一个 GUI 事件。我们也可以使用 push_back，将事件的
// 第一处理权交给其它的事件处理器；或者也可以设置 handle 方法的返回值
// 为 false。OSG 2.x 版还允许使用 addEventHandler 方法来加以替代。
//viewer.getEventHandlers().push_front(keh);
viewer.addEventHandler(keh);

// 注册键值，响应函数。
// 按下 a 键时，触发 toggelSomething 函数。
// （松开 a 键则没有效果）
keh->addFunction('a',toggleSomething);

// 按下 j 键时，触发 startAction 函数。（例如，加快模型运动速度）
// 注意，也可以不添加第二个参数。
keh->addFunction('j',keyboardEventHandler::KEY_DOWN, startAction);

// 松开 j 键时，触发 stopAction 函数。
keh->addFunction('j',keyboardEventHandler::KEY_UP, stopAction);

// 进入仿真循环
// ……
```

9.5 处理键盘输入实现更新回调

9.5.1 本节目标

上一个教程我们讲解了键盘事件处理器类，它用于注册响应函数。本章提供了用于键盘输入的更方便的方案。我们将重载一个 GUIEventHandler 类，而不必再创建和注册函数。在这个类中我们将添加新的代码，以便执行特定的键盘和鼠标事件响应动作。我们还将提出一种键盘事件处理器与更新回调通讯的方法。

9.5.2 问题的提出

教程第 8 部分演示了如何将回调与 DOF 节点相关联，以实现场景中 DOF 节点位置的持续更新。那么，如果我们希望使用键盘输入来控制场景图形中的节点，应该如何处理呢？例如，如果我们有一个基于位置变换节点的坦克模型，并希望在按下 w 键的时候控制坦克向前运动，我们需要进行如下一些操作：

1. 读取键盘事件；
2. 保存键盘事件的结果；
3. 在更新回调中响应键盘事件。

9.5.3 解决方案

第一步：基类 osgGA::GUIEventHandler 用于定义用户自己的 GUI 键盘和鼠标事件动作。我们可以从基类派生自己的类并重载其 handle 方法，以创建自定义的动作。同时还编写 accept 方法来实现 GUIEventHandlerVisitor（OSG 2.0 版本中此类已经废弃）的功能。其基本的框架结构如下所示：

```
class myKeyboardEventHandler : public osgGA::GUIEventHandler
{
public:
virtual bool handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter&);
virtual void accept(osgGA::GUIEventHandlerVisitor& v) { v.visit(*this); };
};

bool myKeyboardEventHandler::handle(const
osgGA::GUIEventAdapter&ea,osgGA::GUIActionAdapter& aa)
{
switch(ea.getEventType())
{
case(osgGA::GUIEventAdapter::KEYDOWN):
{
switch(ea.getKey())
{
case 'w':
std::cout << " w key pressed" << std::endl;
return false;
break;
default:
return false;
}
}
default:
return false;
}
}
```

上述类的核心部分就是我们从基类中重载的 handle 方法。这个方法有两个参数：一个 GUIEventAdapter 类的实例，用于接收 GUI 事件；另一个是 GUIActionAdapter 类的实例，用于生成并向 GUI 系统发送请求，例如重绘请求和持续更新请求。

我们需要根据第一个参数编写代码以包含更多的事件，例如 KEYUP，DOUBLECLICK，DRAG 等。如果要处理按下按键的事件，则应针对 KEYDOWN 这个分支条件来扩展相应的代码。

事件处理函数的返回值与事件处理器列表中当前处理器触发的键盘和鼠标事件相关。如果返回值为 true，则系统认为事件已经处理，不再传递给下一个事件处理器。如果返回值为 false，则传递给下一个事件处理器，继续执行对事件的响应。

为了“安装”我们的事件处理器，我们需要创建它的实例并添加到 osgViewer::Viewer 的事件处理器列表。代码如下：

```
myKeyboardEventHandler* myFirstEventHandler = new myKeyboardEventHandler();
```

```
viewer.getEventHandlerList().push_front(myFirstEventHandler);
```
第二步：到目前为止，我们的键盘处理器还并不完善。它的功能仅仅是在每次按下 w 键时向控制窗口输出。如果我们希望按下键时可以控制场景图形中的元素，则需要在键盘处理器和更新回调之间建立一个通讯结构。为此，我们将创建一个用于保存键盘状态的类。这个事件处理器类用于记录最近的键盘和鼠标事件状态。而更新回调类也需要建立与键盘处理器类的接口，以实现场景图形的正确更新。现在我们开始创建基本的框架结构。用户可以在此基础上进行自由的扩展。下面的代码是一个类的定义，用于允许键盘事件处理器和更新回调之间通讯。

```
class tankInputDeviceStateType
{
public:
tankInputDeviceStateType::tankInputDeviceStateType() :
moveFwdRequest(false) {}
bool moveFwdRequest;
};
```
下一步的工作是确认键盘事件处理器和更新回调都有正确的数据接口。这些数据将封装到 tankInputdeviceStateType 的实例中。因为我们仅使用一个事件处理器来控制坦克，因此可以在事件处理器中提供指向 tankInputDeviceStateType 实例的指针。我们将向事件处理器添加一个数据成员（指向 tankInputDeviceStateType 的实例）。同时我们还会将指针设置为构造函数的输入参量。以上所述的改动，即指向 tankInputDeviceStateType 实例的指针，以及新的构造函数如下所示：

```
class myKeyboardEventHandler : public osgGA::GUIEventHandler {
public:
myKeyboardEventHandler(tankInputDeviceStateType* tids)
{
tankInputDeviceState = tids;
}
// ……
protected:
tankInputDeviceStateType* tankInputDeviceState;
};
```
我们还需要修改 handle 方法，以实现除了输出到控制台之外更多的功能。我们通过修改标志参量的值，来发送坦克向前运动的请求。

```
bool myKeyboardEventHandler::handle(const osgGA::GUIEventAdapter&
ea,osgGA::GUIActionAdapter& aa)
{
switch(ea.getEventType())
{
case(osgGA::GUIEventAdapter::KEYDOWN):
{
switch(ea.getKey())
{
case 'w':
tankInputDeviceState->moveFwdRequest = true;
return false;
```

```
break;
default:
return false;
}
}
default:
return false;
}
}
```
第三步：用于更新位置的回调类也需要编写键盘状态数据的接口。我们为更新回调添加与上述相同的参数。这其中包括一个指向同一 tankInputDeviceStateType 实例的指针。类的构造函数则负责将这个指针传递给成员变量。获得指针之后，我们就可以在回调内部使用其数值了。目前的回调只具备使坦克向前运动的代码，前提是用户执行了相应的键盘事件。回调类的内容如下所示：
```
class updateTankPosCallback : public osg::NodeCallback {
public:
updateTankPosCallback::updateTankPosCallback(tankInputDeviceStateType* tankIDevState)
: rotation(0.0) , tankPos(-15.,0.,0.)
{
tankInputDeviceState = tankIDevState;
}
virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
{
osg::PositionAttitudeTransform* pat =
dynamic_cast<osg::PositionAttitudeTransform*> (node);
if(pat)
{
if (tankInputDeviceState->moveFwdRequest)
{
tankPos.set(tankPos.x()+.01,0,0);
pat->setPosition(tankPos);
}
}
traverse(node, nv);
}
protected:
osg::Vec3d tankPos;
tankInputDeviceStateType* tankInputDeviceState;
};
```
现在，键盘和更新回调之间的通讯框架已经基本完成。下一步是创建一个 tankInputDeviceStateType 的实例。这个实例将作为事件处理器构造函数的参数传入。同时它也是模型位置更新回调类的构造函数参数。当事件处理器添加到视口的事件处理器列表中之后，我们就可以进入仿真循环并执行相应的功能了。
```
// 定义用于记录键盘事件的类的实例。
tankInputDeviceStateType* tIDevState = new tankInputDeviceStateType;
```

```
// 设置坦克的更新回调。
// 其构造函数将传递上面的实例指针作为实参。
tankPAT->setUpdateCallback(new updateTankPosCallback(tIDevState));

// 键盘处理器类的构造函数同样传递上面的实例指针作为实参。
myKeyboardEventHandler* tankEventHandler = new myKeyboardEventHandler(tIDevState);

// 将事件处理器压入处理器列表。
viewer.getEventHandlerList().push_front(tankEventHandler);

// 设置视口并进入仿真循环。
viewer.setSceneData( root );
viewer.realize();

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

不过用户还有更多可以扩展的地方：例如使坦克在按键松开的时候停止运动，转向，加速等等。

10.使用自定义矩阵来放置相机（Positioning a Camera with a User-Defined Matrix）

10.1 本章目标

手动放置相机，以实现场景的观览。

10.2 设置矩阵的方向和位置

我们可以使用 osg::Matrix 类来设置矩阵的数据。本章中我们将使用双精度类型的矩阵类 osg::Matrixd。要设置矩阵的位置和方向,我们可以使用矩阵类的 makeTranslate()和 makeRotate()方法。为了方便起见，这两个方法均提供了多种可重载的类型。本例中我们使用的 makeRotate()方法要求三对角度/向量值作为输入参数。旋转量由围绕指定向量轴所旋转的角度（表示为弧度值）决定。这里我们简单地选用 X，Y，Z 直角坐标系作为旋转参照的向量轴。将平移矩阵右乘旋转矩阵后，即可创建一个单一的表示旋转和平移的矩阵。代码如下：

如下是设置场景的代码。此场景包括一个小型的地形和坦克模型。坦克位于（10，10，8）的位置。

```
int main()
{
```

```
osg::Node* groundNode = NULL;
osg::Node* tankNode = NULL;
osg::Group* root = new osg::Group();
osgProducer::Viewer viewer;
osg::PositionAttitudeTransform* tankXform;

groundNode = osgDB::readNodeFile("\\Models\\JoeDirt\\JoeDirt.flt");
tankNode = osgDB::readNodeFile("\\Models\\T72-Tank\\T72-tank_des.flt");

// 创建绿色的天空布景。
osg::ClearNode* backdrop = new osg::ClearNode;
backdrop->setClearColor(osg::Vec4(0.0f,0.8f,0.0f,1.0f));
root->addChild(backdrop);
root->addChild(groundNode);

tankXform = new osg::PositionAttitudeTransform();

root->addChild(tankXform);
tankXform->addChild(tankNode);
tankXform->setPosition( osg::Vec3(10,10, );
tankXform->setAttitude(
osg::Quat(osg::DegreesToRadians(-45.0), osg::Vec3(0,0,1) ) );

osgGA::TrackballManipulator *Tman = new osgGA::TrackballManipulator();
viewer.setCameraManipulator(Tman);
viewer.setSceneData( root );

viewer.realize();
```

10.3 声明一个用于设置相机的矩阵

矩阵的位置设置为坦克模型后方 60 个单元，上方 7 个单元。同时设置矩阵的方向。

```
osg::Matrixd myCameraMatrix;
osg::Matrixd cameraRotation;
osg::Matrixd cameraTrans;
cameraRotation.makeRotate(
osg::DegreesToRadians(-20.0), osg::Vec3(0,1,0), // 滚转角（Y 轴）
osg::DegreesToRadians(-15.0), osg::Vec3(1,0,0) , // 俯仰角（X 轴）
osg::DegreesToRadians( 10.0), osg::Vec3(0,0,1) ); // 航向角（Z 轴）

// 相机位于坦克之后 60 个单元，之上 7 个单元。
cameraTrans.makeTranslate( 10,-50,15 );

myCameraMatrix = cameraRotation * cameraTrans;
```

10.4 使用矩阵设置视口摄相机

场景的视口类实例使用当前 MatrixManipulator 控制器类（TrackballManipulator，DriveManipulator 等）矩阵的逆矩阵来设置主摄像机的位置。为了在视口中使用我们自定义的摄像机位置和方向矩阵，我们需要首先计算自定义矩阵的逆矩阵。
除了求取逆矩阵之外，我们还需要提供世界坐标系的方向。通常 osgGA::MatrixManipulator 矩阵（osgProducer::Viewer 中使用）使用的坐标系为 Z 轴向上。但是 Producer 和 osg::Matrix（也就是上文所创建的）使用 Y 轴向上的坐标系系统。因此，在获得逆矩阵之后，我们需要将其从 Y 轴向上旋转到 Z 轴向上的形式。这一要求可以通过沿 X 轴旋转-90 度来实现。其实现代码如下所示：

```
while( !viewer.done() )
{
if (manuallyPlaceCamera)
{
osg::Matrixd i = myCameraMatrix.inverse(myCameraMatrix);
Tman->setByInverseMatrix(
osg::Matrix(i.ptr() )
* osg::Matrix::rotate( -3.1415926/2.0, 1, 0, 0 ) );
}
viewer.frame();
}
```

注意：按下 V 键可以手动切换摄像机。

11. 实现跟随节点的相机

提示：自 OSG 0.9.7 发布之后，新的 osgGA::MatrixManipulator 类（TrackerManipulator）允许用户将摄相机“依附”到场景图形中的节点。这一新增的操纵器类可以高效地替代下面所述的方法。

本章教程将继续使用回调和节点路径（NodePath）来检索节点的世界坐标。

11.1 本章目标

在一个典型的仿真过程中，用户可能需要从场景中的各种车辆和人物里选择一个进行跟随。本章将介绍一种将摄像机“依附”到场景图形节点的方法。此时视口的摄像机将跟随节点的世界坐标进行放置。

11.2 概述

视口类包括了一系列的矩阵控制器（osgGA::MatrixManipulator）。因而提供了“驱动控制（Drive）”，“轨迹球（Trackball）”，“飞行（Fly）”等交互方法。矩阵控制器类用于更新摄像机位置矩阵。它通常用于回应 GUI 事件（鼠标点击，拖动，按键，等等）。本文所述的功能需要依赖于相机位置矩阵，并参照场景图形节点的世界坐标。这样的话，相机就可以跟随场景图形中

的节点进行运动了。

为了获得场景图形中节点的世界坐标，我们需要使用节点访问器的节点路径功能来具现一个新的类。这个类将提供一种方法将自己的实例关联到场景图形，并因此提供访问任意节点世界坐标的方法。此坐标矩阵（场景中任意节点的世界坐标）将作为相机位置的矩阵，由 osgGA::MatrixManipulator 实例使用。

11.3 实现

首先我们创建一个类，计算场景图形中的多个变换矩阵的累加结果。很显然，所有的节点访问器都会访问当前的节点路径。节点路径本质上是根节点到当前节点的所有节点列表。有了节点路径的实例之后，我们就可以使用场景图形的方法 computeWorldToLocal(osg::NodePath)来获取表达节点世界坐标的矩阵了。

这个类的核心是使用更新回调来获取某个给定节点之前所有节点的矩阵和。整个类的定义如下：

```
struct updateAccumulatedMatrix : public osg::NodeCallback
{
   virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
   {
      matrix = osg::computeWorldToLocal(nv-&gt;getNodePath() );
      traverse(node,nv);
   }
   osg::Matrix matrix;
};
```

下一步，我们需要在场景图形的更新遍历中启动回调类。因此，我们将创建一个类，其中包括一个 osg::Node 实例作为数据成员。此节点数据成员的更新回调是上述 updateAccumulatedMatrix 类的实例，同时此节点也将设置为场景的一部分。为了读取用于描绘节点世界坐标的矩阵（该矩阵与节点实例相关联），我们需要为矩阵提供一个“get”方法。我们还需要提供添加节点到场景图形的方法。我们需要注意的是，用户应如何将节点关联到场景中。此节点应当有且只有一个父节点。因此，为了保证这个类的实例只有一个相关联的节点，我们还需要记录这个类的父节点。类的定义如下面的代码所示：

```
struct transformAccumulator
{
public:
   transformAccumulator();
   bool attachToGroup(osg::Group* g);
   osg::Matrix getMatrix();
protected:
   osg::ref_ptr&lt;osg::Group&gt; parent;
   osg::Node* node;
   updateAccumulatedMatrix* mpcb;
};
```

类的实现代码如下所示：

```
transformAccumulator::transformAccumulator()
{
```

```
   parent = NULL;
   node = new osg::Node;
   mpcb = new updateAccumulatedMatrix();
   node-&gt;setUpdateCallback(mpcb);
}

osg::Matrix transformAccumulator::getMatrix()
{
   return mpcb-&gt;matrix;
}

bool transformAccumulator::attachToGroup(osg::Group* g)
// 注意不要在回调中调用这个函数。
{
   bool success = false;
   if (parent != NULL)
   {
      int n = parent-&gt;getNumChildren();
      for (int i = 0; i &lt; n; i++)
      {
         if (node == parent-&gt;getChild(i) )
         {
            parent-&gt;removeChild(i,1);
            success = true;
         }
      }
      if (! success)
      {
         return success;
      }
   }
   g-&gt;addChild(node);
   return true;
}
```

现在，我们已经提供了类和方法来获取场景中节点的世界坐标矩阵，我们所需的只是学习如何使用这个矩阵来变换相机的位置。osgGA::MatrixManipulator 类即可提供一种更新相机位置矩阵的方法。我们可以从 MatrixManipulator 继承一个新的类，以实现利用场景中某个节点的世界坐标矩阵来改变相机的位置。为了实现这一目的，这个类需要提供一个数据成员，作为上述的 accumulateTransform 实例的句柄。新建类同时还需要保存相机位置矩阵的相应数据。
MatrixManipulator 类的核心是“handle”方法。这个方法用于检查选中的 GUI 事件并作出响应。对我们的类而言，唯一需要响应的 GUI 事件就是“FRAME”事件。在每一个“帧事件”中，我们都需要设置相机位置矩阵与 transformAccumulator 矩阵的数值相等。我们可以在类的成员中创建一个简单的 updateMatrix 方法来实现这一操作。由于我们使用了虚基类，因此某些方法必须在这里进行定义（矩阵的设置及读取，以及反转）。综上所述，类的实现代码如下所示：

```
class followNodeMatrixManipulator : public osgGA::MatrixManipulator
{
public:
   followNodeMatrixManipulator( transformAccumulator* ta);
   bool handle (const osgGA::GUIEventAdapter&ea, osgGA::GUIActionAdapter&aa);
   void updateTheMatrix();
   virtual void setByMatrix(const osg::Matrixd& mat) {theMatrix = mat;}
   virtual void setByInverseMatrix(const osg::Matrixd&mat) {}
   virtual osg::Matrixd getInverseMatrix() const;
   virtual osg::Matrixd getMatrix() const;
protected:
   ~followNodeMatrixManipulator() {}
   transformAccumulator* worldCoordinatesOfNode;
   osg::Matrixd theMatrix;
};
```

The class implementation is as follows:

```
followNodeMatrixManipulator::followNodeMatrixManipulator( transformAccumulator* ta)
{
   worldCoordinatesOfNode = ta; theMatrix = osg::Matrixd::identity();
}
void followNodeMatrixManipulator::updateTheMatrix()
{
   theMatrix = worldCoordinatesOfNode-&gt;getMatrix();
}
osg::Matrixd followNodeMatrixManipulator::getMatrix() const
{
   return theMatrix;
}
osg::Matrixd followNodeMatrixManipulator::getInverseMatrix() const
{
   // 将矩阵从 Y 轴向上旋转到 Z 轴向上
   osg::Matrixd m;
   m = theMatrix * osg::Matrixd::rotate(-M_PI/2.0, osg::Vec3(1,0,0) );
   return m;
}
void followNodeMatrixManipulator::setByMatrix(const osg::Matrixd& mat)
{
   theMatrix = mat;
}
void followNodeMatrixManipulator::setByInverseMatrix(const osg::Matrixd& mat)
{
   theMatrix = mat.inverse();
```

```
}

bool followNodeMatrixManipulator::handle
(const osgGA::GUIEventAdapter&ea, osgGA::GUIActionAdapter&aa)
{
   switch(ea.getEventType())
   {
      case (osgGA::GUIEventAdapter::FRAME):
      {
         updateTheMatrix();
         return false;
      }
   }
   return false;
}
```

上述的所有类都定义完毕之后，我们即可直接对其进行使用。我们需要声明一个transformAccumulator类的实例。该实例应当与场景图形中的某个节点相关联。然后，我们需要声明 nodeFollowerMatrixManipulator 类的实例。此操纵器类的构造函数将获取transformAccumulator实例的指针。最后，将新的矩阵操纵器添加到视口操控器列表中。上述步骤的实现如下：

```
// 设置场景和视口（包括 tankTransform 节点的添加）……
transformAccumulator* tankWorldCoords = new transformAccumulator();
tankWorldCoords-&gt;attachToGroup(tankTransform);
followNodeMatrixManipulator* followTank =
   new followNodeMatrixManipulator(tankWorldCoords);
osgGA::KeySwitchMatrixManipulator *ksmm = new osgGA::KeySwitchMatrixManipulator();
if (!ksmm)
   return -1;
// 添加跟随坦克的矩阵控制器的。按下“m”键即可实现视口切换到该控制器。
ksmm-&gt;addMatrixManipulator('m',"tankFollower",followTank);
viewer.setCameraManipulator(ksmm);
// 进入仿真循环……
```

11.4 环绕（始终指向）场景中节点的相机

11.4.1 本节目标

创建回调，以实现用于沿轨道环绕，同时指向场景中某个节点的世界坐标矩阵的更新。使用此矩阵的逆矩阵来放置相机。

11.4.2 实现

本章的回调类基于上一篇的osgFollowMe教程。本章中，我们将添加一个新的矩阵数据成员，以保存视口相机所需的世界坐标。每次更新遍历启动时，我们将调用环绕节点的当前轨道世界坐标矩阵。为了实现环绕节点的效果，我们将添加一个“angle”数据成员，其值每帧都会增加。矩阵的相对坐标基于一个固定数值的位置变换，而旋转量基于每帧更新的角度数据成员。为了实现相机的放置，我们还将添加一个方法，它将返回当前的轨道位置世界坐标。类的声明如下所示：

```
class orbit : public osg::NodeCallback
{
public:
   orbit(): heading(M_PI/2.0) {}

   osg::Matrix getWCMatrix(){return worldCoordMatrix;}

   virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
   {
      osg::MatrixTransform *tx = dynamic_cast<osg::MatrixTransform *>(node);
      if( tx != NULL )
      {
         heading += M_PI/180.0;
         osg::Matrixd orbitRotation;
         orbitRotation.makeRotate(
            osg::DegreesToRadians(-10.0), osg::Vec3(0,1,0), // 滚转角（Y 轴）
            osg::DegreesToRadians(-20.0), osg::Vec3(1,0,0) , // 俯仰角（X 轴）
            heading, osg::Vec3(0, 0, 1) ); // 航向角（Z 轴）
         osg::Matrixd orbitTranslation;
         orbitTranslation.makeTranslate( 0,-40, 4 );
         tx->setMatrix ( orbitTranslation * orbitRotation);
         worldCoordMatrix = osg::computeLocalToWorld( nv->getNodePath() );
      }
      traverse(node, nv);
   }
private:
   osg::Matrix worldCoordMatrix;
   float heading;
};
```

使用回调时，我们需要向场景添加一个矩阵变换，并将更新回调设置为“orbit”类的实例。我们使用前述 osgManualCamera 教程中的代码来实现用矩阵世界坐标来放置相机。我们还将使用前述键盘接口类的代码来添加一个函数来更新全局量，该全局量用于允许用户自行选择缺省和“环绕”的视口。

```
int main()
{
   osg::Node* groundNode = NULL;
   osg::Node* tankNode = NULL;
   osg::Group* root = NULL;
   osgViewer::Viewer viewer;
   osg::PositionAttitudeTransform* tankXform = NULL;

   groundNode = osgDB::readNodeFile("\\Models\\JoeDirt\\JoeDirt.flt");
   tankNode = osgDB::readNodeFile("\\Models\\T72-Tank\\T72-tank_des.flt");
```

```
root = new osg::Group();

// 创建天空。
osg::ClearNode* backdrop = new osg::ClearNode;
backdrop->setClearColor(osg::Vec4(0.0f,0.8f,0.0f,1.0f));
root->addChild(backdrop);

tankXform = new osg::PositionAttitudeTransform();
root->addChild(groundNode);
root->addChild(tankXform);
tankXform->addChild(tankNode);
tankXform->setPosition( osg::Vec3(10,10,8) );
tankXform->setAttitude(
   osg::Quat(osg::DegreesToRadians(-45.0), osg::Vec3(0,0,1) ) );

osgGA::TrackballManipulator *Tman = new osgGA::TrackballManipulator();
viewer.setCameraManipulator(Tman);

viewer.setSceneData( root );
viewer.realize();

// 创建矩阵变换节点，以实现环绕坦克节点。
osg::MatrixTransform* orbitTankXForm = new osg::MatrixTransform();
// 创建环绕轨道回调的实例。
orbit* tankOrbitCallback = new orbit();
// 为矩阵变换节点添加更新回调的实例。
orbitTankXForm->setUpdateCallback( tankOrbitCallback );
// 将位置轨道关联给坦克的位置，即，将其设置为坦克变换节点的子节点。
tankXform->addChild(orbitTankXForm);

keyboardEventHandler* keh = new keyboardEventHandler();
keh->addFunction('v',toggleTankOrbiterView);
viewer.addEventHandler(keh);

while( !viewer.done() )
{
   if (useTankOrbiterView)
   {
      Tman->setByInverseMatrix(tankOrbitCallback->getWCMatrix()
                        *osg::Matrix::rotate( -3.1415926/2.0, 1, 0, 0 ));
   }

   viewer.frame();
}
```

```
    return 0;
}
```

提示：按下 V 键来切换不同的视口。