

美国海军（NPS）的 OSG 教程

第一课	OpenSceneGraph 几何体的绘制.....	2
第二课	使用 StateSet 创建带有纹理的几何体.....	5
第三课	使用内嵌几何形状（Shape）对象，改变渲染状态	8
第四课	StateSet 的工作流程	10
第五课	从文件加载模型并放置在场景中	11
第六课	osgText，抬头显示（HUD），渲染元（RenderBin）	13
第七课	搜索并控制开关节点和 DOF（自由度）节点	17
第八课	使用更新回调来更改模型	23
第九课第一节	处理键盘输入	26
第九课第二节	处理键盘输入实现更新回调	29
第十课第一节	使用自定义矩阵来放置相机	32
第十课第二节	实现跟随节点的相机	34
第十课第三节	环绕（始终指向）场景中节点的相机	38
第十课第四节	如何获取节点在世界坐标的位置	40
第十一课第一节	使用两个独立的摄像机浏览场景	42
第十一课第二节	使用多个独立的摄像机观察场景（基于 OSG 1.2 版本）	47
第十二课第一节	使用 OpenGL 顶点着色器和片元着色器	52
第十二课第二节	向着色器传递变量数据	54
第十二课第三节	更新着色器	56
第十二课第四节	如何快速实现渐灰效果	59
第十四课	交集测试.....	65
第十五课第一节	向场景中添加 osgParticle 粒子效果	69
第十五课第二节	粒子系统的保存以及读取	74
第十六课第一节	节点遮掩概述（基于 OSG 1.2 版本）	75
第十六课第二节	节点遮掩示例（基于 OSG 1.2 版本）	76

第一课 OpenSceneGraph 几何体的绘制

本章将介绍一些创建几何体元素的方法。通常我们有这样几种处理几何体的手段：底层手段是使用松散封装的 OpenGL 基元；中级手段是使用 OpenSceneGraph 的基本几何体；高级手段是从文件读入模型。本章教程将主要介绍底层手段的实现方法。这种实现具有很强的灵活性，相应的工作量 也比较大。应用于场景图形级别的几何体通常是从文件读入的，因而顶点的跟踪和处理工作将由文件读取插件完成。

背景

下面将对几个常用的类作简要的介绍：

Geode 类

Geode 类派生自 Node 节点类。节点类（包括 Geode）可以作为场景图形的叶节点添加。Geode 类的实例可以与任意多个可绘制对象 Drawable 类相关联。

Drawable 类

作为可绘制对象基类的 Drawable 类是一个纯虚类，它有六个派生类。其中 Geometry 类中可以直接指定顶点数据，或者指定任意数目的几何基元 PrimitiveSet 类与其关联。

顶点和顶点属性数据（颜色，法线，纹理坐标）是保存在数组中的。多个顶点可以共享同一种颜色，法线和纹理坐标，同时我们还可以使用索引将顶点数组映射给颜色，法线或纹理坐标的数组。

PrimitiveSet 类

这个类松散地封装了 OpenGL 的绘图基元，包括点（POINTS），线（LINES），多段线（LINE_STRIP），封闭线（LINE_LOOP），四边形（QUADS），多边形（POLYGON）等。

代码

下面的代码将设置一个用于显示场景的视窗，一个作为场景图形根节点的 Group 类实例，一个用于记录可绘制对象（Drawable）的几何体节点（Geode），以及一个记录顶点和顶点相关数据的 Geometry 类实例。本例中我们将渲染一个金字塔的形状。

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

现在我们将金字塔几何体与 Geode 关联，并将 Geode 叶节点添加到场景图形的根节点。

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

声明一个顶点数组。每个顶点有三个坐标值，也就是一个 Vec3 类的实例。osg::Vec3Array 类的实例可以用来保存顶点数组。它派生自 STL 库的 vector 模板，因此我们可以使用 push_back 方法向其中追加数组元素。该方法的作用是在向量数组的末尾添加一个元素，因此数组中第一个元素的索引值为 0，紧随其后的第二个元素为 1，以此类推。

我们使用 Z 轴向上的右手坐标系作为参照，数组元素 0-4 用于表达金字塔形体的五个顶点。

```
osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
pyramidVertices->push_back( osg::Vec3( 0, 0, 0 ) ); // 左前
pyramidVertices->push_back( osg::Vec3(10, 0, 0 ) ); // 右前
pyramidVertices->push_back( osg::Vec3(10,10, 0 ) ); // 右后
pyramidVertices->push_back( osg::Vec3( 0,10, 0 ) ); // 左后
pyramidVertices->push_back( osg::Vec3( 5, 5,10 ) ); // 塔尖
```

将这一顶点集合关联到 Geometry 实例上，后者已经与场景的 Geode 叶节点相关联。

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

现在我们创建几何基元类 PrimitiveSet 的实例并添加到金字塔几何体上。金字塔底部的四个点组成一个基面，可以使用 DrawElementsUInt 类来实现。这个类同样继承自 STL 库的 vector 模板，我们可以使用 push_back 顺序向其中添加元素。为了保证背面剔除（backface culling）的正确，我们需要按照逆时针的顺序添加顶点数据。类的构造函数使用几何基元枚举类型（与 OpenGL 的几何基元枚举类型相同）作为输入 参数，另一个输入参数是作为起始点的顶点索引值。

```
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);
```

重复这一过程，添加金字塔的每个面。再次注意，顶点是以逆时针为顺序添加的。

```
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);

```

定义一个 Vec4 的数组，用于保存颜色值。

```

osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f)); //索引 0 红色
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f)); //索引 1 绿色
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f)); //索引 2 蓝色
colors->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f)); //索引 3 白色

```

下一步要将顶点数组的元素与颜色数组的元素对应起来。我们将声明一个与顶点数组有相同个数元素的向量组。它将负责连接各个顶点与颜色。该向量组的索引对应顶点数组的元素，其取值对应颜色数组的索引。如果需要将顶点数组与法线数组或者纹理坐标数组一一对应，那么还需重复这一步骤。

注意在本例中，我们需要将 5 个顶点对应到 4 种颜色上。因此顶点数组的元素 0（左下）和元素 4（塔尖）都需要对应到颜色数组元素 0（红色）上。

```

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，我们将颜色数组以及刚才创建的颜色索引数组与几何体相关联，并设置绑定模式为“按顶点绑定”。

```

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);

```

现在我们已经创建了一个几何体节点，并将其添加到场景中。其中的几何体是可以复用的，例如，如果我们希望在第一个金字塔右侧 15 个单位再放置第二个金字塔，那么我们可以将这个 Geode 节点作为位置变换节点的子节点，并再次添加到场景当中。

//初始化位置变换节点

```

osg::PositionAttitudeTransform* pyramidTwoXForm =
    new osg::PositionAttitudeTransform();

```

//使用 osg::Group 的 addChild 方法，将位置变换节点添加到根节点的子节点上，并将金字塔节点作为变换节点的子节点

```

root->addChild(pyramidTwoXForm);
pyramidTwoXForm->addChild(pyramidGeode);

```

// 初始化一个 Vec3 实例，用于改变模型在场景中的位置

```

osg::Vec3 pyramidTwoPosition(15, 0, 0);
pyramidTwoXForm->setPosition( pyramidTwoPosition );

```

最后，设置视窗类并进入仿真循环。

```

osgViewer::Viewer viewer;

viewer.setSceneData( root );

viewer.run();

```

第二课 使用 **StateSet** 创建带有纹理的几何体

目标：

向使用 OpenGL 基元绘制的[基本几何体](#)添加纹理。

背景 

上一个教程介绍了使用 OpenGL 基元创建基本几何体并在视窗中浏览的方法。本章将介绍如何向几何体添加纹理。为了使代码易于阅读，我们将有关金字塔绘制的代码封装到一个函数中，并返回一个 Geode 指针。代码如下所示：

```

osg::Geode* createPyramid()
{
    osg::Geode* pyramidGeode = new osg::Geode();
    osg::Geometry* pyramidGeometry = new osg::Geometry();
    pyramidGeode->addDrawable(pyramidGeometry);

    // 指定顶点
    osg::Vec3Array* pyramidVertices = new osg::Vec3Array;
    pyramidVertices->push_back( osg::Vec3(0, 0, 0) ); // 左前

```

```

pyramidVertices->push_back( osg::Vec3(2, 0, 0) ); // 右前
pyramidVertices->push_back( osg::Vec3(2, 2, 0) ); // 右后
pyramidVertices->push_back( osg::Vec3( 0,2, 0) ); // 左后
pyramidVertices->push_back( osg::Vec3( 1, 1,2) ); // 塔尖

```

// 将顶点数组关联给几何体

```

pyramidGeometry->setVertexArray( pyramidVertices );

```

// 根据底面的四个顶点创建底面四边形（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);

```

// 创建其他面的代码从略

```

osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //索引 0 红色
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //索引 1 绿色
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //索引 2 蓝色
colors->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); //索引 3 白色

```

```

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); // 顶点 0 对应颜色元素 0
colorIndexArray->push_back(1); // 顶点 1 对应颜色元素 1
colorIndexArray->push_back(2); // 顶点 2 对应颜色元素 2
colorIndexArray->push_back(3); // 顶点 3 对应颜色元素 3
colorIndexArray->push_back(0); // 顶点 4 对应颜色元素 0

```

```

pyramidGeometry->setColorArray(colors);
pyramidGeometry->setColorIndices(colorIndexArray);
pyramidGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);

```

//由于纹理坐标与顶点是一一对应的，因此不需要使用索引数组来进行映射。我们只需要直接使用 setTexCoordArray 方法即可。setTexCoordArray 方法传递 osg::Vec2 的二维坐标数组作为参数，传递的二维坐标数与顶点数相同。数组中元素的位置与顶点数组中顶点的位置一一对应。

```

osg::Vec2Array* texcoords = new osg::Vec2Array(5);
(*texcoords)[0].set(0.00f,0.0f); // 顶点 0 的纹理坐标
(*texcoords)[1].set(0.25f,0.0f); // 顶点 1 的纹理坐标
(*texcoords)[2].set(0.50f,0.0f); // 顶点 2 的纹理坐标
(*texcoords)[3].set(0.75f,0.0f); // 顶点 3 的纹理坐标
(*texcoords)[4].set(0.50f,1.0f); // 顶点 4 的纹理坐标

```

```
pyramidGeometry->setTexCoordArray(0, texcoords);
```

```
return pyramidGeode;
```

```
}
```

加载纹理，创建渲染状态类，关联到节点 [1](#)

我们使用 `StateSet` 类来控制几何基元的渲染状态。下面的代码演示了从文件中读取纹理，创建 `StateSet` 类并设置纹理，以及关联 `StateSet` 到场景节点的方法。代码的前一部分与上一章教程相同。首先我们初始化视窗并创建只有单一金字塔的场景。

```
int main()
```

```
{
```

```
// 声明场景的根节点
```

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

```
osg::Geode* pyramidGeode = createPyramid();
```

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

现在我们添加纹理。我们定义一个纹理实例并设置其数据变更类型为“DYNAMIC”（否则的话，OSG 的优化过程中可能会自动去除这个纹理）。纹理类封装了 OpenGL 纹理模式（`GL_TEXTURE_WRAP`，`GL_TEXTURE_FILTER` 等），以及一个 `osg::Image` 对象。下面的代码将演示如何从文件读入 `osg::Image` 实例并将其关联到纹理。

```
osg::Texture2D* KLN89FaceTexture = new osg::Texture2D;
```

```
//避免在优化过程中出错
```

```
KLN89FaceTexture->setDataVariance(osg::Object::DYNAMIC);
```

```
// 从文件读取图片
```

```
osg::Image* klnFace = osgDB::readImageFile("KLN89FaceB.tga");
```

```
if (!klnFace)
```

```
{
```

```
    std::cout << " couldn't find texture, quitting." << std::endl;
```

```
    return -1;
```

```
}
```

```
// 将图片关联到纹理
```

```
KLN89FaceTexture->setImage(klnFace);
```

`Texture` 类可以关联到渲染状态 `StateSet` 类。下一步我们将创建一个 `StateSet`，将纹理关联到这个渲染状态实例并允许使用纹理，最后将 `StateSet` 关联到几何体节点上。

```
// 创建 StateSet
```

```
osg::StateSet* stateOne = new osg::StateSet();
```

```
// 将纹理关联给 StateSet 的纹理单元 0
```

```
stateOne->setTextureAttributeAndModes
```

```
(0, KLN89FaceTexture, osg::StateAttribute::ON);
```

```
// 将渲染状态关联给金字塔节点
```

```
pyramidGeode->setStateSet(stateOne);
```

```
osgViewer::Viewer viewer;
```

```
//最后我们进入仿真循环:
```

```
viewer.setSceneData( root );
```

```
return viewer.run();  
}
```

第三课 使用内嵌几何形状（Shape）对象，改变渲染状态

目标 [1](#)

使用 `osg::Shape` 实例构建场景。使用 `osg::StateSet` 控制几何形状的渲染。

Shape 类的使用 [1](#)

`Shape` 类是各种内嵌几何形状的基类。它可以用于剔除和碰撞检测，或者设计并生成简单几何形体。下面这些类均派生自 `Shape` 类：

- `TriangleMesh`
- `Sphere`
- `InfinitePlane`
- `HeightField`
- `Cylinder`
- `Cone`
- `CompositeShape`
- `Box`

如果要渲染这些内嵌形体，我们需要首先将其与 `Drawable` 类的实例相关联。我们可以使用 `ShapeDrawable` 类来完成这一功能。它派生自 `Drawable` 类，并提供了关联 `Shape` 实例的方法。正因为 `ShapeDrawable` 是继承自 `Drawable` 的，它的实例因而可以被添加到 `Geode` 实例中。下面的代码演示了在场景中添加单位立方体的方法：

//创建场景的根节点

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

//声明 `Box` 类（派生自 `Shape`）的实例。构造函数的参数为：`osg::Vec3` 的中心位置，浮点数定义统一的高/宽/深度。

//（我们也可以分别输入不同的高度、宽度、深度值）

```
osg::Box* unitCube = new osg::Box( osg::Vec3(0,0,0), 1.0f);
```

//声明 `ShapeDrawable` 类的实例，使用刚才创建的 `unitCube` 作为传递参数。这个类派生自 `Drawable` 类，因此可以直接添加到 `Geode` 实例中。

```
osg::ShapeDrawable* unitCubeDrawable = new osg::ShapeDrawable(unitCube);
```

// 声明 `Geode` 类的实例

```
osg::Geode* basicShapesGeode = new osg::Geode();
```

// 将单位立方体添加到 `Geode` 中

```
basicShapesGeode->addDrawable(unitCubeDrawable);
```

// 将 `Geode` 添加到场景中

```
root->addChild(basicShapesGeode);
```

创建球的方法与此类同。如下面的代码所示：

// 在 origin 创建单位半径的球

```
osg::Sphere* unitSphere = new osg::Sphere( osg::Vec3(0,0,0), 1.0);
```



```
osg::ShapeDrawable* unitSphereDrawable = new osg::ShapeDrawable(unitSphere);
```

现在我们可以对球作位置变换后添加到场景中，以便将其与已经置于原点上的立方体区分开。注意 `unitSphereDrawable` 是不能直接添加到场景中的（它不是 `Node` 的派生类），因此我们需要为其创建一个新的 `Geode` 叶节点：

```
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);
```

设置渲染状态 [🔗](#)

前一个教程已经演示了创建[纹理](#)，从文件加载纹理图片，并关联纹理到 `StateSet` 对象的过程。下面的代码将设置两种不同的渲染状态：一个使用 `BLEND` 纹理模式，另一个使用 `DECAL` 模式。首先是 `BLEND` 模式：

```
//创建使用 BLEND 模式的渲染状态对象
```

```
osg::StateSet* blendStateSet = new osg::StateSet();
```

```
//声明 TexEnv 实例，设置模式为 BLEND
```

```
osg::TexEnv* blendTexEnv = new osg::TexEnv;
```

```
blendTexEnv->setMode(osg::TexEnv::BLEND);
```

```
// 使用纹理 0，也就是上一章加载的纹理
```

```
blendStateSet->setTextureAttributeAndModes
```

```
    (0,KLN89FaceTexture,osg::StateAttribute::ON);
```

```
// 设置纹理单元 0 所用的纹理环境
```

```
blendStateSet->setTextureAttribute(0,blendTexEnv);
```

现在我们重复刚才的过程，但是将纹理环境模式设置为 `DECAL`：

```
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);
```

现在我们将刚才创建的渲染状态对象关联到场景图形的节点。当场景图形执行绘制遍历（根节点->叶节点）时，渲染状态会暂存起来。如果某个节点没有与之关联的渲染状态，那么它将继承其父节点的渲染状态（换句话说，对于有不止一个父节点的节点来说，它可能有多种不同的渲染状态）。

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

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

最后我们进入仿真循环：

```
osgViewer::Viewer viewer;
```

```
viewer.setSceneData( root );
```

```
return viewer.run();
```

第四课 StateSet 的工作流程

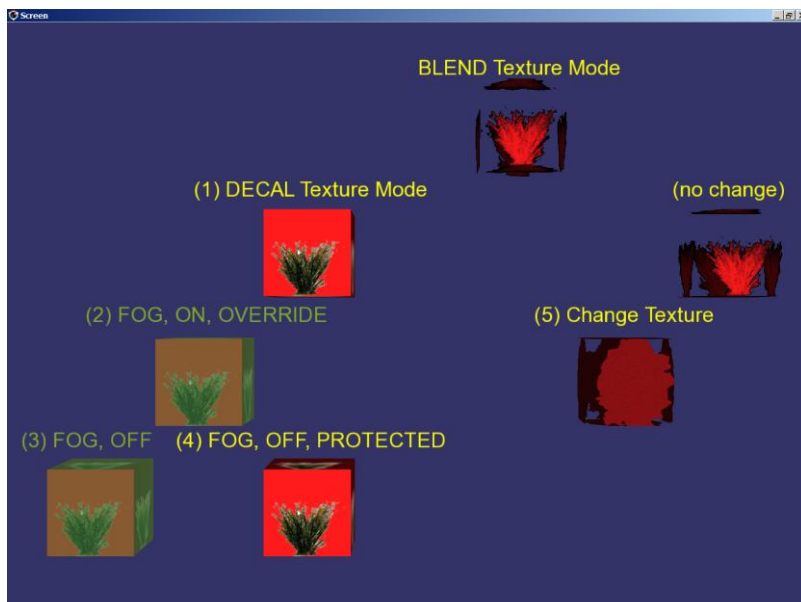
场景图形在遍历场景的过程中，会自动判断哪一些几何体要送入图形管道进行渲染。在遍历过程中，场景图形还要收集几何体渲染状态的信息。这些信息就保存在 `osg::StateSet` 实例中。`StateSet` 包含了一个 OpenGL 属性/值的列表，它可以与场景中的节点相关联。在渲染之前的遍历过程中，`StateSet` 会从根节点一直向叶节点进行积累。如果某个节点只是简单地继承上一节点的状态，那么与它关联的 `StateSet` 也不会发生改变。

我们可以使用一些附加的特性来提升工作的灵活性。渲染状态所保存的属性可以设置为 `OVERRIDE`，也就是说，该节点的所有子节点，无论其渲染状态是什么，都将继承父节点的渲染属性。但是 `OVERRIDE` 也可能存在一种例外，如果某个子节点的渲染属性设置为 `PROTECTED`，那么它将忽略父节点的相应属性，而使用自己设定的属性值。

示例 [1](#)

下面的例子演示了渲染状态对场景图形的影响。根节点使用 `BLEND` 模式。如果它的子节点没有改变任何渲染属性值，则子节点的渲染状态也不会发生改变，如根节点的右子树所示。右子树的节点没有指定渲染状态，因此它的渲染方式与根节点相同。而对于节点 5 来说，虽然纹理的混合模式没有改变，但是它已经关联了新的纹理。

根节点的左子树设置纹理模式为 `DECAL`，其它的渲染属性与根节点相同。节点 3 将 `FOG` 属性设置为 `ON` 并指定为 `OVERRIDE`。对于节点 2 的左子节点（节点 3），其 `FOG` 属性设置为 `OFF`，但是由于没有指定 `PROTECTED` 方式，且父节点为 `OVERRIDE` 方式，因此 `FOG` 属性依然为 `ON`。而对于右子节点(节点 4)，其 `FOG` 属性使用 `PROTECTED` 方式并设定为 `OFF`，此时它将忽略父节点的相应设置。



代码 [1](#)

下面的代码用于控制渲染状态的设定并将其关联到节点。

```
//设置 osg::TexEnv 实例为 BLEND 模式，将 TexEnv 属性和一幅纹理关联到纹理单元 0
blendTexEnv->setMode(osg::TexEnv::BLEND);
stateRootBlend->setTextureAttribute(0,blendTexEnv,osg::StateAttribute::ON);
stateRootBlend->setTextureAttributeAndModes(0,ocotilloTexture,osg::StateAttribute::ON);
```

```

// 对于节点 5，重新设置其渲染单元 0 的纹理。其它的渲染属性将继承父节点的属性（渲染模式属性仍为
BLEND）
stateFiveDustTexture->setTextureAttributeAndModes(0,dustTexture,osg::StateAttribute::ON);

// 设置 osg::TexEnv 实例为 DECAL 模式，将其关联到 stateOneDecal
decalTexEnv->setMode(osg::TexEnv::DECAL);
stateOneDecal->setTextureAttribute(0,decalTexEnv,osg::StateAttribute::ON);

// 对于 stateTwo，设置 FOG 属性为 ON 并设置 OVERRIDE，所有的子树将自动继承它的渲染状态，除非设置
了 PROTECTED 标志
stateTwoFogON_OVRD->setAttribute(fog, osg::StateAttribute::ON);
stateTwoFogON_OVRD->setMode(GL_FOG, osg::StateAttribute::ON | osg::StateAttribute::OVERRIDE);

// stateThree 试图关闭 FOG 属性，但是由于父节点的渲染状态设置为 OVERRIDE，因此 FOG 的设置没有发生变
化（仍为 ON）
stateThreeFogOFF->setMode(GL_FOG, osg::StateAttribute::OFF);

// stateFour 设置了 PROTECTED 标志，因此它可以重载父节点的渲染状态了
stateFourFogOFF_PROT->setMode(GL_FOG, osg::StateAttribute::OFF | osg::StateAttribute::PROTECTED);

// 将上面的各个渲染状态实例应用到场景图形的节点上
root->setStateSet(stateRootBlend);
mtOne->setStateSet(stateOneDecal);
mtTwo->setStateSet(stateTwoFogON_OVRD);
mtThree->setStateSet(stateThreeFogOFF);
mtSix->setStateSet(stateFiveDustTexture);
mtFour->setStateSet(stateFourFogOFF_PROT);

```

第五课 从文件加载模型并放置在场景中

目标 [1](#)

读取几何模型并添加到场景中，调整其中一个模型的位置并设置一个仿真循环来浏览场景。

读取几何模型并添加到场景 [1](#)

OpenSceneGraph 使用文件插件来读取各种格式的文件。目前它主要支持这样几种几何模型文件格式：3dc，3ds，flt，geo，iv，ive，lwo，md2，obj，osg 等，以及这些图片文件格式：bmp，gif，jpeg，rgb，tga，tif。OSG 的发行版附带了多个开放场景图形格式（.osg）的几何模型。在本例中我们将读取一个.osg 文件以及一个 MPI Open Flight（.flt）文件。为了方便系统搜索模型，我们可以在 OSG_FILE_PATH 所指定的路径（例如 C:\Projects\OpenSceneGraph\OpenSceneGraph-Data）下建立一个 models 文件夹，从以下地址下载模型文件并解压：

<http://www.nps.navy.mil/cs/sullivan/osgTutorials/Download/T72-tank.zip>

几何模型需要表示为场景图形的节点。因此，我们应当声明一个指向 osg::Node 实例的指针来加载和处理模型文件（需要增加一些头文件的支持）。

```
#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");

```

上面的语句是加载数据所必需的。现在我们可以将加载的模型作为场景图形的一部分了，将模型作为位置变换节点的子节点载入，以便控制它在场景中的位置。

```

// 声明一个作为场景图形根节点的组节点，它可以接受其它节点作为
// 自己的子节点，因此我们需要使用 Group 节点。
// Node 节点类是泛指所有类型的节点。而 Group 类是节点类型的一种，
// 它可以对子节点进行增减和控制。

osg::Group* root = new osg::Group();
root->addChild(cessnaNode);

// 声明并初始化位置变换节点

osg::PositionAttitudeTransform* tankXform =
    new osg::PositionAttitudeTransform();

//使用 osg::Group 的 addChild 方法将位置变换节点作为根节点
//的子节点添加，坦克节点作为位置变换节点的子节点添加。

root->addChild(tankXform);

tankXform->addChild(tankNode);

// 初始化一个 Vec3 的实例，用于确定坦克在场景中的位置。
osg::Vec3 tankPosit(5,0,0);
tankXform->setPosition( tankPosit );

```

现在我们的场景图形由一个根节点和两个子节点组成，其中一个是飞机（cessna.osg）的几何模型，另一个则是子树。子树中包含了位置变换节点和它的唯一子节点，坦克的几何模型。为了观察这个场景，我们设置一个视窗类并编写仿真循环。代码如下：

```

#include <osgViewer/Viewer>

// 声明视窗类
osgViewer::Viewer viewer;

// 将场景图形关联给视窗类
viewer.setSceneData( root );

// 添加视图的用户控制工具，即轨迹球漫游器
viewer.addCameraManipulator(new osgGA::TrackballManipulator);

```

```
// 创建程序窗口并启动工作线程
viewer.realize();

// 进入仿真循环。在用户按下 ESC 键之前，viewer.done()都会返回 false。
// 用户也可以使用自己的键盘/鼠标事件处理器来替换缺省的处理器
while( !viewer.done() )
{
    //执行新的一帧，其中封装了以下几种 Viewer 类操作：
    //  advance()通知程序向前移动一帧
    //  eventTraversal()收集事件并传递给事件处理器和回调
    //  updateTraversal()用于调用更新回调
    //  renderingTraversals()负责同步所有的渲染线程，并分配拣选，绘制和交换缓存
    viewer.frame();
}
```

现在我们可以编译和运行上面的代码了（注意代码的调用格式，还有 main 函数是否已经添加了）。在运行时按下 h 键将触发帮助菜单。按下 ESC 可以退出程序。

第六课 osgText, 抬头显示（HUD），渲染元（RenderBin）

目标 [🔗](#)

向场景添加文字 - 包括抬头显示信息（HUD）样式的文字，以及放置于场景中的文字。

概述: [🔗](#)

Text 文字类派生自 Drawable 类。换句话说，文字类的实例可以被添加到 Geode 中，并作为几何体渲染。Text 类的方法列表请参照 API 参考文档。此外，“Example osgtext”中也演示了各种方法的使用。本教程仅仅提供对 Text 类各种功能的简单介绍。绘制 HUD 包括两个基本概念：

1. 创建一个子树，其顶层节点设置了适当的投影和模型矩阵。
2. 将 HUD 子树的几何体关联到适当的渲染元（RenderBin），以确保 HUD 几何体在场景图形的其余部分全部渲染完毕后才进行渲染。

用于渲染 HUD 的子树包括了一个投影矩阵和一个模型矩阵。我们使用水平和垂直范围等于屏幕尺寸的正交投影矩阵，以保证文字坐标与像素坐标相等。为了保证模型正确，我们还需要使用单位矩阵作为模型观察矩阵。

渲染 HUD 时，我们需要将文字几何体关联到特定的渲染元（RenderBin）中。后者允许用户指定几何体绘制的顺序。这样我们就可以确保 HUD 几何体是在场景的最后进行渲染的。

代码 [🔗](#)

首先，我们声明所需的变量：osg::Text 和 osg::Projection 对象

```
osg::Group* root = NULL;
osg::Node* tankNode = NULL;
osg::Node* terrainNode = NULL;
osg::PositionAttitudeTransform* tankXform;

//HUD 几何体的叶节点
osg::Geode* HUDGeode = new osg::Geode();
// 用于作 HUD 显示的 Text 实例
osgText::Text* textOne = new osgText::Text();
```

```
// 这个文字实例将跟随坦克显示
osgText::Text* tankLabel = new osgText::Text();
// 投影节点用于定义 HUD 的视景体 (view frustrum)
osg::Projection* HUDProjectionMatrix = new osg::Projection;
```

从文件中读取模型，并设置场景图形的结构，参见前一章教程。

```
// 初始化根节点
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);

// 读入模型并关联到节点
tankNode = osgDB::readNodeFile("t72-tank_des.flt");
terrainNode = osgDB::readNodeFile("JoeDirt.flt");

// 初始化位置变换节点，用于放置坦克模型
tankXform = new osg::PositionAttitudeTransform();
tankXform->setPosition( osg::Vec3d(5,5,8) );

// 构件场景。将地形节点和位置变换节点关联到根节点
root->addChild(terrainNode);
root->addChild(tankXform);
tankXform->addChild(tankNode);
```

下一步，设置场景显示 HUD 文字。这里我们向场景中添加一个子树，其顶层节点使用投影和模型观察矩阵。

```
// 将投影矩阵添加到根节点。投影矩阵的水平和垂直范围与屏幕的尺寸
// 相同。因此该节点子树中的位置坐标将等同于像素坐标
HUDProjectionMatrix->setMatrix(osg::Matrix::ortho2D(0,1024,0,768));

// HUD 模型观察矩阵应使用单位矩阵
osg::MatrixTransform* HUDModelViewMatrix = new osg::MatrixTransform;
HUDModelViewMatrix->setMatrix(osg::Matrix::identity());

// 确保模型观察矩阵不会被场景图形的位置变换影响
HUDModelViewMatrix->setReferenceFrame(osg::Transform::ABSOLUTE_RF);

// 添加 HUD 投影矩阵到根节点，添加 HUD 模型观察矩阵到 HUD 投影矩阵。
// 模型观察矩阵节点的所有子节点都可以使用该投影矩阵进行视景
// 浏览，并使用模型观察矩阵来安置位置
root->addChild(HUDProjectionMatrix);
HUDProjectionMatrix->addChild(HUDModelViewMatrix);
```

现在我们设置几何体。我们创建一个与屏幕坐标对齐的四边形，并设置其颜色和纹理参数。

```

// 将包含 HUD 几何体的 Geode 节点作为 HUD 模型观察矩阵的子节点
HUDModelViewMatrix->addChild( HUDGeode );

// 设置 HUD 的四边形背景，并添加到 Geode 节点
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);

```

我们需要创建一个 `osg::StateSet`，并禁止深度测试（总是在屏幕上绘制），允许 Alpha 混合（使 HUD 背景透明），以保证 HUD 的渲染正确。然后我们使用一个指定数字的渲染元来分配几何体在拣选遍历中的渲染顺序。代码如下所示：

```
//设置渲染状态，使用上面定义的纹理
osg::StateSet* HUDStateSet = new osg::StateSet();
HUDGeode->setStateSet(HUDStateSet);
HUDStateSet->
    setTextureAttributeAndModes(0,HUDTexture,osg::StateAttribute::ON);
```

```
//打开 GL_BLEND 混合模式（以保证 Alpha 纹理正确）
HUDStateSet->setMode(GL_BLEND,osg::StateAttribute::ON);
```

```
// 禁止深度测试，因此几何体将忽略已绘制物体的深度值，直接进行绘制
HUDStateSet->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
HUDStateSet->setRenderingHint( osg::StateSet::TRANSPARENT_BIN );
```

// 确认该几何体在最后进行绘制。渲染元（RenderBin）按照数字顺序执行渲染，因此我们设置一个较大的数字值

```
HUDStateSet->setRenderBinDetails( 11, "RenderBin");
```

最后我们设置文字。由于 `osg::Text` 继承自 `osg::Drawable`，其实例可以作为 `osg::Geode` 实例的子节点添加到场景中。

```
// 添加文字到 Geode 叶节点中
HUDGeode->addDrawable( textOne );
```

```
// 设置 HUD 文字的参数
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) );
```

```
// 声明一个叶节点来保存坦克的标签文字
osg::Geode* tankLabelGeode = new osg::Geode();
```

```
// 将坦克标签添加到场景中
tankLabelGeode->addDrawable(tankLabel);
tankXform->addChild(tankLabelGeode);
```

```
// 设置坦克标签文字的参数，与坦克的坐标对齐，
// 这里使用 XZ_PLANE 保证文字与坦克的 XZ 平面对齐
tankLabel->setCharacterSize(5);
tankLabel->setFont("/fonts/arial.ttf");
tankLabel->setText("Tank #1");
tankLabel->setAxisAlignment(osgText::Text::XZ_PLANE);
```

```
// 设置文字渲染时包括一个对齐点和包围矩形
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，设置视景类并进入仿真循环。

```

osgViewer::Viewer viewer;

viewer.setSceneData( root );

return viewer.run();

```

第七课 搜索并控制开关节点和 DOF（自由度）节点

目标 [1](#)

对于包含开关节点和 DOF 节点的场景图形，搜索指定的节点并更新其内容。

搜索场景图形中的一个有名节点 [1](#)

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>

int main()
{
    osg::Node* tankNode = NULL;
    osg::Group* root = NULL;
    osg::Vec3 tankPosit;
    osg::PositionAttitudeTransform* tankXform;

    tankNode = osgDB::readNodeFile("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 );

osgViewer::Viewer viewer;

viewer.setSceneData( root );

return viewer.run();
}

```

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(); // returns NULL if not a group.
    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;
}

```

按照“访问器”模式搜索有名节点

“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）和拣选（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`），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>

```

#ifndef FIND_NODE_VISITOR_H
#define FIND_NODE_VISITOR_H

```

```

#include <osg/NodeVisitor>
#include <osg/Node>

class findNodeVisitor : public osg::NodeVisitor
{
public:

    // Default constructor - initialize searchForName to "" and
    // set the traversal mode to TRAVERSE_ALL_CHILDREN
    findNodeVisitor();

    // Constructor that accepts string argument
    // Initializes searchForName to user string
    // set the traversal mode to TRAVERSE_ALL_CHILDREN
    findNodeVisitor(const std::string &searchName);

    // The 'apply' method for 'node' type instances.
    // Compare the 'searchForName' data member against the node's name.
    // If the strings match, add this node to our list
    virtual void apply(osg::Node &searchNode);

    // Set the searchForName to user-defined string
    void setNameToFind(const std::string &searchName);

    // Return a pointer to the first node in the list
    // with a matching name
    osg::Node* getFirst();

    // typedef a vector of node pointers for convenience
    typedef std::vector<osg::Node*> nodeListType;

    // return a reference to the list of nodes we found
    nodeListType& getNodeList() { return foundNodeList; }

private:

    // the name we are looking for
    std::string searchForName;

    // List of nodes with names that match the searchForName string
    nodeListType foundNodeList;

};

#endif

```

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）“accept”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("\\Models\\t72-tank\\t72-tank_des.flt"));
tankTwoGroup = dynamic_cast<osg::Group*>
    (osgDB::readNodeFile("\\Models\\t72-tank\\t72-tank_des.flt"));
tankThreeGroup = dynamic_cast<osg::Group*>
    (osgDB::readNodeFile("\\Models\\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” “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”
//“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) );
}

```

就是这样了！然后再编写仿真循环的代码即可。

第八课 使用更新回调来更改模型

本章目标: [1](#)

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。

回调概览 [1](#)

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。

创建一个更新回调 [1](#)

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.....

```
osgViewer::Viewer viewer;

viewer.setSceneData(tankNode);

return viewer.run();
}
```

第九课第一节 处理键盘输入

本章目标： [1](#)

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。

GUI（图形用户接口）事件处理器：GUI 事件适配器和 GUI 动作适配器 [1](#)

`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`，`SCROLLEDOWN`，`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。

简单的键盘接口类 [1](#)

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;
    };
    typedef std::map<int, functionStatusType > keyFunctionMap;
```

// 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:

//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
```

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:
```

```
// ...
```

```
// 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.getEventHandlerList().push_front(keh);
```

```
// 注册键值，响应函数。
```

```
// 按下 a 键时，触发 toggleSomething 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
// ...

```

第九课第二节 处理键盘输入实现更新回调

本章目标: [91](#)

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。

问题的提出: [91](#)

教程 08 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。

解决方案: [91](#)

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，则需要从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);

osgViewer::Viewer viewer;

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

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

return viewer.run();
```

完成了！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。

第十课第一节 使用自定义矩阵来放置相机

本章目标: [1](#)

手动放置相机，以实现场景的观览。

Overview [1](#)

To position a camera manually, we can use the viewer class `setViewByMatrix()` method. This call should be placed between the `viewer.update` and `viewer.frame` calls within the simulation loop. The following guidelines are useful:

- Producer and all classes below `osgGA::MatrixManipulator` (in terms of abstraction) use 'Y' up coordinates. All others -including the Viewer class matrix manipulators - use 'Z' up.
- The inverse of a position/orientation matrix can be used to orient a camera.

With these guidelines, the only steps required to position a camera manually are:

1. Create and initialize a matrix with the correct world position and orientation.
2. Get the inverse of this matrix and...
3. Provide a world up orientation. In this case by rotating from 'Y' up to 'Z' up.

代码: [1](#)

设置矩阵的方向和位置 [1](#)

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(10,10,8) );
    tankXform->setAttitude(
        osg::Quat(osg::DegreesToRadians(-45.0), osg::Vec3(0,0,1) ) );

    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    viewer.setSceneData( root );

    viewer.realize();
```

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;
```

使用矩阵设置视口摄像机 [¶](#)

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() )
{
    viewer.sync();
    viewer.update();
    if (manuallyPlaceCamera)
    {
        osg::Matrixd i = myCameraMatrix.inverse(myCameraMatrix);
        viewer.setViewByMatrix( (
            Producer::Matrix(i.ptr() ))
            * Producer::Matrix::rotate( -M_PI/2.0, 1, 0, 0 ) );
    }
    viewer.frame();
}
```

注意：按下 V 键可以手动切换摄像机。

第十课第二节 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`）允许用户将摄像机“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。

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。de.

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。

代码： [¶](#)

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 updateAccumlatedMatrix : public osg::NodeCallback
{
    virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
    {
        matrix = osg::computeWorldToLocal(nv->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 `updateAccumlatedMatrix` 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<osg::Group> parent;
    osg::Node* node;
    updateAccumlatedMatrix* mpcb;
};
```

类的实现代码如下所示：

```
transformAccumulator::transformAccumulator()
{
    parent = NULL;
    node = new osg::Node;
    mpcb = new updateAccumlatedMatrix();
    node->setUpdateCallback(mpcb);
}

osg::Matrix transformAccumulator::getMatrix()
{
    return mpcb->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->getNumChildren();
        for (int i = 0; i < n; i++)
        {
            if (node == parent->getChild(i) )
            {
                parent->removeChild(i,1);
                success = true;
            }
        }
        if (! success)
        {
            return success;
        }
    }
    g->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;
};
```

类的实现代码如下：

```
followNodeMatrixManipulator::followNodeMatrixManipulator( transformAccumulator* ta)
{
    worldCoordinatesOfNode = ta; theMatrix = osg::Matrixd::identity();
}
void followNodeMatrixManipulator::updateTheMatrix()
{
    theMatrix = worldCoordinatesOfNode->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纵器列表中。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->attachToGroup(tankTransform);
followNodeMatrixManipulator* followTank =
    new followNodeMatrixManipulator(tankWorldCoords);
osgGA::KeySwitchMatrixManipulator *ksmm =
    viewer.getKeySwitchMatrixManipulator();
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->addMatrixManipulator('m',"tankFollower",followTank);

// ... 进入仿真循环...

```

第十课第三节 环绕（始终指向）场景中节点的相机

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。

代码: [🔗](#)

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;
    osgProducer::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) ) );

    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    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.getEventHandlerList().push_front(keh);

while( !viewer.done() )
{
    viewer.sync();
    viewer.update();

    if (useTankOrbiterView)
    {
        osg::Matrixd m = tankOrbitCallback->getWCMatrix();
        osg::Matrixd i = m.inverse(m);
        viewer.setViewByMatrix(
            Producer::Matrix( i.ptr() ) *
            Producer::Matrix::rotate( osg::DegreesToRadians(-90.0), 1, 0, 0 ) );
    }

    viewer.frame();
}
return 0;
}

```

提示：按下 **V** 键来切换不同的视口。

第十课第四节 如何获取节点在世界坐标的位置

获取某个节点的世界坐标（相对世界原点的位置），有一种方法是创建可以更新矩阵的回调函数。有关这一方法的示例可以参照前面的例子。但是这一方法的 弊病在于必须使用更新回调。回调被触发之后，无论用户需要与否，每帧都会自动计算矩阵坐标。如果你不喜欢这种方式的话，可以尝试使用访问器。

使用节点访问器计算世界原点的相对位置 [\[1\]](#)

对于场景图形中的一个 **OSG** 节点，在它和根节点之间可能会有其它的一些变换节点存在。那么应该如何获取节点在世界的坐标呢？从技术上讲是很难的。节点的每一个父节点都有且只有自己的变换矩阵。通常情况下，这些矩阵中包含了相对坐标数据。如果要计算目标节点的世界坐标的话，则需要将根节点和目标节点之间所有的矩阵相乘。访问器模式的使用解决了这一问题。访问器(Visitor)可以跟踪记录场景图形中节点遍历的路径。**OSG** 提供了相应的函数，用于获取 节点路径 **nodePath** 并计算基于路径上各个矩阵的世界坐标。

- 创建从叶节点到根节点的访问器。
- 对于目标节点，启动访问器。
- 访问器将遍历整个场景图形，以得到正确的节点路径。
- 到达根节点之后，计算得到目标节点的世界坐标。
- 使用访问器的 **nodePath** 计算节点世界坐标。

NOTE:

- A few caveats related to cameraNodes here.
- And some info on rotated models here.

C++格式的代码如下：

```
// 该访问器类用于返回某个节点的世界坐标。
// 它从起始节点开始向父节点遍历，并随时将历经的节点记录到 nodePath 中。
// ?第一次到达根节点之后，它将记录起始节点的世界坐标。连结起始节点到根节点路径上的
//所有矩阵之后，即可获得节点的世界坐标。
```

```
class getWorldCoordOfNodeVisitor : public osg::NodeVisitor
{
public:
    getWorldCoordOfNodeVisitor():
        osg::NodeVisitor(NodeVisitor::TRAVERSE_PARENTS), done(false)
    {
        wcMatrix= new osg::Matrixd();
    }
    virtual void apply(osg::Node &node)
    {
        if (!done)
        {
            if ( 0 == node.getNumParents() ) //到达根节点，此时节点路径也已记录完整
            {
                wcMatrix->set( osg::computeLocalToWorld(this->getNodePath()) );
                done = true;
            }
            traverse(node);
        }
    }
    osg::Matrixd* giveUpDaMat()
    {
        return wcMatrix;
    }
private:
    bool done;
    osg::Matrix* wcMatrix;
};
```

```
// 对于场景中的合法节点，返回 osg::Matrix 格式的世界坐标。
//用户创建用于更新世界坐标矩阵的访问器之后，既可获取该矩阵。
// （此函数也可以作为节点派生类的成员函数。）
```

```
osg::Matrixd* getWorldCoords( osg::Node* node)
{
    getWorldCoordOfNodeVisitor* ncv = new getWorldCoordOfNodeVisitor();
    if (node && ncv)
    {
        node->accept(*ncv);
    }
}
```

```

        return ncv->giveUpDaMat();
    }
    else
    {
        return NULL;
    }
}

```

第十一课第一节 使用两个独立的摄像机浏览场景

目标 [¶](#)

在场景中创建两个视口。其中一个用于从坦克驾驶员的视角观察场景。该视口将被渲染于屏幕的上半部分。第二个视口由缺省的 `osgViewer::Viewer` 类接口（轨迹球，飞行等控制器）控制。它将被渲染于屏幕的中下部分。

概述: [¶](#)

OSG 向开发人员提供了各种的抽象层次接口。前面的教程讨论的主要是一些较高层级的接口应用：例如使用 `Viewer` 类来控制视点，场景，交互设备和 窗口系统。OSG 的优势之一，就是可以允许开发者在使用高层次的接口的同时，访问较低层次的抽象接口。本章将使用一些低抽象层级的功能，对视点进行控制， 并使用相应的类渲染场景。

代码: [¶](#)

为了创建两个视口，我们需要提供两个独立可控的摄像机。与 OSG 1.2 版本中所述不同的是，本例中将不再使用 `Prodecer::CameraConfig` 类，而是将多个不同的视口添加到组合视口 `CompositeViewer` 类当中。下面的函数即用于实现添加视口并设置其中的摄像机位置。

```
void createView (osgViewer::CompositeViewer *viewer,
```

```
    osg::ref_ptr<osg::Group> scene,
```

```
    osg::ref_ptr<osg::GraphicsContext> gc,
```

```
    osgGA::TrackballManipulator* Tman,
```

```
    int x, int y, int width, int height)
```

```
{
```

```
    double left,right,top,bottom,near,far, aspectratio;
```

```
    double frusht, fruswid, fudge;
```

```
    bool gotfrustum = true;
```

```
    // 向最终的组合视口添加一个新的视口，并设置其操控方式。
```

```

osgViewer::View* view = new osgViewer::View;

viewer->addView(view);

view->setCameraManipulator(Tman);

// 设置视口的场景数据，并设置摄像机的截锥坐标。

view->setSceneData(scene.get());

view->getCamera()->setViewport(new osg::Viewport(x,y, width,height));

view->getCamera()-> getProjectionMatrixAsFrustum(left,right,

bottom,top,

near,far);

if (gotfrustum)

{

aspectratio = (double) width/ (double) height;

frusht = top - bottom;

fruswid = right - left;

fudge = frusht*aspectratio/fruswid;

right = right*fudge;

left = left*fudge;

view->getCamera()-> setProjectionMatrixAsFrustum(left,right,

bottom,top,

near,far);

}

view->getCamera()->setGraphicsContext(gc.get());

// 添加渲染状态控制器

```

```

osg::ref_ptr<osgGA::StateSetManipulator> statesetManipulator = new osgGA::StateSetManipulator;

statesetManipulator->setStateSet(view->getCamera()->getOrCreateStateSet());

view->addEventHandler( statesetManipulator.get() );

}

```

现在我们已经有了设置摄像机的函数，在仿真的其余部分中，我们将不再赘述有关基本场景建立（包括一个地形模型以及在其上运动的坦克）的内容。相关的代码可以从源程序中获取。我们需要对坦克模型添加一个位移变换节点。这样我们就可以将摄像机的位置置于坦克的后上方，以便进行观察。

```
int main( int argc, char **argv )
```

```

{

// 场景根节点和坦克模型节点指针。

osg::ref_ptr<osg::Group> rootNode;

osg::ref_ptr<osg::Group> ownTank;


osgGA::TrackballManipulator *Tman1 = new osgGA::TrackballManipulator();

osgGA::TrackballManipulator *Tman2 = new osgGA::TrackballManipulator();


// 建立场景和坦克。

if (!setupScene(rootNode, ownTank))

{

std::cout<< "problem setting up scene" << std::endl;

return -1;

}

// 声明一个位于坦克偏后上方的位移变换节点。将其添加到坦克节点。

osg::PositionAttitudeTransform * followerOffset =

new osg::PositionAttitudeTransform();

```

```

followerOffset->setPosition( osg::Vec3(0.0,-25.0,10) );

followerOffset->setAttitude(

osg::Quat( osg::DegreesToRadians(-15.0), osg::Vec3(1,0,0) ) );

ownTank.get()->addChild(followerOffset);

// 声明一个自定义的位移累加器类，以便放置相机。将其关联给上面的变换节点。

transformAccumulator* tankFollowerWorldCoords = new transformAccumulator();

tankFollowerWorldCoords->attachToGroup(followerOffset);

// 构建视口类，以及与其相关的图形设备类。

osgViewer::CompositeViewer viewer;

osg::GraphicsContext::WindowingSystemInterface* wsi =

osg::GraphicsContext::getWindowingSystemInterface();

if (!wsi)

{

osg::notify(osg::NOTICE)

<<"Error, no WindowSystemInterface available, cannot create windows."<<std::endl;

return 1;

}

unsigned int width, height;

wsi->getScreenResolution(osg::GraphicsContext::ScreenIdentifier(0), width, height);

osg::ref_ptr<osg::GraphicsContext::Traits> traits = new osg::GraphicsContext::Traits;

traits->x = 100;

traits->y = 100;

traits->width = width;

```

```

traits->height = height;

traits->windowDecoration = true;

traits->doubleBuffer = true;

traits->sharedContext = 0;

osg::ref_ptr<osg::GraphicsContext> gc = osg::GraphicsContext::createGraphicsContext(traits.get());

if (gc.valid())

{

    osg::notify(osg::INFO)<<"  GraphicsWindow has been created successfully."<<std::endl;

    gc->setClearColor(osg::Vec4f(0.2f,0.2f,0.6f,1.0f));

    gc->setClearMask(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

}

else

{

    osg::notify(osg::NOTICE)<<"  GraphicsWindow has not been created successfully."<<std::endl;

}


// 第一个视口

createView (&viewer,rootNode,gc,Tman1,0, 0, traits->width/2, traits->height);

// 第二个视口

createView (&viewer,rootNode,gc,Tman2,traits->width/2, 0, traits->width/2, traits->height);

viewer.setThreadingModel(osgViewer::CompositeViewer::SingleThreaded);

```

现在我们已经基本建立了仿真的代码。它与前述的程序有少许不同。在每次更新场景图形节点之后，我们都会手动重置控制器的位置，从而设置摄像机的方向和跟随效果。另一个视口的摄像机则保持缺省的位置和接口不变。代码如下所示：

```

while( !viewer.done() )

{

    // 获取跟踪摄像机的句柄。使用相应的方法设置坦克跟踪相机的世界坐标位置矩阵。

    // 注意该矩阵需要从 Y 轴向上旋转到 Z 轴向上的坐标系。

    Tman2->setByInverseMatrix(tankFollowerWorldCoords->getMatrix()

    *osg::Matrix::rotate( -M_PI/2.0, 1, 0, 0 ));

viewer.frame();

}

return 0;

}

```

第十一课第二节 使用多个独立的摄像机观察场景（基于 OSG 1.2 版本）

本教程基于 OSG 1.2 版本）目标： [1](#)

为场景创建多个视图。它们各自独立地进行视口控制。

- 添加一个“坦克驾驶员”视口（稍稍偏上偏后）。驾驶员视口可以观察屏幕高度的 1/3 和整个宽度范围。
- 添加三个关联到炮塔的独立视口。分别对应炮筒的左侧，中间和右侧。这些视图的方向与炮塔的航向角相对应，并各自有一定量的偏移（其中右视图向右偏移 45 度）。炮筒的左，中，右侧视图可以观察到场景底部的 1/3 空间，每个视图均可观察显示器的 1/3 宽度。

概述： [1](#)

OpenSceneGraph 提供了各种级别的开发方法：就较高的抽象层级来说，程序员可以使用 `osgProducer::Viewer` 类的实例。`Viewer` 类封装了视景仿真的所需的各种基本元素：键盘和鼠标接口，输入域，渲染表面，场景管理和帧控制。开发者可以自由地使用这些基本元素进行编程，也可以创建（或修改）自己的封装类。OSG 的一大优势是程序员可以在一个较高的层级上进行开发，而不用关心底层的实现。在本教程中，我们将 `Viewer` 的实例与低层级的 `Producer::Camera` 结合起来，创建场景的各个独立视图以实现坦克的操作。

有关 `Producer` 的基本概念（摄像机，焦距，渲染表面，输入域），以及 `Camera` 类的使用文档，请参见：

<http://www.andesengineering.com/Producer/Tutorial/>

本教程将使用 `Producer::CameraConfig` 类的实例来实现多个独立摄像机的创建和控制。我们将使用 `Producer::Lens` 类方法直接控制摄像机的焦距。

首先，我们需要创建一个摄像机配置 `CameraConfig` 实例，以设置不同摄像机和焦距的参数，并指定位置和方向。配置实例可以作为 `Viewer` 类的参数输入，用于描述摄像机参数和控制方式。为了保证模块性，我们现在需要编写函数来创建所需的摄像机配置实例。

```

enum cameraIndex
{
    DRIVER_CAMERA,
    GUNNER_LEFT_CAMERA,

```

```

    GUNNER_CENTER_CAMERA,
    GUNNER_RIGHT_CAMERA
};

Producer::CameraConfig* setupCameras()
{
    Producer::CameraConfig* tankCameraConfig =
        new Producer::CameraConfig();
    Producer::Camera* tankCameras[4];
    std::string cameraNames[4];
    cameraNames[0]= "Driver";
    cameraNames[1]= "GunnerLeft";
    cameraNames[2]= "GunnerCenter";
    cameraNames[3]= "GunnerRight";
    for (int i=0;i<4;i++)
    {
        tankCameras[i] = new Producer::Camera();
        tankCameras[i]->getLens()->setAutoAspect(true);
        tankCameras[i]->setShareLens(false);
        tankCameraConfig->addCamera(cameraNames[i],tankCameras[i]);
    }

    tankCameras[DRIVER_CAMERA]->
        setProjectionRectangle(0.05f, 0.95f, 0.4f, 0.8f);

    tankCameras[GUNNER_LEFT_CAMERA]->
        setProjectionRectangle(0.00, 0.30, 0.05f, 0.35);
    tankCameras[GUNNER_CENTER_CAMERA]->
        setProjectionRectangle(0.35f, 0.65f, 0.05f, 0.35f);
    tankCameras[GUNNER_RIGHT_CAMERA]->
        setProjectionRectangle(0.70f, 1.0f, 0.05f, 0.35f);

    Producer::RenderSurface* rsOne =
        tankCameras[DRIVER_CAMERA]->getRenderSurface();

    tankCameras[GUNNER_RIGHT_CAMERA]->setRenderSurface( rsOne );
    tankCameras[GUNNER_CENTER_CAMERA]->setRenderSurface( rsOne );
    tankCameras[GUNNER_LEFT_CAMERA]->setRenderSurface( rsOne );

    return tankCameraConfig;
}

```

现在我们已经有了摄像机的配置信息，下一步可以开始构建场景了。这里我们要添加三个主要模型：地形模型位于原点，一台损坏的坦克位于场景中的某个位置，另一台坦克 `ownTank` 则与视口相关联。它与报废坦克的距离很近，因此我们可以在场景中看到损坏的坦克。为了保证模块性，我们编写 `setupScene` 函数。设置场景并将视口与场景中的模型相关联，它将提供对于根节点和 `ownTank` 模型的处理。如下所示：

```

bool setupScene(Producer::ref_ptr<osg::Group> &rootNode,
                Producer::ref_ptr<osg::Group> &ownTank )

```



```

// 场景中包括一台损坏的坦克和观察者坦克。
// 坦克和根节点的指针将作为返回值，以便进行控制
// 读取模型失败时，函数返回 false。
{
    if (rootNode.get() || ownTank.get())
        return false;

    rootNode = 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\\");
    osgDB::setDataFilePathList(pathList);

    Producer::ref_ptr<osg::Node> terrainNode =
        osgDB::readNodeFile("JoeDirt.flt");
    if (!terrainNode)
    {
        std::cout << " no terrain! " << std::endl;
        return false;
    }
    rootNode->addChild(terrainNode.get());

    ownTank = (osg::Group*)
        osgDB::readNodeFile("T72-tank_des.flt");
    if ( ! ownTank)
    {
        std::cout << "no Tank" << std::endl;
        return false;
    }
    osg::PositionAttitudeTransform* ownTankPAT =
        new osg::PositionAttitudeTransform();
    ownTankPAT->setPosition( osg::Vec3(100,100,8) );
    rootNode->addChild(ownTankPAT);
    ownTankPAT->addChild(ownTank.get());

    Producer::ref_ptr<osg::Node> damagedTank =
        osgDB::readNodeFile("T72-tank_des.flt");
    if ( ! damagedTank )
    {
        std::cout << "no Tank" << std::endl;
        return false;
    }
    osg::PositionAttitudeTransform* damagedTankPAT =
        new osg::PositionAttitudeTransform();

```

```

damagedTankPAT->setPosition( osg::Vec3(90,110,8) );
rootNode->addChild(damagedTankPAT);
damagedTankPAT->addChild(damagedTank.get());

findNodeVisitor findSwitch("sw1");
damagedTank->accept(findSwitch);
osgSim::MultiSwitch* damagedTankSwitch =
    dynamic_cast <osgSim::MultiSwitch*> (findSwitch.getFirst());
if (!damagedTankSwitch)
    return -1;
damagedTankSwitch->setSingleChildOn(0,true);

return true;
}

```

现在我们已经建立了基本的场景，可以编写仿真循环的其余部分了。下面的代码将配置摄像机的位置，使其与场景图形中节点的世界坐标相对应。为了获取节点的世界坐标，我们需要使用前面教程中有关 `transformAcumulator` 自定义类的代码。对于驾驶员视图，只需要根据坦克的位置，设置一个相对节点世界坐标较小的偏移值（向后上方）。然后使用场景当前的矩阵漫游器（`matrix manipulator`）来控制摄像机即可。

另外三个摄像机需要放置在炮塔附近，对于炮筒中间的视图，我们需要使用 `tranformAcumulator` 与炮塔节点的偏移值相关联（已知炮塔世界坐标与炮筒的偏移量）。对于左/右炮筒的世界坐标，我们需要对中间视图分别添加一个偏移变换。因此它们都需要与 `tranformAcumulator` 相关联，以便获得摄像机放置的位置。

```

int main( int argc, char **argv )
{
    Producer::ref_ptr<osg::Group> rootNode;
    Producer::ref_ptr<osg::Group> ownTank;

    // 构建地形和坦克模型
    if (!setupScene(rootNode,ownTank))
    {
        std::cout<< "problem setting up scene" << std::endl;
        return -1;
    }

    // 设置摄像机并初始化视景
    Producer::CameraConfig* tankCameras = setupCameras();
    osgProducer::Viewer viewer(tankCameras);

    //设置视景对象
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    viewer.setSceneData( rootNode.get() );

    //获取坦克炮塔节点的句柄
    findNodeVisitor findTurret("turret");
    ownTank->accept(findTurret);
    osgSim::DOFTransform* turretXForm = dynamic_cast <osgSim::DOFTransform*>
        (findTurret.getFirst());
}

```

// 设置炮筒与炮塔的偏移量

```
osg::PositionAttitudeTransform* gunnerXForm =  
    new osg::PositionAttitudeTransform();  
gunnerXForm->setPosition( osg::Vec3(0,-1.5,3.0) );  
turretXForm->addChild(gunnerXForm);
```

// 声明位移累加类 transformAccumulator，用于获取炮筒中心的世界坐标。将炮塔偏移量节点与这个类
相关联

```
transformAccumulator* gunnerWorldCoords = new transformAccumulator();  
gunnerWorldCoords->attachToGroup(gunnerXForm);
```

// ?定义左侧炮塔视图的变换值（相对垂直轴偏移 45 度）

// 关联到炮塔位移节点

```
osg::PositionAttitudeTransform* leftGunnerPAT =  
    new osg::PositionAttitudeTransform();  
leftGunnerPAT->setAttitude(  
    osg::Quat( osg::DegreesToRadians(45.0) , osg::Vec3(0,0,1) ));  
gunnerXForm->addChild(leftGunnerPAT);
```

//?定义一个位移累加类以获取左侧炮筒的世界坐标，将累加类

// 与左侧炮筒的位移节点相关联

```
transformAccumulator* leftGunnerWC = new transformAccumulator();  
leftGunnerWC->attachToGroup(leftGunnerPAT);
```

// 对于右侧炮筒视图，重复上面的过程，注意要偏移-45 度

```
osg::PositionAttitudeTransform* rightGunnerPAT =  
    new osg::PositionAttitudeTransform();  
rightGunnerPAT->setAttitude(  
    osg::Quat( osg::DegreesToRadians(-45.0) , osg::Vec3(0,0,1) ));  
transformAccumulator* rightGunnerWC = new transformAccumulator();  
gunnerXForm->addChild(rightGunnerPAT);  
rightGunnerWC->attachToGroup(rightGunnerPAT);
```

// 定义驾驶员视图的位移变换，添加到坦克节点

```
osg::PositionAttitudeTransform* driverOffsetPAT =  
    new osg::PositionAttitudeTransform();  
driverOffsetPAT->setPosition(osg::Vec3(0,-15,4));  
driverOffsetPAT->setAttitude(  
    osg::Quat( osg::DegreesToRadians(-5.0f), osg::Vec3(1,0,0) ) );  
ownTank->addChild(driverOffsetPAT);
```

//使用位移累加类获取驾驶员视图位移变换的世界坐标

```
transformAccumulator* driverWorldCoords = new transformAccumulator();  
driverWorldCoords->attachToGroup(driverOffsetPAT);
```

// 定义新的矩阵漫游器，它将跟随驾驶员的位移变换位置

```
viewer.getKeySwitchMatrixManipulator()->addMatrixManipulator
('m',"ft",new followNodeMatrixManipulator(driverWorldCoords));
```

现在我们的仿真程序已经基本完成，可以进入仿真循环了。与以往程序不同的是，在更新遍历结束之后，拣选遍历开始之前，我们需要手动设置三个与炮筒关联的摄像机的位置。使用 `Producer::Camera` 的 `setViewByMatrix` 方法，并直接从 `transformAccumulator` 实例中取得坐标数值即可。注意，`transformAccumulator` 返回的矩阵以 Y 轴为垂直轴，但 `Camera` 类的 `setViewByMatrix` 方法设定 Z 轴为垂直轴，因此我们需要首先将矩阵进行旋转再行使用。代码如下：

```
//创建窗口和线程
viewer.realize();

while( !viewer.done() )
{
    // ?等待拣选和更新遍历完成
    viewer.sync();

    // update the scene by traversing it with the the update visitor which will
    //更新场景，执行更新回调
    viewer.update();

    tankCameras->findCamera("GunnerLeft")->setViewByMatrix(
        Producer::Matrix(leftGunnerWC->getMatrix().ptr() ) *
        Producer::Matrix::rotate( -M_PI/2.0, 1, 0, 0 ) );

    tankCameras->findCamera("GunnerCenter")->setViewByMatrix(
        Producer::Matrix(gunnerWorldCoords->getMatrix().ptr() ) *
        Producer::Matrix::rotate( -M_PI/2.0, 1, 0, 0 ) );

    tankCameras->findCamera("GunnerRight")->setViewByMatrix(
        Producer::Matrix(rightGunnerWC->getMatrix().ptr() ) *
        Producer::Matrix::rotate( -M_PI/2.0, 1, 0, 0 ) );


    // 开始这一帧的拣选和更新遍历
    viewer.frame();
}

// 等待拣选和更新遍历完成，然后退出
viewer.sync();

return 0;
}
```

注意：按下 m 键可以在各个摄像机之间切换。

第十二课第一节 使用 OpenGL 顶点着色器和片元着色器

目的： 

使用 OpenGL 着色语言中的顶点和片元着色器，替代原有的固化函数管道的光照和材质计算，用于场景图形中的节点选择。（砖块着色器程序来自 3DLabs）

概述: [1](#)

OpenGL 着色语言允许程序员编写自己的像素和顶点着色器。有关着色语言的更多信息，包括最低的硬件和软件需求，请参阅：

<http://developer.3dlabs.com/openGL2/>

OSG 的 `osg::Program` 和 `osg::Shader` 类允许用户将着色器作为 `StateSet` 的一部分加入选定的场景图形子树。如果要在 `OpenSceneGraph` 中使用自定义的顶点和片元着色器，则需要使用下面的基类：

`osg::Program` - 在应用层级上封装了 OpenGL 着色语言的 `glProgramObject` 函数。`Program` 类的对象继承自 `osg::StateAttribute` 类。即 `osg::Program` 类的实例可以关联到 `StateSet`，并使用 `setAttributeAndModes()` 方法来许可自身的使用。开启 `Program` 渲染状态之后，与此渲染状态相关联的几何体均会使用该 `Program` 的着色器进行渲染。

`osg::Shader` - 在应用层级上封装了 OpenGL 着色语言的 `glShaderObject` 函数。这个类用于管理着色器源代码的加载和编译。`osg::Shader` 类的实例 可以与一个或多个 `osg::Program` 的实例相关联。`Shader` 对象有两种类型：

`osg::Shader::FRAGMENT` 和 `osg::Shader::VERTEX`。

代码: [1](#)

要创建一个使用 OpenGL 像素和片元着色器的程序，可以按照下面的步骤：

- 创建一个 `osg::Program` 实例；
- 创建一个 `osg::Shader` 类的顶点（`VERTEX`）或片元（`FRAGMENT`）着色器实例；
- 加载并编译着色器代码；
- 将着色器添加到 `osg::Program` 实例；
- 将 `osg::Program` 实例关联给 `StateSet` 类，并将其激活；

下面的代码用于加载和应用基本的顶点和片元着色器：

```
osg::StateSet* brickState = tankNode->getOrCreateStateSet();

osg::Program* brickProgramObject = new osg::Program;
osg::Shader* brickVertexObject =
    new osg::Shader( osg::Shader::VERTEX );
osg::Shader* brickFragmentObject =
    new osg::Shader( osg::Shader::FRAGMENT );
brickProgramObject->addShader( brickFragmentObject );
brickProgramObject->addShader( brickVertexObject );
loadShaderSource( brickVertexObject, "shaders/brick.vert" );
loadShaderSource( brickFragmentObject, "shaders/brick.frag" );

brickState->setAttributeAndModes(brickProgramObject, osg::StateAttribute::ON);
```

下面的函数可以方便地加载着色器代码，并将编译后的代码关联给着色器对象：

```
bool loadShaderSource(osg::Shader* obj, const std::string& fileName )
{
    std::string fqFileName = osgDB::findDataFile(fileName);
    if( fqFileName.length() == 0 )
    {
        std::cout << "File \"" << fileName << "\" not found." << std::endl;
        return false;
    }
}
```

```

bool success = obj->loadShaderSourceFromFile( fqFileName.c_str());
if ( !success )
{
    std::cout << "Couldn't load file: " << fileName << std::endl;
    return false;
}
else
{
    return true;
}
}

```

第十二课第二节 向着色器传递变量数据

目的: [🔗](#)

使用 OpenGL 着色语言，实现从调用程序向着色器传递所需的参数。

概述: [🔗](#)

着色器有两种类型的参数：一致变量（Uniform variables）指得是在一帧当中保持恒定的数值，典型的参数包括视口的方向和灯光的方向。而易变变量（Varying variables）对于每一个执行单元（包括顶点着色器中的顶点，以及片元着色器中的片元）都是变化的。

一致变量用于从渲染程序向着色器传递参数。在《OpenGL 着色语言》一书中，凹凸贴图着色器的例子使用了三个一致变量参数：三维向量 `LightPosition`，以及两个 2D 材质变量 `baseTexture` 和 `normalMap`。OSG 类提供了一种直截了当的机制，使得 `OpenSceneGraph` 程序可以向着色器直接传递数据。

代码: [🔗](#)

以下所述为 `OpenSceneGraph` 程序向凹凸贴图着色器传递参数的方法。其代码使用了 `Program` 对象类的方法 `addUniform`，此方法有两个输入参数：一个用于声明着色器一致变量的字符串，以及一个用于记录参数值的变量。声明 2D 材质一致变量的方法与此类似，其输入参数为 2D 材质变量 名称的字符串，以及所用材质单元的整数。用于向凹凸贴图着色器传递所需变量的代码如下：

```

bool loadShaderSource(osg::Shader* obj, const std::string& fileName );

int main( int argc, char **argv )
{
    osg::Group* rootNode = new osg::Group();
    osg::Node* tankNode = NULL;
    osg::Node* terrainNode = NULL;
    osg::PositionAttitudeTransform* tankXform =
        new osg::PositionAttitudeTransform();

    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\\");
}

```

```

osgDB::setDataFilePathList(pathList);

tankNode = osgDB::readNodeFile("T72-tank_des.flt");
terrainNode = osgDB::readNodeFile("JoeDirt.flt");
if (! (tankNode && terrainNode))
{
    std::cout << "Couldn't load models, quitting." << std::endl;
    return -1;
}
rootNode->addChild(terrainNode);
rootNode->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) ) );

// ?将一个“空”Program 对象关联到 rootNode，它相当于定义了一个缺省的
// ?StateAttribute 属性。“空”Program 对象（不与任何 Shader 对象关联）是一种
// 特殊情形，它表示程序将使用 OpenGL 1.x 固化功能的渲染管道。
osg::StateSet* rootState = rootNode->getOrCreateStateSet();
osg::Program* defaultProgramObject = new osg::Program;
rootState->setAttributeAndModes(defaultProgramObject, osg::StateAttribute::ON);

osg::StateSet* bumpMapState = tankNode->getOrCreateStateSet();

osg::Program* bumpMapProgramObject = new osg::Program;

osg::Shader* brickVertexObject =
    new osg::Shader( osg::Shader::VERTEX );
osg::Shader* brickFragmentObject =
    new osg::Shader( osg::Shader::FRAGMENT );

bool ok =
    loadShaderSource( brickVertexObject, "shaders/bumpmap.vert" )
    &&
    loadShaderSource( brickFragmentObject, "shaders/bumpmap.frag" );

if(!ok)
{
    std::cout << "Couldn't load shaders" << std::endl;
    return -1;
}

bumpMapProgramObject->addShader( brickFragmentObject );
bumpMapProgramObject->addShader( brickVertexObject );

```

```

//bumpMapProgramObject->setUniform("LightPosition", osg::Vec3(0,0,1));
//bumpMapProgramObject->setSampler("normalMap",1);
//bumpMapProgramObject->setSampler("baseTexture",0);

osg::Uniform* lightPosU = new osg::Uniform("LightPosition",osg::Vec3(0,0,1));
osg::Uniform* normalMapU = new osg::Uniform("normalMap",1);
osg::Uniform* baseTextureU = new osg::Uniform("baseTexture",0);

bumpMapState->addUniform(lightPosU);
bumpMapState->addUniform(normalMapU);
bumpMapState->addUniform(baseTextureU);

bumpMapState->setAttributeAndModes(bumpMapProgramObject, osg::StateAttribute::ON);

osg::Texture2D* tankBodyNormalMap = new osg::Texture2D;
// 避免在优化过程中静态量出错。
tankBodyNormalMap->setDataVariance(osg::Object::DYNAMIC);
osg::Image* tankBody = osgDB::readImageFile("TankBump.png");
if (!tankBody)
{
    std::cout << " couldn't find texture, quitting." << std::endl;
    return -1;
}

tankBodyNormalMap->setImage(tankBody);
bumpMapState->setTextureAttributeAndModes(1,tankBodyNormalMap,osg::StateAttribute::ON);

osgProducer::Viewer viewer;
viewer.setUpViewer( osgProducer::Viewer::STANDARD_SETTINGS );

viewer.setSceneData( rootNode );
viewer.realize();

while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}
viewer.sync();
return 0;
}

```

第十二课第三节 更新着色器

目的: [91](#)

持续更新 OpenGL 着色器程序所用的变量。

概述: [91](#)

本章演示了如何使用 OpenGL 着色器程序持续更新参数, 从而实现对场景中对象的动态着色效果控制。我们将使用前述的凹凸贴图着色器, 并更新其一致变量 `LightPosition`, 以模拟灯光环绕坦克模型移动的效果。我们将使用上一章所用的 `addUniform` 方法。不过, 这一次我们将使用更新回调来实现变量更新的效果。回调类的主体用于持续改变灯光位置的向量。为了使用这个类的功能, 我们简单地将其关联给场景中节点的更新回调。

代码: [91](#)

着色器更新回调的定义如下:

```
class updateBumpShader : public osg::Uniform::Callback
{
public:
    virtual void operator()
        ( osg::Uniform* uniform, osg::NodeVisitor* nv )
    {
        float angle = nv->getFrameStamp()->getReferenceTime();
        float x = sinf(angle)*.2f;
        float y = cosf(angle)*.2f;
        osg::Vec3 lightPos(x,y,.6f);
        uniform->set(lightPos);
    }
};
```

其余的代码与上一章基本相同: 设置场景和着色器的实例。唯一不同的是, 我们需要将上面的回调类设置为场景中一个节点的更新回调。

```
int main( int argc, char **argv )
{
    osg::Group* rootNode = new osg::Group();
    osg::Node* tankNode = NULL;
    osg::Node* terrainNode = NULL;
    osg::PositionAttitudeTransform* tankXform =
        new osg::PositionAttitudeTransform();

    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\\");
    osgDB::setDataFilePathList(pathList);

    tankNode = osgDB::readNodeFile("T72-tank_des.flt");
    terrainNode = osgDB::readNodeFile("JoeDirt.flt");
    if (! (tankNode && terrainNode))
    {
        std::cout << "Couldn't load models, quitting." << std::endl;
        return -1;
    }
}
```

```

}
rootNode->addChild(terrainNode);
rootNode->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) ) );

osg::StateSet* bumpMapState = tankNode->getOrCreateStateSet();

osg::Program* bumpMapProgramObject = new osg::Program;

osg::Shader* bumpVertexObject =
    new osg::Shader( osg::Shader::VERTEX );
osg::Shader* bumpFragmentObject =
    new osg::Shader( osg::Shader::FRAGMENT );

bool ok =
    loadShaderSource( bumpVertexObject, "shaders/bumpmap.vert" )
    &&
    loadShaderSource
        ( bumpFragmentObject, "shaders/bumpmap.frag" ) ;

if(!ok)
{
    std::cout << "Couldn't load shaders" << std::endl;
    return -1;
}

bumpMapProgramObject->addShader( bumpFragmentObject );
bumpMapProgramObject->addShader( bumpVertexObject );

osg::Uniform* lightPosU =
    new osg::Uniform("LightPosition",osg::Vec3(0,0,1));
osg::Uniform* normalMapU = new osg::Uniform("normalMap",1);
osg::Uniform* baseTextureU = new osg::Uniform("baseTexture",0);

bumpMapState->addUniform(lightPosU);
bumpMapState->addUniform(normalMapU);
bumpMapState->addUniform(baseTextureU);
lightPosU->setUpdateCallback(new updateBumpShader() );

bumpMapState->setAttributeAndModes
    (bumpMapProgramObject, osg::StateAttribute::ON);

```

```

osg::Texture2D* tankBodyNormalMap = new osg::Texture2D;
tankBodyNormalMap->setDataVariance(osg::Object::DYNAMIC);
osg::Image* tankBody = osgDB::readImageFile("TankBump.png");
if (!tankBody)
{
    std::cout << " couldn't find texture, quitting." << std::endl;
    return -1;
}

tankBodyNormalMap->setImage(tankBody);
bumpMapState->setTextureAttributeAndModes
    (1,tankBodyNormalMap,osg::StateAttribute::ON);

osgProducer::Viewer viewer;
viewer.setUpViewer( osgProducer::Viewer::STANDARD_SETTINGS );
viewer.setSceneData( rootNode );
viewer.realize();

while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}
viewer.sync();
return 0;
}

```

第十二课第四节 如何快速实现渐灰效果

A Quick Shader to Simulate Graying-Out (GLOC) [1](#)

已完成代码 [1](#)

下面的片元着色器代码将根据像素到屏幕中心的距离执行片元渐变到灰色的工作。

```
uniform vec4 grayScaleWeights;
```

```
void main( void )
```

```
{
```

```
    // 获取 RGB 纹理颜色
```

```
    vec4 texelColor = texture2D( testTexture, gl_TexCoord[0].xy );
```

```
    //
```

```
    // 转换为灰度
```

```
    //
```

```

//我们可以使用计算红、绿、蓝三色权重的方法来实现图片的灰度处理。
// 标准的灰度转换权重为（0.30， 0.59， 0.11）。因此，光照的灰色计算
//公式，即光照过滤计算公式为：
//
// luminance = 0.30*R + 0.59*G + 0.11*B
//
// 如果我们按照向量形式处理 RGB 颜色，那么实际上我们执行了点积的计算。
//

//   grayScaleWeights = vec4(1,1,1,1);

vec4 scaledColor = texelColor * grayScaleWeights;
float luminance = scaledColor.r + scaledColor.g + scaledColor.b;

//
float screenCenterX = 640;
float screenCenterY = 512;

float deltaX = screenCenterX - gl_FragCoord.x;
float deltaY = screenCenterY - gl_FragCoord.y;

float dist = sqrt(deltaX * deltaX + deltaY * deltaY);
float inRange = 50.;
float outRange = 600.;

if (dist < inRange)
{

    gl_FragColor = texelColor;
}
else
{
    if ( dist > outRange)
    {
        gl_FragColor = vec4(0.25,0.25,0.25,1);
    }

    else
    {
        float scale = (dist < outRange) ? (outRange - dist) / (outRange - inRange) : 1 ;

        float smooth = smoothstep(0,1,scale);
        luminance = clamp(luminance, 0, .25);

        gl_FragColor = mix(texelColor,luminance,(1.-smooth));
    }
}
}

```

```
}
```

剩下的工作：

- 传递一致变量（uniform）参数，用于改变可见锥区（cone of visibility）。
- 传递屏幕中心的参数。

第十三课 向场景中添加告示牌（Billboard）节点

目的： [🔗](#)

向场景中添加告示牌节点，以模拟沙漠中灌木的效果。

概述： [🔗](#)

告示牌（Billboard）节点的特性是围绕用户定义的轴（或者点）进行旋转，从而始终面向一个指定的方向。典型的告示牌可以绕 Z 轴正向旋转，同时面向视口方向，以实现树林的模拟。OSG 发行版本的示例程序 `osgBillboard` 中通过指定旋转轴和法线（朝向方向）的方式，演示了各种模式的告示牌节点。本教程将创建多个树木样式的告示牌，它们环绕 Z 轴正向旋转并始终朝向视口。

Billboard 类（`osg::Billboard`）包括可以绕一个轴或者参考点旋转并朝向指定方向的几何体。Billboard 类继承自 Geode 类。也就是说，继承自 Drawable 类（包括 Geometry 类）的实例可以被添加给 Billboard 对象。我们可以将几何体关联给告示牌节点。因为 Billboard 类是继承自 Geode 类的，我们也可以将 StateSet 关联给告示牌节点。

除了继承自 Geode 类的方法和成员外，告示牌类还包括了用于控制其自身特性的成员和方法：告示牌是否环绕参考点或者轴；如果绕某个轴旋转的话，它应该朝向什么方向。为了控制告示牌的类型，可以使用其中的 `setMode(osg::Billboard::Mode)` 方法。其合法参数有：POINT_ROT_EYE（绕一个点旋转，相对于眼睛位置），POINT_ROT_WORLD（绕一个点旋转，相对于世界坐标），AXIAL_ROT（绕一个轴旋转）。如果指定了 AXIAL_ROT 模式，用户就可以使用 `setAxis(osg::Vec3)` 方法来设置告示牌绕之旋转的轴。用户还可以使用 `setNormal(osg::Vec3)` 方法定义告示牌朝向的法线。为了放置告示牌，用户还需重载 `osg::Geode` 的 `addGeometry` 方法。这个方法有两个参数：一个 Drawable 实例，以及一个标识位置的 `osg::Vec3` 实例。

创建告示牌还需要最后一步工作：当几何体根据法线的设置旋转并朝向视口之后，对其进行光照的计算将导致一种怪异的结果。（光照会随着视口的改变而突变）为了创建各个方向的形象均相同的告示牌，我们需要确信关闭告示牌的光照。综上所述，向场景中添加多个告示牌的基本步骤为：

- 创建一个告示牌实例并添加到场景；
- 创建光照关闭的渲染状态；
- 创建几何体（应用上面的渲染状态）并添加给告示牌节点。

代码： [🔗](#)

需要以下的头文件：

```
#include <osg/Geometry>
#include <osg/Texture2D>
#include <osg/Billboard>
#include <osg/BlendFunc>
#include <osgDB/Registry>
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/PositionAttitudeTransform>
```

首先，我们编写一个简便的函数用于生成与告示牌节点相关联的几何体。此函数有两个参数：一个浮点数用于表示放缩特性，以及一个关联给几何体实例的渲染状态指针。这个函数的返回值是一个几何体指针。我们使用

简单的四边形来生成几何体。关于如何指定四边形的顶点和纹理坐标，请参照前面的教程。唯一需要注意的是，在创建该几何体的时候，我们需要它相对旋转轴置中。函数的定义如下所示：

```
osg::Drawable* createShrub(const float & scale, osg::StateSet* bbState)
{
    float width = 1.5f;
    float height = 3.0f;

    width *= scale;
    height *= scale;

    osg::Geometry* shrubQuad = new osg::Geometry;

    osg::Vec3Array* shrubVerts = new osg::Vec3Array(4);
    (*shrubVerts)[0] = osg::Vec3(-width/2.0f, 0, 0);
    (*shrubVerts)[1] = osg::Vec3( width/2.0f, 0, 0);
    (*shrubVerts)[2] = osg::Vec3( width/2.0f, 0, height);
    (*shrubVerts)[3] = osg::Vec3(-width/2.0f, 0, height);

    shrubQuad->setVertexArray(shrubVerts);

    osg::Vec2Array* shrubTexCoords = new osg::Vec2Array(4);
    (*shrubTexCoords)[0].set(0.0f,0.0f);
    (*shrubTexCoords)[1].set(1.0f,0.0f);
    (*shrubTexCoords)[2].set(1.0f,1.0f);
    (*shrubTexCoords)[3].set(0.0f,1.0f);
    shrubQuad->setTexCoordArray(0,shrubTexCoords);

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

    // 创建一个颜色数组，并为所有的顶点添加单一的颜色。

    osg::Vec4Array* colorArray = new osg::Vec4Array;
    colorArray->push_back(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f) ); // white, fully opaque

    // ?创建颜色索引数组。
    osg::TemplateIndexArray
        <unsigned int, osg::Array::UIntArrayType,4,1> *colorIndexArray;
    colorIndexArray =
        new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,1>;
    colorIndexArray->push_back(0);

    // 使用索引数组将其中的第一个值关联给所有的顶点。
    shrubQuad->setColorArray( colorArray);
    shrubQuad->setColorIndices(colorIndexArray);
    shrubQuad->setColorBinding(osg::Geometry::BIND_OVERALL);

    shrubQuad->setStateSet(bbState);
```

```

    return shrubQuad;
}

```

下一步，我们编写场景设置和仿真循环的执行代码。代码中包括我们刚刚编写的向告示牌添加四边形实例的函数（Billboard 类继承自 Geode 类，因此可以向其添加 Drawable 实例），以及旋转轴和告示牌朝向方向的设置。

```

int main( int argc, char **argv )
{
    osgProducer::Viewer viewer;

    osg::Group* rootNode = new osg::Group();
    osg::Node* tankNode = NULL;
    osg::Node* terrainNode = NULL;
    osg::PositionAttitudeTransform* tankXform =
        new osg::PositionAttitudeTransform();

    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);

    tankNode = osgDB::readNodeFile("T72-tank_des.flt");
    terrainNode = osgDB::readNodeFile("JoeDirt.flt");
    if (! (tankNode && terrainNode))
    {
        std::cout << "Couldn't load models, quitting." << std::endl;
        return -1;
    }
    rootNode->addChild(terrainNode);
    rootNode->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) ) );
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);

    viewer.setSceneData(rootNode);
    viewer.realize();
}

```

创建告示牌并设置其参数。我们希望告示牌可以环绕轴（0，0，1）旋转，并始终朝向视口。为了保证它始终面向视口，我们需要定义告示牌的法线方向为（0，-1，0）。

```

osg::Billboard* shrubBillBoard = new osg::Billboard();
rootNode->addChild(shrubBillBoard);

```

```

shrubBillBoard->setMode(osg::Billboard::AXIAL_ROT);
shrubBillBoard->setAxis(osg::Vec3(0.0f,0.0f,1.0f));
shrubBillBoard->setNormal(osg::Vec3(0.0f,-1.0f,0.0f));

```

下面的代码将设置四边形几何体的渲染状态。我们将使用 Alpha 融合算法，使四边形看起来像是一个复杂的树木的几何形状。

```

osg::Texture2D *ocotilloTexture = new osg::Texture2D;
ocotilloTexture->setImage(osgDB::readImageFile("images\\ocotillo.png"));

osg::AlphaFunc* alphaFunc = new osg::AlphaFunc;
alphaFunc->setFunction(osg::AlphaFunc::GEQUAL,0.05f);

osg::StateSet* billboardStateSet = new osg::StateSet;

billboardStateSet->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
billboardStateSet->setTextureAttributeAndModes
    (0, ocotilloTexture, osg::StateAttribute::ON );
billboardStateSet->setAttributeAndModes
    (new osg::BlendFunc, osg::StateAttribute::ON );
osg::AlphaFunc* alphaFunction = new osg::AlphaFunc;
alphaFunction->setFunction(osg::AlphaFunc::GEQUAL,0.05f);
billboardStateSet->setAttributeAndModes( alphaFunc, osg::StateAttribute::ON );

```

下面，我们需要创建一些 Drawable 几何体并将其添加给告示牌节点实例。我们将使用上面定义的函数来创建几何体；并使用 Billboard 类重载的 addDrawable 方法来添加并放置这些四边形。

```

// ?创建不同放缩系数的几何体，并赋予它们已定义好的渲染状态。
osg::Drawable* shrub1Drawable = createShrub( 1.0f, billboardStateSet);
osg::Drawable* shrub2Drawable = createShrub( 2.0f, billboardStateSet);
osg::Drawable* shrub3Drawable = createShrub( 1.2f, billboardStateSet);

```

```

// 将这些几何体添加到告示牌节点，并设置其不同位置。
shrubBillBoard->addDrawable( shrub1Drawable , osg::Vec3(12,3,8) );
shrubBillBoard->addDrawable( shrub2Drawable , osg::Vec3(10,18,8));
shrubBillBoard->addDrawable( shrub3Drawable , osg::Vec3(6,10,8) )

```

这之后的代码就很寻常了。

```

while( !viewer.done() )
{
    // wait for all cull and draw threads to complete.
    viewer.sync();
    // update the scene by traversing it with the update visitor which will
    // call all node update callbacks and animations.
    viewer.update();
    // fire off the cull and draw traversals of the scene.
    viewer.frame();
}

// wait for all cull and draw threads to complete before exit.
viewer.sync();
return 0;

```



```
}
```

第十四课 交集测试

注意:

OSG 2.0 以上版本中, 已经使用 `IntersectionVisitor` 代替了原有的 `IntersectVisitor` 类。新的类功能更加强大。

目的: [🔗](#)

向场景中添加一系列随机放置着的灌木。使用 `osgUtil` 库的交集函数来获取正确的地形高度值, 以便正确安置灌木告示牌节点。

概述: [🔗](#)

如果我们希望扩展前一个教程的内容, 向场景中的地形添加一些随意放置的告示牌节点, 那么我们最好能够获取任意经纬 (x, y) 位置的地形高度值。我们可以使用 `osgUtil` 库提供的交集运算模块函数来实现这一功能。我们可能会用到的基本类描述如下:

线段 (`osg::LineSegment`) - 交集测试的基础是场景中的射线。线段类提供了一种定义射线的方法。它包括两个 `osg::Vec3` 实例: 一个用于定义线段的起点, 另一个用于定义终点。当交集测试被触发时, 它将检测射线的相交情况并执行相应的操作。

交点 (`osgUtil::Hit`) - 这个类向程序员提供了获取交集检测的基本数据的方法。交点类包括一条射线与场景中几何体相交的状态信息。尤为重要, 它以 `Vec3` 的形式提供了局部和世界坐标的位置以及法线数据。它的成员方法 `getLocalIntersectPoint`, `getLocalIntersectNormal`, `getWorldIntersectPoint` 和 `getworldIntersectNormal` 分别以 `osg::Vec3` 作为返回值, 返回局部/世界坐标的相交点/法线数值。

交点列表 (`osgUtil::IntersectVisitor::HitList`) - 一条单一的线段可能与场景中的多个几何体实例 (或者多次与同一个几何体) 产生交集。对于每一条参与交集测试的线段, 系统均会产生一个列表。这个列表包含了 所有交集测试产生的 `Hit` 实例。如果没有监测到任何交集, 该列表保持为空。

交集访问器 (`osgUtil::IntersectVisitor`) - 射线与场景中几何体的交集测试由交集访问器来创建并实现初始化。`IntersectionVisitor` 类继承自 `NodeVisitor` 类, 因此其创建和 触发机制与 `NodeVisitor` 实例大致相似。访问器需要维护一个进行交集测试的线段列表。而对于其中的每一条线段, 访问器都会创建一个交点列表 (`osgUtil::IntersectVisitor::HitList` 实例)。

代码: [🔗](#)

为了正确获取放置灌木模型所需的地形高度值, 我们需要遵循下面的基本步骤: ps:

- 创建一个 `LineSegment` 实例, 它使用两个 `Vec3` 实例来定义交集测试所用射线的起点和终点。
- 创建一个 `IntersectVisitor` 实例。
- 将 `LineSegment` 实例添加到 `IntersectVisitor` 实例。
- 初始化 `IntersectVisitor` 实例, 使其从场景图形中适当的节点开始遍历。
- 获取交集测试结果的世界坐标。

下面的代码演示了上述步骤的两种实现方式。第一部分是单个 `LineSegment` 添加到 `IntersectVisitor`, 以判断坦克模型的放置 高度是否正确。第二部分是将与告示牌节点相关的多个线段添加到 `IntersectVisitor`, 每一个线段实例均对应一个灌木告示牌对象。针对每个 `LineSegment` 执行与其关联的交集测试之后, 我们既可正确地放置告示牌的位置了。

```
#include <osg/Geometry>
```

```

#include <osg/Texture2D>
#include <osg/Billboard>
#include <osg/BlendFunc>
#include <osgDB/Registry>
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/PositionAttitudeTransform>
#include <ctime>
#include <cstdlib>

#define NUMBER_O_SHRUBS 1250

osg::Drawable* createShrub(const float & scale, osg::StateSet* bbState)
{
    float width = 1.5f;
    float height = 3.0f;

    width *= scale;
    height *= scale;

    osg::Geometry* shrubQuad = new osg::Geometry;

    osg::Vec3Array* shrubVerts = new osg::Vec3Array(4);
    (*shrubVerts)[0] = osg::Vec3(-width/2.0f, 0, 0);
    (*shrubVerts)[1] = osg::Vec3( width/2.0f, 0, 0);
    (*shrubVerts)[2] = osg::Vec3( width/2.0f, 0, height);
    (*shrubVerts)[3] = osg::Vec3(-width/2.0f, 0, height);

    shrubQuad->setVertexArray(shrubVerts);

    osg::Vec2Array* shrubTexCoords = new osg::Vec2Array(4);
    (*shrubTexCoords)[0].set(0.0f,0.0f);
    (*shrubTexCoords)[1].set(1.0f,0.0f);
    (*shrubTexCoords)[2].set(1.0f,1.0f);
    (*shrubTexCoords)[3].set(0.0f,1.0f);
    shrubQuad->setTexCoordArray(0,shrubTexCoords);

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

    shrubQuad->setStateSet(bbState);

    return shrubQuad;
}

int main( int argc, char **argv )
{
    osgProducer::Viewer viewer;

```

```

osg::Group* rootNode = new osg::Group();
osg::Node* tankNode = NULL;
osg::Node* terrainNode = NULL;
osg::PositionAttitudeTransform* tankXform =
    new osg::PositionAttitudeTransform();

tankNode = osgDB::readNodeFile("\\Models\\T72-Tank\\T72-tank_des.flt");
terrainNode = osgDB::readNodeFile("\\Models\\JoeDirt\\JoeDirt.flt");
if (! (tankNode && terrainNode))
{
    std::cout << "Couldn't load models, quitting." << std::endl;
    return -1;
}
rootNode->addChild(terrainNode);
rootNode->addChild(tankXform);
tankXform->addChild(tankNode);

```

下面将获取地形模型在给定 X, Y 位置的高度值, 并放置坦克模型。

```

double tankXPosition = -10.0;
double tankYPosition = -10.0;

osg::LineSegment* tankLocationSegment = new osg::LineSegment();
tankLocationSegment->set(
    osg::Vec3(tankXPosition, tankYPosition, 999) ,
    osg::Vec3(tankXPosition, tankYPosition, -999) );

osgUtil::IntersectVisitor findTankElevationVisitor;
findTankElevationVisitor.addLineSegment(tankLocationSegment);
terrainNode->accept(findTankElevationVisitor);

osgUtil::IntersectVisitor::HitList tankElevationLocatorHits;
tankElevationLocatorHits =
    findTankElevationVisitor.getHitList(tankLocationSegment);
osgUtil::Hit heightTestResults;
if ( tankElevationLocatorHits.empty() )
{
    std::cout << " couldn't place tank on terrain" << std::endl;
    return -1;
}
heightTestResults = tankElevationLocatorHits.front();
osg::Vec3d terrainHeight = heightTestResults.getWorldIntersectPoint();

tankXform->setPosition( terrainHeight );
tankXform->setAttitude(
    osg::Quat(osg::DegreesToRadians(-45.0), osg::Vec3(0,0,1) ) );

```

下面的代码将设置视口, 告示牌节点, 以及渲染状态。

```

viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);

```

```

viewer.setSceneData(rootNode);
viewer.realize();

osg::Billboard* shrubBillBoard = new osg::Billboard();
rootNode->addChild(shrubBillBoard);

shrubBillBoard->setMode(osg::Billboard::AXIAL_ROT);
shrubBillBoard->setAxis(osg::Vec3(0.0f,0.0f,1.0f));
shrubBillBoard->setNormal(osg::Vec3(0.0f,-1.0f,0.0f));

osg::Texture2D *ocotilloTexture = new osg::Texture2D;
ocotilloTexture->setImage(osgDB::readImageFile("images\\ocotillo.png"));

osg::StateSet* billBoardStateSet = new osg::StateSet;

billBoardStateSet->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
billBoardStateSet->setTextureAttributeAndModes(0, ocotilloTexture, osg::StateAttribute::ON );
billBoardStateSet->setAttributeAndModes( new osg::BlendFunc, osg::StateAttribute::ON );

osg::AlphaFunc* alphaFunction = new osg::AlphaFunc;
alphaFunction->setFunction(osg::AlphaFunc::GEQUAL,0.05f);
billBoardStateSet->setAttributeAndModes( alphaFunction, osg::StateAttribute::ON );

```

按照上面所述的检测射线和地形模型碰撞交点的基本步骤，现在我们将向 `IntersectVisitor` 添加多个 `LineSegment` 实例。每条射线的交集测试结果（交点位置）将用于放置相应的告示牌。

```

srand(time(0)); // Initialize random number generator.

```

```

osgUtil::IntersectVisitor isectVisitor;
osg::LineSegment* terrainIsect[NUMBER_O_SHRUBS];

```

```

int randomX, randomY;

```

```

for (int i=0; i< NUMBER_O_SHRUBS; i++)
{
    randomX = (rand() % 100) + 1;
    randomY = (rand() % 100) + 1;
    terrainIsect[i] = new osg::LineSegment(
        osg::Vec3(randomX, randomY, 999) ,
        osg::Vec3(randomX, randomY, -999) );
    isectVisitor.addLineSegment(terrainIsect[i]);
}
terrainNode->accept(isectVisitor);

```

```

osg::Drawable* shrubDrawable[NUMBER_O_SHRUBS];

```

```

for (int j = 0 ; j < NUMBER_O_SHRUBS; j++)
{

```

```

float randomScale = ((rand() % 15) + 1) / 10.0;
shrubDrawable[j] = createShrub( randomScale, billboardStateSet);
osgUtil::IntersectVisitor::HitList hitList = isectVisitor.getHitList(terrainIsect[j]);
if (! hitList.empty() )
{
    osgUtil::Hit firstHit = hitList.front();
    osg::Vec3d shrubPosition = firstHit.getWorldIntersectPoint();

    // osg::Vec3d shrubPosition =
    // isectVisitor.getHitList(terrainIsect[j]).front().getWorldIntersectPoint();

    shrubBillBoard->addDrawable( shrubDrawable[j] , shrubPosition );
}
}

while( !viewer.done() )
{
    // wait for all cull and draw threads to complete.
    viewer.sync();

    // update the scene by traversing it with the update visitor which will
    // call all node update callbacks and animations.
    viewer.update();

    // fire off the cull and draw traversals of the scene.
    viewer.frame();
}
// wait for all cull and draw threads to complete before exit.
viewer.sync();
return 0;
}

```

第十五课第一节 向场景中添加 **osgParticle** 粒子效果

目的： [🔗](#)

向场景中添加自定义的 **osgParticle** 实例，模拟坦克模型在地形上运动时产生的烟尘。

概述： [🔗](#)

添加粒子效果可以有效提高仿真程序的外观和真实性。粒子引擎一般用于模拟烟雾，火焰，尘埃以及其他一些类似的效果。如果要向 OSG 场景中添加粒子效果，通常可以使用下面的一些类：

粒子系统 (**osgParticle::ParticleSystem**) - 维护并管理一系列粒子的生成，更新，渲染和销毁。粒子系统类继承自 **Drawable** 类。粒子的渲染控制因此与其它 **Drawable** 对象的渲染类似：控制其 渲染属性 **StateAttribute** 即可。OSG 提供了一个方便的函数以允许用户控制三个常用的渲染状态属性。方法 **setDefaultAttributes** 可以用于指定材质（或者指定为 **NULL** 以禁用材质），允许/禁止附加的图像融合，允许/禁止光照。

粒子 (`osgParticle::Particle`) - 粒子系统的基本单元。粒子类同时具有物理属性和图像属性。它的形状可以是任意的点 (`POINT`)，四边形 (`QUAD`)，四边形带 (`QUAD_TRIPSTRIP`)，六角形 (`HEXAGON`) 或者线 (`LINE`)。每个粒子都有自己的生命周期。生命周期也就是每个粒子可以存活的秒数。(生命周期为负数的粒子可以存活无限长时间)所有的粒子都具有大小 (`SIZE`)，`ALPHA` 值和颜色 (`COLOR`) 属性。每一组粒子都可以指定其最大和最小值。为了便于粒子生命周期的管理，粒子系统通过改变生命周期的最大和最小值来控制单个粒子的渲染。(根据已经消耗的时间，在最小和最大值之间进行线性插值)

程序员也可以自行指定最小到最大值的插值方法。(参见 `osgParticle::Interpolator` 的代码)

放置器 (`osgParticle::Placer`) - 设置粒子的初始位置。用户可以使用预定义的放置器或者定义自己的放置器。已定义的放置器包括：点放置器 `PointPlacer` (所有的粒子从同一点出生)，扇面放置器 `SectorPlacer` (所有的粒子从一个指定中心点，半径范围和角度范围的扇面出生)，以及多段放置器 `MultiSegmentPlacer` (用户指定一系列的点，粒子沿着这些点定义的线段出生)。

发射器 (`osgParticle::Shooter`) - 指定粒子的初始速度。`RadialShooter` 类允许用户指定一个速度范围 (米/秒) 以及弧度值表示的方向。方向由两个角度指定：`theta` 角 - 与 Z 轴夹角，`phi` 角 - 与 XY 平面夹角。

计数器 (`osgParticle::Counter`) - 控制每一帧产生的粒子数。`RandomRateCounter` 类允许用户指定每帧产生粒子的最大和最小数目。

标准放射极 (`osgParticle::ModularEmitter`) - 一个标准放射极包括一个计数器，一个放置器和一个发射器。它为用户控制粒子系统中多个元素提供了一个标准机制。

粒子系统更新器 (`osgParticle::ParticleSystemUpdater`) - 用于自动更新粒子。将其置于场景中时，它会在拣选遍历中调用所有“存活”粒子的更新方法。

标准编程器 (`osgParticle::ModularProgram`) - 在单个粒子的生命周期中，用户可以使用 `ModularProgram` 实例控制粒子的位置。`ModularProgram` 需要与 `Operator` 对象组合使用。

计算器 (`osgParticle::Operator`) - 提供了控制粒子在其生命周期中的运动特性的方法。用户可以改变现有 `Operator` 类实例的参数，或者定义自己的 `Operator` 类。`OSG` 提供的 `Operator` 类包括：`AccelOperator` (应用常加速度)，`AngularAccelOperator` (应用常角加速度)，`FluidFrictionOperator` (基于指定密度和粘性的流体运动进行计算)，以及 `ForceOperator` (应用常力)。

代码: [🔗](#)

为了使用上面的类创建高度自定义化的粒子系统，我们可以遵循以下的步骤。(括号中的步骤是可选的，如果所建立的粒子系统比较基本，也可以忽略)

- - 创建粒子系统实例并将其添加到场景。
 - (设置粒子系统的渲染状态属性。)
 - 创建粒子对象并将其关联到粒子系统。
 - (设置粒子的参数。)
 - 创建粒子系统更新器，将其关联到粒子系统实例，并添加到场景中。
 - (创建标准放射极，以定义：计数器 - 每帧创建的粒子数，放置器 - 粒子的出生位置，发射器 - 初始速度。)
 - (将标准放射极关联到粒子系统。)

- （创建 `ModularProgram` 标准编程器实例，以控制粒子在生命周期中的运动：首先创建并定义 `Operator` 计算器；然后添加计算器到标准编程器。）

下面的代码将完成上述的步骤。首先，我们需要建立基本的坦克和地形模型。（稍后我们再添加坦克模型到场景中）

```
osg::Group* rootNode = new osg::Group();

osg::Node* terrainNode = new osg::Node();
osgProducer::Viewer viewer;

viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);

terrainNode = osgDB::readNodeFile("\\Models\\JoeDirt\\JoeDirt.flt");
if (! terrainNode)
{
    std::cout << "Couldn't load models, quitting." << std::endl;
    return -1;
}
rootNode->addChild(terrainNode);

osg::Node* tankNode = osgDB::readNodeFile("\\Models\\T72-Tank\\T72-tank_des.flt");
if ( ! tankNode)
{
    std::cout << "no tank" << std::endl;
    return -1;
}
```

建立粒子系统的基本步骤是：创建一个粒子系统对象，一个更新器对象，以及一个粒子对象，同时设置场景。

```
// 创建并初始化粒子系统。
osgParticle::ParticleSystem *dustParticleSystem = new osgParticle::ParticleSystem;

// 设置材质，是否放射粒子，以及是否使用光照。
dustParticleSystem->setDefaultAttributes(
"C:\\Projects\\OpenSceneGraph\\OpenSceneGraph-Data\\images\\dust2.rgb",
false, false);

// 由于粒子系统类继承自 Drawable 类，因此我们可以将其作为 Geode 的子节点加入场景。
osg::Geode *geode = new osg::Geode;

rootNode->addChild(geode);
geode->addDrawable(dustParticleSystem);

// 添加更新器，以实现每帧的粒子管理。
osgParticle::ParticleSystemUpdater *dustSystemUpdater = new osgParticle::ParticleSystemUpdater;

// 将更新器与粒子系统对象关联。
dustSystemUpdater->addParticleSystem(dustParticleSystem);
// 将更新器节点添加到场景中。
```

```
rootNode->addChild(dustSystemUpdater);
```

```
// 创建粒子对象，设置其属性并交由粒子系统使用。
```

```
osgParticle::Particle smokeParticle;
```

```
smokeParticle.setSizeRange(osgParticle::rangef(0.01,20.0)); // 单位：米
```

```
smokeParticle.setLifeTime(4); // 单位：秒
```

```
smokeParticle.setMass(0.01); //单位：千克
```

```
//设置为粒子系统的缺省粒子对象。
```

```
dustParticleSystem->setDefaultParticleTemplate(smokeParticle);
```

下面的代码将使用标准放射极对象来设置粒子的一些基本参数：例如每帧创建的粒子数，新生粒子的产生位置，以及新生粒子的速度。

```
//创建标准放射极对象。（包括缺省的计数器，放置器和发射器）
```

```
osgParticle::ModularEmitter *emitter = new osgParticle::ModularEmitter;
```

```
// 将放射极对象与粒子系统关联。
```

```
emitter->setParticleSystem(dustParticleSystem);
```

```
//获取放射极中缺省计数器的句柄，调整每帧增加的新粒子数目。
```

```
osgParticle::RandomRateCounter *dustRate =
```

```
static_cast<osgParticle::RandomRateCounter *>(emitter->getCounter());
```

```
dustRate->setRateRange(5, 10); // 每秒新生成 5 到 10 个新粒子。
```

```
// 自定义一个放置器，这里我们创建并初始化一个多段放置器。
```

```
osgParticle::MultiSegmentPlacer* lineSegment = new osgParticle::MultiSegmentPlacer();
```

```
//向放置器添加顶点，也就是定义粒子产生时所处的线段位置。
```

```
// （如果将坦克和标准放射极定义与同一位置，那么我们可以实现一种
```

```
// 灰尘粒子从坦克模型后下方的延长线上产生的效果。）
```

```
lineSegment->addVertex(0,0,-2);
```

```
lineSegment->addVertex(0,-2,-2);
```

```
lineSegment->addVertex(0,-16,0);
```

```
// 为标准放射极设置放置器。
```

```
emitter->setPlacer(lineSegment);
```

```
// 自定义一个发射器，这里我们创建并初始化一个 RadialShooter 弧度发射器。
```

```
osgParticle::RadialShooter* smokeShooter = new osgParticle::RadialShooter();
```

```
// 设置发射器的属性。
```

```
smokeShooter->setThetaRange(0.0, 3.14159/2); // 弧度值，与 Z 轴夹角。
```

```
smokeShooter->setInitialSpeedRange(50,100); //单位：米/秒
```

```
// 为标准放射极设置发射器。
```

```
emitter->setShooter(smokeShooter);
```


现在我们将把放射极和坦克模型作为 Transform 变换节点的子节点添加到场景中。放射极和坦克均由变换节点决定其位置。刚才定义的放置器将会根据变换的参量安排粒子的位置。

```
osg::MatrixTransform * tankTransform = new osg::MatrixTransform();
tankTransform->setUpdateCallback( new orbit() ); // 回调函数，使节点环向运动。
```

```
// 把放射极和坦克模型添加为变换节点的子节点。
```

```
tankTransform->addChild(emitter);
tankTransform->addChild(tankNode);
rootNode->addChild(tankTransform);
```

下面的代码将创建一个标准编程器 ModularProgram 实例，用于控制粒子在生命周期中的更新情况。标准编程器对象使用 Operator 计算器来实现对粒子的控制。

```
// 创建标准编程器对象并与粒子系统相关联。
```

```
osgParticle::ModularProgram *moveDustInAir = new osgParticle::ModularProgram;
moveDustInAir->setParticleSystem(dustParticleSystem);
```

```
// 创建计算器对象，用于模拟重力的作用，调整其参数并添加给编程器对象。
```

```
osgParticle::AccelOperator *accelUp = new osgParticle::AccelOperator;
accelUp->setToGravity(-1); // 设置重力加速度的放缩因子。
moveDustInAir->addOperator(accelUp);
```

```
// 向编程器再添加一个计算器对象，用于计算空气阻力。
```

```
osgParticle::FluidFrictionOperator *airFriction = new osgParticle::FluidFrictionOperator;
airFriction->setFluidToAir();
```

```
//airFriction->setFluidDensity(1.2929/*air*//*5.0f);
```

```
moveDustInAir->addOperator(airFriction);
```

```
//最后，将编程器添加到场景中。
```

```
rootNode->addChild(moveDustInAir);
```

下面的代码就是仿真循环的内容了。

```
// add a viewport to the viewer and attach the scene graph.
```

```
viewer.setSceneData(rootNode);
```

```
// create the windows and run the threads.
```

```
viewer.realize();
```

```
while( !viewer.done() )
```

```
{
```

```
    // wait for all cull and draw threads to complete.
```

```
    viewer.sync();
```

```
    // update the scene by traversing it with the update visitor which will
```

```
    // call all node update callbacks and animations.
```

```
    viewer.update();
```

```
    // fire off the cull and draw traversals of the scene.
```

```
    viewer.frame();
```

```

}

// wait for all cull and draw threads to complete before exit.
viewer.sync();

return 0;

```

第十五课第二节 粒子系统的保存以及读取

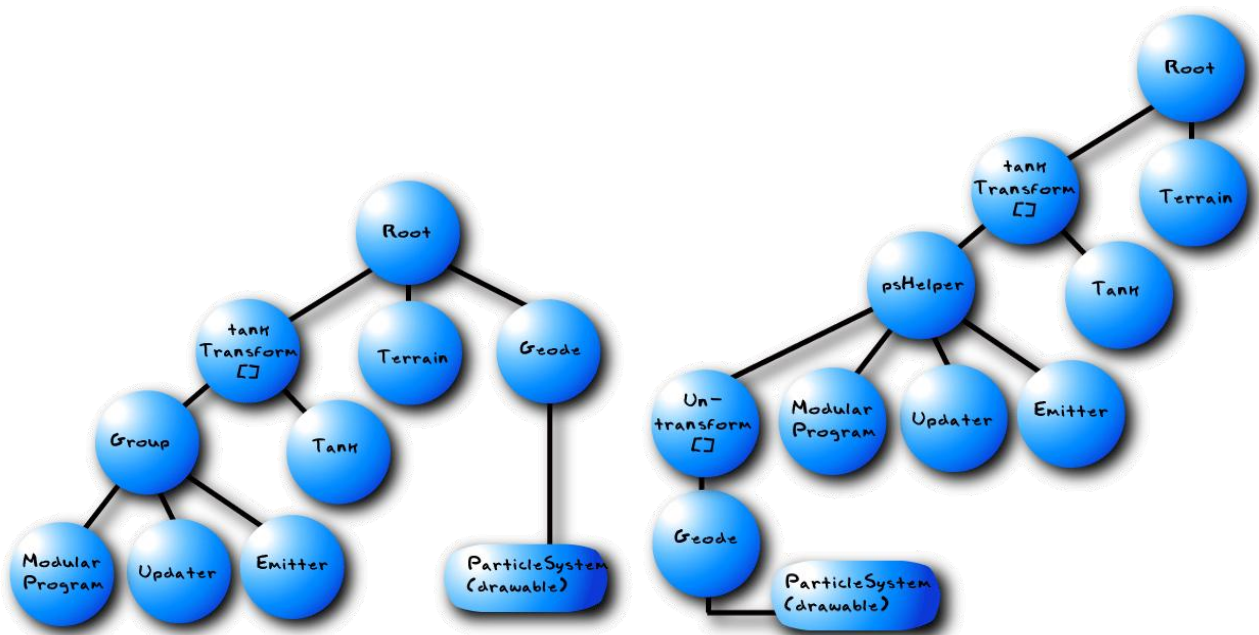
如果你已经在上一章创建了一个粒子系统，你应该已经构建了一个类似图 1 的场景图形。粒子系统对象（**Drawable**）作为 **Geode** 叶节点的子对象，而 **Geode** 的父节点是根节点。其它的粒子系统元素（标准编译器，更新器，放射极对象）均作为位置变换节点的子节点存在。我们的粒子系统的结构设计可谓“通用而合理的”：上一教程中我们已经可以看到场景中坦克的周围有烟尘效果环绕。粒子系统对象并非作为位置变换节点的子节点，因此其稳定性更好。

如果我们把一个漂亮的 **OSG** 效果作为 **osg** 文件保存，那么可能会出现一个小问题。当我们使用 **osgDB::ReadNodeFile()** 读取文件时，我们将仅仅得到一个节点的指针。这个节点将把所有粒子系统相关的元素作为子节点关联。如果我们把这个节点作为变换节点的子节点加入场景中，那么粒子系统（**Drawable**）也会随之变换。也就是说，这个粒子系统已经不再是“通用而合理的”了。

合理的粒子系统要求其粒子系统对象不能进行变换。因此我们有别的方法来实现这一要求。例如读取文件并获取 **Drawable** 对象的句柄，然后将这个 **Drawable** 对象从读入节点的子节点中去除，并重新添加到根节点上。但是如果你有较多数量的不同效果需要处理的话，这种方法就显得繁琐了：需要不停地取消和关联，不停地更改节点树，等等。

将所有的粒子效果置于场景图形的同一节点之下也是有好处的。但是我们必须保证 **Drawable** 对象不能被任何变换节点所影响。要实现这一目的，我们可以针对粒子系统类对象设置一个“绝对”变换。这可能对拣选过程的算法产生不好的影响。（因为查找“绝对变换”节点是需要进行遍历的，它的所有父节点也会因此进行遍历）下面的方案也许更加可行一些：创建一个变换节点，它作为粒子系统对象的父类，将自己以上所有的变换效果均进行反转。它的实现结构如图 2 所示。其源代码可以在下面的地址下载：

<http://www.nps.navy.mil/cs/sullivan/osgTutorials/Download/psHelper.cpp>



Attachments

- [ParticleSystem\(original\).gif](#) (74.4 kB) - added by osg on 08/24/07 14:11:32.
- [ParticleSystem\(helper\).gif](#) (80.1 kB) - added by osg on 08/24/07 14:11:53.

第十六课第一节 节点遮掩概述（基于 OSG 1.2 版本）

节点遮掩的概念允许用户有选择地遍历场景中的图形子树。例如，如果你准备设置一个模型是否可视化的属性，你可以通过调整拣选遍历所用到的节点掩码来实现这一需求。节点遮掩在表达不同摄像机视口下的场景外观时，以及设置节点是否进行渲染时也十分有用。

每个摄像机和节点（包括 `osg::Group` 节点，也就是场景图形中的子树）自身都有一个掩码。在渲染前的场景图形遍历中，系统将进行当前节点掩码和摄像机掩码的逻辑“或”运算比较。如果比较的结果不为 0，则遍历过程继续执行，节点所包含的几何体信息将被传递到渲染系统中。如果逻辑“或”的结果为 0，则之后的遍历过程中止，与该节点相关的几何体也就因此不再被渲染。

代码: [1](#)

作为一个简单的介绍，下面的代码将设置一个包含地形和坦克模型的场景，并在场景中安置两台摄像机。地形的节点掩码为 0x1，坦克的节点掩码为 0x2。场景的两台摄像机视口中，顶部的摄像机掩码为 0x3，底部的摄像机掩码为 0x2。

下面代码的第一部分负责读取和设置场景。`setupCameras` 函数用于向场景中添加摄像机，其中两个用于实现节点遮掩的效果，另外两个作为参照。

（作为参照的摄像机将负责清空背景并渲染文字信息）

```
osg::ref_ptr<osg::Group> root = new osg::Group();
osg::ref_ptr<osg::Node> tankNode = NULL;
osg::ref_ptr<osg::Node> terrainNode = NULL;
```

```

osg::ref_ptr<osg::PositionAttitudeTransform>
    tankXform = new osg::PositionAttitudeTransform;

Producer::CameraConfig* myCameraConfig = setupCameras();
osgProducer::Viewer viewer(myCameraConfig);

osgDB::setDataFilePathList(buildFilePathList());

tankNode = osgDB::readNodeFile("t72-tank_des.flt");
terrainNode = osgDB::readNodeFile("JoeDirt.flt");

tankXform->setPosition( osg::Vec3(5,5,8) );

root->addChild(terrainNode.get());
root->addChild(tankXform.get());

```

用于设置节点掩码的代码如下所示：

```

terrainNode->setNodeMask(0x1);
tankXform->setNodeMask(0x2);

```

设置摄像机的掩码相对比较复杂。与摄像机相关联的场景视图同样需要一个继承掩码，用于设置拣选遍历的工作模式。下面的代码将首先设置场景的继承掩码，然后使用 `setCullMask` 设置场景的掩码。

(Code for setting the masks for the two cameras we don't care about is left out for clarity.)

```

int inheritanceMask =
    (osgUtil::SceneView::VariablesMask::ALL_VARIABLES &
     ~osgUtil::SceneView::VariablesMask::CULL_MASK);

//设置顶部摄像机的掩码
viewer.getSceneHandlerList().at(1)->getSceneView()->setInheritanceMask(inheritanceMask);
viewer.getSceneHandlerList().at(1)->getSceneView()->setCullMask(0x3);

//设置底部摄像机的掩码
viewer.getSceneHandlerList().at(2)->getSceneView()->setInheritanceMask(inheritanceMask);
viewer.getSceneHandlerList().at(2)->getSceneView()->setCullMask(0x2);

```

最后我们进入仿真循环。

```

while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}

```

第十六课第二节 节点遮掩示例（基于 OSG 1.2 版本）

这一次我们将创建一个包含地形，坦克和立方体的场景。坦克和立方体处于同样的位置变换约束下。我们设置两个视图。主视图是通常的 `osgProducer::Viewer`，其视口使用鼠标/键盘控制。屏幕靠下的部分有一个自顶向下显示的视图。在主视图中，我们将看不到立方体，但是能看到坦克和地形。而在另一个视图中，我们可以看到地形和立方体但无法看到坦克。（这有点类似主视图和鹰眼图的效果，鹰眼图中使用立方块来表示坦克的位置）注意：摄像机的载入和渲染是依照字母排序的，这一点在摄像机图像有重叠时非常重要。

代码如下： [🔗](#)

```
#include <osg/PositionAttitudeTransform>
#include <osg/Group>
#include <osg/Node>
#include <osgDB/ReadFile>
#include <osgDB/FileUtils>
#include <osgProducer/Viewer>

#include <osgText/Font>
#include <osgText/Text>
#include <osg/MatrixTransform>
#include <osg/Geode>
#include <osg/Projection>
#include <osg/ShapeDrawable>
#include <osg/Geometry>
#include <osg/MatrixTransform>
#include <osg/PositionAttitudeTransform>
#include <osgUtil/CullVisitor>
#include <osgUtil/SceneView>

class circleAimlessly : public osg::NodeCallback
{
public:
    circleAimlessly(): _angle(0.0) {}
    virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
    {
        osg::MatrixTransform *tx = dynamic_cast<osg::MatrixTransform *>(node);
        if( tx != NULL )
        {
            _angle += M_PI/180.0;
            tx->setMatrix( osg::Matrix::translate( 100.0, 0.0, 20.0 ) *
                osg::Matrix::rotate( _angle, 0, 0, 1 ) );
        }
        traverse(node, nv);
    }
private:
    float _angle;
};

Producer::CameraConfig* setupCameras()
{
```

```

Producer::CameraConfig* myCameraConfig = new Producer::CameraConfig();
Producer::Camera* topDownCamera = new Producer::Camera();
Producer::Camera* mainCamera = new Producer::Camera();

mainCamera->setProjectionRectangle(0.0f, 1.0f, 0.0f, 1.0f);
mainCamera->setShareLens(false);
mainCamera->getLens()->setAutoAspect(true);

topDownCamera->setProjectionRectangle(0.25f, 0.75f, 0.0f, 0.5f);
topDownCamera->setShareLens(false);
topDownCamera->getLens()->setAutoAspect(true);

myCameraConfig->addCamera("topDownCamera",topDownCamera);
myCameraConfig->addCamera("mainCamera",mainCamera);

Producer::RenderSurface* rsOne = mainCamera->getRenderSurface();
topDownCamera->setRenderSurface( rsOne );
return myCameraConfig;
}

bool setupScene(Producer::ref_ptr<osg::Group> &rootNode,
                Producer::ref_ptr<osg::Group> &ownTank )
{
    if (rootNode.get() || ownTank.get())
        return false;

    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\\");
    osgDB::setDataFilePathList(pathList);

    rootNode = new osg::Group();
    Producer::ref_ptr<osg::Node> terrainNode =
        osgDB::readNodeFile("JoeDirt.flt");
    if (!terrainNode)
    {
        std::cout << " no terrain! " << std::endl;
        return false;
    }
    rootNode->addChild(terrainNode.get());

    ownTank = (osg::Group*)
        osgDB::readNodeFile("T72-tank_des.flt");
    if( ! ownTank)
    {

```

```

        std::cout << "no Tank" << std::endl;
        return false;
    }

    osg::MatrixTransform * tankXForm =
        new osg::MatrixTransform();
    tankXForm->setUpdateCallback( new circleAimlessly() );
    rootNode->addChild(tankXForm);
    tankXForm->addChild(ownTank.get());

    osg::Box* unitCube = new osg::Box( osg::Vec3(0,0,0), 20.0f);
    osg::ShapeDrawable* unitCubeDrawable = new osg::ShapeDrawable(unitCube);
    osg::Geode* basicShapesGeode = new osg::Geode();
    basicShapesGeode->addDrawable(unitCubeDrawable);
    tankXForm->addChild(basicShapesGeode);
    basicShapesGeode->setNodeMask(0x2);
    ownTank->setNodeMask(0x1);
    terrainNode->setNodeMask(0x3);

    return true;
}

int main()
{
    Producer::ref_ptr<osg::Group> rootNode;
    Producer::ref_ptr<osg::Group> tankNode;

    if (!setupScene(rootNode,tankNode))
        return -1;

    Producer::CameraConfig* myCameraConfig = setupCameras();
    osgProducer::Viewer viewer(myCameraConfig);

    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    viewer.setSceneData( rootNode.get() );
    viewer.realize();

    int inheritanceMask =
        (osgUtil::SceneView::VariablesMask::ALL_VARIABLES &
         ~osgUtil::SceneView::VariablesMask::CULL_MASK);

    viewer.getSceneHandlerList().at(0)->getSceneView()->setInheritanceMask(inheritanceMask);
    viewer.getSceneHandlerList().at(0)->getSceneView()->setCullMask(0x1);
    viewer.getSceneHandlerList().at(1)->getSceneView()->setInheritanceMask(inheritanceMask);
    viewer.getSceneHandlerList().at(1)->getSceneView()->setCullMask(0x2);

    osg::Matrixd* topDownViewMatrix = new osg::Matrixd(

```

```

        osg::Matrixd::rotate(osg::DegreesToRadians(-90.),1,0,0) *
        osg::Matrixd::translate(0,0,450) );
topDownViewMatrix->set(
    topDownViewMatrix->inverse(*topDownViewMatrix));

while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.getCameraConfig()->findCamera("topDownCamera")->setViewByMatrix(
        Producer::Matrix(topDownViewMatrix->ptr()) *
        Producer::Matrix::rotate( -M_PI/2.0, 1, 0, 0) );
    viewer.frame();
}
}

```