

OSGFreeS

OSG 入门基础教程

Step Into

OpenSceneGraph

杨石兴 (*FreeSouth*), 曹明亮编著

郑州大学虚拟现实实验室

This book to my deepest love girl Xiao Nan

INDEX

我们该如何学习 OSG.....	- 7 -
第一章 MFC 基础	- 8 -
1.1 最精简的单文档 MFC 框架	- 8 -
1.2 MFC 常用的几种基本操作	- 24 -
1.2.1 修改鼠标.....	- 24 -
1.2.2 透明对话框	- 26 -
1.2.3 客户区全屏	- 28 -
1.2.4 音乐播放.....	- 30 -
第二章 基本几何体的绘制.....	- 36 -
2.1 构建 OSG+MFC 框架.....	- 36 -
2.1.1 OSG 渲染过程	- 36 -
2.1.2 MFC+OSG 控制	- 41 -
2.2 矩阵操作.....	- 50 -
2.2.1 矩阵操作的数学原理	- 50 -
2.2.2 osgViewer 对于视口的控制	- 52 -
2.2.3 绘制基本几何体	- 54 -
2.3 绘制各种几何体	- 61 -
2.4 综合例子: 制做时钟	- 65 -
第三章 场景漫游.....	- 76 -
3.1 如何编写动态链接库	- 76 -
3.1.1 OSG 中的 DLL	- 76 -
3.1.2 编写 TravelManipulator.DLL.....	- 78 -
3.1.3 测试 TravelManipulator.DLL.....	- 87 -
3.2 碰撞检测.....	- 89 -
3.2.1 最简单的碰撞检测.....	- 89 -
3.2.2 常用的碰撞检测方法	- 90 -
3.2.2 空间划分 BSP 树与八叉树	- 91 -
3.3 项目协作.....	- 93 -
第四章 路径漫游.....	- 94 -
4.1 应用 .PATH 文件	- 95 -
4.1.1 PATH 文件结构	- 95 -
4.1.2 使用 PATH 文件	- 97 -
4.2 曲线生成插值算法	- 99 -
4.2.1 多项式插值	- 100 -
4.2.2 Hermite 和 Cardinal 曲线.....	- 102 -
4.2.3 Bézier 曲线	- 104 -
4.3 路径漫游.....	- 104 -
4.3.1 路径漫游流程.....	- 105 -
4.3.2 增强 TravelManipulator.dll.....	- 107 -
4.3.3 路径漫游功能演示.....	- 117 -
第五章 粒子系统.....	- 130 -

5.1	OSG 中粒子系统要素	130 -
5.1.1	粒子系统的模拟过程	130 -
5.1.2	osgParticle 中的类	132 -
5.2	爆炸物模拟	143 -
5.3	雨,雪,喷泉的模拟	145 -
5.3.1	雨,雪的模拟	146 -
5.3.2	喷泉的模拟	154 -
5.3.3	雾	158 -
第六章	更新&回调	163 -
6.1	Event&HUD	163 -
6.1.1	事件响应	163 -
6.1.2	HUD 与显示汉字	167 -
6.1.3	显示当前位置的例子	172 -
6.2	MatrixTransform&AnimationPath	178 -
6.2.1	MatrixTransformCallBack	178 -
6.2.2	AnimationPathCallback	180 -
6.3	NodeVisitor	181 -
6.3.1	访问模型结点	181 -
6.3.2	osgFX::Scribe	185 -
6.3.3	Scribe&Pick&Event	187 -
第七章	Viewport&Camera	193 -
7.1	裁切技术	193 -
7.1.1	平面的表示	193 -
7.1.2	裁切分类	195 -
7.1.3	OSG 中的裁切类	196 -
7.2	Camera	203 -
7.2.1	Camera&HUD	203 -
7.2.2	多视图	207 -
第八章	光照&Billboard&LOD	210 -
8.1	Light	210 -
8.1.1	材质与着色	210 -
8.1.2	光照分量	212 -
8.1.3	osg::light	213 -
8.2	Billboard	220 -
8.2.1	Billboard 的数学原理	220 -
8.2.2	osg::Billboard	220 -
8.3	LOD	224 -
8.3.1	LOD 过程	224 -
8.3.2	osg::LOD	224 -
8.3.3	osgSim::Imposter	226 -
关于本书	229 -

我们该如何学习 OSG

OSG 发展至今以来，以非常惊人的速度在 VR 行业占据非常重要的地位，其开源、易学、易用的特性使其在短期内取得了很大的成功。但是国内外还没有关于 OSG 的入门参考书出版，希望本书能给广大初学者以帮助。

OpenSceneGraph 从字面上看只是一个场景管理系统，从某种意义上讲，OSG 的对三维动画等等的支持还不是很强，打个比方来说：OSG 不适合做游戏引擎。因此工业控制，城市规划，机械仿真等方面 OSG 可以带来极大的方便。

学习 OSG 是一个系统的过程，初学者入门可以专著的学习各个功能模块，如离子系统，路径漫游，流动纹理等等，完成某个功能往往需要一定大胆的想法与猜测、基于对 OSG 的理解以及对自己编程能力的自信、在现有的基础上再加以创新，根据项目需要来组织各个模块，完成各项功能。

学习 OSG 大都是从研究 OSG 的例子开始，OSG 的源文件中有一些例子，国内 WWW.VRDEV.NET 论坛上也有关于各种例子的介绍，是中国广大 OSG 爱好者的交流平台。官方网站上的例子往往会第一时间发布在：<http://www.openscenegraph.net/osgwiki/pmwiki.php/Tutorials/Tutorials>。

也可以在此处订阅邮件列表：地址：<http://www.openscenegraph.net/archiver/osg-users/>。

在本书的编写过程中，受到了郑州大学虚拟现实实验室曹明亮与赵老师的大力支持与技术指导，由于作者水平有限，时间极为紧迫，所以书中不足之处甚多，望广大读者遇到时能来信给予极时的纠正，本书的初衷是学习 OSG 的功能，帮助初学者入门，并不是做一个成功的项目与完美的新功能，所以光盘的例子中有许多需要注意的问题与 BUG，到时候会在各章指明。

作者

二零零七年三月二十四日

第一章 MFC 基础

本书将着重讲述 OSG 与 MFC 的结合使用，在实际应用中，WIN32 与 MFC 和 OSG 的结合使用的比较频繁，MFC 与 OSG 结合更具有条理性，更加易学，易用。本章主要学习 MFC 在 OSG 的应用中常用的技巧，为以后的学习打基础。

本章包含以下内容：建立精简的 MFC 单文档框架，MFC 的基本操作，透明窗口制作，鼠标更改，全屏操作，以及音乐播放。

1.1 最精简的单文档 MFC 框架

MFC 自己建立的默认框架过于雍肿，而且在功能上有些根本不需要且不符合要求，而从空项目重建 MFC 单文档程序对初学者来说难度太大，这就需要对默认生成的单文档应用程序加以改造，从而完成要求。

示例一：最精简的 MFC 单文档框架。

第一步：安装完成 VS2005 后，点击新建→新建项目→在项目类型中选择：MFC，在模版中选择：MFC 应用程序，在名称中填入项目名称，比如 osgFree，点击确定。

这时将会出现 MFC 应用程序向导，在概述中大概描述了该程序的基本特征，默认是多文档 | 无数数据库支持 | 不支持复合文档。如果经常使用 MFC，以及对其内部结构有一定的了解，随便建一个便可以自由的通过修改默认源码在自己需要的类型之前来回转换，这里我们少走一些弯路点击：应用程序类型标签或者下一步按钮，页面需要选择如下：



图 1.1

在 vs2005 中，基本全部采用宽字符，这对从 vs2003 的初学者来说进行很多操作带来了极大的不便，点击使用 unicode 库，因为在 2005 中 MFC 已经取消了对窄字符的支持，使用很多宏以及函数可以在各种

字符间轻易的转换，后面将会做介绍。点击下一步，保持默认选择再点击下一步出现如下页面：



图 1.2

在文件扩展名中填入*.ive，电脑中所有的*.ive 文档默认的打开方式为你所建立的这个应用程序。主框架标准中修改的字符将会出现在框架窗口的上方，做如图修改后点击下一步，出现对数据库是否支持的页面，在 OSG 的编程中往往需要使用到数据库，且往往采用 ODBC 方式而非 C/S 格式，在本书中的所有例子中将不会涉及数据库的相关内容，因为选择无，而后点击下一步，出现如图所示页面：



图 1.3

各个选项在选择后将会影响到程序的框架类型，当然无论做怎么样的选择，在程序中都可以再次修改，

比如这里没有选择最大化框，在程序中也可以通过代码进行修改。MFC 是非常自由的，应用程序配置向导只是向着大多数人最常用的方式来进行配置。再次点击下一步，把打印和打印预览选项剔除，当然如果你不在程序中使用控件，ACTIVEX 也可以剔除。

点击下一步在生成的类中保持默认选择，不同的类决定不同的模版，如有滚动条，具有文档模版等等，在这里点击完成。

第二步：在解决方案资源管理器中对如下文件 osgFreeDoc.h osgFreeView.h osgFreeDoc.cpp osgFreeView.cpp 点击右键，选择删除，删除这四个文件，这表示不需要文档支持，且自己要重建 View 类。

第三步：重建 view 类，具体方法为选择菜单：项目→添加类、或者在类视图中，对最上面的 osgFree 标签点击右键→添加类选项，而后选择 C++类，点击确定后出现如下图示：（在类视图中对某类点击右键可以在此类中添加函数和变量，也可以手动添加，以后将不作详细介绍）



图 1.4

类名中输入 osgFreeViewer，.H 与 .CPP 文件名会自动生成，基类中添入 CWnd，点击完成。

第四步：修改代码，现在把各个需要改动的文档，在其改动后的结果示例如下，各行均有详细的注释，对于改变的代码，我们将以粗体显示，以便于理解：

```
MainFrm.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
//该语句保证该头文件只被编译一次，相当于以前的#ifdef #endif对
#pragma once
//包含自己创建的视窗口头文件，此文件是你刚才建立的viewer类的头文件
#include "osgFreeViewer.h"
class CMainFrame : public CFrameWnd
{
//需要动态分配,故把构造函数改为公有
```

```

public :
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)
// 属性
public:

// 操作
public:

// 重写
public:
    //在这个函数中可以修改窗口的样式风格等等，是个经常用的函数体
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
// 实现
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    //在调试与运行模式下，下面的语句会有不同，在调试模式下，需要保存更多的信息，以及保证很多断言成功截获，不过
    //对于下面这两个函数，一般没有人太过注意
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected: // 控件条嵌入成员
    //删除不需要的工具栏与状态栏，在OSG中，我们不会使用它们
    //CStatusBar m_wndStatusBar;
    //CToolBar m_wndToolBar;

    //创建视图窗口的对象
    CosgFreeViewer m_wndCosgFreeViewer;
// 生成的消息映射函数
protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    DECLARE_MESSAGE_MAP()
public :
    //渲染前的准备工作
    BOOL PreToRen(void) ;
    //下面是设置焦点的消息响应，消息响应的添加步骤可以是：在类视中对类CMainFrame击右键→属性，在属性面板//中选中，在下拉列表选中WM_SETFOCUS 在右边点击：添加OnSetFocus WM意思是WINDOWS //MESSAGE，当主框架得到焦点时，可以通过此函数将焦点设置到想要设置的窗口，本程序是CosgFreeViewer
    afx_msg void OnSetFocus (CWnd* pOldWnd) ;
    //消息响应

```

```
        BOOL OnCmdMsg (UINT nID, int nCode, void* pExtra, AFX_CMDHANDLERINFO*  
pHandlerInfo) ;  
}; //MainFrm.h 结束
```

MainFrame.cpp

//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com

// MainFrm.cpp : CMainFrame 类的实现

//

#include "stdafx.h"

#include "osgFree.h"

#include "MainFrm.h"

#ifdef _DEBUG

#define new DEBUG_NEW

#endif

// CMainFrame

IMPLEMENT_DYNCREATE (CMainFrame, CFrameWnd)

//下面是消息响应序列，在vs中添加消息，在下面的代码中有反映，一般来讲也可以手动来添加，通过vs添加比较保险

BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

//创建窗口的消息响应

ON_WM_CREATE ()

//设置焦点的消息响应

ON_WM_SETFOCUS ()

END_MESSAGE_MAP ()

// CMainFrame 构造/析构

CMainFrame::CMainFrame ()

{

// TODO: 在此添加成员初始化代码

}

// CMainFrame 析构

CMainFrame::~~CMainFrame ()

{

}

//创建主窗口消息映射函数

int CMainFrame::OnCreate (LPCREATESTRUCT lpCreateStruct)

{

if (CFrameWnd::OnCreate (lpCreateStruct) == -1)

return -1; //MainFrame.cpp待续

//MainFrame.cpp续

//删除工具栏与状态栏的创建，注意，下面粗体的是被注释掉的

```

        /*if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE |
CBRS_TOP
        | CBRG_GRIPPER | CBRG_TOOLTIPS | CBRG_FLYBY | CBRG_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
        {
            TRACE0("未能创建工具栏\n");
            return -1;        // 未能创建
        }
        if (!m_wndStatusBar.Create(this) ||
            !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
        {
            TRACE0("未能创建状态栏\n");
            return -1;        // 未能创建
        }
        // TODO: 如果不需要工具栏可停靠, 则删除这三行
        m_wndToolBar.EnableDocking(CBRG_ALIGN_ANY);
        EnableDocking(CBRG_ALIGN_ANY);
        DockControlBar(&m_wndToolBar);
        */

        //添加视图窗口的创建, 下面我们会介绍一下这个函数
        if(!m_wndCosgFreeViewer.Create(NULL, NULL, AFX_WS_DEFAULT_VIEW,
            CRect(0, 0, 0, 0), this, AFX_IDW_PANE_FIRST, NULL))
        {
            TRACE0("未能创建窗口\n");
            return -1;
        }
        return 0;
    }
}
//MainFrame.cpp待续

```

函数:

```

virtual BOOL Create( LPCTSTR lpszClassName, LPCTSTR lpszWindowName, DWORD
dwStyle, const RECT& rect,
    CWnd* pParentWnd, UINT nID, CCreateContext* pContext = NULL);

```

参数: **lpszClassName:** 指类名, 一般是WNDCLASS的结构体名, 此结构体中包含有窗口风格、窗口标题, 窗口图标、鼠标、背影等等, 如果指向NULL, 代表使用CWnd的默认属性

lpszWindowName: 指windows name, 一般为一个字符串

dwStyle: 指窗口风格, 使用AFX_WS_DEFAULT_VIEW指使用WINDOWS默认的窗口风格

窗口风格可以是如下些值:

WS_BORDER 创建一个带边框的窗口

WS_CAPTION 创建带标题栏的窗口, 不能与WS_DLGMFRAME同时使用, 因为对话框没有标题栏

WS_CHILD 说明该窗口为子窗口类型, 不能与WS_POPUP一同使用

WS_CHILDWINDOW 与WS_CHILD相同

WS_CLIPCHILDREN 在主窗口做图等操作时将不会包含子窗口占有的区域，在创建主窗口时经常使用

WS_CLIPSIBLINGS 把子窗口关联起来，消息传递时同时更新，只能与WS_CHILD共同使用

WS_DISABLED 创建一个开始时不能使用的窗口

WS_DLGFRAE 带有双边框，但是没有标题栏

WS_GROUP 分组，与按钮的分组类似

WS_HSCROLL 带有水平的滚动条

WS_ICONIC 创建一个开始最小化的窗口与 WS_MINIMIZE 相同！

WS_MAXIMIZE 创建一个最大化窗口

WS_MAXIMIZEBOX 带有最大化按钮

WS_MINIMIZE 创建 初始最小化的窗口，只能与 WS_OVERLAPPED 同时使用

WS_MINIMIZEBOX 带有最小化按钮的窗口

WS_OVERLAPPED 创建一个带有标题栏与边框的窗口

WS_OVERLAPPEDWINDOW 代表以下几种类型WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, 以及 WS_MAXIMIZEBOX 是非常常用的

WS_POPUP 弹出风格，不能与WS_CHILD 公用

WS_POPUPWINDOW 含有以下几种风格 WS_BORDER, WS_POPUP, 和 WS_SYSMENU 其中WS_CAPTION 必须包含此，而使菜单可见

WS_SIZEBOX 与 WS_THICKFRAME 相同

WS_SYSMENU 创建带有系统菜单的窗口

WS_TABSTOP 可使用TAB键进行切换

WS_THICKFRAME 带有薄边框，可以改变窗口大小

WS_TILED 与 WS_OVERLAPPED 相同

WS_TILEDWINDOW 具有以下几种风格WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, 以及WS_MAXIMIZEBOX styles.与 WS_OVERLAPPEDWINDOW相同

WS_VISIBLE 在最初时可见，一般这个是要选的

WS_VSCROLL 带有竖直的滚动条窗口

注：一般而言，WINDOWS中的常量，列如窗口风格，分别为2的N次方即，比如二进制的0001 0010 0100当从现有的风格中加上某种风格时可以使用位或运算，如果当前风格为CS，即：CS |=WS_VSCROLL，这就代表从现有风格中加入WS_VSCROLL，如果十分的有把握甚至可以使用数字来代替风格，最常见的有Message(L"warning",L"WARNING",1,1)；后面的1和1其实各指代一种风格与图标，从当前风格中去掉一种风格为 CS &= ~WS_VSCROLL，也就是CS与该风格的反取并，这样该风格对应的二进制位在CS中将被将被清零，其它的位保持不变

const RECT& rect: 指窗口所占矩形，这里全部填零意思为填充整个框架窗口，也可以指定部分框架窗口或是越过框架窗口的界限，这都是可以的，当拉开框架窗口时，越界的内容将显示，如果不自行对越界的内容进行裁剪，也将占用内存开销。

CWnd *pParentWnd: 是指该窗口的父窗口，顾名思义使用this是指当前窗口

UINT nID: 窗口ID，一般唯一标识一个窗口，一般为一个无符号整型数字

CCreateContext* pContext: 可以为当前指定内容，一般为文档模版，框架等等，很少使用

```
//MainFrame.cpp续 //准备创建时调用
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: 在此处通过修改
```

```

// CREATESTRUCT cs 来修改窗口类或样式，带有标题栏，最大框，最小框，系统菜单等等
cs.style = WS_OVERLAPPED | WS_CAPTION | FWS_ADDTOTITLE
           | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_SYSMENU ;
//取消有边界的客户边扩展风格
cs.dwExStyle &= ~WS_EX_CLIENTEDGE ;
//修改后，重新注册窗口，在注册时可以修改鼠标形状，图标等等，但对于框架窗口修改鼠标是
没有任何意义的，
cs.lpszClass = AfxRegisterWndClass(0) ;
return TRUE;
}
//MainFrame.cpp待续

```

函数：

```

LPCTSTR AFXAPI AfxRegisterWndClass( UINT nClassStyle, HCURSOR hCursor = 0,
HBRUSH hbrBackground = 0, HICON hIcon = 0 );

```

这个函数非常重要，在修改窗口样式时最经常使用，意思为注册你的窗口类，里面的参数意义如下：

参数：UINT nClassStyle: 定义窗口风格，置零代表默认值

HCURSOR hCursor : 指明使用的鼠标指针，默认为IDC_ARROW, 可以从文件中调用鼠标，也可以使用系统中自带的如IDC_WARNING等等

HBRUSH hbrBackground: 背景，可以指定当前窗口客户区背景，使用画刷填充

HICON hIcon: 指定一个图标，做为应用程序的标准图标，

注：如果所有皆为默认，则定义为：IDC_ARROW NULL_BRUSH 图标为WINDOWS波浪

```

//MainFrame.cpp续
// CMainFrame 诊断
#ifdef _DEBUG
//在调试时会使用到此
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}
#endif //_DEBUG
// CMainFrame 消息处理程序

```

//准备工作，在这里添加渲染的准备工作，实际上，在系统空闲时将进行下面操作，如果在系统空闲时不进行渲染，意思就是说只有在窗口需要重绘时才会渲染，那么画面将会不连贯，甚至停止，因为只有当窗口大小改变，窗口移动时才会重绘，OSG是在WM_PAINT中进行的操作

```


BOOL CMainFrame::PreToRen(void)
{
    //这里添加渲染的准备工作
    return 0 ;
}

```

```

void CMainFrame::OnSetFocus (CWnd * )
{
    //将焦点转移到前视图窗口
    m_wndCosgFreeViewer.SetFocus ( ) ;
}
BOOL CMainFrame::OnCmdMsg(UINT nID, int nCode, void *pExtra,
AFX_CMDHANDLERINFO* pHandlerInfo)
{
    //首先让VIEW处理执行消息，我们的渲染将主要在VIEW中利用菜单以及键盘进行控制
    if(m_wndCosgFreeViewer.OnCmdMsg(nID, nCode, pExtra, pHandlerInfo))
        return 1 ;
    //而后把消息送往默认处理，处理后仍向下传递
    return CFrameWnd::OnCmdMsg(nID, nCode, pExtra, pHandlerInfo) ;
} //MainFrame.cpp结束

```

在类视图中选中CosgFreeApp，然后点击右键，属性，选中（重写），在下面选中OnIdle，在右边选择添加OnIdle。也可手动添加，在程序空闲时将会执行该函数，OSG程序中将使用该函数进行渲染。重写OnIdle后，会在**osgFree.h**中多出以下内容：

```

osgFree.h

//by @FreeSouth at ZZU VR LAB for <OSG FreeS >

.....

class CosgFreeApp : public CWinApp
{
    .....
    // 重写
public:
    virtual BOOL InitInstance();

    // 实现
    afx_msg void OnAppAbout();
    DECLARE_MESSAGE_MAP()
    //此函数为重写的函数
    virtual BOOL OnIdle(LONG lCount) ;
};

extern CosgFreeApp theApp;

```

osgFree.cpp中会做比较大的改动，以适合我们的需求

```

osgFree.cpp

//by @FreeSouth at ZZU VR LAB for <OSG FreeS >

// osgFree.cpp : 定义应用程序的类行为。

```



```

//
#include "stdafx.h"
#include "osgFree.h"
#include "MainFrm.h"

//删除文档处理类
//#include "osgFreeDoc.h" //删除文档处理类
#include "osgFreeViewer.h" //添加自己的VIEW类

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CosgFreeApp
//删除打开新建命令,只保留关于对话框
BEGIN_MESSAGE_MAP(CosgFreeApp, CWinApp)
//关于对话框的菜单响应
ON_COMMAND(ID_APP_ABOUT, &CosgFreeApp::OnAppAbout)
// 新建与打开文档的菜单响应,我们要自己处理打开菜单,从这里删除,可以把CTRL+O (打开
//文档)类似的快捷键也一并删去
//ON_COMMAND(ID_FILE_NEW, &CWinApp::OnFileNew)
//ON_COMMAND(ID_FILE_OPEN, &CWinApp::OnFileOpen)
END_MESSAGE_MAP()
// CosgFreeApp 构造
CosgFreeApp::CosgFreeApp()
{
    // TODO: 在此处添加构造代码,
    // 将所有重要的初始化放置在InitInstance 中
}
// 唯一的一个CosgFreeApp 对象

CosgFreeApp theApp;

// CosgFreeApp 初始化
BOOL CosgFreeApp::InitInstance()
{
    // 如果一个运行在Windows XP 上的应用程序清单指定要
    // 使用ComCtl32.dll 版本6 或更高版本来启用可视化方式,
    //则需要InitCommonControlsEx()。否则,将无法创建窗口。
InitCommonControls() ;

CWinApp::InitInstance () ;

    /*程序中将不使用公共控件类,下面的粗体将被注释掉

```

```

INITCOMMONCONTROLSEX InitCtrls;
InitCtrls.dwSize = sizeof(InitCtrls);
// 将它设置为包括所有要在应用程序中使用的
// 公共控件类。
InitCtrls.dwICC = ICC_WIN95_CLASSES;

InitCommonControlsEx(&InitCtrls);
*/

// 初始化OLE 库
if (!AfxOleInit())
{
    AfxMessageBox(IDP_OLE_INIT_FAILED);
    return FALSE;
}
AfxEnableControlContainer();
// 标准初始化
// 如果未使用这些功能并希望减小
// 最终可执行文件的大小，则应移除下列
// 不需要的特定初始化例程
// 更改用于存储设置的注册表项
// TODO: 应适当修改该字符串，
// 例如修改为公司或组织名
SetRegistryKey(_T("应用程序向导生成的本地应用程序"));

//不需要文档模版,自己处理视窗口//下面的粗体都将被注释掉
/*LoadStdProfileSettings(4); // 加载标准INI 文件选项 (包括MRU)
// 注册应用程序的文档模板。文档模板
// 将用作文档、框架窗口和视图之间的连接
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CosgFreeDoc),
    RUNTIME_CLASS(CMainFrame), // 主SDI 框架窗口
    RUNTIME_CLASS(CosgFreeView));
if (!pDocTemplate)
    return FALSE;
AddDocTemplate(pDocTemplate);

// 启用"DDE 执行"
EnableShellOpen();
RegisterShellFileTypes(TRUE);

```

```

// 分析标准外壳命令、DDE、打开文件操作的命令行
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// 调度在命令行中指定的命令。如果

// 用 /RegServer、/Register、/Unregserver 或 /Unregister 启动应用程序，则返回
FALSE。
if (!ProcessShellCommand(cmdInfo))
    return FALSE;
*/
//上面的都被注释掉了

// 唯一的一个窗口已初始化，因此显示它并对其进行更新

//若要创建主窗口,此代码将创建新的框架窗口对象,然后将其设置为应用程序主窗口对象
CMainFrame* pFrame = new CMainFrame ;
if(!pFrame)
    return FALSE ;

m_pMainWnd = pFrame ;

//创建新的框架窗口，取菜单，定义框架类别等等
pFrame ->LoadFrame(IDR_MAINFRAME, WS_OVERLAPPEDWINDOW | FWS_ADDTOTITLE,
NULL, NULL) ;

// 唯一的一个窗口已初始化，因此显示它并对其进行更新
m_pMainWnd->ShowWindow(SW_SHOW);
m_pMainWnd->UpdateWindow();
// 仅当具有后缀时才调用DragAcceptFiles
// 在SDI 应用程序中，这应在ProcessShellCommand 之后发生
// 启用拖/放
//m_pMainWnd->DragAcceptFiles();

return TRUE;
}
// 用于应用程序“关于”菜单项的CAboutDlg 对话框

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

```

```

// 对话框数据
enum { IDD = IDD_ABOUTBOX };
protected:
virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV 支持

// 实现
protected:
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAaboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAaboutDlg, CDialog)
END_MESSAGE_MAP()

// 用于运行对话框的应用程序命令
void CosgFreeApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

// CosgFreeApp 消息处理程序，空闲时将调用此函数
BOOL CosgFreeApp::OnIdle (LONG lCount)
{
    BOOL continue_processing = CWinApp::OnIdle(lCount) ;
    //把当前窗口转到渲染框架窗口，从而在空闲时间进行渲染
    CMainFrame* pFrame = dynamic_cast<CMainFrame*>(m_pMainWnd);

    //在空闲时将做渲染准备
    if (pFrame)
    {
        if (pFrame->PreToRen())
            return TRUE;
    }
    return continue_processing;
}
//osgFree.cpp完

```

osgFreeViewer.h中代码完全为新建的，对OSG的主要控制将发生在这里。需要响应以下二个消息：WM_PAINT,WM_SIZE,其中添加OnFileOpen是对菜单进行的操作，首先选资源视图，选择Menu,双击IDR_MAINFRAME后，在文件根菜单下，找到打开子菜单，对其点右键->添加事件处理程序，在类列表中选择CosgViewer，在左边的消息类型中选择COMMAND，如果选择UPDATE_COMMAND_UI表示要对菜单进行什么操作，比如禁用它，在它前面画个小钩钩，把它改成三态的，等等！所有东西都可以手动添加！

```
osgFreeViewer.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#pragma once
//#include "afxwin.h"//注释掉此头文件
//从窗口类继承
class CosgFreeViewer :
public CWnd
{
public:
    CosgFreeViewer(void) ;
public:
    virtual ~CosgFreeViewer(void) ;

public :
    //需要重写此函数，关于如何添加重写的函数，在前面说过
    virtual BOOL PreCreateWindow(CREATESTRUCT & cs) ;

public :
    DECLARE_MESSAGE_MAP()
    //所有消息映射
public :
    afx_msg void OnPaint() ;
    afx_msg void OnSize(UINT nType, int cx, int cy) ;
    afx_msg void OnFileOpen() ;

};//osgFreeViewer.h结束
```

osgFreeViewer.cpp中将对OSG进行主要的控制，是最重要的文件之一，以后主要的修改工作将在这里完成，除了音乐与全屏的控制主框架(CMainFrame)中以外，其余的操作基本上都在这里！

```
osgFreeViewer.cpp
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#include "StdAfx.h"
//添加需要的头文件
#include "osgFree.h"
#include "osgFreeViewer.h"

//添加消息映射序列
```

```

BEGIN_MESSAGE_MAP(CosgFreeViewer, CWnd)
    ON_WM_PAINT()
    ON_WM_SIZE()
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
END_MESSAGE_MAP()

//构造
CosgFreeViewer::CosgFreeViewer(void)
{
}

//析构
CosgFreeViewer::~~CosgFreeViewer(void)
{
}

//准备创建
BOOL CosgFreeViewer::PreCreateWindow(CREATESTRUCT & cs)
{
    //在这里可以修改CS，在前面已经介绍过
    if (!CWnd::PreCreateWindow(cs))

        return FALSE ;
    return TRUE ;
}

//CosgFreeViewer消息处理函数
void CosgFreeViewer::OnPaint()
{
    CPaintDC dc(this) ;
    //消息处理代码
    //下面的代码暂时使用,将当前窗口刷新成蓝色,当使用osg处理osgFreeViewer的Client时,
    将不会使用下面代码
    CRect rect ;
    CBrush brush(RGB(0, 0, 100)) ;
    //这是全局函数，意思是得到当前客户区矩阵，第一个参数是句柄，在MFC中，会有一个默认的成员
    变量m_hWnd保存的是句柄，也可以通过this->GetSafeHwnd()进行获得
    ::GetClientRect (m_hWnd,&rect) ;
    //运用画刷对客户区进行填充
    dc.FillRect (&rect, &brush) ;
}

//调整大小
void CosgFreeViewer::OnSize (UINT nType, int cx, int cy)
{
    CWnd::OnSize (nType, cx, cy) ;
}

```

```

//消息处理代码
}
//打开文件
void CosgFreeViewer::OnFileOpen()
{
    //命令处理代码
    CFileDialog fileDlg(TRUE) ;
    //创建打开文件对话框
    if(fileDlg.DoModal () == IDOK)
    {
        //打开处理代码
    }
}
}

```

第五步：运行。这时按CTRL+F5看看是不是可以运行了，如果有错，看看是什么错误，在光盘中有完整的可以运行的代码，必须有vs2005的环境，在vs2003下按原码敲进去也可以编译通过。

运行结果：

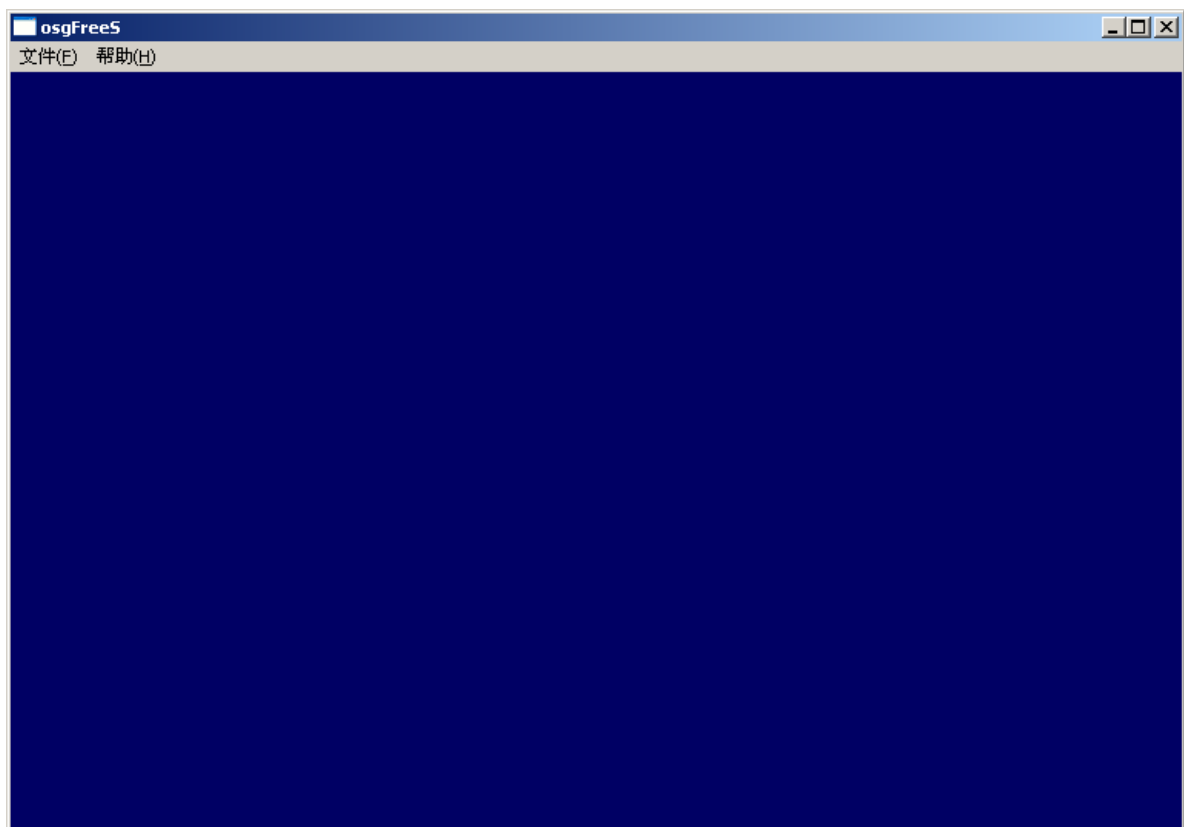


图 1.5

调试的基本操作是开始时点击F5，之后可以查出程序是错在哪一行，如果内存泄漏，向量越界等问题可以直接定位到出问题的函数。之后设置断点F9，F10，F5加上对变量进行监视，步进跟踪，调试程序是一件痛苦而又麻烦的事情，这就要求在写程序时不要粗之大叶，类似于指针必须初始化，向量，的.end()

指的是最后一个内容的后边等等，很多很多，不要小看这些小问题，代码已愈万行之后，一个很小的问题可能要花成星期的时间来发现。

MFC只是一个框架的应用，在以后的章节中我们就假设置作者已经可以熟练的使用MFC,或者做者可以把功能做到其它的框架中去。

下面来理清一下这个应用程序的执行过程：

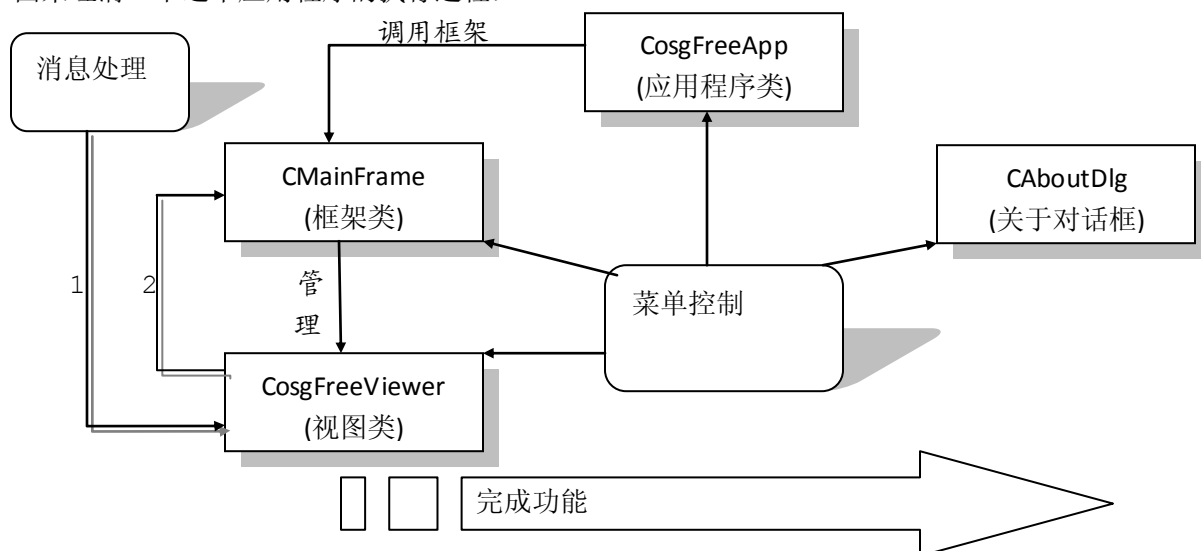


图 1.6

MFC程序通过人为控制可以分为二种，一为控制菜单，二来通过键盘或是鼠标发送消息，或者通过菜单以定时器发送消息，消息处理过程总是先由视图类先行处理，而后再由应用程序框架进行处理，而菜单则可以控制现有的所有的类，在多个类中同时响应则也会按照先视图后框架的顺序，且有些消息框架根本接收不到。

用户要完成的功能往往通过视图类加以控制，一般的功能都是通过视图类来交互完成！我们即将学习的MFC+OSG也是主要由视图类进行管理！

1.2 MFC 常用的几种基本操作

使用 MFC 的技巧非常非常多，下面来介绍非常常见的四个功能，修改鼠标，制作半透明窗口，视口全屏以及添加背景音乐。

1.2.1 修改鼠标

以上一节创建的MFC程序框架为基础，修改鼠标分为两种，一种为修改成为系统自带鼠标，比如在收集点时把鼠标改为十字(cross)，在执行非法操作时鼠标改为禁止，遇忙时等待。另一种最常见的为加载.ANI文件，两种操作异曲同工，需要注意的是需要加载.ANI文件时，需要先把该文件放在res目录或是项目根目录中！

在osgFreeViewer.cpp的PreCreateWindow函数中做如下修改：

osgFreeViewer.cpp

.....

```
BOOL CosgFreeViewer::PreCreateWindow(CREATESTRUCT & cs)
```

```
{
```

```
    if (!CWnd::PreCreateWindow (cs))
```

```
        return FALSE ;
```

//在使用OSG进行渲染时将取消背景填充色，通过下面的代码重新注册窗口类可以改变当前的鼠标与背景色和窗口风格，还有一种比较麻烦的方法是重新注册WNDCLASS结构体。为了保证是宽字符串，需要在"res/StarCraftArrow.ani"前加上L，或者是TEXT都可以，vs8几乎取消了对窄字符的支持！GetStockObject函数，可以返回一个画刷，字体或PEN，具体参数如下：

```
HGDIOBJ GetStockObject( int fnObject);
```

```
/*BLACK_BRUSH 黑色画刷，值为4
```

```
DKGRAY_BRUSH 暗灰色画刷
```

```
GRAY_BRUSH 灰色画刷
```

```
HOLLOW_BRUSH 空画刷，显示为背景
```

```
LTGRAY_BRUSH 亮灰色画刷
```

```
NULL_BRUSH 空画刷，显示为背景
```

```
WHITE_BRUSH 白色画刷
```

```
BLACK_PEN 黑色Pen
```

```
WHITE_PEN 白色pen.
```

```
NULL_PEN 空画笔
```

```
SYSTEM_FONT 得到系统字体，默认情况下中国就是宋体，是在程序中菜单上，对话框等上出现的字体
```

```
DEFAULT_PALETTE 得到系统调色板
```

```
*/
```

```
cs.lpszClass = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW,
```

```
LoadCursorFromFile(L"res/StarCraftArrow.ani"), (HBRUSH) GetStockObject(4)) ;
```

```
    return TRUE ;
```

```
}
```

.....

函数：

```
HCURSOR LoadCursorFromFile(LPCTSTR lpFileName);
```

此函数可以从文件中读入一个鼠标，返回HCURSOR。失败时返回NULL。失败的情况一般为.ANI文件不存在或是做过非法修改！

```
HCURSOR LoadCursor( HINSTANCE hInstance, LPCTSTR lpCursorName);
```

此函数也可以用做取鼠标，不同的是它取的是系统默认的鼠标，第一个参数一般设置为NULL，经常使用的语句是：SetCursor(LoadCursor(NULL, IDC_WAIT));其中第二个参数IDC_WAIT可以是IDC_ARROW：箭头，IDC_UPARROW向上箭头，IDC_CROSS十字架等等！

1.2.2 透明对话框

在OSG中，在场景中可能会经常用到无模式的对话框，以便显示建筑信息等等。这里就需要用到透明对话框的制做。在这里首先介绍一下物体透明的原理。

各种设计透明物体的方法在经过实践的选择以后，alpha混合技术成为主导。它的原理是如果一个物体需要对它后面的物体透明，这就需要将透明物的颜色和其后面物体的颜色进行混合。与每个像素相关联的有一个RGB颜色值和一个Z缓冲器深度值，另外一个成份就是alpha值，可以根据需要生成并存储，可以使用alpha描述给定像素处物体的透明程度，alpha为1表示不透明，为0表示全透明。

通常有些文件格式为RGBA，其中最后一个A就代表alpha通道，它的意义为：如果C1代表遮挡物体颜色，C2代表被挡物体颜色，则挡住区域最终的颜色值为：

$$C = \alpha * C1 + (1 - \alpha) * C2$$

在VS中为我们提供了一个函数，专门用来计算层叠窗口的透明度与不透明度！

下面来看一下这个函数：

函数：

```
BOOL SetLayeredWindowAttributes( HWND hwnd, COLORREF crKey,
    BYTE bAlpha, DWORD dwFlags);
```

参数：HWND hwnd :指的是需要透明的窗口的句柄，需要指明的是该窗口必须具有WS_EX_LAYERED的扩展风格才能具备透明度，可以使用SetWindowLong来对扩展风格进行处理，这是个极常用的函数，等一会儿详细介绍。

COLORREF crKey:顾名思义，这里指的是要透明的颜色，使用的是RGB宏（0~255）

BYTE bAlpha:指的是透明度。0指全透明，255指不透明。

DWORD dwFlags: 指定一种行为，可以是以下两值：

LWA_COLORKEY 把crKey 当作透明色

LWA_ALPHA 使用bAlpha来指定不透明度

函数：

```
LONG SetWindowLong( HWND hWnd, int nIndex, LONG dwNewLong);
```

功能：设置窗口风格

参数：HWND hWnd: 一般指要修改的窗口的句柄

Int nIndex:要设置的风格类型，一般取下列值之一：

GWL_EXSTYLE : 设置扩展风格

GWL_STYLE : 设置新的风格

GWL_ID : 给窗口设置一个新的ID

GWL_USERDATA :给窗口重新定义一个32位长的数字，一般窗口都有一个唯一32位长的数字来描述其各种属性

LONG dwNewLong: 给窗口设置新值（风格），一般是通过GetWindowLong取得同样值后与需要的值取位与运算，或是位且取反运算来去掉该值！GetWindowLong只有前两个参数，返回值为风格（LONG）

下面我们来利用这两个函数把上个例子中的关于对话框做成透明的：

示例二：透明对话框的制做

第一步：在CAboutDlg类上右键，属性，在属性面板的重写按钮下拉菜单中选中添加OnInitDialog()

第二步：在osgFree.cpp中对重写的OnInitDialog进行填充，首先要弄明白的是OnInitDialog()是在窗口初始化时调用的函数，所以在窗口未产生之前就会调用该函数。

osgFree.cpp

```
.....  
BOOL CAboutDlg::OnInitDialog()  
{  
    //建立透明的关于对话框  
    CDialog::OnInitDialog();  
  
    //在窗口样式中添加WS_EX_LAYERED  
    SetWindowLong(this->GetSafeHwnd(),GWL_EXSTYLE,GetWindowLong(this->GetSafeHwnd(),GWL_EXSTYLE)^WS_EX_LAYERED) ;  
    //需要装入User32.DLL  
    HINSTANCE hInst = LoadLibrary(L"User32.DLL");  
  
    if(hInst)  
    {  
        typedef BOOL (WINAPI *MYFUNC)(HWND,COLORREF,BYTE,DWORD) ;  
        MYFUNC fun = NULL;  
        //取得SetLayeredWindowAttributes函数指针  
        CHAR str[] = "SetLayeredWindowAttributes" ;  
  
        fun=(MYFUNC)GetProcAddress(hInst, str) ;  
        if(fun)  
            fun(this->GetSafeHwnd(),0,128,2);  
        FreeLibrary(hInst) ;  
    }  
    return TRUE; // return TRUE unless you set the focus to a control  
    // 异常: OCX 属性页应返回FALSE  
}
```

.....

运行效果:

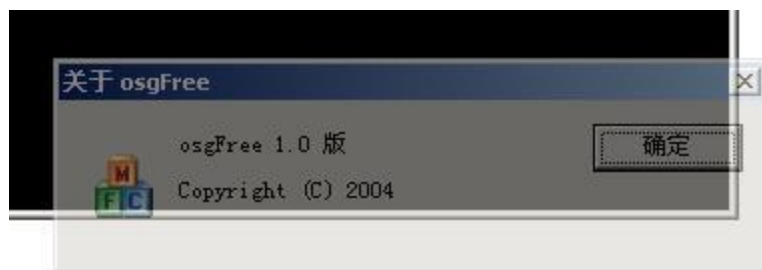


图1.7

1.2.3 客户区全屏

一般来讲对客户区全屏需要以下几个步骤：去掉边框，系统菜单，最大化最小化按钮，菜单（可去可不去），把客户区充斥到整个屏幕。这样就达到了目的。

通过这个例子我们来深入理解一下 MFC 框架。

下面我们在上一个程序的基础上来通过菜单响应来控制建立一个全屏的程序：

示例三：客户区全屏

第一步：定位到资源视图的菜单中，在原菜单的最后一项上添加视图根菜单，全屏子菜单，如下图所示：

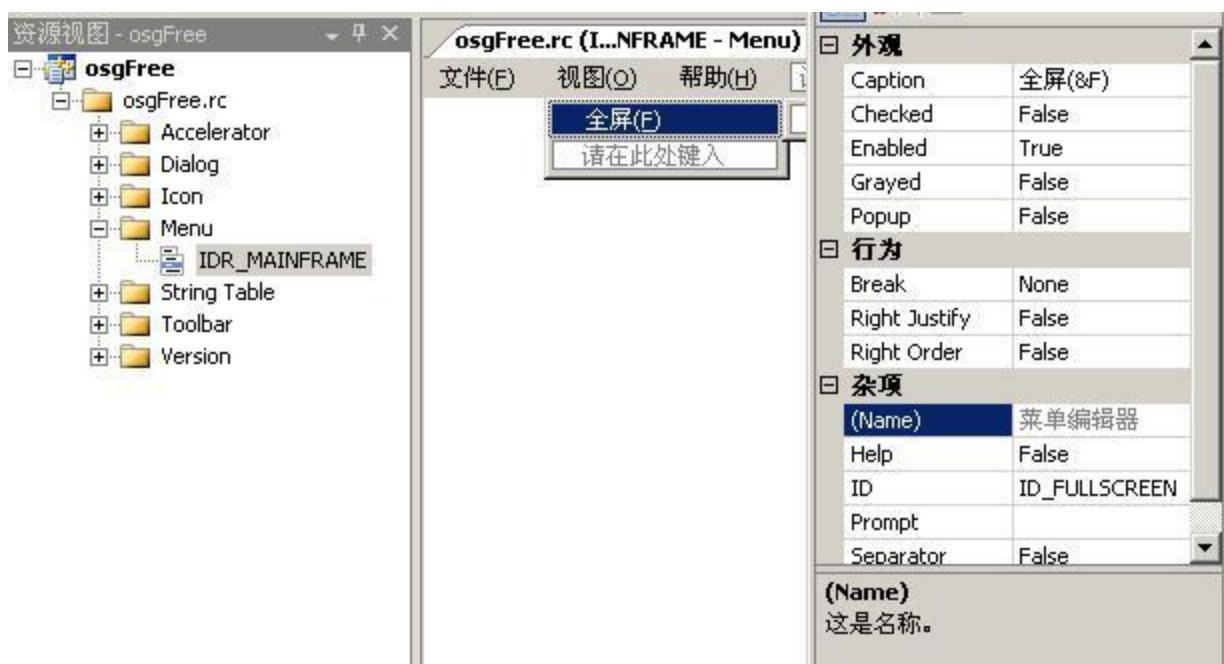


图 1.8

第二步：对全屏菜单右击，选择添加事件处理程序，类列表选择 CMainFrame。选择添加编辑：

此时要想使用菜单控制全屏与非全屏，必须得知道当前程序是否处于全屏之中，因此需要在 CMainFrame 中添加一变量 `m_bFullScreen`，添加变量有多种方法，最简单的是用手直接添，用手添的坏处是容易忘记初始化，好处是灵活性大，想怎么添怎么添，想添哪添哪，一般根据个人习惯不同，我习惯人工处理。而后再添加一个为 `void FullScreen(bool)` 的函数，添加函数也可以通过向导，但是手工添感觉更好。当你对程序的整个条理比较清晰的时候，无论怎样操作都不会出错，或是不会出什么大错。

现在来看文件相应的变化：

在 MainFrm.h 与 MainFrame.cpp 中改动如下：

```
MainFrm.h
.....
class CMainFrame : public CFrameWnd
{
.....
```

```

//控制全屏幕显示
BOOL m_bFullScreen ;
//全屏功能实现
void FullScreen(bool ) ;
public:
    afx_msg void OnFullscreen();
};
.....
MainFrame.cpp
.....
CMainFrame::CMainFrame()
{
    // TODO: 在此添加成员初始化代码
    m_bFullScreen = true ;
}
.....
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....
    FullScreen(TRUE) ;
    return 0 ;
}
.....
//菜单响应
void CMainFrame::OnFullscreen()
{
    //调用功能，如果在全屏就不全屏，否则就全屏
    FullScreen(!m_bFullScreen) ;
    //设置标识
    m_bFullScreen = !m_bFullScreen ;
}
//控制全屏的函数
void CMainFrame::FullScreen (bool bFullScreen)
{
    //得到当前STYLE
    LONG style = ::GetWindowLong(this->m_hWnd,GWL_STYLE);
    //得到当前WND
    CWnd *pWnd=AfxGetMainWnd();

    if(bFullScreen)//全屏显示
    {
        pWnd->DrawMenuBar();
        //去掉边框
        style &= ~(WS_DLGMFRAME | WS_THICKFRAME);
    }
}

```

```

SetWindowLong(this->m_hWnd, GWL_STYLE, style);
//最大化视窗
this->ShowWindow(SW_SHOWMAXIMIZED);
CRect rect;
//得到当前视窗口并稍做移动,保证彻底全屏,可能有的机器不同,会保留任务栏,大多数都会全屏

this->GetWindowRect(&rect);
::SetWindowPos(this->m_hWnd, HWND_TOPMOST, rect.left, rect.top,
rect.right-rect.left + 3, rect.bottom-rect.top + 3, SWP_FRAMECHANGED);
}
else//窗口显示
{
    //得到菜单
    CMenu menu;
//取菜单
    menu.LoadMenu(IDR_MAINFRAME);
    //设置菜单
pWnd->SetMenu(&menu);
    pWnd->DrawMenuBar();
    menu.Detach();
    style |= WS_DLGFRAE | WS_THICKFRAME;
    SetWindowLong(this->m_hWnd, GWL_STYLE, style);
    this->ShowWindow(SW_SHOWNORMAL);
} ;
}
//MainFrame.cpp完

```

1.2.4 音乐播放

关于音乐在 WINDOWS 中的控制就可以独自成书, 立体声在三维场景中的控制也可以独自成书, 无论 OPENAL 还是 OSGAL, 人们总很难把立体声在三维场景中的应用与场景的渲染拿到同一个天平上来, 因此很多时候需要用简单的音乐播放来替代复杂的立体声效果。

音乐播放最好放在 CMainFrame 中, 因为菜单频繁的控制 VIEWER 类, 有几率可能会使音乐中断且在框类中调用在程序开始创建时便开始播放音乐, 并不是在打开模型后或是在打开模型时, 在换场景等等对 VIEWER 类的操作中, 不会使音乐受到任保影响, 有时候需要重新加载 VIEWER 类来释放内存, 那么音乐将会中断, 而在 CMainFrame 中将不会出现这种情况!

下面来介绍本节中使用的“发声函数”, 在 MCI 中, 往往会创立一个窗口, 我们不需要此窗口, 故而需要把窗口来隐藏掉:

函数:

```

HWND MCIWndCreate( HWND hwndParent,    HINSTANCE hInstance,
DWORD dwStyle, LPSTR szFile);

```

功能：该函数创建MCI窗口，(MCI:Media Control Interface)。

参数：**hwndParent：**父窗口句柄，将被挂名在此窗口下

hInstance：创建一个实例与MCIWindow相关！

dwStyle：指定风格，类似与CreateWindowEx的功能，也可以用之实现，该标识一般取下列值：

MCIWNDF_NOAUTOSIZEWINDOW MCI：具有此风格窗口的大小将不会改变。

MCIWNDF_NOAUTOSIZEMOVIE：同上。

MCIWNDF_NOERRORDLG：继承ERROR对话框给USER

MCIWNDF_NOMENU：隐藏菜单，一般丢失的菜单上的选项可以在系统菜单上找到

MCIWNDF_NOOPEN：在面板上隐藏打开菜单，将从系统菜单上增加打开选项

MCIWNDF_NOPLAYBAR：隐藏工具栏

MCIWNDF_NOTIFYANSI：在向父窗口发送设置参数修改消息时，让MCIWnd使用ANSI字符替换Unicode字符只能与MCIWNDF_NOTIFYMODE 配合使用在NT及2000以上的系统之上。

MCIWNDF_NOTIFYMODE 向系统发送MCIWNDF_NOTIFYMODE 消息，无论当前在进行什么操作，且在lparam中指定新值如MCI_MODE_STOP。

MCIWNDF_NOTIFYPOS 当对播放器指定回放操作时将会发生此消息且记下当前回放位置

MCIWNDF_NOTIFYMEDIA当打开文件时会发送该消息，lparam中存有新文件之路径串

MCIWNDF_NOTIFYSIZE 当窗口大小改变时会发送此消息

MCIWNDF_NOTIFYERROR 向主窗口发送错误消息

MCIWNDF_NOTIFYALL 使所有的NOTIFY风格被使用

MCIWNDF_RECORD 在面板上增加Record键以完成其功能

MCIWNDF_SHOWALL 使所有的MCIWNDF_SHOW 风格被使用

MCIWNDF_SHOWMODE 当前配置模式被显示在WINDOWS标题栏中，当前模式可以通过MCIWndGetMode宏得到

MCIWNDF_SHOWNAME 在MCIWindow的标题栏中显示当前文件名

MCIWNDF_SHOWPOS 在MCIWindow的标题栏中显示当前播放进度

LPSTR szFile：指定要播放的文件名字串，为平凡字串（意思为非宽字符）

函数：

BOOL ShowWindow(HWND hWnd, int nCmdShow)；

功能：隐藏或显示指定窗口

参数：**HWND hWnd：**指定要隐藏或要显示的窗口

Int nCmdShow：参数一般习惯为取TRUE|true|1为显示，取FALSE|false|0为隐藏，而实际上01只是它们的前两个值，一般它可以取下列值：

SW_HIDE：隐藏窗口，同时激活其它窗口

SW_SHOW：显示窗口且激活其在当前位置

SW_SHOWNA：显示窗口正常如果在激活仍旧在激活，一般不使用这个

SW_SHOWNORMAL：正常显示窗口，如果在最小化之中将会被负以焦点从而以正常尺寸在当前位置显示，这个一般在第一次创建窗口时使用，

关于MCI的更多内容请查阅MSDN：

下面我们来播放音乐，在上一章例子的基础上做一些改动：

示例三：播放音乐。

第一步：把要播放的音乐（建议使用.MP3，.WAV 过大）拷至文件根目录下，否则需要在播放文件名前加路径，在调试程序编译运行时的根目录指的是放有很多.CPP,.H的这个当前目录，而独立运行指的是

可执行程序所在目录。

第二步：点击菜单：项目→osgFree 属性页，在弹出的对话框中，配置项默认为对当前活动 Debug 进行配置，如果不调试，而发布 Release 则需要对 Release 进行配置，一般来讲 Debug 版本中包含更多的信息，但是 Release 适合发布，一般文件比较小。

点击下面列表中的链接器，按如图配置：

在附加选项中添加 vfw32.lib 这是链接文件，在 OSG 中最经常使用与制做的功能模块便是 WIN32 DLL，更加方便的复用，ActiveX 控件适合做成品的功能模块，而 DLL 更适合某些具体的功能模块，如某 DLL 可到处实现对任何 OSG 程序任何场景的漫游，只需要添加不多的几行代码，而 ActiveX 更适合已经做成的小漫游场景用在 VS 的其它类型的程序框中，最经常见到的是 VB 与 WEB。

下面的配置也可以使用另一种方法，可以在 osgFree 属性页（如下图）的链接器→输入→右边面板的附加依赖项中添加 vfw32.lib 这样的话所加内容将会出现在如图示的右边所有选项的灰色框中，读者可自己下去试一下。

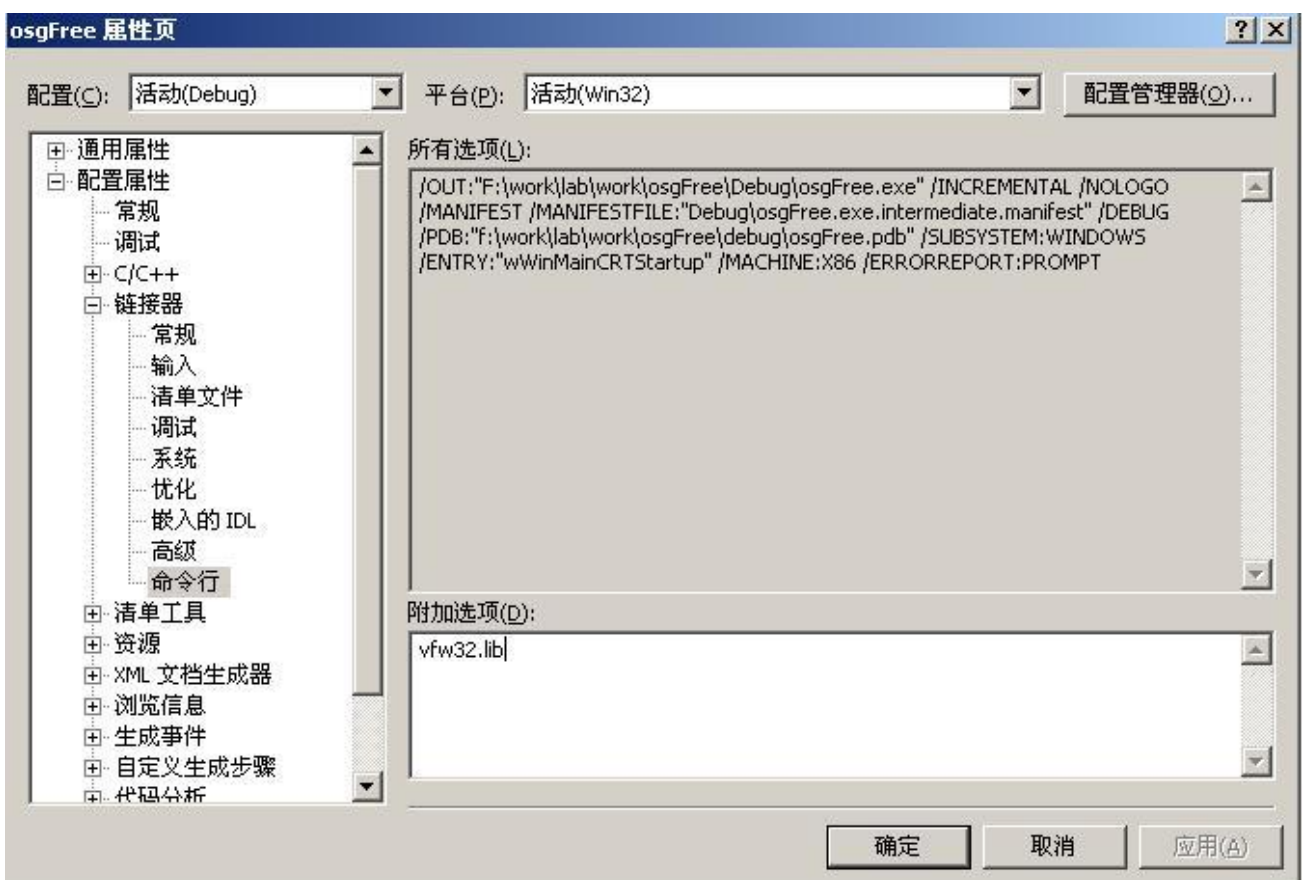


图 1.9

也可以通过语句添加，在 stdafx.h 中加入如下语句#pragma comment(lib, "vfw32.lib") 便可动态的加入 vfw32.lib 文件。

第三步：菜单→项目→添加类，取类名为 PSound，无基类，别的选项不用动，这时在当前项中会多出两个文件：PSound.h PSound.cpp，主要通过修改这两个文件来完成功能。

PSound.h 的内容如下：

```
PSound.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#pragma once
```



```

//包含必要的头文件,
#include<windows.h>
#include<vfw.h>
//#include<stdio.h>
class PSound
{
public:
    PSound(void) ;
public:
    ~PSound(void) ;

    //播放音乐
    void playsound() ;
    //停止音乐
    void stopsound() ;

private:
    //父窗口句柄
    HWND hnd;

public:
    //创建播放器窗口
    void CreateMusic (HWND hWnd) ;
}; //PSound.h结束

```

下面来看 PSound.cpp 文件

```

PSound.cpp
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#include "StdAfx.h"
#include "PSound.h"
PSound::PSound()
{
}

void PSound::playsound()
{
    //打开PLAY的宏,可参照vfw.h
    MCIWndPlayFrom(hnd, 0);
}
void PSound::stopsound()
{
    //停止播放的宏,可参照vfw.h
    MCIWndStop(hnd);
}

```

```

void PSound::CreateMusic(HWND hWnd)
{
    //创建播放窗口
    hnd=MCIWndCreate(hWnd,NULL,MCIWNDF_NOPLAYBAR|MCIWNDF_SHOWALL,TEXT("play.m
p3"));
    //隐藏窗口
    ::ShowWindow (hnd, FALSE) ;
} //PSound.cpp结束

```

第三步：在 CMainFrame 中添加播放音乐的代码。

首先在 MainFrm.h 中加入头文件与相关变量。

如下面改动：

MainFrm.h

```

//只编译一次即可,相当于以前的#ifdef #define #endif对
#pragma once
//包含自己创建的视窗口头文件
#include "osgFreeViewer.h"
#include "PSound.h"
class CMainFrame : public CFrameWnd
{
    .....
public:
    PSound *play ;
    .....
};
//MainFrm.h结束

```

在 MainFrm.cpp 中加入播放代码

MainFrm.cpp

```

.....
CMainFrame::CMainFrame()
{
    // TODO: 在此添加成员初始化代码
    m_bFullScreen = true ;
    play = new PSound();
}
.....
//创建主窗口消息映射函数
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....

    play ->CreateMusic (m_hWnd) ;
}

```

```
    play ->playsound () ;  
    return 0 ;  
}  
.....  
//MainFrm.cpp结束
```

编译运行一下看看是否有音乐。

通过本章的学习可以掌握以下内容：在 MFC 中使用菜单控制程序完成功能，添加/删除类、变量、函数。

到目前为止大略的介绍了 MFC 的相关的基本操作与经常使用的四个技术：改变鼠标，透明窗口制作，全屏，添加音乐。至于使用什么框架，对于一个好手来说都是不成问题的，对初学者来说 MFC 框架更具有清晰性。下一章我们将进入 OSG 世界进行学习，将会使用 MFC+OSG 框架与 WIN32 控制台框架，最后会介绍 WINDOWS 框架，使用什么框架对于 OSG 来说关系都是不大的，关键在于理解 OSG 的基本操作与渲染流程。下一章让我们来共同学习 OpenSceneGraph 的基本操作。

第二章 基本几何体的绘制

绘制基本的几何体在 OSG 的应用中并不占有主导地位,本章的目的是通过绘制基本的几何体来熟悉 OSG 的基本的操作与渲染流程。熟练掌握 OSG+MFC 与 OSG+WIN32 控制台程序的应用。

本章将介绍内容: OSG 的渲染过程, OSG+MFC 与结合渲染, OSG+WIN32 控制台程序的基本构建, OSG 中基本的几何体的绘制, OSG 中的矩阵操作。osgviewer 与 osgconv 的使用, 窄字符与宽字符间的来回转换。

本章将做以下示例: 构建 OSG+MFC 的最基本框架, 编写绘制基本图元的程序, 移动与旋转图元。

2.1 构建 OSG+MFC 框架

在上一章我们已经成功的建造一个 MFC 精简的单文档程序, 我们将在此基础上对其进行改造从而适合用户 OSG 的控制。

2.1.1 OSG 渲染过程

首先来对 OSG 进行一下简单的理解, 主要是渲染过程的理解。

首先我们来看一下 OSG 的渲染过程, 如下图:

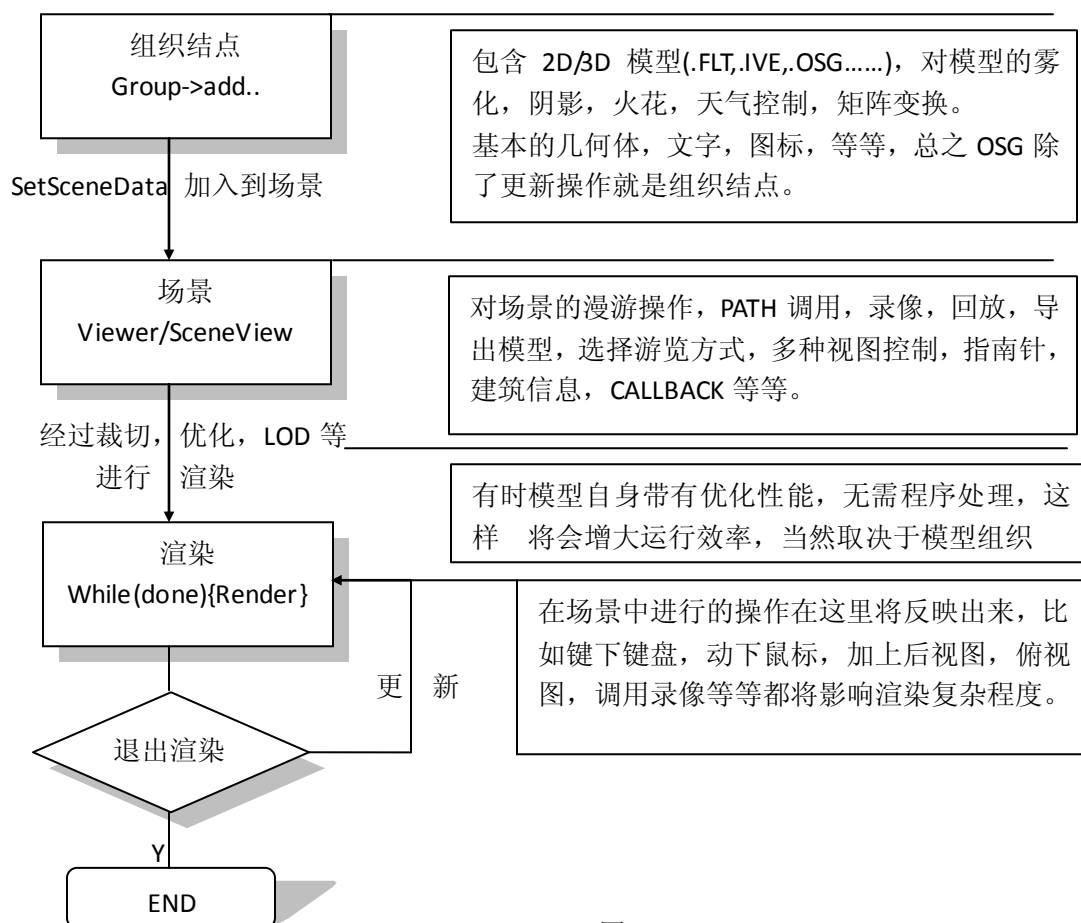


图 2.1

我们来举一个例子程序来理解 OSG 是如何实现以上过程, 且为我们以后的学习做准备, 在这里我们要

使用 OSG 的自带例子。

示例四：最简单的 OSG 程序

第一步：新建一个 WIN32 控制台项目。取名为 osgFree，点击确定再点击下一步，选中空项目复选框，在这里我们要建立一个无头文件，无源文件的 WIN32 控制台项目。

第二步：修改 VS 使其可以找到 OSG 头文件以及库文件，点击菜单工具→选项→项目和解决方案→VC++ 目录，在右边的包含文件与库文件中分别添加 OSG 安装的 include 与 lib 路径，注意一定要选择 include 根目录，例如你的 OSG1.2 装在 C:\Program files 下，那么路径就定位在 C:\Program files\OSG1.2\include 下，不要选在 C:\Program files\OSG1.2\include\osg 下，这样会找不到头文件。

经过第二步的修改以后，在以后的 OSG 程序设计中就不需要再对 VS 的框架做什么设置了，这一步是永久性的修改。

在以前没有 OSG 安装包的时候，往往还需要配置系统 PATH，系统的 PATH 是指应用程序在执行时可以自动搜索这些目录，以例找到所依赖的库或文件。PATH 可以通过 CMD 中的命令 PATH 查看，也可以通过右键我的电脑→属性→高级→环境变量中的 PATH 进行配置。

第三步：在类视图中选中该项目，选中项目属性，在链接器：命令行中加入 lib 文件，在 OSG 中需要加入的 lib 文件一般有以下几个：OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib osgUtild.lib，如果需要 Release 版本，只需要把 d 去掉就可以了，其实在所有配置中把所有库都配置成不带 d 的版本也是可以的，不过在调试中可能会出现一些问题。

关于增加 lib 的三种方法在上一章详细介绍过。

第四步：在源文件中击右键，添加新项，选择 CPP 源文件而后输入文件名，这样就可以开始编程了。

在 Main.cpp 中键入以下代码：

```
Main.cpp
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>

int main()
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();

    osg::ref_ptr <osg::Node> node = osgDB::readNodeFile("glider.osg");
    viewer->setSceneData(node.get ());
    viewer->realize();

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

    viewer->sync();
```

```
return 0;  
}
```

运行效果:

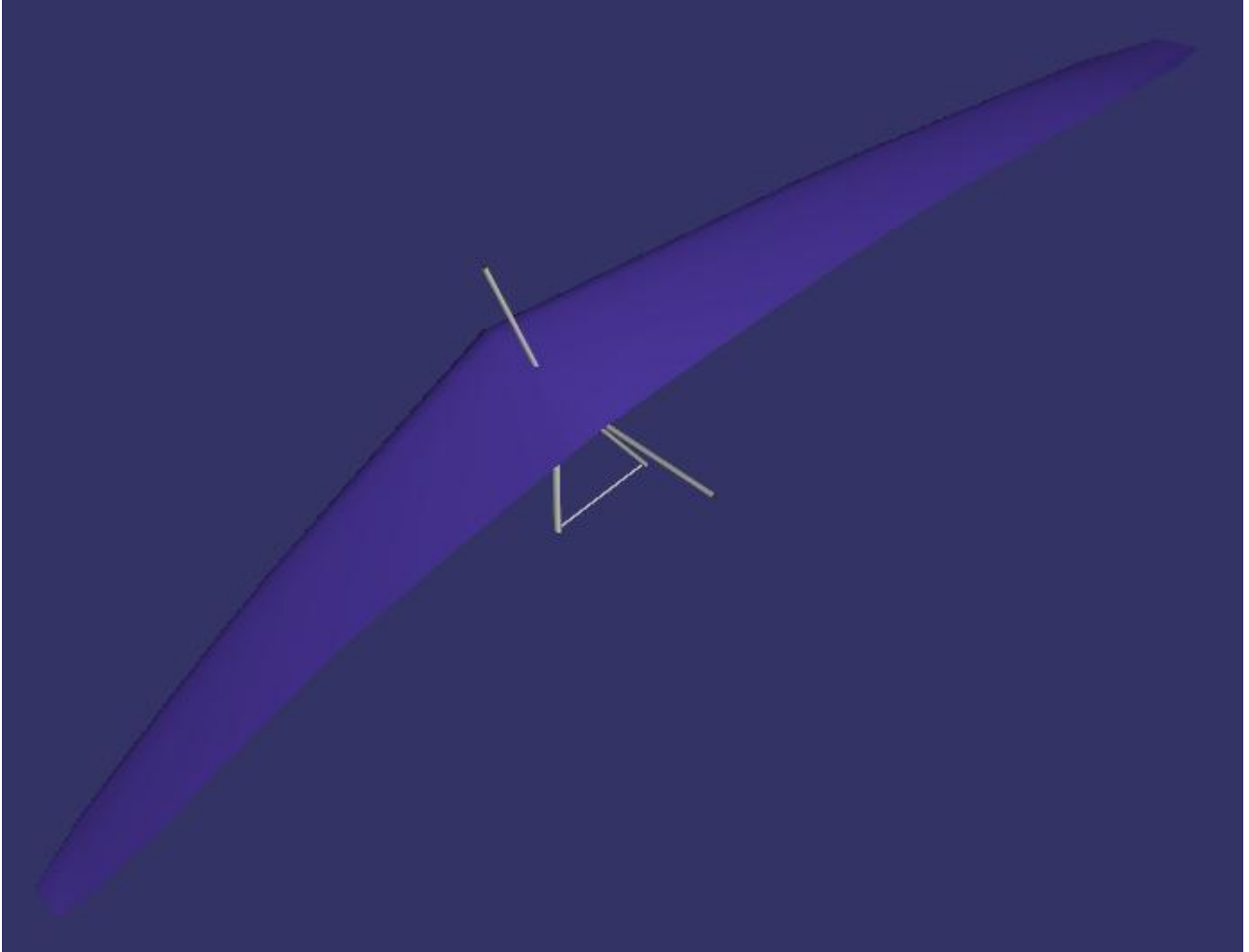


图 2.2

代码注释:

下面来解释以上代码以及程序所具有的功能:

在开始时包含必要的头文件, 在申请变量时加上 `osg::ref_ptr` 是指该变量是一个指针, 需要立即分配内存, OSG 很好的实现了超级指针, 意思是说在需要时才分配内存, 不需要时自动释放内存, 不用手动的 `DELETE` 等等, 但是老实说, OSG 本身是完美的, 但是在使用 OSG 时因为不小心或是与编译器的兼容而导致的内存暴泄所造成的问题确实让人分外头痛。

在使用 `osg::ref_ptr` 时, 当申请的参数做为指参数时, 往往不能直接写入变量名, 需要 `×.get()`, 从而得到指针实体。

在 `osg::ref_ptr` 中, 重载了一般判断指针的所有运算符, 如 `== != < >` 等等, 他有三个比较重要的成员变量列举如下:

osg::ref_ptr 成员函数

```
T* get() const //返回值为得到指针地址，一般做为参数时使用
bool valid() const //返回值类似于判断指针是否异常，如果异常如为空或野指针则返回 false
T* release() //功能为释放该指针所占用的内存，类似于 DELETE ptr
```

osgProducer::viewer 是一个场景的管理类，与之处于相同地位的还有 osgUtil::SceneView，本书中将主要使用 osgProducer::viewer 进行管理。与之相应的 OSG 有一个可执行程序名为 osgViewer，可以用 CMD 在任间路径打开（前提是安装了 OSG，且加入了 PATH），可以查看 OSG 所支持的模型类型，例如如下命令将出来与该代码相同的结果：

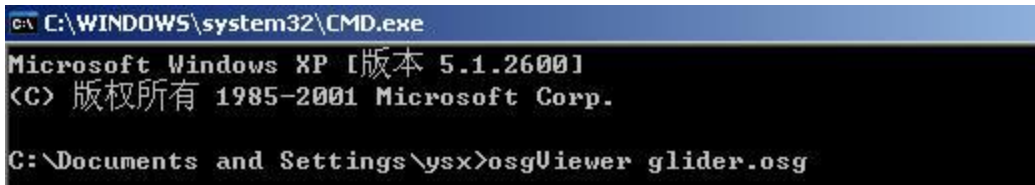


图 2.3

下面的代码：

```
Viewer ->setUpViewer(unsigned int options = STANDARD_SETTINGS);
参数: options 共可以取以下值:
NO_EVENT_HANDLERS 设置该选项清空 Viewer 中标准的 EventHandler ,而添加事件需要通过
getEventHandlerList().push_back(myEventHandler) 或者添加矩阵操作器，从而在其中
addCameraManipulator(myCameraManipualtor). 响应事件。
TRACKBALL_MANIPULATOR trackball 操作方式
DRIVE_MANIPULATOR Driver 操作方式
FLIGHT_MANIPULATOR Flight 操作方式
TERRAIN_MANIPULATOR Terrain 操作方式
UFO_MANIPULATOR UFO 操作方式
STATE_MANIPULATOR State 操作方式，这里决定众多的纹理显示，雾效等等
HEAD_LIGHT_SOURCE 在视口前添加灯光
SKY_LIGHT_SOURCE 在场景顶添加灯光
STATS_MANIPULATOR States 操作方式
VIEWER_MANIPULATOR Viewer 操作方式，关于帧速，写出模型等功能
ESCAPE_SETS_DONE 通过 ESC 键退出渲染
STANDARD_SETTINGS 是下列风格的并集 TRACKBALL_MANIPULATOR, DRIVE_MANIPULATOR,
FLIGHT_MANIPULATOR, TERRAIN_MANIPULATOR, STATE_MANIPULATOR, HEAD_LIGHT_SOURCE,
STATS_MANIPULATOR, VIEWER_MANIPULATOR, ESCAPE_SETS_DONE
osgDB::readNodeFile 函数，可以从文件中读入模型。
viewer->setSceneData(node.get ());设置场景数据，在这个程序中仅仅为一个飞机模型。
Viewer->realize();实现渲染表面。
Viewer->done(); 判断是否退出渲染，一般可以通过下面这个函数进行设置，意思为设置退出时间
void setDoneAtElapsedTime(double elapsedTime) ，单位应该是毫秒。
While 循环中进行的是更新渲染每帧操作。
```

程序的操作：

里面的所有的操作均是由 osgViewer 中自带的功能，当然也可以屏蔽掉，但是功能已经十分强大：

键盘	功能
1	TraceBall 操作器
2	Flight 操作器
3	Driver 操作器
4	Terrain 操作器
5	UFO 操作器
b	开/关背面剔除
f	全屏控制
h	帮助
l	开关灯光
o	将当前场景保存入 saved_model.osg 文件中
O/PrtSrn	抓图到 saved_image.rgb 中
s	显示帧速
t	开关纹理
v	开关垂直同步
w	点，线，面显示切换
z	录像/回放，路径存为 saved_animation.path 文件中
Z	回放/录像，路径存为 saved_animation.path 文件中

表 2.1

下面我们来仔细的看一下 osgViewer 工具：

osgViewer 是一个基于 OpenSceneGraph 的图形查看器，写它的初衷是要给大家展示如何使用 OSG 来写一个 viewer，但是它的功能已经足够强大到应用到实践中去。

在 CMD 中打开时，osgViewer 操作如下：

```
--dem <filename> 打开一个 image/dem, 高程图渲染
--display <type> MONITOR | POWERWALL | REALITY_CENTER | EAD_MOUNTED_DISPLAY
--display <type> MONITOR | POWERWALL | REALITY_CENTER | EAD_MOUNTED_DISPLAY
--help-all 显示所有的帮助
--help-env 显示与环境有关的帮助
--help-keys 显示与键盘鼠标有关的帮助
--image <filename> 渲染一幅画
--rgba 申请 RGBA 缓存
--run-till-elapsed-time 指明运行时间
--run-till-frame-number <integer> 指明运行的帧数
--stencil 请求 stencil 缓存
--stereo 显示立体
--stereo <mode> ANAGLYPHIC | QUAD_BUFFER | HORIZONTAL_SPLIT | VERTICAL_SPLIT
| LEFT_EYE | RIGHT_EYE | ON | OFF
```

到目前为止从最简单的 OSG 程序讲述了 OSG 的基本的渲染过程。下一节我们将讨论 OSG 与 MFC 相结合的控制。

2.1.2 MFC+OSG 控制

在 OSG 的渲染过程中，外界对其控制仅仅限于键盘与鼠标，因而更多的功能无法进行合理的安，比如我要模拟一些天气，天气有数十种，每种天气又有轻重不同，比如有大雪，中雪，小雪，这样的话光使用 WIN32 控制台+OSG 便显得有些捉襟见肘。

在模拟有些东西时 WIN32 根本无能为力，比如发送特定消息，等等，这就是 MFC 与 WINDOWS 的强项。控制台程序仅仅适合实验，算法模拟等等。

因此马上必须要引入一种框架来对其进行管理，可以选择 WINDOWS 程序框架，在开始时说过 WINDOWS 程序不具有清晰性，不易于入门，因此我们在本章开头定义了一个 MFC 程序，在 MFC 程序中，我们将主要通过菜单来控制 OSG 程序，键盘消息如果使用 MFC 进行拦截不做处理还将传递到 OSG 中去，下面我们以图来看一下 OSG 与 MFC 的具体交互过程：如图所示

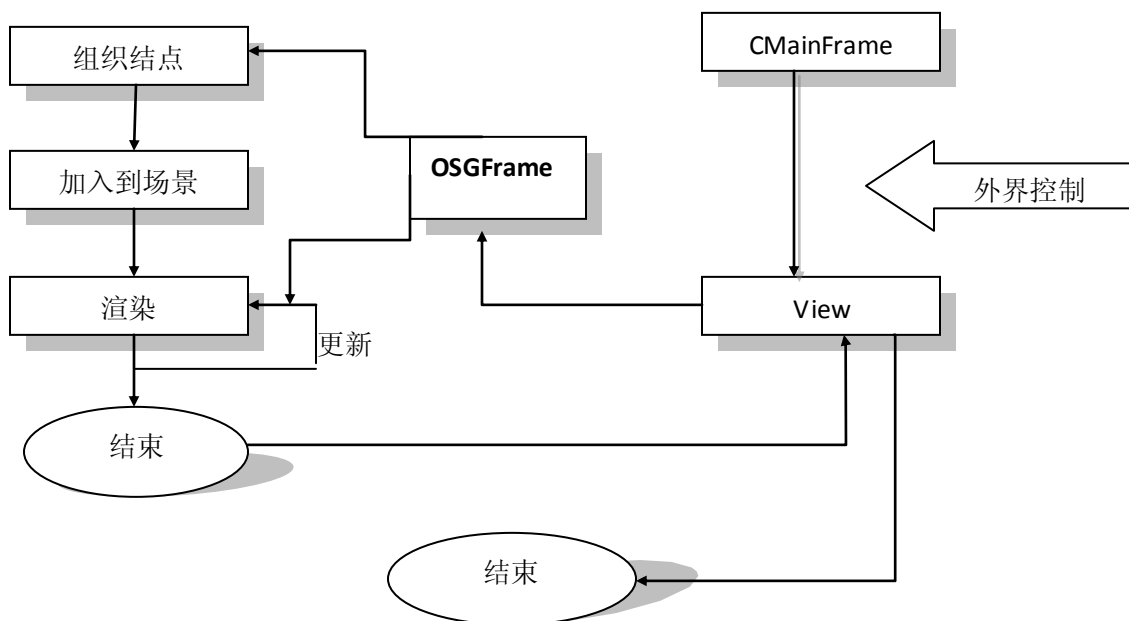


图 2.4

从图中可以看出，由于 MFC 中没有涉及 OSG 的变量，因而必须有一个类来统领 OSG 的全局，负责从组织结点一直到渲染结束的全过程，这个类就是如图所示的 OSGFrame，MFC 中代码过多之后，内存就会发生莫名其妙的紊乱，对于 OSG 来讲独立于 MFC 之外是十分必要的，采用一个非常平凡的接口来与 MFC 进行交互。

从图中可以看出，当渲染结束时，程序并没有退出，而退出程序的，只有 MFC 本身，当 view 发出退出消息时，哪怕不是在渲染也会退出。图中从 CMainFrame 到 view 到结束，这是应用程序的主线，而 osgFrame 只是程序中的支线，主线负责控制整个应用程序，而 OSG 只是其控制中的一部分。

那好根据这种思想，我们就来利用上节所做的例子，稍做调整使其变为一个可以演示模型的独立的例子。那么现在从第一章构建的最简单的 MFC 框架出发来构建 MFC+OSG 应用程序。

示例五：构建 MFC+OSG 基本框架

第一步：首先必须要添加 lib 依赖，在第一章的解决方案中，并没有添加与 OSG 相关的 LIB，点击菜单项目 → osgFree 属性 → 配置属性 → 链接器 → 命令行，在其中加入 OpenThreadsWin32d.lib

Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib
osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib osgUtil.d.lib, 如果要生成 Release 则需要不带 d 的版本。

第二步: 在程序中新添加一个类, 类名为 OSGFrame 用来进行 MFC 与 OSG 的交互, 添加类步骤为菜单项目→添加类→选择一般 C++类→类名填入 OSGFrame, 其它的默认, 表示无基类, 不是虚析构函数与内联, 添加类后发现项目中多了两个文件, 一个为 OSGFrame.h, 一个为 OSGFrame.cpp

下面依次在两个文件中添入如下内容, 当然函数与变量都可以通过 vs 来添加, 手动也可以添加:

```
OSGFrame.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#pragma once
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <CString>
#include <string>

//此函数用于OSG与MFC进行交互
//MFC中不会涉及OSG的变量
class OSGFrame
{
public:
    OSGFrame(void);
public:
    ~OSGFrame(void);

    //操作
public :
    //设置场景数据
    void setSceneData(CString );
    //渲染
    bool renderFrame() ;
    //对OSG进行初始化, 需要一个句柄, 来决定父窗口
    void Initialised(HWND) ;
    //是否已经初始化
    bool m_bInitializtion ;
    //viewer
    osg::ref_ptr <osgProducer::Viewer > m_pViewer ;
};
//OSGFrame.h结束
```

下面来看一下 OSGFrame.cpp 中的代码

```
OSGFrame.cpp
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#include "StdAfx.h"
```

```

#include "OSGFrame.h"//包含头文件
//构造
OSGFrame::OSGFrame(void)
{
    //初始化, 如果不初始化m_pViewer, 将会有类似于野指针的运行中断, 需要跟踪内存才会发现
    m_pViewer = new osgProducer::Viewer ();
    //初始时没有初始化OSG
    m_bInitializtion = false ;
}
//析构
OSGFrame::~~OSGFrame(void)
{
    //如果已经初始化且可用则析构
    if(m_pViewer.valid () && m_bInitializtion)
    {
        m_pViewer ->sync ();
    }
}
//初始化OSG设备
void OSGFrame::Initialised(HWND hwnd)
{
    //如果没有初始化, 就进行初始化
    if (!m_bInitializtion)
    {
        //进行viewer的标设置, 取消当按下ESC键程序退出功能
        m_pViewer->setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS &
            ~osgProducer::Viewer::ESCAPE_SETS_DONE);

        //使开始时不全屏, 以后会修改F键, 取消F键的所有操作, 否则窗口尺寸与框架将会不符, 程序崩溃
        m_pViewer->getCamera(0) ->getRenderSurface() ->fullScreen(false);
        //设置viewer所使用的窗口
        m_pViewer->getCamera(0) ->getRenderSurface() ->setWindow((const
        Producer::Window)hwnd);
        //首先设置一个组的空间, 如果是第一次需要打开一个默认模型, 则可以取消下面几行的注释添加FirstOpenSetNode函数
        //if (_bFirstOpen)
        //{
        //    FirstOpenSetNode ();
        //}
        //else
        //{
            //从文件中获得
            m_pViewer->setSceneData (new osg::Group ());
        }
    }
}

```

```

        //};
        //实例化
        m_pViewer->realize(Producer::CameraGroup::SingleThreaded);
        m_bInitializtion = true;
    }
    renderFrame();
}
//设置场景数据,最后一个字母写错成e,以免与viewer标准函数混肴
void OSGFrame::setSceneData(CString pathFileName)
{
    //申请根结点
    osg::Group * root = new osg::Group();

    //读取文件,作为结点
    osg::Node* pNode;

    //从文件中读取Model,在代码后,将详解各字符类型间的转换,下面只是一种方法W TO A
    USES_CONVERSION;
    std::string str= W2A(pathFileName);
    pNode = osgDB::readNodeFile(str);
    //加入到根结点
    root->addChild(pNode);
    m_pViewer->sync();
    m_pViewer->setSceneData(root);
    if (root != NULL)
    {
        //更新
        m_pViewer->update();
        m_pViewer->frame();
        //选择像机, 往往都会选择0 (TraceBall) 或者3 (Driver)
        m_pViewer->selectCameraManipulator(0);
        m_pViewer->getKeySwitchMatrixManipulator()//一行写不下了
        ->getMatrixManipulatorWithIndex(0)->home(0.0);
    }
}
//更新
bool OSGFrame::renderFrame()
{
    //渲染每一帧
    if (m_bInitializtion && m_pViewer.valid() && (m_pViewer->getSceneData() !=
NULL))
    {
        m_pViewer->sync();
        m_pViewer->update();
    }
}

```

```

        m_pViewer->frame();
        return true;
    }
    else
    {
        return false;
    }
}
} // OSGFrame 结束

```

关于字符之间的转换，平时使用的下面几种字符在 VS2005 中属于 ASCII 字符，简称 A 类型字符：

ASCII 字符（窄字符）：CHAR char LPCSTR LPSTR PCHAR PCSTR PSTR std::string

Unicode 字符（宽字符）：LPCWSTR LPWSTR PCWSTR PTCHAR PWCHAR PWSTR WCHAR CString

以及在 VS2005 下所有 T 字符类型

最常用的转换 A 为窄字符 W 为宽字符：

A to W 的方式有：

W = _T("A") ; W = TEXT("A"); W = L"A"; W = A2W(A);

W to A 的方式有：

A = W2A(W) ; A = W2BSTR(W) ;

第三步：在 osgFreeViewer.h 中添加 OSGFrame 类对象，改动如下：

```

osgFreeViewer.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#pragma once
//#include "afxwin.h"
#include "OSGFrame.h" //必须要包含此头文件
#include <CString>
//从窗口类继承
class CosgFreeViewer :
    public CWnd
{
public:
    CosgFreeViewer(void);

    //加入OSG场景管理变量
    OSGFrame *m_osgFrame ;

public:
    virtual ~CosgFreeViewer(void);

public :
    //继承虚准备创建函数
    virtual BOOL PreCreateWindow(CREATESTRUCT & cs) ;

public :
    DECLARE_MESSAGE_MAP()

```

```

//所有消息映射

public :
    afx_msg void OnPaint() ;
    afx_msg void OnSize(UINT nType, int cx, int cy) ;
    afx_msg void OnFileOpen() ;
public:
}; //osgFreeViewer.h结束

```

第四步：在 osgFreeViewer.cpp 中添加管理 OSGFrame 的代码：

```

osgFreeViewer.cpp
//OSG FreeS Source Code by FreeSouth. At zzu. vr. lab.
#include "StdAfx.h"
//添加需要的头文件
#include "osgFree.h"
#include "osgFreeViewer.h"

//添加消息映射序列
BEGIN_MESSAGE_MAP(CosgFreeViewer, CWnd)
    ON_WM_PAINT()
    ON_WM_SIZE()
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen)
END_MESSAGE_MAP()
//构造
CosgFreeViewer::CosgFreeViewer(void)
{
    //初始化
    m_osgFrame = new OSGFrame () ;
}
//析构
CosgFreeViewer::~~CosgFreeViewer(void)
{
    //删除
    delete m_osgFrame;
}
//准备创建
BOOL CosgFreeViewer::PreCreateWindow(CREATESTRUCT & cs)
{
    if (!CWnd::PreCreateWindow (cs))
        return FALSE ;

    //在使用OSG进行渲染时将取消背景填充色，
    cs.lpszClass = AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW,
LoadCursorFromFile(L"res/StarCraftArrow.ani"));
}

```

```

//设置为NULL_BRUSH, 取消最后一个参数默认为NULL_BRUSH
    return TRUE ;
}
//CosgFreeViewer消息处理函数
void CosgFreeViewer::OnPaint()
{
    CPaintDC dc(this) ;
    //消息处理代码
    //在OnPaint中写入渲染代码,用WM_PAINT是个惯例,在需要重绘时,系统会自动完成,而使用
    WM_DRAW则可能会出现一些问题
    m_osgFrame ->Initialised (m_hWnd) ;
}
//响应的WM_SIZE消息, 可手动添加消息映射即可
void CosgFreeViewer::OnSize (UINT nType, int cx, int cy)

{
    CWnd::OnSize (nType, cx, cy) ;
    //消息处理代码
    //当窗口大小改变时,重绘当前帧
    m_osgFrame ->renderFrame () ;
}
//打开文件
void CosgFreeViewer::OnFileOpen()
{
    //命令处理代码
    CFileDialog fileDlg(TRUE) ;
    if(fileDlg.DoModal () == IDOK)
    {
        //打开处理代码
        CString fileName (fileDlg.GetPathName ()) ;
        m_osgFrame ->setSceneDate (fileName) ;
    }
}
//osgFreeViewer.cpp结束

```

如果说仅仅添加这些代码, 也可以运行, 但是会死掉, 因为只有重绘时才会渲染更新帧, 因此需要控制CPU在空闲时间也要进行渲染。

第五步: 修改CMainFrame类

把CMainFrame中的下面两个函数改成如下所示:

```

BOOL CMainFrame::PreToRen (void)
{
    //这里添加渲染的准备工作
    return m_wndCosgFreeViewer.m_osgFrame ->renderFrame () ;
}

```

```
void CMainFrame::OnSetFocus (CWnd * )
{
    //将焦点转移到前视图窗口
    m_wndCosgFreeViewer.SetFocus () ;
}
```

把CosgFreeApp类中的一个函数修改成如下所示:

```
BOOL CosgFreeApp::OnIdle (LONG lCount)
{
    BOOL continue_processing = CWinApp::OnIdle(lCount) ;
    CMainFrame* pFrame = dynamic_cast<CMainFrame*>(m_pMainWnd);

    //在空闲时将做渲染准备,否则只有在重绘时才需要渲染
    if (pFrame)
    {
        if (pFrame->PreToRen())

    }
    return continue_processing;
}
```

如果在此时运行,当按下F键时会卡,处理方法有很多,最简单的办法是使用键盘钩子把F/f键钩掉,造成卡的原因是osgProducer::viewer的渲染窗口与主窗口不一致,无法进行裁切所致。有时候再点F键还可以回过来,现在在CMainFrame中添加钩子:

在MainFrm.cpp中做如下修改:

首先在OnCreate中添加钩子,关于钩子不是本节的主要内容,会使用这一功能即可

```
MainFrm.cpp
.....
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    .....

    //::SetCursor (cur) ;
    //安装一个键盘钩子,把F键钩掉,在以后我们也可以使用OSG中的键盘事件来屏蔽掉F键!
    SetWindowsHookEx (WH_KEYBOARD, KeyboardProc, NULL, GetCurrentThreadId()) ;
    FullScreen(m_bFullScreen) ;
    return 0 ;
}
.....
```

在MainFrm.cpp文件的开始加入:

```
MainFrm.cpp
```



```

.....
IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

//在此处添加一个消息映射,当设置焦点时,该映射会响应
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
    ON_WM_CREATE()
    ON_WM_SETFOCUS()
    ON_COMMAND(ID_FULLSCREEN, &CMainFrame::OnFullscreen)
END_MESSAGE_MAP()
//关于F键的钩子
LRESULT CALLBACK KeyboardProc(int code , WPARAM wParam, LPARAM lParam)
{
    //当按下F/f键时,通知程序已经处理,不再往下传递!
    if(0x46==wParam || 0x66 == wParam)
        return 1 ;
    else
        return 0 ;
}
// CMainFrame 构造/析构
CMainFrame::CMainFrame()
{
    m_bFullScreen = true ;
}.....

```

编译运行。运行结果如图：



图 2.5

到目前为止修改全部完毕，下面我们来看一下OSG+MFC中这几个函数的工作流程：

如下图所示：

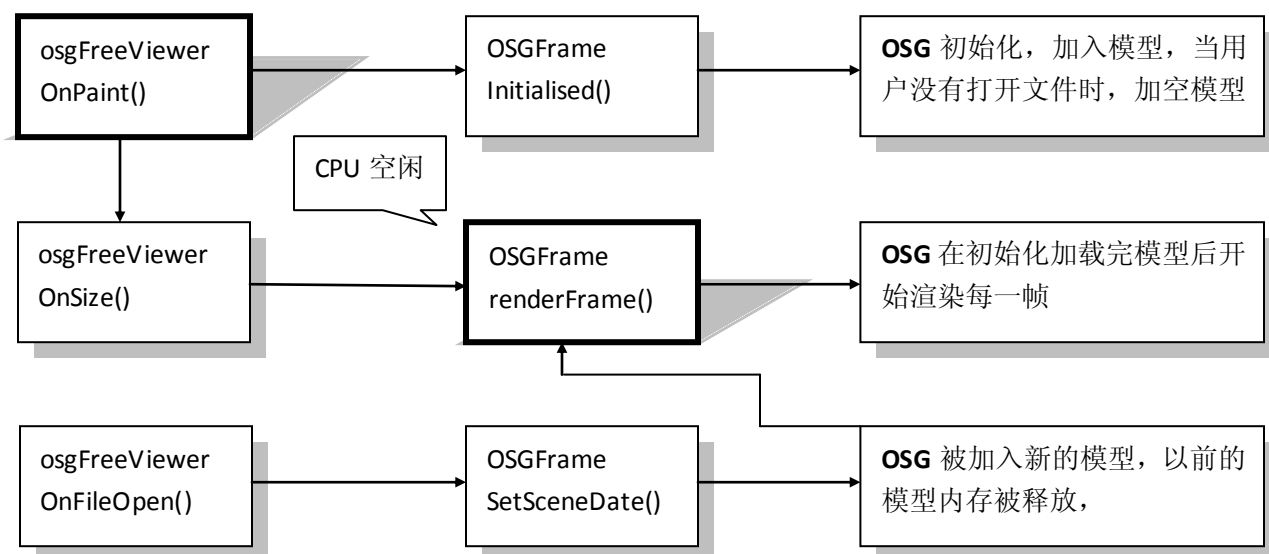


图2.6

在CPU空闲时会一直渲染，只需要初始化一次，就可以只需要对加入的数据修改就可以展现不同的模型，这就是为什么打开文件后不用初始化的缘故。

内存大释放是发生在`OnFileOpen`的时候，这个时候`OSGFrame`控制OSG重新加载模型，清除以前的所有的模型组，这样以前的内存并没有野掉，而是被释放掉了，通过做实验可以看得出来，连续打开一个模型几次，帧速是不会怎么改变的，证明每打开一个模型原先的内存会被释放掉！

到此为止已经建立完成一个非常好的MFC+OSG的框架了，以后我们的例子将有很多是在这个框架中进行的。

2.2 矩阵操作

其实绘制基本的几何体非常的简单，难的是要对它们进行的矩阵的操作，如旋转，平移等等，OSG 中应用矩阵的操作非常的平凡，从视口的移动到物体的移动再到路径的漫游，节点的回调等等，因此有必要我们首先来了解一下 OSG 中所涉及的关于矩阵的数学知识。

2.2.1 矩阵操作的数学原理

首先我们抛开代码来不管，来想象一下 OSG 中如何对一个物体有哪几种操作：平移，旋转，缩放。这三种基本的操作如果是一个物体可以反映在物体上，当然也可以反映在视口上。如果把石头从 A 移到 B，那么视口中石头从左上角跑到右下角，可以移动石头，当然也可以移动视口，因此 OSG 中有移动视口与移动物体两大类型的操作，同样旋转与缩放也是一样的。

首先我们来看一下图示，来理解一下刚才的概念。

请看下面图示，意思为石头从 A 移到 B 达到同样效果的两种不同的操作：

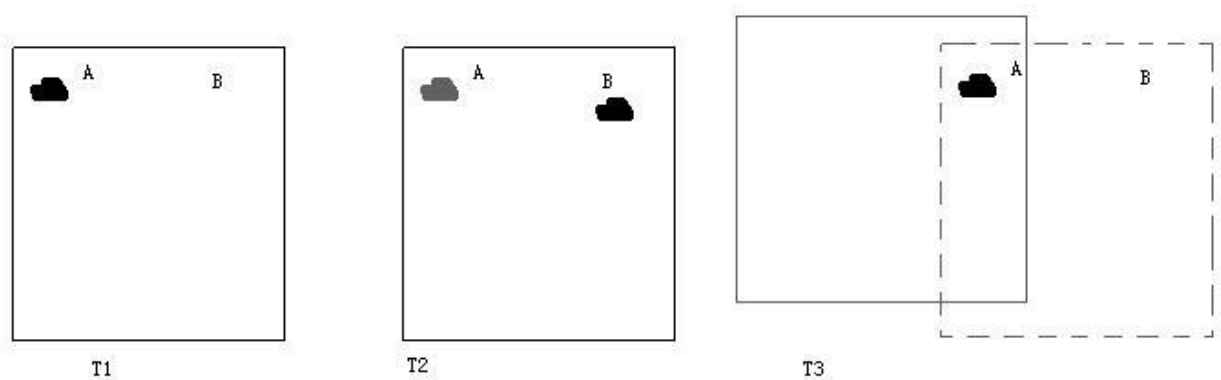


图 2.7

从效果上来看，T2 操作与 T3 操作具有相同的效果，因此对一个物体采用的操作也分为这两种，同样旋转也相同，可以是视口绕着物体转，也可以是物体绕着视口转，静止的两个物体也就等于一个物体，递归得知任何多与视口无分别相对移动的物体都是可以通过移动视口或是移动物体来达到相同的渲染效果的！然而在现实中往往界限并不是十分的分明！

下面来看一下物体的**平移**：

如果说当前物体的位置为 $(X, Y, Z, 1)$ ，那么用一个比如名叫 `transform(a, b, c)` 的函数其实执行了哪些操作达到移动后的位置是 $(X+a, Y+b, Z+c, 1)$ 的效果呢？

下面来介绍一下平移矩阵：

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{pmatrix}$$

注意到，当向量 $(X, Y, Z, 1)$ 与矩阵 T 相乘后得到结果是 $(X+a, Y+b, Z+c, 1)$ ，因此矩阵 T 称为平移矩阵移动一个点从一个点到另一个，其实就是该点的坐标与 T 相乘后得到新结果。

值得注意的是，当向量 $(X, Y, Z, 0)$ 与 T 相乘之后并没有产生预定的效果，原因是方向向量受平移操作的影响！

如果拿 T 来与该向量向乘，则 T 应该等于 T 的转秩。

旋转矩阵：

物体当前所处位置为 $(X, Y, Z, 1)$ ，使用 `Rotate(axis, angle)`，其中 `axis` 表示绕某轴旋转，而 `angle` 则表示旋转的角度，自然可以想到会有三种情况，分别绕 X, Y, Z 轴旋转，下面来看一下这三个旋转矩阵：

$$T_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad T_y = \begin{pmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad T_z = \begin{pmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

可以看到，对一个绕任意角度 a 的 3×3 旋转矩阵 R 来说，其对角线和为固定值，即称为迹 $\text{迹} = 1 + 2\cos a$

缩放矩阵：

缩放矩阵是指当某向量与该矩阵相乘之后，原值按比例缩小：

$$T_s = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

注意：在矩阵的 $(3, 3)$ 位置，如果值为非 1，则会引入有除法，除法是效率奇低的运算，因此长时间以来，人们把此位置永远置 1，起调节做用，从而这些矩阵也有时候被简化为 3×3 的矩阵进行运算，但这样是不科学的，毕竟需要考虑 $(3, 3)$ 位的取值。

通常来讲缩放矩阵左上 3×3 部分是不允许为负的，如果为负则被称为反射矩阵，这样将会产生顺时针排放的顶点序列从而产生很多不正确的效果，比如光照，纹理和裁切等等。

通常为了保持原有的比例，需要 $a=b=c$ ，当然不是绝对的。

关于矩阵还有很多内容，一般最基础的就这些，平移，旋转与缩放。

变换叠加：

一般来讲变换不是单一的，比如需要绕 x 轴旋转 120 度再缩小百分之二十最后向 y 轴奔跑一百米，那么该怎么计算呢，最显然的是首先 $R=P \cdot T_x(120)$ ，然后再拿给定的结果 $R2=R \cdot T_s(0.5)$ 依次类推，那么结果也就是 $P \cdot T_x(120) \cdot T_s(0.5)$ ，从而得出可以先把各种变换相乘之后最后再与 P 相乘，这样可以省去许多操作，很简单的例子，比如需要向前走一百米，再向后走一百米，计算之后就不需要走动，只需要向后转一百八十度就可以了，再举个例子，旋转六十度后再对 x 进行缩百分之五十，那么现在缩的对应的点如果先缩再旋转，是相差六十度的。

故而矩阵 $AB \neq BA$ 。

注意：四元组是与矩阵完全不同的东西，对于四元组比矩阵的优越性以及四元组代替矩阵的各种算法可以查阅相关的资料。

2.2.2 osgViewer 对于视口的控制

一般而言，无需对 **OSG** 当前的视口做任何的挑剔，从某种意义上来讲 **OSG** 最大程度上优化了对当前模型第一次出现的第一感觉，但是如果缺乏对观察系统最基本的了解，对于 **OSG** 的视窗有时候对于某些现象也会相当的迷惑，比如在进行多通道演示时在单机上的楼盘往往会变形，长宽不等、比例失调。对单个物体沿 x 轴移动一百个单位，但是从 **Viewer** 上看完全没有改变，让某物体绕自身 y 轴转动，它却沿着圆心某处大动干戈，最可恨的是好不容易做好的离子系统却发现雪是在从地上向天上飘。

因此现在来介绍一下关于视口的基本知识：

对于一个观察系统必须含有以下几个要素：

- 1：一个观察点的坐标，这是必须的，必须得清楚你在什么位置进行观察。
- 2：一个关于观察点的坐标系，这是定位当前空间内所有物体的坐标系，是必须的
- 3：一个观察平面，如果说你从显示器上看一栋楼时，严格的说你仅仅只看到显示器这样一个平面，也就是说有一个观察平面，场景的二维图像贴到该平面上。
- 4：观察的范围的平截体，举个例子说如果你闭上眼睛就什么也看不到了，这就是观察范围的定义。

这四个要素不可缺少，下面将以图示来说明这四个要素分别在现实中代表什么：
如下图所示：

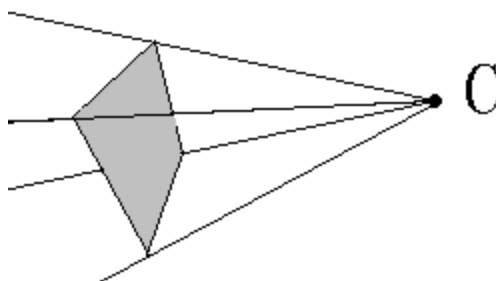


图 2.8

图中 **C** 表示观察点，灰色平面称为观察平面，而 **C** 与观察平面间的部分称为视见体，这是现实生活中的最小的视见体。

其中基于平行与观察平面的坐标系称为观察坐标系。

拿人本身来讲，看物体时可以仰头看，这说明当前视口绕当前观察坐标系平行于视平面的 **x** 轴进行旋转，同样你也可以扭头看，这是视口绕当前观察坐标系平行于视平面的 **y** 轴进行旋转，当然你也可以转歪着脖子看，这就是绕重直于视平面的 **z** 轴进行旋转，这些轴的定位不能说与平行于地平线垂直于天空的坐标系毫无关系。观察坐标系与世界坐标系存在一个矩阵的关系，也就是存在一个矩阵与之相乘可以使观察坐标转换为世界坐标。这个矩阵可以想象得到与仰角，方位角以及视点有关。关于此可以查阅相关资料。

下面来介绍两种最常见视见体几何模型：

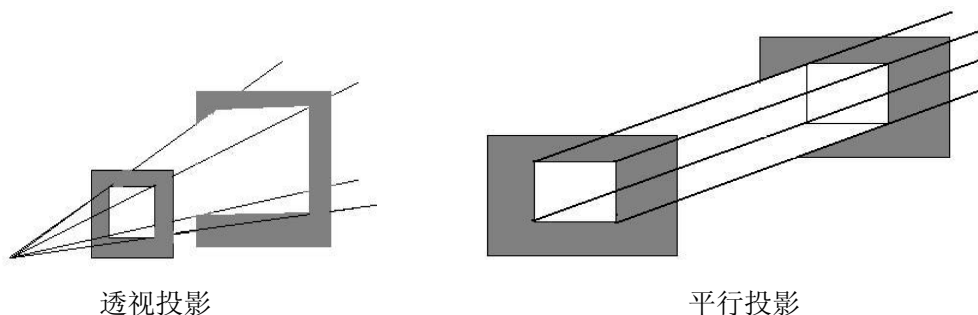


图 2.9

观察体的意义，需要根据观察场景进行裁切，透视变换，顶点混合等等处理，观察体的数学意义非常重大，因为漫游系统所见的东西为了保证效率都是经过根据观察体进行处理过的东西。

下面来介绍世坐标系与局部坐标系的转化：

局部坐标系：一个物体，给人的第一感觉是它基于 **y** 轴对称，其实这时指的是它的局部坐标系，为了建模的方便性，我们以参照物为原点是具有现实意义的，但是更多的操作则是在世界坐标系中。

世界坐标系，每个物体都有自己的局部坐标系，就像刚才说的眼睛一样。众多物体所在的全局坐标系称为世界坐标系，大多数的计算是基于世界坐标系的，比如光照，这就要求在存储物体时，对其顶点变换的同时必须对其法向进行变换。其实世界坐标系给人的感觉更有条理性。

当使用 `osgViewer` 观察一个模型时，你可以编代码把该模型使用 `PositionAttitudeTransform` 进行移动，但是移动后发现根本没有改变，这是因为 `viewer` 也在跟着物体移动，而使用 `MatrixTransform` 则可以轻易的对当前视口进行操作，在 `osgViewer` 打开模型没有按下空格键时，可以清楚的看到变化。

下面以图示来说明 `osgViewer` 对视口的观察体：

如下图所示：

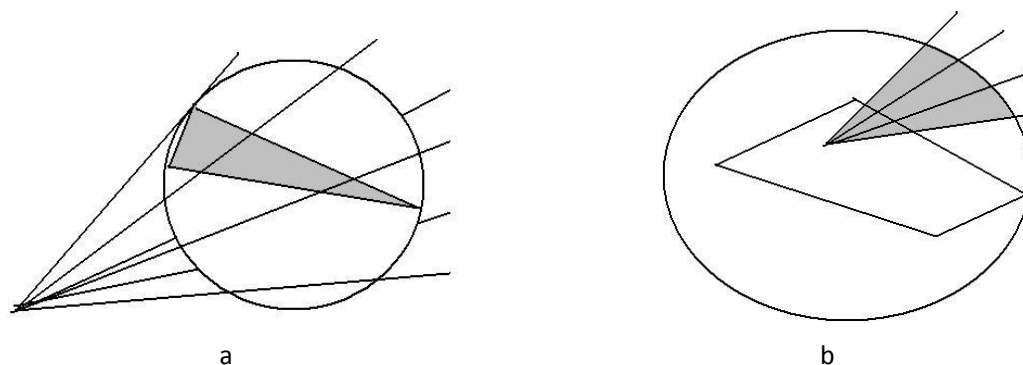


图 2.10

图 2.9a 中表示 `osgViewer` 默认的观察模式，当键下空格时，会恢复这种状态，即：视口体略略大于 `BoundingBox` 的外沿，这样就可以保证所有的物体都可以在视口中。

图 2.9b 中表示我们需要的结果，其中灰色表示视察体，我们需要在场景中漫游，下一章我们介绍场景漫游，碰撞检测以及 DLL 编程内容。

2.2.3 绘制基本几何体

下面通过修改上一章的代码来达到加载几何体的定义，首先我们要绘制一个顶点颜色混合的实心三角形在场景中，将介绍颜色绑定以及几何图形的绘制，对于颜色混合的插值算法我们不予考虑，`OSG` 内部实现。

运行效果：

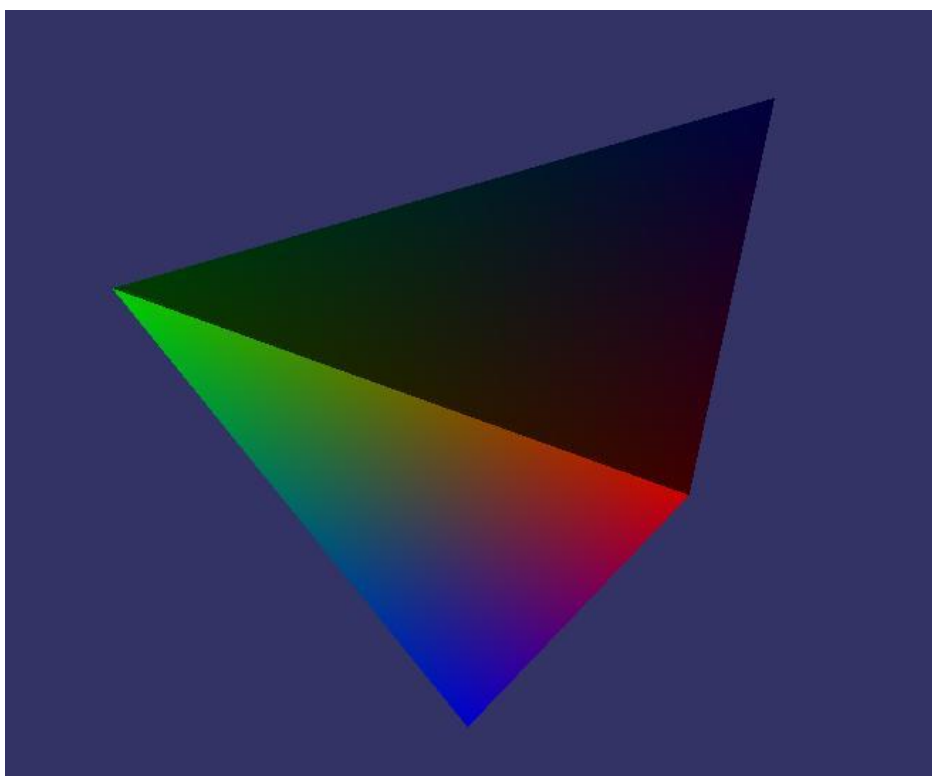


图 2.11

现在来介绍对上一章代码所做的改动，对于关键部分会给予详细的介绍：

除了增加菜单控制以外，OSG的功能都不需再对MFC进行修改，只涉及OSGFrame类，我们要让程序一开始运行就有几何体，而且打开菜单可以使用，这就要判断是不是程序刚刚运行，如果程序刚运行则加入三角形，否则加入打开菜单中的路径模型文件

示例六：绘制基本几何体

首先来看对于OSGFrame.h的改动

```
OSGFrame.h
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
#pragma once
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <osgDB/FileUtils>
#include <osg/Geode>
#include <osg/Geometry>
#include <osg/MatrixTransform> //矩阵变换，新加入的头文件
#include <CString>
#include <string>

//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
//此函数用于OSG与MFC进行交互
//MFC中不会涉及OSG的变量
class OSGFrame
{
public:
    OSGFrame(void);
public:
    ~OSGFrame(void);
    //操作
public:
    //设置场景数据
    void setSceneData(CString );
    //渲染
    bool renderFrame() ;
    //对OSG进行初始化
    void Initialised(HWND) ;
    //是否初始化
    bool m_bInitializtion ;
    //viewer
    osg::ref_ptr <osgProducer::Viewer > m_pViewer ;
    //默认打开控制
    bool m_bFirstOpen ;
public:
    //第一次打开时设置的模型，默认时没有模型，是蓝底屏幕
```

```

void FirstOpenSetNode() ;
//添加几何体，这是关键的函数，创建三角形
osg::Node *SetGeometry() ;
};OSGFrame.h结束

```

下面来看对OSGFrame.cpp的修改

OSGFrame.cpp

```

#include "StdAfx.h"
#include "OSGFrame.h"
//构造
OSGFrame::OSGFrame(void)
{
    //初始化
    m_pViewer = new osgProducer::Viewer () ;
    m_bInitializtion = false ;
    m_bFirstOpen = true ;//第一次打设置为TRUE，当使用打开菜单此，此标识为FALSE
}
//析构
OSGFrame::~~OSGFrame(void)
{
    //如果已经初始化且可用则析构
    if(m_pViewer.valid () && m_bInitializtion)
    {
        m_pViewer ->sync () ;
    }
}
//初始化OSG设备
void OSGFrame::Initialised(HWND hwnd)
{
    //如果没有初始化,就进行初始化
    if (!m_bInitializtion)
    {
        //进行viewer的标设置,取消当按下ESC键程序退出功能
        m_pViewer->setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS &
            ~osgProducer::Viewer::ESCAPE_SETS_DONE);

        //使开始时不全屏,以后会修改F键
        m_pViewer->getCamera(0) ->getRenderSurface() ->fullScreen(false);

        //设置viewer所使用的窗口
        m_pViewer->getCamera(0) ->getRenderSurface() ->setWindow((const
        Producer::Window)hwnd);
    }
}

```



```

        //第一次打开将打开一个默认模型,从文件中打开时该标识置FALSE
        if (m_bFirstOpen)
        {
            //在该函数中加载第一次打时的模型
            FirstOpenSetNode() ;//
        }

        else
        {
            //从文件中获得
            m_pViewer->setSceneData (new osg::Group ()) ;
        };
        //实例化
        m_pViewer->realize (Producer::CameraGroup::SingleThreaded);
        m_bInitializtion = true;
    }
    renderFrame() ;
}

.....

//第一次渲染时场景的结点
void OSGFrame::FirstOpenSetNode()
{
    osg::Group * root = new osg::Group () ;

    //读取文件,作为结点
    root ->addChild (SetGeometry()) ;

    //设置场景数据结点
    m_pViewer->setSceneData (root) ;

    m_bFirstOpen = false ;
}

//设置几何体,两个三角形
osg::Node *OSGFrame::SetGeometry ()
{
    osg::Group* root = new osg::Group();
    //Geometry node,用于组织几何体,几何体加入geode, geode加入到根root中
    osg::Geode* triangleGeode = new osg::Geode();
    osg::Geometry* triangleGeometry = new osg::Geometry();

    //下面的这两个加入过程也可以放在创建三角形以后
    //把三角型几何体加入到Geode中去
    triangleGeode->addDrawable(triangleGeometry);

```

//把该几何体结点加入到根结点中

```
root->addChild(triangleGeode); //等于增加了一个三角形
```

//现在来申请三角形的三个顶点,实际上来讲Vec3Array 可以看做是vector<osg::Vec3>,在现实
中也可以用此相互的代替

//建立三角形的三个顶点放入向量中

```
osg::Vec3Array* triangleVertices = new osg::Vec3Array;  
triangleVertices->push_back( osg::Vec3( 0, 0, 0) );  
triangleVertices->push_back( osg::Vec3(10, 0, 0) );  
triangleVertices->push_back( osg::Vec3(0 , 7, 0) );
```

//把这些顶点关联到三角形中

```
triangleGeometry->setVertexArray( triangleVertices );
```

//然后决定这些顶点的组织方式

```
osg::DrawElementsUInt* triangleBase =  
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);  
triangleBase->push_back(2);  
triangleBase->push_back(1);  
triangleBase->push_back(0);  
triangleGeometry->addPrimitiveSet(triangleBase);
```

/*如果画别的几何体,可设置osg::PrimitiveSet::TRIANGLES为相应组织方式,与OPENGL相同,
具有下列值 POINTS = GL_POINTS, LINES = GL_LINES, LINE_STRIP = GL_LINE_STRIP,
LINE_LOOP = GL_LINE_LOOP,
TRIANGLES = GL_TRIANGLES, TRIANGLE_STRIP = GL_TRIANGLE_STRIP, TRIANGLE_FAN =
GL_TRIANGLE_FAN, QUADS = GL_QUADS,
QUAD_STRIP = GL_QUAD_STRIP, POLYGON = GL_POLYGON
*/

//下面来设置颜色,用四元组来存放,把三个顶点关联三个颜色

```
osg::Vec4Array* colors = new osg::Vec4Array;  
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //红  
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //绿  
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //蓝
```

//建立颜色结点索引,该类是从ARRAY继承而来

```
osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,3,3>  
*colorIndexArray;  
colorIndexArray =new osg::TemplateIndexArray<unsigned int,  
osg::Array::UIntArrayType,3,3>;  
colorIndexArray->push_back(0);  
colorIndexArray->push_back(1);  
colorIndexArray->push_back(2);
```

```

//设置颜色数组
triangleGeometry->setColorArray(colors);
//设置索引
triangleGeometry->setColorIndices(colorIndexArray);
//设置绑定方式,逐点
triangleGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
osg::MatrixTransform *trans = new osg::MatrixTransform;
trans->setMatrix(osg::Matrix::rotate(osg::DegreesToRadians(125.0),1.0,0.0,0.0));
//把物体旋转125度,围绕x轴,再次加入到场景中
//移动后再次加入几何体到根结点中,此时加入两个三角形
trans->addChild(triangleGeode);
root->addChild(trans);
return (osg::Node*) root ;
} //OSGFrame.cpp结束

```

此时程序结束,下面来介绍一下osg::MatrixTransform这个类:

如果不加移动,二个三角形毫无疑问将重合,在加入第一个三角形时,它的坐标是000,而加入第二个时,首先把其加入到MatrixTransform中,然后把trans再加入到场景中,trans所起的作用就是把自己的child移动变换后加入到parents中去,主要起变换位置的作用,来看它的经常的应用,主要有以下几个:

```

//设置结点回调
tran->setUpdateCallback(new osg::AnimationPathCallback(osg::Vec3(20, 0, 0),
osg::Z_AXIS, osg::inDegrees(145.0f)));
//设置旋转
trans->setMatrix(osg::Matrix::rotate(osg::DegreesToRadians(125.0),1.0,0.0,0.0));
//设置移动
trans->setMatrix(osg::Matrixd::translate(osg::Vec3f(10.0f, 0.0f,0.0f)));

```

这是最常用的三种格式,关于其它的函数功能,不一一列举,读者可以去查阅帮助文档。

下面我们来使用另一种操作来移动几何体,我们来建立两个三角形,相距40.0f且另一个绕自身的z轴旋转。z轴指指向天空的轴。

在OSGFrame.h中加入头文件: `<osg/AnimationPath><osg/PositionAttitudeTransform>`

在SetGeometry ()中做如下修改:

```

OSGFrame.cpp
//by @FreeSouth at ZZU VR LAB for <OSG FreeS > ieysx@163.com
osg::Node *OSGFrame::SetGeometry ()
{
    osg::Group* root = new osg::Group();
    //Geometry node,可用于组织几何体

```

```

osg::Geode* triangleGeode = new osg::Geode();
osg::Geometry* triangleGeometry = new osg::Geometry();
//把三角型几何体加入到Geode:Geometry node 中去
triangleGeode->addDrawable(triangleGeometry);
//把该几何体结点加入到根结点中
//root->addChild(triangleGeode);把该句删除
.....

triangleGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
//对该图形进行移动
osg::PositionAttitudeTransform* trans0 =
    new osg::PositionAttitudeTransform();

osg::PositionAttitudeTransform* trans1 =
    new osg::PositionAttitudeTransform();

osg::MatrixTransform *tran=new osg::MatrixTransform ();
    tran->setUpdateCallback(new osg::AnimationPathCallback(osg::Vec3(20, 0,
0), osg::Z_AXIS, osg::inDegrees(145.0f)));

    //加入三角形
root->addChild(tran);
//加入第二个三角形
root->addChild(trans1);
//移动变换
trans0->addChild(triangleGeode);
trans1->addChild(triangleGeode);
tran ->addChild (trans0) ;
//设置20 0 0点为局部坐标原点
trans0->setPivotPoint (osg::Vec3f (20.0, 0.0f, 0.0f)) ;
osg::Vec3 position0(20,0,0);
//设置位置, 在Position0点
trans0->setPosition( position0 );

osg::Vec3 position1(-20,0,0);
trans1->setPosition( position1 );
//返回根结点
return (osg::Node*) root ;
}

```

这样就可以得到另一个三角形绕自己z轴转的效果, 主要使用了下面这个类, 现在我们来看一下这个类, osg::PositionAttitudeTransform, 这个是直接设置在世界坐标系中的位置, 而并非基于原始的地方进行平移等操作, 比之刚才更加直接。

常用的操作:

```
//设置原点
```

```
trans0->setPivotPoint (osg::Vec3f (20.0, 0.0f, 0.0f)) ;
```

```
//设置位置
```

trans0->setPosition(position0);注意: 一旦设置原点, 对其ChildNode基于位置的操作将全部是基于该原点的局部坐标系:

很多时候复杂的移动操作要涉及几个类, 一般有以下几个:

Osg::Matrix*: *表示如osg::Matrixd, osg::Matrixf等

一般负责生成矩阵或是向量, 从而应用到变换当中去

Osg::PositionAttitudeTransform

用于确定位置, 设置局部坐标系

osg::AnimationPath

用于生成路径以及回调

Osg::MatrixTransform

用于各种变换, 用处最广

下面以图示来说明各类对模型的操作:

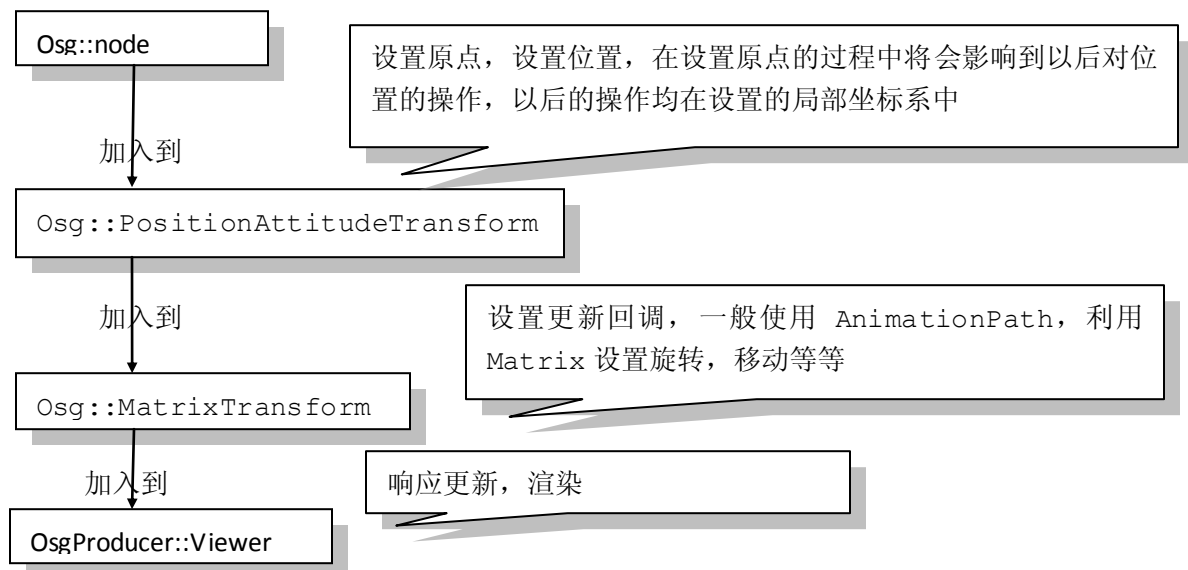


图2.12

到此为止, 对主要的矩阵操作做了一定的介绍, 通过多次调试程序, 走出OSG中的陷阱与BUG, 更好的理解OSG。这一节需要很多次的尝试才可以熟练掌握。

2.3 绘制各种几何体

OSG 中支持很多种的几何体, 绘制方法千篇一律, 可以以此类推, 下面在场中绘制两个三角形, 一个长方体, 一个圆锥, 外加一个圆柱:

需要注意的是各种几何体均有精度, 大家一定都对 OPENGL 非常熟悉, 画一个球体如果精度很就是一个八面体, 精度大了自然顶点就多, 三角形面就多:

运行效果:

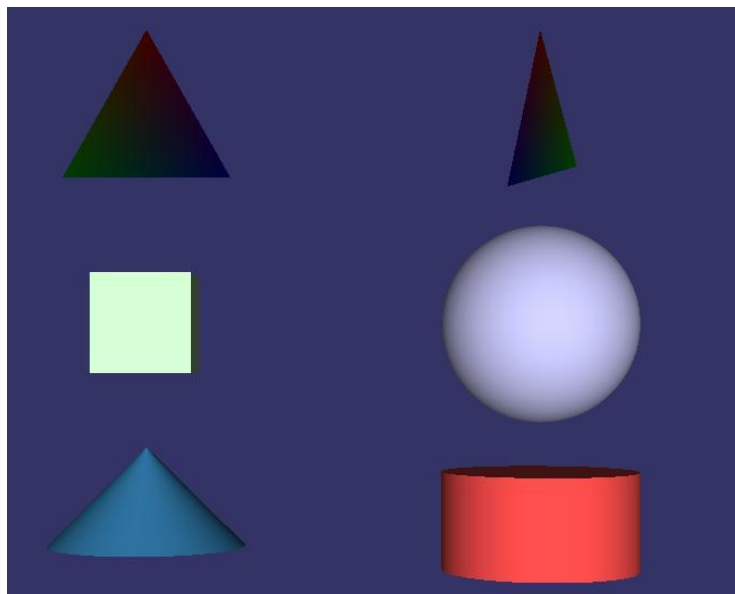


图 2.13

示例七：绘制众多几何体

在OSGFrame.h中加入以下头文件： <osg/ShapeDrawable>

在OSGFrame.cpp的SetGeometry () 中修改如下：

```
//设置几何体,两个三角形
osg::Node *OSGFrame::SetGeometry ()
{
    osg::Group* root = new osg::Group();
    //Geometry node, 可用于组织几何体
    osg::Geode* triangleGeode = new osg::Geode();
    osg::Geometry* triangleGeometry = new osg::Geometry();

    //把三角型几何体加入到Geode:Geometry node 中去
    triangleGeode->addDrawable(triangleGeometry);
    //把该几何体结点加入到根结点中
    //root->addChild(triangleGeode);

    //现在来申请三角形的三个顶点,实际上来讲Vec3Array 可以看做是vector<osg::Vec3>,在现实
    //中也可以用此相互的代替
    osg::Vec3Array* triangleVertices = new osg::Vec3Array;
    triangleVertices->push_back( osg::Vec3( 0, 0, 10) );
    triangleVertices->push_back( osg::Vec3(-8.5, 0, -5) );
    triangleVertices->push_back( osg::Vec3(8.5, 0, -5) );

    //把这些顶点关联到三角形中
    triangleGeometry->setVertexArray( triangleVertices );

    //然后决定这些顶点的组织方式
```

```

osg::DrawElementsUInt* triangleBase =
    new osg::DrawElementsUInt(osg::PrimitiveSet::TRIANGLES, 0);
triangleBase->push_back(2);
triangleBase->push_back(1);
triangleBase->push_back(0);
triangleGeometry->addPrimitiveSet(triangleBase);

//下面来设置颜色,用四元组来存放
osg::Vec4Array* colors = new osg::Vec4Array;
colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //红
colors->push_back(osg::Vec4(0.0f, 1.0f, 0.0f, 1.0f) ); //绿
colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //蓝

//建立颜色结点索引
osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,3,3>
*colorIndexArray;
colorIndexArray =new osg::TemplateIndexArray<unsigned int,
osg::Array::UIntArrayType,3,3>; //Geometry() 待续
colorIndexArray->push_back(0); //Geometry()续
colorIndexArray->push_back(1);
colorIndexArray->push_back(2);

//设置颜色数组
triangleGeometry->setColorArray(colors);
//设置索引
triangleGeometry->setColorIndices(colorIndexArray);
//设置绑定方式,逐点
triangleGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);

//对该图形进行移动
osg::PositionAttitudeTransform* trans0 =
    new osg::PositionAttitudeTransform();

osg::PositionAttitudeTransform* trans1 =
    new osg::PositionAttitudeTransform();

osg::MatrixTransform *tran=new osg::MatrixTransform ();
//tran->setMatrix (osg::Matrix::translate (osg::Vec3 (0, 0, 10)) );
tran->setUpdateCallback(new osg::AnimationPathCallback(osg::Vec3(20, 0, 0),
osg::Z_AXIS, osg::inDegrees(145.0f)));

```

```

    //加入三角形
    root->addChild(tran);
    root->addChild(trans1);
    trans0->addChild(triangleGeode);
    trans1->addChild(triangleGeode);
    tran ->addChild (trans0) ;
    osg::Vec3 position0(20,0,0);
    trans0->setPosition( position0 );
    osg::Vec3 position1(-20,0,0);
    trans1->setPosition( position1 );

    //下面来创建圆形
    osg::ref_ptr<osg::Geode> geode = new osg::Geode ;
    //决定圆的半径
    const float radius = 5.0 ;
    //决定圆形的精确度
    osg::ref_ptr<osg::TessellationHints> hints = new osg::TessellationHints;
    hints->setDetailRatio(2.0f);
    //加入到drawable中
    osg::ref_ptr<osg::ShapeDrawable> shape;
    //现在开始画圆
    shape = new osg::ShapeDrawable(new osg::Sphere(osg::Vec3(20.0f, 0.0f, -20.0f),
radius * 2), hints.get());
    //设置颜色
    shape->setColor(osg::Vec4(0.8f, 0.8f, 1.0f, 1.0f));
    //加入到结点中
    geode->addDrawable(shape.get());

    //画长方体
    osg::ref_ptr<osg::Geode> geode2 = new osg::Geode ;
    //创建长方体，原点，长宽高以及精度
    shape = new osg::ShapeDrawable(new osg::Box(osg::Vec3(-20.0f, 0.0f, -20.0f),
10.0, 10.0, 10.0), hints.get());
    //设置颜色
    shape->setColor(osg::Vec4(0.8f, 1.0f, 0.8f, 1.0f));
    geode->addDrawable(shape.get());

    root ->addChild (geode2.get()) ;

    //画扇形与圆柱形
    shape = new osg::ShapeDrawable(new osg::Cone(osg::Vec3(-20.0f, 0.0f, -40.0f),
radius*2, 10.0), hints.get());
    shape->setColor(osg::Vec4(0.2f, 0.5f, 0.7f, 1.0f));
    geode->addDrawable(shape.get());

```



```

    shape = new osg::ShapeDrawable(new osg::Cylinder(osg::Vec3(20.0f, 0.0f,
-40.0f), radius*2, 10.0), hints.get());
    shape->setColor(osg::Vec4(1.0f, 0.3f, 0.3f, 1.0f));
    geode->addDrawable(shape.get());
    root ->addChild (geode.get ());
    return (osg::Node*) root ;
}

```

2.4 综合例子：制做时钟

示例八：制做时钟

制做时钟分为几步：

第一步是画表盘，因为表盘是不会动的，第二步是创建三个针，时针，分针，秒针，第四步是根据当前系统时间初始化三个针且设置三个针绕轴转的速度。

注意：获取当前时间可能是准确的，但是当表走一定时候时，可能因为缺乏校准而不准确，原因是因为 CPU 快慢不同，程序执行时可能受到 CPU 的影响，比如卡一下，死机一下，或是只是让它慢以下，这样就导致了客观时间与 CPU 的时差，关于这样问题很多，一个解决方法是隔一段时间校准一次，不过本节只是一个小例子，不用那么大动干戈。

用直线来画圆要用一点小技巧，确定圆心 A, 半径 r, 圆的周边坐标为 $(A+rcos\alpha, A+rsin\alpha)$, 精度取决于 $\Delta\alpha$

运行结果：

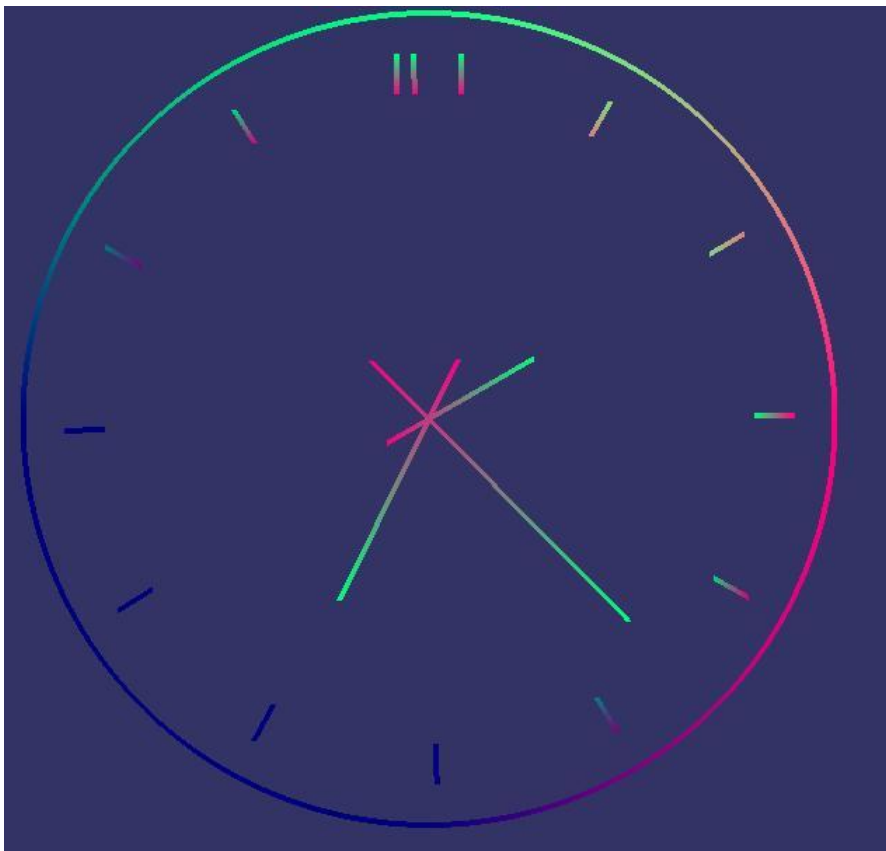


图 2.14

代码如下：其中函数可以手动添加，也可以自动添加。基于上一个程序，改动的地方只有OSGFrame.h与OSGFrame.cpp

在代码中会有详细的解释：

OSGFrame.cpp

```
//OSG FreeS Source Code by FreeSouth. At zzu. vr. lab. ieysx@163.com
#pragma once
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <osgDB/FileUtils>
#include <osg/Geode>
#include <osg/Geometry>
#include <osg/PositionAttitudeTransform>
#include <osg/MatrixTransform>
#include <CString>
#include <string>
#include <osg/AnimationPath>
#include <osg/ShapeDrawable>
#include <osg/LineWidth>

class OSGFrame
{
public:
    OSGFrame(void);
public:
    ~OSGFrame(void);
    //操作
public :
    //设置场景数据
    void setSceneDate(CString ) ;
    //渲染
    bool renderFrame() ;
    //对OSG进行初始化
    void Initialised(HWND) ;
    //是否初始化
    bool m_bInitializtion ;
    //viewer
    osg::ref_ptr <osgProducer::Viewer > m_pViewer ;
    //默认打开控制
    bool m_bFirstOpen ;
public :
    //第一次打开时设置的模型
    void FirstOpenSetNode() ;
    osg::Node* CreateClockFace(void) ;
```

```

//创建表针, 起点到终点画直线
osg::Node* CreateHandler(osg::Vec3 &, osg::Vec3 &);
// 创建路径, 三个参数, 绕center点转, 转的radius, 多长时间looptime内转多少angle
osg::AnimationPath* createAnimationPath(const osg::Vec3& center, float
radius, double looptime, float angle);
// 创建动态表针
osg::Node* CreateMoveHandler(void);
};
//OSGFrame.h结束

```

下面来看OSGFrame.cpp文件:

```

OSGFrame.cpp
//OSG FreeS Source Code by FreeSouth. At zzu. vr. lab. ieysx@163.com
#include "StdAfx.h"
#include "OSGFrame.h"
//构造
OSGFrame::OSGFrame(void)
{
    //初始化
    m_pViewer = new osgProducer::Viewer ();
    m_bInitializtion = false ;
    m_bFirstOpen = true ;
}
//析构
OSGFrame::~~OSGFrame(void)
{
    //如果已经初始化且可用则析构
    if(m_pViewer.valid () && m_bInitializtion)
    {
        m_pViewer ->sync ();
    }
}
//初始化OSG设备
void OSGFrame::Initialised(HWND hwnd)
{
    //如果没有初始化, 就进行初始化
    if (!m_bInitializtion)
    {
        //进行viewer的标设置, 取消当按下ESC键程序退出功能
        m_pViewer->setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS &
~osgProducer::Viewer::ESCAPE_SETS_DONE);

        //使开始时不全屏, 以后会修改F键
    }
}

```

```

        m_pViewer->getCamera(0)->getRenderSurface()->fullScreen(false);
        //设置viewer所使用的窗口
        m_pViewer->getCamera(0)->getRenderSurface()->setWindow((const
Producer::Window)hwnd);
        //首先设置一个组的空间,如果是第一次打开将打开一个默认模型,否则从文件中打开模型
        if (m_bFirstOpen)
        {
            FirstOpenSetNode();
        }
        else
        {
            //从文件中获得
            m_pViewer->setSceneData (new osg::Group ());
        };
        //实例化
        m_pViewer->realize(Producer::CameraGroup::SingleThreaded);
        m_bInitializtion = true;
    }
    renderFrame();
}
//设置场景数据,最后一个字母写错成e,以免与viewer标准函数混淆
void OSGFrame::setSceneDate(CString pathFileName)
{
    //申请根结点
    osg::Group * root = new osg::Group ();
    //读取文件,作为结点
    osg::Node* pNode ;
    //从文件中读取Model
    USES_CONVERSION;
    std::string str= W2A(pathFileName);
    pNode = osgDB::readNodeFile (str);
    //加入到根结点
    root ->addChild (pNode);
    m_pViewer->sync();
    m_pViewer->setSceneData(root);

    if (root != NULL)
    {
        //更新
        m_pViewer->update();
        m_pViewer->frame();
        m_pViewer->selectCameraManipulator(0);
        m_pViewer->getKeySwitchMatrixManipulator()->getMatrixManipulatorWithIndex(
0)->home(0.0);
    }
}

```

```

    }
}
//更新
bool OSGFrame::renderFrame ()
{
    if (m_bInitializtion && m_pViewer.valid() && (m_pViewer->getSceneData() !=
NULL))
    {
        m_pViewer->sync();
        m_pViewer->update();
        m_pViewer->frame();
        return true;
    }
    else
    {
        return false;
    }
}
//第一次渲染时场景的结点
void OSGFrame::FirstOpenSetNode()
{
    osg::Group * root = new osg::Group () ;
    //设置表针与表盘
    root ->addChild (CreateClockFace ()) ;
    root ->addChild (CreateMoveHandler());
    //设置场景数据结点
    m_pViewer->setSceneData (root) ;

    m_bFirstOpen = false ;
}
//创建表盘
osg::Node * OSGFrame::CreateClockFace ()
{
    //申请根结点，做为返回值
    osg::Group * root = new osg::Group () ;

    //设置线宽
    osg::ref_ptr <osg::LineWidth> LineSize = new osg::LineWidth;
    LineSize ->setWidth (4.0) ;
    //打开线宽属性
    root->getOrCreateStateSet () ->setAttributeAndModes (LineSize.get
()), osg::StateAttribute::ON);
    //表盘几何结点
    osg::Geode* clockGeode = new osg::Geode();

```

```

//表盘圈
osg::Geometry* clockGeometry = new osg::Geometry();
//数字
osg::Geometry* NumPoint = new osg::Geometry();
//加入到Geode中去
clockGeode->addDrawable(clockGeometry);
clockGeode->addDrawable(NumPoint);
//加入表盘结点
root->addChild(clockGeode);
//申请数字结点顶点
osg::ref_ptr<osg::Vec3Array> Num = new osg::Vec3Array;
//存放经过插值得到的所有点，而后画直线连成表盘
osg::ref_ptr<osg::Vec3Array> allPoints = new osg::Vec3Array;
//表盘颜色
osg::Vec4Array* colors = new osg::Vec4Array;
//表针颜色
osg::Vec4Array* colors2 = new osg::Vec4Array;
//加入颜色，得到十二个数字的顶点
for (double i = 0.0; i < 6.28; i += 0.52)
{
    colors2 ->push_back (osg::Vec4f (sin(i), cos(i), 0.5, 1.0));
    colors2 ->push_back (osg::Vec4f (cos(i), sin(i), 0.5, 1.0));
    if(i == 0.0)
    {
        //保证每个顶点一种颜色，在12:00这个数上多画两道予以区分
        colors2 ->push_back (osg::Vec4f (sin(i), cos(i), 0.5, 1.0));
        colors2 ->push_back (osg::Vec4f (cos(i), sin(i), 0.5, 1.0));
        Num ->push_back (osg::Vec3 (45*sin(i)-4,-0.0, 45*cos(i)));
        Num ->push_back (osg::Vec3 (40*sin(i)-4,-0.0, 40*cos(i)));
        Num ->push_back (osg::Vec3 (45*sin(i)+4,-0.0, 45*cos(i)));
        Num ->push_back (osg::Vec3 (40*sin(i)+4,-0.0, 40*cos(i)));
    }
    else
    {
        Num ->push_back (osg::Vec3 (45*sin(i),-0.0, 45*cos(i)));
        Num ->push_back (osg::Vec3 (40*sin(i),-0.0, 40*cos(i)));
    };
}
//得到半径是50的钟表盘面外圈的所有顶点， $\Delta a=0.02$ ,在外圈共压入314个点，形成314条线
for (double i = 0.0; i < 6.28; i += 0.02)
{
    colors ->push_back (osg::Vec4f (sin(i), cos(i), 0.5, 1.0));
    allPoints ->push_back (osg::Vec3 (50*sin(i),-0.0, 50*cos(i)));
}

```

```

//得到所有顶点
clockGeometry->setVertexArray( allPoints.get () );
//画线
osg::DrawElementsUInt * clockBase = new
osg::DrawElementsUInt(osg::PrimitiveSet::LINE_LOOP ,0);
for (int i =0 ; i<(int)(allPoints ->size ()) ; i++)
{
    clockBase->push_back(i) ;
} ;
clockGeometry->addPrimitiveSet(clockBase);

//得到数字顶点
NumPoint ->setVertexArray (Num.get ()) ;

osg::DrawElementsUInt * PointBase = new
osg::DrawElementsUInt(osg::PrimitiveSet::LINES , 0) ;
for (int i = 0 ; i<(int)(Num ->size ()); i++)
{
    //画线
    PointBase ->push_back (i) ;
}
NumPoint ->addPrimitiveSet (PointBase) ;

//使用颜色索引
osg::TemplateIndexArray
<unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
    new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,4>;
for (int i =0 ; i<(int)(allPoints ->size ()) ; i++)
{
    colorIndexArray->push_back(i) ;
} ;

osg::TemplateIndexArray
<unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArrayNum;
colorIndexArrayNum =
    new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,4>;
for (int i =0 ; i<(int)(Num ->size ()) ; i++)
{
    colorIndexArrayNum->push_back(i) ;
} ;

```

```

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

NumPoint ->setColorArray(colors2);
NumPoint->setColorIndices(colorIndexArrayNum);
NumPoint->setColorBinding(osg::Geometry::BIND_PER_VERTEX);

return (osg::Node *) root ;

}
//创建表针从res到des
osg::Node* OSGFrame::CreateHandler(osg::Vec3 &res, osg::Vec3 &des)
{
    osg::Group * root = new osg::Group () ;
    //设置线宽为4
    osg::ref_ptr <osg::LineWidth> LineSize = new osg::LineWidth;
    LineSize ->setWidth (4.0) ;
    //打开线宽
    Root->getOrCreateStateSet()->setAttributeAndModes(LineSize.get
(),osg::StateAttribute::ON);
    //申请针结点
    osg::Geode* handleGeode = new osg::Geode();
    osg::Geometry* handleGeometry = new osg::Geometry();
    handleGeode->addDrawable(handleGeometry);
    root->addChild(handleGeode);
    osg::ref_ptr <osg::Vec3Array > allPoints = new osg::Vec3Array ;

    osg::Vec4Array* colors = new osg::Vec4Array;
    colors ->push_back (osg::Vec4f (1.0, 0.0, 0.5, 1.0)) ;
    colors ->push_back (osg::Vec4f (0.0, 1.0, 0.5, 1.0)) ;
    //把起点与终点压入
    allPoints ->push_back (res) ;
    allPoints ->push_back (des) ;
    //设置要画的顶点
    handleGeometry->setVertexArray( allPoints.get () );
    osg::DrawElementsUInt * handleBase = new
osg::DrawElementsUInt(osg::PrimitiveSet::LINE_LOOP ,0);
    for (int i =0 ; i<(int)(allPoints ->size ()) ; i++)
    {
        handleBase->push_back(i);
    } ;
    handleGeometry->addPrimitiveSet(handleBase);

```



```

//设置颜色
osg::TemplateIndexArray
    <unsigned int, osg::Array::UIntArrayType,4,4> *colorIndexArray;
colorIndexArray =
    new osg::TemplateIndexArray<unsigned int, osg::Array::UIntArrayType,4,4>;

for (int i =0 ; i<(int)(allPoints ->size ()) ; i++)
{
    colorIndexArray->push_back(i);
} ;

handleGeometry->setColorArray(colors);
handleGeometry->setColorIndices(colorIndexArray);
handleGeometry->setColorBinding(osg::Geometry::BIND_PER_VERTEX);
return (osg::Node *) root ;
}

//创建表针移动路径
osg::AnimationPath* OSGFrame::createAnimationPath(const osg::Vec3& center,
float radius, double looptime, float angle)
{ //创建path
    osg::AnimationPath* animationPath = new osg::AnimationPath;
    //设置循环
    animationPath->setLoopMode(osg::AnimationPath::LOOP);
    //设置总共关键点数
    int numSamples = 40;
    float yaw = 0.0f;
    //步长
    float yaw_delta = 2.0f*osg::PI/((float)numSamples-1.0f);
    float roll = osg::inDegrees(30.0f);
    //设置循环时间
    double time=0.0f;
    double time_delta = looptime/(double)numSamples;
    for(int i=0;i<numSamples;++i)
    {
        osg::Vec3
position(center+osg::Vec3(sinf(yaw)*radius,cosf(yaw)*radius,0.0f));
        osg::Quat
rotation(osg::Quat(roll,osg::Vec3(0.0,1.0,0.0))*osg::Quat((yaw+osg::inDegrees
(angle)),osg::Vec3(0.0,1.0,0.0)));
        //把关键点与时间压入形成路径
animationPath->insert(time,osg::AnimationPath::ControlPoint(position,rotation
));
        yaw += yaw_delta;
        time += time_delta;
    }
}

```

```

    }
    return animationPath;
}

//创建移动表针
osg::Node* OSGFrame::CreateMoveHandler(void)
{
    //得到当前的时间
    SYSTEMTIME st ;
    ::GetLocalTime (&st) ;

    //创建秒针，分针，时针
    osg::Node * second = CreateHandler(osg::Vec3 (-10.0,-0.0 , 0.0), osg::Vec3
(35,-0.0 ,0.0)) ;
    osg::Node * minute = CreateHandler(osg::Vec3 (-8.0, -0.0, 0.0), osg::Vec3 (25,
-0.0, 0.0)) ;
    osg::Node * hour = CreateHandler(osg::Vec3 (-6.0, -0.0, 0.0), osg::Vec3 (15,
-0.0, 0.0)) ;
    //利用当前系统时间初始化角度
    float animationLength = 60.0F;
    osg::Vec3 center(0.0, 0.0, 0.0);
    float radius = 0.0 ;
    float angle0 , angle1, angle2 ;
    angle0 = st.wSecond *6.0 ;
    angle1 = st.wMinute *6.0 + angle0 * (1/60.0) ;
    angle2 = st.wHour * 30.0 + angle1 * (1/60) ;
    //周期越来越长，秒针周期为60S
    osg::AnimationPath* secondPath =
createAnimationPath(center,radius,animationLength, -120.0+angle0) ;//+ angle2);
    animationLength *= 60.0f ;

    //分针为60^2
    osg::AnimationPath* minutePath = createAnimationPath (center, radius,
animationLength, -120.0+angle1); ;//+angle1);
    animationLength *=60.0f ;

    //时针为60^3
    osg::AnimationPath* hourPath = createAnimationPath (center, radius,
animationLength, -120.0+angle2) ;//+angle0);

    osg::Group* model = new osg::Group;

    osg::MatrixTransform* positioned1 = new osg::MatrixTransform;

```

```

positioned1->setDataVariance(osg::Object::STATIC);
positioned1->addChild(second);

osg::MatrixTransform* positioned2 = new osg::MatrixTransform;
positioned2->setDataVariance(osg::Object::STATIC);
positioned2->addChild(minute);

osg::MatrixTransform* positioned3 = new osg::MatrixTransform;
positioned3->setDataVariance(osg::Object::STATIC);
positioned3->addChild(hour);

osg::PositionAttitudeTransform* xform1 = new osg::PositionAttitudeTransform;
xform1->setUpdateCallback(new
osg::AnimationPathCallback(secondPath,0.0,1.0));
xform1->addChild(positioned1);

osg::PositionAttitudeTransform* xform2 = new osg::PositionAttitudeTransform;
xform2->setUpdateCallback(new
osg::AnimationPathCallback(minutePath,0.0,1.0));
xform2->addChild(positioned2);

osg::PositionAttitudeTransform* xform3 = new osg::PositionAttitudeTransform;
xform3->setUpdateCallback(new
osg::AnimationPathCallback(hourPath,0.0,1.0));
xform3->addChild(positioned3);

model->addChild(xform1);
model->addChild(xform2);
model->addChild(xform3);
return model;
} //OSGFrame.cpp完

```

到此这一章就全部结束了，制做钟表的代码虽然有些笨拙，但是反映了一个重要的思想，就是使用 PATH 的思想，理解这个程序有利于第四章的学习！

第三章 场景漫游

按照自己的需要做出符合逻辑的矩阵操作器一直是初学者的难点，OSG 为矩阵操作器提供了虚的接口 `osgGA::MatrixManipulator`，`osgViewer` 中的 `traceball` UFO 等操作器皆是从它派生而来，做自己的操作器一般需要重载这个接口，操作器中可以添加很多的功能：例如本章要学习的碰撞检测，`pick` 取点，旋转模型以适合各种视图甚至于显示当前位置等等，很多功能都可以通过矩阵操作器来完成，原因是操作器中有事件响应以及矩阵操作，这本身就是场景漫游的两个十分重要的核心的处理。

本章还会介绍如何使用动态链接库，把功能做成动态链接库最直接的好处是可以在不同的项目代码中，不必编写相应代码就可以实现动态链接库的功能，一劳永逸，类似于控件！

本章还将介绍 DLL 编程的方法，编译出一经编译可以四处使用的动态链接库，关于动态链接库会在下面做详细的介绍。

本章过后读者可以掌握：如何制作矩阵操作器，简单的碰撞检测，熟练的编写 DLL，其中关于漫游的动态链接库使用非常广泛，在不同的实验程序中，只需要添加短短的几行代码就可以实现漫游，是非常方便的！

3.1 如何编写动态链接库

这一节我们来学习如何制作与使用 DLL。

这一节我们来主要的介绍一下如何来编写一个动态链接库，其实 OSG 本身的所有的功能都是给作者提供一个链接文件(.LIB)，一个头文件(.H)，一个库文件(.DLL)。

对于动态链接库的更多内容，需要查阅相关资料，这里只是基本应用，入门是本书的宗旨。在本章过后的各章例子均采用 DLL 功能块编写，所以这一节如果不熟悉，需要先看一下。

3.1.1 OSG 中的 DLL

我们来看一下程序使用自己制作的 DLL 的过程，刚才说过 DLL 必须包含 3 个要素 LIB,.H,.DLL，发布版本的程序，只需要包含 DLL，即可。

应该注意的是：我们在使用 OSG 时，如果我们的编译器版本与 OSG 版本不一致，打个比方来说，编译器版本是：VS2003，而 OSG 的版本是 OSG1.2(VS2005 配套使用)，那么在编写程序的过程中，也许代码没有语法错误，但是在运行时会发生错误，这种错误一般为无法找到某种程序的切入点在*.DLL 上。或者是已经编译完成的程序，因为系统崩溃或是更新，又安装了不同版本的 OSG，比如 VS2003 编译完成的 OSG 程序，现在要使用 OSG1.2 的库，那么也会报同样的错：无法找到切入点。

由于 OSG 自发布以来，版本过多，且 VS 的版本也不少，在早期由于人们习惯于使用 VS6.0，在上面留下了很多重要的程序模块、功能控件，这些模块的源代码可能已经丢失，或者这些模块干脆是从网上下得，当 VS 版本升级时，便不能再适用，所以来讲东西更新过快不一定是好事，有时候反而让人措手不及。

在 OSG 中解决办法非常简单，重新编译链接一遍以前的程序就好了，前提是你必须得有源代码和改掉因为编译器版本更新所带来的语法错误，一般来讲 VS 是向后兼容的，VS2005 可以正确的编译以前所有版本的且在以前已经编译通过的源程序。

下面我们来看一下动态链接库与可执行程序的关系：

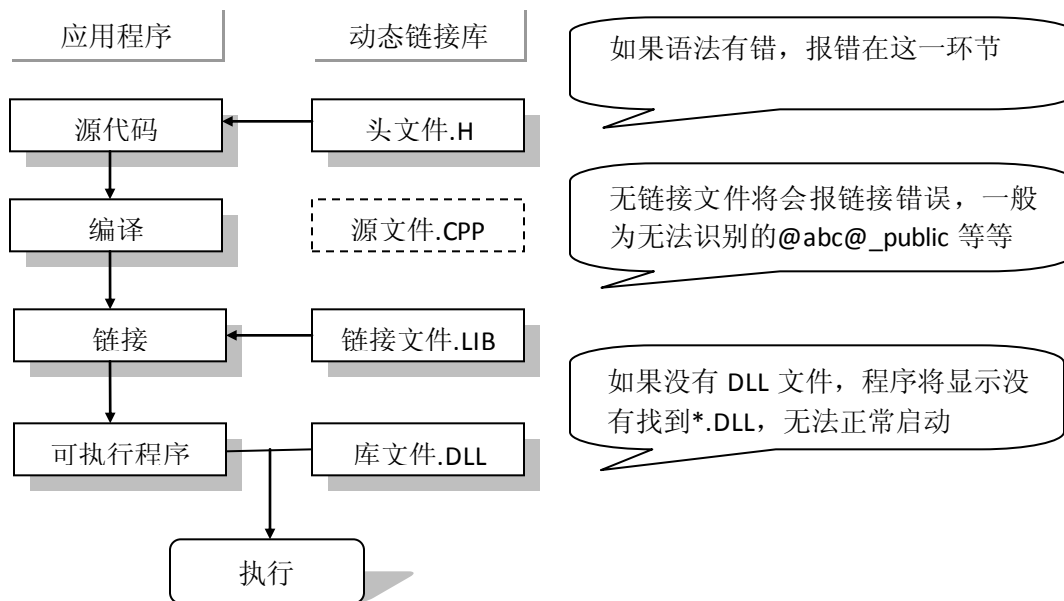


图 3.1

下面来说一下关于上图，在源代码中的错误一般是因为没有包含头文件.H 引起的，VS2005 中，如果没有包含头文件，那么将会报无法识别的标识符，在写代码时一般没有关于参数的智能提示，所以一般来讲，代码有错，可以轻易的在写代码时就可以找出来。

包含头文件后需要在项目中导入.LIB 文件，以前曾介绍过三种方法来导入.LIB 文件，一般来讲这种错误非常常见，如果没有导入.LIB 文件，在链接阶段会报无法解析的项目符号，一般与你的某些函数名类似，之所以是类似原因是 VS 在生成这些文件时，自动在其前后加了一些特殊符号，这样可以防止重名的函数发生紊乱现象。一旦发现无法解析的项目符号错误时，立即查找是否加全了所有需要的.LIB 文件。

下面来看一下程序在执行过程中使用显示当前坐标的动态链接(ShowPosition.dll)库来说明应用程序是如何使用动态链接库的：

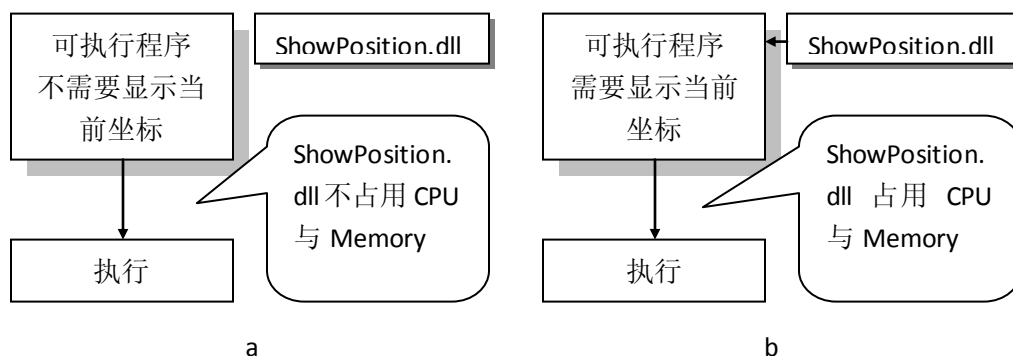


图 3.2

从图中可以看出，当程序执行时，是动态的加载动态链接库，当不使用某些功能时，这部分内存将不会被占用。然而使用一个程序，则在编译运行时，这部分功能已经加入到程序中去，当程序在加载时空间就已经预留，所以动态链接库是动态加载程序的，比之普通大混沌代码更为合理。

在 OSG 早期，在中国使用初学者使用 OSG 是这样一个过程，首先把源代码全部下下来，然后全部编译与运行一遍，看看有没有错，这个时间一般是一个下午，如果因为你的编译器配置有误，比如没有加 INCLUDE 路径等等，那么又得重新编译一遍，又是一个下午。作者在开始使用 OSG 时，光编译程序用了两天时间。

在编译完成之后，把所有的 LIB 文件拷到一个地方，所有的 DLL 文件拷到一个地方，在 PATH 中加入路径，等等，非常痛苦。其实这就是 DLL 的使用过程，第一步先要编代码：如 OSG 的源码，第二步编译执行生成 LIB 与 DLL 文件，第三步在别的应用程序中加以调用。

OSG 中共有 125 个 DLL 文件，其中调试版(带 D)与发布版(不带 D)的几乎各占一半，如果程序中没有具体的功能，在发布应用程序时不必要把这些文件全部带上，一般只带不带 D 的版本，去掉没用的 DLL。比方说你的程序只打开 FLT 文件，那么类似于 3DS.DLL 这样的就不需要再加载了。

3.1.2 编写 TravelManipulator.DLL

现在我们来按步骤编写最常用的 TravelManipulator.dll,这个功能块，具有的功能是场景漫游，设置速度，在漫游时与一般网络游戏中的漫游方式基本相同：

WASD 分别为：向前后左右走，鼠标可向任意方向拖动场景，

Up down left right: 为向前，左转，向后，右转，这套操作完全是仿枪击游戏反恐的一套操作。Home end 上向下移动

编写矩阵操作器需要重载 osgGA::MatrixManipulator，如何理解呢？这是属于 C++语言的功底，不过下面我们来看一下重载在 OSG 中的描述，如图所示：

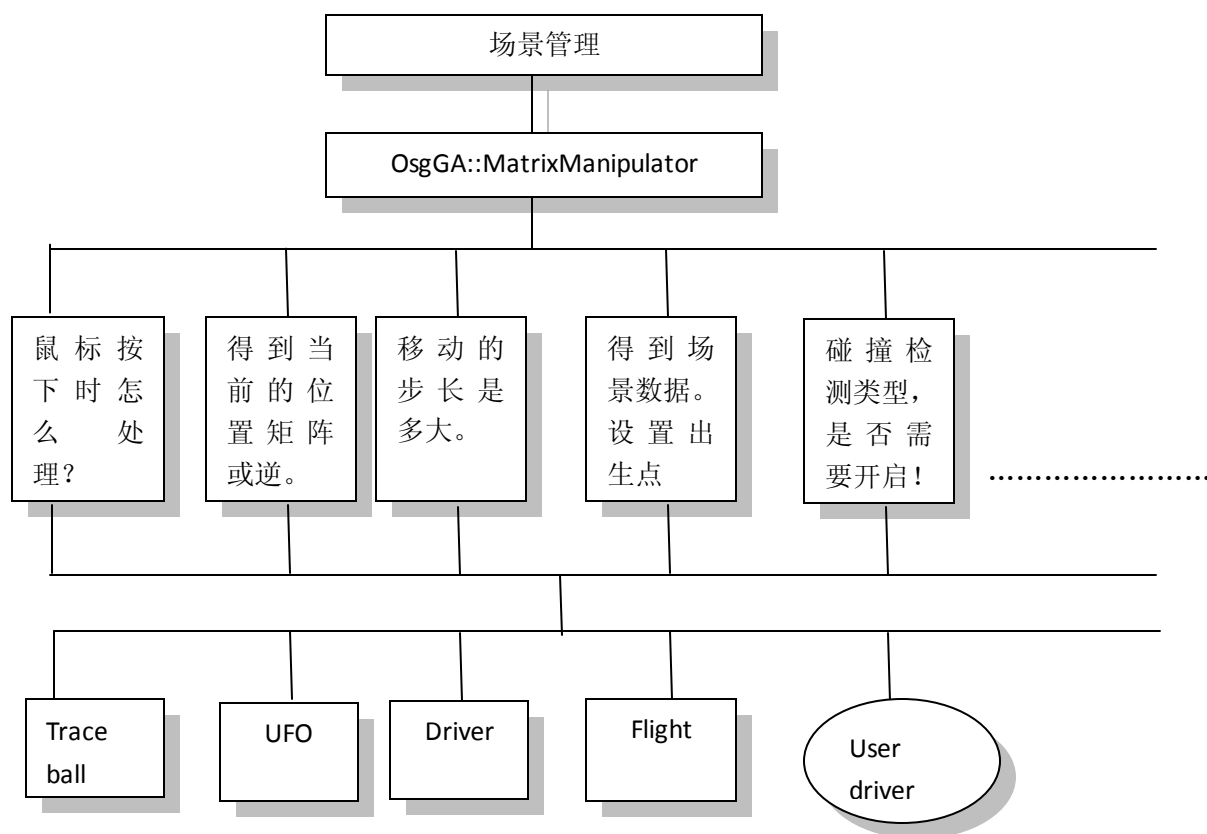


图 3.3

从图中可以看出，OSG 为场景的管理提供了一个虚的接口，这样的一个接口包括了场景管理经常有的操作，而用户可以继承这个接口，从而以自己的方式来各个完成这些操作。这就像饭店这个概念，无非是用来吃饭的，但是分为中餐店与西餐店，但是都是用来吃饭的，只是吃的方式不同而已。这取决于自己的盐味是否大，如果喜欢吃暴辣菜，那么就到中餐馆，讲营养学到西餐馆。矩阵操作器就比如是要管饱这个功能，至于如何做，要由下面具体的继承对象来完成。下面我们来看一下 osgGA::MatrixManipulator 都有哪

些操作:

类 **osgGA::MatrixManipulator**

virtual const char * className () const :类名

//下面两个函数经常用于下面的操作, 比如在 **Driver** 或是在别的操作器中, 如何来设置越来越变大的速度呢? 比如设置成点一下速度加 0.1, 由于场景的大小不同, 在房间的 0.1 与广场上的 0.1 不同, 因此需要根据场景的半径来设置, 每一次增加半径的百分比来完成加速度或是初始速度的设置, 如果比例尺相同, 那么半径大的就应该速度增加的大吗? 所以要使用缩放

virtual void setMinimumDistance (float minimumDistance) : 场景缩放比率

float getMinimumDistance () const 得到场景缩放比率

//下面的函数用于更新坐标系, 指出哪边是东西, 哪边是南北

virtual void setCoordinateFrameCallback (CoordinateFrameCallback *cb)坐标系更新

CoordinateFrameCallback * getCoordinateFrameCallback () 得到当前更新坐标系

const CoordinateFrameCallback * getCoordinateFrameCallback () const 常量指针版本

osg::CoordinateFrame getCoordinateFrame (const osg::Vec3d &position) const 得到坐标系

//得到/设置向量或矩阵或矩阵,

osg::Vec3d getSideVector (const osg::CoordinateFrame &cf) const

osg::Vec3d getFrontVector (const osg::CoordinateFrame &cf) const

osg::Vec3d getUpVector (const osg::CoordinateFrame &cf) const

virtual void setByMatrix (const osg::Matrixd &matrix)=0

virtual void setByInverseMatrix (const osg::Matrixd &matrix)=0

virtual osg::Matrixd getMatrix () const =0

virtual osg::Matrixd getInverseMatrix () const =0

//下面两个函数经常用在 **SceneView** 中

virtual osgUtil::SceneView::FusionDistanceMode getFusionDistanceMode () const

virtual float getFusionDistanceValue () const

//主要用于场景的局部控制

void setIntersectTraversalMask (unsigned int mask)

unsigned int getIntersectTraversalMask () const

//场景数据

virtual void setNode (osg::Node *)

virtual const osg::Node * getNode () const

virtual osg::Node * getNode ()

//设置出生点

virtual void setHomePosition (const osg::Vec3d &eye, const osg::Vec3d ¢er, const osg::Vec3d &up, bool autoComputeHomePosition=false)

//得到出生点

virtual void getHomePosition (osg::Vec3d &eye, osg::Vec3d ¢er, osg::Vec3d &up) const

//自动计算出生点, 比如 **Driver** 操作器

virtual void setAutoComputeHomePosition (bool flag)

bool getAutoComputeHomePosition () const

//计算出生点与事件操作

```

virtual void  computeHomePosition ()
virtual void  home (const GUIEventAdapter &, GUIActionAdapter &)
virtual void  home (double)
virtual void  init (const GUIEventAdapter &, GUIActionAdapter &)
virtual bool  handle (const GUIEventAdapter &ea, GUIActionAdapter &us)
virtual void  accept (GUIEventHandlerVisitor &v)

```

熟悉了所要完成的功能与基类后，现在来编写并测试程序。

示例九：编写 TravelManipulator.DLL

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：TravelManipulator 在应用程序设置中选 DLL 与空项目选项。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB: OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib osgUtild.lib

第三步：菜单项目→添加类，类名填入 TravelManipulator,基类填入：osgGA::Manipulator 点击确定。

在文件 TravelManipulator.h 中加如如下代码，可以敲进去，也可以使用 VS 进行添加：关键地方会给予注释：

```

TravelManipulator.h
//by FreeSouth for OSG FreeS
//本来TRAVEL_DLL, 需要在.H与.CPP中都需要定义, 在.H文件中定义成//_declspec(dllexport),
在.CPP文件中定义成_declspec(dllimport)
//一个为导入, 一个为导出, 如果在CPP文件中已经定义了TRAVEL_DLL, 那么这三行预编//译就相当于
不起做用, 如果没有定义TRAVEL_DLL, 则定义成导出, 因此可以做为DLL;
//的头文件来使用
#ifdef TRAVEL_DLL
#else
#define TRAVEL_DLL _declspec(dllimport)
#endif
#include <osgProducer/Viewer>
#include <osgGA/MatrixManipulator>
#include <osgUtil/IntersectVisitor>
#include <osg/LineSegment>
#include <osg/Point>
#include <osg/Geometry>
#include <vector>
class TRAVEL_DLL TravelManipulator :public osgGA::MatrixManipulator
{
//构造函数
public:
    //构造函数
    TravelManipulator ();

    //析构函数

```



```

~ TravelManipulator (void);
// 把漫游加入到场景之中
static TravelManipulator * TravelScene(osg::ref_ptr<osgProducer::Viewer>
viewer);

private:
//viewer
osg::ref_ptr<osgProducer::Viewer>    m_pHostViewer;

unsigned int        m_nID;
float               m_fMoveSpeed;
osg::Vec3           m_vPosition;
osg::Vec3           m_vRotation;

public:
//判断鼠标左键是否按下，可用来旋转窗口
BOOL m_bLeftButtonDown;
//鼠标的x值
float m_fpushX;
//激活所选的漫游方式
void Active(void);
// 设置矩阵
virtual void setByMatrix(const osg::Matrixd& matrix);
// 设置逆矩阵
virtual void setByInverseMatrix(const osg::Matrixd& matrix);
//得到矩阵
virtual osg::Matrixd getMatrix(void) const;
// 得到逆矩阵
virtual osg::Matrixd getInverseMatrix(void) const ;
// 主要事件控制器
virtual bool handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& us);
// 屏幕角度，经数千次的测试，2.5是最佳角度，转向不会太快，这个角意思为鼠标从屏幕最左端
拖到最右边场景所转的角度
float M_fAngle;
// 位置变换函数
void ChangePosition(osg::Vec3& delta);
//碰撞检测是否开启
bool m_bPeng;
//鼠标的y值
float m_fpushY;
//得到速度
float getSpeed() ;
//设置速度

```

```

void setSpeed(float &) ;
//设置当前或出生位置
void SetPosition(osg::Vec3 &position) ;
//得到当前位置
osg::Vec3 GetPosition() ;
} ;//TravelManipulator结束

```

现在来看一下 TraverManipulator.CPP 中的实现:

```

TravelManipulator.cpp
//定义导出, 与头文件中的TRAVEL_DLL必须同名对应
#define TRAVEL_DLL _declspec(dllexport)
#include "TravelManipulator.h"
//构造函数, 分别初始化x,y鼠标值与碰撞检测是开启的
TravelManipulator::TravelManipulator(): m_fMoveSpeed(1.5f)
, m_bLeftButtonDown(FALSE)
, m_fpushX(0)
, M_fAngle(2.5)
, m_bPeng(true)
, m_fpushY(0)
{
    m_vPosition = osg::Vec3(15.0f, -130.0f, 3.0f);
    m_vRotation = osg::Vec3(osg::PI_2, 0.0f, 0.0f);
}
TravelManipulator::~TravelManipulator()
{
}
// 把漫游加入到场景之中, 静态成员
TravelManipulator * TravelManipulator::TravelScene(osg::ref_ptr
<osgProducer::Viewer> viewer)
{
    //可以获得并使用系统提供的浏览方式, 这个静态函数用于初始化与激活, 在类静态函数中初始化,
    这属于C++的知识, 把析构函数设置为保护就是……, 自己看吧
    TravelManipulator* camera = new TravelManipulator;
    //得到像机的ID
    camera->m_nID = viewer->addCameraManipulator(camera);
    //TEMP
    camera->m_pHostViewer = viewer;
    return camera;
}
//激活所选的漫游方式
void TravelManipulator::Active(void)
{
    m_pHostViewer->selectCameraManipulator(m_nID);
}

```

```

// 虚函数
void TravelManipulator::setByMatrix(const osg::Matrixd& matrix)
{
}

// 得到逆矩阵
void TravelManipulator::setByInverseMatrix(const osg::Matrixd& matrix)
{
}

osg::Matrixd TravelManipulator::getMatrix(void) const
{
    //得到旋转后的矩阵,其实就是当前点的视口矩阵
    osg::Matrixd mat;
    mat.makeRotate(m_vRotation._v[0], osg::Vec3(1.0f, 0.0f, 0.0f),
        m_vRotation._v[1], osg::Vec3(0.0f, 1.0f, 0.0f),
        m_vRotation._v[2], osg::Vec3(0.0f, 0.0f, 1.0f));
    return mat * osg::Matrixd::translate(m_vPosition);
}

// 得到逆矩阵, 其实就是当前视口位置的逆矩阵
osg::Matrixd TravelManipulator::getInverseMatrix(void) const
{
    osg::Matrixd mat;
    mat.makeRotate(m_vRotation._v[0], osg::Vec3(1.0f, 0.0f, 0.0f),
        m_vRotation._v[1], osg::Vec3(0.0f, 1.0f, 0.0f),
        m_vRotation._v[2], osg::Vec3(0.0f, 0.0f, 1.0f));
    return osg::Matrixd::inverse(mat * osg::Matrixd::translate(m_vPosition));
}

// 主要事件控制器
bool TravelManipulator::handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& us)
{
    //得到鼠标在屏目上的x,y坐标,一般左上为(0,0),右下为(1,1),用于鼠标控制模型的移动,转向等
    float mouseX = ea.getX();
    float mouseY = ea.getY();
    switch (ea.getEventType())
    {
    case (osgGA::GUIEventAdapter::KEYDOWN):
        {
            if (ea.getKey() == 0x20) //' '
                {
                    //如果是空格,只需要更新就可以了,不需要再计算回到原点了,VIEWER中的此操作被屏蔽掉了
                    us.requestRedraw();
                    us.requestContinuousUpdate(false);
                    return true;
                }
        }
    }
}

```

```

}
if (ea.getKey() == 0xFF50) //home, 如果是HOME键
{
    //则需要向上移动, 此函数来根据向量改变当前坐标
    ChangePosition(osg::Vec3 (0, 0, m_fMoveSpeed)) ;
    return true;
}
if (ea.getKey() == 0xFF57) //end
{
    //end键向下移动
    ChangePosition(osg::Vec3 (0, 0, -m_fMoveSpeed)) ;
    return true;
}
if (ea.getKey() == 0x2B) //+
{
    //+ 为速度加1.0
    m_fMoveSpeed += 1.0f;
    return true;
}
if (ea.getKey() == 0x2D) //-
{
    //-为速度减1.0, 如果速度小于1., 则等于1
    m_fMoveSpeed -= 1.0f;
    if (m_fMoveSpeed < 1.0f)
    {
        m_fMoveSpeed = 1.0f;
    }
    return true;
}
if (ea.getKey() == 0xFF52 || ea.getKey() == 0x57 || ea.getKey() == 0x77)
{
    //up键与, w/w都将使其向前移动, 在后面解释为什么需要一个角度
    ChangePosition(osg::Vec3 (0, m_fMoveSpeed *
sinf(osg::PI_2+m_vRotation._v[2]), 0)) ;
    ChangePosition(osg::Vec3 (m_fMoveSpeed *
cosf(osg::PI_2+m_vRotation._v[2]), 0, 0)) ;
    return true;
}
if (ea.getKey() == 0xFF54 || ea.getKey() == 0x53 || ea.getKey() == 0x73)
{
    //down与s/s键都将向下移动, 屏蔽掉系统的帧速以后, 自己可以再做一个, 在下一章将介绍HUD
    ChangePosition(osg::Vec3 (0, -m_fMoveSpeed *
sinf(osg::PI_2+m_vRotation._v[2]), 0)) ;
    ChangePosition(osg::Vec3 (-m_fMoveSpeed *
cosf(osg::PI_2+m_vRotation._v[2]), 0, 0)) ;

```

```

        return true;
    }
    if (ea.getKey () == 0x41 || ea.getKey () == 0x61)
    {
        //点下a/A键时, 会向左移动
        ChangePosition(osg::Vec3 (0, m_fMoveSpeed *
cosf(osg::PI_2+m_vRotation._v[2]), 0)) ;
        ChangePosition(osg::Vec3 (-m_fMoveSpeed *
sinf(osg::PI_2+m_vRotation._v[2]), 0, 0)) ;
        return true;
    }
    if (ea.getKey () == 0x44 || ea.getKey () == 0x64)
    {
        //点下D键时, 会向右移动
        ChangePosition(osg::Vec3 (0, -m_fMoveSpeed *
cosf(osg::PI_2+m_vRotation._v[2]), 0)) ;
        ChangePosition(osg::Vec3 (m_fMoveSpeed *
sinf(osg::PI_2+m_vRotation._v[2]), 0, 0)) ;
        return true;
    }

    if (ea.getKey () == 0xFF53)
    {
        //right键会向右旋转
        m_vRotation._v[2] -= osg::DegreesToRadians(M_fAngle);
    }
    if (ea.getKey () == 0xFF51)
    {
        //left键会向左旋转
        m_vRotation._v[2] += osg::DegreesToRadians(M_fAngle);
    }
    if (ea.getKey () == 0x48 || ea.getKey () == 0x68)
    {
        //H/h键减小增加旋转角度
        M_fAngle -= 0.2 ;
        return true ;
    }
    if (ea.getKey () == 0x47 || ea.getKey () == 0x67)
    {
        //G/g键增加旋转角度
        M_fAngle += 0.2 ;
        return true ;
    }
}

```

```

        return false;
    }
    case (osgGA::GUIEventAdapter::PUSH): //左键按下
        if ( ea.getButton () == 1)
        {
            m_fpushX = mouseX ;
            m_fpushY = mouseY ;
            m_bLeftButtonDown = true ;
        }
        return false ;
    case (osgGA::GUIEventAdapter::DRAG):
        if ( m_bLeftButtonDown)
        {
            //在拖动时改变角度，只有在此时与按下LEFT与RIGHT时，才会改变角度
            m_vRotation._v[2] -= osg::DegreesToRadians(M_fAngle *
(mouseX-m_fpushX));
            m_vRotation._v[0] += osg::DegreesToRadians(1.1*(mouseY-m_fpushY)) ;
            //控制向下看时不至于不符合实际一直转圈看到后面
            if (m_vRotation._v [0] >= 3.14)
                m_vRotation._v [0] = 3.14 ;
            if (m_vRotation._v [0] <= 0)
                m_vRotation._v [0] = 0 ;
        }
        return false ;
    case (osgGA::GUIEventAdapter::RELEASE):
        if ( ea.getButton () == 1)
        {
            m_bLeftButtonDown = false ;
        }
        return false ;
    default:
        return false;
    }
}
// 位置变换函数
void TravelManipulator::ChangePosition(osg::Vec3& delta)
{
    if (m_bPeng)
    {
        //看新值与旧值之间的连线是否与模型有交点!
        osg::Vec3 newPos = m_vPosition + delta;
        osgUtil::IntersectVisitor iv;

        osg::ref_ptr<osg::LineSegment>line = new osg::LineSegment(newPos,

```

```

        m_vPosition);
    osg::ref_ptr<osg::LineSegment> lineZ = new osg::LineSegment(newPos +
    osg::Vec3(0.0f, 0.0f, m_fMoveSpeed),
        newPos - osg::Vec3(0.0f, 0.0f, m_fMoveSpeed));
    iv.addLineSegment(lineZ.get());
    iv.addLineSegment (line.get());
    m_pHostViewer->getSceneData()->accept(iv);
    if (!iv.hits())
    {
        m_vPosition += delta;
    }
}

else
    m_vPosition += delta;
}

//设置行走速度
void TravelManipulator::setSpeed (float &sp)
{
    m_fMoveSpeed = sp ;
}

//得到行走速度
float TravelManipulator::getSpeed ()
{
    return m_fMoveSpeed ;
}

```

终于完了，现在编译运行，如果成功会出现如下页面：



图 3.4

这是测试控件用的东西，类似于测试 ActiveX 的容器。我们关掉，不予理会。下面我们先来测试一下这个 Travel 操作器是否好用。

3.1.3 测试 TravelManipulator.DLL

下面我们来建立一个只需几行代码就可以完成漫游功能的程序，我们将在 WIN32 控制台下来测试该控件，在 MFC 与别的框架下加的代码是相同的 WIN32 更加简明扼要。

第一步：菜单文件→新建→项目→WIN32/WIN32 控制台应用程序，项目名取 test，在应用程序设置中点击空项目后完成。

第二步：菜单项目→test 属性→配置属性→链接器→命令行中添加：OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib osgUtild.lib **TravelManipulator.lib** 注意，要添加刚才我们设计的 lib。

第三步：把刚才 TravelManipulator 项目以下三个文件拷贝到该项目目录中：TravelManipulator.h TravelManipulator.lib TravelManipulator.DLL，（该项目目录是指放有该项目源文件的目录）

第四步：在解决方案视图中，对源文件文件夹点右键→添加→新建项→代码/CPP 文件，文件名输入 main，然后在该文件中添加下面内容：

```
Main.cpp
//by FreeSouth for OSGFreeS
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include "TravelManipulator.h"
Int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    viewer->setSceneData (node);
    TravelManipulator * travel = TravelManipulator::TravelScene (viewer) ;
    travel ->Active () ;
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
    viewer->sync();
    return 0;
} //main.cpp结束
```

注：ceep.ive 文件在光盘 ive 文件夹中。任何模型都可以，这个模型是个楼盘模型，比较大。

TravelManipulator * travel = TravelManipulator::TravelScene(viewer) ;其中 TravelScene 是类中静态成员函数，众所周知，静态成员函数其实是不属生类成的。在静态函数中分配类对象是一种防止过多申请滥用类的保护，如果把析构函数申请成保护型的，那么就不能直接的申请类对象，而必须通过类中已有的类对象引出类对象，此设计模式在编程时经常使用，具有安全性高的特点！

把 LIB .H DLL 三个文件保存好，可以使用在所有的 OSG 程序中，以后我们所讲的功能大部分都将使用 DLL 实现，使用 WIN32 控制，这样更具有清晰析性，在 WIN32 中与在 MFC 中，是一样的，从第二章开始我们对程序所做的改动已经几乎不涉及 MFC，只是对 OSGFrame 这个类进行的改动，因此用什么框架都是无所谓。可以自己把这个 DLL 用在自己以前做过的程序中试一下，效果会非常好的！

运行效果：



图 3.5

下面我们来讲一下一个实用技术，它在 `TravelManipulator.dll` 中被运用。

3.2 碰撞检测

碰撞检测一直是人们的关心与感兴趣的话题。碰撞检测几乎充斥于图形学所有的行业，在游戏编程中如何判断魔法是否打中敌人，台球发球后的冲撞是否符合实际，传奇中的野蛮冲撞是否可以穿墙，CS 中为什么会有透视眼的作弊器，在仿真行业中导弹击中目标，交通事故两辆车冲撞的合理模拟，古代建筑在建造过程中的力的依赖。这都把焦点聚在碰撞检测这个焦点上。

3.2.1 最简单的碰撞检测

首先我们先来看一下我们这个程序做的一个几乎是最简单的碰撞检测：

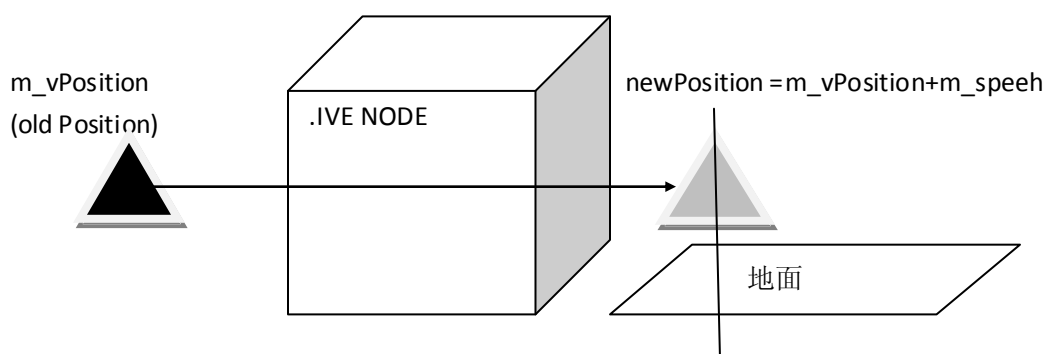


图 3.6

`TravelManipulator.dll` 中用到的就是如图所示的原理，黑三角形代表没有移动之前的位置，控制移动的函数是 `ChangePosition(osg::Vec3&delta)`，参数意思是要移动的相对于当前点的增量，在黑三角形没有移动时该函数在计算时先假设一点 `newPosition` 为移动后的点，而后通过连接这两个点，而后通过判断与场景的模型是否有交点来判定这个移动可不可以执行，如图所示，两者之间有个大盒子，是穿不过去的，所以只有保持

在原地。就算没有这个盒子，移动后的新点又与地面在某种程序上有一个交点，这证明移动是不可行的。这可以防止用户穿过地板到达地下去。

在这里要介绍一下 OSG 中做碰撞检测用到的主要的工具：

```
osg::LineSegment 表示一个线段的类，包括一个起点一个终点构成一条线段。 如：
osg::ref_ptr<osg::LineSegment> line = new osg::LineSegment(newPos, m_vPosition);
osgUtil::IntersectVisitor 是接受线段的类，通过 iv.addLineSegment (line.get());来添加一条线段到列表当中。
m_pHostViewer->getSceneData()->accept(iv);把线体段队列加入到场景中，使用 iv.hits()来判断是否有碰撞。有碰撞则返回 TRUE，无碰撞则返回 FALSE！
osgUtil::IntersectVisitor::HitList hlist; 可以得到相交的点的 具体位置。从而计算出距离。
```

碰撞检测所存的问题：

从上图可以看出，如果某个墙，恰巧它有一个小洞，而此洞恰好有此线段穿过，那么就可能穿过去了，再说地面，后面的灰三角形明明没有接触到地面，却已经产生焦点。这就是碰撞检测存在的问题之一：**精度**。

再说两球在空间中相撞，撞到后两球该如何行动呢？在没有考虑任何因素的情况下可以运用能量守恒加动量守恒以及牛顿第二定律算出碰撞后的轨迹，然而现实中往往有空气阻力，球表面光滑度，空气密度，是否完全碰撞，是否发生形变等等，这就是碰撞检测存在的问题之二：**仿真度**。

这两个问题到现在为止，也没有太完美的算法，但是这就像放电影一样，对于肉眼迟钝的分辨率已经可以轻松的蒙骗过去，下面来介绍碰撞检测经常用到的几种方法。

3.2.2 常用的碰撞检测方法

OSG 玩家最关心的莫属**小车爬坡的算法**了，小车根据地形来行走，一般来讲性价比最高的算法是这样的。

从小车的四个轮子（几个轮子取决于精度，最简单的是一个点）中发出向下的一条射线，从原理上讲应该计算小车射线的出发点与地面的交点距离是否为正，如果为正则说明小车并未紧贴地面，如果为负说明小车已经陷入地中，直至该值的模<precision（某精度）。

如下图所示：

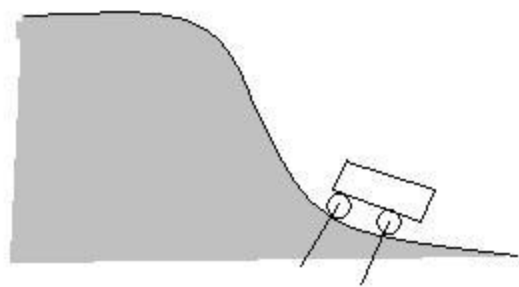


图 3.7

当图中射线与地面的交点的距离小于精度，那么说明小车在理想的范围内是在地面上的。在 OSG 中

实现这个算法很容易，读者可以自己去实践，也可以参照 `osgGA::DriverManipulator` 中的一些算法。其实只要利用好 `osg::LineSegment` `osgUtil::IntersectVisitor` `osgUtil::IntersectVisitor::HitList` 这三个类的使用方法，做出一个想做的碰撞检测算法在 OSG 中是非常简单的，因为它避开了最难的问题，如何检测是否有碰撞的。

3.2.2 空间划分 BSP 树与八叉树

包围体：

一般来讲对一个物体很难精确的进行描述，例如一把气枪，一个兔子，等等，如果直接拿原物体去进行计算，计算量会出奇的大。这样就需要确定一个包围体把其近似成规则的图形或图元，一般来讲对于从中心点到最远点和最近点变化不大的物体可以使用圆形，对于人体则可以使用圆柱形，对于楼盘可以使用长方体，而精度最高应用最为广泛的是物体凸包，一般来讲需要对物体的明显特征进行处理，比如一个人拿着一个长十米的木棒，你当然不能把人看成圆柱体，也不能看成一个大圆球，因为木棒很细，这样就不精确，于是凸包应运而生。当一个场景中包围体很多的时候，往往需要对其进行一般理论上的组织，一般会使用树（树是个极为优良的建模工具）。

下面来举一个包围体的例子，如图：

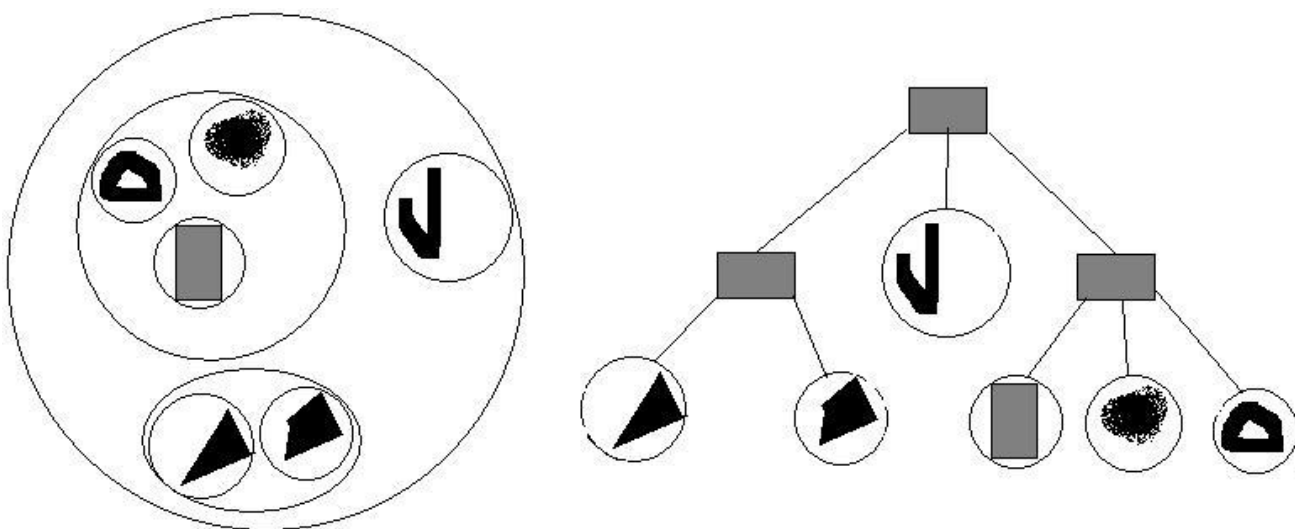


图 3.8

如图所示，在左图的场景中，所有的物体都被圆形包围，该场景又被划分为右面的层次树，树的层次越低则需要总遍历的次数就越少，但是在每一次上的开销就越大，这要看场景具体的情况来决定。一般在碰撞过程中，如何组织场景变得特别重要，下面介绍两种最常用的空间划分树：它们共同结点是子结点包含整个场景，叶子结点并不相互重叠。下面介绍的这两种树在计算机图形界大名鼎鼎。

BSP 树

BSP 树分为两种，轴对齐 BSP 树与多边形 BSP 树。BSP 树的思想来源于二叉树，它是把空间均分成两个部分。

轴对齐 BSP 树允许在空间的任何位置对空间一分为二，而不必要非得在中点，如图所示：

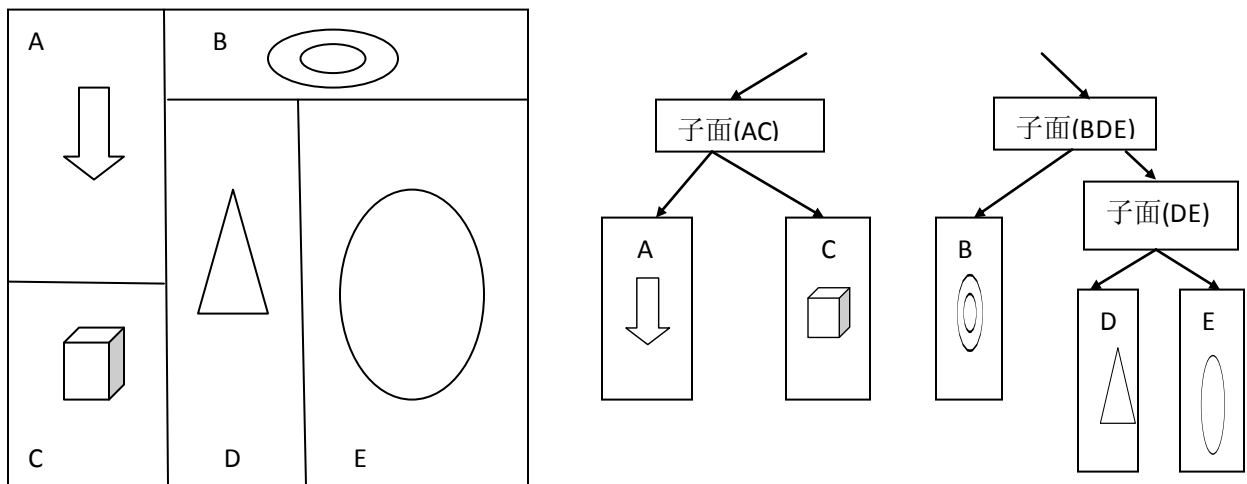


图 3.9

多边形对齐 BSP 树则使用的是多边形的边来进行划分：随便选取一个多边形，从而把空间分成两部分，各部分再进行细分：

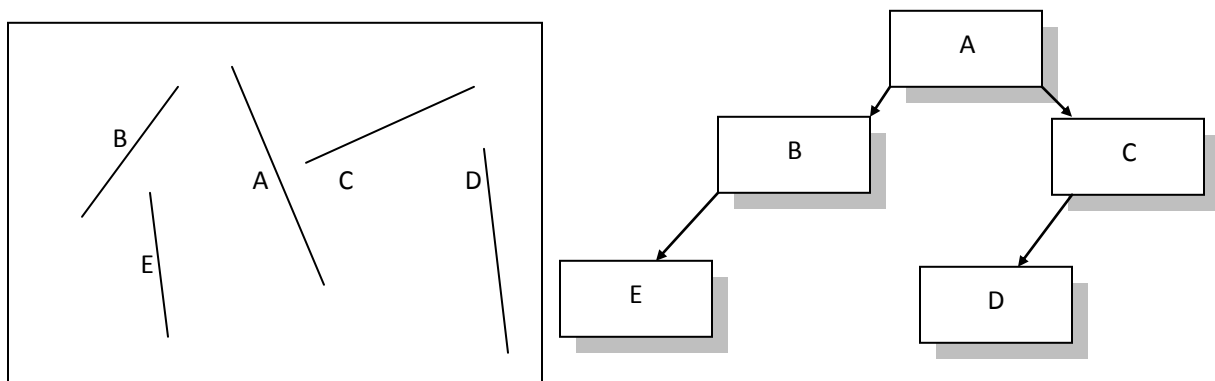


图 3.10

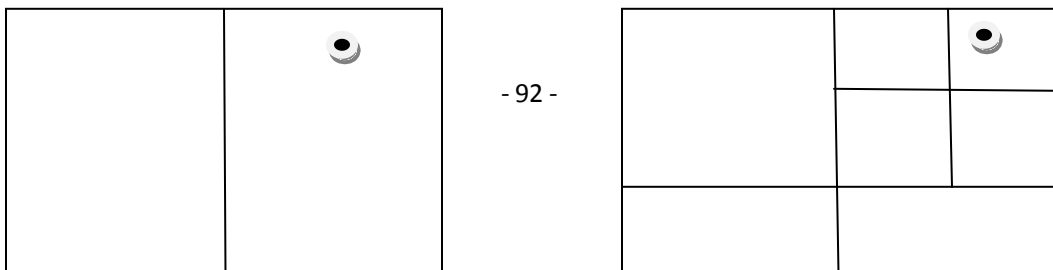
八叉树

八叉的使用在碰撞检测上严格来讲是比 BSP 要多，在早期的图形学书籍中介绍八叉树做为碰撞检测的算法几乎达成一种共识。

八叉树对场景的划分是递归的过程，严格的说，先把场景一分为四，看看与此四个哪一个有交点，如果都有交点就再分别一分为四，看看对这十六块哪个有交点，对有交点的块，再一分为四，直至找到符合精度满意的交点位置或物体为止。

下面以图示来说明划分的过程：

如图 3.11



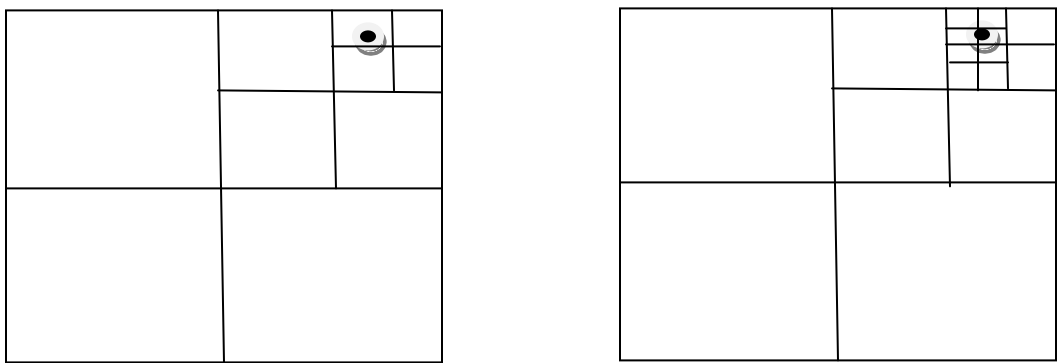


图 3.11

在 OSG 中，关于碰撞的检测，内部已经实现，读者要做的就是实现自己所要的精度的碰撞检测，如上一节做的那个碰撞检测，虽然精度很差，但是反映了做碰撞检测的一般方法，下面的部分需要读者自己去实践。

3.3 项目协作

一般而言，在一个解决方案下需要不同的项目进行协作，今天我们学习了动态链接的知识以后，可以轻松的实现同一解决方案下的多个项目。

把上面的编写 TravelManipulator.dll 与 test 项目合并成一个解决方案中，这样在修改代码与整体控制上要好的多。

我们要先创建主项目，别的项目是依赖在此之上，显然，test 是主项目，而 TravelManipulator 是辅助其完成某项功能的。

现在我们来一步一步的创建：

第一步：按照 3.1.3 的步骤创建 test 项目。

第二步：按照 3.1.2 的步骤来创建 TravelManipulator 项目，但是在开始时需要选择**添入解决方案**：如图：

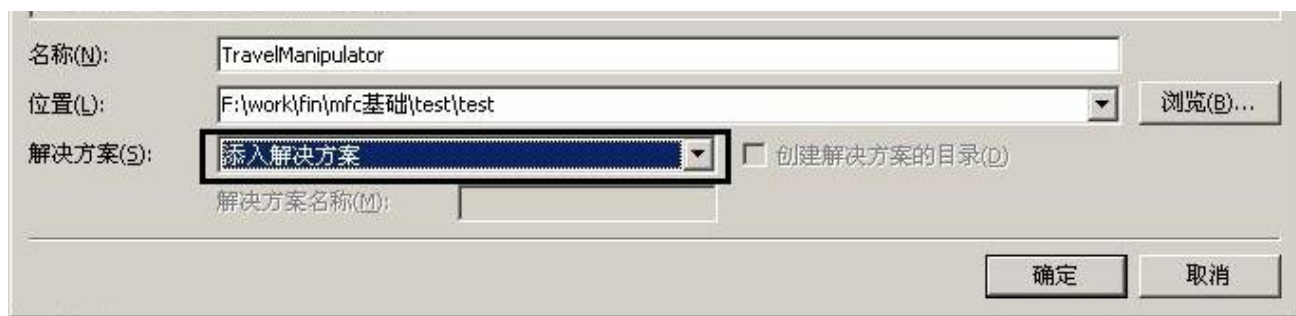


图 3.12

注意：TravelManipulator 是 WIN32 控制台空的 DLL 型项目。

在这里编译运行需要注意，使用 CTRL+F5，默认运行的是主项目，也就是粗体显示的项目，而运行别的

项目则需要使用如图所示的其中之一的操作：



图 3.13

可以在左图中点击重新生成***，也可以在左图中点击批生成，而后在右图中点击生成或重新生成。

注意：可以在包含的头文件中加路径，也可以加../表示返回根目录的上一层，如加入下面代码：`#include "../TravelManipulator/TravelManipulator.h"`相当把 `TravelManipulator.h` 复制到 `test` 中。

通过本章的学习，可以掌握 `DLL` 基础编程与场景漫游控制器的制做。下一章我们将介绍一个十分重要的课题：路径漫游。

第四章 路径漫游

如何按照固定来进行漫游，这一直是个头痛的问题，好在 `OSG` 中为我们提供了强大的形成路径的类，我们才能从早期 `OPENGL` 场景漫游可怕的阴霾中走出来。

在早期按路径漫游主要有几个难点：一、插值，其实 `3D` 的世界就是经过一系列插值形成的世界，在俱备突出特点的特性之间进行平滑的过度一直是 `3D` 主要攻克的难题之一。无论是对图形图像进行插值还是对光照，视口旋转进行插值，都是极有难度的课题。

本章我们将介绍如何使用 `OSG` 本身的功能，构造一个视转平滑的自动漫游程序。

4.1 应用 .PATH 文件

应用 `osgProducer::viewer` 进行操作时, 如果键下 `Z` 键, 以后所做的操作将会被记录在一个根目录的 .PATH 文件中, 再按下 `z` 键 (注意大小写, 如果第一次键下的是大写键, 第二次就键小写, 反之亦然), `viewer` 会把你刚才走过路再原封不动的重复一遍, 而且还会从你最后的地方到起点来个平滑过度。

我们应该可以想到, 如果可以形成一个这样的 `PATH` 文件或是可以控制读取 `PATH` 的类, 使用我们手动做成的 `PATH` 路径, 那么随心所欲的漫游应该不成问题。

4.1.1 PATH 文件结构

在 `.ive` 的文件夹中, 有一个 `.PATH` 文件, 用写字板打开时会如图所示:

```
p 15 -128.5 3 0.707106796640858 0 0 0.707106765732237
0.783823921755419 15 -112 3 0.707106796640858 0 0 0.707106765732237
1.76563529722353 15.3964729309082 -74.5002593994141 3 0.706562352315367 -0.0277428215268407 -0.0277428203141635 0.7065623214305
2.94250358634966 42.4997291564941 -73.0639419555664 3 0.484619668193324 -0.500402683482764 -0.515379514444678 0.499124120489292
3.89776466003361 85.9306793212891 -74.9808349609375 3 0.428319413884764 -0.548840621165792 -0.565918787791613 0.441647345596269
4.34589759312985 103.406562805176 -86.1694030761719 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.51762388795224 109.22233581543 -90.9050827026367 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.59324327533248 112.711799621582 -93.7464904785156 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.63452924882911 113.874954223633 -94.6936264038086 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.67631472715108 115.038108825684 -95.6407623291016 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.71721405932877 116.201263427734 -96.5878982543945 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.75854445187866 118.527572631836 -98.4821701049805 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.79989328252613 119.690727233887 -99.4293060302734 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.84122982110855 120.853881835938 -100.376441955566 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.88257278508861 122.017036437988 -101.323577880859 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.92431579991312 124.34334564209 -103.217849731445 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
4.96521541145592 124.34334564209 -103.217849731445 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
5.00652010241525 126.660654846101 -105.112121582031 3 0.298511595898951 -0.628031364816927 -0.649067611341549 0.308510401489845
```

图 4.1

可以清楚的看到图中共有八列: 究竟这八列分别代表什么意思, 我们来看一下 `osgProducer` 的源文件, 是按下 `Z` 键来记录运行路径, 自然可以想到是由 `Event` 序列响应 `Z` 键, 从而执行了某些功能: 打开 `OSG` 安装目录源文件: `OSG1.2_vc80\src\OSG_OP_OT-1.2\OpenSceneGraph\VisualStudio\OpenSceneGraph.dsw` 中在 `ViewerEventHandler.cpp` 第 1165 行可以看到下面语句: (是响应 `Z` 键下的, 证明是通过 `EventHandler` 来控制录入文件操作):

EventHandler.cpp 第1165行

```
.....
case 'z' :
.....
if (viewer->getAnimationPath())
{
    std::ofstream fout("saved_animation.path");
    viewer->getAnimationPath()->write(fout);
    fout.close();
}
.....
```

可以看到 viewer 调用了 AnimationPath 类中的 write, 把存储在 AnimationPath 中的信息通过输出流导出到"saved_animation.path"中去了, 只要我们设置一个合理的 AnimationPath, 同样可以导出 PATH, 现在看一下 AnimationPath 类中的 write 操作:

在 Core osg 中寻找类: AnimationPath 并定位到 AnimationPath.cpp 第 101 行, 有如下代码:

```
AnimationPath.cpp (第101行)
.....
void AnimationPath::write(std::ostream& fout) const
{
    int prec = fout.precision();
    fout.precision(15);

    const TimeControlPointMap& tcpm = getTimeControlPointMap();
    for(TimeControlPointMap::const_iterator tcpmitr=tcpm.begin();
        tcpmitr!=tcpm.end();
        ++tcpmitr)
    {
        const ControlPoint& cp = tcpmitr->second;
        fout<<tcpmitr->first<<" "<<cp.getPosition()<<"
"<<cp.getRotation()<<std::endl;
    }

    fout.precision(prec);
}
.....
```

其中 **TimeControlPointMap** 是一个关于点与位置的 map, 故写入信息中的第一个变量, 也就是 PATH 文件的第一列: fout<<tcpmitr->first 肯定是路径的时间流逝, 第 2~4 列为: cp.getPosition(), 为视口在世界坐标中的位置, 第 5~8 列为: cp.getRotation() 为旋转矩阵: 当前位置加上什么时间到达这个位置, 再加上旋转视口的朝向, 这就构成了 PATH 文件的整体结构。

因此来讲生成 PATH 文件需要有三个要素: 时间, 控制点, 窗口朝向。有这三个要素就可以形成 AnimationPath 类, 从而按此类进行漫游, 当然也可以导出 PATH 文件。

下面有图示来说明漫游文件的这三个要素:

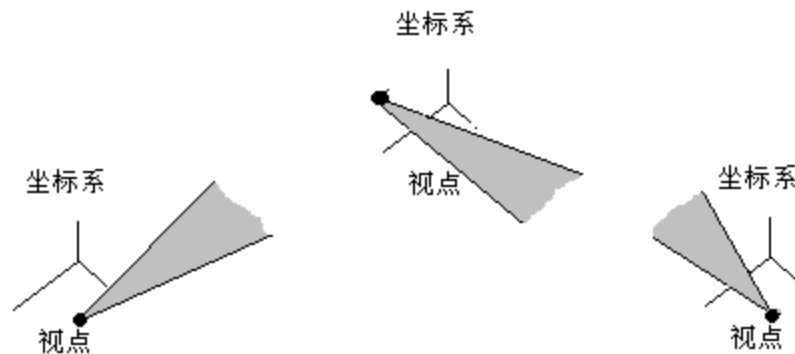


图 4.2

可以看出，在漫游的过程中，改变的只有时间、位置、旋转角度。第一步需要输入这三个量，第二步让 OSG 进行插值形成路径。第三步按路径漫游，这是本章的基本思想。

4.1.2 使用 PATH 文件

现在我们要在 Viewer 中打开 PATH 文件，首先看一下 osgProducer 中是点击 Z 键时是怎么做的：定位到 EventHandler 文件第 1185 行中：如下所示：

```
EventHandler.cpp 第1185行
.....

if (ea.getKey()=='Z')
{
//首先设置一个操作器，是路径操作器，
    osgGA::AnimationPathManipulator* apm = 0;
//操作器ID=0
    unsigned int apmNo = 0;

    .....
//然后分配空间
    apm = new osgGA::AnimationPathManipulator();
//得到ID
    apmNo = viewer->addCameraManipulator(apm);
//录入刚才的AnimationPath进行漫游
    apm->setAnimationPath(viewer->getAnimationPath());
//选择该操作器
    viewer->selectCameraManipulator(apmNo);
}
```

现在我们要打开一个 PATH 文件同样需要这样几步：第一步：申请一个操作器，第二步：申请一个

Animationpath (因为我们在 viewer 没有保存 AnimationPath 我们需要从文件中获得), 第三步: 从文件中读入 AnimationPath, 第四步加入到操作器中, 第五步 viewer 选择该镜头进行操作。

现在我们来新建一个 WIN32 控制台项目试一下该程序:

示例十一: 使用路径文件 PATH

第一步: 菜单文件→新建→项目→WIN32/WIN32 控制台应用程序, 项目名取 testPath, 在应用程序设置中点击空项目后完成。

第二步: 菜单项目→testPath 属性→配置属性→链接器→命令行中添加: OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib osgUtild.lib

第三步: 拷一个 PATH 文件到该项目根目录, 光盘的 IVE 文件中存放有一个 PATH 文件, 也可以自己按 Z 键进行录制。

第三步: 在解决方案视图中, 对源文件文件夹点右键→添加→新建项→代码/CPP 文件, 文件名输入 main, 然后在该文件中添加下面内容:

```
Main.cpp
//By FreeSouth
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/AnimationPath> //必须包含下面两个头文件
#include <osgGA/AnimationPathManipulator>

Int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer> viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    //下面这个文件在光盘IVE文件夹中, 如果没有模型, 将看不到效果, 没有参照物感觉不到移动
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    viewer->setSceneData (node);

    //这个是path的文件名, 可以更改成自己喜欢的
    std::string filename("PATH.path");
    //申请一个输入流, 属于C++的语法知识
    std::ifstream in(filename.c_str());
    //申请一个AnimationPath
    osg::AnimationPath *amp = new osg::AnimationPath ;
    //从文件中读入到AnimaitonPath中
    amp ->read (in) ;
    //关闭流
    in.close () ;

    //PATH操作器
    osgGA::AnimationPathManipulator* apm = new osgGA::AnimationPathManipulator
();
    //操作器ID
```

```

unsigned int apmNo = 0;

//把操作器加入到viewer中
apmNo = viewer->addCameraManipulator(apm);
//设置路径到操作器中
apm->setAnimationPath(amp) ;
//选择该操作器进行漫游（经过上面的操作，该操作器已经有从刚才文件中的PATH了）
viewer->selectCameraManipulator(apmNo);

viewer->realize();
while (!viewer->done())
{
    viewer->sync();
    viewer->update();
    viewer->frame();
}
viewer->sync();
return 0;
} //main.cpp结束

```

按照上面的操作后，程序在运行一开始应该就在楼盘中进行着漫游。可以使用自己的路径进行漫游。到此为止我们讲述了漫游的基本思想：下面来梳理一下：

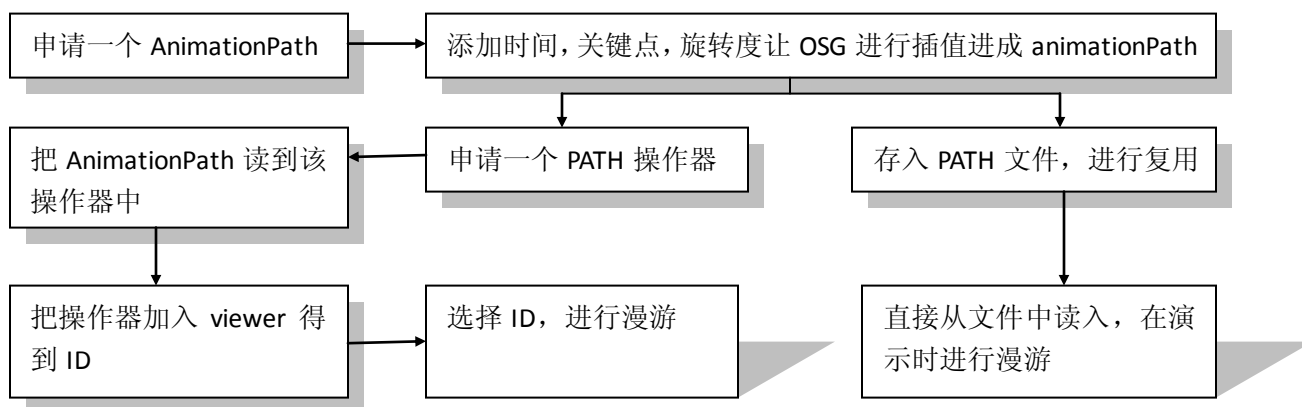


图 4.3

本节我们介绍了如何来使用 PATH 文件。下一节我们来学习一下有关插值的数学原理。

4.2 曲线生成插值算法

在这里我们简单的介绍一下线性插值，就算 osg 为我们提供了该算法，但是如果需要使用曲线预览路径，你该怎么办呢，虽然说不能精确的形成一条完美的曲线，但是通过插值算法近似于所要走的路径是完

全可行的，所以在下一节我们仍会使用到插值算法来对关键点进行曲线拟合，形成一条类似于要走路径的曲线。我们将在下一节使用 Hermite 曲线拟合关键点，生成类似于要走路径的曲线。

现在我们首先来介绍一下插值算法的基础，这是数值分析的内容：

4.2.1 多项式插值

个人感觉代数与几何的完美过渡并不很多，从学习计算数学开始只发生过两处：**插值与矩阵**。下面我们就来介绍一下多项式插值算法，因为一个多项式代表一条曲线，所以对点进行插值形成多项式从而可以形成一条曲线。多项式插值一节本就可以独自成书了，这里只是简单的介绍一下原理，这本是介绍事情是怎么回事的入门书，如要深究下去需要查阅很多资料。

首先来看一下线性插值：

线性插值：

如果说实际曲线为 $f(x)$ ，那么已知上面两点，根据所学的知识，就可以做一条直线 $h(x)$ 来对该 $f(x)$ 进行模拟：

如图所示：

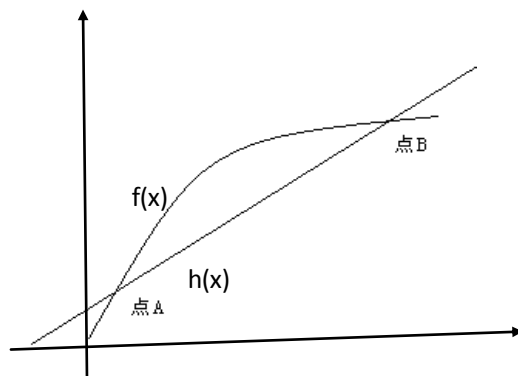


图 4.4

这种插值较容易理解而且比较简单： $h(x) = ax + b$ ，把两点代入此方程就可以解出斜率 a 与截距 b 。

如果推广到一般（拉格朗日插值）可以变换形式如下，设 $A(x_0, y_0)$, $B(x_1, y_1)$:

$$h(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0}$$

从上式中我们提取两个比较有意义的量： $L_0(x) = \frac{x - x_1}{x_0 - x_1}$ 称为 x_0 一次插值基函数，之所以称为一次插值是因为插值结果最高次为一次。

$L_1(x) = \frac{x - x_0}{x_1 - x_0}$ 称为 x_1 的一次插值基函数。

与一次插值相对应的为二次插值，顾名思义插值最高次为二次，这样就需要三次控制点来确定此插值函数。

二次插值

三个控制点来控制一条二次曲线（抛物线） $h(x)$ 来近似的模拟曲线 $f(x)$ 的结果图示如下：





图 4.5

很容易得出如下结论：设置 $h(x)=ax^2+bx+c$ 时方程只有唯一解，因为三个已知数可以确定一个有三个未知数的方程。故如果用抛物线来模拟该曲线，则近似情况如图且只有一种抛物线可以过此三个点，因此也可以推出下面的结论，任何抛物线最多有三个交点。可以使用两个抛物线来解一下，只有唯一解（注意不要把形如双曲的函数称为抛物线，抛物线是一一映射）。

很容易我们把点 ABC 代入 $h(x)$ 中，可以得到准确的结果。

现在来看我们的一般结果（使用基函数来描述）：

首先假设点 A 的坐标 (x_0, y_0) ，点 B 的坐标 (x_1, y_1) ，点 C 的坐标为 (x_2, y_2) ，可以很容易的计算出关于 x_0, x_1, x_2 基函数：

$$L_0(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \quad L_1(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \quad L_2(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_1)(x_2-x_0)}$$

从而得到插值的一般式：

$$h(x) = y_0 L_0 + y_1 L_1 + y_2 L_2$$

从而很容易的解出 a, b, c.

n 次插值

由 $n+1$ 个控制点拟合成的 n 次多项式称为 n 次插值，下面我们由上面关于基函数的结论可以得出 n 次插值的一般式：

$$L_i(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_n)}{(x_i-x_0)(x_i-x_1)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_n)} \quad \text{其中 } i = 0, 1, \dots, n$$

有了这 $n+1$ 个插值基函数，可以轻易求得：

$$h(x) = \sum_{j=0}^n y_j L_j(x)$$

下面我们来介绍一下我们下一节中要使用的插值曲线：Hermite 曲线以及其变种 Cardinal 曲线

4.2.2 Hermite 和 Cardinal 曲线

在现实生活中，一般很难解出有很多控制点的一条曲，往往需要分段插值。

Hermite 曲线：

Hermite 曲线比别的曲线有过人之处的原因是：只需要两个点与两点处的切线向量来进行控制，下面我们来假定起始点为 P_0 ，以及结束点为 P_1 ，以及起始点的切线 m_0 和结束点的切线 m_1 来定义，可以将三次 Hermite(埃尔米特)插值 $p(t)$ 用下列式表示：

$$P(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (t^3 - t^2)m_1 + (-2t^3 + 3t^2)p_1 \quad t \in [0,1]$$

很明显这是个曲线段，因此 $p(t)$ 也称为三次样条 Hermite 曲线段，由上述条件（起始点为 P_0 ，以及结束点为 P_1 ，以及起始点的切线 m_0 和结束点的切线 m_1 ），所以有如下等式：

$$p(0) = p_0 \quad p(1) = p_1 \quad \frac{\partial p}{\partial t}(0) = m_0 \quad \frac{\partial p}{\partial t}(1) = m_1$$

由上述等式确定： $h(x) = at^3 + bt^2 + ct + d \quad t \in [0,1]$ 可以解出 a, b, c, d 由 t 从 0 增到 1 便可以得出曲线

因为切线的长度是不知道的，所以即使对同两点进行插值，即便此两点处的切线斜率固定，但是根据切线长度也可以插出很多种不同中的曲线：如图：

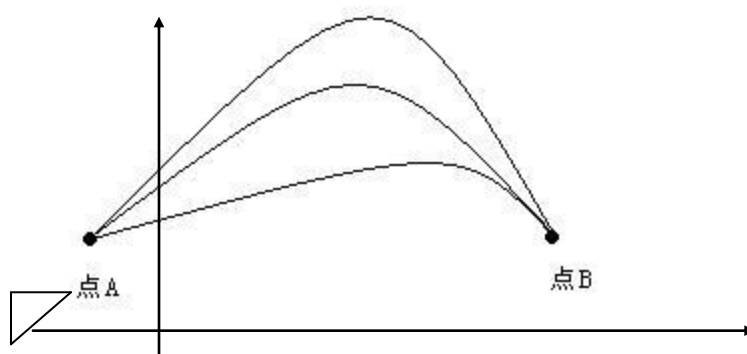
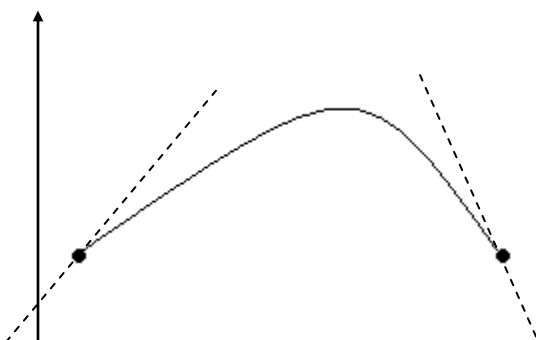


图 4.6

对于点 A 与点 B，虽然有相同的切线斜率，但是确有不同的曲线生成，原因是因为有着不同的切线长度。

Cardinal 曲线：（我们将在下一节中使用的曲线）

那么如何来确定两点的切线斜率才会让曲线看起来不至于十分的离谱呢，Cardinal 在 Hermite 的基础上增加了对斜率的控制，基本思想是使用四个点来确定一条曲线段，前两个点与后两个点控制斜率，从而确定了中间两个点的曲线段：如图所示：



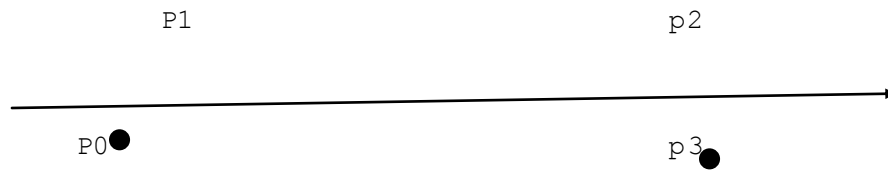


图 4.7

通过持续的压入 n 个关键点，便可以轻易的形成一个过 $n-2$ 个关键点的曲线，在关键头部与尾部进行处理便可以得到过此 n 个关键点的曲线，下面来介绍 Cardinal 的边界条件：

由下列条件确定 Hermite 方程： $h(x) = at^3 + bt^2 + ct + d$ $t \in [0,1]$

$$h(0) = p_1$$

$$h(1) = p_2$$

$$h(0)' = (1-t) * (p_2 - p_0)$$

$$h(1)' = (1-t) * (p_3 - p_1)$$

从而解出 a, b, c, d 然后把 t 从 0 变化到 1 解出整条曲线。

在下一节中我们将使用这条曲线近似拟合 OSG 将要走的路径，当然要对头与尾进行一下处理，可以把头压进去两次，或者向上移动一个单位再压进去，都是可以的，尾部也一样。

下面来看一下 Cardinal 曲线在 OSG 中的运行效果：如图 4.8：

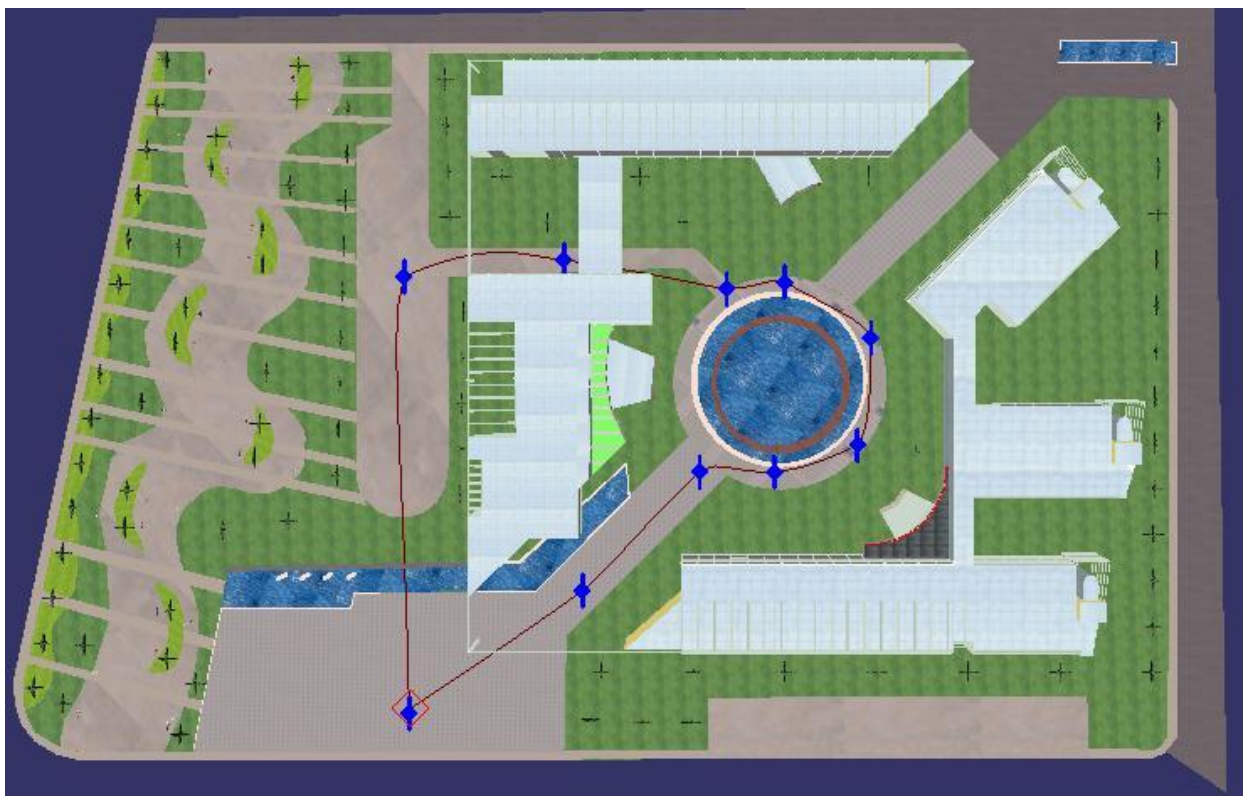


图 4.8

其中棒形标记代表关键点，曲线为拟合形成的曲线，左下角的四边形为起点。

4.2.3 Bzier 曲线

贝塞尔曲线大名鼎鼎，他解决了线性插值过于刚性的问题，对方案平滑度的模拟做了很好的解答。

通过对两点反复进行线性插值便可以得到一个贝塞尔曲线：

线性插值公式为： $p(t) = p_0 + t(p_1 - p_0)$ 变形后为： $p(t) = (1-t)p_0 + tp_1$;

再对其进行线性插值可得到下面关系：

$$\begin{aligned} p(t) &= (1-t)p_0 + tp_1 \\ &= (1-t)[(1-t)a + tb] + t[(1-t)b + tc] \\ &= (1-t)^2 a + 2(1-t)tb + t^2 c \end{aligned}$$

其中最大自由度为 2，也就是二次插值。很明显，贝塞尔曲线不会过关键点，在对数个点进行模拟时它只过起点与终点，朝向是切线。

现在考虑到一般化，对 $n+1$ 个点求解贝塞尔曲线：使用 p_i 代表所有的关键点，当应用 k 次线性插值之后，就可以得到中间控制点 p_i^k

具有 $n+1$ 个控制点的贝塞尔曲线可用如下的递归公式：其中 p_i^0 表示初始控制点 p_i

$$p_i^t(t) = (1-t)p_i^{k-1} + tp_{i+1}^{k-1}(t)$$

下面以图示来看一下贝塞尔曲线的构造过程，比如有四个控制点 $t=0.5$ ：
如图：

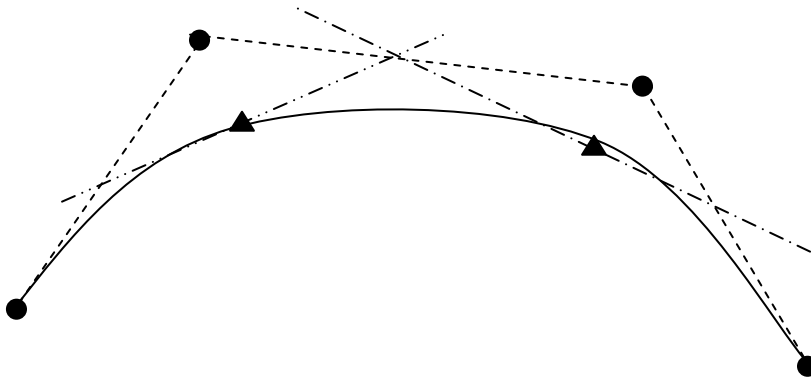


图 4.9

其中虚线相交的地方都是中点 ($t=0.5$)，这样进行一个递归的过程，就形成了如图所示的贝塞尔曲线，图中的三角是第一次二次插值得到的点，以其做为关键点再进行插值便可以轻松得到贝塞尔曲线。

4.3 路径漫游

在实际应用中，路径漫游存在有以下几个步骤：第一步打开路径菜单说明已经开始编辑路径了，第二步从点击菜单开始便可以在模型中取关键点，第三步打开路径菜单说明已经完成编辑，这时形成预览的 Cardinal 曲线，第四步开始漫游。这是最简单的一次性路径漫游，有了这个基础，读者想要做出一个类似于 VEGA 可以对关键点进行修改，删除，定义高度，时间的算法就不那么遥不可及了。

4.3.1 路径漫游流程

下面首先来从理论上来解决上述四个步骤：

第一步：打开路径控制菜单，说明已经开始编辑路径了，这可以通过 MFC 中的菜单加以控制，设置一个 `BOOL m_bPick` 型的变量，来说明已经开始取点。

第二步：模型中取点，可以在 `TravelManipulator.dll` 中做相应的修改，在接到 `m_bPick` 的变量为真后，便启用一个向量，在此向量中加入以后所点的每一个点，并在点下该点时加入一个多边形，以示标识在某个地方已经点过了。

第三步：打开路径菜单说明已经完成编辑，这时其实进行了以下的操作把所有的点存到 `OSGFrame` 中，在 `OSGFrame` 中对这些点应用 `Cardinal` 插值算法进行拟合，形成一条曲线，显示在屏幕上，把所有的关键点加上时间以及关键点切向的角度压入 `AnimationPath` 中形成路径。

第四步：打开路径菜单说明已经开始漫游，这也是通过菜单控制的，在开始漫游时可以隐藏曲线（下面的例子中选择不隐藏），`OSG` 中提供有隐藏结点的操作，隐藏某个结点的操作是 `Node ->SetNodeMask(0)`，显示的操作是 `Node->SetNodeMask(1)`，然后把 `AnimationPath` 压入到 `AnimationPathManipulator` 中，在 `viewer` 中压入 `AnimationPathManipulator` 并选择操作 `ID` 进行漫游。

这就是整个流程，下面我们来首先做一下准备工作来介绍一下重要的类：

首先要介绍的是：`osg::AnimationPath` 类：

类内容如下所示：

类 `osg::AnimationPath`

//这个 map 中 first 是时间，second 中是控制点，等会介绍 `ControlPoint` 类

`typedef std::map< double, ControlPoint > TimeControlPointMap`

//得到运行到某时间的矩阵

`bool getMatrix (double time, Matrixf &matrix) const`

`bool getMatrix (double time, Matrixd &matrix) const`

//得到运行到某时间的逆矩阵

`bool getInverse (double time, Matrixf &matrix) const`

`bool getInverse (double time, Matrixd &matrix) const`

//得到运行到某时间的控制点

`virtual bool getInterpolatedControlPoint (double time, ControlPoint &controlPoint) const`

//插入控制点与时间，这个函数很重要

`void insert (double time, const ControlPoint &controlPoint)`

//得到时间序列的第一值，如果时间序列无值则返回零

`double getFirstTime () const`

//得到时间序列的最后一值，如果无值则返回零

`double getLastTime () const`

//得到时间序列的最后一值与最初值的差

`double getPeriod () const`

//设置循环模式

`void setLoopMode (LoopMode lm)`

```

//得到循环模式
LoopMode getLoopMode () const
//设置时间控制点序列
void setTimeControlPointMap (TimeControlPointMap &tcpm)
//得到时间控制点序列
TimeControlPointMap & getTimeControlPointMap ()
//得到时间控制点序列
const TimeControlPointMap & getTimeControlPointMap () const
//是否为空
bool empty () const
//从 PATH 文件中读入路径
void read (std::istream &in)
//写路径到 PATH 文件中
void write (std::ostream &out) const

```

再来看一下类 ControlPoint

类 ControlPoint

```

//设置控制点位置
void setPosition (const osg::Vec3d &position)
//得到控制点位置
const osg::Vec3d & getPosition () const
//旋转
void setRotation (const osg::Quat &rotation)
const osg::Quat & getRotation () const
//缩放
void setScale (const osg::Vec3d &scale)
const osg::Vec3d & getScale () const
//在第一点与最后一点按比率插入关键点
void interpolate (float ratio, const ControlPoint &first, const ControlPoint &second)
void interpolate (double ratio, const ControlPoint &first, const ControlPoint &second)
//得到矩阵
void getMatrix (Matrixf &matrix) const
void getMatrix (Matrixd &matrix) const
//得到逆矩阵
void getInverse (Matrixf &matrix) const
void getInverse (Matrixd &matrix) const

```

首先我们来看一下生成 AnimationPath 的具体操作:

这里的时间是指 position 的位置, 当运行多少时间时到达这个位置, 在之前已经说过漫游只有三个要素: 时间, 位置, 旋转度, 在这里对应有时间, position, rotation, 所以通过连续的插入关键点、时间、旋转度就可以形成成一个 AnimationPath

```

animationPath->insert (time, osg::AnimationPath::ControlPoint (position, rota
tion));

```

时间的计算是: 首先根据压入的所有关键点, 用两点间距离公式得到所有的关键点的直线距离总长度:

`float OSGFrame::GetAllDistant(void)` ;从而设置在单位长度时需要的时间乘以总长度就可以得到总的运行时间，可以得到两点间的距离与单位时间的乘积便是这两点间的实际运行时间，因此是可以设置漫游不匀速的。

最后压入关键点与所有的时间后形成 `AnimationPath` ,再把此路径压入到 `AnimationPath` 操作器中，然后把操作器压入到 `viewer` 中，再从 `viewer` 中选择 `ID`。这个操作已经进行过了，在这里再梳理一下：

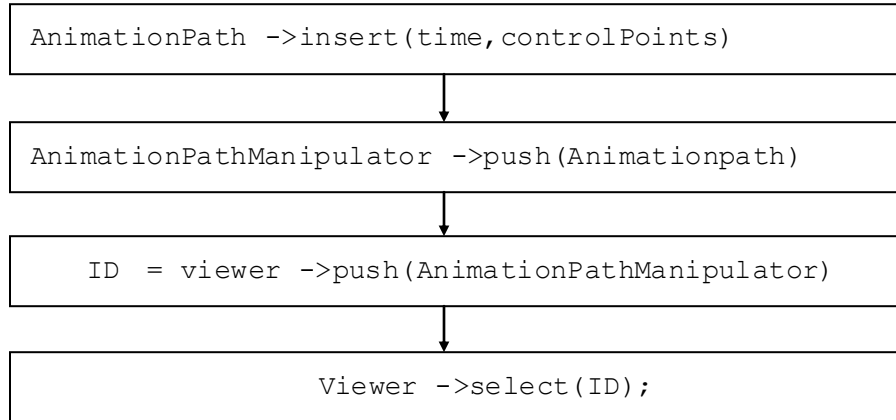


图 4.10

下面我们来进行具体的步骤：

4.3.2 增强 `TravelManipulator.dll`

在这里我们需要 `TravelManipulator.dll` 具有如下功能：

- 第一：可以检测到开始/停止收集点。
- 第二：可以把收集到的点传出去。
- 第三：可以调整场景适合收集点。
- 第四：可以在场景中做标记以标识所收集的点。

现在在 `TravelManipulator.dll` 的头文件中做如下修改：

```
TravelManipulator.h
//OSGFreeS Code By YangShiXing
#ifdef TRAVEL_DLL

#else

#define TRAVEL_DLL __declspec(dllexport)

#endif
#include <osgProducer/Viewer>

#include <osgGA/MatrixManipulator>
#include <osgUtil/IntersectVisitor>
```

```

#include <osg/LineSegment>
#include <osg/Point>
#include <osg/Geometry>

#include <vector>
class TRAVEL_DLL TravelManipulator :public osgGA::MatrixManipulator

{
.....

    //设置位置
    void SetPosition(osg::Vec3 &position) ;
    //得到位置
    osg::Vec3 GetPosition() ;
    // 移动到该点
    void MoveTo(osg::Vec3 dest);
    //控制点序列
    osg::ref_ptr <osg::Vec3Array> point ;
    //开始收集控制点
    bool m_bCollectPoint;
public:
    // 得到控制点
    void SetControlPoint(osg::ref_ptr <osg::Vec3Array> point);
    // pick
    virtual void pick(osgProducer::Viewer* viewer, const osgGA::GUIEventAdapter&
ea);

    // pick是否开启
    bool m_bPick;
    //设置是否开启PICK
    void SetPick(bool m_b);
    //把控制点清空
    void ClearPoint(void);
    //调节视窗以适应收集点，成平面状态
    void SceneAdjustToCollectCtrlPoint(bool);
    //对控制点上画一个图标
    osg::Node * DrawCtrlPoints(osg::Vec3 position);

    //传出控制点
    osg::ref_ptr <osg::Vec3Array > GetControlPoint();
    //传出控制点
    void GetControlPoint(osg::ref_ptr <osg::Vec3Array >);
    // 创建路径
    osg::AnimationPath* CreateAnimationPath(void);
    //第一个点用一个四边形画出来，看出是点下的起点，这里放的是它的指针

```

```

osg::Node *m_pNodeosgFirstNodePointDraw;

//得到第一个点标记，为一个四边形
osg::Node * GetFirstDrawNode(void);
//是否存入的是第一个点，如果是第一个点就画一个与别的点不同的标记
bool m_bFirstPush;
// 画第一个点
osg::Node * DrawFirstNode(osg::Vec3 position);
//是否清楚路径
bool m_bPathClean;
//设置，第一次压入点
void SetFirstPush(bool m_bPush);
} ;
//TravelManipulator.h结束

```

下面我们来看一下这个函数画的图形：以便于读者理解代码：

DrawFirstNode(osg::Vec3 position) 以 position 为中心画四边形

osg::Node * DrawCtrlPoints(osg::Vec3 position);画的是一个棒形

在下图中左边画的是 **DrawFirstNode**，右边画的是 **DrawCtrlPoints**

如图所示：

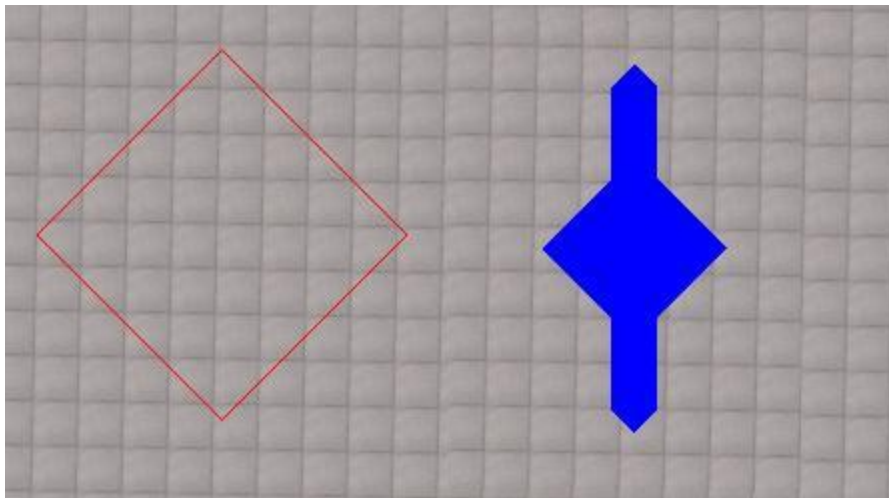


图 4.11

下面是关于 TravelManipulator.cpp 的实现代码：

```

TravelManipulator.cpp
//by @FreeSouth for OSGFreeS
#define TRAVEL_DLL _declspec(dllexport)

#include "..\TravelManipulator.h"
TravelManipulator::TravelManipulator(): m_fMoveSpeed(1.5f)

```

```

, m_bLeftButtonDown(FALSE)
, m_fpushX(0)
, M_fAngle(2.5)
, m_bPeng(true)
, m_fpushY(0)
, m_bCollectPoint(false)
, m_bPick(false)
, m_pNodeosgFirstNodePointDraw(NULL)
, m_bFirstPush(false)
, m_bPathClean(true)
{
    m_vPosition = osg::Vec3(15.0f, -130.0f, 3.0f);
    m_vRotation = osg::Vec3(osg::PI_2, 0.0f, 0.0f);
    point = new osg::Vec3Array;
}
TravelManipulator::~TravelManipulator()
{
}
// 把漫游加入到场景之中,静态成员
TravelManipulator * TravelManipulator::TravelScene(osg::ref_ptr
<osgProducer::Viewer> viewer)
{
    //可以获得并使用系统提供的浏览方式
    TravelManipulator* camera = new TravelManipulator;

    //选择一种浏览方式
    camera->m_nID = viewer->addCameraManipulator(camera);
    //TEMP
    camera->m_pHostViewer = viewer;
    return camera;
}
.....
// 主要事件控制器
bool TravelManipulator::handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& us)
{
    float mouseX = ea.getX();
    float mouseY = ea.getY();
    switch(ea.getEventType())
    {
    case(osgGA::GUIEventAdapter::KEYDOWN):
        {
            .....

            //如果键下x键则调整场景以适合取点

```

```

        if ( ea.getKey () == 0x58 || ea.getKey () == 0x78) //X
        {
            SceneAdjustToCollectCtrlPoint(m_bCollectPoint) ;
            return true ;
        }
        return false;

    }

case (osgGA::GUIEventAdapter ::PUSH ):

    if ( ea.getButton () == 1)

    {
        m_fpushX = mouseX ;
        m_fpushY = mouseY ;

        m_bLeftButtonDown = true ;
    }
    if ( m_bPick )
    {//如果pick被打开，则执行pick
        osgProducer::Viewer* viewer =
dynamic_cast<osgProducer::Viewer*>(&us);
        pick(viewer , ea) ;
    }
    return false ;
    .....
}
}
.....

osg::Vec3 TravelManipulator::GetPosition ()
{
    return m_vPosition ;
}
// 移动到该点
void TravelManipulator::MoveTo(osg::Vec3 dest)
{
    ChangePosition (osg::Vec3 (dest - m_vPosition)) ;
}
// 得到控制点
void TravelManipulator::SetControlPoint(osg::ref_ptr <osg::Vec3Array> cpoint)
{
    point = cpoint ;
}
// pick的应用

```

```

void TravelManipulator::pick(osgProducer::Viewer* viewer, const
osgGA::GUIEventAdapter& ea)
{
    //用于访问检测到碰撞的具体位置
    osgUtil::IntersectVisitor::HitList hlist;
    //设置碰撞模型为整个场景中的模型
    osg::Group* root = dynamic_cast<osg::Group*>(viewer->getSceneData());
    if (!root) return;
    if (m_pHostViewer->computeIntersections(ea.getX(),ea.getY(),hlist))
    {
        //计算碰撞，结果存放在hlist中
        osgUtil::IntersectVisitor::HitList::iterator hitr=hlist.begin();
        //如果说以前没有开始取点，则清空所有点
        if (!m_bFirstPush)
        {
            point ->clear ();
            //得到取到的点，从屏上发出射线与模型交的坐标
            point ->push_back (osg::Vec3 (hitr->getLocalIntersectPoint().x
(),hitr->getLocalIntersectPoint().y(), 3.0 )) ;
            //把第一个点画成四边形
            osgFirstNodePointDraw =
DrawFirstNode(hitr->getLocalIntersectPoint()) ;
            //把该四边形加入到场景中去
            root ->addChild( m_pNodeosgFirstNodePointDraw );
            m_bFirstPush = true ;
        }
        else
        {
            //不是刚开始取点，则一直压入
            point ->push_back (osg::Vec3 (hitr->getLocalIntersectPoint().x
(),hitr->getLocalIntersectPoint().y(), 3.0 )) ;
            //画出棒形在地图所点处
            root ->addChild( DrawCtrlPoints(hitr->getLocalIntersectPoint()));
        }
    }
}

//设置pick状态
void TravelManipulator::SetPick(bool m_b)
{
    m_bPick = m_b ;
}

//清空所有点
void TravelManipulator::ClearPoint(void)
{
    if (!(point->empty ()))
        point.get () ->clear();
}

```



```

}
//调整视口，看到全貌，适应取点
void TravelManipulator::SceneAdjustToCollectCtrlPoint (bool m_bShow)
{
    m_vPosition = osg::Vec3 (15.0f, -130.0f, 3.0f) ;
    m_vRotation = osg::Vec3 (osg::PI_2, 0.0f, 0.0f) ;
    if (!m_bShow)
    {
        m_vRotation._v[0] += osg::DegreesToRadians (-90.0f) ;
        ChangePosition (osg::Vec3 (0, 150, 800)) ;
        m_bCollectPoint =! m_bShow ;
        //在远处靠近时需要把速度设置大一点
        m_fMoveSpeed = 5.0f ;
        return ;
    }
    if (m_bShow)
    {
        m_fMoveSpeed = 1.5f ;
        m_bCollectPoint =! m_bShow ;
        return ;
    } ;
}

//画出控制点
osg::Node * TravelManipulator::DrawCtrlPoints (osg::Vec3 position)
{
    //下面的操作应该非常熟，是第二章的内容
    osg::Group * root = new osg::Group () ;
    //设置点的大小
    osg::ref_ptr <osg::Point > pointsize = new osg::Point ;
    pointsize ->setSize (5.0) ;
    root->getOrCreateStateSet ()->setAttributeAndModes (pointsize.get
(), osg::StateAttribute::ON) ;
    //设置几何结点
    osg::Geode* pyramidGeode = new osg::Geode() ;
    osg::Geometry* pyramidGeometry = new osg::Geometry() ;
    //可画几何结点加入
    pyramidGeode->addDrawable (pyramidGeometry) ;
    root->addChild (pyramidGeode) ;
    osg::ref_ptr <osg::Vec3Array> trangle = new osg::Vec3Array ;
    trangle ->push_back (osg::Vec3 (position.x () -2, position.y () , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x () -0.5, position.y () -1.5 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x () -0.5, position.y () -3.5 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x () , position.y () -4 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x () +0.5, position.y () -3.5 , 5)) ;

```

```

    trangle ->push_back (osg::Vec3 (position.x ()+0.5, position.y ()-1.5 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x ()+2, position.y () , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x ()+0.5, position.y ()+1.5 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x ()+0.5, position.y () +3.5, 5)) ;
    trangle ->push_back (osg::Vec3 (position.x () , position.y ()+4 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x ()-0.5, position.y ()+3.5 , 5)) ;
    trangle ->push_back (osg::Vec3 (position.x ()-0.5, position.y ()+1.5 , 5)) ;
    //把点充入画区
    pyramidGeometry->setVertexArray( trangle.get () );
    //设置所画格式
    osg::DrawElementsUInt * pyramidBase = new
osg::DrawElementsUInt(osg::PrimitiveSet::POLYGON , 0);
    pyramidBase->push_back(11);
    pyramidBase->push_back(10);
    pyramidBase->push_back(9);
    pyramidBase->push_back(8);
    pyramidBase->push_back(7);
    pyramidBase->push_back(6);
    pyramidBase->push_back(5);
    pyramidBase->push_back(4);
    pyramidBase->push_back(3);
    pyramidBase->push_back(2);
    pyramidBase->push_back(1);
    pyramidBase->push_back(0);
    pyramidGeometry->addPrimitiveSet(pyramidBase);
    //颜色设置
    osg::Vec4Array* colors = new osg::Vec4Array;
    colors->push_back(osg::Vec4(0.0f, 0.0f, 1.0f, 1.0f) ); //index 0 red
    //设置颜色
    pyramidGeometry->setColorArray(colors);
    pyramidGeometry->setColorBinding(osg::Geometry::BIND_OVERALL );
    return (osg::Node *) root ;
}
osg::ref_ptr <osg::Vec3Array > TravelManipulator::GetControlPoint()
{
    return point ;
}
// 创建路径, 在这里先看一下是如何创建的路径, 这个函数不是很准确, 在后面我们会在OSGFrame中重
//写一个函数
osg::AnimationPath* TravelManipulator::CreateAnimationPath(void)
{
    //设置路径
    osg::AnimationPath* animationPath = new osg::AnimationPath;
    animationPath->setLoopMode(osg::AnimationPath::LOOP);

```

```

//设置共有多少点，分为多少次插入
int numSamples = point ->size ();
float yaw = 0.0f;
float yaw_delta = 20.0;
//初始角度为零
float roll = osg::inDegrees(00.0f);
//初始时间为零
double time=0.0f;
double time_delta = 0.5;
int i = 0 ;
std::vector <osg::Vec3 > ::iterator iter ;
for(;i<numSamples && iter != point ->end ();++i, iter ++)
{
    osg::Vec3 position(*iter);
    iter ++ ;
//设置旋转，注意，这里的旋转是不准确的，在OSGFrame中我们将会重建此函数
    osg::Quat
rotation(osg::Quat(roll,osg::Vec3(0.0,1.0,0.0))*osg::Quat(-(yaw+osg::inDegrees
s(90.0f)),osg::Vec3(0.0,0.0,1.0)));
animationPath->insert(time,osg::AnimationPath::ControlPoint(position,rotation
));

    yaw += 1.5;
    time += time_delta;

}
return animationPath;
}
//得到控制点
void TravelManipulator::GetControlPoint(osg::ref_ptr <osg::Vec3Array > des)
{
    if (!(des ->empty ()))
        des ->clear () ;
    std::vector <osg::Vec3 > ::iterator iter = point ->begin () ;
    for ( ; iter != point ->end () ; iter ++)
    {
        des ->push_back (osg::Vec3 ((*iter).x(), (*iter).y(), 3.0)) ;
    };
}
//得到第一个画的点，可用于隐藏结点用
osg::Node * TravelManipulator::GetFirstDrawNode(void)
{
    return m_pNodeosgFirstNodePointDraw ;
}

```

```

// 划第一个点
osg::Node * TravelManipulator::DrawFirstNode(osg::Vec3 position)
{
    //这里是第二章的内容，很简单的！
    osg::Group * root = new osg::Group () ;
    //设置几何结点
    osg::Geode* pyramidGeode = new osg::Geode();
    osg::Geometry* pyramidGeometry = new osg::Geometry();
    //可画几何结点加入
    pyramidGeode->addDrawable(pyramidGeometry);
    root->addChild(pyramidGeode);
    osg::ref_ptr<osg::Vec3Array> trangle = new osg::Vec3Array ;
    trangle ->push_back (osg::Vec3 (position.x()-4, position.y(), 5.0)) ;
    trangle ->push_back (osg::Vec3 (position.x(), position.y()-4, 5.0)) ;
    trangle ->push_back (osg::Vec3 (position.x()+4, position.y(), 5.0)) ;
    trangle ->push_back (osg::Vec3 (position.x(), position.y()+4, 5.0)) ;
    //把点充入画区
    pyramidGeometry->setVertexArray( trangle.get () );
    //设置所画格式
    osg::DrawElementsUInt * pyramidBase = new
    osg::DrawElementsUInt(osg::PrimitiveSet::LINE_LOOP,0);
    pyramidBase->push_back(3);
    pyramidBase->push_back(2);
    pyramidBase->push_back(1);
    pyramidBase->push_back(0);
    pyramidGeometry->addPrimitiveSet(pyramidBase);
    //颜色设置
    osg::Vec4Array* colors = new osg::Vec4Array;
    colors->push_back(osg::Vec4(1.0f, 0.0f, 0.0f, 1.0f) ); //index 0 red
    //设置颜色
    pyramidGeometry->setColorArray(colors);
    pyramidGeometry->setColorBinding(osg::Geometry::BIND_OVERALL );
    return (osg::Node *) root ;
}
//设置开始取点
void TravelManipulator::SetFirstPush(bool m_bPush)
{
    m_bFirstPush = m_bPush ;
}

```

编译一下，通过后，我们将使用此 DLL 辅助 MFC+OSG 框架进行漫游：

4.3.3 路径漫游功能演示

现在我们来拿一个 MFC+OSG 程序来进行一下测试,开始主要控制阶段,在这一节中我们要完成以下功能:

添加漫游菜单项,响应菜单项,建立关键点曲线,创建路径,漫游。本节演示的程序一次运行只演示一次,目的是展示如何漫游,至于编辑点,通过菜单停止,或是重新收集点都需要读者自己去细做。

首先第一步:

打开 OSG+MFC 的最基本的框架:

示例十二: 路径漫游

第一步: 建立如图所示的菜单项:



图 4.12

第二步: 分别为它们添加对 CosgFreeViewer 类的响应函数如下:

```
public:  afx_msg void OnSelectpathfile();
public:  afx_msg void OnColloctpoint();
public:  afx_msg void OnOvercollect();
public:  afx_msg void OnDisplaypath();
```

下面来看一下这个功能是如何制做完成的, 首先**第一步: 点击开始创建路径**开始对模型点击收集点, 在点击的过程中不断的调用 TravelManipulator 中的功能画点, **第二步: 点击开始创建路径**, 用 Cardinal 曲线拟合所有点, 而后创建 AnimationPath, **第三步: 点击演示路径**开始演示刚才创建的路径。

下面以图示来说明整个过程:

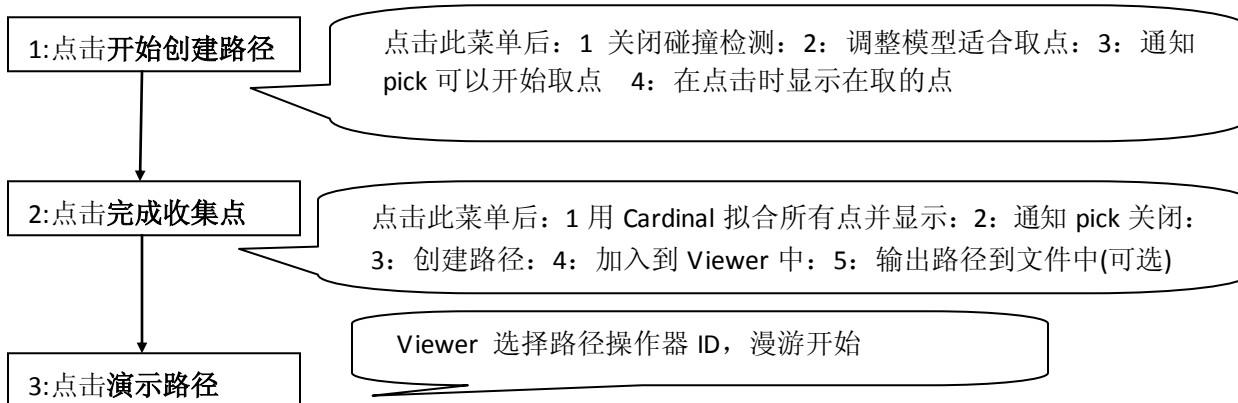


图 4.13

现在我们来看一下将要使用的函数的主要的功能:

```
//控制点文件, 这里是从 TravelManipulator 中传过来的所有的控制点
osg::ref_ptr<osg::Vec3Array>m_refCtrlPoints;
//压入曲线, 把 Cardinal 曲线压入当前场景显示
void PushCardinal(void);
//创建 Cardinal 曲线
osg::Node * CreateCardinal(void);
//创建 Cardinal 曲线
void Cardinal(osg::ref_ptr<osg::Vec3Array> temp);
//得到曲线的四个参数, 在前面有介绍过此曲线的原理, 忘了可以看一下
float GetCoefficient(float p0, float p1, float p2, float p3, float t) ;
//创建路径
osg::AnimationPath* CreatePath(void) ;
//演示路径
void playPath(osg::AnimationPath* path);
//得到从 res 到 des 的演示时间, 好控制 CtrlPoints
float GetRunTime(osg::Vec3 res, osg::Vec3 des);
//得到路径的总折线长度
float GetAllDistant(void);
```

下面来看一下主要的代码改变:

```
osgFreeViewer.h
//OSG FreeS Source Code by FreeSouth. At zzu. vr. lab.
#pragma once
#include "OSGFrame.h"
#include <CString>
#include <string>
//从窗口类继承
class CosgFreeViewer :
    public CWnd
{
public:
    CosgFreeViewer(void);
    //加入OSG场景管理变量
    OSGFrame *m_osgFrame ;
public:
    virtual ~CosgFreeViewer(void);
public :
    //继承虚准备创建函数
    virtual BOOL PreCreateWindow(CREATESTRUCT & cs) ;
public :
    DECLARE_MESSAGE_MAP()
    //所有消息映射
public :
```

```

afx_msg void OnPaint() ;
afx_msg void OnSize(UINT nType, int cx, int cy) ;
afx_msg void OnFileOpen() ;
public://下面是四个菜单映射文件
    afx_msg void OnSelectpathfile();
public:
    afx_msg void OnColloctpoint();
public:
    afx_msg void OnOvercollect();
public:
    afx_msg void OnDisplaypath();
};                                     //osgFreeViewer.h结束

```

CosgFreeViewer.cpp 的改动如下:

```

CosgFreeViewer.cpp
//OSG FreeS Source Code by FreeSouth. At zzu. vr. lab.
#include "StdAfx.h"
//添加需要的头文件
#include "osgFree.h"
#include "osgFreeViewer.h"
//添加消息映射序列

BEGIN_MESSAGE_MAP(CosgFreeViewer, CWnd)
    ON_WM_PAINT()
    ON_WM_SIZE()
    ON_COMMAND(ID_FILE_OPEN, OnFileOpen) //消息映射
    ON_COMMAND(ID_SELECTPATHFILE, &CosgFreeViewer::OnSelectpathfile)
    ON_COMMAND(ID_COLLOCTPOINT, &CosgFreeViewer::OnColloctpoint)
    ON_COMMAND(ID_OVERCOLLECT, &CosgFreeViewer::OnOvercollect)
    ON_COMMAND(ID_DISPLAYPATH, &CosgFreeViewer::OnDisplaypath)
END_MESSAGE_MAP()

//构造
CosgFreeViewer::CosgFreeViewer(void)
{
    //初始化
    m_osgFrame = new OSGFrame ();
}
//析构
CosgFreeViewer::~CosgFreeViewer(void)
{
    //删除
    delete m_osgFrame;
}

```

```

.....
//打开文件
void CosgFreeViewer::OnFileOpen()
{
    //命令处理代码
    CFileDialog fileDlg(TRUE) ;
    if(fileDlg.DoModal () == IDOK)
    {
        //打开处理代码
        CString fileName (fileDlg.GetPathName ()) ;
        m_osgFrame ->setSceneDate (fileName) ;
    }
}
//选择路径文件进行漫游.PATH
void CosgFreeViewer::OnSelectpathfile()
{
    CFileDialog fileDlg(TRUE) ;
    CString fileName ;
    if(fileDlg.DoModal () == IDOK)
    {
        //打开处理代码
        fileName=(fileDlg.GetPathName ()) ;
    }
    USES_CONVERSION;
    //转化为OSG支持的ASCII字符
    std::string str= W2A(fileName) ;
    //演示所示文件
    m_osgFrame ->DisplayPath (str) ;
}
//开始取点，注意点的高度是三米，如果有高三3米的物体，那点将不会显示在物体上，所以最好点在地//
面上
void CosgFreeViewer::OnColloctpoint()
{
    //把碰撞检测关了！
    m_osgFrame ->travels ->m_bPeng = false ;
    //调整视口适合取点！
    m_osgFrame ->travels ->SceneAdjustToCollectCtrlPoint (false) ;
    //开始取点！
    m_osgFrame ->travels ->SetPick (true ) ;
}
//完成取点
void CosgFreeViewer::OnOvercollect()
{
    //得到控制点

```



```

    m_osgFrame ->travels ->GetControlPoint (m_osgFrame ->m_refCtrlPoints.get
    ());
//设置不需要取点
    m_osgFrame ->travels ->SetPick (false) ;
    //显示Cardinal曲线
    m_osgFrame ->PushCardinal () ;
}
//演示路径
void CosgFreeViewer::OnDisplaypath()
{
//得到控制点
    m_osgFrame ->travels ->GetControlPoint (m_osgFrame ->m_refCtrlPoints.get
    ());
//创建并演示路径
    m_osgFrame ->playPath (m_osgFrame ->CreatePath ()) ;
} //CosgFreeViewer.cpp结束

```

来看一下 OSGFrame.h 中做的主要的改动

```

OSGFrame.h
//OSG FreeS Source Code by FreeSouth.At zzu. vr. lab.
#pragma once
#include <osgProducer/Viewer>
#include <osgDB/ReadFile>
#include <osgDB/FileUtils>
#include <CString>
#include <string>
#include <fstream>
#include <osg\LineWidth>
#include <osg\FrontFace>
#include <iostream>
#include <istream>
#include <osgGA\AnimationPathManipulator>
#include "../travel/TravelManipulator.h"
//此函数用于OSG与MFC进行交互
//MFC中不会涉及OSG的变量
class OSGFrame
{
public:
    OSGFrame(void);
public:
    ~OSGFrame(void);
    //操作
public :
    //设置场景数据

```

```

void setSceneDate(CString ) ;
//渲染
bool renderFrame() ;
//对OSG进行初始化
void Initialised(HWND) ;

//是否初始化
bool m_bInitializtion ;
//viewer
osg::ref_ptr<osgProducer::Viewer> m_pViewer ;
TravelManipulator *travels ;

//控制点文件
osg::ref_ptr<osg::Vec3Array>m_refCtrlPoints;
//压入曲线
void PushCardinal(void) ;

//创建曲线
osg::Node * CreateCardinal(void) ;
//创建曲线
void Cardinal(osg::ref_ptr<osg::Vec3Array> temp) ;
//得到Cordinal曲线系数
float GetCoefficient(float p0, float p1, float p2, float p3, float t) ;

//创建路径
osg::AnimationPath* CreatePath(void) ;
//演示路径
void playPath(osg::AnimationPath* path) ;
//得到两点间运行时间
float GetRunTime(osg::Vec3 res, osg::Vec3 des) ;
//得到路径总长度
float GetAllDistant(void) ;
public:

//演示路径，按str所指文件
void DisplayPath(std::string& str) ;
} //OSGFrame.h结束

```

主要是 OSGFrame.cpp 中所做的修正：

下面首先来看一下在创建路径时的角度问题，如果需要自己做角度问题，那会非常的复杂，曾经想要自己完成插值等过程，发现难度很大，不要说弯处的晃动，光是角度的问就有四四一十六种情况，很复杂的，在 OSG 中固定了角度只有以下四种情况：

如图所示：

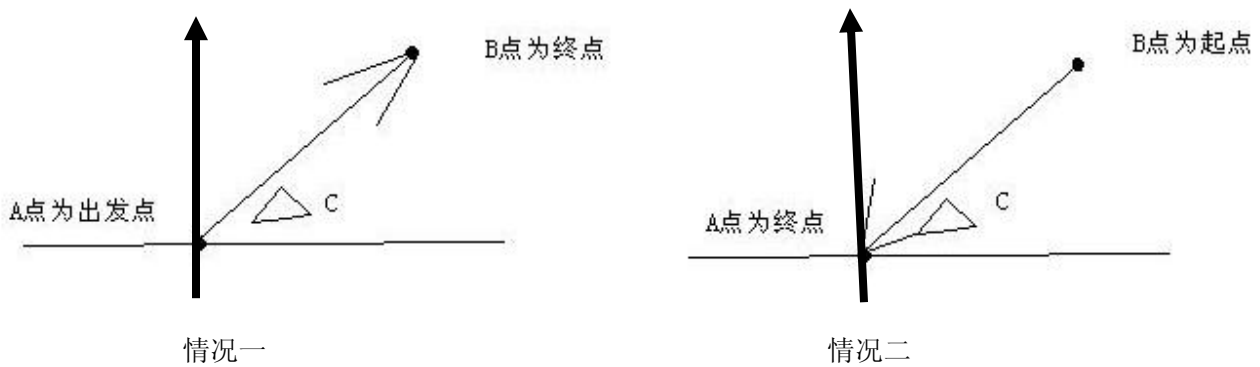


图 4.14

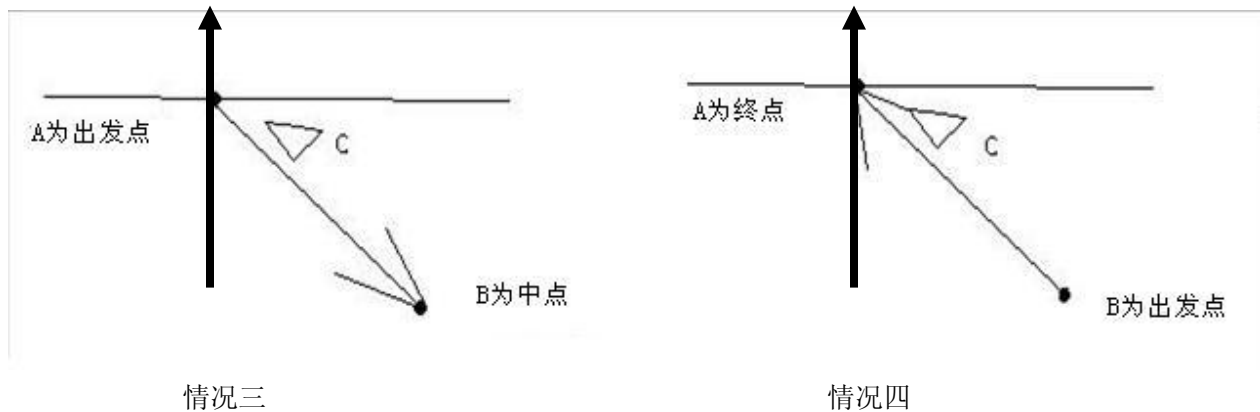


图 4.15

图中黑箭头所指的方向为 OSG 中的绝对零度方向

从上图中可以看出，从 A 到 B 与从 B 到 A 只是朝向上的不同，与 x 轴的夹角同为 C，因此势必对此种情况做出判断，否则旋转时会造成一定的混乱，当然下面的程序对于旋转的处理也不是最好的，可以跟据自己的喜好来重新设计，处理一些拐弯抹角抖动的地方。

情况一：b.x > a.x 向右上 OSG 中最终角度为 $1.57 - C$ ，也就是 90 度减去 C，黑箭头夹角。

情况二：b.x > a.x 向左下 OSG 中最终角度为 $-(1.57 + C)$ ，射线与黑箭头的夹钝负角。

情况三：b.x > a.x 向右下 OSG 中最终角度为 $1.57 + C$ ，也就是 90 度加上 C，与黑箭头的张角。

情况四：b.x > a.x 向左上 OSG 中最终角度为 $-C$ ，如果 b.x = a.x，那么视口不需要动

理清楚之后，来看一下 OSGFree.cpp 的源码：

OSGFrame.cpp

//OSG FreeS Source Code by FreeSouth.At zzu. vr. lab.

```
#include "StdAfx.h"
```

```
#include "OSGFrame.h"
```

```
//构造
```

```
OSGFrame::OSGFrame(void)
```

```
{
```

```
    //初始化
```

```
    m_pViewer = new osgProducer::Viewer ();
```

```
    m_bInitializtion = false ;
```

```

    travels = TravelManipulator::TravelScene (m_pViewer) ;
    travels ->Active () ;
    //为控制点分配空间
    m_refCtrlPoints = new osg::Vec3Array ;
}
.....
//按照路径文件漫游
void OSGFrame::DisplayPath(std::string& str)
{
    std::string filename = str ;
    std::ifstream in(filename.c_str ()) ;
    osg::AnimationPath *amp = new osg::AnimationPath ;
    //从流中读入文件
    amp ->read (in) ;
    //关闭流
    in.close () ;
    //路径操作器
    osgGA::AnimationPathManipulator* apm = new osgGA::AnimationPathManipulator
();
    //得到ID并加入到场景中
    unsigned int apmNo = 0;
    apmNo = m_pViewer->addCameraManipulator(apm);
    apm->setAnimationPath(amp) ;
    //选择ID, 进行漫游
    m_pViewer->selectCameraManipulator(apmNo);
}
//制造并压入Cardinal曲线
void OSGFrame::PushCardinal(void)
{
    //创建曲线
    osg::Node *pNodeCardinal = CreateCardinal() ;
    //如果场景中没有包含此结点, 则压入
    if(!((osg::Group *) (m_pViewer ->getSceneData ()))
->containsNode(pNodeCardinal))
        ((osg::Group*) (m_pViewer ->getSceneData ())) ->addChild(pNodeCardinal) ;
}
//创建曲线结点
osg::Node * OSGFrame::CreateCardinal(void)
{
    //所有的曲线结点, 用户直线连接模拟
    osg::ref_ptr <osg::Vec3Array > allPoints = new osg::Vec3Array ;
    //对所有点进行插值
    Cardinal (allPoints .get ()) ;
    //申请一组
    osg::Group * root = new osg::Group () ;
    //设置线宽

```

```

    osg::ref_ptr<osg::LineWidth> LineSize = new osg::LineWidth;
    LineSize ->setWidth (1.0) ;
    //打开线宽
    root->getOrCreateStateSet()->setAttributeAndModes(LineSize.get
    ( ),osg::StateAttribute::ON);
    //设置几何结点
    osg::Geode* lineGeode = new osg::Geode();
    osg::Geometry* lineGeometry = new osg::Geometry();
    //可画几何结点加入
    lineGeode->addDrawable(lineGeometry);
    root->addChild(lineGeode);
    //把点充入画区
    lineGeometry->setVertexArray( allPoints.get () );
    //设置所画格式
    osg::DrawElementsUInt * lineBase = new
    osg::DrawElementsUInt(osg::PrimitiveSet::LINE_LOOP ,0);
    for (int i =0 ; i<(int)(allPoints ->size ()) ; i++)
    {
        lineBase->push_back(i);
    } ;
    lineGeometry->addPrimitiveSet(lineBase);
    //颜色设置
    osg::Vec4Array* colors = new osg::Vec4Array;
    colors->push_back(osg::Vec4(0.4f, 0.0f, 0.0f, 0.5f) );
    //设置颜色
    lineGeometry->setColorArray(colors);
    lineGeometry->setColorBinding(osg::Geometry::BIND_OVERALL );
    return (osg::Node *) root ;
}
//得到曲线
void OSGFrame::Cardinal(osg::ref_ptr<osg::Vec3Array> temp)
{
    //处理一头一尾两个控制点，各加入一个点，至于为什么，上一节的插值原理中有述
    std::vector<osg::Vec3> ::iterator iter = m_refCtrlPoints ->begin () ;
    osg::ref_ptr<osg::Vec3Array> CtrlTwo = new osg::Vec3Array ;
    CtrlTwo->push_back (osg::Vec3 ((*iter).x()-1, (*iter).y()-1, (*iter).z())) ;
    CtrlTwo ->insert (++(CtrlTwo ->begin ()), m_refCtrlPoints ->begin (),
    m_refCtrlPoints ->end ()) ;
    iter = m_refCtrlPoints ->end () ;
    iter -- ;
    CtrlTwo->push_back (osg::Vec3 ((*iter).x()-1, (*iter).y()-1, (*iter).z())) ;
    iter = CtrlTwo ->begin () ;

    //要得到系数加入所有点

```

```

for ( ; iter != CtrlTwo->end () ; *iter ++ )
{ //关于得到系数的数学原理在上一节中有述
    osg::Vec3 p0 = *iter++ ;
    osg::Vec3 p1 = *iter++ ;
    osg::Vec3 p2 = *iter++ ;
    if (iter == CtrlTwo ->end() )
        return ;
    osg::Vec3 p3 = *iter ;
    iter -- ;
    iter -- ;
    iter -- ;
    float t = 0 ;
    for ( ; t <=1 ; t=t+0.1)
    { //得到系数，并得到最后x,y且加入
        temp ->push_back ( osg::Vec3 (GetCoefficient(p0.x() , p1.x() , p2.x() ,
p3.x() , t) , GetCoefficient(p0.y() , p1.y() , p2.y() , p3.y() , t) , 5.0)) ;
    }
}
}
// 得到系数
float OSGFrame::GetCoefficient(float p0, float p1, float p2, float p3, float t)
{
    float d = p1 ;
    float c = 0.5 * ( 1-t) * (p2 - p0) ;
    float a = 0.5 * ( t-1) * p0 + p1 * ( 1.5 + 0.5 * t ) - ( 1.5 + 0.5 * t) * p2
+ 0.5 * ( 1-t) * p3 ;
    float b = p2 - a - d - c ;
    return ( a * t * t * t + b * t * t + c * t + d ) ;
}
//创建路径
osg::AnimationPath* OSGFrame::CreatePath(void)
{
    //得到总的路径时间
    double looptime = 0.3 * GetAllDistant() ;
    //对路径一头一尾压入两个点，否则点击二个点将不会移动
    std::vector <osg::Vec3 > ::iterator iter =m_refCtrlPoints->end () ;
    std::vector <osg::Vec3 > ::iterator iter2 ;
    iter--;
    iter2 = --iter ;
    iter ++ ;
    m_refCtrlPoints ->push_back (osg::Vec3 ((*iter).x()-(*iter2)).x () ,
(*iter).y()-(*iter2).y() , 3.0)) ;
    //申请一个AnimationPath
    osg::AnimationPath* animationPath = new osg::AnimationPath;

```

```

animationPath->setLoopMode(osg::AnimationPath::LOOP);
//需要多少次插值
int numSamples = m_refCtrlPoints ->size ();
float yaw = 0.0f;
float yaw_delta = 0.5;
float roll = osg::inDegrees(90.0f);

//得到每两点的平均运行时间，这是一种模式，不一定使用
double time=0.0f;
double time_delta = looptime/(double)numSamples;

float angle = 0.0 ;
iter = m_refCtrlPoints ->begin () ;
for(int i=1;i<numSamples;++i, iter ++){
    osg::Vec3 position(*iter);
    iter ++ ;
    if (iter != m_refCtrlPoints ->end ())
    {
        //这只是一种情况，计算夹角
        if ((*iter).x() > position.x())
        {
            angle = 1.57-atan( ((*iter).y () - position.y()) / ((*iter).x() -
position.x())) ;
            if (angle < 0)//极罕见的情况，一般不会发生，
                angle = 1.57 + angle ;
        } ;
        if ((*iter).x() < position.x())
        {
            angle = -(1.57+atan( ((*iter).y () - position.y()) / ((*iter).x()
- position.x())))) ;
            if (angle > 0)//极罕见的情况，一般不会发生
                angle = -(1.57 - angle) ;
        }
    } ;//调整视口，面朝正前方
    osg::Quat rotation( osg::Quat(roll,osg::Vec3(1.0,0.0,0.0)) *
osg::Quat(-angle,osg::Vec3(0.0,0.0,1.0)));
    osg::Quat rotationY(osg::Quat ( -(3.1415926/6.0), osg::Vec3 (1.0, 0.0,
0.0))) ;

    //插入时间与控制点，第一次时时间为零
    animationPath->insert(time,osg::AnimationPath::ControlPoint(position,rotat
ion));
    //增加时间

```

```

        time += GetRunTime (position, *iter) ;
        iter -- ;
    }
    //导出路径文件, 可选
    std::ofstream fout("FreePath.path");
    animationPath->write(fout);
    fout.close();
    return animationPath;
}

//运行路径
void OSGFrame::playPath(osg::AnimationPath* path)
{
    //按AnimationPath漫游, 下面的操作应该很熟悉
    osgGA::AnimationPathManipulator* apm = new osgGA::AnimationPathManipulator
    ();
    unsigned int apmNo = 0;
    apmNo = m_pViewer->addCameraManipulator(apm);
    apm->setAnimationPath(path) ;
    m_pViewer->selectCameraManipulator(apmNo);
}

//得到两点间运行总时间
float OSGFrame::GetRunTime(osg::Vec3 res, osg::Vec3 des)
{
    float distant = sqrt ( (des.x () - res.x())*(des.x () - res.x())  + (des.y
    ())-res.y())*(des.y ()-res.y()) ) ;
    double init = 0.3 ;
    return (init * distant);
}

//得到路径总长度
float OSGFrame::GetAllDistant(void)
{
    //两点间的距离段相加
    float distant = 0.0 ;
    float p = 2.0 ;
    std::vector <osg::Vec3 > ::iterator iter = m_refCtrlPoints ->begin ();
    int i = 1 ;
    const int size = m_refCtrlPoints ->size () ;
    if ((m_refCtrlPoints ->size ()) <= 1)
        return 0;
    else
    {
        for (;i < size ; i++,iter ++)
            //两点间的距离公式

```



```

        osg::Vec3 temp = *iter ;
        iter ++ ;
        distant += sqrt ( (temp.x () - (*iter).x())*(temp.x () - (*iter).x())
+ (temp.y ()-(*iter).y ()) * (temp.y ()-(*iter).y())) ;
        iter -- ;
    } ;
}
return distant ;
}

```

编译运行。

到此为止这一章已经结束，通过这一章的学习，应该可以熟练的掌握路径漫游的技巧，上面的程序只是实现路径漫游的功能，让读者知道是怎么回事，生产中使用的漫游程序非常复杂，控制十分严密，需要复杂的结构，但是原理上与上述是一样的，所以无 BUG 的漫游程序需要读者自己开发出来。

关于插值曲线的原理，由于本书不是一本数学书，只是介绍常用的曲线的形成过程，关于更多的计算方法请参阅图形学与数值分析方面的书籍。

下一章我们将学习粒子系统。

第五章 粒子系统

本章我们将来学习一下 OSG 的粒子系统与粒子系统的一般原理与实现过程。

本章包括以下内容：OSG 中的粒子系统要素，粒子系统的原理，使用 OSG 中的粒子系统模拟爆炸，雪，喷泉等效果。附加的，将介绍雾效的制做，雾效使用的不是粒子系统。

通过本章的学习，可以使用 OSG 进行简单的场景模拟，天气模拟等，但是 OSG 中的粒子系统虽然使用方便，但是仿真度不高，一般人们都会在建模上多下功夫，不愿意把功夫花在 OSG 的粒子系统中，比如用流动纹理代替喷泉，当然高仿真度的粒子模拟必须得依赖高精度的算法实现，主要还是靠编程实现，但是就目前而言建模仍然是模拟的核心部分。

本章将编写一个关于爆炸的示例程序，一个雪(雨)的动态链接库，一个喷泉的动态链接库控件。

5.1 OSG 中粒子系统要素

Osg 中关于粒子系统的操作都在 `osgParticle` 名字空间中，大多数的操作简单易用，下面来介绍一下 `osgParticle` 中都有哪些基本的操作与 `osg` 中粒子系统的加入过程。

5.1.1 粒子系统的模拟过程

在 OSG 中提供有专门的粒子系统工具，名字空间为 `osgParticle`，OSG 对经常使用的粒子模拟都做了专门的类，如：`ExplosionEffect` 用于爆炸的模拟，`FireEffect` 用于火的模拟，`ExplosionDebrisEffect` 用于爆炸后四散的颗粒模拟，等等。这些类基本上使用起来极其方便，稍后我们来介绍这几个类。

在 OSG 中有下面这几个类组成 OSG 粒子系统的中坚部分：

```
osgParticle::Particle : 粒子模版，决定粒子的大小，颜色，生命周期，等等。
osgParticle::ParticleSystem: 粒子系统的总体属性，粒子总数，纹理等等
osgParticle::Counter : 粒子产生的数目范围
osgParticle::Placer : 粒子出生点的形状，如环形，圆形，点形
osgParticle::Shooter : 粒子发射器，决定粒子的初速
osgParticle::Emitter : 上述模版以及操作都为这个类服务，类名：发射器
osgParticle::Program : 可接受对粒子的操作，如轨迹的定义，矩阵的变换等等
osgParticle::Operator : 粒子操作或用户自定义粒子操作
osgParticle::ParticleEffect : 可单独使用的粒子渲染，模仿烟雾，爆炸等等。
osgParticle::osgPrecipitationEffect : 这可能是个新类，可以使用雾效（其实不属于粒子系统），雪效等
```

上面的基类中下面各有很多的派生类，这些派生类大多数的使用方法大同小异，比 `BoxPlacer`（出生点为盒状态）与 `SectorPlacer`（出生点可定义为圆形，环形），同为 `Placer`，使用方法基本雷同。

派生类中的函数意义大都一目了然，故只要理清楚粒子系统的创建步骤，就可以很轻易的创建一个符合自己意图的粒子系统了。

在 OSG 中使用粒子系统一般要经历以下几个步骤：

第一步：确定意图（包括粒子的运动方式等等诸多方面）。**第二步：建立粒子模版**，按所需要的类型确定粒子的角度（该角度一经确定，由于粒子默认使用有 Billboard 所以站在任何角度看都是一样的），形状（圆形，多边形等等），生命周期等。**第三步：建立粒子系统**，设置总的属性，**第四步：设置发射器**（发射器形状，发射粒子的数目变化等），**第五步：设置操作**（旋转度，风力等等因素）。**第六步：加入结点，更新。**

下面我们来以图示来描述一下各个部分是怎么协调工作的：在下一小节我们将剖析这个图：

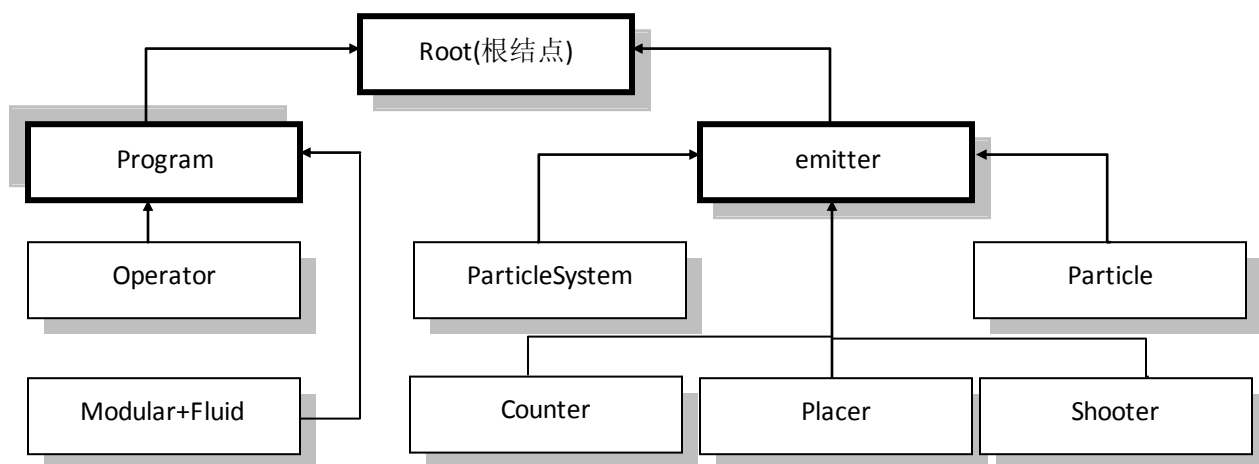


图 5.1

从上图可以看出，粒子系统各类间的协同工作流程。下面以图示来解释一下上图，更为清晰：

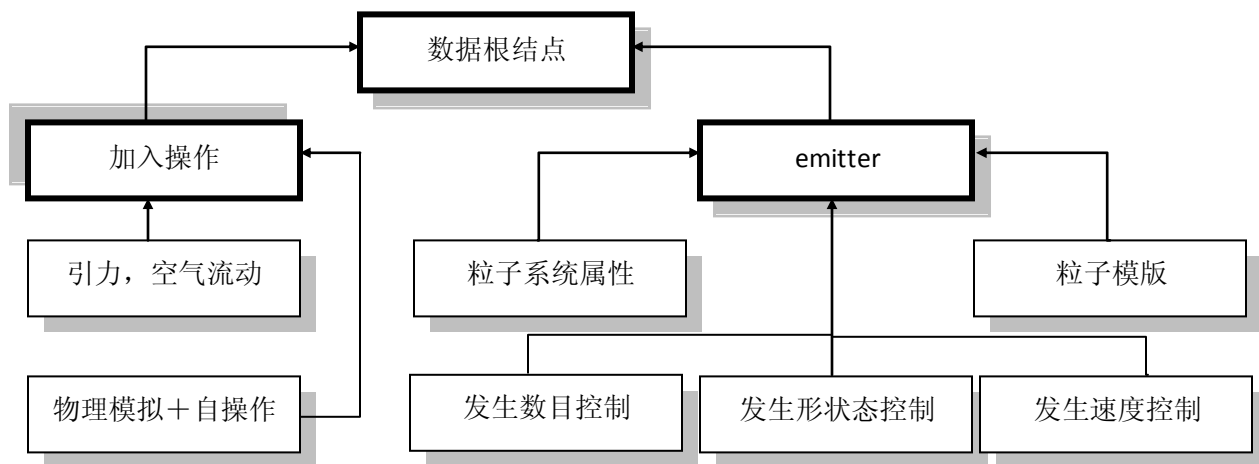


图 5.2

从下图来解释上图，相对应的操作。下面我们再来看一下关于这一类粒子系统的模拟过程：

下图（图 5.3）的这些类，基本上都可以独立的使用，当然也可以设置模版与粒子系属性来满足要求，但是老实讲确实需要一定的创新能力才能做出好的效果来，有些时候往往想的效果和出现的效果有很大的差异。

这系类有时候甚至直接申请加入到场景中就可以有明显且不错的效果，这是因为这些类的初始化都做过非常好的处理，打个比方来说：如果人们使用粒子系统都是做下雪的，那么可能粒子系统刚 new 一个，加进去出现的就是雪效。

下面的粒子系统对于模仿灾难场面有明显的效果，第一个 (ExplosionDebrisEffect) 是爆炸物四溅的模拟，可以设置范围等等一系列的参数，第二个 (ExplosionEffect) 是爆炸的模拟，不得不说，做的很优秀。第三个 (FireEffect) 关于火光的模拟，第四个 Smoke(Trail)Effect，这个搞笑了，是关于烟雾的模拟，加上括号可以对轨迹做特殊的处理，是两个不同的类。细心的程序员可以做出有时候非常滑稽的效果。

这些类都是从：osgParticle::ParticleEffect 派生而来的。

具有共同的操作。

下面来具体的看一下这些类的关系图：

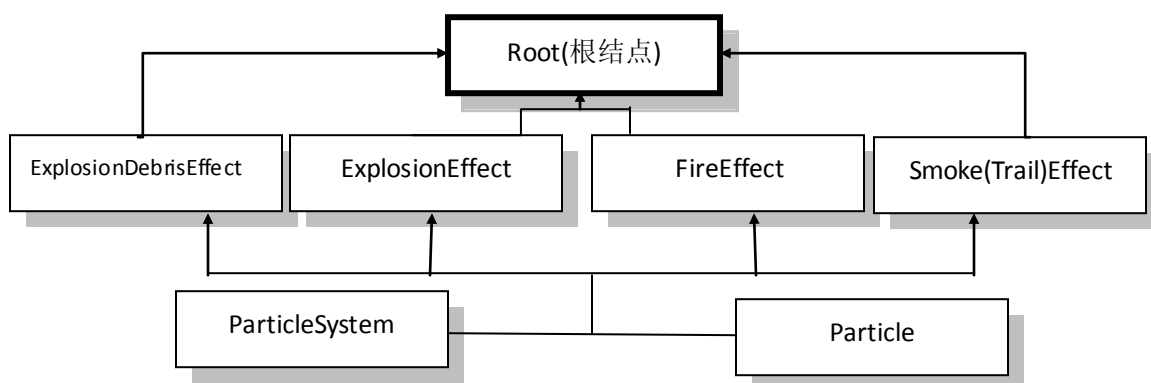


图 5.3

5.1.2 osgParticle 中的类

现在我们来解释一下这些类中都有哪些操作可以在 OSG 界大放光彩，首先我们从最基本的看起：

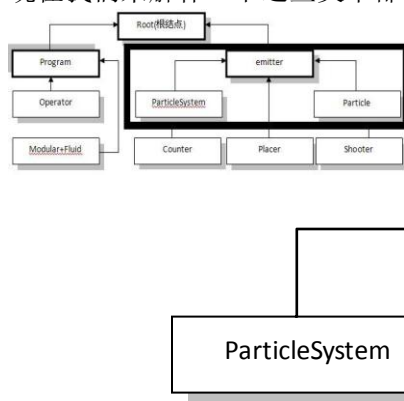


图 5.4

这里进行的操作是建立一个模版，一个系统，加入到发射器中发射：
首先来看一下类 ParticleSystem 与类 Particle，来看一下类常用的操作

类 osgParticle::Particle

```

//得到粒子的形状，Shape 有以下枚举值（这些值大家应该都非常熟悉）：POINT, QUAD,
QUAD_TRIANGLESTRIP, HEXAGON, LINE
Shape getShape () const
//设置粒子的形状，值得注意的是如果定义与点线有关的，最好设置一下点大与线宽，这两个
（point/linewidth）属于 osg::StateAttribute，这就使使用起来非常方便。
void setShape (Shape s)
//查看粒子系统的生命周期是否结束
bool isAlive () const
//得到粒子系统的生存周期，量纲应该是秒
double getLifeTime () const
//得到粒子的年龄，已运行了多长时间，量纲同样是秒
double getAge () const
//得到形状状态的变化范围，rangef 是个包含两个值的小类，因为点与线是通过 osg::StateAttribute 设置的，
所以这个操作以及设置范围的操作只对多边形有效
const rangef & getSizeRange () const
//得到透明度的变化范围
const rangef & getAlphaRange () const
//得到颜色的变化范围
类 osgParticle::Particle
const rangev4 & getColorRange () const
//量纲应该是米
float getRadius () const
//得到粒子重量，量纲应该是 KG
float getMass () const
//得到重量的倒数，这个不知道有什么用
float getMassInv () const
//得到位置向量
const osg::Vec3 & getPosition () const
//得到速度
const osg::Vec3 & getVelocity () const
//得到上帧更新时的位置
const osg::Vec3 & getPreviousPosition () const
//得到角度向量
const osg::Vec3 & getAngle () const
//得到角速度
const osg::Vec3 & getAngularVelocity () const
//得到上一帧渲染时的角速度
const osg::Vec3 & getPreviousAngle () const
//得到当前的颜色值
const osg::Vec4 & getCurrentColor () const
//得到当前的透明度
//摧毁粒子模版
void kill ()
//设置生命周期，量纲应该是秒，如果设置成<=0，则粒子永不消失

```

```

void setLifeTime (double t)
//设置尺寸变化范围，当然还是只对多边形状态是有效的
void setSizeRange (const rangef &r)
//设置在生命周期中透明度的变化范围
void setAlphaRange (const rangef &r)
//设置在生命周期中颜色的变化范围
void setColorRange (const rangev4 &r)
//设置半径
void setRadius (float r)
//重量
void setMass (float m)
//位置
void setPosition (const osg::Vec3 &p)
//速度
void setVelocity (const osg::Vec3 &v)
//加上当前速度
void addVelocity (const osg::Vec3 &dv)
//改变速度向量值
类 osgParticle::Particle
void transformPositionVelocity (const osg::Matrix &xform)
//改变速度向量值
void transformPositionVelocity (const osg::Matrix &xform1, const osg::Matrix &xform2, float r)
//设置角度
void setAngle (const osg::Vec3 &a)
//设置角速度
void setAngularVelocity (const osg::Vec3 &v)
//增加角速度到当前
void addAngularVelocity (const osg::Vec3 &dv)
//改变角速度向量
void transformAngleVelocity (const osg::Matrix &xform)
//更新&开始渲染
bool update (double dt)
void beginRender () const
//渲染，粒子系统会自动调用
void render (const osg::Vec3 &xpos, const osg::Vec3 &px, const osg::Vec3 &py, float scale=1.0f) const
//结束渲染
void endRender () const
//得到当前尺寸
float getCurrentSize () const
//设置当前粒子
void setPreviousParticle (int previous)
//得到上一帧渲染粒子
int getPreviousParticle () const
//设置下一帧渲染粒子

```

```
void setNextParticle (int next)
```

```
//得到下一帧渲染粒子
```

```
int getNextParticle () const
```

类 ParticleSystem

//得到/设置粒子排列方式，是 Billboard 还是 Fixed，一般使用 Billboard，但是 Billboard 在旋转过程中如果粒子叶片过大，可能被某些物体挡住，从而失真

```
Alignment getParticleAlignment () const
```

```
void setParticleAlignment (Alignment a)
```

```
//得到 BoundingBox
```

```
const osg::BoundingBox & getDefaultBoundingBox () const
```

```
//设置 BoundingBox,可能会导致缩放
```

```
void setDefaultBoundingBox (const osg::BoundingBox &bbox)
```

```
//得到是否启用 DoublePass 渲染
```

```
bool getDoublePassRendering () const
```

```
//启用 DoublePass 渲染
```

```
void setDoublePassRendering (bool v)
```

```
//当前是束在冻结中，如果被冻结，将会暂停渲染直至开启 类 ParticleSystem
```

类 ParticleSystem

```
bool isFrozen () const
```

```
//设置是否冻结
```

```
void setFrozen (bool v)
```

```
//得到总粒子数
```

```
int numParticles () const
```

```
//得到死亡粒子数
```

```
int numDeadParticles () const
```

```
//判断是否所有粒子都已死亡
```

```
bool areAllParticlesDead () const
```

```
//得到指向第 i 个粒子的指针
```

```
Particle * getParticle (int i)
```

```
const Particle * getParticle (int i) const
```

```
//从已有模版中创建新模版
```

```
virtual Particle * createParticle (const Particle *ptemplate)
```

```
// 杀死第 i 个粒子
```

```
virtual void destroyParticle (int i)
```

```
//重新激活第 i 个粒子
```

```
virtual void reuseParticle (int i)
```

```
//得到默认粒子模版
```

```
Particle & getDefaultParticleTemplate ()
```

```
const Particle & getDefaultParticleTemplate () const
```

```
//设置默认粒子模版
```

```
void setDefaultParticleTemplate (const Particle &p)
```

```
//得到是否冻结被裁切剔除的粒子（可能不需要显示）
```

```
bool getFreezeOnCull () const
```

```

//设置是否冻结被裁切剔除的粒子（可能不需要显示）
void setFreezeOnCull(bool v)
//设置默认属性，纹理名
void setDefaultAttributes(const std::string &texturefile="", bool emissive_particles=true, bool
lighting=false, int texture_unit=0)
//计算 BoundingBox
virtual osg::BoundingBox computeBound () const

```

上述两个类是决定粒子系统中大的方面，而关于发射器的操作则是由：发射的随机数目范围，发射点的形状与发射时的速度取向三个部分组成，即便不设置这三个部分，发射器也会创建一个默认的含有默认这三个部分的操作的发射器。

这三个类皆派生有适合使用的派生类，派生类中的操作大同小异。

派生类设计的目的是为了使用方便，当然读者也可以自己派生出一个适合的派生类，这就像在第三章派生 `MatrixManipulator` 一样。

下面我们来从成员上来看一下这三个主要的类都有哪些使用起来非常顺手的操作：

如图所示三类中的派生类与发射之间的关系：

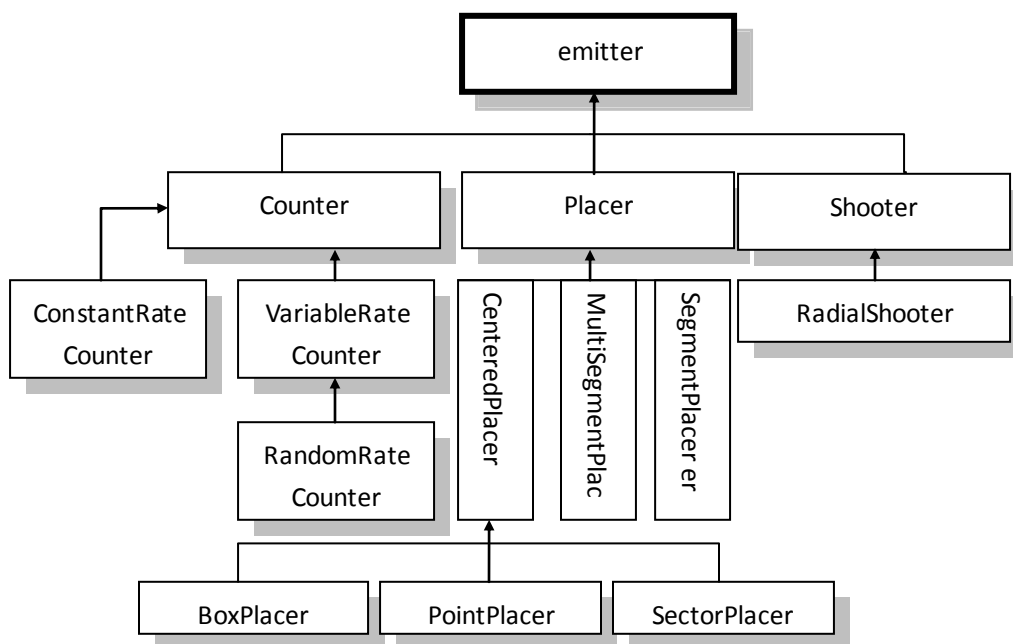


图 5.5

在粒子系统与粒子模版加入到发射器以后，最重要的就是设置数目范围，发射器，发射形状这三个十分重要的类，当然为了使用方便，从这三个基类中派生了很多容易使用的类，我们来分别看一下这三个基类中的操作：

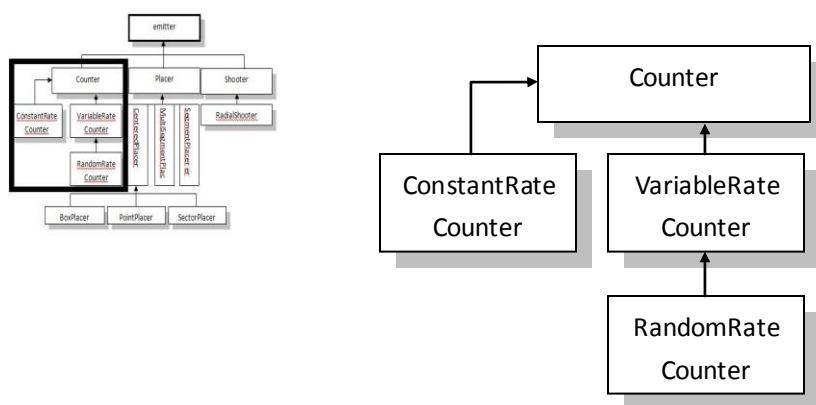


图 5.6

如图 5.6 所示，这个类主要用于创建粒子数目

类 Counter

//创建粒子的数目，纯虚函数，必须在派生类中实现

virtual int numParticlesToCreate (double dt) const =0

//此类用于创建连续发射的粒子，创建在单位时间内发射的粒子

类 ConstantRateCounter

//设置创建的最小粒子数目/每秒（但不是每秒创建的粒子数目）

void setMinimumNumberOfParticlesToCreate (int minNumToCreate)

//得到创建的最小粒子数目/每秒

int getMinimumNumberOfParticlesToCreate () const

//设置每秒创建的粒子数（但并不是场景中现有的粒子数）

void setNumberOfParticlesPerSecondToCreate (double numPerSecond)

//得到每秒钟创建的粒子数

double getNumberOfParticlesPerSecondToCreate () const

//在当前帧中创建的粒子数

virtual int numParticlesToCreate (double dt) const

//此类用于创建当前场景内共有的粒子，创建场景中一共有多少粒子，注意与 ConstantRateCounter 的区别

类 VariableRateCounter

//得到场景中粒子数的最小与最大值

const rangef & getRateRange () const

//设置场景中粒子数的最小与最大值

void setRateRange (const rangef &r)

//设置场景中粒子的最小与最大值

void setRateRange (float minrange, float maxrange)

//此类从 VariableRateCounter 派生而来

类 RandomRateCounter

//当前帧创建的粒子总数

Int numParticlesToCreate (double dt) const

上述三个类都是从 Counter 派生而来，具有它的所有操作，一般用于对粒子数目在场景中给予限制。

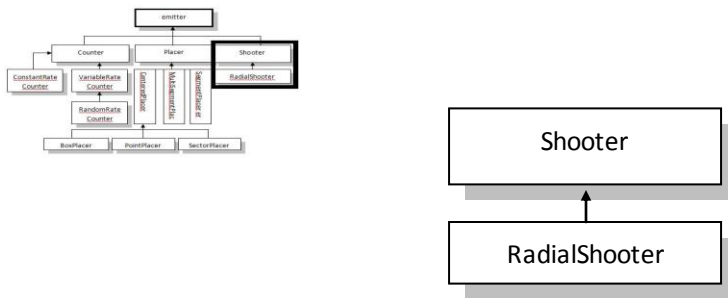


图 5.7

如图 5.7 所示：这两个类用于创建发射器，粒子将从此进行发射，具备限制速度与方向的作用类 **RadialShooter**

//得到 Theta，此角度一般认为是速度方向（量）与 Z 轴的夹角

const rangef & getThetaRange () const

//设置 Theta 范围，此角度一般认为是速度方向（量）与 Z 轴的夹角

void setThetaRange (const rangef &r)

//设置 Theta 范围

void setThetaRange (float r1, float r2)

//得到 Phi 范围，此角度一般认为是速度在 X-Y 平面上的投影与 X 轴的夹角

const rangef & getPhiRange () const

//设置 Phi 范围，此角度一般认为是速度在 X-Y 平面上的投影与 X 轴的夹角

void setPhiRange (const rangef &r)

//设置 Phi 范围，此角度一般认为是速度在 X-Y 平面上的投影与 X 轴的夹角

void setPhiRange (float r1, float r2)

//得到初始速度范围

const rangef & getInitialSpeedRange () const

//设置初始速度范围

void setInitialSpeedRange (const rangef &r)

//设置初始速度范围

void setInitialSpeedRange (float r1, float r2)

//得到/设置最初速度旋转的范围

const rangev3 & getInitialRotationalSpeedRange () const

void setInitialRotationalSpeedRange (const rangev3 &r)

void setInitialRotationalSpeedRange (const osg::Vec3 &r1, const osg::Vec3 &r2)

//设置需要发身的粒子模版，一般需要系统自动调用

void shoot (Particle *P) const

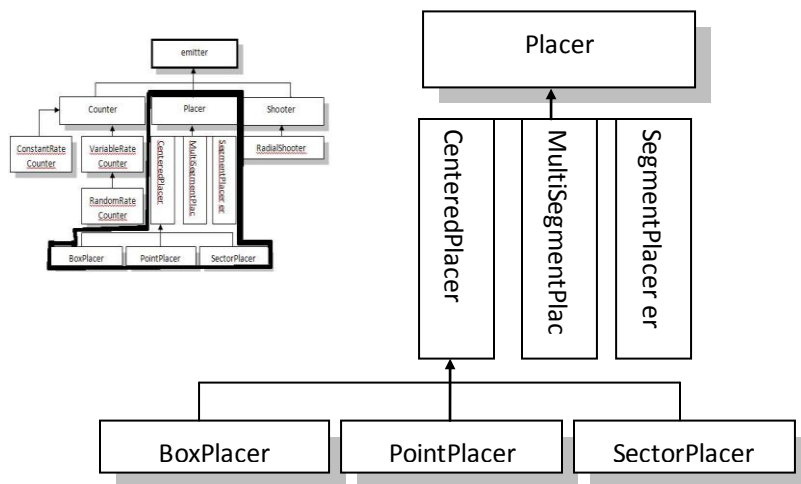


图 5.8

如图 5.8 所示：这个类负责创建发射源的形状，可以是点形，圆形，环形，等：

虚基类 Placer

//取得&放置粒子模版

virtual void place (Particle *P) const=0

//得到粒子源位置

virtual osg::Vec3 getControlPosition () const=0

这个类创建线段型的发射源

类 SegmentPlacer

//得到线段起点，用线段的起点与终点来确定发射源的线段

const osg::Vec3 & getVertexA () const

//设置线段的起点

void setVertexA (const osg::Vec3 &v)

void setVertexA (float x, float y, float z)

//得到线段的终点

const osg::Vec3 & getVertexB () const

//设置线段的终点

void setVertexB (const osg::Vec3 &v)

void setVertexB (float x, float y, float z)

void setSegment (const osg::Vec3 &A, const osg::Vec3 &B)

//设置粒子模版，一般 OSG 调用

void place (Particle *P) const

//得到控制点，为线段的中点

osg::Vec3 getControlPosition () const

设置多线段相边的发射源

类 MultiSegmentPlacer

//设置线段的总顶点数

```

    int numVertices () const
//得到第 i 个顶点
    const osg::Vec3 & getVertex (int i) const
//设置第 i 个顶点
    void setVertex (int i, const osg::Vec3 &v)
    void setVertex (int i, float x, float y, float z)
//增加一个顶点，在末尾
    void addVertex (const osg::Vec3 &v)
    void addVertex (float x, float y, float z)
//删除第 i 个顶点
    void removeVertex (int i)
//设置粒子模版，OSG 自己调用
    void place (Particle *P) const
//得到控制点，是顶点序列的第一个点
    osg::Vec3 getControlPosition () const

```

设置圆环形发射源，为虚基类

```

类 CenterPlacer
//得到圆心
    const osg::Vec3 & getCenter () const
//设置圆心
    void setCenter (const osg::Vec3 &v)
    void setCenter (float x, float y, float z)

```

设置盒型发射源从 CenterPlacer 派生而来

```

类 BoxPlacer
//得到 X 坐标范围
    const rangef & getXRange () const
//设置 X 坐标范围
    void setXRange (const rangef &r)
    void setXRange (float r1, float r2)
//得到 Y 坐标范围
    const rangef & getYRange () const
//设置 Y 坐标范围
    void setYRange (const rangef &r)
    void setYRange (float r1, float r2)
//得到 Z 坐标范围
    const rangef & getZRange () const
//设置 Z 坐标范围
    void setZRange (const rangef &r)
    void setZRange (float r1, float r2)
//设置粒子模版，OSG 自己调用
    void place (Particle *P) const
//得到控制点，认为是中心原点

```

```
osg::Vec3  getControlPosition () const
```

设置点形发射源从 CenterPlacer 派生而来

类 PointPlacer

//设置粒子模版，OSG 自己调用

```
void  place (Particle *P) const
```

//得到控制点，是 Center

```
osg::Vec3  getControlPosition () const
```

设置环形发射源从 CenterPlacer 派生而来

类 SectorPlacer

//得到环形半径的范围

```
const rangef &  getRadiusRange () const
```

//设置环形半径的范围

```
void  setRadiusRange (const rangef &r)
```

```
void  setRadiusRange (float r1, float r2)
```

//得到 phi 范围，在这里代表圆心角

```
const rangef &  getPhiRange () const
```

//设置 phi 范围，在这里代表圆心角

```
void  setPhiRange (const rangef &r)
```

```
void  setPhiRange (float r1, float r2)
```

//设置粒子源，OSG 自己调用

```
void  place (Particle *P) const
```

//得到控制点，圆心

```
osg::Vec3  getControlPosition () const
```

下面来看一下另一个帮派的粒子效果：

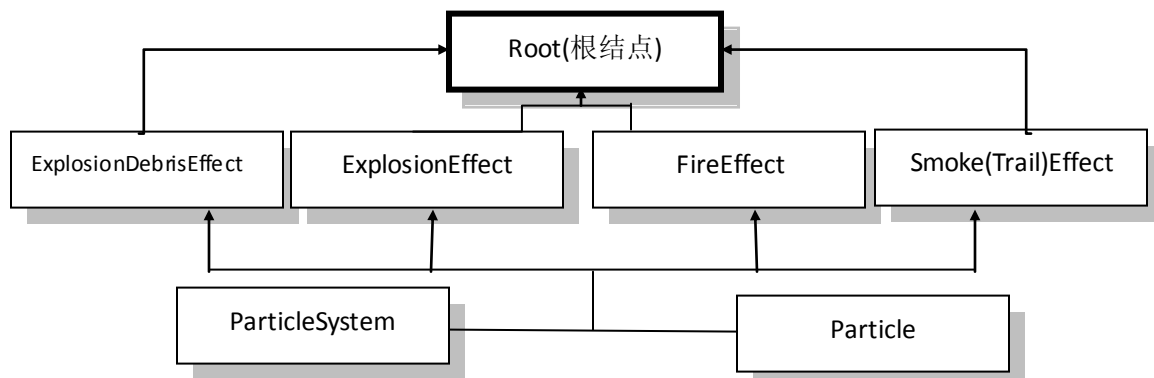


图 5.8

首先来看一下 Effect 系列的基类 `osgParticle::ParticleEffect`：从它派生出的四个类操作基本相同

类 ParticleEffect

//设置/得到纹理图片名

```

void  setTextureFileName (const std::string &filename)
const std::string &  getTextureFileName () const
//设置/得到默认粒子模版
void  setDefaultParticleTemplate (const Particle &p)
const Particle &  getDefaultParticleTemplate () const
//设置/得到具体位置
void  setPosition (const osg::Vec3 &position)
const osg::Vec3 &  getPosition () const
//设置/得到缩放率
void  setScale (float scale)
float  getScale () const
//设置/得到剧烈程度
void  setIntensity (float intensity)
float  getIntensity () const
//设置/得到开始时间
void  setStartTime (double startTime)
double  getStartTime () const
//设置/得到发射器的持续/间隔时间
void  setEmitterDuration (double duration)
double  getEmitterDuration () const
void  setParticleDuration (double duration)
double  getParticleDuration () const
//设置/得到风速影响
void  setWind (const osg::Vec3 &wind)
const osg::Vec3 &  getWind () const
//判断是否所有粒子都已死亡，场景中看不见粒子并不代表已经死亡
bool  areAllParticlesDead () const

```

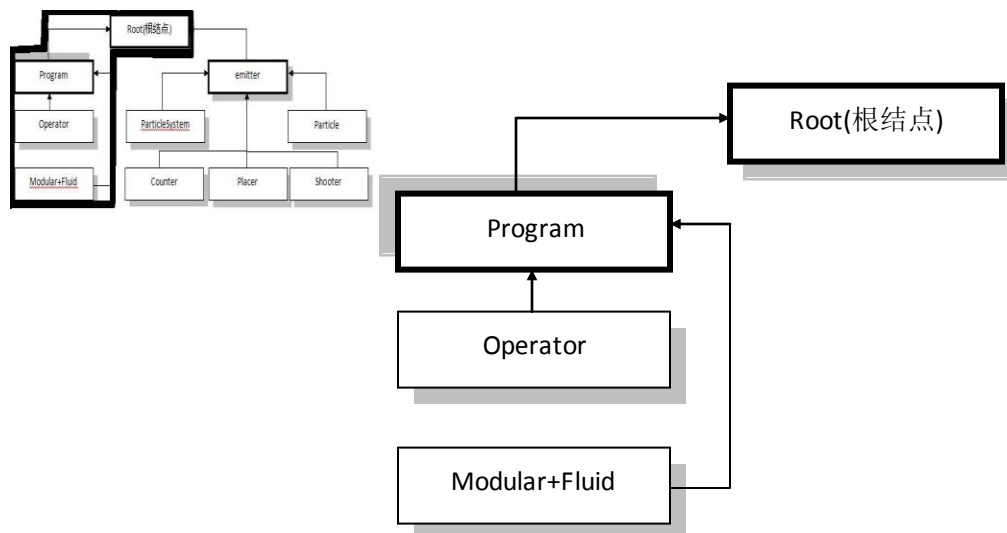


图 5.9

如图 5.9: 这里有对粒子系统的最基本的操作, 包括模拟气流等等, 用户也可以自己定义自己的 Operator:

此类定义空间影响，最常见的是气流：

```
类：FluidProgram
//设置粘性指数
void setFluidViscosity (float v)
//得到粘性指数
float getFluidViscosity () const
//设置密度
void setFluidDensity (float d)
//得到密度
float getFluidDensity () const
//设置风速
void setWind (const osg::Vec3 &wind)
//得到风速
const osg::Vec3 & getWind () const

//设置加速度
void setAcceleration (const osg::Vec3 &v)
//得到加速度
const osg::Vec3 & getAcceleration () const
//设置向下引力
void setToGravity (float scale=1.0f)
//设置模拟空气
void setFluidToAir ()
//设置模拟水下
void setFluidToWater ()
```

这基本上是粒子系统所有的基本的操作。下面我们来具体的实践一下各个类。

5.2 爆炸物模拟

在具备以上基础后，其实粒子系统的编程就变得不是那么复杂了，现在我们来使用 ParticleEffect 看看它的具体用法：

要做一个模拟爆炸后冒烟的场景：

示例十三：爆炸物模拟：

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testFire 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB: OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrainind.lib osgTextd.lib osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

第四步：在其中键如如下代码：

Main.cpp

```
#include <osgProducer/Viewer>
#include <osg/Group>
#include <osgParticle/ExplosionEffect>
#include <osgParticle/ExplosionDebrisEffect>
#include <osgParticle/SmokeEffect>
#include <osgParticle/FireEffect>
void CreateFire(osg::Group *root)
{
    //设置风速
    osg::Vec3 wind(1.0f,0.0f,0.0f);
    //设置位置
    osg::Vec3 position(0, 0, 0);
    //申请爆炸物, 1.0 为缩放比, 当前为不缩放, 还有一个默认的剧烈程度的参数, 默认为 1.0
    osgParticle::ExplosionEffect* explosion = new
osgParticle::ExplosionEffect(position, 1.0f);
    //申请爆炸后的碎片, 1.0 为缩放比, 当前为不缩放, 还有一个默认的剧烈程度的参数, 默认为
1.0
    osgParticle::ExplosionDebrisEffect* explosionDebri = new
osgParticle::ExplosionDebrisEffect(position, 1.0f);
    //建立烟模拟, 1.0 为缩放比, 当前为不缩放, 还有一个默认的剧烈程度的参数, 默认为 1.0
    osgParticle::SmokeEffect* smoke = new osgParticle::SmokeEffect(position,
1.0f);
    //建立火焰模拟, 1.0 为缩放比, 当前为不缩放, 还有一个默认的剧烈程度的参数, 默认为 1.0
    osgParticle::FireEffect* fire = new osgParticle::FireEffect(position,
1.0f,5.0f);
    //设置风力影响
    explosion->setWind(wind);
    explosionDebri->setWind(wind);
    smoke->setWind(wind);
    fire->setWind(wind);
    //加入到根结点
    root->addChild(explosion);
    root->addChild(explosionDebri);
    root->addChild(smoke);
    root->addChild(fire);
}
int main(int argc, char **argv)
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::Group *root = new osg::Group;
    //创建结点
    CreateFire(root);
}
```



```
viewer.setSceneData(root);  
viewer.realize();  
  
while( !viewer.done() )  
{  
    viewer.sync();  
    viewer.update();  
    viewer.frame();  
  
}  
viewer.sync();  
viewer.cleanup_frame();  
viewer.sync();  
return 0;  
} //Main.cpp 结束
```

运行结果:

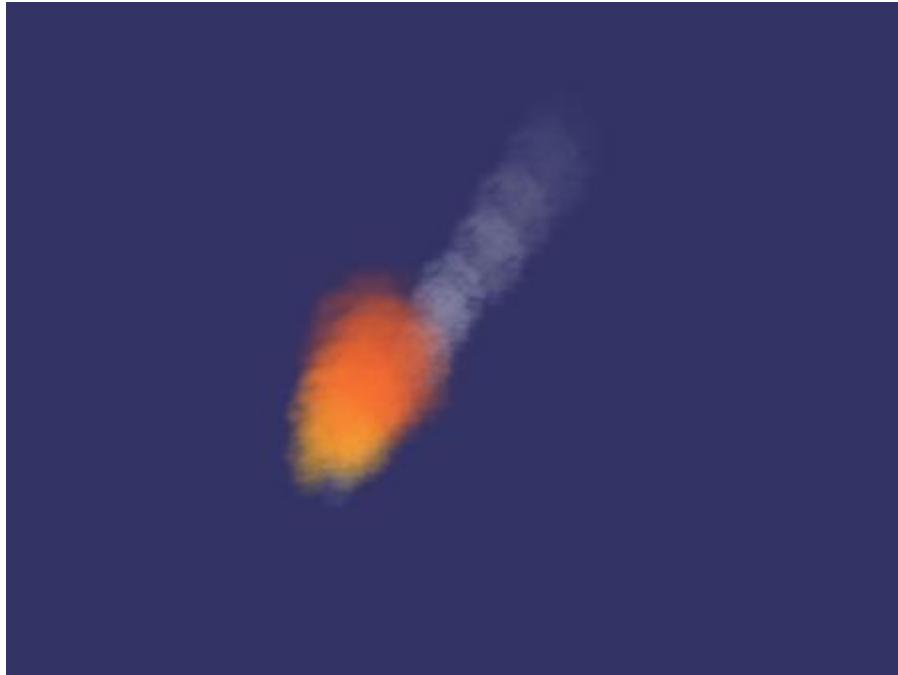


图 5.10

短短的几行代码，产生的效果还算可以，这就是 OSG 的粒子系统使用的方便之处。
由上一节的基础，上面的代码应该很容易理解。

5.3 雨,雪,喷泉的模拟

雨与雪的模拟可以归为一类，只是速度慢一些，纹理必须有变化，喷泉的模拟也不是很难。下面就来做一个雨与雪通用的 DLL，与一个喷泉的 DLL。

5.3.1 雨，雪的模拟

首先来看一下雨与雪的不同，即而决定引出哪些接口，首先雨与雪的纹理不同，所以需要动手设定纹理，故需要接口：`std::string m_strFileName;`

其次需要通用接口：`float m_fRotateDe;`表示雪片的旋转下落程度，一般雨是垂直下落的，只受重力影响

`float m_fTheSize; float m_fTheSize2;`代表雨/雪在下落过程中的大小变化。

`float m_fTheNum;`粒子总数

`float m_fWindX;`受 X 轴的风力影响

`float m_fTheSpeed;`下落的速度

`bool m_bTheRain;`是否下雨

这只是引出接口的一个示例，这个 DLL 的能力还是很弱的，更多读者想要加的接口的添加也是很容易的，这里只是举个事例来说明粒子系统的使用：

下面来建立 `Snow.dll`

示例十四：建立 `Snow.dll`

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：`Snow` 在应用程序设置中选 DLL 与空项目选项。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LB: `OpenThreadsWin32d.lib` `Producerd.lib` `osg.d.lib` `osgDBd.lib` `osgFXd.lib` `osgGAd.lib` `osgParticled.lib` `osgProducerd.lib` `osgSimd.lib` `osgTerrain.d.lib` `osgTextd.lib` `osgUtil.d.lib`

第三步：菜单项目→添加类，类名填入 `Snow`。

而后在 `Snow.h` 中加入以下内容：

`Snow.h`

```
#ifndef SNOW_DLL
#else
#define SNOW_DLL __declspec(dllexport)
#endif
#include <osgParticle/Particle>
#include <osgParticle/ParticleSystem>
#include <osgParticle/ParticleSystemUpdater>
#include <osgParticle/ModularEmitter>
#include <osgParticle/ModularProgram>
#include <osgParticle/RandomRateCounter>
#include <osgParticle/SectorPlacer>
#include <osgParticle/RadialShooter>
#include <osgParticle/AccelOperator>
#include <osgParticle/FluidFrictionOperator>
#include <osg/Node>
#include <osg/Group>
#include <osg/Geode>
#include <string>
//控制建立雪/雨效的类
```

```

class SNOW_DLL Snow
{
public:
    Snow(void);
    ~Snow(void);
    // 返回雪的结点
    osg::Node * CreateSnow(osg::Group* root);
private:
    osgParticle::ParticleSystem * createSnowSystem(osg::Group* );
public:
    // 旋转的程度
    float m_fRotateDe;
    // 雪/雨片大小
    float m_fTheSize;
    // 粒子的数目
    float m_fTheNum;
    // 受 X 轴向风力影响
    float m_fWindX;
    // 雪花落地大小
    float m_fTheSize2;
    // 下落速度
    float m_fTheSpeed;
    // 是否有雨
    bool m_bTheRain;
    //纹理
    std::string m_strFileName ;
};//Snow.h 结束

```

在 Snow.cpp 中加入以下内容:

```

Snow.cpp
//OSGFreeS Source Code By FreeSouth
#define SNOW_DLL_declspec(dllexport)
#include ".\snow.h"
//雪花使用,重载 Operator
class VortexOperator: public osgParticle::Operator {
public:
    VortexOperator()
        : osgParticle::Operator(), center_(0, 0, 0) , intensity_(0.1f) , m_fRotateDe(0.6)
    {
        axis_.set(0, 0, m_fRotateDe) ;
    }

    VortexOperator(const VortexOperator &copy, const osg::CopyOp &copyop) =
    osg::CopyOp::SHALLOW_COPY)

```

```

        :   osgParticle::Operator(copy,   copyop),   center_(copy.center_),   axis_(copy.axis_),
intensity_(copy.intensity_) {}
    META_Object(osgParticle, VortexOperator);
    void setCenter(const osg::Vec3 &c)
    {
        center_ = c;
    }
    void setAxis(const osg::Vec3 &a)
    {
        axis_ = a / a.length();
    }
    void beginOperate(osgParticle::Program *prg)
    {
        if (prg->getReferenceFrame() == osgParticle::Program::RELATIVE_RF) {
            xf_center_ = prg->transformLocalToWorld(center_);
            xf_axis_ = prg->rotateLocalToWorld(axis_);
        } else {
            xf_center_ = center_;
            xf_axis_ = axis_;
        }
    }
    void operate(osgParticle::Particle *P, double dt)
    {
        float l = xf_axis_ * (P->getPosition() - xf_center_);
        osg::Vec3 lc = xf_center_ + xf_axis_ * l;
        osg::Vec3 R = P->getPosition() - lc;
        osg::Vec3 v = (R ^ xf_axis_) * P->getMassInv() * intensity_;
        osg::Vec3 newpos = P->getPosition() + v * dt;
        P->setPosition(newpos);
    }
protected:
    virtual ~VortexOperator() {}
private:
    osg::Vec3 center_;
    osg::Vec3 xf_center_;
    osg::Vec3 axis_;
    osg::Vec3 xf_axis_;
    float intensity_;
public:
    // 旋转的程度
    float m_fRotateDe;
};
//雪花使用构造函数
Snow::Snow(void)

```

```

: m_fRotateDe(0.6)
, m_fTheSize(0.75)
, m_fTheNum(500)
, m_fWindX(1.0)
, m_fTheSize2(1.5)
, m_fTheSpeed(1.0)
, m_bTheRain(false)
{
}
Snow::~~Snow(void)
{
}
// 返回雪的结点
osg::Node * Snow::CreateSnow(osg::Group* root)
{
    //创建粒子系统
    osgParticle::ParticleSystem *ps2 = createSnowSystem(root);
    osgParticle::ParticleSystemUpdater *psu = new osgParticle::ParticleSystemUpdater;
    psu->addParticleSystem(ps2);
    return psu;
}
osgParticle::ParticleSystem * Snow::createSnowSystem(osg::Group* root)
{
    osgParticle::Particle ptemplate;
    //设置生命周期
    ptemplate.setLifeTime(20);
    //设置图片与时间相关的属性
    //尺寸与时间的关系
    ptemplate.setSizeRange(osgParticle::range(m_fTheSize, m_fTheSize2));
    //透明度与时间的关系
    ptemplate.setAlphaRange(osgParticle::range(1.0f, 1.5f));
    //设置颜色与时间的关系
    ptemplate.setColorRange(osgParticle::rangeVec4(
        osg::Vec4(1.0, 1.0f, 1.0f, 1.5f),
        osg::Vec4(1.0, 1.f, 1.0f, 0.0f)));
    // 离子的物理属性
    ptemplate.setRadius(0.05f);    // 5 厘米宽
    ptemplate.setMass(0.5f);    // 50 克重
    //下面我们将要建立离子系统了,我们关闭纹理的绑定,因为它自己可以完成功能
    osgParticle::ParticleSystem *ps = new osgParticle::ParticleSystem;
    ps->setDefaultAttributes(m_strFileName, false, false);
    // 对粒子系统关联离子模版
    ps->setDefaultParticleTemplate(ptemplate);
    //下面我们要创建一个发射器,会是 ModularEmitter,我们将定义数目等详细参数
    osgParticle::ModularEmitter *emitter = new osgParticle::ModularEmitter;

```

```

emitter->setParticleSystem(ps);
// 建立数目
osgParticle::RandomRateCounter *counter = new osgParticle::RandomRateCounter;
counter->setRateRange(m_fTheNum, m_fTheNum);
emitter->setCounter(counter);
//设定冲击面,会是一个半径为的圆,离子将会在这个圆之中
osgParticle::SectorPlacer *placer = new osgParticle::SectorPlacer;
placer->setCenter(0, 0, 150.0);
//半径比较大, 如不合适可更改, 也可引出接口更改
placer->setRadiusRange(0, 600);
placer->setPhiRange(0, 2 * osg::PI);
emitter->setPlacer(placer);
//受地心引力影响,初速度为零
osgParticle::RadialShooter *shooter = new osgParticle::RadialShooter;
shooter->setInitialSpeedRange(0, 0);
emitter->setShooter(shooter);
// 增加结点到发射器
root->addChild(emitter);
//创建一个类来模仿地心引力
osgParticle::ModularProgram *program = new osgParticle::ModularProgram;
program->setParticleSystem(ps);
// 创建一个对象来模仿地心引力
osgParticle::AccelOperator *op1 = new osgParticle::AccelOperator;
op1->setToGravity(m_fTheSpeed);
program->addOperator(op1);
// 不是雨就下雪我们自己定义的操作
if ( !m_bTheRain)
{
VortexOperator *op2 = new VortexOperator;
op2 ->m_fRotateDe = m_fRotateDe ;
op2->setCenter(osg::Vec3(8, 0, 0));
program->addOperator(op2);
};
// 添加空气的流动性
osgParticle::FluidFrictionOperator *op3 = new osgParticle::FluidFrictionOperator;
op3->setFluidToAir();
op3 ->setWind (osg::Vec3 (m_fWindX, 0, 0)) ;
program->addOperator(op3);
root->addChild(program);
osg::Geode *geode = new osg::Geode;
geode->addDrawable(ps);
root->addChild(geode);
return ps;
}

```

编译运行后，现在新建一个控制台项目进行测试该 DLL。

测试步骤：

第一步： 菜单文件→新建→项目→WIN32/WIN32 控制台应用程序，项目名取 testSnow，在应用程序设置中点击空项目后完成。

第二步： 菜单项目→test 属性→配置属性→链接器→命令行中添加：OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib osgUtild.lib **Snow.lib,TravelManipulator.lib** 注意，要添加刚才我们设计的 lib。加漫游。

第三步： 除此外我们要在场景中应用漫游功能，把刚才 Snow 项目以下三个文件拷贝到该项目目录中：TravelManipulator.h,TravelManipulator.lib,TravelManipulator.dll,Snow.h ,Snow.lib ,Snow.dll，（该项目目录是指放有该项目源文件的目录）

第四步： 在解决方案视图中，对源文件文件夹点右键→添加→新建项→代码/CPP 文件，文件名输入 main。

在 main.cpp 中输入下列内容：

```
Main.cpp
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include "TravelManipulator.h"
#include "Snow.h"
Int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    //加上雪的效果
    Snow * snow = new Snow() ;
    snow ->m_strFileName = "Images/particle.rgb" ;
    snow ->m_fTheSpeed = 1.0 ;
    snow ->m_fTheNum   = 500 ;
    snow ->m_fRotateDe = 0.6 ;
    snow ->m_fWindX = 1.0;
    snow ->m_fTheSize = 0.75 ;
    snow ->m_fTheSize2 = 1.5 ;
    ((osg::Group *)node)->addChild ( snow ->CreateSnow ((osg::Group *)node));
    viewer->setSceneData(node);
    //加上漫游功能
    TravelCamera * travel = TravelCamera::TravelScene (viewer) ;
    travel ->Active () ;
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
}
```

```

    }
    viewer->sync();
    return 0;
} //main 结束

```

运行结果:

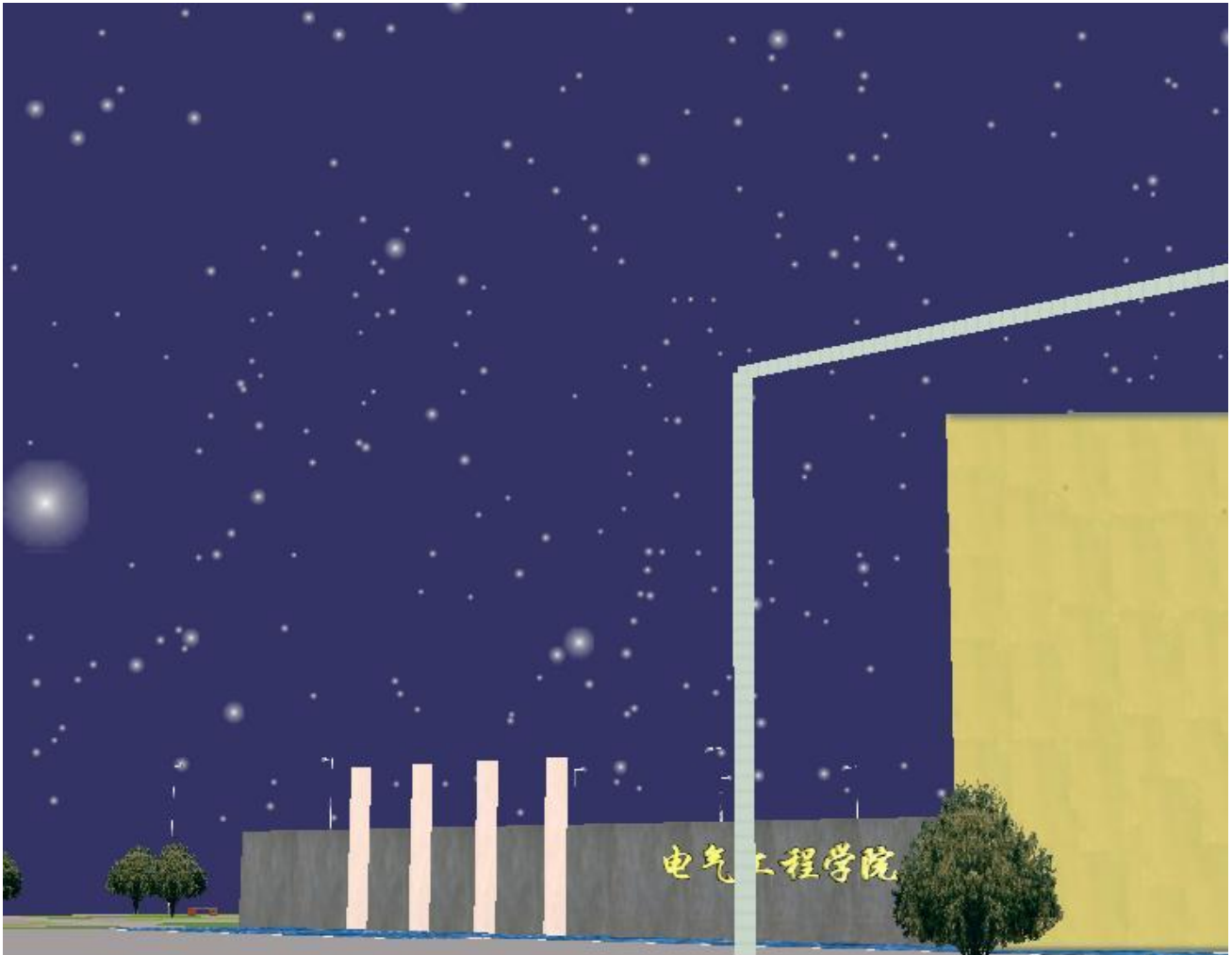


图 5.11

下面是测试雨的代码:

Main.cpp

```

#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include "TravelManipulator.h"
#include "Snow.h"
Int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    //加上雨的效果
    Snow * rain = new Snow();
}

```



```

rain ->m_bTheRain  = TRUE ;
rain ->m_fTheNum = 1000 ;
rain ->m_strFileName = "rain.rgb" ;
rain ->m_fTheSize  = 1.00 ;
rain ->m_fTheSize2 = 1.5;
rain ->m_fRotateDe = 0.0 ;
rain ->m_fTheSpeed = 10 ;
rain ->m_fWindX = 0 ;
((osg::Group *)node)->addChild ( rain ->CreateSnow ((osg::Group *)node)) ;
//加上漫游功能
TravelCamera * travel = TravelCamera::TravelScene (viewer) ;
travel ->Active () ;
viewer->realize();
while (!viewer->done())
{
    viewer->sync();
    viewer->update();
    viewer->frame();
}
viewer->sync();
return 0;
} //main 结束

```

运行效果:



图 5.12

5.3.2 喷泉的模拟

现在我们来建立一个喷泉的 DLL，现在先来讨论一下需要引出哪些接口：

粒子数：float m_fTheNum;

粒子大小：float m_fTheSize;

初速度：float m_fTheSpeech;

摆放位置：osg::Vec3 m_VecPosition;

现在我们来建立喷泉的动态链接库:Conduit.DLL

示例十四：喷泉的模拟

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：Conduit 在应用程序设置中选 DLL 与空项目选项。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB: OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib osgUtil.d.lib

第三步：菜单项目→添加类，类名填入 Conduit。把头文件与源文件名分别在前面加个 C，即 Conduit.h 一般类前加 C 代表 CLASS，头文件中 MFC 默认不加此 C，但是有时候需要加上。

而后在 Conduit.h 中加入以下内容：

```
Conduit.h
//by freesouth
#ifndef CONDUIT_DLL
#else
#define CONDUIT_DLL _declspec(dllexport)
#endif
#include <osgProducer/Viewer>
#include <osg/Group>
#include <osg/Geode>
#include <osgParticle/Particle>
#include <osgParticle/PointPlacer>
#include <osgParticle/ParticleSystem>
#include <osgParticle/ParticleSystemUpdater>
#include <osgParticle/ModularEmitter>
#include <osgParticle/ModularProgram>
#include <osgParticle/RandomRateCounter>
#include <osgParticle/RadialShooter>
#include <osgParticle/AccelOperator>
#include <osgParticle/FluidFrictionOperator>
class CONDUIT_DLL Conduit
{
public:
    Conduit(void);
    ~Conduit(void);
```

```

    osgParticle::ParticleSystem * CreateConduitScene(osg::Group* root);
    float m_fTheNum;
    float m_fTheSize;
    float m_fTheSpeech;
    osg::Node * CreateConduit(osg::Group * root);
    osg::Vec3 m_VecPosition;
}; //Conduit.h 结束

```

而后在 Conduit.cpp 中加入以下内容：

Conduit.cpp

```

//OSGFreeS   Source code by FreeSouth
#define CONDUIT_DLL _declspec(dllexport)
#include "Conduit .h"
Conduit::Conduit(void)
: m_fTheNum(800)
, m_fTheSize(0.2)
, m_fTheSpeech(100)
{
    //默认位置
    m_VecPosition.set (15, 64, 3);
}
Conduit::~~Conduit(void)
{
}
osgParticle::ParticleSystem * Conduit::CreateConduitScene(osg::Group* root)
{
    osgParticle::Particle ptemplate;
    //生周期为 2，对于喷泉已经够了
    ptemplate.setLifeTime(2);
    //设置图形变化范围
    ptemplate.setSizeRange(osgParticle::rangef(0.1f, 0.1f));
    //设置透明度变化范围
    ptemplate.setAlphaRange(osgParticle::rangef(1.0f, 0.5f));
    //设置颜色范围
    ptemplate.setColorRange(osgParticle::rangev4(
        osg::Vec4(0.7f, 1.0f, 1.0f, 1.5f),
        osg::Vec4(0.8f, 0.8f, 1.0f, 0.0f)));
    //设置半径
    ptemplate.setRadius(m_fTheSize);
    // 设置重量
    ptemplate.setMass(0.05f);
    osgParticle::ParticleSystem *ps = new osgParticle::ParticleSystem;
    ps->setDefaultAttributes("Images/smoke.rgb", false, false); //纹理
    //加入模版

```

```

ps->setDefaultParticleTemplate(ptemplate);
//建立发射器,中包含发射枪,数目及位置设定
osgParticle::ModularEmitter *emitter = new osgParticle::ModularEmitter;

//加入模版及总属性
emitter->setParticleSystem(ps);
//数目变化
osgParticle::RandomRateCounter *counter = new osgParticle::RandomRateCounter;
//数目变化,当前场景中的粒子数目
counter->setRateRange(m_fTheNum, m_fTheNum);
//加入到发射器中
emitter->setCounter(counter);
//设置位置
osgParticle::PointPlacer *placer = new osgParticle::PointPlacer;
placer->setCenter(m_VecPosition);
//加入到发射器中
emitter->setPlacer(placer);
//设置发射枪,可以设置初速度等
osgParticle::RadialShooter *shooter = new osgParticle::RadialShooter;
//设置初速度
shooter->setInitialSpeedRange(m_fTheSpeech, 0);
emitter->setShooter(shooter);
root->addChild(emitter);

//设置影响操作,
osgParticle::ModularProgram *program = new osgParticle::ModularProgram;
program->setParticleSystem(ps);

//速度操作
osgParticle::AccelOperator *op1 = new osgParticle::AccelOperator;
op1->setToGravity();
program->addOperator(op1);
osgParticle::FluidFrictionOperator *op3 = new osgParticle::FluidFrictionOperator;
op3->setFluidToAir();
program->addOperator(op3);
root->addChild(program);
osg::Geode *geode = new osg::Geode;
geode->addDrawable(ps);
root->addChild(geode);
return ps;
}
osg::Node * Conduit::CreateConduit(osg::Group * root)
{
    osgParticle::ParticleSystem *ps2 = CreateConduitScene(root);

```

```

    osgParticle::ParticleSystemUpdater *psu = new osgParticle::ParticleSystemUpdater;
    psu->addParticleSystem(ps2);
    return psu ;
} //Conduit.cpp 结束

```

编译运行。

现在我们来测试这个 DLL 的功能是否正常：

第一步： 菜单文件→新建→项目→WIN32/WIN32 控制台应用程序，项目名取 testConduit，在应用程序设置中点击空项目后完成。

第二步： 菜单项目→testConduit 属性→配置属性→链接器→命令行中添加：OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib osgUtil.d.lib **Conduit.lib,Conduit.h** 注意，要添加刚才我们设计的 lib。

第三步： 除此外我们要在场景中应用漫游功能，把刚才 Conduit 项目以下三个文件拷贝到该项目目录中：Conduit.dll,Conduit.lib,Conduit.h，（该项目目录是指放有该项目源文件的目录）

第四步： 在解决方案视图中，对源文件文件夹点右键→添加→新建项→代码/CPP 文件，文件名输入 main.

在 main 中添加如下内容：

```

#include <osgProducer/Viewer>
#include "Conduit .h"
int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    osg::Group * node = new osg::Group ;
    //在此处加入喷泉
    Conduit od ;
    od.m_VecPosition .set (0, 0, 0) ;
    node ->addChild (od.CreateConduit (node));

    viewer->setSceneData(node);
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
    viewer->sync();
    return 0;
}

```

运行效果：



图 5.13

5.3.3 雾

我们来看一下 OSG 中关于雾的类，FOG 是 StateAttribute 下的类，所以使用起来非常方便：

```

类: Fog
//查看雾的属性是否被开启
virtual bool  getModeUsage (ModeUsage &usage) const
//设置/得到模式，分为线性雾与全局雾，线性雾时，可以设置浓度渐变，全局雾全局浓度相同
void  setMode (Mode mode)
Mode  getMode () const
//设置/得到密度
void  setDensity (float density)
float  getDensity () const
//设置/得到线性雾开始点浓度
void  setStart (float start)
float  getStart () const
//设置/得到线性雾结束点的浓度
void  setEnd (float end)
float  getEnd () const
//设置/得到雾的颜色，这个不常用
void  setColor (const Vec4 &color)
const Vec4 &  getColor () const

```

我们现在来设计一个控制雾的 DLL。

示例十五：雾的 DLL

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：FogControl 在应用程序设置中选 DLL 与空项目选项。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB: OpenThreadsWin32d.lib

Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib
osgTerrain.d.lib osgTextd.lib osgUtil.d.lib

第三步：菜单项目→添加类，类名填入 FogControl。

在 FogControl.H 中添入以下内容：

```
FogControl.H
//By FreeSouth for OSGBFreeS source code
#ifdef FOGCONTROL_DLL
#else
#define FOGCONTROL_DLL _declspec(dllexport)
#endif
#include<osg/Fog>
#include<osg/Group>
class FOGCONTROL_DLL FogControl
{
//构造析构
public:
    FogControl(void);
    ~FogControl(void);
    // 雾的颜色
    osg::Vec4 m_color;
    // 浓度
    float m_fDensity;
    // 线性雾近点浓度
    float m_fLinearNear;
    // 线性雾远点浓度
    float m_fLinearFar;
    // 创建雾场景
    void CreateFogScene(osg::Group* rootnode);
    // 关闭雾场景
    void CloseFogScene(osg::Group* rootnode);
    // 雾的类型
    bool m_fogType;
}; //FogControl.h 结束
```

在 FogControl.cpp 中加入以下内容：

```
FogControl.cpp
//By FreeSouth for OSGBFreeS source code
#define FOGCONTROL_DLL _declspec(dllexport)
#include ".\fogcontrol.h"
FogControl::FogControl(void)
: m_fDensity(0.01)
, m_fLinearNear(20)
, m_fLinearFar(10)
```

```

, m_fogType(false)
{
    //设置颜色
    m_color.set (1.0, 1.0, 1.0, 1.0);
}
FogControl::~FogControl(void)
{
}
// 创建雾场景
void FogControl::CreateFogScene(osg::Group* rootnode)
{
    //创建一场大雾
    osg::ref_ptr<osg::Fog>fog = new osg::Fog;
    fog->setColor(osg::Vec4(1.0,1.0,1.0,1.0));
    if (!m_fogType)
    {
        //设置模式
        fog->setMode(osg::Fog::EXP);
        //设置浓度
        fog->setDensity(m_fDensity);
    }
    else
    {
        //设置模式
        fog->setMode(osg::Fog::LINEAR);
        //设置浓度
        fog->setDensity(m_fDensity);
        //设置线性值
        fog->setStart(m_fLinearNear);
        fog->setEnd(m_fLinearFar);
    }
    rootnode->getOrCreateStateSet()->setAttributeAndModes(fog.get(),osg::StateAttribute::ON);
}
// 关闭雾场景
void FogControl::CloseFogScene(osg::Group* rootnode)
{
    osg::ref_ptr<osg::Fog>fog = new osg::Fog;
    rootnode->getOrCreateStateSet()->setAttributeAndModes(fog.get(),osg::StateAttribute::OFF);
}

```

编译运行。

现在对该动态链接库进行一下测试：

第一步： 菜单文件→新建→项目→WIN32/WIN32 控制台应用程序，项目名取 testFogControl，在应用程序设置中点击空项目后完成。

第二步： 菜单项目→testConduit 属性→配置属性→链接器→命令行中添加： OpenThreadsWin32d.lib

Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib
osgTerraind.lib osgTextd.lib osgUtild.lib **TravelManipulator.lib,FogControl.lib** 注意，要添加刚才我们设计的
lib。具备漫游功能

第三步：除此外我们要在场景中应用漫游功能，把刚才 FogControl 项目以下三个文件拷贝到该项目目录中：FogControl.dll,FogControl.lib, FogControl.h，（该项目目录是指放有该项目源文件的目录）

第四步：在解决方案视图中，对源文件文件夹点右键→添加→新建项→代码/CPP 文件，文件名输入 main。
在 main.cp 中输入：

Main.cpp

```
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include "TravelManipulator.h"
#include "FogControl.h"
Int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();

    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    //加入雾的控制
    FogControl fogControl ;
    fogControl.CreateFogScene ((osg::Group*)node) ;
    viewer->setSceneData(node);
    //加入漫游功能
    TravelCamera * travel = TravelCamera::TravelScene (viewer) ;
    travel ->Active () ;
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
    viewer->sync();
    return 0;
}
```

编译运行。

运行效果：



图 5.14

本章对 OSG 中的粒子系统做了最基本的介绍，有了本章的基础，做一些小的粒子系统的东西就可以实现了。

第六章 更新&回调

本章我们来学习一下 osg 中常用的控制方法，以及对结点的访问。

关于 osg 中的常用更新控制方法主要通过键盘鼠标，路径，矩阵操作这三大类。通过本章的学习，读者应掌握 osg 中的事件列表的使用方法，NODE 序列的概念，使用矩阵更新。

这一章：我们将做几个例子，利用 `osgFX::Scribe` 做一个 pick 的例子，使用 `EventHandler` 和 `HUD` 做一个显示鼠标指针指向的坐标的例子与简单的显示汉字，和访问已存在模型中的结点。

首先我们来看一下很多初学者最关心的问题：对于鼠标与键盘的响应。在这里我们只介绍在 `osgProducer` 中常使用的 `EventHandler`，`sceneView` 中使用的 `KeyBoardCallBack` 可以自己看一下 osg 中的例子。从某种意义上讲更新就是创建一种监听效果。

6.1 Event&HUD

`NodeCallback` 下我们最经常使用的几个 `Callback` 类，其中包括所有的矩阵操作器。`osgProducer::Viewer` 中含有一个 `EventHandlerList` 变量，且有针对该变量的：`getEventHandlerList ()->pushback` 操作。

`osg::AnimationPathCallback` 可以使用固定路径对其进行更新，让场景中的物体可以来回走动。

`osgUtil::TransformCallback` 可以使用移动进行更新，其实更常用的是 `AnimationPathCallback` 与 `MatrixTransform` 的结合。

现在我们来首先来看一下 `EventHandler`：

6.1.1 事件响应

首先来看一下 `EventHandler` 的被执行过程：如图所示：

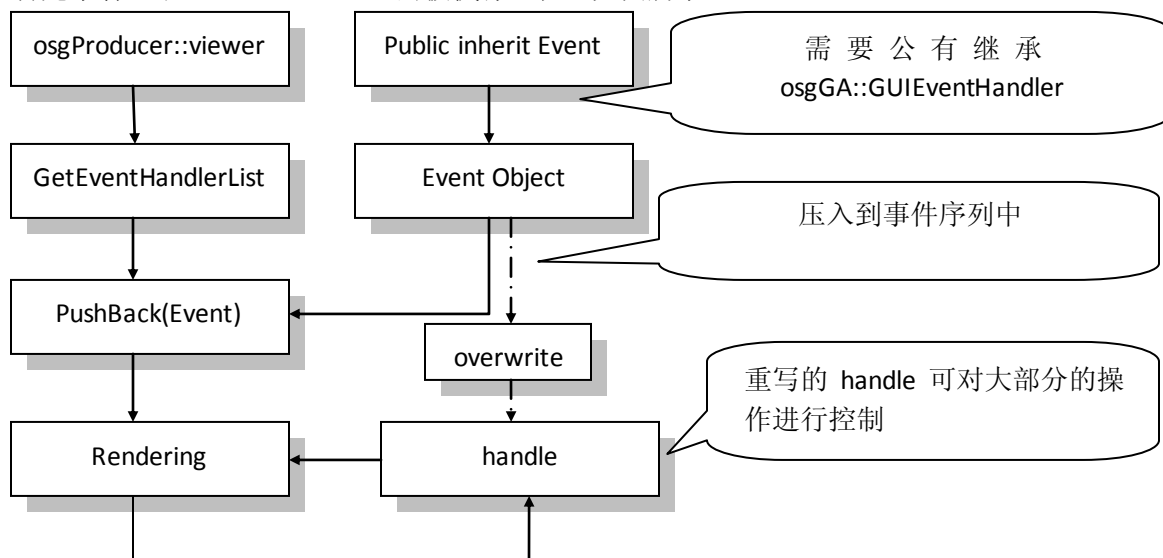


图 6.1

从上图中可以清楚的看到，接受一件总被监听的事件分为以下步骤：

第一步从 `osgGA::EventHandlerList` 中继承一个共有类并实现其中 `Handler` 操作，第二步从 `Viewer` 中得到此事件序列，且把该类压入到事件序列中去。这就是添加事件控制的总过程。

现在来看一下 `Handler` 中可以控制一些什么操作：

```
handle (const GUIEventAdapter &, GUIActionAdapter &):
GUIActionAdapter 中的两个操作一般极少使用，下面我们只来看一下 GUIEventAdapter
//设置与得到事件类型，setEventType 相当于 SendMessage(WM_KEYDOWN,,,,,等等
void setEventType (EventType Type)
virtual EventType getEventType () const
我们来看一下其中常用的事件类型：
EventType:使用枚举值，按序列值++
    NONE = 0,//无事件
PUSH//鼠标键按下，通过 getButton 可以知道是左键还是右键,还是中间键
RELEASE//鼠标键弹开，可以先知道按下的是哪个鼠标键，而后再进行处理
DOUBLECLICK//双击，
DRAG,//拖动，可以得到按下，再得到弹起，从而处理在左键按下拖动的情况
MOVE,//移动，这个基本上每时每刻都在发生，与 DRAG 可以重叠
KEYDOWN, //判断哪个键按下
KEYUP, //判断是哪个键弹起
//设置/得到事件响应时间，量纲是秒
void setTime (double time)
double getTime () const
double time () const
//设置/得到键盘上键按下或弹起的信息
void setKey (int key)
virtual int getKey () const
//设置/得到鼠标上某键按下或弹起的信息
void setButton (int button)
int getButton () const
//设置/得到窗口尺寸，一般而言认为左上角是 (0,0)，右下角是 (1,1)
void setWindowSize (float Xmin, float Ymin, float Xmax, float Ymax)
void setXmin (float x)
float getXmin () const
void setXmax (float x)
float getXmax () const
void setYmin (float y)
float getYmin () const
void setYmax (float y)
float getYmax () const
//设置/得到鼠标当前在窗口的位置，与 setWindowSize 等设置窗口的量纲有关
void setX (float x)
float getX () const
void setY (float y)
float getY () const
```

```

//设置/得到当前鼠标键的状态
void setButtonMask (unsigned int mask)
unsigned int getButtonMask () const
//设置/得到键盘上键状态
void setModKeyMask (unsigned int mask)
unsigned int getModKeyMask () const
//设置鼠标轮的种种活动，滚动的单位变化值等等
void setScrollingMotion (ScrollingMotion motion)
float getScrollingDeltaX () const
float getScrollingDeltaY () const
void setScrollingMotionDelta (float x, float y)
ScrollingMotion getScrollingMotion () const
//把 x 坐标规格化，为 (-1,1) 范围，-1 为最左边
float getXnormalized () const
//把 y 坐标规格化，为 (-1,1) 范围，1 为最右边
float getYnormalized () const

```

以上所有的事件几乎涵盖了所有的正常的操作。下面我们来看一个小例子：
例子功能，通过点击鼠标左键使场景中的物体显示与隐藏：

示例十七：Event

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testEvent 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib
osgUtil.d.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

第四步：菜单项目→添加类，类名填入 CPickHandler，在基类中添入 osgGA::GUIEventHandler
如果出现提示说什么没有发现这个基类，不用管它，继续添加。

首先我们来处理 PickHandler.h 在其中添入如下内容：

```

PickHandler.h
//By FreeSouth for osgFreeS Source code
#pragma once
#include <osgGA/GUIEventHandler>
#include <osgProducer/Viewer>
//此类为从 GUIEventHandler 继承而来
class CPickHandler : public osgGA::GUIEventHandler
{
public:
    CPickHandler(osgProducer::Viewer* viewer);
    //主事件响应函数

```

```

        virtual      bool      handle(const      osgGA::GUIEventAdapter&      ea,
osgGA::GUIActionAdapter& aa) ;
    protected:
        ~CPickHandler(void) ;
    private:
        //用来获取 viewer 中的模型
        osgProducer::Viewer* m_Viewer;
        //控制是否显示场景中的模型
        bool m_bShow ;
}; //PickHandler.h 结束

```

再来看一下 PickHandler.cpp 是内容:

```

pickHandler.cpp
#include "Pickhandler.h"
CPickHandler::CPickHandler(osgProducer::Viewer* viewer)
    : m_Viewer(viewer) , //初始化
    m_bShow(true) //默认时显示
{
}
CPickHandler::~CPickHandler(void)
{
}
bool      CPickHandler::handle(const      osgGA::GUIEventAdapter&      ea,
osgGA::GUIActionAdapter& aa)
{
    //得到事件类型
    switch(ea.getEventType())
    {
        //如果为左键按下
        case(osgGA::GUIEventAdapter::PUSH):
            if (ea.getButton() & osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON)
            {
                m_bShow = !m_bShow ;

                //得到场景中的数据且隐藏/显示
                m_Viewer ->getSceneData() ->setNodeMask (m_bShow) ;
            }
            return true;
        }
    return false;
} //pickHandler.cpp

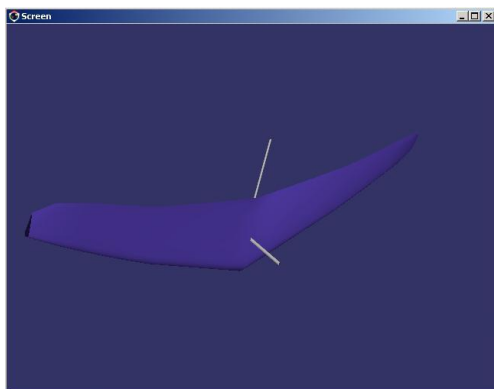
```

第一次点击时显示，那么第二次显示时就隐藏。

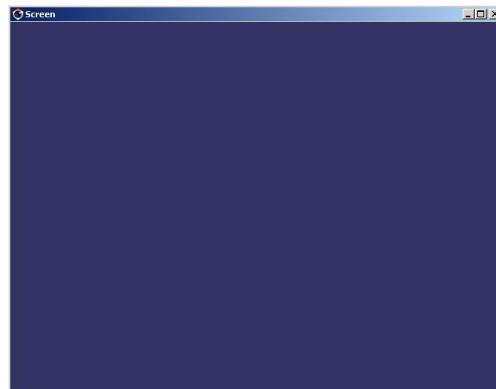
现在来添加 main.cpp 中的内容:

```
Main.cpp
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include "PickHandler.h"
int main(int, char**)
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer();
    viewer.setSceneData(osgDB::readNodeFile("glider.osg"));
    //压入事件
    viewer.getEventHandlerList().push_front(new CPickHandler(&viewer));
    viewer.realize();
    while (!viewer.done())
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    return 0;
}
```

运行效果:



a: 点击出现



b: 点击消失

图 6.2

6.1.2 HUD 与显示汉字

在场景量示 3D 的汉字, 像一座建筑物一样摆在那儿, 往往是不需要的, 立体字是在建模时期就做好的内容, 不需要用户过多的干预。但是 HUD, 确实是百分百程序员的活儿。创建 3D 字与创建 HUD 的过程几乎是一样的, 因此我们这里来讲一下关于 OSG 中创建文字的一些知识。

文字显示的原理:

文字在 3D 场景中显示往往要经历以下几步：

读取字体点阵信息→转化为图像→反走样→最终图像。

在反走样期间可以处理可种模糊效果，在最终图像形成时可以设置如何摆放。

在这里我不免要感叹一下 osg 的优秀，它总是那么的措手不及的提供给人们方便。

我们先来看一下 Text 中有哪些常用的操作：

类 Text

//设置字体，虽然说字体有单独的类，但是很少使用，但是字体贴图是在这个阶段形成的

```
void setFont (Font *font=0)
```

//设置/得到字体，如 setFont("fonts/SIMYOU.TTF");

```
void setFont (const std::string &fontfile)
```

```
const Font * getFont () const
```

//设置/得到字体显示的宽高

```
void setFontResolution (unsigned int width, unsigned int height)
```

```
unsigned int getFontWidth () const
```

```
unsigned int getFontHeight () const
```

//设置/得到文字的具体内容

```
void setText (const String &text)
```

```
void setText (const std::string &text)
```

```
void setText (const std::string &text, String::Encoding encoding)
```

```
void setText (const wchar_t *text)
```

```
String & getText ()
```

```
const String & getText () const
```

//更新

```
void update ()
```

//设置/得到字体的大小，以及比率，一般后一个参数不会使用，否则会失真（改变高宽比）

```
void setCharacterSize (float height, float aspectRatio=1.0f)
```

```
float getCharacterHeight () const
```

//得到文字的高宽比

```
float getCharacterAspectRatio () const
```

//设置/设置字体大小模式，决定字体是否随着距离的改变大小形状是否会改变。

```
void setCharacterSizeMode (CharacterSizeMode mode)
```

```
CharacterSizeMode getCharacterSizeMode () const
```

//设置/得到高宽，这将固字字体的显示范围

```
void setMaximumWidth (float maximumWidth)
```

```
float getMaximumWidth () const
```

```
void setMaximumHeight (float maximumHeight)
```

```
float getMaximumHeight () const
```

//设置/得到字体的显示位置

```
void setPosition (const osg::Vec3 &pos)
```

```
const osg::Vec3 & getPosition () const
```

//设置/得到摆放方式，从左到右 `LEFT_TO_RIGHT`，从右到左 `RIGHT_TO_LEFT`，垂直 `VERTICAL`

```
void setAlignment (AlignmentType alignment)
```

```
AlignmentType getAlignment () const
```



```

//设置依赖平面, XY_PLANE 表示 XY 平面, 等等
void setAxisAlignment (AxisAlignment axis)
//设置/得到旋转
void setRotation (const osg::Quat &quat)
const osg::Quat & getRotation () const
//设置/得到是否自动面朝屏幕, 有 Billboard 的意思
void setAutoRotateToScreen (bool autoRotateToScreen)
bool getAutoRotateToScreen () const
//设置/得到排列方式从左到右 LEFT_TO_RIGHT , 从右到左 RIGHT_TO_LEFT , 垂直 VERTICAL
void setLayout (Layout layout)
Layout getLayout () const
//设置/得到颜色
void setColor (const osg::Vec4 &color)
const osg::Vec4 & getColor () const
//设置/得到阴影类型
void setBackdropType (BackdropType type)
BackdropType getBackdropType () const
//设置/得到阴影的离开程度与方向
void setBackdropOffset (float offset=0.07f)
void setBackdropOffset (float horizontal, float vertical)
float getBackdropHorizontalOffset () const
float getBackdropVerticalOffset () const
//设置/得到阴影颜色
void setBackdropColor (const osg::Vec4 &color)
const osg::Vec4 & getBackdropColor () const
//设置阴影实现方式
void setBackdropImplementation (BackdropImplementation implementation)
BackdropImplementation getBackdropImplementation () const
//设置/得到颜色映射方式, 可以得到渐变效果
void setColorGradientMode (ColorGradientMode mode)
ColorGradientMode getColorGradientMode () const
void setColorGradientCorners (const osg::Vec4 &topLeft, const osg::Vec4 &bottomLeft, const osg::Vec4 &bottomRight, const osg::Vec4 &topRight)
const osg::Vec4 & getColorGradientTopLeft () const
const osg::Vec4 & getColorGradientBottomLeft () const
const osg::Vec4 & getColorGradientBottomRight () const
const osg::Vec4 & getColorGradientTopRight () const
//得到 BoundingBox
virtual osg::BoundingBox computeBound () const

```

下面我们来使用上面的操作来显示汉字:

示例十八: 显示汉字

第一步: 新建→项目→WIN32/WIN32 控制台应用程序, 项目名称填入: text 其它的保持默认。

第二步: 在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB:

OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib
osgUtil.d.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为main.cpp

第四步：在其中键入如下代码：

Main.cpp

```
#include <osgProducer/Viewer>
#include <osg/Geode>
#include <osgText/Font>
#include <osgText/Text>
osg::Group* create3DText()
{
    //申请一个画图节点
    osg::Geode* geode = new osg::Geode;
    //设置字体大小为 300
    float characterSize=300.0f;

    //设置位置
    osg::Vec3 pos(osg::Vec3(0, 0, 0));

    //设置字体
    osgText::Text* text1 = new osgText::Text;
    //注意：这里设置的是幼圆体，关于如何查看字体名称，可以把字体复制到任何位置查看，在
    WINDOWS/FONTS 中的中文字体名称往往是不正确的，它显示的是字体代表什么样的字体的名称（好绕）
    text1->setFont("fonts/SIMYOU.TTF");

    //设置字体大小与高宽比
    text1->setCharacterSize(characterSize,1.0);
    //设置位置
    text1->setPosition(pos);

    //设置阴影颜色与离开值，类型
    text1->setBackdropColor(osg::Vec4(0.0, 1, 0,1));
    text1->setBackdropOffset(0.05);
    text1->setBackdropType(osgText::Text::DROP_SHADOW_BOTTOM_RIGHT);
    //设置依赖方式，选择为屏幕依赖，无论怎么旋转都会面朝屏幕
    text1->setAxisAlignment(osgText::Text::SCREEN);

    //设置字体大小模式，为与屏幕相关
    text1->setCharacterSizeMode(osgText::Text::SCREEN_COORDS);

    //必须显示为宽字，如果不能显示汉字，请查看 CMD，是怎么回事，一般为找不到字体
```

```

    text1->setText(L"显示汉字");
    geode->addDrawable(text1);
    osg::Group* rootNode = new osg::Group;
    rootNode->addChild(geode);
    return rootNode;
}

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

    //得到创建的字体
    viewer.setSceneData( create3DText() );
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();

    return 0;
} //Main.cpp 结束

```

编译运行效果如图：



图 6.3

这里要再重述一下关于显示汉字的要点，第一必须得确定你有汉字字体，系统的字体文件存放在：
C:\WINDOWS\FONTS 下，点击最后几个查看，查看名称是把该字体复制（必须使用快捷键）出到任何地方，便可以看到文件名。

第二必须设置成宽字，关于宽字在第一章有述，现在在这里再重述一下：

关于字符之间的转换，平时使用的下面几种字符在 VS2005 中属于 ASCII 字符，简称 A 类型字符：

ASCII 字符（窄字符）：CHAR char LPCSTR LPSTR PCHAR PCSTR PSTR std::string

Unicode 字符（宽字符）：LPCWSTR LPWSTR PCWSTR PTCHAR PWCHAR PWSTR WCHAR CString

以及在 VS2005 下所有 T 字符类型

最常用的转换 A 为窄字符 W 为宽字符：

A to W 的方式有：

```
W = _T("A") ; W = TEXT("A") ; W = L"A" ; W = A2W(A) ;
```

W to A 的方式有：

```
A = W2A(W) ; A = W2BSTR(W) ;
```

6.1.3 显示当前位置的例子

当鼠标在屏幕滑动时，对应 3D 的场景中总会有一个坐标，也就是从屏幕上发出一条射线，与 3D 模型的场景会有很多的交点，我们取最后一个交点为鼠标在世界坐标系中的坐标。

在这里我们来说一下该 DLL 的原理：

首先第一步，压入事件，第二步响应左键 PICK，射出射线，第三步，得到交点，取最后一个交点，第四步，用 HUD 显示在屏幕上，注意在事件中可以更新文字，因此必须在事件中加入文字指针指向文字变量。

申请一个 Camera，把 Camera 设置为最后渲染，保持其在最上，把 Camera 加入到整个结点中。在以后几章中，我们将详解 Camera。

下面以图示来表示整个原理：

如图：

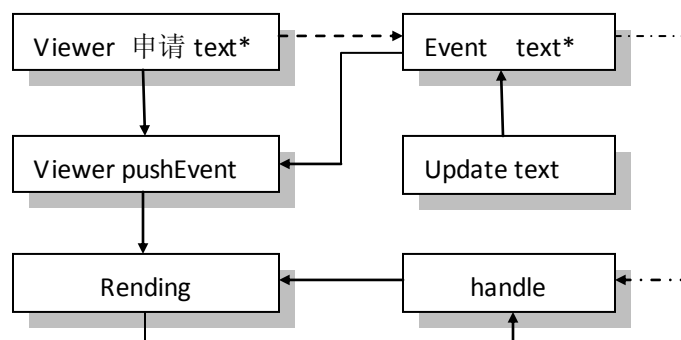


图 6.4

下面来创建这个 DLL：功能：显示当前位置与鼠标所指模型的位置

示例十九：显示当前位置

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：ShowPosition 在应用程序设置中选 DLL 与空项目选项。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib osgUtil.d.lib

第三步：菜单项目→添加类，类名填入 ShowPosition，在基类中输入 osgGA::MatrixManipulator。如果不提示无此基类，是否继续添加，不必理会，继续添加。

而后在 ShowPosition.h 中加入以下内容:

```
ShowPosition.h
//osgFrees Source code by FreeSouth
#ifdef SHOWPOSITION_DLL
#else
#define SHOWPOSITION_DLL __declspec(dllexport)
#endif
#include <osgProducer/Viewer>
#include <osg/Geode>
#include <osgText/Text>
#include <sstream>
class SHOWPOSITION_DLL PickHandler :
    public osgGA::GUIEventHandler
{
public:
    PickHandler(void);
    ~PickHandler(void);
public:
    //把 viewer 传进来,把数据更新的传进来
    PickHandler(osgProducer::Viewer* viewer,osgText::Text* updateText):
        _viewer(viewer),
        _updateText(updateText) {}
    //事件响应设备
    bool handle(const osgGA::GUIEventAdapter& ea,osgGA::GUIActionAdapter&
us);

    //pick 事件发生
    virtual void pick(const osgGA::GUIEventAdapter& ea);
    //设置字符串具体内容
    void setLabel(const std::string& name);
    void setPosition()
    { //设置位置
        const double *a = _viewer ->getPosition();
        position.set (a[0], a[1], a[2]);
    }
protected:
    osgProducer::Viewer* _viewer;
    osg::ref_ptr<osgText::Text> _updateText;
    osg::Vec3 position ;
public:
    ;
};
//创建 HUD 的类
class SHOWPOSITION_DLL CreateHUD
```

```

{
public:
    CreateHUD() {} ;
    ~CreateHUD() {} ;
    osg::Node * createHUD(osgText::Text* updateText)
    {
        //设置一个 CameraNode 来设置 TEXT
        osg::CameraNode* hudCamera = new osg::CameraNode;
        //设置绝对参考
        hudCamera->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
        //取屏幕
        hudCamera->setProjectionMatrixAsOrtho2D(0,1280,0,1024);
        //使用默认的矩阵做为视口矩阵
        hudCamera->setViewMatrix(osg::Matrix::identity());
        //设置成为 POST 渲染顺序
        hudCamera->setRenderOrder(osg::CameraNode::POST_RENDER);
        //深度清理,还有一个是颜色清理
        hudCamera->setClearMask(GL_DEPTH_BUFFER_BIT);
        //选择一种字体
        std::string timesFont("fonts/COURBD.ttf");
        // 把灯关了,而且要关掉深度,要始终保持在最上面
        osg::Vec3 position(750.0f,1000.0f,0.0f);

        //字体间的间距是多少
        osg::Vec3 delta(0.0f,-60.0f,0.0f);
    /* {不需要
        //加一几何结点
        osg::Geode* geode = new osg::Geode();
        //创建一个状态
        osg::StateSet* stateset = geode->getOrCreateStateSet();
        //关灯
        stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
        //关深度
        stateset->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
        //设置一个易读的名称
        geode->setName("simple");
        //HUD 镜头加入该几何结点,这些操作始终如此眼花缭乱
        hudCamera->addChild(geode);
    }*/
    {
        osg::Geode* geode = new osg::Geode();
        osg::StateSet* stateset = geode->getOrCreateStateSet();
        stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
        stateset->setMode(GL_DEPTH_TEST,osg::StateAttribute::OFF);
    }
}

```

```

        geode->setName("The text label");
        geode->addDrawable( updateText );
        hudCamera->addChild(geode);
        updateText->setCharacterSize(20.0f);
        updateText->setFont(timesFont);
        updateText->setColor(osg::Vec4(1.0f,1.0f,0.0f,1.0f));
        updateText->setText("");
        updateText->setPosition(position);
        position += delta;
    }
    return hudCamera;
}
}; //ShowPosition.h 结束

```

现在来看一下：ShowPosition.cpp

```

ShowPosition.cpp
//osgFreeS Source By FreeSouth
#define SHOWPOSITION_DLL __declspec(dllexport)
#include "..\pickhandler.h"
PickHandler::PickHandler(void)
{
}
PickHandler::~~PickHandler(void)
{
}
bool PickHandler::handle(const osgGA::GUIEventAdapter&
ea, osgGA::GUIActionAdapter&)
{
    switch(ea.getEventType())
    {
        //鼠标移动
        case(osgGA::GUIEventAdapter::FRAME):
        {
            //鼠标在什么时候
            pick(ea);
            setPosition();
        }
        return false;
    default:
        return false;
    }
}
void PickHandler::pick(const osgGA::GUIEventAdapter& ea)
{
    //得到碰撞点序列

```

```

        osgUtil::IntersectVisitor::HitList hlist;
        //设置坐标
        std::string gdlist="";
        if (_viewer->computeIntersections(ea.getX(),ea.getY(),hlist))
        {
            osgUtil::IntersectVisitor::HitList::iterator hitr=hlist.begin();
            {
                char a[400] ;
                memset(a, 0, 400) ;
                //格式化字符串
                sprintf(a,"Mouse Position: %f %f %f\nViewer Position: %f %f %f",
hitr->getLocalIntersectPoint().x (), hitr->getLocalIntersectPoint().y (),
hitr->getLocalIntersectPoint().z (),position.x (),position.y (),position.z ());
                gdlist += a;
            }
        }
        setLabel(gdlist);
    }
    void PickHandler::setLabel (const std::string &name)
    {
        if (_updateText.get()) _updateText->setText(name);
    }

```

编译动行。

而后我们将测试这个 DLL:

第一步: 菜单文件→新建→项目→WIN32/WIN32 控制台应用程序, 项目名取 testShowPosition, 在应用程序设置中点击空项目后完成。

第二步: 菜单项目→test 属性→配置属性→链接器→命令行中添加: OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrainind.lib osgTextd.lib osgUtild.lib ShowPosition.lib,TravelManipulator.lib 注意, 要添加刚才我们设计的 lib。加漫游。

第三步: 除此外我们要在场景中应用漫游功能, 把刚才 testShowPosition 项目以下三个文件拷贝到该项目目录中: TravelManipulator.h,TravelManipulator.lib,TravelManipulator.dll, ShowPosition.h , ShowPosition.lib , ShowPosition.dll, (该项目目录是指放有该项目源文件的目录)

第四步: 在解决方案视图中, 对源文件文件夹点右键→添加→新建项→代码/CPP 文件, 文件名输入 main.

在 main.cpp 中输入下列内容:

```

Main.cpp
//osgFreeS Source code by FreeSouth
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>

```



```

#include "TravelManipulator.h"
#include "PickHandler.h"
#include <osgText/Text>
Int main(int, char**)
{
    //申请更新字符串
    osg::ref_ptr<osgText::Text>updateText = new osgText::Text ;
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    //该模型在光盘的 IVE 文件夹中
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    TravelCamera * travel = TravelCamera::TravelScene (viewer) ;
    travel ->Active () ;
    //加入 HUD
    CreateHUD nodeHud ;
    //设置更新字符串
    ((osg::Group *)node) ->addChild (nodeHud.createHUD (updateText.get ()) ) ;
    //压入事件, 更新
    viewer->getEventHandlerList
        ().push_front
        (new
PickHandler(viewer.get(), updateText.get())) ;
    viewer->setSceneData (node);
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
    viewer->sync();

    return 0;
} //Main.cpp 结束

```

编译运行。

这里写的测试步骤中, 按所需要步骤, 可以添加在 MFC 等各种框架中, 都是可行的。

再来重述一下本节要注意的几个地方: 在显示汉字时要确定包含该字体, 设置成宽字符。在加入事件时注意事件是个列表, 响应是有先后的顺序的。

下图是运行效果:

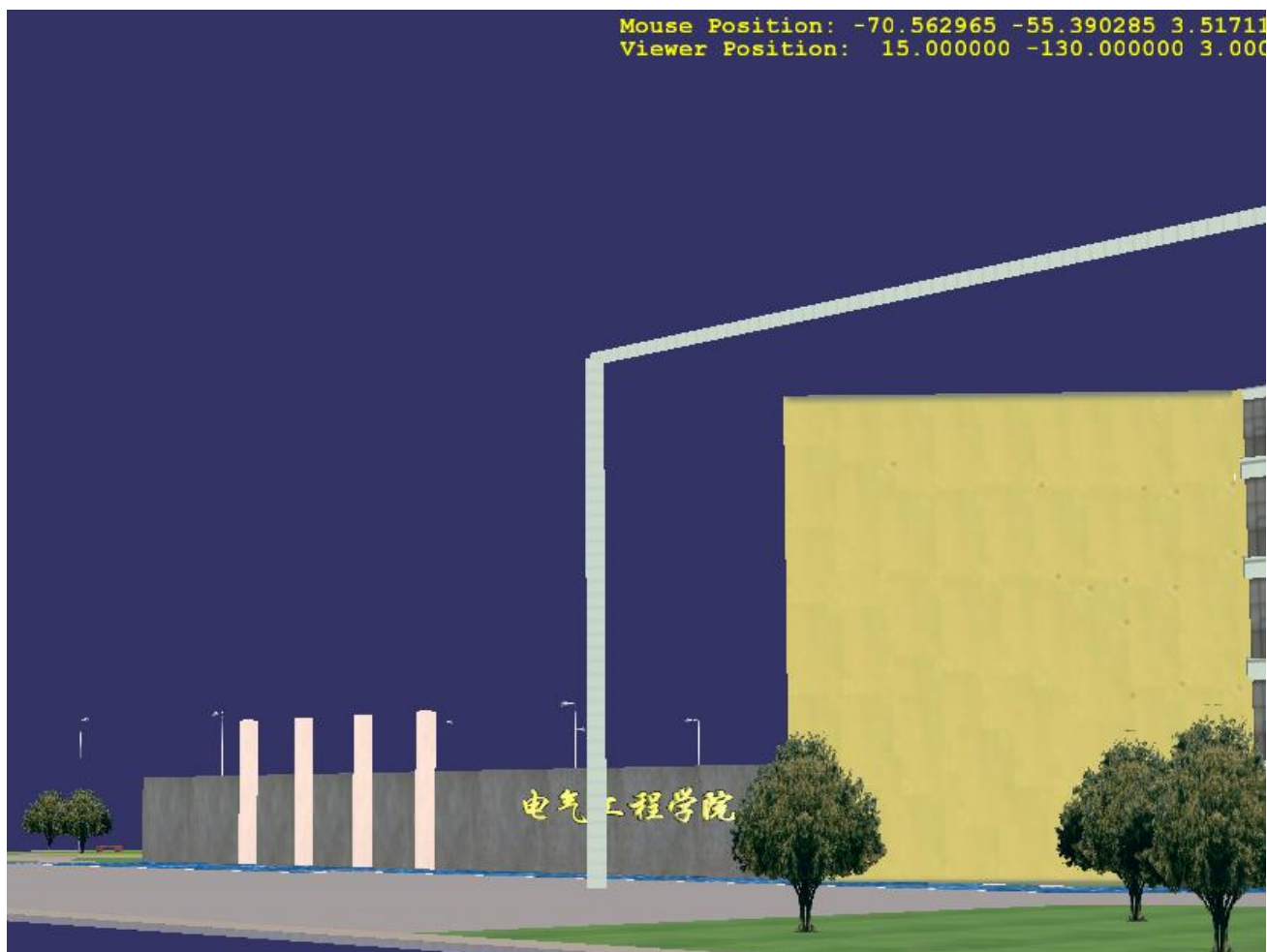


图 6.5

注意：在图片右上角显示了当前位置与鼠标所指的位置。

6.2 MatrixTransform&AnimationPath

下面我们来说明一下关于MatrixTransformCallBack与AnimationPathCallBack的使用方法，关于做简单与复杂的运动效果，几种使用方法各有所长。

下面我们来看一下具体的使用方法：

6.2.1 MatrixTransformCallBack

我们来做一个例子自己从 NodeVisitor 自己来做一个 MatrixTransform，从而为下一节学习 NodeVisitor 打基础：

示例二十：CALLBACK

现在我们要来实现，让一物体从 A 到 B 来来回回的拉锯，这里没有钜与大树模型，要不会非常形象的。

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：MyTransoform 其它的保持默认。

第二步：在菜单项目 → 属性 → 配置属性 → 链接器 -> 命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtilid.lib

第三步：对解决方案资源管理器视图中源文件击右键 → 添加 → 新建项 → 代码/CPP ，文件名取为
main.cpp

在 main.cpp 中加入以下内容：

```
Main.cpp
//osgFreeS Source by FreeSouth
#include <osgProducer/Viewer>
#include <osg/Math>
#include <osgDB/ReadFile>
#include <osg/NodeCallback>
#include <osg/MatrixTransform>
class MyTransformCallback : public osg::NodeCallback
{
public:
    MyTransformCallback(float angularVelocity)
    { //这个变量决定来来回回的速度
        m_angularVelocity = angularVelocity;
    }
    //注意这里的一个运算符重载
    virtual void operator() (osg::Node* node, osg::NodeVisitor* nv)
    {
        //在这里可以定义我们的任何操作
        osg::MatrixTransform* transform =
dynamic_cast<osg::MatrixTransform*>(node);
        if (nv && transform && nv->getFrameStamp())
        {
            //1.0 决定来来回回的步长
            double time = nv->getFrameStamp()->getReferenceTime();
transform->setMatrix(osg::Matrix::translate(0.0f,1.0f+cosf(time*_angular_velopity),0.0f));
        }
        traverse(node,nv);
    }
protected:
    float m_angularVelocity;
};
int main( int argc, char **argv )
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
```

```

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

osg::Node * node = osgDB::readNodeFile ("glider.osg") ;
osg::MatrixTransform* transform = new osg::MatrixTransform();
//定义 Callback
transform->setUpdateCallback(new MyTransformCallback(5.0f));
transform->addChild(node);
root ->addChild (transform) ;
viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}
viewer.sync();
viewer.cleanup_frame();
viewer.sync();
return 0;
} //main.cpp End

```

编译运行。

在 Operator 中可以定义任何的操作，在下一节中我们将举例说明 NodeVisitor 中的用法。

6.2.2 AnimationPathCallback

对于这个类，与 AnimationPath 相关，相信大家对它都非常的熟悉，可以自定义路径等等。是做高精度路径回调的首选：

关于 AnimationPath 的使用方法，在第四章多有描述，现在举例子说如何设置 CALLBACK:

其实与上例大同小异：

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：MyTransoform 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
 OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
 osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
 osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP ，文件名取为 main.cpp

在 main.cpp 中加入以下内容：

Main.cpp

```
//osgFreeS By FreeSouth
#include <osgDB/ReadFile>
#include <osg/MatrixTransform>
#include <osgProducer/Viewer>
#include <osg/Math>
int main( int argc, char **argv )
{
    osgProducer::Viewer viewer ;
    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS) ;

    osg::Group* root = new osg::Group;
    osg::Node * node = osgDB::readNodeFile ("glider.osg") ;
    osg::MatrixTransform *ma = new osg::MatrixTransform ;
    //AnimationPathCallback 中可以跟任何路径，稍微改一下就可以从文件中读入路径，功能很强大
    ma->setUpdateCallback (new osg::AnimationPathCallback(osg::Vec3(0, 0,
0), osg::Z_AXIS, osg::inDegrees(45.0f))) ;
    ma->addChild (node) ;
    root ->addChild (ma) ;
    viewer.setSceneData( root ) ;
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();
    return 0;
}
```

上面两个小例子做完后，多尝试几次自定义的操作，对移动的回调就会有很精确的掌握了。

6.3 NodeVisitor

这是一个极有用的类，可以访问结点序列，使用的方法大同小异，需要掌握各结点的特点，以及清楚各种结点的使用方法才能熟练的使用 NodeVisitor.

6.3.1 访问模型结点

下面我们来看一下 NodeVisitor 的工作流程：

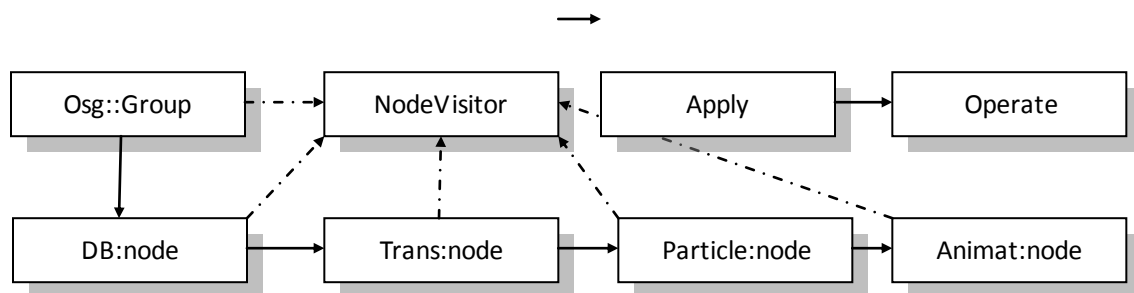


图 6.6

在主结点 accept 之后，结点数据立即传至 NodeVisitor 中去，应用 apply 函数，可以将数据定任一些操作，更多的操作还是需要硬性的制做与调用。

下面我们来举一反三简单的看一下 NodeVisitor 访问数据结点时的情景，我们现在有一个模型，这个模型是由一头牛和一个飞机组成，我们将使用 NodeVisitor 访问模型中的结点，使飞机隐藏，这个模型在 osg 文件夹中的 Cowandcessna.osg 中。在一个模型中找到某个结点可以使用很多种方法，可以得到结点的名称，看是否等于这个名称，也可以得到结点的 ID 看是否与此 ID 相同，一般而言在建模的过程当中，结点的组织是相当的有序的，找到某个结点并不是十分的困难。

下面我们来建立这个小程序：

示例二十一：CALLBACK

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：textNodeVisitor 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrainind.lib osgTextd.lib
osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入以下内容：

```

Main.cpp
//osgFreeS Source by FreeSouth
#include <osg/NodeVisitor>
#include <osgDB/ReadFile>
#include <osg/Group>
#include <osgProducer/Viewer>

class nodehide : public osg::NodeVisitor
{
public:
    nodehide() :

```

```

        osg::NodeVisitor(osg::NodeVisitor::TRAVERSE_ALL_CHILDREN)
    {
    }
    virtual void apply(osg::Group& node)
    {
//把接受结点的数据传送过来，由于我们知道是个 DB 结点，所以直接相等
        group = &node;
//向下传递
        traverse(node); //注释这一行与 setmaskt 的那一行，把 setmaskt 中的前两行取消注
释便可得到飞机，现在得到的是牛
    }
    void setmaskt()
    {
        //osg::Group * ff=(osg::Group *) (group ->getChild (0)) ;
        //ff ->getChild(0) ->setNodeMask (0);
        group ->setNodeMask (0) ;
    }
    osg::Group *group;
};
int main( int argc, char **argv )
{

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

    osg::Group* root = new osg::Group;
    //读取模型，该模型的结构会在下面给出
    osg::Node * node = osgDB::readNodeFile ("cowandcessna.osg") ;

    root ->addChild (node) ;

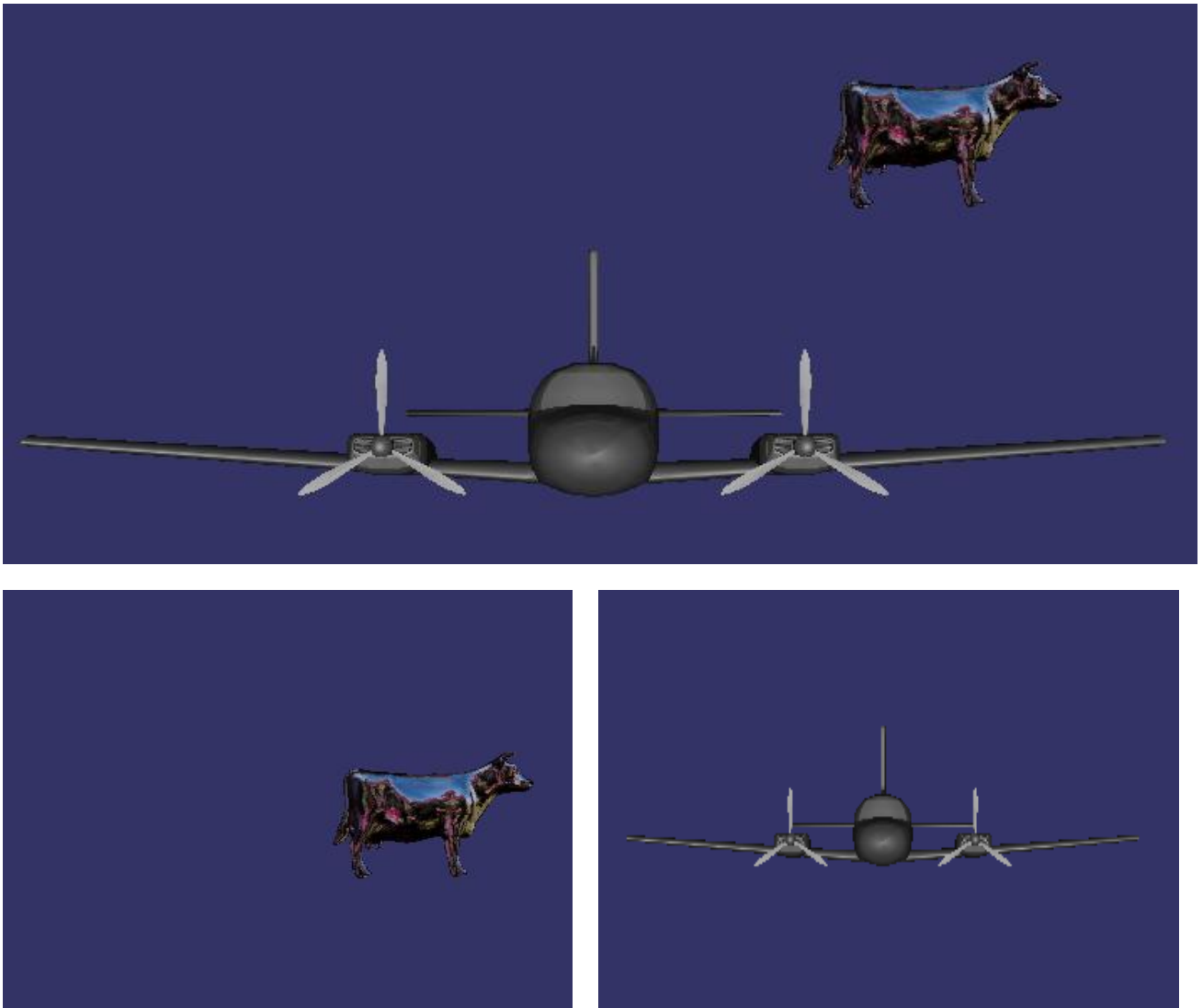
    //使用结点访问
    nodehide cc ;
    root ->accept (cc) ;
    //调用隐藏
    cc.setmaskt () ;
    viewer.setSceneData( root );
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
}

```

```
viewer.sync();  
viewer.cleanup_frame();  
viewer.sync();  
return 0;  
} //Main.cpp 结束
```

编译运行。

运行结果如图所示：



上图为整个模型，下左为隐藏飞机，下右为隐藏牛。

图 6.6

要想理清以上程序并不是一件容易的事，在 osg 中的模型结点组织很复杂，下面来看一下刚才这个模型的组织：

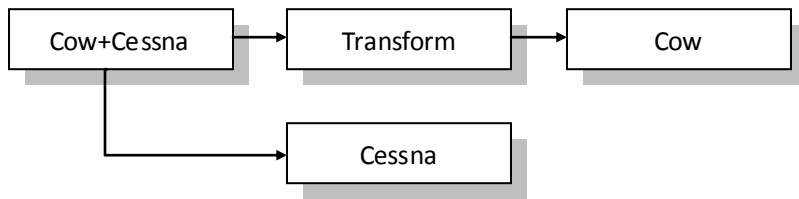


图 6.7

就经验来看，如果要访问已经建好的模型，可以非常不谦虚的说，不是一件容易的事情，甚至很难，这都得基于对 OSG 的理解。

比如下图中的模型，最下方的 Box 到底是哪个父亲的第多少个儿子呢，如果一个组已经分配了空间，而另一个组没有分配空间直接负的指针，又是怎么样一种情况呢？

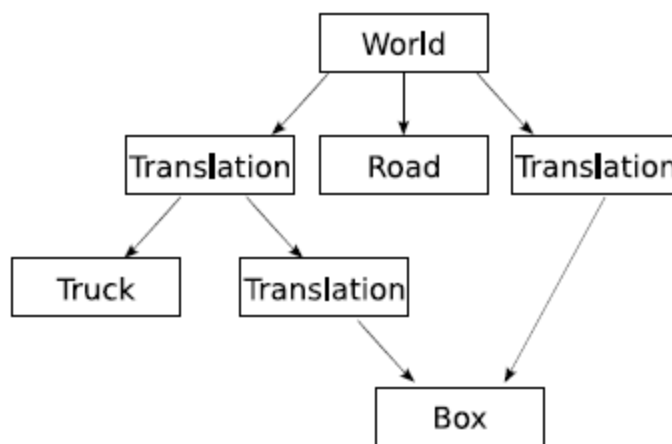


图 6.8

所以尽量不要动已经建好的模型，模型不好，可以用软件来修改，OSG 只是一个渲染工具，而并不是建模工具。当然这只是建议。

因此说 `osg::NodeVisitor` 的更多应用是在 OSG 本身程序的结点中，知根知底的访问，这才是最方便的方法，最好把每个模型类定义一个名称，这样访问起来更方便。

6.3.2 osgFX::Scribe

`osgFX` 名字空间中的类并不很多，现在这里介绍一个可能在以后会经常使用的类，`Scribe`，这个类的作用是修饰结点，当结点为一个图形元素时，可能以高亮显示，当是 3D 物体时，会显示 3D 物体的三角形网，这就是这个类最直接的功能，现在来做一个显示 `glider.osg` 外部网的例子，非常简单，比 OSG 官方显示 `cow.osg` 外部网的例子要简单的多。

示例：Scribe

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：`textScribe` 其它的默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
`OpenThreadsWin32d.lib` `Producerd.lib` `osgd.lib` `osgDBd.lib` `osgFXd.lib` `osgGAd.lib`
`osgParticled.lib` `osgProducerd.lib` `osgSimd.lib` `osgTerraind.lib` `osgTextd.lib`
`osgUtild.lib`

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP ，文件名取为 main.cpp

在 main.cpp 中加入以下内容：

```
Main.cpp
//osgFreeS Source Code By FreeSouth
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osgFX/Scribe>
int main( int argc, char **argv )
{

    osgProducer::Viewer viewer;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::Node* loadedModel = osgDB::readNodeFile("glider.osg");

    //把模型加入到 Scribe 中
    osgFX::Scribe * sc= new osgFX::Scribe ;
    sc->setWireframeColor (osg::Vec4 (1.0, 1, 1,1)) ;
    sc ->addChild (loadedModel) ;

    osg::Group * group = new osg::Group ;
    group ->addChild (sc) ;
    viewer.setSceneData( group );
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();
    return 0;

}                                     //main.cpp 结束
```

运行结果：

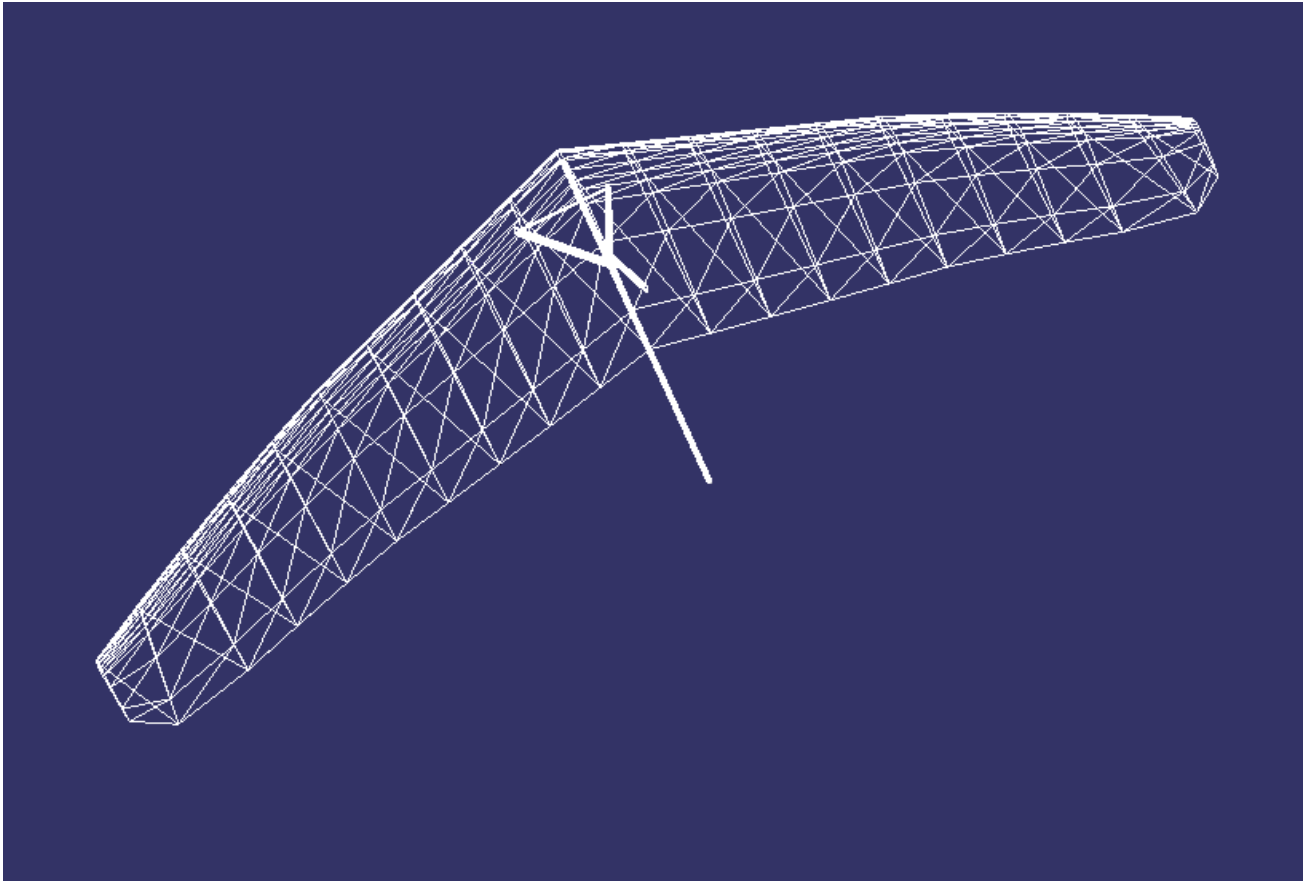


图 6.8

6.3.3 Scribe&Pick&Event

这一节我们来做一个 pick 点击模型的例子，运用本章学习的知识：

点击原理为：首先压入事件，这个事件可以判断左键是否按下，如果左键按下，则执行 PICK，在 PICK 中控制模型是否被 Scribe，在模型方面，首先加入到 Scribe 中并把 Scribe 隐藏，这样显示的就是一个正常的模型，当被点击时显示 Scribe 这样就得到一个高亮的模型。

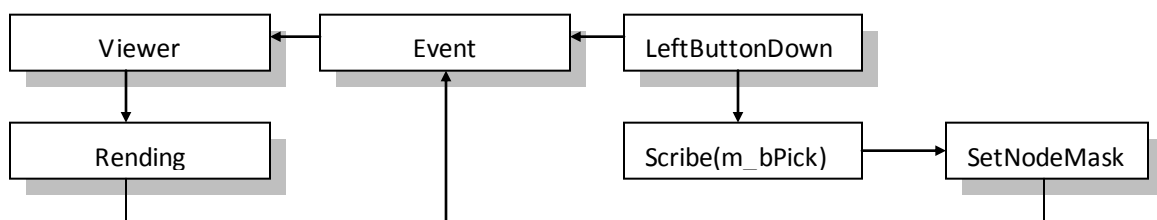


图 6.9

示例：pickNew

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：textScribe 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB:
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtilid.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

第四步：菜单项目→添加类，类名填入 CPickHandler,在基类中添入 osgGA::GUIEventHandler
如果出现提示说什么没有发现这个基类，不用管它，继续添加。

第五步：项目→添加类，类名填入 PickNode,在基类中添入 osg::PositionAttitudeTransform
如果出现提示说什么没有发现这个基类，不用管它，继续添加。

因为我们在添加模型时，需要让模型移动一下，否则两个模型同时挤在中间，虽然模型移动的方法有很多种，这里我们直接继承 osg::PositionAttitudeTransform。

在 PickNode.h 中

```
PickNode.h
//osgFreeS Source Code by FreeSouth
#pragma once
#include <osg/PositionAttitudeTransform>
#include <osgFX/Scribe>
class PickNode :

//从该类继承，用于移动
    public osg::PositionAttitudeTransform
{
public:
    PickNode(void);
    //在该函数中，执行隐藏/显示 Scribe
    void onPicked(void);
protected:
    ~PickNode(void);
private:
    osg::ref_ptr<osgFX::Scribe>m_Scribe;
    bool m_bPicked;//初始化为显示
};
```

在 PickNode.cpp 中

```
PickNode.cpp
//osgFreeS Source Code by FreeSouth
#include <osgDB/ReadFile>
#include "PickNode.h"
PickNode::PickNode(void)
    : m_bPicked(false)
{
    m_Scribe = new osgFX::Scribe ;
```

```

        this->addChild(m_Scribe.get());
        osg::Node* node = osgDB::readNodeFile("cow.osg");
        this->addChild(node);
        m_Scribe->addChild(node);
        m_Scribe->setNodeMask(0);
    }
    PickNode::~PickNode(void)
    {
    }
    void PickNode::onPicked(void)
    {
        m_bPicked = !m_bPicked;
        m_Scribe->setNodeMask(m_bPicked);
    } //PickNode.cpp 结束

```

在 PickHandler.h 中

```

PickHandler.h
#pragma once
#include <osgGA/GUIEventHandler>
#include <osgProducer/Viewer>
class CPickHandler :
    public osgGA::GUIEventHandler
{
public:
    CPickHandler(osgProducer::Viewer* viewer);
    virtual bool handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter &aa);
protected:
    ~CPickHandler(void);
    void Pick(float x, float y);
private:
    osgProducer::Viewer* m_Viewer;
}; //PickHandler.h 中

```

在 PickHandler.cpp 中

```

PickHandler.cpp
#include "PickNode.h"
#include "Pickhandler.h"
CPickHandler::CPickHandler(osgProducer::Viewer* viewer)
    : m_Viewer(viewer)
{
}
CPickHandler::~CPickHandler(void)
{
}

```

```

    }
    bool CPickHandler::handle(const osgGA::GUIEventAdapter& ea,
osgGA::GUIActionAdapter& aa)
    {
        //得到事件类型
        switch(ea.getEventType())
        {
            //如果为鼠标左键按下
            case(osgGA::GUIEventAdapter::PUSH):
                if (ea.getButton() & osgGA::GUIEventAdapter::LEFT_MOUSE_BUTTON)
                {
                    //执行 Pick
                    Pick(ea.getX(), ea.getY());
                }
                return true;
            }
            return false;
        }
    }
    void CPickHandler::Pick(float x, float y)
    {
        //存放结点序列
        osgUtil::IntersectVisitor::HitList hlist;
        if (m_Viewer->computeIntersections(x, y, hlist))
        {
            osgUtil::Hit&hit = hlist.front();
            //存放相交结点序列
            osg::NodePath&np = hit._nodePath;
            //从相交序列的最外侧遍历起，如果从最里面遍历，也是可以的
            for (int i=np.size()-1; i>=0; --i)
            {
                PickNode* pPickNode= dynamic_cast<PickNode*>(np[i]);
                if (pPickNode!= NULL)
                {
                    pPickNode->onPicked();
                }
            }
        }
    }
} //PickHandler.cpp 结束

```

Main.cpp中

```

Main.cpp
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>

```

```

#include "PickHandler.h"
#include "PickNode.h"
int main(int, char**)
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer();
    osg::Group* scene = new osg::Group();

    //此结点为对比结点
    osg::Node* cessna = osgDB::readNodeFile("cessna.osg");
    scene->addChild(cessna);

    //创建 PICKNODE
    PickNode* pPickNode = new PickNode();
    //移动
    pPickNode->setPosition(osg::Vec3(40, 40, 0));
    scene->addChild(pPickNode);
    viewer.setSceneData(scene);
    viewer.getEventHandlerList().push_front(new CPickHandler(&viewer));
    viewer.realize();
    while (!viewer.done())
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    return 0;
}

```

编译运行：

注意：如果需要处理模型，使其可点，那么必须使其模型经过 PICKNODE 的处理，且如果模型本身是一大块，使用建模建好的模型，或是模型的结构已经混乱不清，则处理起来极其困难，可以这么说：如果建模者有一点失误，将导致程序员百般辛苦。有些时候模型出现的问题让人哭笑不得。比如有些透明的面，看见什么也没有，就是走不过去。

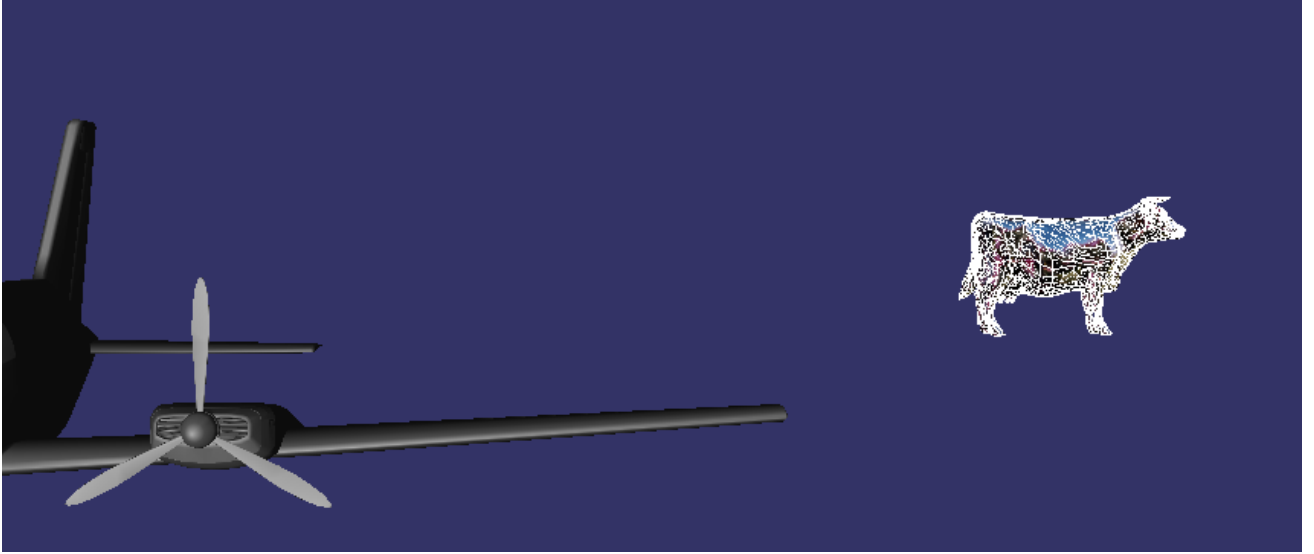


图 6.9

到本章结束，应该基本掌握事件的使用方法，CALLBACK 的使用方法，NODEVISITOR 的使用方法，需要更多的实践才能熟练掌握。

第七章 ViewPort&Camera

这一章我们将进行视口和相机的学习，从某种意义上讲二者有共同之处，而事实上，对于相机的操作对于视口往往都是可以的，对于视口的操作有一部分对于相机则不能，比如相机中很难加入复杂的事件响应，本章所讨论的操作对于视口和相机往往都是通用的。

本书中介绍的视口都是由 `osgProducer::Viewer` 控制的。

本章基于对视口与相机的理解将完成以下功能：制做各个角度的 Camera 视图，制做透明窗口，裁切窗口。

7.1 裁切技术

描述原理不是本书的宗旨，因此这个部分只是简略的介绍一下几种裁切方式以及在后面要使用到的部分数学知识。

7.1.1 平面的表示

在 `osg::Plane` 中有很多方法表示平面，因为后面要使用平面进行裁切，所以在这里讲一下关于平面的数学表示方法。

在 2D 世界中，一条直线可以把世界分为两个部分，在 3D 世界中一个平面可以起到同样的效果。下面我们来看一下 3D 中的平面：

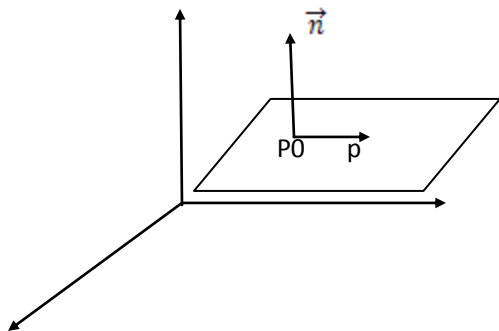


图 7.1

很显然，图中所示平面为向量 $P_0(x_0, y_0, z_0)$ 和法向量 \vec{n} 唯一确定。

对于空间内任意一点 $p(x, y, z)$ ，设置平面 π 为过固定点 $P_0(x_0, y_0, z_0)$ 且法向量为 $\vec{n}(A, B, C)$ 的平面，则 P 在 π 上的充要条件是 $p_0p \cdot \vec{n}$ (或其点积为 0)。

故由此得： $A(x-x_0)+B(y-y_0)+C(z-z_0)=0$ ；

化简可得： $Ax + By + Cz + D = 0$ ；其中 $D = -Ax_0 - By_0 - Cz_0$ ；

故一个平面可以由四个参数 (A, B, C, D) 来表示。

在 2D 世界中二点确定一条直线，在 3D 中自然是三点确定一个平面。

故一个平面可以由三个参来表示。

上面两种表示方法是非常常用的表示方法。

下面我们再看另一种表示方法：如图所示：

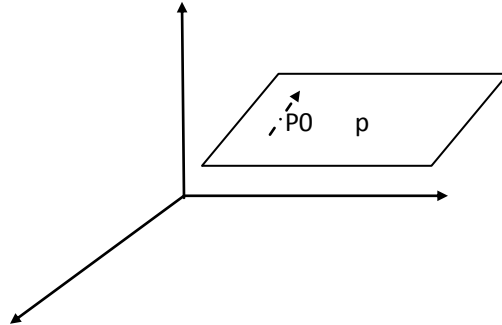


图 7.2

在上图中所示，一个平面可以由一个方向向量 $p0$ 与此外的一点 p 组成。

故一个平可以由方向向量与向量外点 p 组成。

同理也可以由平面法向量与固定点 p 组成。

现在来看一下 `osg::Plane` 类的各种操作：

```
//用于确定一个平面
类 osg::Plane
//平面方程的四参数
    Plane (float a, float b, float c, float d)
//使用一个四维向量来表示这四参数
    Plane (const Vec4 &vec)
//平面法向与距圆点的距离
    Plane (const Vec3 &norm, float d)
//三个点确定一个平面
    Plane (const Vec3 &v1, const Vec3 &v2, const Vec3 &v3)
//平面法向与平面上一点
    Plane (const Vec3 &norm, const Vec3 &point)
//运算符重载/拷贝，set 的意义与上面的构造函数相同
Plane & operator= (const Plane &p1)
void set (const Plane &p1)
void set (float a, float b, float c, float d)
void set (const Vec4 &vec)
void set (const Vec3 &norm, float d)
void set (const Vec3 &v1, const Vec3 &v2, const Vec3 &v3)
void set (const Vec3 &norm, const Vec3 &point)
//反转平面，相当于把法向置负
void flip ()
//规格化，各除以 SQRT
void makeUnitLength ()
```

```

//计算与 BoundingBox 的最高点与最低点, 属于计算 BoundingBox 的加速算法
void calculateUpperLowerBBCorners ()
bool valid () const //是否可用, 运算符重载以及返回指针
bool operator== (const Plane &plane) const
bool operator!= (const Plane &plane) const
bool operator< (const Plane &plane) const
float * ptr ()
const float * ptr () const
//做为一个四维向量返回
Vec4 & asVec4 ()
const Vec4 & asVec4 () const
//得到法向量
osg::Vec3 getNormal () const
//计算与此点的距离是多少, 点到面的距离
float distance (const osg::Vec3 &v) const
//分别计算与向量, BoundingSphere, BoundingBox 是否有交点
int intersect (const std::vector< Vec3 > &vertices) const
int intersect (const BoundingSphere &bs) const
int intersect (const BoundingBox &bb) const
//按矩阵/逆矩阵移动
void transform (const osg::Matrix &matrix)
void transformProvidingInverse (const osg::Matrix &matrix)

```

7.1.2 裁切分类

裁切在 OSG 的使用者来看应该包含两层意思: 第一是对场景没有贡献的部分, 比如物体的背面, 以及遮挡, 以及视锥以外的部分, 第二是对用户没有使用价值的部分, 比如要做一个放大镜, 需要把 Camera 裁切成固定的形状, 比如多面裁切类圆形状。

首先我们来看第一类, 对场景没有贡献的裁切: 下面我们来看一下这种裁切的三种情况:

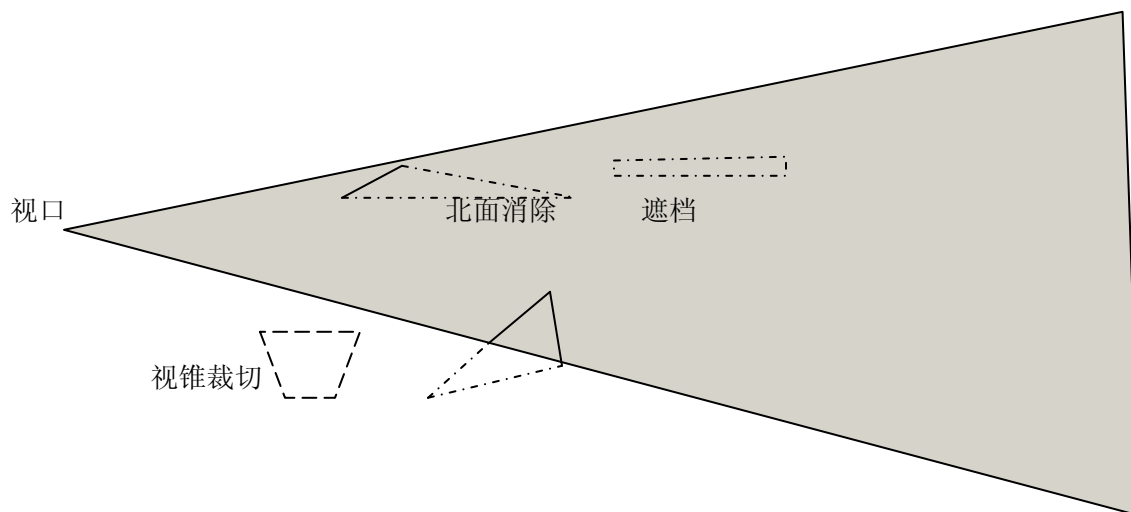


图 7.3

从理论上讲裁切的操作可以发生在渲染以前或是渲染以后, 但是理想的裁切是发生在渲染以前, 也就

是通过严格的计算，发送给 OSG 只是可见的集合。

背面裁切：

在场景中，如果一个物体的投影的空间中是顺时针方向顶点集合，那么这个物体就是背向的，因此背向与正面裁切大同小异。通过法线的计算可以轻易的计算出是否是背面图形，如果它的法向的 z 分量为负就说明它是面向屏目（ z 轴）内侧的，那么就可以被切掉。

还有一种计算是否背面的方法是从该多边形上发出一点到视点，如果此向量与多边形的夹角为竟然为钝角，那么就说明这是背向的。背面的多边形可以不进行光栅化。

视锥体的裁切：

可以使用多边形与平面相交的算法来完成。判断在视口线以下/以上的物体给予保留。

遮档裁切：

通过计算与视口的连线是否与别的物体有交点来完成，或者是当前面出现物体时就已经关闭其最高点与最低点与视口连线后的所有通道。

关于裁切技术有很多的理论知识，在图形学的书籍上大量有述。

7.1.3 OSG 中的裁切类

Osg 中关于裁切有多种类可以实现，也有多种方法可以参考，我们首先来看一下最基本的方法：

可以将视口直接进行收缩，因此视口的大小变小，则需要汇制的多边形也就变小，可以通过这个类来完成：Osg::Viewport。

下面我们来看一下这个类：

类 ViewPort:功能：用来控制视口

//定义视口的大小，x,y 为起始点坐标，屏幕的左下角为原点，是屏幕上的位置，并非窗口相对位置

Viewport (int x, int y, int width, int height)

//设置/得到视口，参数意义同上

void setViewport (int x, int y, int width, int height)

void getViewport (int &x, int &y, int &width, int &height) const

int & x ()

int x () const

int & y ()

int y () const

int & width ()

int width () const

int & height ()

int height () const

//看是否可用

bool valid () const

//得到高宽比

```
float aspectRatio () const
//计算视口矩阵，算法为 V_Local*MVPW
const osg::Matrix computeWindowMatrix () const
```

现在我们来做一个例子使用 ViewPort 来裁切窗口：

示例：使用 ViewPort

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：ViewPort 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

```
Main.cpp
//ByFreeSouth for osgFreeS Code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/Viewport>
#include <osg/StateSet>
int main(int, char**)
{
    //定义视口为从左下角到中间,这是屏幕上的位置，并不是相对位置
    osg::Viewport * vv = new osg::Viewport (10, 10, 500, 500) ;
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    viewer->setSceneData (osgDB::readNodeFile ("ceep.ive"));

    //对结点组group实现裁切,注：对于不裁切的结点在视窗口中将不受影响
    group ->getOrCreateStateSet () ->setAttributeAndModes (vv,
osg::StateAttribute ::ON ) ;
    viewer->realize();
    while (!viewer->done())
    {
        viewer->sync();
        viewer->update();
        viewer->frame();
    }
    viewer->sync();
    return 0;//Main.cpp
```

运行效果如下：



图 7.4

现在来看一下面裁切，我们可以使用一个面来对某个物体进行裁切，比如我们要做一个放大镜，设置一个 Camera 之后，发现只能是方形的，因此我们就可以对其视口进行裁切，下面我们来看一下关于面裁切的类：

功能：定义裁切平面来进行裁切

类 ClipPlane

//构造函数，第一个为平面 ID，

ClipPlane (unsigned int no, const Vec4d &plane)

ClipPlane (unsigned int no, const Plane &plane)

//ABCD 四个数，在第一节中有介绍

ClipPlane (unsigned int no, double a, double b, double c, double d)

//得到在属性序列中的 ID

virtual unsigned int getMember () const

//得到是否可用

virtual bool getModeUsage (ModeUsage &usage) const

//设置一个平面来进行裁切

void setClipPlane (const Plane &plane)

void setClipPlane (double a, double b, double c, double d)

void setClipPlane (const Vec4d &plane)

//得到一个裁切平面

const Vec4d & getClipPlane () const

//设置平面数量

void setClipPlaneNum (unsigned int num)

unsigned int getClipPlaneNum () const

如果需要多面进行裁切，则可以定义一个 ClipNode 进行裁切，ClipNode 是一个多面的裁切序列：

```
类：ClipNode
//创建切片盒子，根据 BoundingBox，
void createClipBox (const BoundingBox &bb, unsigned int clipPlaneNumberBase=0)
//添加一个切面
bool addClipPlane (ClipPlane *clipplane)
//删除一个切面
bool removeClipPlane (ClipPlane *clipplane)
bool removeClipPlane (unsigned int pos)
//得到切面总数
unsigned int getNumClipPlanes () const
//得到一个切面/序列
ClipPlane * getClipPlane (unsigned int pos)
const ClipPlane * getClipPlane (unsigned int pos) const
void getClipPlaneList (const ClipPlaneList &cpl)
ClipPlaneList & getClipPlaneList ()
const ClipPlaneList & getClipPlaneList () const
//设置 BoundingSphere
virtual BoundingSphere computeBound () const
```

这个类是个非常好用的一个类，下面我们来看一下具体的效果，如果我们使用平面 VEC4 (1,1,1,1) 对 CEEP.IVE 进行裁切，看一下会发生什么效果：

示例：ClipPlane

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：ClipPlane 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

```
Main.cpp
//by FreeSOUTH for osgFreeS Source Code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/ClipPlane>
int main(int, char**)
{

    //定义切面
    osg::ClipPlane *cc = new osg::ClipPlane (0, 1, 1, 1,1) ;
```

```

osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
viewer->setUpViewer();
viewer->setSceneData (osgDB::readNodeFile ("ceep.ive"));

//打开属性
group ->getOrCreateStateSet () ->setAttributeAndModes (cc,
osg::StateAttribute ::ON ) ;
viewer->realize();
while (!viewer->done())
{
    viewer->sync();
    viewer->update();
    viewer->frame();
}
viewer->sync();
return 0;
}

```

编译运行。

如果使用切片序列 (osg::ClipNode) 进行裁剪，需要注意几个问题，必须在每个切片中加入的模型是相同的，才能保证切的是同一片模型，且这个函数的原理是对模型进行剪切，而并非对窗口进行限制，这二者不能混淆。

使用 ViewPort 进行切割时，如果加入的是 Viewer ->SceneData 中的内容，那么就证明切分的是窗口，它将直接限制在屏幕像素点上的行为，这体现在如果使用 ClipNode 进行切割过的模型将仍旧可能显示在 ViewPort 允许显示的点上，而使用 ViewPort 切过的模型将直接限制在窗口上的行为，而模型是完好的。

下面来看一下运行效果：

如下图：

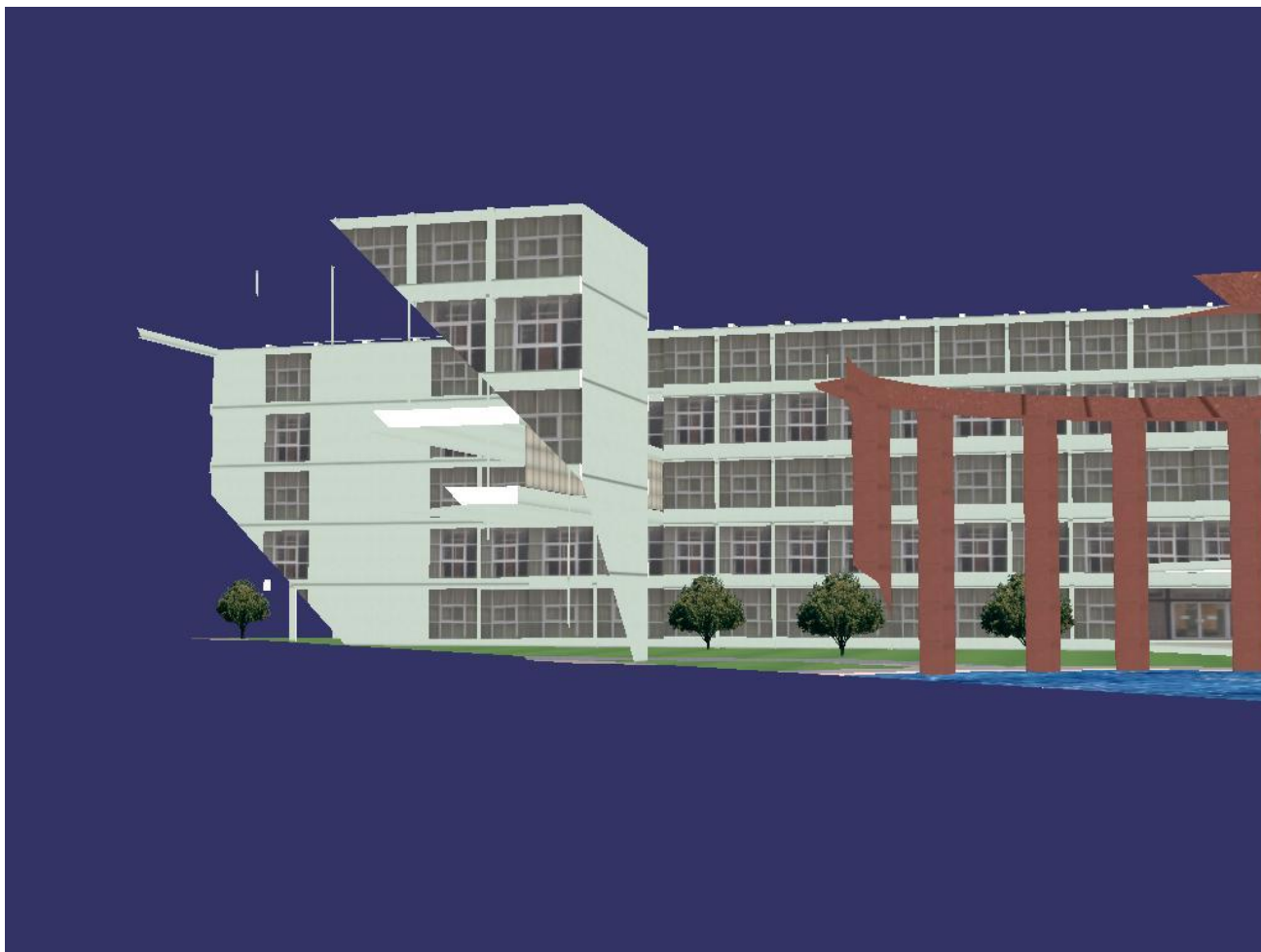


图 7.5

可以看到有明显的切割的痕迹。

现在来看一下另外一个可以剪切视口的类（注意不是操作模型）：

```

类 Scissor : 功能，剪切窗口
//限制高宽，与 ViewPort 相同
Scissor (int x, int y, int width, int height)
//得到是否可用
virtual bool  getModeUsage (ModeUsage &usage) const
//设置得到/高宽
void  setScissor (int x, int y, int width, int height)
void  getScissor (int &x, int &y, int &width, int &height) const
//得到 x 分量，
int &  x ()
int  x () const
//得到 y 分量，
int &  y ()
int  y () const
int &  width ()
int  width () const
int &  height ()

```

```
int height () const
```

下面我们编一段代码测试一下这个类的功能：

示例：Scissor

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：Scissor 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtilde.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP ，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

```
Main.cpp
//By FreeSouth For osgFreeS Source code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/Scissor>
int main(int, char**)
{
    //初始化切割视口
    osg::Scissor * cc = new osg::Scissor (200, 100, 800, 600) ;
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer () ;
    viewer->setUpViewer() ;
    viewer->setSceneData (osgDB::readNodeFile ("ceep.ive")) ;

    //打开属性，
    group ->getOrCreateStateSet () ->setAttributeAndModes (cc,
osg::StateAttribute ::ON ) ;

    viewer->realize() ;
    while (!viewer->done())
    {
        viewer->sync() ;
        viewer->update() ;
        viewer->frame() ;
    }
    viewer->sync() ;
    return 0 ;
}
```

编译运行。

来查看一下结果，如图 7.6 就会发现与操作 ViewPort 是相同的结果，同样是对视口的绝对坐标进行操作，这里最后再说明要注意的内容：屏幕左下角为零点，右上角为 (1024,768) 点，当然取决于你的分辨率。

如图：



图 7.6

可以清楚的看到，模型只显示在屏目的中间位置。

7.2 Camera

相机的作用是建立一个不受视口骚扰的分窗口，如果使用灵活，相机完全有着十分神奇的功能，如鹰眼全图，指北针，放大镜，前后左右俯视图，等等。

下面首先来看几个与相机的使用密切相关的部分：

7.2.1 Camera&HUD

HUD 放在 Camera 中显示是个十分重要的应用，在上一章的例子：ShowPosition 中使用的就是把 HUD

放在 Camera 中进行使用，当然 Camera 中可以放置任何内容，放置的模型可以与主窗口的相同，也可以不相同。

我们首先来看一下 Camera 这个类：

类 CameraNode

//设置/得到清屏的颜色，这将使默认的黑色背景改变，这就像 OSG 大窗口默认的蓝色背景一样

void setClearColor (const Vec4 &color)

const Vec4 & getClearColor () const

//如果置 GL_DEPTH_BUFFER_BIT，则背景显示为透明，如果没有模型挡住，直接将显示大窗口中的模型

void setClearMask (GLbitfield mask)

GLbitfield getClearMask () const

//设置混合颜色，比如将红色置 false 那么红色将与后面的颜色进行混合，从而窗口透明，但是效果很不怎么样，因为模型本身是不透光的，与贴图不同：可以这么轻易的处理

void setColorMask (osg::ColorMask *colorMask)

void setColorMask (bool red, bool green, bool blue, bool alpha)

const ColorMask * getColorMask () const

ColorMask * getColorMask ()

//设置视口，该视口为视口的相对位置，如果主视口受过切割，好么将从切割后的位置开始算起

void setViewport (osg::Viewport *viewport)

void setViewport (int x, int y, int width, int height)

const Viewport * getViewport () const

Viewport * getViewport ()

//设置移动次序，这里次序是指，M*N 与 N*M 在矩阵界是不相同的

void setTransformOrder (TransformOrder order)

TransformOrder getTransformOrder () const

//设置透视参数，平头截体，平行透视等等众多方式，主要做用是设置显示模型的大小，朝向，旋转等等，如果设置多视图，比如后视图，旋转模型就在这一步里执行

void setProjectionMatrix (const osg::Matrixf &matrix)

void setProjectionMatrix (const osg::Matrixd &matrix)

void setProjectionMatrixAsOrtho (double left, double right, double bottom, double top, double zNear, double zFar)

void setProjectionMatrixAsOrtho2D (double left, double right, double bottom, double top)

void setProjectionMatrixAsFrustum (double left, double right, double bottom, double top, double zNear, double zFar)

void setProjectionMatrixAsPerspective (double fovy, double aspectRatio, double zNear, double zFar)

//得到这些透视参数

osg::Matrixd & getProjectionMatrix ()

const osg::Matrixd & getProjectionMatrix () const

bool getProjectionMatrixAsOrtho (double &left, double &right, double &bottom, double &top, double &zNear, double &zFar)

bool getProjectionMatrixAsFrustum (double &left, double &right, double &bottom, double &top, double &zNear, double &zFar)

bool getProjectionMatrixAsPerspective (double &fovy, double &aspectRatio, double &zNear, double &zFar)

//在这一步了可以设置旋转等等很多操作，也可以得到前后左右视图等等，但是与上面的概念上是不一样的，这里是设置视口

```
void setViewMatrix (const osg::Matrixf &matrix)
void setViewMatrix (const osg::Matrixd &matrix)
void setViewMatrixAsLookAt (const osg::Vec3 &eye, const osg::Vec3 &center, const osg::Vec3 &up)
osg::Matrixd & getViewMatrix ()
const osg::Matrixd & getViewMatrix () const
void getViewMatrixAsLookAt (osg::Vec3 &eye, osg::Vec3 &center, osg::Vec3 &up, float lookDistance=1.0f)
//得到逆视口矩阵
Matrixd getInverseViewMatrix () const
//这个参数很重要，必须设置成最后渲染 POST_RENDER，否则会被别的渲染层次给挡住
void setRenderOrder (RenderOrder order, int orderNum=0)
RenderOrder getRenderOrder () const
int getRenderOrderNum () const
//计算某物在世界坐标系/局部坐标系中的位置
virtual bool computeLocalToWorldMatrix (Matrix &matrix, NodeVisitor *) const
virtual bool computeWorldToLocalMatrix (Matrix &matrix, NodeVisitor *) const
```

现在我们来建立一个显示 HUD 的子窗口：

示例二十:HUD

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：CameraHUD 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
osgUtd.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

```
Main.cpp
//by FreeSouth For osgFreeS Source Code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/Geode>
#include <osg/Depth>
#include <osg/CameraNode>
#include <osgText/Text>
osg::Node* createHUD ()
{
    osg::Geode* geode = new osg::Geode();
    //设置字体，字体光盘文件夹IVE中可以找到
    std::string caiyun("fonts/STCAIYUN.TTF");
    //设置状态，关闭灯光
```

```

osg::StateSet* stateset = geode->getOrCreateStateSet();
stateset->setMode(GL_LIGHTING, osg::StateAttribute::OFF);
osg::Vec3 position(150.0f, 500.0f, 0.0f);
//设置字体属性
osgText::Text* text = new osgText::Text;
geode->addDrawable( text );
//设置字体
text->setFont(caiyun);
//设置位置
text->setPosition(position);
text->setText(L"生日快到了:(4:12)");
//设置相机
osg::CameraNode* camera = new osg::CameraNode;
//设置透视矩阵
camera->setProjectionMatrix(osg::Matrix::ortho2D(0,1024,0,768));
camera->setReferenceFrame(osg::Transform::ABSOLUTE_RF);
//得到默认设置
camera->setViewMatrix(osg::Matrix::identity());
//设置背景为透明，否则的话可以设置ClearColor
camera->setClearMask(GL_DEPTH_BUFFER_BIT);
//设置渲染顺序，必须在最后渲染
camera->setRenderOrder(osg::CameraNode::POST_RENDER);
camera->addChild(geode);
return camera;
};

int main( int argc, char **argv )
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::ref_ptr<osg::Node> scene = osgDB::readNodeFile("ceep.ive");
    osg::ref_ptr<osg::Group> group = new osg::Group;
    if (scene.valid()) group->addChild(scene.get());
    //创建HUD
    group->addChild(createHUD());
    viewer.setSceneData(group.get());
    viewer.realize();

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

```

```

viewer.sync();
viewer.cleanup_frame();
viewer.sync();
return 0;
}

```

编译运行。

注意：如果没有中文字体，内容将不会正常的显示。中文字体的名称并不是字库里的名称，需要拷出来查看名称。

运行结果如图所示：

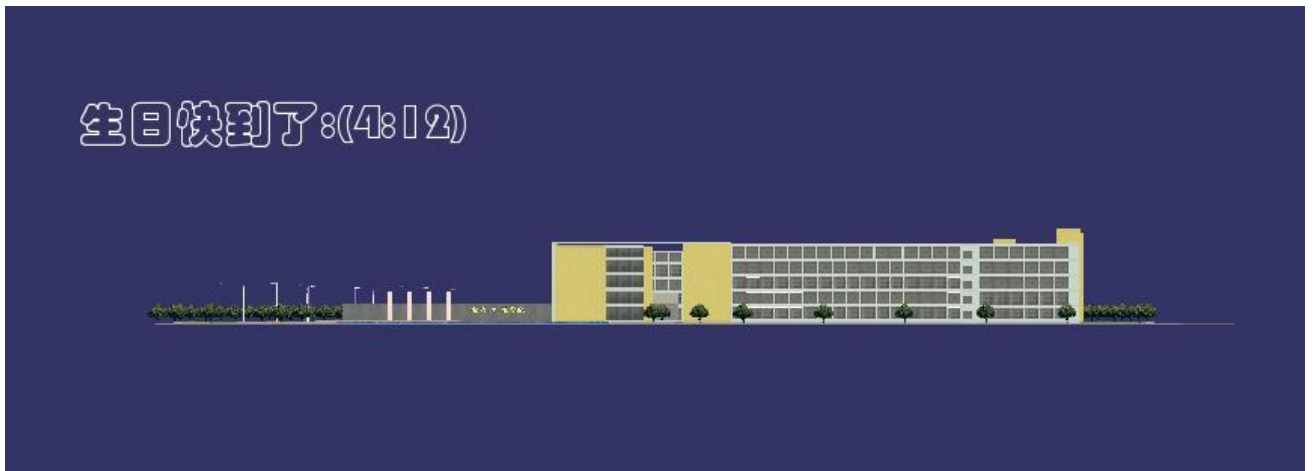


图 7.7

7.2.2 多视图

有了以上的例子，多视图的制做变得非常简单，这里需要强调的是，小窗口中的模型可以不与主模型相一致，最简单的例子说，可以没有模型，当然也可以有别的模型，向上例子中里面放的有模型 HUD。

下面我们来做一个关于视图的例子，只是做了一个窗口，别的视图同理，可以把各个视图拼接，组成一个类似于凹凸镜的例子。等等，现在我们来做一个只有一个视图的例子：做为本章的结尾：

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：MulCamera 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
 OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
 osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrainind.lib osgTextd.lib
 osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

Main.cpp

```
//by FreeSouth for osgFreeS Source Code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osg/CameraNode>
#include <osg/FrontFace>

osg::Node* CreateBackViewer(osg::Node* PNode) ;
int main(int, char**)
{
    osg::ref_ptr<osgProducer::Viewer>viewer = new osgProducer::Viewer ();
    viewer->setUpViewer();
    osg::Group * group = new osg::Group;
    //读取结点，此文件在光盘IVE文件夹中
    osg::Node* node = osgDB::readNodeFile("ceep.ive");
    group ->addChild (node) ;
    group ->addChild (CreateBackViewer(node)) ;
    viewer->setSceneData (group);
    viewer->realize();

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

    viewer->sync();
    return 0;
}

//创建后视图
osg::Node* CreateBackViewer(osg::Node* PNode)
{
    osg::CameraNode* camera = new osg::CameraNode;
    //设置视口相对主窗口位置
    camera->setViewport(10,10,300,200);
    camera->setCullingActive(true);
    camera->setReferenceFrame(osg::Transform::RELATIVE_RF);
    camera->setTransformOrder(osg::CameraNode::POST_MULTIPLY);
    //反转过来
    camera->setProjectionMatrix(osg::Matrixd::scale(-1.0f,1.0f,1.0f));
    //使背景透明
    camera->setClearMask(GL_DEPTH_BUFFER_BIT);
    //设置最后渲染，否则会被挡住
```



```
camera->setRenderOrder(osg::CameraNode::POST_RENDER);  
camera->addChild(PNode);  
//顺时针为正面  
camera->getOrCreateStateSet()->setAttribute(new  
osg::FrontFace(osg::FrontFace::CLOCKWISE));  
return camera;  
} //Main.cpp结束
```

编译运行。

运行结果如图 7.8:

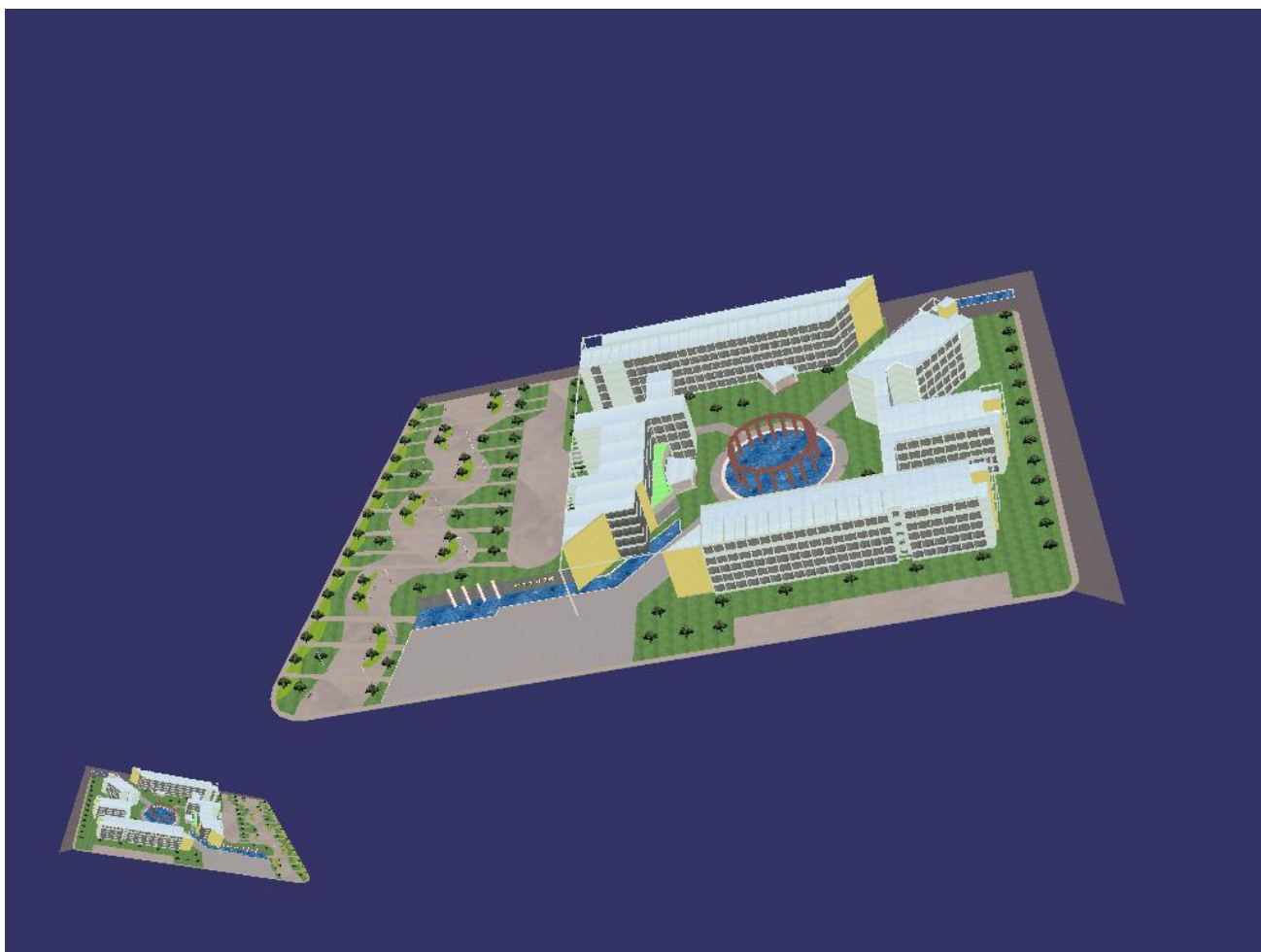


图 7.8

到现在为止本章的内容就结束了，通过本章的学习，应该掌握视口与模型的剪切，Camera 的使用等等知识。这些东西都需要很多次的尝试才能熟练的掌握，需要一定的想象力和大胆的创新能力。

第八章 光照&Billboard&LOD

这一章主要来介绍一下 OSG 中关于光源的设置以及光照着色的原理，和 Billboard 技术。关于 LOD 本身就可以独自成书，在本章只是简单的介绍一下在 OSG 中 LOD 制做 LOD 的例子。

本章将完成一个设置光源的例子，一个设置 Billboard 的例子，以及用模型代替和用动态图片代替的 LOD 的例子。

在本章的学习中，会逐步的掌握以下类的使用 `osg::light`, `osg::Billboard`, `osg::LOD` 和 `osgSim::impostor`。

在 OSG 中，对光照与 Billboard 的处理非常足够，但是对于 LOD 确显得不是很让人理想，然而从某种意义上讲 LOD 是根据人不同的需要而做的程度不同，OSG 只能给人们提供方便的场景管理，关于生面网格，计算复杂的地形变换这需要人们自己来做，一个通用的 LOD 算法可能因为需求不同而不存在，但是已经有很多成熟的算法与处理过程和很多让人意想不到的方法来达到人们心目中想要的效果。

8.1 Light

在很久以前人们就开始注意使用光照使模型接近真实，可信。在现实生活中人们之所以看到物体就是因为有了光源，虽然说在 3D 世界中没有光源也可以模拟 3D 场景，但是缺乏真实感。

下面我们来分开介绍一下光照的原理与 OSG 中光照的应用。

8.1.1 材质与着色

物体色彩的明暗变化主要取决于两个特性，一是物体本身的材质参数，二是取决于光源的不同特性。

材质

在现实生活中，光源照在不同材质的物体上会显示出不同的效果。因此在 3D 场景中需要对各种不同的材质进行模拟。在实际中光照也分为很多不同的类型，十分精确的模拟光照虽然是难度十分大的工作，但是现如今的仿真程度已经足够欺骗人类的双眼或是满足它刻画的需求。

在实时系统中，材质会有许多的参数，随着环境的不同而不同，比如在不同的光照情况下会有不同的效果，这就需要来看该材质受该光照影响的程度。实体的效果是由光照以及物体受该光照影响的参数构成的。下面来列举一下材质受光照影响的一些参数：

环境材质颜色、漫反射材质颜色、镜面反射材质颜色、光亮度参数、放射性材质颜色。

光源特性

光源的主要特性包括以下几个方面：

1、光的色彩，在 OSG 中的 `light` 类中可以设置光照的颜色。光的色彩一般使用传统的 RGB 的格式来描述，三种色光的比例便构成了不同的色相构成。因此光照可以使用一个三维向量(R,G,B)来表示。

2、光的强度，光的强弱由 RGB 的组成成分来决定，但是三色光对光强的影响程度却各不相同，总光强度为： $I = 0.30R + 0.59G + 0.11B$

3、光源类型，光源从目前的角度来看总共分为三种不同的类型，**点光源**：点光源位于空间的某个位置上，并从该位置均匀的发射光线。**平行光源**：平行光源来自于空间的无穷远位置，所以发射的光可以看成是平行的，如太阳的光对于地球来看就可以当成平行光源，室内的电灯泡可以当成点光源。**漫射光源**：是从一个面上各个不同的角度发射的光源。在有的书上有时候也把点光源与平行光源合称为直射光源。

下面以图示来显示三种不同的光源的不同特性：

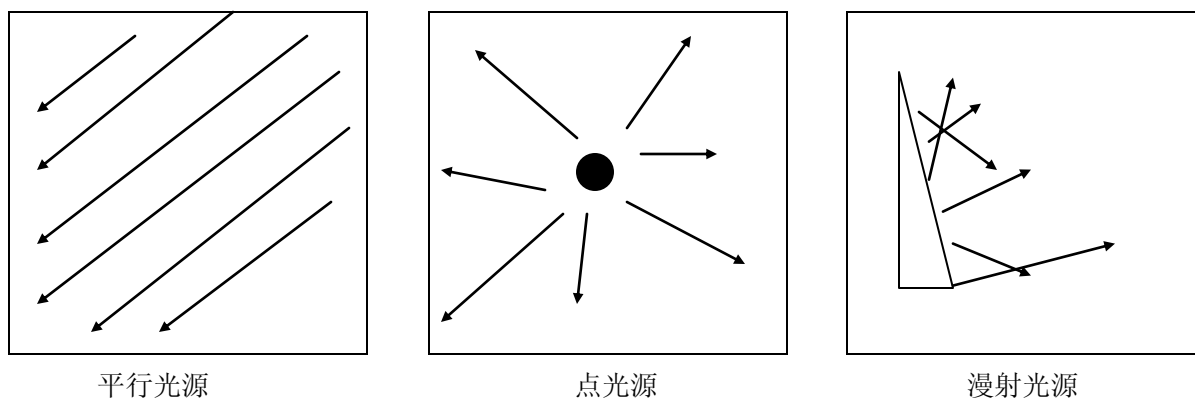


图 8.1

着色方法

一般而言，对一个物体着色，现实中经常使用的有三种方法：扁平着色，Gouraud 着色与 Phong 着色，下面对这三种方法简单的做一下描述，扁平着色是对物体的表面多边形负一个颜色值，这样处理出来的效果，可以想到有点像马赛克，并不理想，Gouraud 是对多边形的顶点进行着色，而利用插值算法计算出多边形内部各顶点的颜色值。Phong 是对多边形的法线进行插值，从而计算出每个像素点的颜色，因此对物体指出法线是必要的，否则无法进行光照的处理。

物体的表面特征：物体的表面特征由以下几个参数构成：

1、反射系数：这是由物体表面的形状与材质构成的，镜子将反射平行光，木块反射的可能就是漫射光，因此反射系数主要有两部分构成：漫反射系数与镜面反射系数。这在 OSG 中对应于两种参数：Diffuse 和 Specular. 镜面反射出的光线一般认为仍是光源的色彩。

2、透射系数：透射系数是指物体透光的程序，比如毛玻璃与木块的透光能力是不十分相同的。此系数一般记为一个 $[0,1]$ 记为从完全透明到完全不透明。

3、表面方向：使用多边形的法线来表示。所以定义物体时，法线是必须的，在上一章讲平面时可以知道，法线是固定平面的一种广用方法。

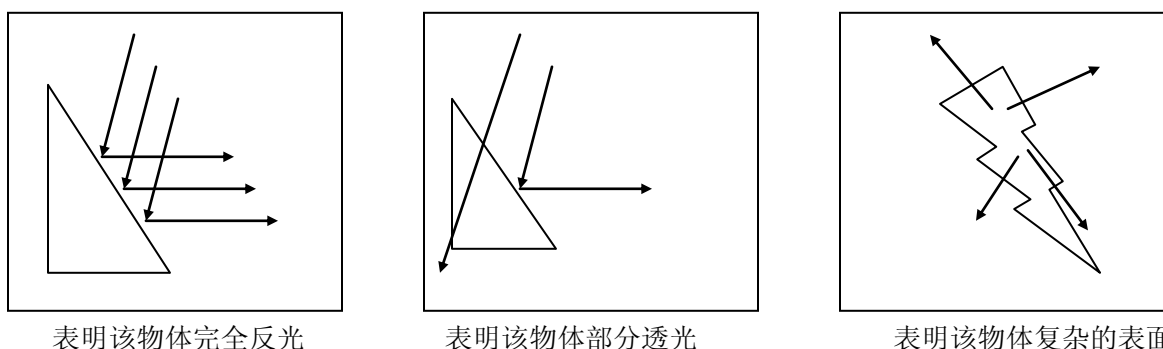


图 8.2

在 OSG 中对光源的描述有众多的参数，对物体也有是否接受光源的相关描述。对于一个已经贴好图的物体，可以不授受光源，也可以接受光源。这反映在物体的属性中，在场景中可以设置局部光源，也可以设置全局光源。对于光的控制往往在 OSG 中大场景中显得不是十分的显眼。

OSG 中的光源有以下几个参数：全局光：Ambient，漫射光（混合光）：Diffuse, 镜面光：Specular. 以及光源的位置：Position，以及光源的方向：Direction, 以及光线的颜色：Color, 和光源的衰减程度：Attenuation 和

光源的剪切: Cutoff.

下面我们来看一下这几个量是如何回事:

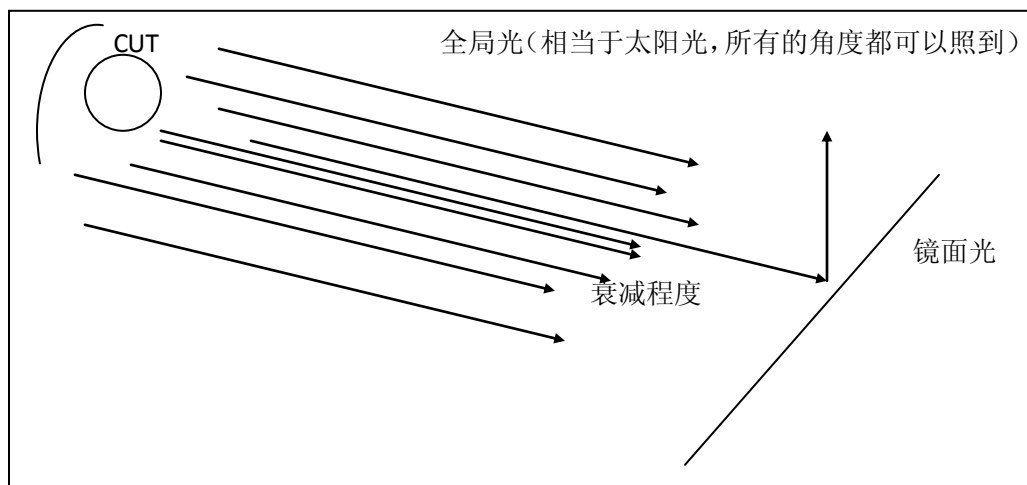


图 8.3

上图简单的来描述一下 OSG::LIGHT 的各种常用的参数。

下面具体来介绍一下光照的三个分量: 漫反射分量, 镜面反射分量, 与环境(全局)分量。

8.1.2 光照分量

1、漫反射分量:

曾记得高中曾经学过物理, 上面说漫反射是满足光线反射定律的, 从而画出一个漫反射的图形, 在每个凹凸不平的平面都画出一条入射光线与反射光线与法线夹角相同的图示, 也许通过本节, 将有不同的理解。

光线是被完全吸收还是沿任意方向反射出去, 这取决于光子的颜色与材质的颜色。

基于漫反射有一个著名的 Lambert 定律, 定律说: 对于理想漫反射的物体表面(完全不光滑且没有光泽)来说: 反射光由物体表面法线 N 与光向量 I 的夹角 A 的余弦值来决定, 光向量是指从物体表面的点 P 到光源的向量, 如图所示:

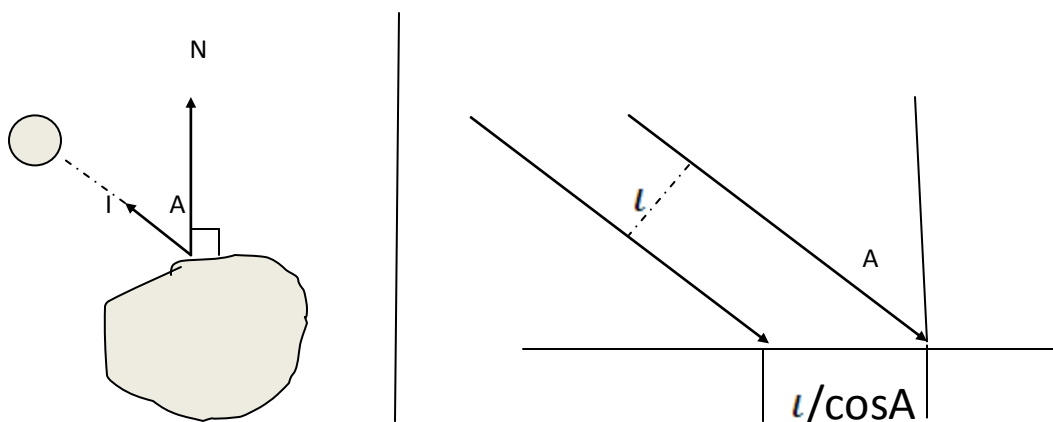


图 8.4

因此漫反射的分量可以表示为 $d = N \cdot I = \cos A$, 当 $A > 90^\circ$ 时, 漫反射光线为零, 也就是说明物体并没有朝

向光源。8.4 右图表示，光线间的列宽固定为 l ，那么到达物体的距离变为 $l/\cos A$ ，由于光源强度与距离成反比，所以这也就说明漫反射光照必定与 $\cos A$ 成比例，应该不是很难理解，当 $A=90$ 时不会接受到光照。

如果光照的漫反射颜色为 S ，物体的漫反射颜色为 M ，那么漫反射分量的颜色为： $(N \cdot I) MS$ ，其中 MS 分量相乘。

2、镜面反射分量：

镜面反射的目的是产生高光，高光的目的是表明物体表面的弯曲程度。从而也可以判断出光源的方向与位置。常见的一个玻璃球上有一个亮斑，这就属于镜面反射分量的做用。

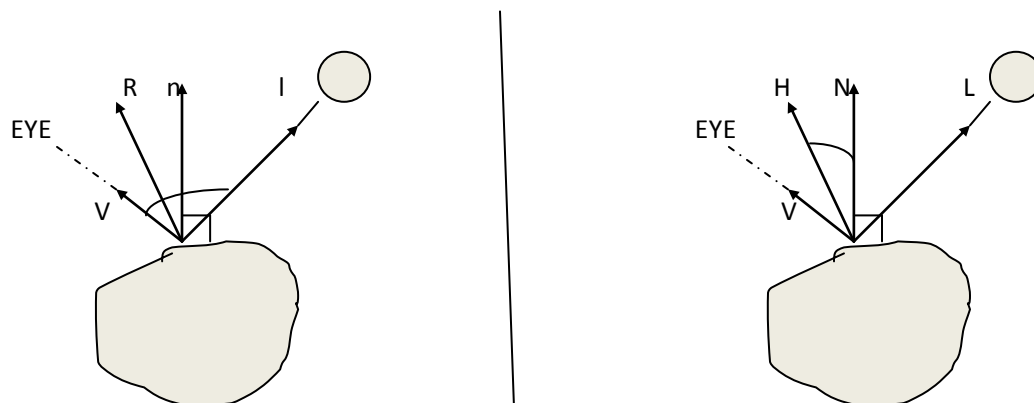


图 8.5

如图中 V 是表示表面点到视点 EYE 之间的向量， R 表示光线向量 I 在法线周围的反射，那么镜面光分量 S 可以表示为： $S=(R \cdot V)^M = (\cos A)^M$ ， A 表示左图中所示的角度。右图所示的为镜面光照方程的几何意义：

$H=(I+V)/\|I+V\|$ ，镜面衰减因子由 $(N \cdot H)$ 决定，但是同样依赖于向量 V ， OSG 中使用的是这种镜面光照方程。

光向量 I 在法线 N 附近反射，产生的反射向量 $R=2(N \cdot I)N-I$ 其中的向量都是规格化的。

同样，如果材质参数为 U ，光源镜面反射颜色为 S 那么镜面反射分量的颜色为： $(N \cdot H)^M US$ ， US 表示分量相乘。

3、环境分量

在现实生活中，光源可能从一个物体之上弹到另外一个物体之上，相比之下简单的光照模型被反射与漫射的光不会再用其它的光源方向中，因此光照模型往往需要包含一些环境因素，如果环境色参数为 M ，光源环境色参数为 S ，则最终分量颜色为： MS ，表示分量相乘。这就意味着在没有任何光照的情况下，物体也会接收一部分的光照。

4、光照方程

光照最终强度为各个参数分量之和

8.1.3 osg::light

在了解了光源所拥有的各个参数之后，我们来看一下 OSG 中的光源类： $osg::light$ 是从 $osg::StateAttribute$ 派生而来，从而决定是非常易用的，给我的感觉是从 $osg::StateAttribute$ 派生出来的类都非常的易用。

类 $osg::light$ 功能：设置光源

//得到标识，这通常使用在使用多光源与 $CLIP$ 中使用

virtual unsigned int getMember () const

//得到是否可用

```

virtual bool  getModeUsage (ModeUsage &usage) const
//设置/得到哪个光源在处于工作
void  setLightNum (int num)
int  getLightNum () const
//设置/得到全局光照颜色
void  setAmbient (const Vec4 &ambient)
const Vec4 &  getAmbient () const
//设置/得到漫射（混合）光照颜色
void  setDiffuse (const Vec4 &diffuse)
const Vec4 &  getDiffuse () const
//设置/得到镜面光照颜色
void  setSpecular (const Vec4 &specular)
const Vec4 &  getSpecular () const
//设置/得到光源的位置
void  setPosition (const Vec4 &position)
const Vec4 &  getPosition () const
//设置得到光源的朝向，在 CUT 时可能会使用到
void  setDirection (const Vec3 &direction)
const Vec3 &  getDirection () const
//以下是设置得到各种衰减度越大表示衰减的越可怕
void  setConstantAttenuation (float constant_attenuation)
float  getConstantAttenuation () const
void  setLinearAttenuation (float linear_attenuation)
float  getLinearAttenuation () const
void  setQuadraticAttenuation (float quadratic_attenuation)
float  getQuadraticAttenuation () const
//设置/得到 Exponent 与 Cutoff
void  setSpotExponent (float spot_exponent)
float  getSpotExponent () const
void  setSpotCutoff (float spot_cutoff)
float  getSpotCutoff () const

```

与类 `osg::light` 密切相关的还有一个类：

`osg::LightSource`，这个类是从 `osg::Group` 直接派生而来，用于加入到场景的数据中。

下面我们来举一个光源的例子来看一下具体的应用：

该例子的功能是在一个飞机的上方加一个光源。

示例：light

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testLight 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
 OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
 osgParticled.lib osgProducerd.lib osgSimd.lib osgTerraind.lib osgTextd.lib
 osgUtild.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为

main.cpp

在 main.cpp 中加入如下代码:

```
Main.cpp
//by FreeSouth for osgFreeS Source Code

#include <osgProducer/Viewer>
#include <osg/Group>
#include <osg/Node>
#include <osg/Light>
#include <osg/LightSource>
#include <osg/StateAttribute>
#include <osg/Geometry>
#include <osg/Point>
#include <osgDB/ReadFile>
#include <osg/Geode>
#include <osg/ShapeDrawable>

//创建一个球，代表光源的位置，参数为球的位置
osg::Geode * createSpot(osg::Vec3 position)
{
    osg::Geode* geode = new osg::Geode ;
    //设置球的半径为0.1
    const float radius = 0.1 ;
    //设置球的精细程度为2.0
    osg::ref_ptr<osg::TessellationHints> hints = new osg::TessellationHints;
    hints->setDetailRatio(2.0f);
    osg::ref_ptr<osg::ShapeDrawable> shape;
    shape = new osg::ShapeDrawable(new osg::Sphere(position, radius),
    hints.get());
    //设置球的颜色为白色
    shape->setColor(osg::Vec4(1.0f, 1.0f, 1.0f, 1.0f));
    geode->addDrawable(shape.get());
    return geode ;
};

//创建光源，参数为飞机模型的包围球，在此例子后面我们会介绍包围球与盒。
osg::Node* createLights(osg::BoundingSphere& bb)
{
    osg::Group* lightGroup = new osg::Group;
    float modelSize = bb.radius()*10.0;
    //创建一个光源
    osg::Light* myLight = new osg::Light;
    //ID设置为0
```

```

myLight->setLightNum(0);
//设置光照的位置
myLight->setPosition(osg::Vec4(bb.center().x(),bb.center().y(),bb.center().z()+bb.radius()),1.0f));
//设置光照的二个参数，其中镜面光没有设置
myLight->setAmbient(osg::Vec4(1.0f,1.0f,1.0f,1.0f));
myLight->setDiffuse(osg::Vec4(1.0f,1.0f,1.0f,1.0f));

//衰减
myLight->setConstantAttenuation(1.0f);
myLight->setLinearAttenuation(1.0f/modelSize);
myLight->setQuadraticAttenuation(2.0f/osg::square(modelSize));
//创建lightsource
osg::LightSource* lightS = new osg::LightSource;
lightS->setLight(myLight);
//开启光照属性
lightS->setLocalStateSetModes(osg::StateAttribute::ON);
//加入光源
lightGroup->addChild(lightS);
//加入光源所在的大圆点
lightGroup->addChild(createSpot(osg::Vec3(bb.center().x(),bb.center().y(),bb.center().z()+bb.radius())));
return lightGroup;
}

//创建场景，组织结点
osg::Node* createScene()
{
    osg::Group* root = new osg::Group;
    osg::Node * node = osgDB::readNodeFile("glider.osg");
    osg::BoundingSphere bb;
    bb = node->getBound();
    root->addChild(node);
    root->addChild(createLights(bb));
    return root;
}

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

    osg::Node* rootnode = createScene();
    viewer.setSceneData( rootnode );

```



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

编译运行。

运行效果如下图：



图 8.6

下面我们来看一下这两个重要的类，包围球与包围盒。包围的思想得从碰撞检测说起，说到包围就必须得讲到精确度：

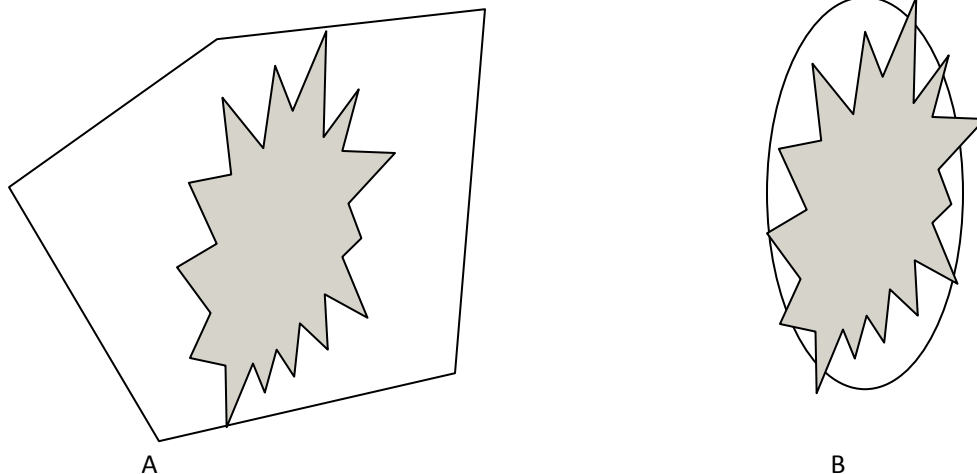


图 8.7

从图中可以得知，图中的灰色图形是很难用东西来把它描述出来的，因此采用了如图所示的两种方式，一种多边形包围，一种椭圆形包围，明显的可以看出二者的问题分别在哪里，图 A 虽然包住了，但是精度较差，有大量的空白，比如在碰撞检测时，还没有碰上就测出来了。图 B 的问题再于包的精确度还可以，但是有的地方没有包住，这就导致可能比如在碰撞检测时，车都撞上了还可以向前走。

OSG 中有两个化繁为简的类 `osg::BoundingBox` 与 `osg::BoundingSphere` 一个是圆球形的包围体，一个是盒状的包围体，OSG 中采用的是类似于凸闭包的最大包围的方法，保证可以完全包住，但是精度怎么样……，只能用于粗略的计算。

类 **BoundingBox**:功能包围结点的几何体盒子

//看是否可用

bool valid () const

//设置/得到盒子的 XYZ，下面几种不同的方法

void set (float xmin, float ymin, float zmin, float xmax, float ymax, float zmax)

void set (const Vec3 &min, const Vec3 &max)

float & xMin ()

float xMin () const

float & yMin ()

float yMin () const

float & zMin ()

float zMin () const

float & xMax ()

float xMax () const

float & yMax ()

float yMax () const

float & zMax ()

float zMax () const

//得到盒子的中心点

const Vec3 center () const

//得到盒子的半径

float radius () const

//得到盒子半径的平方

```

float radius2 () const
//得到盒子的 pos 顶点的坐标，跟据估计，共有八个顶点
const Vec3 corner (unsigned int pos) const
//扩展盒子包含参数中所含的物体
void expandBy (const Vec3 &v)
void expandBy (float x, float y, float z)
void expandBy (const BoundingBox &bb)
void expandBy (const BoundingSphere &sh)
//计算与参数中盒子相交的盒子
BoundingBox intersect (const BoundingBox &bb) const
//看该子与参数中的盒子是否有交点
bool intersects (const BoundingBox &bb) const
//看该盒子是否包围该点
bool contains (const Vec3 &v) const

```

下面我们再来看一下 osg::BoundingSphere

类 osg::BoundingSphere,功能：得到某结点的包围球

```

//初始化包围球，可以从盒子与球初始化
BoundingSphere (const Vec3 &center, float radius)
BoundingSphere (const BoundingSphere &bs)
BoundingSphere (const BoundingBox &bb)
//看是否可用
bool valid () const
//设置圆心与半径
void set (const Vec3 &center, float radius)
Vec3 & center ()
const Vec3 & center () const
float & radius ()
float radius () const
float radius2 () const
//从参数中扩展该球，
void expandBy (const Vec3 &v)
void expandRadiusBy (const Vec3 &v)
void expandBy (const BoundingSphere &sh)
void expandRadiusBy (const BoundingSphere &sh)
void expandBy (const BoundingBox &bb)
void expandRadiusBy (const BoundingBox &bb)
//看是否包含某点
bool contains (const Vec3 &v) const
//看两球是否相交
bool intersects (const BoundingSphere &bs) const

```

关于光源的各个参数是需要实践的，希望广大读者在下面多做实验，来掌握其中的各个参数。

8.2 Billboard

Billboard 在植树造林方面可谓大出风头。随着人们想象力的与日俱增，与化繁为简的工作方式，使 Billboard 这种性价比很高的东西得到了广泛的应用。

在粒子系统中 Billboard 也颇为重要，占有一席之地。

8.2.1 Billboard 的数学原理

下面以图形始终对准视点的情形来介绍一下 Billboard 的数学原理。
如图所示：

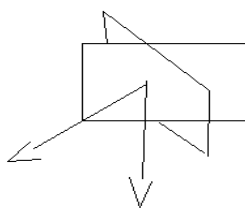


图 8.8

在许多书中称 Billboard 为广告版技术，现在设置广告版的法向量为 B ，观察体到广告版的位置单位向量为 L ，那么广告牌在随着视点转动的过程中需要转的角度 $D = \cos^{-1}(LB)$ ，计算出这个角度就可以使它按照这个角度旋转了。

OSG 中的 Billboard 可以设置物体指向任意一个相对于视口的方向。

总之来说，广告牌是一种冒的特例。

8.2.2 osg::Billboard

现在我们来看一下 OSG 中的 Billboard 类：

类 Billboard 功能：使物体始终朝向视口

//设置/得到模式，具全来讲是绕着什么转，有三个值：POINT_ROT_EYE, POINT_ROT_WORLD, AXIAL_ROT

void setMode (Mode mode)

Mode getMode () const

//设置/得到绕着的轴（如果是绕轴转动的话）

void setAxis (const Vec3 &axis)

const Vec3 & getAxis () const

//设置面朝的面的法向

void setNormal (const Vec3 &normal)

const Vec3 & getNormal () const

//设置/得到物体位置（序列）

void setPosition (unsigned int i, const Vec3 &pos)

const Vec3 & getPosition (unsigned int i) const

void setPositionList (PositionList &pl)

```

PositionList & getPositionList ()
const PositionList & getPositionList () const
//加入结点
virtual bool addDrawable (Drawable *gset)
virtual bool addDrawable (Drawable *gset, const Vec3 &pos)
//移除结点
virtual bool removeDrawable (Drawable *gset)
//计算矩阵
bool computeMatrix (Matrix &modelview, const Vec3 &eye_local, const Vec3 &pos_local) const
//计算包围球
virtual BoundingSphere computeBound () const

```

下面我们来具体的做一个例子：

例子功能，一果小树始终朝向视口，旁边有一只飞机做对比：

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testBillboard 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
 OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
 osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.lib osgTextd.lib
 osgUtil.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

在 main.cpp 中加入如下代码：

```

Main.cpp
//By FreeSOUTH for osgFreeS Source Code
#include <osg/Node>
#include <osg/Geometry>
#include <osg/Texture2D>
#include <osg/Billboard>
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
//创建一个贴着树的视图
osg::Drawable* createSquare (const osg::Vec3& corner, const osg::Vec3& width, const
osg::Vec3& height, osg::Image* image=NULL)
{
    osg::Geometry* geom = new osg::Geometry;
    osg::Vec3Array* coords = new osg::Vec3Array(4);
    (*coords)[0] = corner;
    (*coords)[1] = corner+width;
    (*coords)[2] = corner+width+height;
    (*coords)[3] = corner+height;
    //设计顶点相关
    geom->setVertexArray (coords);
}

```

```

//计算法向量
osg::Vec3Array* norms = new osg::Vec3Array(1);
(*norms)[0] = width*height;
(*norms)[0].normalize();
//设置法向量序列
geom->setNormalArray(norms);
geom->setNormalBinding(osg::Geometry::BIND_OVERALL);
//设置纹理关联
osg::Vec2Array* tcoords = new osg::Vec2Array(4);
(*tcoords)[0].set(0.0f,0.0f);
(*tcoords)[1].set(1.0f,0.0f);
(*tcoords)[2].set(1.0f,1.0f);
(*tcoords)[3].set(0.0f,1.0f);
geom->setTexCoordArray(0,tcoords);
geom->addPrimitiveSet(new osg::DrawArrays(osg::PrimitiveSet::QUADS,0,4));
//如果有贴图
if (image)
{
    osg::StateSet* stateset = new osg::StateSet;
    osg::Texture2D* texture = new osg::Texture2D;
    texture->setImage(image); //设置贴图

stateset->setTextureAttributeAndModes(0,texture,osg::StateAttribute::ON);
    stateset->setMode(GL_BLEND,osg::StateAttribute::ON); //开启A通道
    stateset->setMode(GL_LIGHTING,osg::StateAttribute::OFF);
    //把灯光关闭,如果处理不好灯光,将会一片黑,有贴图就可以了
    geom->setStateSet(stateset);
}
return geom;
}

osg::Node* createModel()
{
    osg::Group* root = new osg::Group();
    osg::Billboard* bbd = new osg::Billboard();
    //设置方式为绕轴转
    bbd->setMode(osg::Billboard::AXIAL_ROT);
    //绕z轴转
    bbd->setAxis(osg::Vec3(0.0f,0.0f,1.0f));
    //面朝Y轴
    bbd->setNormal(osg::Vec3(0.0f,1.0f,0.0f));
    bbd->addDrawable(createSquare(osg::Vec3(-0.5f,0.0f,-0.5f),osg::Vec3(1.0f,0.0f,0.0f),osg::Vec3(0.0f,0.0f,1.0f),osgDB::readImageFile("Images/tree0.rgba")),
        osg::Vec3(5.0f,0.0f,0.0f));
}

```

```

    root->addChild(bbd);
    return root;
}
int main( int argc, char **argv )
{
    osgProducer::Viewer viewer;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::Node* rootNode = createModel();
    ((osg::Group *)rootNode) ->addChild (osgDB::readNodeFile ("glider.osg")) ;
    viewer.setSceneData(rootNode);
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();
    return 0;
}

```

编译运行。

运行结果：

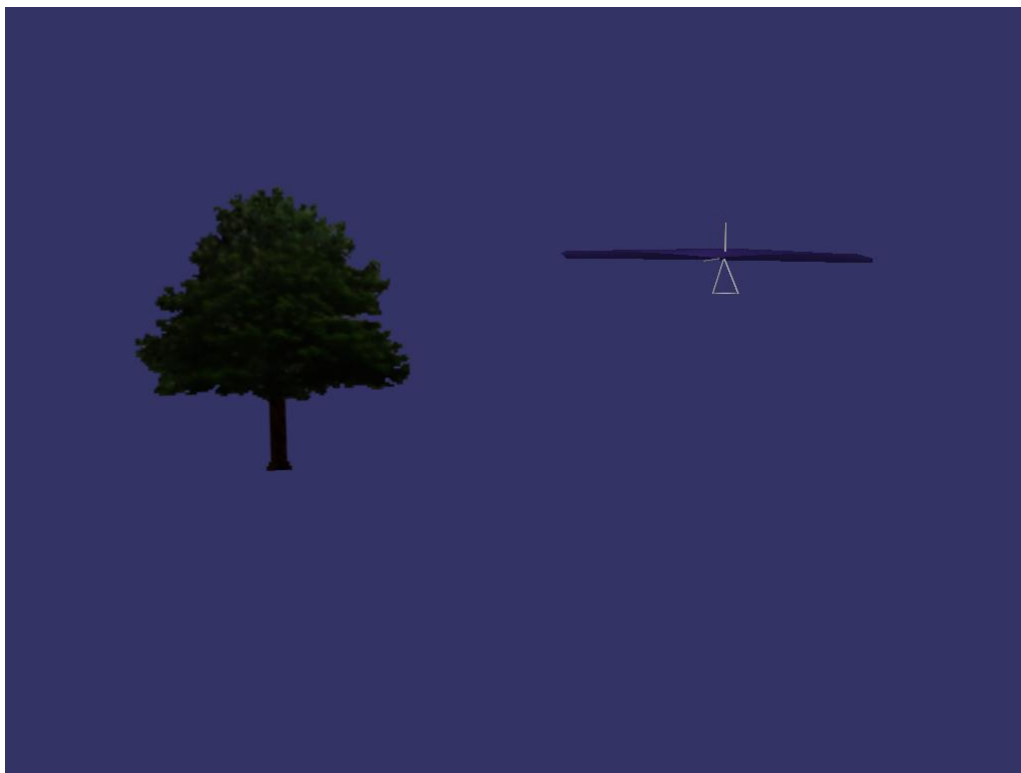


图 8.7

在使用 BBD 时，可能遇到很多意想不到的问题，比如纹理切割，等等。还有就是在设置 BBD 的时候，可能发现它会转圈，这是因为设置的轴与绕得方式有问题，可以多试几下。

8.3 LOD

LOD 本身就是个很复杂的东西，但是确应用得很广的一种解决问题的方法。现如今大多数，尤其在游戏行业，LOD 的使用技术基本上最基本的构造技术，拿最新出的风游《完美世界》来说，随着视口的前进远处的山峰会慢慢的显示出轮廓，近处的物体会由透明到完全可见，周围一片雾蒙蒙。这就是 LOD。

本节我们将使用 OSG 中的 LOD 类，以及使用动态图片代替模型与传统的图片不同，老实说：效果极其惊人的好，等一会儿你就可以看到了。

8.3.1 LOD 过程

LOD 过程包含有三个过程，即：LOD 的生成，切换以及选择，现在首先来介绍一下 LOD 的过渡问题：如何平滑的从一个过程过渡到另外一个过程是一个非常严肃的问题，这将直接关系到 LOD 效果的好坏，记得在一年前，我在看一个演示时，突然出现的楼盘着实吓了我一个大跳。

LOD 的切换：

最差的切换方式为离散几何 LOD，在后台首先准备多个模型，一个比一个简化，在距离近时换上另外一个模型，这种方法是最基本的方法，突跃现象十分吓人，故是为最差的选择。

其次是线性混合型的切换，在后台对两个离散 LOD 进行一次线性混合，但是这会大大的增加系统的开销，且在缓存中容易出现意想不到的问题。还有一种为淡入淡出式 LOD，对两个 LOD 同时可视，在临切时，一个 LOD 的透明值逐渐增大，另一个逐渐缩小。这样的结果还算差强人意，在现实中也略有应用。

使用最广的还是 CLOD，在近处到远处的过程中，把某多边形近处的两个顶点合成一个顶点，这样就造成塌陷，随着越来越远，顶点越来越少，开销就越来越少，反之亦然，现实中这种方法得到了广泛的应用。

LOD 的选取：

一般而言 LOD 是基于距离来选取的，也有少数是基于时间与投影面积的。

8.3.2 osg::LOD

下面我们来看一下 LOD 在 OSG 中的类：osg::LOD

类 LOD :功能，制做 LOD

//添加一个结点

virtual bool addChild (Node *child)

//并设置在离视点的哪个范围内出现它

virtual bool addChild (Node *child, float min, float max)

//移除一个结点

virtual bool removeChildren (unsigned int pos, unsigned int numChildrenToRemove=1)

//设置/得到中心点模式，用自定义中心点还是模型中心点 USE_BOUNDING_SPHERE_CENTER, USER_DEFINED_CENTER

void setCenterMode (CenterMode mode)

CenterMode getCenterMode () const

//设置中心点


```

void setCenter (const Vec3 &center)
const Vec3 & getCenter () const
//设置半径
void setRadius (float radius)
float getRadius () const
//设置/得到模式为视点距离模式还是屏幕像素距离模式
void setRangeMode (RangeMode mode)
RangeMode getRangeMode () const
//设置距离，在哪个段中显示
void setRange (unsigned int childNo, float min, float max)
//得到显示的最小距离
float getMinRange (unsigned int childNo) const
//最大距离
float getMaxRange (unsigned int childNo) const
//有几个孩子，即：有几个段显示
unsigned int getNumRanges () const
void setRangeList (const RangeList &rangeList)
const RangeList & getRangeList () const
//计算包围球
virtual BoundingSphere computeBound () const

```

下面我们来做一个例子，远处显示一个飞机，再远处显示起了火的飞机（均为 OSG 自带的模型）。

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testLOD 其它的保持默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.lib osgTextd.lib
osgUtd.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP，文件名取为 main.cpp

```

Main.cpp
//By FreeSouth for osgFreeS Source Code
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
#include <osgProducer/Viewer>
#include <osg/LOD>
int main( int argc, char **argv )
{
    osgProducer::Viewer viewer ;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::Node * node = osgDB::readNodeFile ("cessnafire.osg") ;
    //创建LOD,第一个距离为0-100，第二个为100-200
    osg::LOD * lod = new osg::LOD ;
    lod ->addChild (osgDB::readNodeFile ("cessna.osg"),0,100) ;
}

```

```

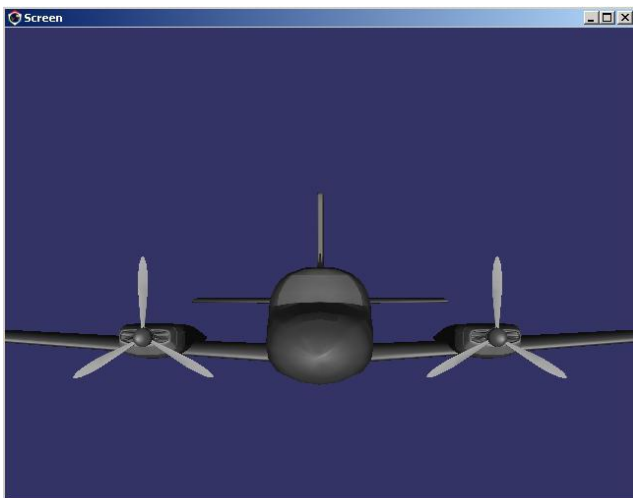
lod->setRangeMode (osg::LOD::DISTANCE_FROM_EYE_POINT) ;
lod->addChild (node, 100, 200) ;
osg::Group* root = new osg::Group;
root ->addChild (lod) ;

viewer.setSceneData( root );
viewer.realize();
while( !viewer.done() )
{
    viewer.sync();
    viewer.update();
    viewer.frame();
}
viewer.sync();
viewer.cleanup_frame();
viewer.sync();
return 0;
}

```

编译运行。

运行结果：



在近处 0—100



在远处 100—200

图 8.8

8.3.3 osgSim::Imposter

这个类为大家提供了一种动态图片来显示场景的过程，在远处时它会变成一张图片，且随着角度的不同，图片会不同。

我们来建立这样一个例子，看一下在近处显示与远处显示有什么不同，效果怎么样。

示例：Imposter

第一步：新建→项目→WIN32/WIN32 控制台应用程序，项目名称填入：testImposter 其它的保持

默认。

第二步：在菜单项目→属性→配置属性→链接器->命令行中添加下列 LIB：
OpenThreadsWin32d.lib Producerd.lib osgd.lib osgDBd.lib osgFXd.lib osgGAd.lib
osgParticled.lib osgProducerd.lib osgSimd.lib osgTerrain.d.lib osgTextd.lib
osgUtil.d.lib

第三步：对解决方案资源管理器视图中源文件击右键→添加→新建项→代码/CPP ，文件名取为
main.cpp

```
Main.cpp
//by FreeSouth for osgFrees Source Code
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osgSim/Impostor>
int main( int argc, char **argv )
{
    osgProducer::Viewer viewer ;
    viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
    osg::Node * node = osgDB::readNodeFile ("ceep.live") ;
    //创建imposter,在0-50000距离内显示模型，在1000距离以外显示贴图
    osgSim::Impostor * sim = new osgSim::Impostor ;
    sim->addChild (node,0,50000) ;
    sim ->setImpostorThreshold(1000) ;
    osg::Group* root = new osg::Group;
    root ->addChild (sim) ;
    viewer.setSceneData( root );
    viewer.realize();
    while( !viewer.done() )
    {
        viewer.sync();
        viewer.update();
        viewer.frame();
    }
    viewer.sync();
    viewer.cleanup_frame();
    viewer.sync();
    return 0;
}
```

编译运行。

如图：除了有个框框之外，你能看出二者有什么区别吗？

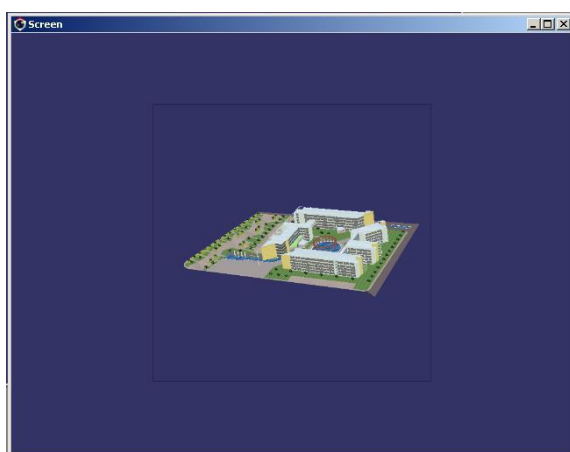
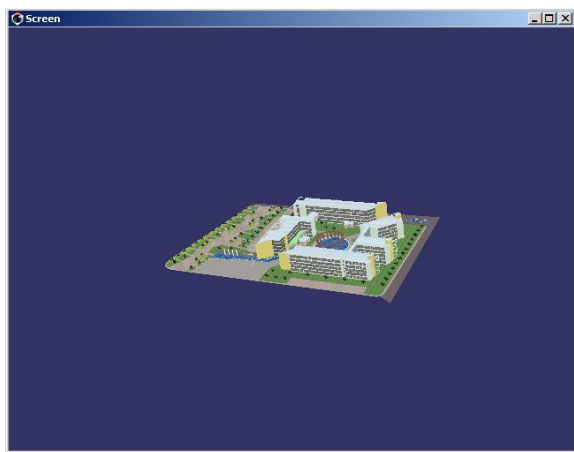


图 8.9

本章到现在就已经结束了，希望大家在这章的学习在中有所收益。

关于本书

写到这里，我感觉到分外的遗憾，写这本书的初衷是要介绍更多的关于 OSG 方面的基础知识，本书本还应该包括下面的内容：多线程与 OSG，多通道演示，外设控制，OSGEdit 的使用以及高程图的渲染与地形的生成。关于多线程与 OSG 和高程图的渲染与地形生成，我觉得应该属于高级的内容，而且由于我的水平有限，有可能不能很好的讲述清楚，且这些东西本来就十分的不容易，如果把它记录下来，可能要花费十分长的时间，我是在做别的事情时，间断来写这本书的，所以很遗憾，我**必须**继续我以前所做的事情。时间无多。

从学习 C 语言到学习 C++ 到学习 OPENGL, D3D 与 JAVA 再到 OSG，每一步都有着十分艰辛的历程，我只是觉得自己很幸运，在我二十岁年华之际遇到我的两位好老师：以及郑州大学虚拟实验室曹明亮与赵红领老师。一个好的好师可以让人少走许多弯路，给人以思想上的熏陶与渲染（三句不离本行）。两个好的老师可以让你的编程水平平步青云，指鹿为马，扶摇直上。到四月十二日时就是我二十二岁的生日，也许从那天开始，我再也不会再从事 OSG 与虚拟现实这个行业了，我对所有对我做过帮助的老师与同学表示衷心的感谢。

我很遗憾的感到，学习 OSG 不是为了使用 OSG，也许这是件很矛盾的事情，我只是觉得 OSG 的功能太小，国内研究 OSG 的人太少，成功的例子太少，愿意真正介绍 OSG 的人太少，最后 OSG 玩来玩去做出来的大的项目竟然靠的全是建模，这是一件多么荒谬的事情。建模虽然说在游戏编程与虚拟现实行业占有半壁江山，但是只是觉得 OSG 没有占有像 OPENGL 与 D3D 在游戏行业那样的位置，那样具有更多技巧性的工作。是它完美了吗？

我希望大家可以踊跃的写出关于 OSG 更多的技术专题。不要停留在表面工作上来，不要到头来一个贴在 OSG 官网上的 DEMO 竟然除了建了一个好模型外一无所有。

我再重述一遍关于如何学好 OSG 的我的感受：第一步是：从原理上弄懂 OSG 中官方所有的例子，不懂的要去看相关图形学书籍。大量的编代码。第二步是：大胆的想象，只通过修改 OSG 官方的例子就可以完成许多不可想象的功能。第三步是：读源码，有很多东西，不从源码继承是做不出来的。例子毕竟有限。第四步，也是重要的一步：写自己的算法，自己的引擎，自己的软件。

最后我祝愿所有初学者通过参考本书，可以更好的学习 OSG。

如对书中的代码有疑问：请发邮件至 ieysx@163.com 或登录 WWW.VRDEV.NET

FreeSouth
2007 年愚人节