|  |  |  |
| --- | --- | --- |
|  |  | |
| **海洋GIS三维引擎软件** | |
| XXXX-XXX-XXX-XXXX-20XX | |
| 详细设计 | |
|  | |
| 中电科海洋信息技术研究院有限公司  2016年11月 | |

# 1.范围

### 1.1系统概述

海洋GIS三维引擎软件主要应用于海洋态势三维显示、海洋环境模拟等，能够实现空间信息从全球到微观尺度的一体化表达、信息查询、量算分析、战场态势标绘、仿真与推演等功能。具有跨平台特点，可在Windows、Linux操作系统上运行，提供二次开发功能，支持多种软件架构。具有地理数据处理及显示功能、常用功能、量算功能、标绘功能、辅助数据处理功能、海洋显示支撑功能等六大功能。

### 1.2文档概述

本文档包含海洋GIS三维引擎软件详细设计的相关内容，是下一步开展软件配置项管理、实施面向接口编程的依据。

# 2.软件功能

海洋GIS三维引擎软件具备以下六大类基本功能：

* + 1. 地理数据处理及显示功能
    2. 常用功能
    3. 量算功能
    4. 标绘功能
    5. 辅助数据处理功能
    6. 海洋显示支撑功能

### 2.1地理数据处理及显示功能

1. 能够高效处理和管理全球多源、多时相、多分辨率的格式公开的地图数据，包括图像格式数据（如TIFF、JPEG、PNG、BMP、DDS 等）、各种标准影像数据、地形高程模型（DEM）数据（如GeoTIFF、IMG等）、地名、矢量地图数据（如shp、mif等）等；
2. 能够支持用户将各类矢量信息（如shp、mif等）处理为多级图像数据叠加到三维球显示，支持用户使用插件模式扩充自有数据格式；
3. 能够构建地形金字塔模型并根据相机参数进行灵活调度。

### 2.2常用功能

1. 支持对显示区的管理，可设置显示区的范围，可管理相机及辅助相机（主要是相机的镜头调整、定位、各种坐标系的切换矩阵），可对屏幕、地理、世界坐标进行相互转换，可对场景操作器管理；
2. 能够进行各类常见操作，如漫游、缩放、俯仰、旋转、复原等；
3. 支持罗盘指北针的渲染以及显隐、样式控制；
4. 支持天空、星空、大气、云的渲染以及显隐、样式控制；
5. 支持经纬网格、高斯网格以及其他投影网格的渲染以及显隐、样式控制；
6. 支持常见地理坐标系（如WGS84、CSCG2000等）及常见投影坐标系（如墨卡托投影、高斯投影等），以及各坐标系之间的转换计算。

### 2.3量算功能

1. 能够支持空间信息查询与提取，包括点查询、矩形区域查询、不规则区域查询等，支持以图形化方式显示查询结果；
2. 能够支持基本量算分析，包括坐标量算、高程量算、距离量算、面积量算、剖面量算、坡度量算、通视计算、方位角计算等，需能将量算过程中产生的辅助信息（如角度、长度等）进行显示，计算结果需能按贴地纹理和不贴地纹理两种方式显示。

### 2.4标绘功能

1. 能够支持符号标绘标绘对象包括三维实体模型（如3DS、FLT、obj等）、三维军队标号、图元符号（如折线类、曲线类、闭合折线类、圆弧类、面、方体、球体、椎体等），支持模型采用多层次细节(LOD)技术显示；
2. 能够符号属性标绘，属性对象包括标牌、高度指示线、探测属性、轨迹线、威力圈范围、声音、尾焰等；
3. 能够支持符号管理功能，包括对标绘符号数据进行分层管理，支持按名称或者坐标查询符号，支持对符号的快速拾取等工能；
4. 能够支持符号编辑功能，包括支持符号选中、平移、缩放、旋转、俯仰、翻滚等功能，支持对属性的添加、删除、REDO/UNDO等编辑功能，支持对属性所涉及的颜色、线性、填充等进行编辑；
5. 支持标绘面板功能，即将a）中涉及的符号标绘、符号属性标绘、符号管理及编辑等功能集成至简洁易用的标绘面板；
6. 能够支持多种特效显示，包括爆炸、烟火、尾焰等效果；雾、雨、雪、红外、夜视等气象效果；海浪、云、天空等效果；环境、装备声音等音效功能；
7. 能够支持空中目标、空间目标、用频目标等目标的态势显示。空中目标态势显示功能包括原始航线、光滑后的航线、航空管道、高度指示线等；空间目标态势显示功能包括轨道、星下点轨迹、姿态球、卫星相机地面轨迹及历史轨迹、作用圈、覆盖带轨迹、卫星传感器波束等；用频目标态势显示功能需显示各类常见功率域包络，包括球类、扇形类、天波超视距类、相控阵类等。
8. 支持仿真引擎，提供仿真时间管理及仿真控制功能，能驱动场景实体进行简单的仿真推演；
9. 支持场景记录与回放功能，记录与回放内容包括操作过程、标绘过程等。

### 2.5辅助数据处理功能

1. 支持将矢量（如shp、mif、GJB5068等）、高程（如GeoTIFF、IMG等）、影像（如TIFF、JPEG、PNG、BMP、DDS 等）等数据格式转换为平台自定义格式；
2. 支持常见矢量数据格式（如shp、mif、GJB5068等）、常见高程数据格式（如GeoTIFF、IMG等）等的相互转换；
3. 支持用户将各类矢量信息（如shp、mif等）处理为多级切片格式叠加到平台显示；
4. 支持将高程数据处理为晕渲图切片叠加到平台显示。

### 2.6海洋显示支撑功能

1. 海面渲染方式需包括四种，第一种：仿真方式（即波浪方式）；第二种：影像方式；第三种：电子海图切片方式；第四种：透明方式。方式间可以任意切换。
2. 海底与陆地的无缝接边；
3. 海底数据支持两种，第一种：公开的从互联网可下载的深度数据；第二种：从电子海图数据中提取的海底深度数据（定义海图中提取数据的存储方式，可提取的数据包括：等深线及深度点数据等）。同时，需对海底数据进行插值建模；
4. 海底纹理需方式支持三种方式。第一种：纯色；第二种：电子海图纹理；第三种：根据海底深度生成的晕渲图。方式间可以任意切换；
5. 海洋水体得有沉浸感，可以通过多种方式衬托出水体的存在，比如水中气泡、水中物体（鱼类、钻井平台、从水底观测水面的船底等）等。
6. 支撑水文、气象等信息的显示。显示方式支持两种，第一种栅格数据显示，第二种矢量数据显示。同时支持多帧纹理、体数据绘制等。

# 3.概要设计

### 3.1组件化体系结构

海洋GIS三维引擎软件采取层次化、组件化的体系结构，包括三个层次，每个层次由一系列相互协作的组件构成。如图1所示：



图1 组件化体系结构

#### 2.1.1层次化体系结构

海洋GIS三维引擎软件内部采取层次化的体系结构，其中包括主控层、功能层和应用层三个层次：

* 主控层：负责统一接收和调度系统的功能命令、鼠标事件、键盘事件和绘图事件，能够通过标准的接口统一分发给功能处理层，并对返回的消息进行统一的处理。主控层为整个系统提供一个统一的入口点，为各项功能提供一个集成框架，实现对系统的总体控制以及各个功能处理模块之间的无缝集成与协调工作，各模块之间是松耦合或无耦合关系；
* 功能层：主要完成三维态势图显示的各项功能，包括三维场景控制、快速定位、坐标量算、目标叠加、模型输入、自定义绘制、数据容量及绘制速度、三维实物模型容量等。功能层中的各个处理模块接收既可由内部控制层统一分发的各项系统功能命令、鼠标事件、键盘事件和绘图事件完成相应的功能处理，也可以通过获取专有接口，进行API级的调用，实现各项功能；
* 应用层：是位于功能层之上的一个非定向模块，是用户针对特定应用的一种系统级体现；该层主要利用功能层提供的接口进行开发，将相应的功能进行有机组合，以实现特定的应用，如三维查询、三维分析、三维建筑物等；应用层的操作应遵循相应的开发规则，确保应用的正确实施。

#### 2.1.2组件化体系结构

海洋GIS三维引擎软件采用组件化的软件设计思想，在一个主控模块的全局控制下，将各个基础功能模块设计成独立的内部组件，通过主控对每个组件进行注册和加/卸载，各组件之间通过主控进行消息和数据的交互。开发人员依据组件协议可以创建自己的扩展模块，实现平台的开发、应用和维护。

### 3.2二次开发与系统集成

#### 2.2.1系统集成

在使用方式上，海洋GIS三维引擎软件无缝集成到系统应用中，通常是在统一的模式下，通过主控对窗口内的各种信息进行统一的显示控制和操作，软件与应用中其它图形相关的模块有机地集成在一起协同工作，共享统一的绘图接口、鼠标消息、键盘消息、菜单命令等系统资源，并通过统一的图形定位与显示控制，实现各类图形显示的精确配准与协调控制。

#### 2.2.2二次开发接口

海洋GIS三维引擎软件提供灵活的二次开发接口，把各项功能以应用编程接口（API）的形式提供给二次开发用户使用，使用户可以方便地将各项功能应用到各种需求中去。

# 4.详细设计

### 4.1系统结构

从系统结构层面，海洋GIS三维引擎软件分为两大部分：核心功能部分及数据服务部分（如图2所示）。其中，核心功能部分又分为图形基础层、专业核心层、业务应用层和应用交互层；数据服务部分负责为核心功能提供各种数据服务，包括本地数据、局域网数据和广域网数据。

图2 系统结构图

文字

骨骼动画

阴影

街景图

点云

特效

渲染后台

**信息查询**

**Query**

**专业核心层**

海洋

Ocean

粒子系统

场景图（SceneGraph）

**空间分析**

**SpatialAnalysis**

**业务应用层**

**文件数据**

**本地数据**



**地图绘制**

**DBMS**

**Gateway**

**局域网络**

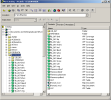
**空间数据库**

**Internet**

**Services**

**广域网络**

**要素集服务**



**元数据服务**

**数据服务**

**应用交互层**



**地图服务**

**定位服务**

**分析工具集**

**Linux**

**客户端**

**Windows**

**客户端**

**……**

**客户端**

**安卓**

**客户端**

**浏览器**

**客户端**

**业务建模**

**业务**

**逻辑**

**业务**

**逻辑**

**仿真**

**组件**

**后端处理与维护工具**

**业务**

**逻辑**

***HTTP协议***

**仿真**

**组件**

**实体建模**

**仿真专题**

**实体模型**

标绘

Plot

电磁

插件

航天

插件

航空

插件

…

**地理环境**

**实体模型**

**图形建模**

**图形引擎**

**公用基础（Common和public）**

**主控**

**MainCtrl**

**视景器**

**Viewer**

**交互适配器**

**GUI Adapter**

天空

Sky

地形

Earth

建筑物

Building

**图形基础层**

### 4.2模块之间的关系

海洋GIS三维引擎软件分为四大类模块（如图3所示），包括：主控、组件、插件、基本功能库。

**1）主控**

该类模块负责组件注册、接口查询、消息与命令分发、鼠标状态选择等。

**2）组件**

该类模块提供相应的功能与接口，可以与用户交互，如三维分析组件、三维海洋组件、三维天空组件、三维标绘组件等。其中，三维主控是一个特殊的组件，它同时又是一个二级主控（是主控的组件又是三维各模块的主控），负责对所有三维组件进行注册、查询。

**3）插件**

该类模块以任务链方式提供各种读写、绘图等功能，不能与用户交互。如各类地形数据读取插件、三维模型读写插件、三维标绘模型及属性创建插件等。

**4）基本功能库**

该类模块提供基本功能的动态库，使用者需静态依赖该类库。如三维渲染引擎模块、公共基础库模块、三维地图模块等。

主 控

（VBF\_MainCtrlDll）

**三维主控**

**（VBF\_3DMainCtrl）**

二维、标图等

（预留接口）

对各组件（或二级主控）进行注册、查询

二级主控（是主控的组件又是三维各模块的主控）对三维组件进行注册、查询

视图与人机交互

（VBF\_Display）

对视景器、视图、GUI进行封装。包括VBF\_Widget（窗口部件）、VBF\_Manipulator（相机操作器）、VBF\_Dragger（模型拖放器）、VBF\_Viewer（视景器、视图）等动态库。

三维绘制引擎

（VBF\_3DEngine）

对OpenGL的封装，负责渲染模型的管理与绘制。每个实体模型对应一个渲染模型，实体模型从业务角度对地理世界进行建模，渲染模型从OpenGL的角度进行建模。包括VBF\_SceneGraph（场景图组织）、VBF\_SGUtil（渲染工具，即渲染内核）、VBF\_SGDB（数据库管理）以及基础图形绘制扩展（VBF\_Text，文字；VBF\_Particle，粒子系统；VBF\_Cal，骨骼动画；VBF\_Animation，自定义动画等）

VBF接口

（VBF\_Interface）

二次开发数据类型与接口，供外部使用

天空盒、大气、天气（雨、雪）、太阳、月亮、星空等

通用绘图组件

（VBF\_3DPlotDll）

创建与绘制各种实体模型，提供了父子关系、属性、过程等机制，支持外部插件建立实体模型、属性、过程等

航空插件

（\_Aviation）

电磁插件

（\_ElecMag）

基本图形插件

（\_Shape）

三维海洋组件

（VBF\_3DOcean）

建筑物组件

（VBF\_3DBuilding）

三维天空组件

（VBF\_3DSky）

星空组件

（VBF\_Space）

基础布局组件

（VBF\_3DLayout）

公共基础库

（VBF\_Common）

各模块公用的基础性功能。包括VBF\_Base（引用计数与状态管理）、VBF\_Thread（线程库）、VBF\_XML（XML数据访问）、VBF\_Math（数学计算）、VBF\_GeomMath（地理计算）、VBF\_Utils（文件、动态库等）

三维地球

（VBF\_Earth）

与地球三维相关的模块。包括VBFE\_Earth（内核框架，空间参考、地图、图层）、VBF\_Features（矢量数据的管理与显示）、VBF\_Annotation（注记）、VBF\_Symbology（地图符号）、VBF\_EarthUtil（工具）、VBF\_EarthDrivers（各种地理空间数据加载器）

空间数据库

（VBF\_DB）

自定义格式的空间数据加载器，同VBF\_EarthDrivers，主要提供了VBF各版本数据及自定义缓存数据的加载。

VBF公用部分

（VBF\_Public）

内部使用的公用数据类型、公用接口，目前与VBF\_Interface未完全分开，二者通用

数据IO封装

（VBF\_IOWrappers）

对图像、模型、对象等各种专业数据格式进行访问的读写器。包括OSG自己的格式、各种图像、影像格式、三维模型格式、压缩格式等。

指北针、鼠标位置显示、视点位置显示、帮助等

常用模型插件

（\_Common）

开发库

（Lib）

**基础动态库**

**主控与功能组件**

组件：提供相应的功能与接口、可以响应用户的输入

插件：绘图插件，提供各类模型、属性、过程等的创建

主控：组件注册、接口查询、消息与命令分发、鼠标状态选择

提供基本功能的动态库，不与用户交互

提供二次开发的接口、数据类型、库

图3模块之间的关系图

### 4.3模块内部关系

在上述四大类模块中，有三个最为关键的模块：三维渲染引擎模块、三维地图模块及三维标绘模块，它们承担着整个软件的核心绘制与管理功能。

#### 4.3.1三维渲染引擎模块

该模块负责提供基本的图形和渲染功能（如图4所示）。整个模块又分为前台和后台两大部分。前台负责构建场景图、事件处理、用户交互等，后台负责图形渲染。二者均采用树状结构完成对图形数据的管理。

#### 4.3.2三维地图模块

该模块负责三维地图的构建与渲染（如图5所示），地图（Map）与地图节点（MapNode）是其中两个最为关键的项。其中，地图直接与用户交互，负责管理各个图层（包括高程图层、影像图层、矢量图层等）、空间参考系、高程瓦片大小、缓存策略等；地图节点根据用户设置的地图信息，构建多分辨率场景图。

#### 4.3.3三维标绘模块

该模块负责各类实体模型及其属性、过程的构建、管理与渲染（如图6所示），这些模型及其属性、过程均采用插件方式进行创建。

其中实体模型又分为点状实体模型和非点状实体模型。点状实体模型是指只有一个定位点的模型，可以移位、缩放、旋转，如布告牌、3D模型、骨骼动画、基本几何体、雷达、波束等；非点状实体模型是指有多个定位点的模型，可以对每个定位点移位，如航线、空域、通联线等。

模型属性是实体所具有的状态，为了使其结构和配置更灵活，采用了属性概念，可以使用XML文档配置实体属性。每个模型可以配置多个属性，每个属性也可以配置给多个模型。常见的属性有敌我标示、生命期、标牌、尾线、尾焰、尾迹、飘带、高度指示线等。属性可以替代过程，但为了概念清晰设计了过程。

模型过程是实体所发生的某种运动，又分为时间点触发的过程和时间段触发的过程。时间点触发的过程如分离、爆炸、消失等；时间段触发的过程如飞行、惯性运动等。

渲染器

Renderer

场景数据

SeceneData

2

1

组件

Component

视图

View

相机/子相机

Camera/Slave

图形设备/窗口

GraphicsContext

1

1

视景器

Viewer

n

1

事件处理器

GUIEventHandler

n

1

1

相机操作器

Manipulator

1

1…

n

n

1…

n

1…

n

1

1

1

地图

Map

地图图层

MapLayer

标绘文档

PlotDoc

……

标绘图层

PlotLayer

实体模型

PlotEntity

n

1

n

1

n

1

**场景显示的内容**

图4 三维渲染引擎模块

内部关系图

前台窗口及交互

前台与后台接口

后台绘制

前台场景接口

前台场景内容

场景视图

SceneView

渲染台

RenderStage

渲染元

RenderBin

裁剪操作器

CullVisitor

1

1

1

1

状态图

StateGraph

1

1

渲染叶

RenderLeaf

1

n

1

n

**状态树**

**渲染树**

状态机

StateMachine

*渲染信息*

*RenderInfo*

1

1

1

1

1

1

**场景树**

场景图节点

SGNode/Group

可视对象

Drawable

场景图页节点

Geode

1

n

1

n

1

1

状态属性

StateAttribute

状态集合

StateSet

1

n

数据插件

EarthDrivers

数据源

Sourcre

掩码图层

MaskLayer

影像图层

ImageLayer

模型图层

ModelLayer

地图图层

MapLayer

高程图层

ElevationLayer

图5 地图模块内部关系图

数据源需要数据插件进行读写，可以扩展插件支持不同数据

地图配置选项

MapOptions

概述

Profile

回调

MapCallbacks

缓存

Cache

空间参考

SpatialReference

地图

Map

地图配置项：包括图名、坐标系、高程瓦片大小、缓存策略等

地形图层

TerrainLayer

地物

Feature

1

n

回调

Callbacks

符号

Symbology

图层配置选项

ConfigOptions

注记

Annotation

地图节点：用于构建地图对应的场景图，属于内部概念。该节点层次复杂，在地形节点下分为四个根节点，即把整个世界空间分为四部分（只用了其中的两个，即0-0-0和0-1-0），每部分有一个分页LOD节点。每一级的每一个分页节点包含一个自身节点（低分辨率）和四个子节点（高一级分辨率）。

矢量数据节点

Map.Models

地图节点

SGNodeMap

矢量图层节点

ModelLayer

地形引擎节点

TerrainEngine

覆盖节点

TerrainOverlay

地形数据引擎容器节点

TerrainEngineContainer

分级瓦片节点

Tiles

地形节点

Terrain

1

n

记录标图数据

图6 三维标绘模块内部关系图

1

n

**父子**

1

n

标图文档

VBF\_3DPlotDoc

1

n

标图图层

VBF\_3DPlotLayer

绘图（标图）

VBF\_3DPlot

实体过程

VBF\_MarkProcess

1

1

**分类型**

**分类**

模型跟踪器

MarkTracker

图形编辑

VBF\_3DPlotEditor

1

n

事件

ProTimerPointTrigger

时间段触发的过程，如飞行、惯性运动

实体过程

VBF\_MarkProcess

1

1

模型输入器

MarkInputHandler

图形编辑含图形输入和图形修改

模型输入器

MarkInputHandler

1

1

模型风格配置

MarkStyleManager

非点状实体模型：是指有多个定位点的模型，可以对每个定位移位。Common插件中实现了多边形、贴地多边形、Polyline、Wall（垂面线）等。

移动局部点（**非点状**）

DraggerTranslateLocal

标图实体模型

VBF\_MarkModel

模型跟踪器

MarkTracker

模型跟踪器：

用于对模型的位置姿态、大小进行修改，分为点状和非点状。每种跟踪器由不同的拖放器（dragger）组合而成。

平移

DraggerTranslate

点状

VBF\_MarkModelPoint

PP

**点**

**状**

**模**

**型**

**编**

**辑**

实体属性

VBF\_MarkAttr

分层控制实体模型，包括显示开关、显示距离等

标图实体模型

VBF\_MarkModel

1

n

1

n

标图插件

VBF\_3DPlot\_Common ：通用

VBF\_3DPlot\_Shape ：图形

VBF\_3DPlot\_ElecMag：电磁

VBF\_3DPlot\_Aviation：航空

过程

ProTimerTrigger

PP

模型输入器：用于输入各种实体模型，除通用的点、线、面模型外，特殊的模型（如箭头）还拥有自己的输入器

时间点触发的过程，如分离、爆炸、消失等

使用XML文件，配置模型编码、数据、属性等

实体属性

VBF\_MarkAttr

实体属性：是实体所具有的状态，为了使其结构和配置更灵活，采用了属性概念，可以使用XML文档配置实体属性。每个模型可以配置多个属性，每个属性也可以配置给多个模型。常见的属性有敌我标示（IFF）、生命期（LifeTime）、标牌（Label）、尾线（TailLine）、尾焰、尾迹、飘带、高度指示线（HintLine）等。属性可以替代过程，但为了概念清晰设计了过程。

非点状

VBF\_MarkModelNotPoint

旋转

DraggerRotate

点状实体模型：是指只有一个定位点的模型，可以移位、缩放、旋转。Common插件中包括布告牌（Raster，包括静态和动态）、3D模型（Mesh，支持3ds、flt等多种格式）、骨骼动画（Cal3D）、LOD、组合（Combine）、单四边形（SingleQuad）、交叉四边形（CrossQuad）、二维矩形（Rect2D）、文字（Text）、基本几何体、粒子系统等，Shape插件实现了各种几何图形。

缩放

DraggerScale

# 5.接口设计

### 5.1开发包框架结构

整个开发包命名为VBF\_SOURCE，可部署在本机任意路径下（路径名中不能含有中文）。其子文件夹含义说明如表1所示：

表1 文件夹VBF\_SOURCE说明

|  |  |
| --- | --- |
| 子文件夹名称 | 子文件夹说明 |
| VBF\_App | 应用程序示例 |
| VBF\_CORE4.0 | 核心库的源代码及接口 |
| VBF\_Sample | 核心库各类功能的示例代码 |

其中，核心库代码VBF\_CORE4.0的子文件夹说明如表2所示：

表2 文件夹VBF\_CORE4.0说明

|  |  |
| --- | --- |
| 子文件夹名称 | 子文件夹说明 |
| debug | debug版动态库 |
| release | release版动态库 |
| VBF\_3DCom | 各类三维组件源码，包括三维分析、三维查询、三维海洋、三维天空等 |
| VBF\_3DEngine | 对OpenGL的封装，负责渲染模型的管理与绘制。每个实体模型对应一个渲染模型，实体模型从业务角度对地理世界进行建模，渲染模型从OpenGL的角度进行建模。包括VBF\_SceneGraph（场景图组织）、VBF\_SGUtil（渲染工具，即渲染内核）、VBF\_SGDB（数据库管理）以及基础图形绘制扩展（VBF\_Text，文字；VBF\_Particle，粒子系统；VBF\_Cal，骨骼动画VBF\_Animation，自定义动画等） |
| VBF\_AuxEarth | 辅助库源码，包括与地球有关的算法、操作器及专题地图等 |
| VBF\_AuxSG | 辅助库源码，包括与场景图有关的节点、算法及二维图形等 |
| VBF\_Common | 各模块公用的基础性功能。包括VBF\_Base（引用计数与状态管理）、VBF\_Thread（线程库）、VBF\_XML（XML数据访问）、VBF\_Math（数学计算）、VBF\_GeomMath（地理计算）、VBF\_Utils（文件、动态库等） |
| VBF\_DB | 自定义格式的空间数据加载器，同VBF\_EarthDrivers，主要提供了VBF各版本数据及自定义缓存数据的加载 |
| VBF\_Display | 对视景器、视图、GUI进行封装。包括VBF\_Widget（窗口部件）、VBF\_Manipulator（相机操作器）、VBF\_Dragger（模型拖放器）、VBF\_Viewer（视景器、视图）等动态库 |
| VBF\_Earth | 与地球三维相关的模块。包括VBFE\_Earth（内核框架，空间参考、地图、图层）、VBF\_Features（矢量数据的管理与显示）、VBF\_Annotation（注记）、VBF\_Symbology（地图符号）、VBF\_EarthUtil（工具）、VBF\_EarthDrivers（各种地理空间数据加载器） |
| VBF\_Interface | 二次开发数据类型与接口，供外部使用 |
| VBF\_IOWrappers | 对图像、模型、对象等各种专业数据格式进行访问的读写器。包括OSG自己的格式、各种图像、影像格式、三维模型格式、压缩格式等 |
| VBF\_MainCtrl | 主控源码 |
| VBF\_Plot | 三维标绘源码 |
| VBF\_Public | 内部使用的公用数据类型、公用接口，目前与VBF\_Interface未完全分开，二者通用 |
| VBF\_Symbol | 军标符号库源码 |
| VBF\_UI | 图形界面源码 |

### 5.2环境依赖

#### 5.2.1硬件环境要求

* 处理器：不低于四核（主流）
* 内存：2G以上
* 硬盘：500G以上
* 显卡：独立显卡
* 显存：1G以上

#### 5.2.2软件环境要求

* Windows环境，WindowsXP（含）以上版本（含32位和64位）
* Linux环境，内核2.6（含）以上版本（含32位和64位）
* Qt 4.7.4（32位）

### 5.3主要接口

#### 5.3.1主控模块

该模块中的主要接口包括：

##### 1）IVBF\_MainCtrl

主控接口。用户可通过全局函数VBF\_CreateMainCtrl()创建主控：

// 参数：szDllName -- 主控所在动态库的名字，不包含Debug版的标识和.dll后缀， 如“VBF\_3DMainCtrl”，而不是"VBF\_3DMainCtrlD.dll"

// pIMainCtrl -- 返回的主控接口

// pLib -- 返回的主控动态库接口

// szAppPath -- 应用程序的完整路径

// szSysSetPath -- 配置文件夹的完整路径（如果为NULL，表示位于应用程序下的/SysSet/文件夹）

bool VBF\_CreateMainCtrl(const char\* szDllName, IVBF\_MainCtrl\*& pIMainCtrl, VBF\_DynamicLibrary\*& pLib,const char\* szAppPath, const char\* szSysSetPath=NULL);

接口IVBF\_MainCtrl的定义如下：

class IVBF\_MainCtrl : public IVBF\_Observer, public IVBF\_Subject

{

public:

virtual ~IVBF\_MainCtrl() {}

public:

virtual char\* GetType() = 0;

// 获取/设置主控的ID值。

virtual long GetID() = 0 ;

virtual void SetID(long lID) = 0 ;

// 获取/设置主控的用途。

virtual void SetPurpose( const char\* pPurpose )=0;

virtual char\* GetPurpose( ) = 0;

// 删除系统主控对象。

virtual void Release() = 0;

// 从动态库文件注册组件。

// 根据指定的组件动态库文件，注册组件。参数为组件动态库文件的全路径名。

// szComponentFile:组件所在动态库的文件路径。

// return 如果成功删除返回true，否则返回false。

virtual bool RegisterComponentFile(const char\* szComponentFile) = 0;

// 从配置文件注册组件。

// 根据指定的组件配置文件，注册组件。参数为组件配置文件的全路径名。

// 组件配置文件中包含要注册的一组组件动态库名称。

// szComponentCfgFile ：组件所在配置文件的路径。

virtual bool RegisterComponentCfgFile(const char\* szComponentCfgFile) = 0;

// 注册/卸载已加入系统中的全部组件。

virtual void RegisterComponentCfgFileDefault() = 0;

// 删除指定组件。

// szComponentID:组件ID。

// return 如果成功删除返回true，否则返回false。

virtual bool RemoveComponent(const char\* szComponentID) = 0;

// 指定组件ID，获取组件接口指针。

// szComponentID:组件ID。

// return 组件指针，如果没有该组件返回NULL。

virtual IVBF\_Component\* GetComponentPtr(const char\* szComponentID) = 0;

// 获取指定的接口指针。

// 由于无法知道每个接口的类型，所以每个接口都以void\*返回，

// 调用者需要通过强制类型转换成需要的接口类型。

// szInterfaceID:接口ID。

// 接口指针。 szComponentID,排出的组件ID，防止反复查找，形成死循环

virtual void\* QueryInterface(const char\* szInterfaceID,const char\* szComponentID = NULL) = 0;

// 执行指定命令。

// szCommandID:命令ID。

// pInParam:输入参数，其含义有命令本身决定。

// pOutParam:输出参数，其含义有命令本身决定。

// return 如果执行成功返回true，否则返回false。

virtual bool ExecuteCommand(const char\* szCommandID, IVBF\_PropertySet\* pInParam, IVBF\_PropertySet\* pOutParam) = 0;

// 选中交互工具。

// szToolID:交互工具ID。

// return 如果选择成功返回true，否则返回false。

virtual bool SelectTool(const char\* szToolID) = 0;

// 获取当前工具。

// return 当前工具ID。

virtual const char\* GetCurrentTool() = 0;

// nMouseEvent:鼠标事件类型，可以是如下值之一:

// DTIS\_MOUSE\_LBUTTONDOWN、DTIS\_MOUSE\_LBUTTONUP、 DTIS\_MOUSE\_LBUTTONDBLCLK、DTIS\_MOUSE\_RBUTTONDOWN、

// DTIS\_MOUSE\_RBUTTONUP、DTIS\_MOUSE\_MOVE、DTIS\_MOUSE\_WHEEL、DTIS\_MOUSE\_CANCEL。

// nKeyStatus: 键盘状态，可以是如下值的任意个:

// DTIS\_MOUSE\_LBUTTON、DTIS\_MOUSE\_RBUTTON、DTIS\_MOUSE\_SHIFT、DTIS\_MOUSE\_CONTROL、DTIS\_MOUSE\_MBUTTON。

// nX，nY:鼠标的当前位置。

// nDelta:鼠标滚轮的滚动尺寸。

// return 0表示鼠标事件不成功，其它值表示成功，每个值的含义由具体组件决定。

//---------------------------------------------------------------

virtual bool OnMouseEvent(int nMouseEvent, int nKeyStatus, int nX, int nY, int nDelta) = 0;

// nKeyEvent:键盘事件类型:可以是如下值之一

// DTIS\_KEY\_DOWN、DTIS\_KEY\_UP、DTIS\_KEY\_CHAR。

// nChar:键盘按下的字符。

// nRepCount:重复按下次数。

// nFlag:状态参数，具体含义参见msdn中OnKeyDown,OnKeyUp中nFlags的说明。

// return 0表示鼠标事件不成功，其它值表示成功，每个值的含义由具体组件决定。

virtual bool OnKeyEvent(int nKeyEvent, int nChar, int nRepCount, int nFlag) = 0;

// 命令更新。

// 设置工具条或菜单中响应的命令的更新信息，该函数由系统调用。

// szCommandID:命令id。

// pEnable:是否可用。

// pCheck:是否选中。

// pRadio:是否单选。

// szText:文字内容。

// return 如果已经处理命令返回true，否则返回false。

virtual bool UpdateUI(const char\* szCommandID, int\* pEnable, int\* pCheck, int\* pRadio, char\* szText) = 0;

// 订购/取消订购指定的消息。

// 该函数实现消息的“订购-发布”模式，系统主控会把消息发到订购了该消息的组件。

// szMessage: 消息ID。

// pIMessage: 消息订购者

virtual void SubscribeMessage(const char\* szMessage, IVBF\_Component\* pIComponent) = 0;

virtual void UnSubscribeMessage(const char\* szMessage, IVBF\_Component\* pIComponent) = 0;

// 发送消息。

// 该函数用于向其它组件发送消息，系统主控会把消息发到所有响应该消息的组件。

// szMessage:消息ID。

// nValue, pValue:消息参数，具体含义有消息本身决定。

virtual void SendMessage(void\* pSender,const char\* szMessage, int nValue, void\* pValue) = 0;

// 激活。

// 在多文档窗口中，每次切换当前文档时，都要把当前文档设置为活动状态。主控会向每个组件发送消息，

// 每个组件可以在此时做一些切换动作（如工具条上的对话框可以在此时设置与其关联的主控，一些非模态对话框可以在此时切换其关联的主控）。

virtual void SetActive(bool bActive) = 0;

virtual void SetAppPath(const char\* szPath)=0;

virtual char\* GetAppPath()=0;

virtual void SetSysSetPath(const char\* szPath)=0;

virtual char\* GetSysSetPath()=0;

};

##### 2）IVBF\_3DMainCtrl

三维主控接口。三维主控（IVBF\_3DMainCtrl）是主控管理下的一个组件，同时又是一个二级主控，负责三维组件的注册、命令分发、事件处理、绘图等任务。

class IVBF\_3DMainCtrl : public IVBF\_MainCtrl

{

public:

virtual ~IVBF\_3DMainCtrl() {}

// 获取主Viewer

virtual IVBF\_Viewer\* GetViewer()=0;

// 添加/获取3D视图指针

virtual bool AddView(IVBF\_View\* pView)=0;

virtual void RemoveView(IVBF\_View\* pView)=0;

virtual int GetNumViews()=0;

virtual IVBF\_View\* GetView(int nIndex)=0;

// 获取场景图根节点

virtual osg::IVBF\_SGNodeGroup\* GetSceneGraph()=0;

// 获取地球根节点

virtual osg::IVBF\_SGNodeGroup\* GetEarthNode()=0;

// 获取地图接口

virtual IVBF\_Map\* GetMainMap()=0;

// 开始执行主循环

virtual int Start()=0;

// 获取日志接口的指针，以便各个组件向其中写入日志信息 (如出错信息)

virtual IVBF\_Log\* GetLogPtr()=0;

};

##### 3）IVBF\_Message

消息接口。主控与各个组件之间的消息分发，通过函数IVBF\_MainCtrl::SendMessage()函数实现，只有订购了该消息的组件才能收到并响应该消息。

class IVBF\_Message

{

public:

virtual ~IVBF\_Message() {}

public:

// GetMessageCount:获取该组件相应的消息数。

// return 组件相应的消息数。

virtual int GetMessageCount() = 0;

// 获取指定消息ID，与次序无关。

// nIndex ：消息ID索引号，用于遍历所有消息。

// return 消息ID。

virtual const char\* GetMessageID(int nIndex) = 0;

// OnMessage:响应消息。

// szMessage ：消息ID。

// nValue，pValue ：消息的参数，具体含义有消息本身决定。

virtual void OnMessage(void\* pSender,const char\* szMessage, int nValue, void\* pValue) = 0;

// 订购消息

virtual void SubscribeMessage() = 0;

};

##### 4）IVBF\_Tool

工具接口。主控通过函数IVBF\_MainCtrl::SelectTool()函数分发工具，并调用每个组件的该接口来完成工具交互。每个交互工具，可通过鼠标键盘事件完成对应功能。

class IVBF\_Tool

{

public:

virtual ~IVBF\_Tool() {}

public:

// 指定的交互工具被选中。

// szToolID ：交互工具ID。

// pInParam ：输入参数，具体含义有交互工具本身决定。

// return 0表示不成功，其它值表示成功。

virtual int OnToolSelected(const char\* szToolID, IVBF\_PropertySet\* pInParam) = 0;

// 指定的交互工具被撤销选中。

// szToolID ：交互工具ID。

// pOutParam ：输出参数，具体含义有交互工具本身决定。

// return 0表示不成功，其它值表示成功。

virtual int OnToolUnSelected(const char\* szToolID, IVBF\_PropertySet\* pOutParam) = 0;

// 获取组件支持的交互工具数目。

virtual int GetToolCount() = 0;

// 获取组件所支持的指定交互工具ID，与次序无关。

// nIndex ：交互工具的索引号。

virtual const char\* GetToolID(int nIndex) = 0;

// 获取右键菜单的名称。

// 根据交互工具的不同右键菜单的名称可以不同，每个名称对应于一个资源文件。

// return 右键菜单的名称，如果返回""，表示没有右键菜单。

virtual const char\* GetContextMenuName() = 0;

};

##### 5）IVBF\_Command

命令接口。主控通过函数IVBF\_MainCtrl::ExecuteCommand()函数分发命令，并调用每个组件的该接口来完成命令处理。一个命令处理不需要鼠标键盘操作，即可完成一个单独的功能。

class IVBF\_Command

{

public:

virtual ~IVBF\_Command() {}

public:

// ExecuteCommand:执行指定命令。

// szCommandID:命令ID。

// pInParam: 输入参数，其含义有命令本身决定。

// pOutParam:输出参数，其含义有命令本身决定。

// return 如果执行成功返回true，否则返回false。

virtual bool ExecuteCommand(const char\* szCommandID, IVBF\_PropertySet\* pInParam, IVBF\_PropertySet\* pOutParam) = 0;

// GetCommandCount:获取组件支持的命令数目。

// return 组件支持的命令数目。

virtual int GetCommandCount() = 0;

// 获取组件所支持的指定命令ID，返回命令ID，与次序无关。

// nIndex ：命令ID索引号，用于遍历命令ID。

// return 命令ID。

virtual const char\* GetCommandID(int nIndex) = 0;

};

##### 6）IVBF\_UIProcessor

命令更新接口。当系统菜单和工具条命令需要更新的时候，系统主控调用每个组件的该接口来完成命令更新，在新建工具条的时候，调用该接口来完成工具条上对话框的创建。该接口所处理的ID应该包含IVBF\_Command、IVBF\_Tool接口中处理的所有ID。

class IVBF\_UIProcessor

{

public:

virtual ~IVBF\_UIProcessor() {}

public:

// 获取组件处理的命令更新数。

// return 命令更新数。

virtual int GetUIProcessorCount() = 0;

// 获取组件处理的命令更新的ID ，与次序无关。

// nIndex ：用于遍历ID。

// return 命令更新的ID。

virtual const char\* GetUIProcessorID(int nIndex) = 0;

// 命令更新。

// 设置工具条或菜单中响应的命令的更新信息，该函数有系统调用 。

// szCommandID:命令id。

// pEnable:是否可用。 pCheck:是否选中。 pRadio:是否单选。 szText:文字内容。

// return 如果已经处理命令返回true，否则返回false。

virtual bool UpdateUI(const char\* szCommandID, int\* piEnable, int\* piCheck, int\* piRadio, char\* szText) = 0;

};

##### 7）IVBF\_Observer

观察者接口。观察者模式中的观察者，可用于非标准组件接收主控发送的各类消息。

class IVBF\_Observer

{

public:

virtual ~IVBF\_Observer() {}

public:

// 消息响应函数

// 响应消息，进行实际处理

// szMessage：被发送的消息

// nValue：消息的整数参数

// pValue：消息的指针参数

virtual void OnEvent(const char\* szMessage, int nValue=0, void\* pValue=NULL) = 0;

};

例如，如果想要接收到三维标绘组件（IVBF\_3DPlot）发出的“模型被选中消息”（ VBF\_MESSAGE\_3DPLOT\_MARK\_SELECTED），可首先从接口IVBF\_Observer派生一个类CVBF\_Observer，并在OnEvent()函数中响应该消息，完整代码如下：

// VBF\_Observer.h

class CVBF\_Observer : public IVBF\_Observer

{

public:

CVBF\_Observer() {}

virtual ~CVBF\_Observer() {}

virtual void OnEvent(const char\* szMessage, int nValue=0, void\* pValue=NULL);

};

// VBF\_Observer.cpp

void CVBF\_Observer::OnEvent(const char\* szMessage, int nValue/\*=0\*/, void\* pValue/\*=NULL\*/)

{

// 模型被选中了

if( strcmp(szMessage, VBF\_MESSAGE\_3DPLOT\_MARK\_SELECTED)==0 && nValue>0 && pValue!=NULL )

{

std::vector< observer\_ptr<IVBF\_MarkModel> >\* pMarkList = (std::vector< observer\_ptr<IVBF\_MarkModel> >\*)pValue;

if(NULL==pMarkList) return;

for(int i=0; i<pMarkList->size(); i++)

{

observer\_ptr<IVBF\_MarkModel>& opMark = pMarkList->at(i);

if(!opMark.valid()) continue;

const std::string& strMarkID = opMark->GetID();

}

}

}

然后将该类的实例化对象绑定到三维主控即可，完整代码如下：

CVBF\_Observer observer;

g\_pI3DMainCtrl->Attach(&observer);

#### 5.3.2三维渲染引擎模块

渲染引擎由一个视景器（IVBF\_Viewer）构成，每个视景器又包含若干视图（IVBF\_View），每个视图对应一个主相机（IVBF\_Camera），对场景的操作就是通过各类相机操作器（IVBF\_CameraManipulator）完成的。

上述几个接口的定义如下：

##### 1）IVBF\_Viewer

视景器接口。用户可通过全局函数VBF\_CreateViewer(EVBF\_VIEWER\_TYPE type)创建单视图视景器视景器（参数type=EVBF\_VIEWER\_TYPE\_SIMPLE）或多视图视景器（参数type=EVBF\_VIEWER\_TYPE\_COMPOSITE），或者通过函数IVBF\_3DMainCtrl:: GetViewer()直接从三维主控中获取视景器。

class IVBF\_Viewer : virtual public osg::CVBF\_Object

{

public:

// IVBF\_Viewer(){}

// IVBF\_Viewer(const IVBF\_Viewer& cs,const osg::CopyOp& copyop=osg::CopyOp::SHALLOW\_COPY):CVBF\_Object(cs,copyop) {}

// inline explicit IVBF\_Viewer(bool threadSafeRefUnref):CVBF\_Object(threadSafeRefUnref){}

// 0、配置与初始化

virtual bool readConfiguration(const std::string& filename) = 0; // 从配置文件读取数据配置视景器

virtual bool IsRealized() const = 0;// 是否有窗口

virtual void Realize() = 0;// 设置窗口和相关线程

// 1、与视图相关的内容

virtual void addView(IVBF\_View\* view)=0; // 添加视图

virtual void removeView(IVBF\_View\* view)=0; // 移除视图

virtual IVBF\_View\* getView(unsigned i=0) =0; // 获取视图

virtual const IVBF\_View\* getView(unsigned i=0) const =0;

virtual unsigned int getNumViews() const =0; // 获取视图数量

// 2、与仿真循环相关的内容

virtual int run()=0; // 运行

virtual void SetDone(bool done)=0; // 设置完成标志，以通知视景器的工作完成并将退出帧循环（frame loop）

virtual bool IsDone() const = 0; // 是否完成并将退出出帧循环

virtual bool IsPaused() = 0;

virtual void SetPaused(bool bPaused) = 0;

virtual void frame(double simulationTime=USE\_REFERENCE\_TIME)=0; // 运行一帧

virtual void advance(double simulationTime=USE\_REFERENCE\_TIME) = 0; // 仿真时钟向前一帧

virtual void eventTraversal() = 0; // 事件响应

virtual void updateTraversal() = 0; // 更新

virtual void renderingTraversals() =0; // 渲染

virtual bool checkNeedToDoFrame() = 0;// 检测是否需要运行下一帧，与按需求运行ON\_DEMAND相关

virtual void setRunMaxFrameRate(double frameRate)=0;// 设置最大帧速率

virtual double getRunMaxFrameRate() const=0; // 获取最大帧速率

enum FrameScheme // 帧运行方案

{

ON\_DEMAND, // 按需求运行

CONTINUOUS // 连续运行

};

// 设置/获取帧运行方案

virtual void setRunFrameScheme(FrameScheme fs)=0;

virtual FrameScheme getRunFrameScheme() const=0;

// 3、线程相关的接口

virtual void SetThreadingModel(EVBF\_ThreadingModel threadingModel) =0; // 设置线程模型

virtual EVBF\_ThreadingModel GetThreadingModel() const =0; // 获取线程模型

virtual bool areThreadsRunning() const =0; // 线程是否在运行

virtual void StopThreading() =0; // 停止视景器的所有线程

virtual void StartThreading() =0; // 启动视景器的所有线程

/\*\*设置/获取增量编译操作，可避免加载大（多）模型时造成的帧率下降 \*/

virtual void setIncrementalCompileOperation(osgUtil::IncrementalCompileOperation\* ico)=0;

virtual osgUtil::IncrementalCompileOperation\* getIncrementalCompileOperation()=0;

// 4、回调函数

virtual void addUpdateOperation(osg::Operation\* operation)=0; // 添加1个更新处理器

virtual void removeUpdateOperation(osg::Operation\* operation)=0;// 移除1个更新处理器

virtual void setUpdateOperations(osg::OperationQueue\* operations)=0;// 设置多组更新处理器

virtual osg::OperationQueue\* getUpdateOperations()=0; // 获取多组更新处理器

virtual const osg::OperationQueue\* getUpdateOperations() const=0;

// 5、获取视图与场景数据

virtual IVBF\_View\* getViewWithFocus() = 0;// 获取焦点视图

virtual const IVBF\_View\* getViewWithFocus() const = 0;

virtual void GetViews(CVBF\_Views& views, bool onlyValid=true) = 0; // 获取所有的视图

virtual void GetWindows(CVBF\_Windows& windows, bool onlyValid=true)=0;// 获取所有的窗口

virtual void GetContexts(CVBF\_Contexts& contexts, bool onlyValid=true) = 0;// 获取所有的GC

virtual CVBF\_FrameStamp\* getViewerFrameStamp() = 0; // 获取帧戳

// 设置结束栅栏的位置

virtual void SetEndBarrierPosition(EVBF\_EndBarrierPosition bp)=0;

virtual EVBF\_EndBarrierPosition GetEndBarrierPosition() const =0;

// 获取鼠标和键盘的用法

virtual void GetUsage(ApplicationUsage& usage) const = 0;

virtual bool IsFirstFrame() = 0;

// 6、设置/获取统计对象，用于统计帧相关的时间和场景图数据

virtual void setViewerStats(CVBF\_Stats\* stats) = 0;

virtual CVBF\_Stats\* getViewerStats() = 0;

virtual const CVBF\_Stats\* getViewerStats() const = 0;

// 设置/获取场景预处理的自定义操作（场景预处理器）

virtual void setRealizeOperation(osg::Operation\* op) = 0;

virtual osg::Operation\* getRealizeOperation() = 0;

// 设置/获取自定义的退出键。默认键是Escape (osgGA::GUIEVentAdapter::KEY\_Escape)。 设置0则关掉此特性

virtual void setKeyEventSetsDone(int key)=0;

virtual int getKeyEventSetsDone() const=0;

// 设置/获取是否允许按下某个键（QUIT\_APPLICATION）之后直接退出这种做法

virtual void setQuitEventSetsDone(bool flag)=0;

virtual bool getQuitEventSetsDone() const=0;

};

##### 2）IVBF\_View

视图接口。用户可通过全局函数VBF\_CreateView()创建视图，并通过IVBF\_Viewer::AddView()函数添加给指定的视景器。

class IVBF\_View : public osg::IVBF\_ViewBrief, public ::IVBF\_GUIActionAdapter

{

public:

virtual IVBF\_Viewer\* GetViewer() = 0;

// 设置/获取统计对象，用于统计帧相关的时间和场景图数据

virtual void setStats(CVBF\_Stats\* stats)=0;

virtual CVBF\_Stats\* getStats()=0;

virtual const CVBF\_Stats\* getStats() const=0;

// 设置/获取开始时钟（Tick）

virtual void setStartTick(CVBF\_Timer\_t tick)=0;

virtual CVBF\_Timer\_t getStartTick() const=0;

// 设置/获取场景数据

virtual void setSceneData(osg::IVBF\_SGNode\* node) = 0;

virtual osg::IVBF\_SGNode\* getSceneData() = 0;

virtual const osg::IVBF\_SGNode\* getSceneData() const = 0;

// 设置/获取个分页数据库，用于数据的动态调度

virtual void setDatabasePager(osgDB::DatabasePager\* dp) = 0;

virtual osgDB::DatabasePager\* getDatabasePager() = 0;

virtual const osgDB::DatabasePager\* getDatabasePager() const = 0;

typedef std::list< ref\_ptr<osgGA::GUIEventHandler> > EventHandlers;

// 添加/移除/获取事件处理器

virtual void addEventHandler(osgGA::GUIEventHandler\* eventHandler) = 0;

virtual void addEventHandlerToHead(osgGA::GUIEventHandler\* eventHandler) = 0;

virtual void removeEventHandler(osgGA::GUIEventHandler\* eventHandler) = 0;

virtual EventHandlers& getEventHandlers() = 0;

virtual const EventHandlers& getEventHandlers() const = 0;

/\*\* 把CVBF\_SGNodePath设置到场景中任何一个激活的CoordinateSystemNode path。

CoordinateSystemNode path用于帮助应用程序和相机操作器处理 地心坐标系统 \*/

virtual void setCoordinateSystemNodePath(const osg::CVBF\_SGNodePath& nodePath)=0;

virtual osg::CVBF\_SGNodePath getCoordinateSystemNodePath() const =0;

// 设置/获取EventQueue，用于和 外部非窗口相关的事件 统一

virtual void setEventQueue(osgGA::CVBF\_EventQueue\* eventQueue)=0;

virtual osgGA::CVBF\_EventQueue\* getEventQueue()=0;

virtual const osgGA::CVBF\_EventQueue\* getEventQueue() const=0;

// 设置/获取相机操作器。该操作器根据事件（鼠标键盘等）控制主相机的位置。 resetPosition：是否把操作器设为home position

virtual void setCameraManipulator(IVBF\_CameraManipulator\* manipulator, bool resetPosition = true)=0;

virtual IVBF\_CameraManipulator\* getCameraManipulator()=0;

virtual const IVBF\_CameraManipulator\* getCameraManipulator() const=0;

// 执行操作器的home函数

virtual void home()=0;

// 根据显示设备自动创建相机

virtual void setUpViewAcrossAllScreens()=0;// 在所有的屏幕上创建子相机和相关的GraphicsWindows

virtual void setUpViewInWindow(int x, int y, int width, int height, unsigned int screenNum=0)=0;// 在单窗口上创建一个单相机

virtual void setUpViewOnSingleScreen(unsigned int screenNum=0)=0;// 创建一个全屏幕的单相机

// 用于半球面显示器的工作，使用6个子相机渲染cube map的6个面，第7个相机进行变形校正

virtual void setUpViewFor3DSphericalDisplay(double radius=1.0, double collar=0.45, unsigned int screenNum=0, osg::Image\* intensityMap=0, const osg::Matrixd& projectorMatrix = osg::Matrixd())=0;

// 在球形显示器上显示主场景to as panoramic 2:1 texture and then doing distortion correction to present onto a spherical display.

virtual void setUpViewForPanoramicSphericalDisplay(double radius=1.0, double collar=0.45, unsigned int screenNum=0, osg::Image\* intensityMap=0, const osg::Matrixd& projectorMatrix = osg::Matrixd())=0;

// 设置裸眼立体显示的Philips WoWvx显示器

virtual void setUpViewForWoWVxDisplay(unsigned int screenNum, unsigned char wow\_content, unsigned char wow\_factor, unsigned char wow\_offset, float wow\_disparity\_Zd, float wow\_disparity\_vz, float wow\_disparity\_M, float wow\_disparity\_C)=0;

virtual bool containsCamera(const osg::IVBF\_Camera\* camera) const=0;// 判断是否包含给定的相机

virtual const osg::IVBF\_Camera\* getCameraContainingPosition(float x, float y, float& local\_x, float& local\_y) const=0;// 获取包含鼠标位置x,y（在主相机窗口中的坐标）的相机，并返回在该相机对应的局部窗口中的位置。

virtual bool computeIntersections(float x,float y, osgUtil::LineSegmentIntersector::Intersections& intersections,osg::IVBF\_SGNode::NodeMask traversalMask = 0xffffffff)=0;

virtual bool computeIntersections(float x,float y, const osg::CVBF\_SGNodePath& nodePath, osgUtil::LineSegmentIntersector::Intersections& intersections,osg::IVBF\_SGNode::NodeMask traversalMask = 0xffffffff)=0;

virtual void computeActiveCoordinateSystemNodePath()=0;

// 把相机按深度分为多个子相机

virtual bool setUpDepthPartitionForCamera(osg::IVBF\_Camera\* cameraToPartition, CVBF\_DepthPartitionSettings\* dps=0)=0;

// 把激活的（active）相机按深度分为多个子相机

virtual bool setUpDepthPartition(CVBF\_DepthPartitionSettings\* dsp=0)=0;

// 设置/获取该视图的显示设置

virtual void setDisplaySettings(osg::DisplaySettings\* ds)=0;

virtual osg::DisplaySettings\* getDisplaySettings()=0;

virtual const osg::DisplaySettings\* getDisplaySettings() const=0;

// 设置/获取空间融合距离模式和值，仅用于双目立体

virtual void setFusionDistance(osgUtil::SceneView::FusionDistanceMode mode,float value=1.0f)=0;

virtual osgUtil::SceneView::FusionDistanceMode getFusionDistanceMode() const=0;

virtual float getFusionDistanceValue() const=0;

// 把场景的数据设置给相机

virtual void assignSceneDataToCameras()=0;

virtual osg::IVBF\_SGNodeGroup\* getNeedIntersectData() = 0;

};

##### 3）IVBF\_Camera

相机接口。用户可通过全局函数VBF\_CreateCamera ()创建相机，或者通过IVBF\_View::getCamera()函数获取主相机。

class IVBF\_Camera : public IVBF\_SGNodeGroupTransform, public CullSettings

{

public:

IVBF\_Camera(){}

IVBF\_Camera(const IVBF\_Camera& cs,const CopyOp& copyop=CopyOp::SHALLOW\_COPY):IVBF\_SGNodeGroupTransform(cs,copyop),CullSettings(cs) {}

explicit IVBF\_Camera(bool threadSafeRefUnref):IVBF\_SGNodeGroupTransform(threadSafeRefUnref){}

// 设置/获取相机所属的视图

virtual void setView(IVBF\_ViewBrief\* view) =0;

virtual IVBF\_ViewBrief\* getView() =0;

virtual const IVBF\_ViewBrief\* getView() const =0;

// 设置/获取统计对象，用于统计帧相关的时间和场景图数据

virtual void setStats(::CVBF\_Stats\* stats) =0;

virtual ::CVBF\_Stats\* getStats() =0;

virtual const ::CVBF\_Stats\* getStats() const =0;

/\*\* 设置/获取是否允许接收事件 \*/

virtual void setAllowEventFocus(bool focus) =0;

virtual bool getAllowEventFocus() const =0;

/\*\* 设置/获取显示设置DisplaySettings\*/

virtual void setDisplaySettings(osg::DisplaySettings\* ds) =0;

virtual osg::DisplaySettings\* getDisplaySettings() =0;

virtual const osg::DisplaySettings\* getDisplaySettings() const =0;

/\*\* 设置/获取glClear()中使用的clear mask，默认为GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT \*/

virtual void setClearMask(GLbitfield mask) =0;

virtual GLbitfield getClearMask() const =0;

/\*\* Set the clear color used in glClearColor().

\* glClearColor is only called if mask & GL\_COLOR\_BUFFER\_BIT is true\*/

virtual void setClearColor(const osg::Vec4& color) =0;

virtual const osg::Vec4& getClearColor() const =0;

/\*\* Set the clear accum used in glClearAccum().

\* glClearAcumm is only called if mask & GL\_ACCUM\_BUFFER\_BIT is true. \*/

virtual void setClearAccum(const osg::Vec4& color) =0;

virtual const osg::Vec4& getClearAccum() const =0;

/\*\* Set the clear depth used in glClearDepth(). Defaults to 1.0

\* glClearDepth is only called if mask & GL\_DEPTH\_BUFFER\_BIT is true. \*/

virtual void setClearDepth(double depth) =0;

virtual double getClearDepth() const =0;

/\*\* Set the clear stencil value used in glClearStencil(). Defaults to 0;

\* glClearStencil is only called if mask & GL\_STENCIL\_BUFFER\_BIT is true\*/

virtual void setClearStencil(int stencil) =0;

virtual int getClearStencil() const =0;

/\*\* Set the color mask of the camera to use specified osg::ColorMask. \*/

virtual void setColorMask(osg::ColorMask\* colorMask) =0;

virtual void setColorMask(bool red, bool green, bool blue, bool alpha) =0;

virtual const ColorMask\* getColorMask() const =0;

virtual ColorMask\* getColorMask() =0;

/\*\* Set the viewport of the camera to use specified osg::Viewport. \*/

virtual void setViewport(osg::Viewport\* viewport) =0;

virtual void setViewport(int x,int y,int width,int height) =0;

virtual const Viewport\* getViewport() const =0;

virtual Viewport\* getViewport() =0;

enum TransformOrder

{

PRE\_MULTIPLY,

POST\_MULTIPLY

};

/\*\* Set the transformation order for world-to-local and local-to-world transformation.\*/

virtual void setTransformOrder(TransformOrder order) =0;

virtual TransformOrder getTransformOrder() const =0;

enum ProjectionResizePolicy

{

FIXED, /\*\*< Keep the projection matrix fixed, despite window resizes.\*/

HORIZONTAL, /\*\*< Adjust the HORIZONTAL field of view on window resizes.\*/

VERTICAL /\*\*< Adjust the VERTICAL field of view on window resizes.\*/

};

/\*\* Set the policy used to determine if and how the projection matrix should be adjusted on window resizes. \*/

virtual void setProjectionResizePolicy(ProjectionResizePolicy policy) =0;

/\*\* Get the policy used to determine if and how the projection matrix should be adjusted on window resizes. \*/

virtual ProjectionResizePolicy getProjectionResizePolicy() const =0;

/\*\* Set the projection matrix. Can be thought of as setting the lens of a camera. \*/

virtual void setProjectionMatrix(const osg::Matrixf& matrix) =0;

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

/\*\* Set to an orthographic projection. See OpenGL glOrtho for documentation further details.\*/

virtual void setProjectionMatrixAsOrtho(double left, double right, double bottom, double top, double zNear, double zFar) =0;

/\*\* Set to a 2D orthographic projection. See OpenGL glOrtho2D documentation for further details.\*/

virtual void setProjectionMatrixAsOrtho2D(double left, double right, double bottom, double top) =0;

/\*\* Set to a perspective projection. See OpenGL glFrustum documentation for further details.\*/

virtual void setProjectionMatrixAsFrustum(double left, double right, double bottom, double top, double zNear, double zFar) =0;

/\*\* Create a symmetrical perspective projection, See OpenGL gluPerspective documentation for further details.

\* Aspect ratio is defined as width/height.\*/

virtual void setProjectionMatrixAsPerspective(double fovy,double aspectRatio, double zNear, double zFar) =0;

/\*\* Get the projection matrix.\*/

virtual osg::Matrixd& getProjectionMatrix() =0;

virtual const osg::Matrixd& getProjectionMatrix() const =0;

/\*\* Get the orthographic settings of the orthographic projection matrix.

\* Returns false if matrix is not an orthographic matrix, where parameter values are undefined.\*/

virtual bool getProjectionMatrixAsOrtho(double& left, double& right, double& bottom, double& top, double& zNear, double& zFar) const =0;

/\*\* Get the frustum setting of a perspective projection matrix.

\* Returns false if matrix is not a perspective matrix, where parameter values are undefined.\*/

virtual bool getProjectionMatrixAsFrustum(double& left, double& right, double& bottom, double& top, double& zNear, double& zFar) const =0;

/\*\* Get the frustum setting of a symmetric perspective projection matrix.

\* Returns false if matrix is not a perspective matrix, where parameter values are undefined.

\* Note, if matrix is not a symmetric perspective matrix then the shear will be lost.

\* Asymmetric matrices occur when stereo, power walls, caves and reality center display are used.

\* In these configurations one should use the 'getProjectionMatrixAsFrustum' method instead.\*/

virtual bool getProjectionMatrixAsPerspective(double& fovy,double& aspectRatio, double& zNear, double& zFar) const =0;

/\*\* Set the view matrix. Can be thought of as setting the position of the world relative to the camera in camera coordinates. \*/

virtual void setViewMatrix(const osg::Matrixf& matrix) =0;

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

virtual osg::Matrixd& getViewMatrix() =0;

virtual const osg::Matrixd& getViewMatrix() const =0;

/\*\* Set to the position and orientation of view matrix, using the same convention as gluLookAt. \*/

virtual void setViewMatrixAsLookAt(const osg::Vec3d& eye,const osg::Vec3d& center,const osg::Vec3d& up)=0;

virtual void getViewMatrixAsLookAt(osg::Vec3d& eye,osg::Vec3d& center,osg::Vec3d& up,double lookDistance=1.0) const=0;

virtual void getViewMatrixAsLookAt(osg::Vec3f& eye,osg::Vec3f& center,osg::Vec3f& up,float lookDistance=1.0f) const=0;

virtual Matrixd getInverseViewMatrix() const=0;

enum RenderOrder

{

PRE\_RENDER,

NESTED\_RENDER,

POST\_RENDER

};

/\*\* 设置/获取相机子图的渲染顺序，relative to any camera that this subgraph is nested within.

\* 渲染到纹理选择PRE\_RENDER，渲染"屏显"选择POST\_RENDER.\*/

virtual void setRenderOrder(RenderOrder order, int orderNum = 0)=0;

virtual RenderOrder getRenderOrder() const=0;

/\*\* Get the rendering order number of this camera relative to any sibling cameras in this subgraph.\*/

virtual int getRenderOrderNum() const=0;

/\*\* Return true if this Camera is set up as a render to texture camera, i.e. it has textures assigned to it.\*/

virtual bool isRenderToTextureCamera() const=0;

enum RenderTargetImplementation

{

FRAME\_BUFFER\_OBJECT, // 帧缓存对象（Frame Buffer Object），可以用来实现离屏渲染，渲染结果不会体现在图形窗口中

PIXEL\_BUFFER\_RTT, // 纹理烘焙（Render To Texture，RTT）

PIXEL\_BUFFER, // 像素缓存（Pixel Buffer）

FRAME\_BUFFER, // 帧缓存（Frame Buffer），可以适用于较广泛的硬件平台上

SEPERATE\_WINDOW

};

// 设置/获取渲染目标实现方式

virtual void setRenderTargetImplementation(RenderTargetImplementation impl)=0;

virtual RenderTargetImplementation getRenderTargetImplementation() const=0;

// 设置渲染目标实现方式，and fall-back that's used if the former isn't available

virtual void setRenderTargetImplementation(RenderTargetImplementation impl, RenderTargetImplementation fallback)=0;

/\*\* Get the render target fallback.\*/

virtual RenderTargetImplementation getRenderTargetFallback() const=0;

// 设置/获取draw buffer，在每帧绘制开始时采用。GL\_NONE表示选择最合适的buffer

virtual void setDrawBuffer(GLenum buffer)=0;

virtual GLenum getDrawBuffer() const=0;

// 设置/获取read buffer，在每帧绘制开始时采用。GL\_NONE表示选择最合适的buffer

virtual void setReadBuffer(GLenum buffer)=0;

virtual GLenum getReadBuffer() const =0;

enum BufferComponent

{

DEPTH\_BUFFER, // 深度缓存（DEPTH\_BUFFER）

STENCIL\_BUFFER,

PACKED\_DEPTH\_STENCIL\_BUFFER,

COLOR\_BUFFER, // 颜色缓存（COLOR\_BUFFER）

COLOR\_BUFFER0, // OpenGL为多重渲染目标（MRT）的支持提供了多达十六个颜色缓存，0-15

COLOR\_BUFFER1 = COLOR\_BUFFER0+1,

COLOR\_BUFFER2 = COLOR\_BUFFER0+2,

COLOR\_BUFFER3 = COLOR\_BUFFER0+3,

COLOR\_BUFFER4 = COLOR\_BUFFER0+4,

COLOR\_BUFFER5 = COLOR\_BUFFER0+5,

COLOR\_BUFFER6 = COLOR\_BUFFER0+6,

COLOR\_BUFFER7 = COLOR\_BUFFER0+7,

COLOR\_BUFFER8 = COLOR\_BUFFER0+8,

COLOR\_BUFFER9 = COLOR\_BUFFER0+9,

COLOR\_BUFFER10 = COLOR\_BUFFER0+10,

COLOR\_BUFFER11 = COLOR\_BUFFER0+11,

COLOR\_BUFFER12 = COLOR\_BUFFER0+12,

COLOR\_BUFFER13 = COLOR\_BUFFER0+13,

COLOR\_BUFFER14 = COLOR\_BUFFER0+14,

COLOR\_BUFFER15 = COLOR\_BUFFER0+15

};

/\*需要特别注意的是：如果希望使用 FBO来实现纹理烘焙或者场景截图的话，不可以将场景主摄像机的setRenderTargetImplementation直接设置为相应的枚举量，那样将无法正常地看到场景（因为主摄像机对应的渲染台已经将场景绘制的结果绑定到FBO 上了）。正确的作法是在场景树中增加一个Camera节点，设置“渲染目标实现方式”为FBO 方式；并通过Camera::setRenderOrder 设定它的渲染顺序，设置为PRE\_RENDER 可以保证这个摄像机在主场景之前执行绘制（它创建了一个“前序渲染台”，存入RenderStage::\_preRenderList列表），从而实现“渲染到纹理”的效果。\*/

/\*\* Attach a buffer with specified OpenGL internal format.\*/

virtual void attach(BufferComponent buffer, GLenum internalFormat)=0;

/\*\* Attach a Texture to specified buffer component.

\* The level parameter controls the mip map level of the texture that is attached.

\* The face parameter controls the face of texture cube map or z level of 3d texture.

\* The mipMapGeneration flag controls whether mipmap generation should be done for texture.\*/

virtual void attach(BufferComponent buffer, osg::Texture\* texture, unsigned int level = 0, unsigned int face=0, bool mipMapGeneration=false,

unsigned int multisampleSamples = 0, unsigned int multisampleColorSamples = 0)=0;

/\*\* Attach a Image to specified buffer component.\*/

virtual void attach(BufferComponent buffer, osg::Image\* image, unsigned int multisampleSamples = 0, unsigned int multisampleColorSamples = 0)=0;

/\*\* Detach specified buffer component.\*/

virtual void detach(BufferComponent buffer)=0;

struct Attachment

{

Attachment():

\_internalFormat(GL\_NONE),

\_level(0),

\_face(0),

\_mipMapGeneration(false),

\_multisampleSamples(0),

\_multisampleColorSamples(0) {}

int width() const

{

if (\_texture.valid()) return \_texture->getTextureWidth();

if (\_image.valid()) return \_image->s();

return 0;

};

int height() const

{

if (\_texture.valid()) return \_texture->getTextureHeight();

if (\_image.valid()) return \_image->t();

return 0;

};

int depth() const

{

if (\_texture.valid()) return \_texture->getTextureDepth();

if (\_image.valid()) return \_image->r();

return 0;

};

GLenum \_internalFormat;

ref\_ptr<Image> \_image;

ref\_ptr<Texture> \_texture;

unsigned int \_level;

unsigned int \_face;

bool \_mipMapGeneration;

unsigned int \_multisampleSamples;

unsigned int \_multisampleColorSamples;

};

typedef std::map< BufferComponent, Attachment> BufferAttachmentMap;

/\*\* Get the BufferAttachmentMap, used to configure frame buffer objects, pbuffers and texture reads.\*/

virtual BufferAttachmentMap& getBufferAttachmentMap() =0;

virtual const BufferAttachmentMap& getBufferAttachmentMap() const =0;

enum ImplicitBufferAttachment

{

IMPLICIT\_DEPTH\_BUFFER\_ATTACHMENT = DisplaySettings::IMPLICIT\_DEPTH\_BUFFER\_ATTACHMENT,

IMPLICIT\_STENCIL\_BUFFER\_ATTACHMENT = DisplaySettings::IMPLICIT\_STENCIL\_BUFFER\_ATTACHMENT,

IMPLICIT\_COLOR\_BUFFER\_ATTACHMENT = DisplaySettings::IMPLICIT\_COLOR\_BUFFER\_ATTACHMENT,

USE\_DISPLAY\_SETTINGS\_MASK = (~0)

};

typedef int ImplicitBufferAttachmentMask;

virtual void setImplicitBufferAttachmentMask(ImplicitBufferAttachmentMask renderMask = DisplaySettings::DEFAULT\_IMPLICIT\_BUFFER\_ATTACHMENT, ImplicitBufferAttachmentMask resolveMask = DisplaySettings::DEFAULT\_IMPLICIT\_BUFFER\_ATTACHMENT)=0;

virtual void setImplicitBufferAttachmentRenderMask(ImplicitBufferAttachmentMask implicitBufferAttachmentRenderMask)=0;

virtual void setImplicitBufferAttachmentResolveMask(ImplicitBufferAttachmentMask implicitBufferAttachmentResolveMask)=0;

virtual ImplicitBufferAttachmentMask getImplicitBufferAttachmentRenderMask(bool effectiveMask = false) const=0;

virtual ImplicitBufferAttachmentMask getImplicitBufferAttachmentResolveMask(bool effectiveMask = false) const=0;

/\*\* Create a operation thread for this camera.\*/

virtual void createCameraThread()=0;

/\*\* Assign a operation thread to the camera.\*/

virtual void setCameraThread(OperationThread\* gt)=0;

virtual OperationThread\* getCameraThread()=0;

virtual const OperationThread\* getCameraThread() const =0;

/\*\* Set the GraphicsContext that provides the mechansim for managing the OpenGL graphics context associated with this camera.\*/

virtual void setGraphicsContext(GraphicsContext\* context)=0;

virtual GraphicsContext\* getGraphicsContext()=0;

virtual const GraphicsContext\* getGraphicsContext() const=0;

/\*\* Set the Rendering object that is used to implement rendering of the subgraph.\*/

virtual void setRenderer(osg::GraphicsOperation\* rc)=0;

virtual osg::GraphicsOperation\* getRenderer()=0;

virtual const osg::GraphicsOperation\* getRenderer() const=0;

/\*\* Set the Rendering cache that is used for cached objects associated with rendering of subgraphs.\*/

virtual void setRenderingCache(osg::CVBF\_Object\* rc)=0;

virtual osg::CVBF\_Object\* getRenderingCache()=0;

virtual const osg::CVBF\_Object\* getRenderingCache() const =0;

/\*\* Draw callback for custom operations.\*/

struct OSG\_EXPORT DrawCallback : virtual public CVBF\_Object

{

DrawCallback() {}

DrawCallback(const DrawCallback&,const CopyOp&) {}

META\_Object(osg, DrawCallback);

/\*\* Functor method called by rendering thread. Users will typically override this method to carry tasks such as screen capture.\*/

virtual void operator () (osg::RenderInfo& renderInfo) const;

/\*\* Functor method, provided for backwards compatibility, called by operator() (osg::RenderInfo& renderInfo) method.\*/

virtual void operator () (const osg::IVBF\_Camera& /\*camera\*/) const {}

};

/\*\* 设置/获取自定义的绘图初始化（initial draw），用于绘制相机的子树及每一个渲染台之前进行初始化。\*/

virtual void setInitialDrawCallback(DrawCallback\* cb) =0;

virtual DrawCallback\* getInitialDrawCallback() =0;

virtual const DrawCallback\* getInitialDrawCallback() const =0;

// 设置渲染前回调，以便在渲染场景之前（此时需要的渲染台已经准备好）进行自定义的操作

virtual void setPreDrawCallback(DrawCallback\* cb) =0;

virtual DrawCallback\* getPreDrawCallback() =0;

virtual const DrawCallback\* getPreDrawCallback() const =0;

// 设置渲染后回调，以便在绘制场景后（此时需要的渲染台已经准备好）进行自定义的操作

virtual void setPostDrawCallback(DrawCallback\* cb) =0;

virtual DrawCallback\* getPostDrawCallback() =0;

virtual const DrawCallback\* getPostDrawCallback() const =0;

/\*\* Set the final draw callback for custom operations to be done after the drawing of the camera's subgraph and all of the post render stages has been completed.\*/

virtual void setFinalDrawCallback(DrawCallback\* cb) =0;

virtual DrawCallback\* getFinalDrawCallback() =0;

virtual const DrawCallback\* getFinalDrawCallback() const =0;

virtual OpenThreads::Mutex\* getDataChangeMutex() const =0;

};

##### 4）IVBF\_CameraManipulator

相机操作器接口。完成对三维场景的操控，用户可根据需要从该接口派生出各类操作器，并通过函数IVBF\_View::setCameraManipulator()设置给对应的视图。

class IVBF\_CameraManipulator : virtual public osgGA::GUIEventHandler

{

public:

virtual ~IVBF\_CameraManipulator() {}

// 1、坐标系相关接口

// 设置/获取"坐标系回调"，该回调告诉操作器上、东、北方向

virtual void setCoordinateFrameCallback(IVBF\_CoordinateFrameCallback\* cb)=0;

virtual IVBF\_CoordinateFrameCallback\* getCoordinateFrameCallback()=0;

virtual const IVBF\_CoordinateFrameCallback\* getCoordinateFrameCallback() const=0;

virtual void SetCallback(IVBF\_ManipulatorCB\* cb)=0;

// 获取坐标系

virtual osg::CoordinateFrame getCoordinateFrame(const osg::Vec3d& position) const =0;

// 获取向上、向前、向侧方向的向量

virtual osg::Vec3d getSideVector(const osg::CoordinateFrame& cf) const =0;

virtual osg::Vec3d getFrontVector(const osg::CoordinateFrame& cf) const =0;

virtual osg::Vec3d getUpVector(const osg::CoordinateFrame& cf) const =0;

// 2、矩阵相关接口

virtual void setByMatrix(const osg::Matrixd& matrix) = 0; // 通过矩阵设置操作器

virtual void setByInverseMatrix(const osg::Matrixd& matrix) = 0;// 通过矩阵的逆设置操作器

virtual osg::Matrixd getMatrix() const = 0; // 获取操作器的矩阵

virtual osg::Matrixd getInverseMatrix() const = 0; // 获取操作器矩阵的逆,典型的使用方式是作为model view matrix

// 3、空间融合距离，用于双目立体

virtual osgUtil::SceneView::FusionDistanceMode getFusionDistanceMode() const= 0;

virtual float getFusionDistanceValue() const= 0;

// 4、设置/获取求交遍历的掩码

virtual void setIntersectTraversalMask(unsigned int mask)= 0;

virtual unsigned int getIntersectTraversalMask() const= 0;

// 5、与节点相关的函数（有些操作器需要（如跟踪），有些不需要）

virtual void setNode(osg::IVBF\_SGNode\*)= 0;

virtual const osg::IVBF\_SGNode\* getNode() const= 0;

virtual osg::IVBF\_SGNode\* getNode()= 0;

// 6、与默认位置相关的函数

// 人工设置操作器的默认位置

virtual void setHomePosition(const osg::Vec3d& eye, const osg::Vec3d& center, const osg::Vec3d& up, bool autoComputeHomePosition=false)= 0;

virtual void getHomePosition(osg::Vec3d& eye, osg::Vec3d& center, osg::Vec3d& up) const= 0;

virtual void setAutoComputeHomePosition(bool flag)= 0; // 设置是否自动计算操作器的默认位置

virtual bool getAutoComputeHomePosition() const= 0; // 获取是否自动计算操作器的默认位置

virtual void computeHomePosition(const osg::IVBF\_Camera\* camera = NULL, bool useBoundingBox = false)=0;// 计算默认位置

virtual void home(const osgGA::GUIEventAdapter&, ::IVBF\_GUIActionAdapter&)=0; // 把相机移到默认位置

virtual void home(double /\*currentTime\*/)=0; // 把相机移到默认位置

virtual void init(const osgGA::GUIEventAdapter& ,::IVBF\_GUIActionAdapter&)=0; // 重置（初始化）

virtual bool handle(const osgGA::GUIEventAdapter& ea,::IVBF\_GUIActionAdapter& us)=0;// 处理事件

};

#### 5.3.3三维地图模块

整个三维场景由一个地图接口（IVBF\_Map）构成，它包含若干图层（IVBF\_MapLayer）。图层的类型包括：高程图层（CVBF\_LayerTerrainElevation）、影像图层（CVBF\_LayerTerrainImage）、模型图层（CVBF\_LayerModel）和掩码图层（CVBF\_LayerMask）。

上述几个接口的定义如下：

##### 1）IVBF\_Map

地图接口。

class IVBF\_Map : public ::CVBF\_Referenced

{

public:

IVBF\_Map():CVBF\_Referenced ( true ) { }

virtual ~IVBF\_Map() {}

public:

virtual int getNumImageLayers() const=0;// 获取影像图层的数量

virtual IVBF\_MapLayer\* getImageLayerByName( const std::string& name ) const=0;// 通过名字获取影像图层

virtual IVBF\_MapLayer\* getImageLayerByUID( int layerUID ) const=0;// 通过ID获取影像图层

virtual IVBF\_MapLayer\* getImageLayerAt( int index ) const=0;// 通过索引获取影像图层

virtual int getNumModelLayers() const=0;// 通过索引获取模型图层

virtual IVBF\_MapLayer\* getModelLayerByName( const std::string& name ) const=0;// 通过名字获取模型图层

virtual IVBF\_MapLayer\* getModelLayerByUID( int layerUID ) const=0;// 通过ID获取模型图层

virtual IVBF\_MapLayer\* getModelLayerAt( int index ) const=0;// 通过索引获取模型图层

virtual int getNumElevationLayers() const=0;// 通过索引获取高程图层

virtual IVBF\_MapLayer\* getElevationLayerByName( const std::string& name ) const=0;// 通过名字获取高程图层

virtual IVBF\_MapLayer\* getElevationLayerByUID( int layerUID ) const=0;// 通过ID获取高程图层

virtual IVBF\_MapLayer\* getElevationLayerAt( int index ) const=0;// 通过索引获取高程图层

virtual bool AddModelLayer(const std::string& strFileName)=0;// 从配置文件添加模型图层（与earth一致）

virtual void RemoveModelLayer(const std::string& name)=0;

};

##### 2）CVBF\_Map

地图图层接口的实现类，可通过查询接口的方法获取到：

CVBF\_Map\* pMap = (CVBF\_Map\*) pIMainCtrl->QueryInterface(VBF\_INTERFACE\_3D\_MAIN\_MAP);

该类包含各种图层和渲染选项，每个图层包含实际的数据。CVBF\_Map是CVBFO\_SGNodeMap渲染的主要数据模型。通过earth文件创建图层时，可以指定多个影像数据源和多个高程数据源，数据源的顺序决定渲染顺序， 在earth文件中处于最前的在渲染时处于最底层渲染；所以如果有高低精度不同的影响数据或者高程数据， 在创建earth文件时要将粗精度的数据放在上方xml节点，高精度的放在其下面的节点。

class OSGEARTH\_EXPORT CVBF\_Map : public IVBF\_Map

{

friend class CVBF\_MapInfo;

public:

// 构造空图

CVBF\_Map( const CVBF\_OptionsMap& options =CVBF\_OptionsMap() );

public:

// 获取控制图特征的选项

const CVBF\_OptionsMap& GetMapOptions() const { return m\_MapOptions; }

// 获取图初始创建的选项

const CVBF\_OptionsMap& getInitialMapOptions() const { return m\_MapOptionsInit; }

// 1、获取地图的总体信息

// 获取图的master profile，该值在调用autoCalculateProfile之后才可用

const CVBF\_Profile\* getProfile() const;

// 获取空间参考

const CVBF\_SpatialReference\* getSRS() const { return m\_ipProfile.valid() ? m\_ipProfile->getSRS() : 0L; }

// 2、获取地图的图层

// 2.1 Image图层

int getNumImageLayers() const;// 获取影像图层的数量

Revision getImageLayers( CVBF\_LayersImage& out\_layers ) const;// 把影像图层的引用拷贝到输出列表，该方法是线程安全的，返回修订本（当数据拷贝时才生效）

IVBF\_MapLayer\* getImageLayerByName( const std::string& name ) const;// 通过名字获取影像图层

IVBF\_MapLayer\* getImageLayerByUID( UID layerUID ) const;// 通过ID获取影像图层

IVBF\_MapLayer\* getImageLayerAt( int index ) const;// 通过索引获取影像图层

// 2.2 高程图层

int getNumElevationLayers() const;// 获取高程图层的数量

Revision getElevationLayers( CVBF\_LayerTerrainElevations& out\_layers ) const;// 把高程图层的引用拷贝到输出列表，该方法是线程安全的

IVBF\_MapLayer\* getElevationLayerByName( const std::string& name ) const;// 通过名字获取高程图层

IVBF\_MapLayer\* getElevationLayerByUID( UID layerUID ) const;// 通过ID获取高程图层

IVBF\_MapLayer\* getElevationLayerAt( int index ) const;// 通过索引获取高程图层

// 2.3 模型图层（基本是矢量图层）

int getNumModelLayers() const;// 通过索引获取模型图层

Revision getModelLayers( CVBF\_LayersModel& out\_layers ) const;// 把模型图层的引用拷贝到输出列表，该方法是线程安全的

IVBF\_MapLayer\* getModelLayerByName( const std::string& name ) const;// 通过名字获取模型图层

IVBF\_MapLayer\* getModelLayerByUID( UID layerUID ) const;// 通过ID获取模型图层

IVBF\_MapLayer\* getModelLayerAt( int index ) const;// 通过索引获取模型图层

virtual bool AddModelLayer(const std::string& strFileName);

virtual void RemoveModelLayer(const std::string& strFileName);

// 2.4 Mask图层

int getTerrainMaskLayers( CVBF\_LayersMask& out\_list ) const;// 把地形Mask图层的引用拷贝到输出列表，该方法是线程安全的

// 3、添加/移除图层

// 3.0 在所有图层上的操作

void addMapCallback( CVBF\_MapCallback\* callback ) const;// 添加Map changed回调，这些回调在图层被添加、移除或调整顺序时被调用

void removeMapCallback( CVBF\_MapCallback\* callback ); // 移除Map changed回调

void beginUpdate(); // 1、开始/结束一批操作，如果想一次添加多个图层时调用，添加图层后调用 endUpdate()

void endUpdate(); // 2、在beginUpdate()和endUpdate()之间, 不激发任何callbacks（在addMapCallback中添加的）

void clear();// 清除所有图层

void setLayersFromMap( const CVBF\_Map\* map );// 用map的图层替换this的图层（除了地形掩码图层）

// 3.1 Image图层

void addImageLayer( CVBF\_LayerTerrainImage\* pLayer );// 添加一个图像图层

void insertImageLayer( CVBF\_LayerTerrainImage\* pLayer, unsigned int index );// 插入一个图像图层

void removeImageLayer( CVBF\_LayerTerrainImage\* pLayer );// 移除一个图像图层

void moveImageLayer( CVBF\_LayerTerrainImage\* pLayer, unsigned int newIndex );// 移动一个影像图层（把图层移到列表中另一个索引位置）

// 3.2 高程图层

void addElevationLayer( CVBF\_LayerTerrainElevation\* pLayer ); // 添加一个高程图层

void removeElevationLayer( CVBF\_LayerTerrainElevation\* pLayer );// 移除一个高程图层

void moveElevationLayer( CVBF\_LayerTerrainElevation\* pLayer, unsigned int newIndex );// 移动一个高程图层（把图层移到列表中另一个索引位置）

// 3.3 模型图层

void addModelLayer( CVBF\_LayerModel\* pLayer ); // 添加一个模型图层

void insertModelLayer( CVBF\_LayerModel\* pLayer, unsigned int index );// 插入一个模型图层

void removeModelLayer( CVBF\_LayerModel\* pLayer );// 移除一个模型图层

void moveModelLayer( CVBF\_LayerModel\* pLayer, unsigned int newIndex );// 移动一个模型图层（把图层移到列表中另一个索引位置）

// 3.4 Mask图层

void addTerrainMaskLayer( CVBF\_LayerMask\* pLayer ); // 添加一个地形掩码图层

void removeTerrainMaskLayer( CVBF\_LayerMask\* pLayer ); // 移除一个地形掩码图层

public:

// 获取用户提供的（user-provided）选项

const osgDB::ReaderWriter::Options\* getGlobalOptions() const;

void setGlobalOptions( const osgDB::ReaderWriter::Options\* options );

// 设置/获取名字

void setName( const std::string& name );

const std::string& getName() const { return m\_sName; }

/\*\* 创建覆盖TileKey给定范围的一个高程图，如果需要，降低分辨率

\* 说明: 缺省情况下，返回大地高（HAE，height above ellipsoid）高程图，即使TileKey profile是平均海平面（MSL，mean sea level）垂直基准。

\* 这是因为该函数常用于产生一个可绘制的高程图，可以通过设置参数convertToHAE=false改变这一行为。

\*

\* 参数key： 定义高程图的区域和理想的LOD

\* 参数fallbackIfNecessary： 如果不能生成该键值对应的高程图，降低LOD，直到能够生成

\* 参数out\_hf： 返回生成的高程图

\* 参数out\_isFallback： 告诉调用者是否降低了分辨率

\* 参数convertToHAE： 返回的高程图是否转换为大地高

\* 参数samplePolicy： 采样（插值）策略

\* 参数progress： (optional) progress callback.

\*/

bool getHeightField( const CVBF\_TileKey& key, bool fallbackIfNeessary, ef\_ptr<osg::HeightField>& out\_hf, bool\* out\_isFallback =0L, bool bToHAE =true, levationSamplePolicy samplePolicy = SAMPLE\_FIRST\_VALID, ProgressCallback\* progress =0L)const;

// 设置/获取缓存，NULL表示不用缓存

void setCache( Cache\* cache );

Cache\* getCache() const;

// 获取数据模型的修订本号（revision #）。每次添加、移除或移动图层，修订本号都会改变。

Revision getDataModelRevision() const;

// 判断map的坐标系统类型是否是地心坐标系统

bool isGeocentric() const;

// 把一个map frame同步到地图的当前版本，如果同步发生了则返回true，否则返回false

bool sync( class CVBF\_MapFrame& frame ) const;

// 获取地图关联的数据库选项

const osgDB::Options\* getDBOptions() const { return \_dbOptions.get(); }

const CVBF\_Profile\* getProfileNoVDatum() const { return m\_ipProfileNoVDatum.get(); }

enum ModelParts

{

IMAGE\_LAYERS = 1 << 0,

ELEVATION\_LAYERS = 1 << 1,

TERRAIN\_LAYERS = IMAGE\_LAYERS | ELEVATION\_LAYERS,

MODEL\_LAYERS = 1 << 2,

MASK\_LAYERS = 1 << 3,

MASKED\_TERRAIN\_LAYERS = TERRAIN\_LAYERS | MASK\_LAYERS,

ENTIRE\_MODEL = 0xff

};

protected:

virtual ~CVBF\_Map();

private:

std::string m\_sName; // 图名

CVBF\_OptionsMap m\_MapOptions; // 地图选项

const CVBF\_OptionsMap m\_MapOptionsInit;// 初始构造时传入的选项，开始时与m\_MapOptions相同，但后来m\_MapOptions可能被修改

CVBF\_LayersImage m\_ImageLayers; // 影像图层

CVBF\_LayerTerrainElevations m\_ElevationLayers; // 高程图层

CVBF\_LayersModel m\_ModelLayers; // 模型图层

CVBF\_LayersMask m\_TerrainMaskLayers;// 地形掩码图层

MapCallbackList \_mapCallbacks; // 回调

ref\_ptr<const osgDB::ReaderWriter::Options> \_globalOptions;

CVBF\_ReadWriteMutex \_mapDataMutex;

ref\_ptr<const CVBF\_Profile> m\_ipProfile; // 概述

ref\_ptr<const CVBF\_Profile> m\_ipProfileNoVDatum;

ref\_ptr<Cache> m\_ipCache; // 缓存

Revision \_dataModelRevision;

ref\_ptr<osgDB::Options> \_dbOptions;

private:

void calculateProfile();

};

##### 3）CVBFO\_SGNodeMap

地图的场景图根节点，CVBF\_Map是数据模型，CVBFO\_SGNodeMap是数据的视图。

class OSGEARTH\_EXPORT CVBFO\_SGNodeMap : public osg::CVBF\_SGNodeGroup

{

public:

// 通过参数列表从".earth"转载一个CVBFO\_SGNodeMap

static CVBFO\_SGNodeMap\* load( class ::ArgumentParser& arguments );

public:

CVBFO\_SGNodeMap();

CVBFO\_SGNodeMap( const CVBF\_OptionsMapNode& options );

CVBFO\_SGNodeMap( CVBF\_Map\* pMap );//通过CVBF\_Map创建一个空的CVBFO\_SGNodeMap

CVBFO\_SGNodeMap( CVBF\_Map\* pMap, const CVBF\_OptionsMapNode& options );

public:

virtual const char\* libraryName() const { return "osgEarth"; }

virtual const char\* className() const { return "CVBFO\_SGNodeMap"; }

// 获取CVBFO\_SGNodeMap对应（渲染）的CVBF\_Map

const CVBF\_Map\* GetMap() const;

CVBF\_Map\* GetMap();

// 获取空间Map的参考，便捷函数

const CVBF\_SpatialReference\* GetMapSRS() const;

// 获取内存中的地形场景

const CVBF\_Terrain\* getTerrain() const;

CVBF\_Terrain\* getTerrain();

// 从场景图中茶中最顶层的CVBFO\_SGNodeMap，不存在则返回NULL

static CVBFO\_SGNodeMap\* findMapNode( osg::IVBF\_SGNode\* graph, unsigned traversalMask =~0 );

static CVBFO\_SGNodeMap\* get( osg::IVBF\_SGNode\* graph, unsigned travMask =~0 ) { return findMapNode(graph, travMask); }

// 地形模型是否是geocentric

bool isGeocentric() const;

// 获取模型节点（包含所有的CVBF\_LayerModel对应的Node）

osg::IVBF\_SGNodeGroup\* getModelLayerGroup() const;

// 获取特定CVBF\_LayerModel对应的Node

osg::IVBF\_SGNode\* getModelLayerNode( CVBF\_LayerModel\* layer ) const;

// 添加/移除地形的装饰器（decorates）

void addTerrainDecorator( osg::IVBF\_SGNodeGroup\* decorator );

void removeTerrainDecorator( osg::IVBF\_SGNodeGroup\* decorator );

// 获取地图节点的覆盖装饰器，通常不需要直接访问，而是安装一个CVBFO\_SGNodeOverlayDrapeable

CVBFO\_SGNodeDecoratorTerrainOverlay\* getOverlayDecorator() { return m\_pOverlayDecorator; }

// 获取地图节点的配置项

const CVBF\_OptionsMapNode& GetMapNodeOptions() const;

// 获取渲染地形表面的地形引擎

CVBFO\_SGNodeTerrainEngine\* getTerrainEngine() const;

// 在.earth文件中有<external>节点，该节点可被EARTH读取但不进行任何处理，可用于其他部分或应用层

const CVBF\_Config& externalConfig() const { return \_externalConf; }

CVBF\_Config& externalConfig() { return \_externalConf; }

// 在潜在的地形引擎上设置自定义的纹理合成技术（texture compositor technique），

void setCompositorTechnique( class TextureCompositorTechnique\* tech );

public: // special purpose

CVBFO\_SGNodeMap( CVBF\_Map\* pMap, const CVBF\_OptionsMapNode& options, bool initMap );

public: // 重载 osg::Node

virtual osg::BoundingSphere computeBound() const;

virtual void traverse( class osg::NodeVisitor& nv );

protected:

virtual ~CVBFO\_SGNodeMap();

private:

ref\_ptr<CVBF\_Map> m\_ipMap;

ref\_ptr<osg::IVBF\_SGNodeGroup> m\_ipNodeModels;

ref\_ptr<osg::IVBF\_SGNodeGroup> m\_ipNodeOverlayModels;

CVBFO\_SGNodeDecoratorTerrainOverlay\* m\_pOverlayDecorator;

CVBF\_OptionsMapNode \_mapNodeOptions;

CVBF\_Config \_externalConf;

// keep track of nodes created by model layers

typedef std::map<CVBF\_LayerModel\*,osg::IVBF\_SGNode\*> ModelLayerNodeMap;

ModelLayerNodeMap \_modelLayerNodes;

osg::IVBF\_SGNodeGroup\* \_maskLayerNode;

unsigned \_lastNumBlacklistedFilenames;

ref\_ptr<CVBFO\_SGNodeTerrainEngine> m\_ipTerrainEngine;

bool m\_bTerrainEngineInitialized;

osg::IVBF\_SGNodeGroup\* m\_pTerrainEngineContainer;

public: // CVBF\_MapCallback proxy

void onModelLayerAdded( CVBF\_LayerModel\*, unsigned int );

void onModelLayerRemoved( CVBF\_LayerModel\* );

void onModelLayerMoved( CVBF\_LayerModel\* layer, unsigned int oldIndex, unsigned int newIndex );

public: // ModelLayerCallback

void onModelLayerOverlayChanged( CVBF\_LayerModel\* layer );

struct TileRangeData : public ::CVBF\_Referenced

{

TileRangeData(double minRange, double maxRange) : \_minRange( minRange ), \_maxRange( maxRange ) { }

double \_minRange;

double \_maxRange;

};

private:

ref\_ptr< ModelLayerCallback > \_modelLayerCallback;

ref\_ptr< CVBF\_MapCallback > \_mapCallback;

void init();

CVBF\_MapNodeCullData\* getCullData(osg::IVBF\_Camera\* camera) const;

typedef std::map<osg::IVBF\_Camera\*, ref\_ptr<CVBF\_MapNodeCullData> > CullDataMap;

mutable CullDataMap \_cullData;

mutable CVBF\_ReadWriteMutex \_cullDataMutex;

};

##### 4）CVBF\_SpatialReference

地图使用的空间参考系，描述参考椭球、高程基准、投影方式。

class OSGEARTH\_EXPORT CVBF\_SpatialReference : public IVBF\_SpatialReference, public CVBF\_Referenced

{

public:

// 创建空间参考系， 两个参数分别表示水平基准和垂直基准，如果未指定垂直基准，系统将使用椭球的大地基准面（geodetic datum for the ellipsoid）

static CVBF\_SpatialReference\* create(const std::string& init, const std::string& vinit="");

static CVBF\_SpatialReference\* get(const std::string& init, const std::string& vinit="") { return create(init,vinit); }

// 利用一个osg坐标系节点创建一个空间参考系， 如果输入参数的信息不完备，返回NULL

static CVBF\_SpatialReference\* create(osg::CVBF\_SGNodeGroupCoordSys\* csn);

// 利用一个OGR空间参考句柄创建一个空间参考系，如果xferOwnership设为true，该IVBF\_SpatialReference对象负责在析构时释放OGR句柄

static CVBF\_SpatialReference\* createFromHandle(void\* ogrHandle, bool xferOwnership=false);

public: // 基本变换

// 将一个点的坐标从一个SRS变换到另一个SRS，如果变换成功返回true，否则返回false

virtual bool transform(const osg::Vec3d& input, const IVBF\_SpatialReference\* outputSRS, osg::Vec3d& output) const;

// 将一组点的坐标从一个SRS变换到另一个SRS，只有当所有点都变换成功时才返回true，否则返回false

virtual bool transform(std::vector<osg::Vec3d>& input, const IVBF\_SpatialReference\* outputSRS) const;

// 变换一个二维坐标（一种比较方便的变换方法）

virtual bool transform2D(double x, double y, const IVBF\_SpatialReference\* outputSRS, double& out\_x, double& out\_y) const;

public: // 坐标单位变换

// 将一个距离值从一个SRS变换到另一个SRS，如果其中一个SRS是地理坐标系（即有角度单位），对应的距离值为大圆距离

virtual double transformUnits(double distance, const IVBF\_SpatialReference\* outputSRS) const;

public: // 与世界坐标系之间的变换

// 将一个点的坐标从该SRS变换到世界坐标系，必要时会根据SRS的高程基准对z坐标进行归一化，并转换到ECFF

virtual bool transformToWorld(const osg::Vec3d& input, osg::Vec3d& out\_world) const;

/\*\*

将一个点的坐标从世界坐标系变换到该SRS

@param world [in] 世界坐标系下点的坐标

@param out\_local [out] 该SRS下点的坐标

@param out\_geodeticZ [out, optional] 大地高 (即HAE: Height Above Ellipsoid)

\*/

virtual bool transformFromWorld(const osg::Vec3d& world, osg::Vec3d& out\_local, double\* out\_geodeticZ=0L) const;

public: // 外部变换

/\*\*

把一个空间范围由该SRS变换到另一个SRS，变换后的范围实际上为原始范围的最小包围矩形（MBR: Minimum Bounding Rectangle）

\*/

virtual bool transformExtentToMBR(const IVBF\_SpatialReference\* to\_srs, double& in\_out\_xmin, double& in\_out\_ymin, double& in\_out\_xmax, double& in\_out\_ymax ) const;

/\*\*

把一个空间范围由该SRS变换到另一个SRS，并输出转换后范围的包围矩形

\*/

virtual bool transformExtentPoints(const IVBF\_SpatialReference\* to\_srs, double in\_xmin, double in\_ymin, double in\_xmax, double in\_ymax, double\* x, double\* y, unsigned numx, unsigned numy ) const;

public: // 各种属性

virtual bool isGeographic() const; // 是否为地理坐标系 （经度/纬度/msl）

virtual bool isGeodetic() const; // 是否为大地坐标系 （经度/纬度/大地高）

virtual bool isProjected() const; //是否为投影坐标系 （即局部坐标系）

virtual bool isECEF() const; // 是否为ECEF坐标系 （地心坐标系/米）

virtual bool isMercator() const;//是否使用Mercator投影

virtual bool isSphericalMercator() const; // 是否使用球体Mercator伪投影（Spherical Mercator pseudo-projection），即把地球模拟为球体而非椭球体

/\*\* 是否使用极射赤平投影（polar sterographic projection）

极射赤平投影：把面和线放在一个投影球的中心，通过球心的面和线延伸后与球面相交，分别形成弧线和点。把球面上弧线和点与投影球的顶点（上半球或下半球的极点）相连，投影到赤道平面上，即为极射赤平投影。赤平投影通常在构造地质学中用以解决地质构造的角度和方位问题。

\*/

virtual bool isNorthPolar() const;

virtual bool isSouthPolar() const;

//是否为用户自定义的SRS （即GDAL不支持的一种SRS)

virtual bool isUserDefined() const;

// 该SRS中的坐标是否来源于一个连续空间，非连续空间中的相邻坐标在地图中的实际位置不一定相邻

virtual bool isContiguous() const;

//是否为内部定义的Unified Cube投影

virtual bool isCube() const;

// 是否为内部定义的局部切平面投影（Local Tangent Plane projection）

virtual bool isLTP() const { return \_is\_ltp; }

// 是否为geographic plate carre SRS

virtual bool isPlateCarre() const { return \_is\_plate\_carre; }

virtual const std::string& getName() const;//获取该SRS的名称

// 获取该SRS使用的参考椭球

virtual const osg::EllipsoidModel\* getEllipsoid() const;

// 获取描述该SRS的标准字符串（WKT，Well-Known Text）

// WKT: 是一种文本标记语言，用于表示矢量几何对象、空间参考系以及空间参考系之间的转换。该格式由OGC制定

virtual const std::string& getWKT() const;

virtual const std::string& getInitType() const;//获取初始类型（如PROJ4, WKT，等等）

//获取初始关键值

typedef std::pair<std::string, std::string> Key;

virtual const Key& getKey() const;

// 获取水平基准面的初始字符串

virtual const std::string& getHorizInitString() const;

// 获取垂直基准面的初始字符串

virtual const std::string& getVertInitString() const;

//获取该SRS使用的基准面的标识符（可以为空）

virtual const std::string& getDatumName() const;

// 获取该SRS使用的坐标单位

virtual const CVBF\_Units& getUnits() const;

// 判断两个SRS是否完全相同

virtual bool isEquivalentTo(const IVBF\_SpatialReference\* rhs) const;

// 判断两个SRS使用的水平基准面是否相同

virtual bool isHorizEquivalentTo(const IVBF\_SpatialReference\* rhs) const;

// 判断两个SRS使用的垂直基准面是否相同

virtual bool isVertEquivalentTo(const IVBF\_SpatialReference\* rhs) const;

// 获取该SRS使用的地理坐标系

virtual const IVBF\_SpatialReference\* getGeographicSRS() const;

/\*\* 获取该SRS使用的大地坐标系，实际上就是在地理坐标系+垂直基准（Z坐标表示大地高） \*/

virtual const IVBF\_SpatialReference\* getGeodeticSRS() const;

/\*\* 获取该SRS椭球使用的ECEF参考系 \*/

virtual const IVBF\_SpatialReference\* getECEF() const;

/\*\* 获取垂直基准，如果为null，系统使用一个默认的大地垂直基准 \*/

virtual const CVBFO\_VerticalDatum\* getVerticalDatum() const;

/\*\* 利用该SRS中的一个坐标点，构建一个局部坐标系到世界坐标系的变换矩阵 \*/

virtual bool createLocalToWorld(const osg::Vec3d& point, osg::Matrixd& out\_local2world) const;

/\*\* 利用该SRS中的一个坐标点，构建一个世界坐标系到局部坐标系的变换矩阵 \*/

virtual bool createWorldToLocal(const osg::Vec3d& point, osg::Matrix& out\_world2local) const;

/\*\* Creates and returns a local trangent plane SRS at the given reference location.

The reference location is expressed in this object's SRS, but it tangent to

the globe at getGeographicSRS(). LTP units are in meters. \*/

/\*\*

在指定参考点处构建并返回该点的局部切平面SRS，单位为米。

参考点位于该SRS内，与球在getGeographicSRS()处相切???

\*/

virtual const IVBF\_SpatialReference\* createTangentPlaneSRS(const osg::Vec3d& refPos) const;

// 在指定经纬度坐标处构建该点的横轴Mercator投影

virtual const IVBF\_SpatialReference\* createTransMercFromLongitude( const Angular& lon ) const;

// 在指定经纬度坐标处和UTM分带，构建通用横轴Mercator投影（UTM）,该函数比上一个函数效率稍高一些。

virtual const IVBF\_SpatialReference\* createUTMFromLonLat(const Angle& lon, const Angle& lat) const;

/\*\* 复制该SRS，为区别起见，复制后的SRS标识为Plate Carre模式 \*/

virtual const IVBF\_SpatialReference\* createPlateCarreGeographicSRS() const;

/\*\* 以该SRS为基础，创建一个新的坐标系节点 \*/

virtual osg::CVBF\_SGNodeGroupCoordSys\* createCoordinateSystemNode() const;

/\*\* 将该SRS的信息填充到一个坐标系节点中 \*/

virtual bool populateCoordinateSystemNode(osg::CVBF\_SGNodeGroupCoordSys\* csn) const;

/\*\*

以该SRS为基础，创建一个新的Locator对象。

@param xmin, ymin, xmax, ymax Locator的坐标范围，单位：度

@param plate\_carre 是否使用plate\_carre模式

\*/

virtual CVBFO\_GeoLocator\* createLocator(double xmin, double ymin, double xmax, double ymax, bool plate\_carre=false) const;

protected:

CVBF\_SpatialReference(void\* handle, const std::string& type);

CVBF\_SpatialReference(void\* handle, bool ownsHandle=true );

virtual ~CVBF\_SpatialReference();

void init();

// 用户可在派生的子类中重载下列函数，注意：重载时需要同时调用基类的函数

virtual void \_init();

virtual bool \_isEquivalentTo(const CVBF\_SpatialReference\* srs, bool considerVDatum=true) const;

virtual bool preTransform(std::vector<osg::Vec3d>&) const { return true; }

virtual bool postTransform(std::vector<osg::Vec3d>&) const { return true; }

bool transformXYPointArrays(double\* x, double\* y, unsigned numPoints, const CVBF\_SpatialReference\* out\_srs) const;

bool transformZ(std::vector<osg::Vec3d>& points, const CVBF\_SpatialReference\* outputSRS, bool pointsAreGeodetic) const;

typedef std::map<Key, ref\_ptr<CVBF\_SpatialReference> > SRSCache;

static SRSCache& getSRSCache();

protected:

bool \_initialized;

void\* \_handle;

bool \_owns\_handle;

bool \_is\_geographic;

bool \_is\_mercator;

bool \_is\_spherical\_mercator;

bool \_is\_north\_polar, \_is\_south\_polar;

bool \_is\_cube;

bool \_is\_contiguous;

bool \_is\_user\_defined;

bool \_is\_ltp;

bool \_is\_plate\_carre;

bool \_is\_ecef;

unsigned \_ellipsoidId;

std::string m\_sName;

Key \_key;

std::string \_wkt;

std::string \_proj4;

std::string \_init\_type;

std::string \_datum;

CVBF\_Units \_units;

ref\_ptr<osg::EllipsoidModel> \_ellipsoid;

ref\_ptr<CVBF\_SpatialReference> \_geo\_srs;

ref\_ptr<CVBF\_SpatialReference> \_geodetic\_srs; // \_geo\_srs with a NULL vdatum.

ref\_ptr<CVBF\_SpatialReference> \_ecef\_srs;

ref\_ptr<CVBFO\_VerticalDatum> \_vdatum;

typedef std::map<std::string, void\*> TransformHandleCache;

TransformHandleCache \_transformHandleCache;

private:

static CVBF\_SpatialReference\* create(const Key& key, bool useCache);

static CVBF\_SpatialReference\* createFromWKT(const std::string& wkt, const std::string& name ="");

static CVBF\_SpatialReference\* createFromPROJ4(const std::string& proj4, const std::string& name = "");

static CVBF\_SpatialReference\* createCube();

CVBF\_SpatialReference\* fixWKT();

};

##### 5）IVBF\_MapLayer

地图图层接口。

class IVBF\_MapLayer : public ::CVBF\_Referenced

{

public:

virtual ~IVBF\_MapLayer() {}

public:

// 图层的唯一标识

virtual int getUID() const = 0;

// 获取图层名

virtual const std::string& getName() const = 0;

virtual bool getEnabled() const = 0;

// 是否绘制（或别的应用）该图层

virtual void setVisible( bool value ) = 0;

virtual bool getVisible() const = 0;

virtual void Dirty() = 0;

virtual void AddObserver(IVBF\_LayerObserver\* pLayerObserver)=0;

virtual void RemoveObserver(IVBF\_LayerObserver\* pLayerObserver)=0;

};

##### 6）CVBF\_LayerTerrain

地形图层，是高程和影像图层的基类。

class OSGEARTH\_EXPORT CVBF\_LayerTerrain : public CVBF\_Layer

{

protected:

CVBF\_LayerTerrain( const TerrainLayerOptions& initOptions, TerrainLayerOptions\* runtimeOptions );

CVBF\_LayerTerrain( const TerrainLayerOptions& initOptions, TerrainLayerOptions\*runtimeOptions, CVBFO\_SourceTile\* tileSource );

virtual ~CVBF\_LayerTerrain();

public:

// 获取图层的选项（初始化选项、运行时选项）

const TerrainLayerOptions& getInitialOptions() const { return \_initOptions; }

const TerrainLayerOptions& getTerrainLayerRuntimeOptions() const { return \*\_runtimeOptions; }

// 是否启用该图层。Note, a layer is enabled/disabled once and its status cannot be changed.

bool getEnabled() const { return \*\_runtimeOptions->enabled(); }

// 是否绘制（或别的应用）该图层

void setVisible( bool value );

bool getVisible() const { return getEnabled() && \*\_runtimeOptions->visible(); }

// 获取图层名

const std::string& getName() const { return getTerrainLayerRuntimeOptions().name(); }

// 获取该图层的profile

const CVBF\_Profile\* getProfile() const;

// 获取该图层所用的数据源

CVBFO\_SourceTile\* getTileSource() const;

// 获取该图层的瓦片大小

unsigned int getTileSize() const;

// 该图层是否描绘动态数据，也就是说瓦片数据是否会变化

bool isDynamic() const;

// 给定的瓦片键值对于该层是否有效

virtual bool isKeyValid(const CVBF\_TileKey& key) const;

// 给定的瓦片键值对应的数据是否在缓存中

virtual bool isCached(const CVBF\_TileKey& key) const;

// 告诉地形图层将用于什么样的场景轮廓（球形、平面形），这是可选项，由于含有更多如何使用数据的信息，对图层优化有用

virtual void setTargetProfileHint( const CVBF\_Profile\* profile );

// The cache bin for storing data generated by this layer

virtual CacheBin\* getCacheBin( const CVBF\_Profile\* profile );

// 获取该图层所用的缓存

Cache\* getCache() const { return \_cache.get(); }

// 是否仅用缓存中的数据

bool isCacheOnly() const

{

return

\_runtimeOptions->cachePolicy().isSet() &&

\_runtimeOptions->cachePolicy()->usage() == CachePolicy::USAGE\_CACHE\_ONLY;

}

public: // 重载Layer的接口

virtual IVBF\_SequenceControl\* getSequenceControl();

public:

// 存储在缓存中的地形图层的元数据

struct CacheBinMetadata

{

CacheBinMetadata() { }

CacheBinMetadata( const CacheBinMetadata& rhs ) :

\_empty ( rhs.\_empty ),

\_cacheBinId ( rhs.\_cacheBinId ),

\_sourceName ( rhs.\_sourceName ),

\_sourceDriver ( rhs.\_sourceDriver ),

\_sourceProfile( rhs.\_sourceProfile ),

\_cacheProfile ( rhs.\_cacheProfile ) { }

CacheBinMetadata( const CVBF\_Config& conf )

{

\_empty = conf.isEmpty();

conf.getIfSet ( "cachebin\_id", \_cacheBinId );

conf.getIfSet ( "source\_name", \_sourceName );

conf.getIfSet ( "source\_driver", \_sourceDriver );

conf.getObjIfSet( "source\_profile", \_sourceProfile );

conf.getObjIfSet( "cache\_profile", \_cacheProfile );

}

CVBF\_Config getConfig() const

{

CVBF\_Config conf( "osgearth\_terrainlayer\_cachebin" );

conf.addIfSet ( "cachebin\_id", \_cacheBinId );

conf.addIfSet ( "source\_name", \_sourceName );

conf.addIfSet ( "source\_driver", \_sourceDriver );

conf.addObjIfSet( "source\_profile", \_sourceProfile );

conf.addObjIfSet( "cache\_profile", \_cacheProfile );

return conf;

}

bool \_empty;

CVBF\_Optional<std::string> \_cacheBinId;

CVBF\_Optional<std::string> \_sourceName;

CVBF\_Optional<std::string> \_sourceDriver;

CVBF\_Optional<CVBF\_OptionsProfile> \_sourceProfile;

CVBF\_Optional<CVBF\_OptionsProfile> \_cacheProfile;

};

// 获取缓存的信息

bool getCacheBinMetadata( const CVBF\_Profile\* profile, CacheBinMetadata& output );

protected:

virtual void initTileSource();

CacheBin\* getCacheBin( const CVBF\_Profile\* profile, const std::string& binId );

protected:

ref\_ptr<CVBFO\_SourceTile> \_tileSource;

ref\_ptr<const CVBF\_Profile> \_profile;

ref\_ptr<const CVBF\_Profile> \_targetProfileHint;

bool \_tileSourceInitAttempted;

bool \_tileSourceInitFailed;

unsigned \_tileSize;

ref\_ptr<osgDB::Options> \_dbOptions;

void setCachePolicy( const CachePolicy& cp );

const CachePolicy& getCachePolicy() const;

private:

std::string m\_sName;

std::string \_referenceURI;

OpenThreads::Mutex \_initTileSourceMutex;

TerrainLayerOptions \_initOptions;

TerrainLayerOptions\* \_runtimeOptions;

ref\_ptr<Cache> \_cache;

// maps profile signature to cache bin pointer.

struct CacheBinInfo

{

ref\_ptr<CacheBin> \_bin;

CVBF\_Optional<CacheBinMetadata> \_metadata;

};

typedef std::map< std::string, CacheBinInfo > CacheBinInfoMap; // indexed by profile signature

CacheBinInfoMap \_cacheBins;

CVBF\_ReadWriteMutex \_cacheBinsMutex;

void init();

virtual void fireCallback( TerrainLayerCallbackMethodPtr method ) =0;

friend class CVBF\_Map;

void setDBOptions( const osgDB::Options\* dbOptions );

void initializeCachePolicy( const osgDB::Options\* );

void storeProxySettings( osgDB::Options\* );

// 设置图层的缓存

void setCache( Cache\* cache );

};

##### 7）CVBF\_LayerTerrainElevation

高程图层。

class OSGEARTH\_EXPORT CVBF\_LayerTerrainElevation : public CVBF\_LayerTerrain

{

public:

// 根据指定的配置选项构建高程图层，该选项需包含必要的数据源驱动器配置选项

CVBF\_LayerTerrainElevation( const ElevationLayerOptions& options );

// 根据名字和数据源驱动器配置选项构建高程图层

CVBF\_LayerTerrainElevation( const std::string& name, const CVBF\_OptionsDriverSourceTile& driverOptions );

// 根据图层选项和用户提供的数据源创建高程图层

CVBF\_LayerTerrainElevation( const ElevationLayerOptions& options, CVBFO\_SourceTile\* tileSource );

virtual ~CVBF\_LayerTerrainElevation() { }

// 获取创建该图层的初始选项

const ElevationLayerOptions& getElevationLayerOptions() const { return \_runtimeOptions; }

virtual const TerrainLayerOptions& getTerrainLayerRuntimeOptions() const { return \_runtimeOptions; }

// 添加/移除一个图层属性更新回调

void addCallback( ElevationLayerCallback\* cb );

void removeCallback( ElevationLayerCallback\* cb );

public: // methods

// 创建一个与给定TileKey的范围与LOD对应的高度图

virtual GeoHeightField createHeightField( const CVBF\_TileKey& key, ProgressCallback\* progress =0L );

protected:

// 从瓦片数据源（tile source）直接创建一个高度图

osg::HeightField\* createHeightFieldFromTileSource( const CVBF\_TileKey& key, ProgressCallback\* progress);

// assembles tiles from a layer that is not in the same profile as the map, and

// returns a single tile in the map's profile.

osg::HeightField\* assembleHeightFieldFromTileSource( const CVBF\_TileKey& key, ProgressCallback\* progress );

virtual std::string suggestCacheFormat() const;

virtual void initTileSource();

private:

ElevationLayerOptions \_runtimeOptions;

ElevationLayerCallbackList \_callbacks;

virtual void fireCallback( TerrainLayerCallbackMethodPtr method );

virtual void fireCallback( ElevationLayerCallbackMethodPtr method );

ref\_ptr<CVBFO\_SourceTile::HeightFieldOperation> \_preCacheOp;

void init();

};

##### 8）CVBF\_LayerTerrainImage

影像图层。

class OSGEARTH\_EXPORT CVBF\_LayerTerrainImage : public CVBF\_LayerTerrain

{

public:

CVBF\_LayerTerrainImage( const ImageLayerOptions& options );

CVBF\_LayerTerrainImage( const std::string& name, const CVBF\_OptionsDriverSourceTile& driverOptions );

CVBF\_LayerTerrainImage( const ImageLayerOptions& options, CVBFO\_SourceTile\* tileSource );

virtual ~CVBF\_LayerTerrainImage() {}

public:

// 获取图层的选项（初始化选项、运行时选项）

const ImageLayerOptions& getImageLayerOptions() const { return \_runtimeOptions; }

virtual const TerrainLayerOptions& getTerrainLayerRuntimeOptions() const { return \_runtimeOptions; }

// 添加/移除一个回调函数，用于该图层向外通知属性变化

void addCallback( ImageLayerCallback\* cb );

void removeCallback( ImageLayerCallback\* cb );

/\*\* Override: see CVBF\_LayerTerrain \*/

virtual void setTargetProfileHint( const CVBF\_Profile\* profile );

// 添加/移除一个颜色过滤器

void addColorFilter( ColorFilter\* filter );

void removeColorFilter( ColorFilter\* filter );

// 获取颜色过滤器数组

const ColorFilterChain& getColorFilters() const;

public:

// 设置/获取图像图层的透明度，0..1] -> [透明..不透明]

void setOpacity( float opacity );

float getOpacity() const { return \*\_runtimeOptions.opacity(); }

void disableLODBlending();

bool isLODBlendingEnabled() const { return \*\_runtimeOptions.lodBlending(); }

float getMinVisibleRange() const { return \*\_runtimeOptions.minVisibleRange();}

void setMinVisibleRange( float minVisibleRange );

float getMaxVisibleRange() const { return \*\_runtimeOptions.maxVisibleRange();}

void setMaxVisibleRange( float maxVisibleRange );

//xahg h00017

CVBF\_Optional<int>& shareImageUnit() { return \_shareImageUnit; }

const CVBF\_Optional<int>& shareImageUnit() const { return \_shareImageUnit; }

//xahg h00017

bool isShareEnabled() const { return \*\_runtimeOptions.share(); }

public:

// 创建一个与key对应的GeoImage，该图像使用key的profile，并自动被重投影、拼接、裁剪

virtual GeoImage createImage( const CVBF\_TileKey& key, ProgressCallback\* progress = 0, bool forceFallback =false);

// 用自己的profile创建图像

GeoImage createImageInNativeProfile(const CVBF\_TileKey& key, ProgressCallback\* progress, bool forceFallback, bool& out\_isFallback);

public: // CVBF\_LayerTerrain override

CacheBin\* getCacheBin( const CVBF\_Profile\* profile );

virtual int unref() const;

virtual int ref() const;

protected:

// Creates an image that's in the same profile as the provided key.

GeoImage createImageInKeyProfile(const CVBF\_TileKey& key, ProgressCallback\* progress, bool forceFallback, bool& out\_isFallback);

// Fetches an image from the underlying CVBFO\_SourceTile whose data matches that of the key extent.

GeoImage createImageFromTileSource(const CVBF\_TileKey& key, ProgressCallback\* progress, bool forceFallback, bool& out\_isFallback);

// Fetches multiple images from the CVBFO\_SourceTile; mosaics/reprojects/crops as necessary, and

// returns a single tile. This is called by createImageFromTileSource() if the key profile

// doesn't match the layer profile.

GeoImage assembleImageFromTileSource(const CVBF\_TileKey& key, ProgressCallback\* progress, bool& out\_isFallback);

virtual void initTileSource();

protected:

ImageLayerOptions \_runtimeOptions;

ref\_ptr<CVBFO\_SourceTile::ImageOperation> \_preCacheOp;

ref\_ptr<osg::Image> \_emptyImage;

ImageLayerCallbackList \_callbacks;

//xahg h00017

CVBF\_Optional<int> \_shareImageUnit;

virtual void fireCallback( TerrainLayerCallbackMethodPtr method );

virtual void fireCallback( ImageLayerCallbackMethodPtr method );

void init();

void initPreCacheOp();

};

##### 9）CVBF\_LayerModel

模型图层，主要指矢量数据。

class OSGEARTH\_EXPORT CVBF\_LayerModel : public CVBF\_Layer

{

public:

// 根据模型图层的配置选项创建模型图层

CVBF\_LayerModel( const ModelLayerOptions& options );

// 根据数据源驱动器配置项创建模型图层

CVBF\_LayerModel( const std::string& name, const CVBF\_OptionsDriverSourceModel& options );

// 根据模型图层的配置选项和数据源创建模型图层

CVBF\_LayerModel(const ModelLayerOptions& options, CVBFO\_SourceModel\* pSource );

// 根据图层名和一个已经存在的节点创建模型图层//!!!可以去掉？？？

CVBF\_LayerModel(const std::string& name, osg::IVBF\_SGNode\* pINode);

virtual ~CVBF\_LayerModel();

public:

// 获取图层的名称

const std::string& getName() const { return \*\_runtimeOptions.name(); }

// 获取图层的初始配置选项

const ModelLayerOptions& getModelLayerOptions() const { return \_initOptions; }

// 获取模型数据源

CVBFO\_SourceModel\* getModelSource() const { return m\_ipModelSource.get(); }

public:

// 初始化，由Map调用

void initialize( const osgDB::Options\* options );

// 创建描述该图层的场景图节点

osg::IVBF\_SGNode\* createSceneGraph( const CVBF\_Map\* map, const osgDB::Options\* dbOptions, ProgressCallback\* progress );

public: // properties

// 获取/设置是否渲染该图层

bool getVisible() const;

void setVisible(bool value);

// 是否启用该图层

bool getEnabled() const;

// 该图层是作为正常几何体，还是作为地形覆盖层绘制

bool getOverlay() const;

void setOverlay( bool overlay );

// 图层的根节点是否启用光照

void setLightingEnabled( bool value );

bool isLightingEnabled() const;

public:

// 添加/移除一个回调函数，该回调函数向外部通知图层属性变化

void addCallback( ModelLayerCallback\* cb );

void removeCallback( ModelLayerCallback\* cb );

private:

ref\_ptr<CVBFO\_SourceModel> m\_ipModelSource;

const ModelLayerOptions \_initOptions;

ModelLayerOptions \_runtimeOptions;

Revision \_modelSourceRev;

ModelLayerCallbackList \_callbacks;

UpdateLightingUniformsHelper \_updateLightingUniformsHelper;

typedef std::set< observer\_ptr<osg::IVBF\_SGNode> > NodeObserverSet;

NodeObserverSet \_nodeSet;

virtual void fireCallback( ModelLayerCallbackMethodPtr method );

void copyOptions();

};

##### 10）CVBF\_LayerMask

掩码图层，它是一个特殊的图层，用于盖掉地形的一部分（相当于插图）。

class OSGEARTH\_EXPORT CVBF\_LayerMask : public CVBF\_Layer

{

public:

// 根据配置选项构建掩码图层

CVBF\_LayerMask( const MaskLayerOptions& options =MaskLayerOptions() );

// 根据数据驱动器选项构建掩码图层

CVBF\_LayerMask( const std::string& name, const CVBF\_OptionsDriverSourceMask& options );

// 根据掩饰数据源构建掩码图层

CVBF\_LayerMask(const MaskLayerOptions& options, CVBFO\_SourceMask\* source );

virtual ~CVBF\_LayerMask() { }

// 获取掩码图层数据源

CVBFO\_SourceMask\* getMaskSource() const { return \_maskSource.get(); }

virtual int getUID() const { return 0; }

virtual bool getEnabled() const { return true; }

virtual void setVisible( bool value ){}

virtual bool getVisible() const { return true; }

public:

// 获取名字

const std::string& getName() const { return \*\_runtimeOptions.name(); }

// 获取被掩饰掉区域的几何边界多边形

osg::Vec3dArray\* getOrCreateBoundary( float heightScale = 1.0, const CVBF\_SpatialReference\* srs = NULL, ProgressCallback\* progress =0L );

public:

// 初始化

void initialize( const osgDB::Options\* dbOptions, const CVBF\_Map\* map );

private:

MaskLayerOptions \_initOptions; // 初始化选项

MaskLayerOptions \_runtimeOptions; // 运行时选项

ref\_ptr<CVBFO\_SourceMask> \_maskSource; // 数据源

Revision \_maskSourceRev; // 数据源版本

ref\_ptr<osg::Vec3dArray> \_boundary; // 边界

ref\_ptr<osgDB::Options> \_dbOptions; // 数据库选项

void copyOptions();

};

#### 5.3.4三维标绘模块

整个三维标绘模块通过接口IVBF\_3DPlot完成管理，它包含若干标图文档（IVBF\_3DPlotDoc），该文档负责记录标图数据，但场景中每次只能有一个标图文档处于激活状态。

每个标图文档由若干标图图层（IVBF\_3DPlotLayer）构成，从而可实现对模型的分层控制。每个图层又可由若干实体模型构成。实体模型种类繁多，根据类型分布在各个插件中，提供各类模型、属性、过程等的创建，使用时通过配置文件3DRegisterMarkCreator.cfg实现动态加载。

##### 1）IVBF\_3DPlot接口

标绘接口。负责管理标图文档、创建模型、编辑模型、串行化等。该接口位于VBF\_3DPlot.dll中，以组件方式动态加载。可通过如下方式获取到该接口：

IVBF\_3DPlot\* pI3DPlot = (IVBF\_3DPlot\*)pIMainCtrl->QueryInterface(VBF\_INTERFACE\_3DPLOT);

该接口的定义如下：

class IVBF\_3DPlot : public CVBF\_Referenced

{

public:

virtual ~IVBF\_3DPlot() {}

public:

// 设置/获取标图模块的子状态，默认值为VBF\_PLOT\_STATE\_NONE

virtual void SetState(EVBF\_PLOT\_STATE nState)=0;

virtual EVBF\_PLOT\_STATE GetState()=0;

// 获取模型编辑接口

virtual IVBF\_3DPlotEditor\* GetEditor()=0;

// 获取系统自动创建的缺省文档，该文档不允许删除

virtual IVBF\_3DPlotDoc\* GetDefaultDoc()=0;

virtual std::string GetDefaultDocName()=0;

// 添加/删除/查找一个标图文档

virtual IVBF\_3DPlotDoc\* AddDoc(const std::string& strDocName)=0;

virtual bool DeleteDoc(const std::string& strDocName)=0;

virtual bool DeleteDoc(IVBF\_3DPlotDoc\* pIDoc)=0;

virtual IVBF\_3DPlotDoc\* GetDoc(const std::string& strDocName)=0;

// 激活一个标图文档

virtual void ActivateDoc(const std::string& strDocName)=0;

virtual IVBF\_3DPlotDoc\* GetActiveDoc()=0;

virtual const std::string& GetActiveDocName()=0;

// 遍历标图文档

virtual int GetDocCount()=0;

virtual IVBF\_3DPlotDoc\* GetDoc(int nIndex)=0;

// 获取所有标图文档

virtual void GetAllDocs(std::vector<IVBF\_3DPlotDoc\*>& vDocs)=0;

// 输入一个军标模型，通过人机交互的方式

virtual void InputMark(unsigned int nMarkType, long nMarkCode=-1)=0;

virtual IVBF\_MarkInputHandler\* GetInputHandler()=0; // 获取输入处理器

// 通过指定类型，创建一个军标模型

virtual IVBF\_MarkModel\* CreateMarkModel(const std::string& strMarkID, unsigned int nMarkType)=0;

// 通过指定MarkStyle.xml配置文件中的类型编码，创建一个军标模型并读取模型信息

virtual IVBF\_MarkModel\* CreateMarkModel(const std::string& strMarkID, const std::string& strCode)=0;

// 通过指定xml节点，创建一个军标模型并读取模型信息

virtual IVBF\_MarkModel\* CreateMarkModel(TiXmlElement\* pElemMark)=0;

// 创建一个军标属性

virtual IVBF\_MarkAttr\* CreateMarkAttr(unsigned int nAttrType)=0;

// 创建一个过程

virtual IVBF\_MarkProcess\* CreateMarkProcess(unsigned int nProcessType)=0;

virtual void ProcessSimulate()=0; // 过程模拟

virtual void StatStartTime(double& dStartTime)=0;// 统计所有过程的起始时间

// 获取仿真时钟

virtual IVBF\_SimulationTimer\* GetSimulationTimer()=0;

// 给所有点状模型设置一个统一的自动缩放比

virtual float GetPointMarkAutoScale()=0;

// 给所有点状模型设置一个统一的缩放比系数

virtual void SetPointMarkScaleDelta(float fScaleDelta)=0;

virtual float GetPointMarkScaleDelta()=0;

// 工具性函数：根据模型指针，获取其所在的文档和图层名称

virtual bool GetDocLayerNameOfMark(IVBF\_MarkModel\* pIMark, std::string& strDocName, std::string& strLayerName)=0;

// 工具性函数：根据模型所在的文档、图层及模型ID号，获取模型指针

virtual IVBF\_MarkModel\* FindMark(const std::string& strDocName, const std::string& strLayerName, const std::string& strMarkID)=0;

// 在所有文档内查找模型

virtual IVBF\_MarkModel\* FindMark(const std::string& strMarkID)=0;

// 串行化，将标图信息写入指定的xml文件或节点

virtual bool WriteToXML(const std::string& strFn)=0;

virtual bool WriteToXML(TiXmlElement\* pElemPlot)=0;

// 串行化，从指定的xml文件或节点读取标图信息

virtual bool ReadFromXML(const std::string& strFn)=0;

virtual bool ReadFromXML(TiXmlElement\* pElemPlot)=0;

virtual void SetLineSmooth(bool bSmooth)=0;

virtual void SetPolySmooth(bool bSmooth)=0;

// 获取专门为非点状模型创建的根节点，避免他们受到父模型矩阵的影响

virtual osg::IVBF\_SGNodeGroup\* GetNodeForNotPointMarks()=0;

virtual void UpdateEciMatrix(const Matrix& mat)=0;

};

##### 2）IVBF\_3DPlotEditor接口

标图编辑器接口。主要负责模型的编辑。该接口可通过函数IVBF\_3DPlot::GetEditor()获取到。

进入模型编辑状态的示例代码如下：

// 通过鼠标编辑模型，使其旋转

pIMainCtrl->SelectTool(VBF\_TOOL\_MARK\_EDIT);

IVBF\_3DPlot\* pI3DPlot = (IVBF\_3DPlot\*)pIMainCtrl->QueryInterface(VBF\_INTERFACE\_3DPLOT);

if(pI3DPlot) pI3DPlot->SetState(VBF\_PLOT\_STATE\_ROTATE);

该接口的定义如下：

class IVBF\_3DPlotEditor : public CVBF\_Referenced

{

public:

virtual ~IVBF\_3DPlotEditor() {}

// 设置/获取模型编辑使用的Dragger是否固定大小 (单位：像素)

virtual void SetDraggerFixedSizeInPixels(bool bEnable, float fSizeInPixels=260)=0;

virtual void GetDraggerFixedSizeInPixels(bool& bEnable, float& fSizeInPixels)=0;

// 设置/获取Dragger中线段的宽度（默认值为2.0）

virtual void SetDraggerLineWidth(float fWidth)=0;

virtual float GetDraggerLineWidth()=0;

// 设置/获取Dragger中线段是否光滑（默认值为true，如果某些显卡的线段中出现黑色细线，可将该选项设置为false）

virtual void EnableDraggerLineSmooth(bool bEnable)=0;

virtual bool IsDraggerLineSmoothEnabled()=0;

// 设置/获取Dragger中三个坐标轴的颜色（默认值分别为红、绿、蓝）

virtual void SetDraggerAxisColor(const osg::Vec4f& colorX, const osg::Vec4f& colorY, const osg::Vec4f& colorZ)=0;

virtual void GetDraggerAxisColor(osg::Vec4f& colorX, osg::Vec4f& colorY, osg::Vec4f& colorZ)=0;

virtual void SetQuickSelectInfTipAsChart(IVBF\_MarkGraphicsCreator\* pICreator)=0;

};

##### 3）IVBF\_3DPlotDoc接口

标图文档接口。用于记录标图数据，主要负责管理各个标图图层。

class IVBF\_3DPlotDoc : public osg::CVBF\_Object

{

public:

virtual ~IVBF\_3DPlotDoc() {}

public:

// 获取标图文档的名称

virtual const std::string GetName()=0;

// 删除一个或一组军标

virtual void DeleteMark(const std::string& strLayerName, const char\* szMarkID)=0;

virtual void DeleteMark(const std::string& strLayerName, std::vector<std::string>& vsMarkID)=0;

// 通过ID号，在所有图层上查找军标，返回第一个找到的军标

virtual IVBF\_MarkModel\* GetMark(const std::string& strMarkID)=0;

// 清除当前选择的军标

virtual void ClearSelected()=0;

// 图层操作，strLayerName: 图层名

virtual IVBF\_3DPlotLayer\* AddLayer(const std::string& strLayerName,bool bECI = false)=0; // 添加图层

virtual IVBF\_3DPlotLayer\* GetLayer(const std::string& strLayerName)=0;// 查找图层

virtual IVBF\_3DPlotLayer\* GetActiveLayer()=0;

virtual void ClearAllLayers()=0;

virtual void DeleteLayer(const std::string& strLayerName)=0; // 删除图层

virtual void ActivateLayer(const std::string& strLayerName)=0; // 激活图层

virtual int GetLayerCount()=0;

virtual void GetAllLayers(std::vector<IVBF\_3DPlotLayer\*>& vLayers)=0; // 获取所有图层

virtual void GetAllLayerNames(std::vector<std::string>& vLayerNames)=0;// 获取所有图层名

virtual void AddMarkToLayer(const std::string& strLayerName, IVBF\_MarkModel\* pMarkModel)=0;

// 文档是否激活

virtual bool IsActive()=0;

};

##### 4）IVBF\_3DPlotLayer接口

标图图层接口。主要负责控制图层中的各类实体模型，包括模型的添加、删除、显示开关、显示距离等。

class IVBF\_3DPlotLayer : public osg::CVBF\_Object

{

public:

virtual ~IVBF\_3DPlotLayer() {}

public:

/\* 获取图层所在的标图文档指针 \*/

virtual DWORD GetDocPtr()=0;

/\* 设置/获取图层可视的最小距离，默认为0.0 (单位：米) \*/

virtual void SetDistMin(float fDistMin)=0;

virtual float GetDistMin()=0;

/\* 设置/获取图层可视的最大距离，默认为FLT\_MAX (单位：米) \*/

virtual void SetDistMax(float fDistMax)=0;

virtual float GetDistMax()=0;

virtual void SetVisible(bool bVisible)=0; // 设置图层是否显示

virtual bool IsVisible()=0; // 判断图层是否可见

// 图层是否激活

virtual bool IsActive()=0;

virtual const std::string& GetLayerName()=0; // 获取图层名

virtual void AlterLayerName(const std::string& strLayerName)=0;// 更改图层名

// 添加模型

virtual void AddMark(IVBF\_MarkModel\* pIMarkModel)=0;

// 删除模型

virtual void DeleteMark(const std::string& strMarkID)=0;

virtual void DeleteMark(IVBF\_MarkModel\* pIMarkModel)=0;

virtual void DeleteAllMarks()=0;

// 移除模型（不删除模型的指针，注意：返回值IVBF\_MarkModel\*在很短时间之后引用计数将为0，如果外部想使用该指针必须增加引用计数）

virtual IVBF\_MarkModel\* RemoveMark(const std::string& strMarkID)=0;

virtual void RemoveMark(IVBF\_MarkModel\* pIMarkModel)=0;

// 查找模型

virtual IVBF\_MarkModel\* GetMark(const std::string& strMarkID)=0;

// 获取该层上的所有军标模型

virtual void GetAllMarks(std::vector<IVBF\_MarkModel\*>& vMarks)=0;

// 选择单个模型 (如果没选中，返回NULL)

virtual IVBF\_MarkModel\* SelectMark(const std::string& strMarkID, bool bClear=true)=0;

// 选择多个模型 (如果一个都没选中，返回false)

virtual bool SelectMark(const std::vector<std::string>& vMarkIDList, bool bClear=true)=0;

// 获取当前所有被选中的目标

virtual void GetSelectedMarks(std::vector<IVBF\_MarkModel\*>& vSelMarks)=0;

virtual void ClearSelectedMark()=0; // 清空选择集

virtual void DeleteSelectedMark()=0; // 删除选择的军标

// 是否为内部图层（默认为false）

virtual bool IsInnerLayer()=0;

virtual void SetInnerLayer(bool bInnerLayer)=0;

// 是否允许串行化（默认为true）

virtual bool IsAllowSerialize()=0;

virtual void SetAllowSerialize(bool bAllow)=0;

// 向图层中挂接/移除一个节点，注意：该节点必须为军标模型的根节点

virtual void AddSGNode(osg::IVBF\_SGNode\* pNode)=0;

virtual void RemoveSGNode(osg::IVBF\_SGNode\* pNode)=0;

// 绑定/解除一个消息观察者（负责接收该图层发出的各类消息）

virtual void AttachObserver(IVBF\_ObserverRef\* pIObserver)=0;

virtual void DetachObserver(IVBF\_ObserverRef\* pIObserver)=0;

// 获取图层的场景图根节点

virtual osg::IVBF\_SGNodeGroup\* GetRootNode()=0;

};

##### 5）IVBF\_MarkModel接口

标图实体模型接口，是所有实体模型的共同基类。模型可通过IVBF\_3DPlot:: CreateMarkModel()函数创建，且只有在添加到图层后才可见。创建一个模型并添加到图层的示例代码如下：

// 创建一个编号为1009002的模型pIMark，并添加到图层pILayer

IVBF\_MarkModelMesh\* pIMark = (IVBF\_MarkModelMesh\*)pI3DPlot->CreateMarkModel(strMarkID, "1009002");

if(pIMark==NULL) return;

// 设置模型的地理坐标、缩放系数等参数

// ......

// 添加到图层

pILayer->AddMark(pIMark);

该接口的定义如下：

class IVBF\_MarkModel : public osg::CVBF\_Object

{

public:

virtual ~IVBF\_MarkModel() {}

public:

/\* 设置军标的类型编码

参数：strCode -- 军标模型的类型编码(建议使用16位：前8位为主码，后8位为子码)

\*/

virtual void SetCode(const std::string& strCode)=0;

/\* 获取军标的类型编码 \*/

virtual const std::string& GetCode()=0;

/\* 获取军标模型的类型 (军标模型的类型参见枚举VBF\_MARK\_TYPE的定义)\*/

virtual unsigned int GetType()=0;

/\* 获取军标模型的类型描述，以便用户理解 \*/

virtual const std::string& GetTypeDesc()=0;

/\* 获取军标模型的全层唯一编号 \*/

virtual const std::string& GetID()=0;

/\* 设置军标模型的全层唯一编号，如果该模型已经记录某个图层中，不允许修改ID号，返回false（请用户慎重使用该函数） \*/

virtual bool SetID(const std::string& strID)=0;

/\* 判断军标模型是否为点状实体模型 (即是否通过唯一的坐标点定位) \*/

virtual bool IsPointMarkModel()=0;

/\* 军标模型的重要性级别 (数值越小，级别越高) \*/

virtual void SetLevel(DWORD nLevel)=0;

virtual DWORD GetLevel()=0;

/\* 军标模型可视的最小最大距离，分别默认为0.0和FLT\_MAX (单位：米) \*/

virtual void SetDistMin(float fDistMin)=0;

virtual float GetDistMin()=0;

virtual void SetDistMax(float fDistMax)=0;

virtual float GetDistMax()=0;

// 获取模型到视点的距离（单位：米）

virtual double GetDistToEye()=0;

// 根据模型到视点的距离，实时判断模型是否在可见距离内

virtual bool IsInDistRange()=0;

/\* 军标模型的可见性 \*/

virtual void SetVisible(bool bVisible)=0; // 设置模型是否可见

virtual bool IsVisible()=0; // 判断模型是否可见

/\* 军标模型的选中状态 \*/

virtual void SetSelected(bool bSelected)=0; // 设置模型是否被选中

virtual bool IsSelected()=0; // 判断模型是否被选中

// 设置/获取选中模型时，包围盒是否自动可见（默认为true）

virtual void SetBoundingBoxVisibleWhenSelected(bool bVisible)=0;

virtual bool IsBoundingBoxVisibleWhenSelected()=0;

/\* 设置模型是否存活 \*/

virtual void SetAlive(bool bAlive)=0;

virtual bool IsAlive()=0;

/\* 设置/获取军标模型被选中时包围盒的可见性 \*/

virtual void SetBoundingBoxVisible(bool bVisible)=0;

virtual bool IsBoundingBoxVisible()=0;

// 设置/获取包围盒的样式

virtual void SetBoundingBoxStyle(const VBF\_BOUNDINGBOXSTYLE& style)=0;

virtual const VBF\_BOUNDINGBOXSTYLE& GetBoundingBoxStyle()=0;

/\*\* 获取模型在各个坐标系下的包围盒 \*/

virtual const osg::BoundingBox& GetBoundingBoxInModel()=0;

virtual const osg::BoundingBox& GetBoundingBoxInWorld()=0;

virtual const osg::BoundingBoxd& GetBoundingBoxInGeo()=0;

/\*\* 获取模型包围盒中心的地理坐标 \*/

virtual osg::Vec3d GetBoundingBoxCenterGeo()=0;

/\* 军标模型所在的图层 \*/

virtual void SetLayerPtr(DWORD dwData)=0; // 设置模型所在图层的指针

virtual DWORD GetLayerPtr()=0; // 获取模型所在图层的指针

// 将图层指针设置为NULL（该函数仅由系统内部调用）

virtual void SetLayerPtrAsNull()=0;

/\* 军标模型的文本信息 \*/

virtual void SetInfo(const std::string& strInfo)=0; // 设置模型的文本信息

virtual const std::string& GetInfo()=0; // 获取模型的文本信息

// 设置/获取是否可查询该模型的信息（默认值为true）

virtual void SetQueriable(bool bQueriable)=0;

virtual bool IsQueriable()=0;

// 设置/获取该模型被查询到后的显示样式

virtual void SetQueryStyle(const VBF\_MARKQUERYSTYLE& style)=0;

virtual const VBF\_MARKQUERYSTYLE& GetQueryStyle()=0;

// 设置/获取模型是否可编辑（默认值为true）

virtual void SetEditable(bool bEditable)=0;

virtual bool IsEditable()=0;

/\* 军标模型对应的内存数据 (该数据由外部负责管理，系统内部不负责释放) \*/

virtual void SetData(DWORD dwData)=0;// 设置模型对应的内存数据的首地址

virtual DWORD GetData()=0; // 获取模型对应的内存数据的首地址

// 设置预留标志，系统使用了最后4位

virtual void SetReserveFlag(DWORD dwFlag, bool bOR=true)=0;

virtual DWORD GetReserveFlag()=0;

/\* 获取军标模型当前的位置，如果有父模型，返回子模型的相对坐标；否则，返回模型的世界坐标 \*/

virtual const osg::Vec3d& GetPos()=0;

/\* 获取模型当的世界坐标 \*/

virtual const osg::Vec3d& GetWorldPos()=0;

/\* 设置/获取模型的地理坐标（即：经过局部变换前的地理坐标，该值由用户设置） \*/

virtual void SetGeoPos(const osg::Vec3d& vGeoPos)=0;

virtual const osg::Vec3d& GetGeoPos()=0;

// 获取军标模型实际的地理坐标（即：经过局部变换后的地理坐标，该值由系统内部计算）

virtual osg::Vec3d GetActualGeoPos()=0;

// ------------------------------------------------------------------------------

// 处理父子关系

// ------------------------------------------------------------------------------

/\* 设置军标模型的父模型 (每个模型最多只能有一个父模型)

参数：pIParent -- 父模型的指针

\*/

virtual void SetParent(IVBF\_MarkModel\* pIParent)=0;

/\* 获取军标模型的父模型

返回：pIParent -- 父模型的指针，NULL表示没有父模型

\*/

virtual IVBF\_MarkModel\* GetParent()=0;

// 获取军标模型的父模型层次列表（从最顶层父模型开始，到该模型结束）

virtual std::vector<IVBF\_MarkModel\*> GetParentalPaths()=0;

/\* 为军标模型添加一个子模型

参数：pIChild -- 子模型的指针

不允许Child重复，如果重复则返回false

\*/

virtual bool AddChild(IVBF\_MarkModel\* pIChild)=0;

// 移除一个子模型（不删除子模型的指针）

virtual IVBF\_MarkModel\* RemoveChild(const std::string& strChildID)=0;

virtual void RemoveChild(IVBF\_MarkModel\* pIChild)=0;

// 删除一个子模型（删除子模型的指针）

virtual void DeleteChild(const std::string& strChildID)=0;

virtual void DeleteChild(IVBF\_MarkModel\* pIChild)=0;

/\* 获取军标模型的子模型个数 \*/

virtual int GetChildCount()=0;

/\* 获取军标模型的某个子模型

参数：nIndex -- 该子模型的序号 (可结合GetChildCount()函数实现子模型的遍历)

返回：第n个子模型的指针

\*/

virtual IVBF\_MarkModel\* GetChild(int nIndex)=0;

/\* 获取军标模型的某个子模型

参数：strChildID -- 该子模型的ID号

返回：子模型的指针

\*/

virtual IVBF\_MarkModel\* GetChild(const std::string& strChildID)=0;

/\* 获取军标模型的某个子模型（深度查找所有，即查找子模型的子模型）

参数：strChildID -- 该子模型的ID号

返回：子模型的指针

\*/

virtual IVBF\_MarkModel\* GetChildNested(const std::string& strChildID)=0;

// 分离出指定的子模型

virtual IVBF\_MarkModel\* ApartChild(const std::string& strChildID)=0;

// 获取子模型的地理坐标

virtual osg::Vec3d GetChildGeoPos(const std::string& strChildID)=0;

// 设置标识：该模型的根节点等待添加到某个父节点

virtual void SetParentNode(osg::IVBF\_SGNodeGroup\* pParentNode)=0;

// ------------------------------------------------------------------------------

// 处理模型属性

// ------------------------------------------------------------------------------

// 属性添加/删除/查找

virtual bool AddMarkAttr(IVBF\_MarkAttr\* pIAttr)=0;

virtual void DeleteMarkAttr(unsigned int nAttrType)=0;

virtual IVBF\_MarkAttr\* LookupMarkAttr(unsigned int nAttrType)=0;

virtual IVBF\_MarkAttr\* AddMarkAttr(unsigned int nAttrType)=0;

// 属性遍历

virtual unsigned int GetMarkAttrCount()=0;

virtual IVBF\_MarkAttr\* GetMarkAttr(unsigned int nIndex)=0;

// ------------------------------------------------------------------------------

// 处理模型推演过程

// ------------------------------------------------------------------------------

virtual bool AddMarkProcess(IVBF\_MarkProcess\* pIProcess)=0;

virtual void DeleteMarkProcess(unsigned int nIndex)=0;

virtual unsigned int GetMarkProcessCount()=0;

virtual IVBF\_MarkProcess\* GetMarkProcess(unsigned int nIndex)=0;

virtual void ProcessSimulate(IVBF\_SimulationTimer\* pSimulationTimer)=0;

virtual void StatStartTime(double& dStartTime)=0;

// 判断通过地理坐标是否选中模型

virtual bool SelectByGeoPoint(osg::Vec3d& vCoordGeo)=0;

// 强行开启/关闭深度测试 (默认为开启)

virtual void EnableDepthTest(bool bEnable)=0;

virtual bool IsDepthTestEnabled()=0;

// 强行开启/关闭深度写入 (默认为开启)

virtual void EnableDepthWrite(bool bEnable)=0;

virtual bool IsDepthWriteEnabled()=0;

// 强行开启/关闭光照 (默认为关闭)

virtual void EnableLight(bool bEnable)=0;

virtual bool IsLightEnabled()=0;

// 开启/关闭点光滑 (默认为开启)

virtual void EnablePointSmooth(bool bEnable)=0;

virtual bool IsPointSmoothEnabled()=0;

// 开启/关闭线光滑 (默认为开启)

virtual void EnableLineSmooth(bool bEnable)=0;

virtual bool IsLineSmoothEnabled()=0;

// 开启/关闭面光滑 (默认为关闭，开启后在某些机器上会出现三角形的边线)

virtual void EnablePolygonSmooth(bool bEnable)=0;

virtual bool IsPolygonSmoothEnabled()=0;

// 是否开启地球背面的裁剪，点状模型默认为true，其它默认为false

virtual void EnableClipByEarth(bool bEnable)=0;

virtual bool IsClipByEarthEnabled()=0;

// 设置/获取模型是否随父模型同步混色（默认值为true）

virtual void EnableBlendWithParent(bool bEnable)=0;

virtual bool IsBlendWithParentEnabled()=0;

// 设置/获取模型是否混色以及混合的颜色

virtual void SetBlendColor(bool bEnable, const osg::Vec4f& color=osg::Vec4f(1,1,1,1), bool bModulate=false)=0;

virtual void GetBlendColor(bool& bEnable, osg::Vec4f& color, bool& bModulate)=0;

// 开始/停止模型闪烁

virtual void StartFlicker(int nFlickerTimes, int nFlickerInterval, bool bFlickerAlways=false)=0;

virtual void StopFlicker()=0;

// 设置模型的渲染顺序

virtual void SetRenderOrder\_Opaque(int nOrder)=0; // 不透明模型

virtual void SetRenderOrder\_Transparent(int nOrder)=0; // 透明模型

// 快捷设置模型的渲染顺序

virtual void SetRenderHint\_Opaque()=0; // 不透明模型

virtual void SetRenderHint\_Transparent()=0; // 透明模型

virtual void SetRenderHint\_Default()=0;

// 获取模型的各个场景图节点

virtual osg::IVBF\_SGNodeGroup\* GetRootNode()=0; // 获取模型的场景图根节点

virtual osg::IVBF\_SGNodeGroup\* GetGroupNode()=0;// 获取模型的组节点，所有子节点挂接在此下

virtual osg::IVBF\_SGNode\* GetModelNode()=0;// 获取模型的实际数据节点

/\* 模型做某种动作 (如车轮滚动、炮塔旋转等)

参数：szActionName -- 动作名称（如滚动、旋转等）

bEnabled -- 动作开启或停止

pActionParam -- 动作的参数 (如滚动速度等，具体类型和含义由动作本身决定)

\*/

virtual void DoAction(const char\* szActionName, bool bEnabled, VBF\_MARKACTIONPARAM\* pActionParam)=0;

// 绑定/解除一个消息观察者（负责接收该模型发出的各类消息）

virtual void AttachObserver(IVBF\_ObserverRef\* pIObserver)=0;

virtual void DetachObserver(IVBF\_ObserverRef\* pIObserver)=0;

// 右键弹出菜单

virtual void EnablePopupMenu(bool bEnable)=0;

virtual bool IsPopupMenuEnabled()=0;

virtual void GetPopupMenuItems(std::vector<CVBF\_PopupMenuItem>& vPopupMenuItemInf)=0;

virtual bool ProcessPopupMenuItem(unsigned int nID)=0;

// 是否允许串行化（默认为true）

virtual bool IsAllowSerialize()=0;

virtual void SetAllowSerialize(bool bAllow)=0;

// 串行化

virtual bool WriteToXML(TiXmlElement\* pElemMark)=0; // 写入：将模型信息写入指定的xml节点

virtual bool ReadFromXML(TiXmlElement\* pElemMark)=0; // 读取：从指定的xml节点读取模型信息

// 初始化 (在构造函数之后由系统调用)

virtual bool Init()=0;

// 更新 (在每帧绘制前由系统调用)，返回值为false表示未做更新处理（如模型不可见时）

virtual bool Update()=0;

};

模型可以只有一个定位点，称为“点状模型”（IVBF\_MarkModelPoint），如武器装备模型、二维栅格模型、几何图形模型等；也可以不止一个定位点，称为“非点状模型”（ IVBF\_MarkModelNotPoint），如空域线模型、通联线模型、航迹模型等。

##### 6）IVBF\_MarkModelPoint接口

点状实体模型接口，是所有点状实体模型的共同基类。该类模型只有一个定位点，可以进行旋转、缩放、平移等操作。

class IVBF\_MarkModelPoint : public IVBF\_MarkModel

{

public:

virtual ~IVBF\_MarkModelPoint() {}

public:

/\*\* 设置/获取缩放比，默认为1.0 \*/

virtual void SetScale(float fScaleX, float fScaleY, float fScaleZ)=0;

virtual void GetScale(float& fScaleX, float& fScaleY, float& fScaleZ)=0;

/\*\* 设置固定像素大小 \*/

virtual void SetFixedSizeInPixels(float fFixedSize)=0;

virtual float GetFixedSizeInPixels()=0;

virtual void GetRealtimeScale(float& fScaleX, float& fScaleY, float& fScaleZ)=0;

virtual void GetParentRealtimeScale(float& fScaleX, float& fScaleY, float& fScaleZ)=0;

// 获取系统内部自动计算的缩放比（如统一缩放、自动缩放等）

virtual void GetInnerScale(float& fScaleX, float& fScaleY, float& fScaleZ)=0;

// 设置/获取模型的旋转顺序（默认值为VBF\_ROTATE\_ORDER\_ZXY）

virtual void SetRotateOrder(EVBF\_ROTATE\_ORDER nOrder)=0;

virtual EVBF\_ROTATE\_ORDER GetRotateOrder()=0;

/\*\*

设置/获取模型坐标系下的旋转角，默认值均为0.0

角度说明：x/y/z依次表示翻滚角/俯仰角/航向角(单位：角度)，右手坐标系下，从轴正向往原点看，逆时针为正

\*/

virtual void SetRotate(float fRotateX, float fRotateY, float fRotateZ)=0;

virtual void GetRotate(float& fRotateX, float& fRotateY, float& fRotateZ)=0;

/\*\* 设置/获取模型坐标系下的初始旋转角（用于校正模型的建模朝向，默认值均为0.0）\*\*/

virtual void SetRotateInit(float fRotateX, float fRotateY, float fRotateZ)=0;

virtual void GetRotateInit(float& fRotateX, float& fRotateY, float& fRotateZ)=0;

// 获取所有父模型的旋转矩阵

virtual osg::Matrixd GetParentRotateMatrix()=0;

/\*\* 设置/获取模型坐标系下的平移，默认为0.0 \*/

virtual void SetTrans(float fTransX, float fTransY, float fTransZ)=0;

virtual void GetTrans(float& fTransX, float& fTransY, float& fTransZ)=0;

/\*\* 设置/获取模型坐标系下的初始平移（用于校正模型的建模原点，默认值均为0.0）\*\*/

virtual void SetTransInit(float fTransX, float fTransY, float fTransZ)=0;

virtual void GetTransInit(float& fTransX, float& fTransY, float& fTransZ)=0;

// 是否允许设置固定大小 (单位：像素)

virtual void EnableFixedSizeInPixels(bool bEnable)=0;

virtual bool IsFixedSizeInPixelsEnabled()=0;

// 是否允许自动调整大小

virtual void EnableAutoScale(bool bEnable)=0;

virtual bool IsAutoScaleEnabled()=0;

// 是否允许实时调整缩放比 (即能否通过热键实现统一缩放)

virtual void EnableScaleDelta(bool bEnable)=0;

virtual bool IsScaleDeltaEnabled()=0;

// 是否启用地形匹配

virtual void EnableMatchTerrain(bool bEnable)=0;

virtual bool IsMatchTerrainEnabled()=0;

// 是否随父模型旋转，默认为true

virtual void EnableRotateWithParent(bool bEnable)=0;

virtual bool IsRotateWithParentEnabled()=0;

// 是否随父模型缩放，默认为true

virtual void EnableScaleWithParent(bool bEnable)=0;

virtual bool IsScaleWithParentEnabled()=0;

// 获取模型坐标系与世界坐标系之间的转换矩阵

virtual osg::Matrixd GetModelToWorldMatrix()=0;

virtual osg::Matrixd GetWorldToModelMatrix()=0;

// 把点坐标由世界坐标系变换到模型坐标系

virtual osg::Vec3d PointWorldToModel(const osg::Vec3d& ptWorld)=0;

// 计算从模型原点到模型中某点的向量（模型坐标系下）与地球的交点，返回交点及其与模型原点的距离（世界坐标系下）

virtual bool Vector0ToPointInModelIntersectWithEarth(const osg::Vec3d& vPointInModel, double dHeightAdjust, osg::Vec3d& ptIntersect, double& dDist)=0;

// 判断某点（世界坐标系下）是否在模型范围内

virtual bool Contains(const osg::Vec3d& ptWorld)=0;

virtual bool Contains(const osg::Vec3d& ptWorld, osg::Vec3d& ptModel, double& dDist)=0;

/\*\* 设置模型坐标系下累加的旋转角/缩放比/平移值（在当前值的基础上） \*/

virtual void SetDeltaRotate(float fDeltaRotateX, float fDeltaRotateY, float fDeltaRotateZ)=0;

virtual void SetDeltaScale(float fDeltaScaleX, float fDeltaScaleY, float fDeltaScaleZ)=0;

virtual void SetDeltaTrans(float fDeltaTransX, float fDeltaTransY, float fDeltaTransZ)=0;

virtual void SetDeltaTransGeo(float fDeltaTransX, float fDeltaTransY, float fDeltaTransZ)=0;

/\*\* 恢复模型坐标系下模型初始状态，即无任何旋转角/缩放比/平移值 \*/

virtual void SetUnTransformed()=0;

// 获取模型本身的旋转矩阵

virtual osg::Matrixd GetRotateMatrix()=0;

/\*\* 是否直接使用外部设置的旋转矩阵（模型坐标系下） \*/

virtual void SetRotateMatrixOut(const osg::Matrixd& mat)=0;

// 获取点状模型的各个场景图节点

virtual osg::IVBF\_SGNodeGroupTransformMatrix\* GetModelToWorldNode()=0;// 获取模型的定位节点，即模型坐标系->世界坐标系

virtual osg::IVBF\_SGNodeGroupTransformMatrix\* GetModelTransformNode()=0; // 获取模型的局部变换节点

// 设置/获取仿真时间方式

virtual void SetSimTimeMode(EVBF\_SIMTIME\_MODE nMode)=0;

virtual EVBF\_SIMTIME\_MODE GetSimTimeMode()=0;

// 定长方式下，设置/获取仿真时间步长，即发送相邻两个坐标数据之间的时间间隔 (只适用于定长仿真方式，单位：秒)

virtual void SetSimTimeStep(float fStep)=0;

virtual float GetSimTimeStep()=0;

/\* 设置军标模型当前的位置(地理坐标)和姿态，同时对坐标和角度进行插值

参数：vGeoPos -- 模型当前的位置(地理坐标)

fRotateX -- 模型当前的滚转角，即模型从初始化状态起绕X轴旋转的角度，逆时针为正 (单位：角度，值域为[-180, 180])

fRotateY -- 模型当前的俯仰角，即模型从初始化状态起绕Y轴旋转的角度，逆时针为正 (单位：角度，值域为[-90, 90])

fRotateZ -- 模型当前的偏航角，即模型从初始化状态起绕Z轴旋转的角度，逆时针为正 (单位：角度，值域为[-180, 180])

strInfo -- 模型在该位置处的信息，以便在轨迹点中查询（可以为空字符串）

坐标系说明：右手坐标系，从轴正向往原点看，逆时针为正

\*/

virtual void StepGeoPos(const osg::Vec3d& vGeoPos, float fRotateX, float fRotateY, float fRotateZ, const std::string& strInfo="")=0;

// 获取军标模型在当前帧的轨迹点信息（该信息通过StepGeoPos()函数设置，如果模型尚未开始运动，返回false）

virtual bool GetTrailPoint(VBF\_TRAILPOINT& ptTrail)=0;

// 设置/获取模型的观察模式

virtual void SetViewMode(EVBF\_VIEW\_MODE nMode)=0;

virtual EVBF\_VIEW\_MODE GetViewMode()=0;

};

##### 7）IVBF\_MarkModelNotPoint接口

非点状实体模型接口，是某些非点状实体模型（如箭头军标）的共同基类。该类模型不止一个定位点，不可进行旋转、缩放、平移等操作，但可对控制点进行编辑。

class IVBF\_MarkModelNotPoint : public IVBF\_MarkModel

{

public:

virtual ~IVBF\_MarkModelNotPoint() {}

public:

// 设置/获取控制点的地理坐标，用于编辑

virtual void SetCtrlPointsGeo(const std::vector<osg::Vec3d>& vPointsGeo) = 0;

virtual std::vector<osg::Vec3d>& GetCtrlPointsGeo() = 0;

virtual bool IsSameHeight() = 0;

virtual void setHeight(double dHeight)=0;// 对于需要设置整体高度的模型来说，需要这个接口

virtual double getHeight()=0;

// 获得全局编辑的控制点（经纬度坐标）

virtual osg::Vec3d GetLocationGeoPoint()=0;

};

##### 8）IVBF\_MarkAttr接口

实体模型的属性接口，是所有实体模型属性的共同基类。属性是实体所具有的状态，为了使其结构和配置更灵活，采用了属性概念，可以使用XML文档配置实体属性。每个模型可以配置多个属性，每个属性也可以配置给多个模型。常见的属性有敌我标示（IFF）、生命期（LifeTime）、标签（Label）、轨迹线（TrailLine）、导弹尾焰（MissileFire）、高度指示线（HintLine）等。用户可根据需要动态添加或删除某些属性。

属性接口可通过两种方法创建：

方法一、通过IVBF\_3DPlot:: CreateMarkAttr()函数创建一个属性接口，然后将该接口添加给某个模型。示例代码如下：

// 为模型pIMark创建并添加一个标签属性

IVBF\_MarkAttrLabel\* pIAttrLabel = dynamic\_cast<IVBF\_MarkAttrLabel\*>(pI3DPlot->CreateMarkAttr(VBF\_MARK\_ATTR\_LABEL));

if(pIAttrLabel!=NULL) pIMark->AddMarkAttr(pIAttrLabel);

方法二：通过IVBF\_MarkModel::AddMarkAttr()函数直接创建并添加一个属性接口。示例代码如下：

// 为模型pIMark创建并添加一个标签属性

IVBF\_MarkAttrLabel\* pIAttrLabel = dynamic\_cast<IVBF\_MarkAttrLabel\*>(pIMark->AddMarkAttr(VBF\_MARK\_ATTR\_LABEL));

该接口定义如下：

class IVBF\_MarkAttr : public osg::CVBF\_Object

{

public:

virtual ~IVBF\_MarkAttr() {}

public:

// 获取属性类型

virtual unsigned int GetType()=0;

// 获取属性类型的描述，以便用户理解

virtual const std::string& GetTypeDesc()=0;

// 设置/获取可见性

virtual void SetVisible(bool bVisible)=0;

virtual bool IsVisible()=0;

// 设置/获取属性所绑定的实体模型

virtual void SetMarkModel(IVBF\_MarkModel\* pIMark)=0;

virtual IVBF\_MarkModel\* GetMarkModel()=0;

// 设置/获取属性最小最大可见距离，如果没有设置，默认使用属性所绑定的实体模型的可见距离

// 如果获取的返回值为false，表示用户没有设置属性的最小最大可见距离

virtual void SetDistMin(float fDistMin)=0;

virtual bool GetDistMin(float& fDistMin)=0;

virtual void SetDistMax(float fDistMax)=0;

virtual bool GetDistMax(float& fDistMax)=0;

// 复制

virtual IVBF\_MarkAttr\* Clone()=0;

// 设置属性参数列表

virtual bool ExecuteCommand(const char\* szCommandID, IVBF\_PropertySet\* pInParam, IVBF\_PropertySet\* pOutParam)=0;

// 右键弹出菜单

virtual void GetPopupMenuItems(std::vector<CVBF\_PopupMenuItem>& vItems)=0;

virtual bool ProcessPopupMenuItem(unsigned int nID)=0;

// 串行化

virtual bool WriteToXML(TiXmlElement\* pElemAttr)=0; // 写入：将模型属性信息写入指定的xml节点

virtual bool ReadFromXML(TiXmlElement\* pElemAttr)=0; // 读取：从指定的xml节点读取模型属性信息

// 初始化 (在构造函数之后由系统调用)

virtual bool Init()=0;

// 更新 (在每帧绘制前由系统调用)，返回值为false表示未做更新处理（如模型不可见时）

virtual bool Update()=0;

};

##### 9）IVBF\_MarkProcess接口

实体模型的过程接口，是所有实体模型过程的共同基类。过程表示时间段触发的过程，如路径运动、惯性运动、分离、爆炸等。用户可根据需要动态添加或删除某些过程。

创建并添加一个过程接口的示例代码如下：

// 为模型pIMark创建并添加一个路径运动过程

IVBF\_MarkProcessMove\* pIProcessMove = (IVBF\_MarkProcessMove\*)pI3DPlot->CreateMarkProcess(VBF\_MARK\_PROCESS\_MOVE);

if(pIProcessMove!=NULL) pIMark->AddMarkProcess(pIProcessMove);

该接口的定义如下：

class IVBF\_MarkProcess : public osg::CVBF\_Object

{

public:

virtual ~IVBF\_MarkProcess() {}

public:

// 获取过程类型

virtual int GetType()=0;

// 获取过程的类型描述

virtual const std::string& GetTypeDesc()=0;

// 设置过程所绑定的实体模型

virtual void SetMarkModel(IVBF\_MarkModel\* pIMarkModel)=0;

virtual void ProcessSimulate(IVBF\_SimulationTimer\* pSimulationTimer)=0; // 过程模拟

//统计起始时间,为了统计最早的过程的时间，仅统计绝对时间而且仅统计时间区段，相对时间时返回0

virtual void StatStartTime(double& dStartTime)=0;

// 串行化

virtual bool WriteToXML(TiXmlElement\* pElemProc)=0; // 写入：将模型过程信息写入指定的xml节点

virtual bool ReadFromXML(TiXmlElement\* pElemProc)=0; // 读取：从指定的xml节点读取模型过程信息

};