Chapter 5

3D Application

5.1 Introduction

When all tools of the station were ready, it was the time to test the station. But it is impossible to test the station directly with an UAV, it will be expensive and unsafe, for this reason it was decided to make a new simulation tool. The aim is to get a 3D scene with some UAVs flying in it, and with tools previously seen to schedule new functions (some of them like we saw in the last chapter) for our station. In that way we can simulate the situation of an operator control and monitoring one or more UAVs.

We need a Software tool that allows us to work with 3D graphics to get a simulation nearer to the reality that we can. The first thought was OpenGL (Open Graphics Library), which is a standard specification defining a cross-language, cross-platform API for writing applications that produce 2D and 3D computer graphics. The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation. It is also used in video games, where it competes with Direct3D on Microsoft Windows platforms (see Direct3D vs. OpenGL). OpenGL is managed by the non-profit technology consortium, the Khronos Group.

The OpenGL API (Application Programming Interface) began as an initiative by SGI to create a single, vendor-independent API for the development of 2D and 3D graphics applications. Prior to the introduction of OpenGL, many hardware vendors had different graphics libraries. This situation made it expensive for software developers to support versions of their applications on multiple hardware platforms, and it made porting of applications from one hardware platform to another very time-consuming and difficult. SGI saw the lack of a standard graphics API as an inhibitor to the growth of the 3D marketplace and decided to lead an industry group in

creating such a standard.

The result of this work was the OpenGL API, which was largely based on earlier work on the SGIő IRIS GLŹ library. The OpenGL API began as a specification, then, SGI produced a sample implementation that hardware vendors could use to develop OpenGL drivers for their hardware. The sample implementation has been released under an open source license.

OpenGL could be the solution, but more recently has emerged other cross-platform graphics toolkit called OpenSceneGraph, it is based on OpenGL, but in a higher level. Now we are going to see the main futures and an overview about OpenSceneGraph.

5.2 OpenSceneGraph

The OpenSceneGraph is an OpenSource, cross-platform graphics toolkit for the development of high-performance graphics applications such as flight simulators, games, virtual reality and scientific visualization. It is based around the concept of a SceneGraph, providing an object-oriented framework on top of OpenGL. This frees the developer from implementing and optimizing low-level graphics calls and provides many additional utilities for rapid development of graphics applications. We will start with an introduction to scene graphs and then we will see important features of OpenSceneGraph, more information is available in the official website [15].

Figure 5.1: OSG logo

5.2.1 Introduction to Scene Graphs

A scene graph is a hierarchical tree data structure that organizes spatial data for efficient rendering. Figure 5.2 illustrates an abstract scene graph consisting of terrain, a cow, and a truck.

The scene graph tree is headed by a top-level root node. Beneath the root node, group nodes organize geometry and the rendering state that controls their appearance. Root nodes and group nodes can have zero or more children. (However, group nodes with zero children are essentially no-ops.) At the bottom of the scene graph, leaf nodes contain the actual geometry that make up the objects in the scene.

Applications use group nodes to organize and arrange geometry in a scene. Imagine a 3D database containing a room with a table and two

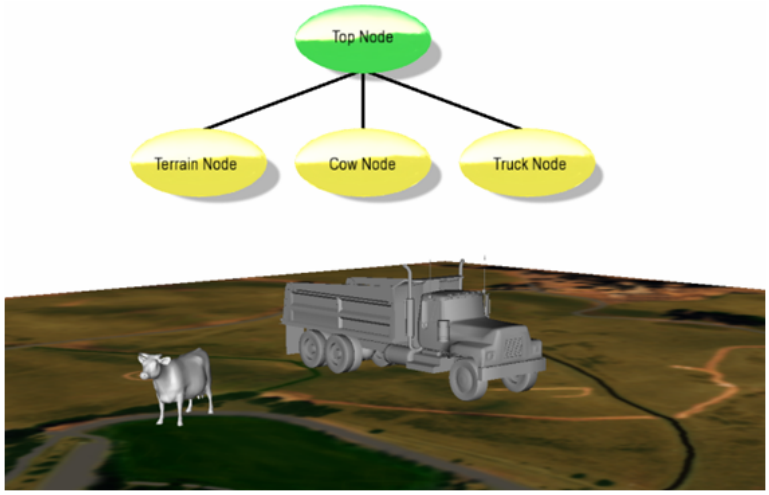

Figure 5.2: Scene Graph example

identical chairs. You can organize a scene graph for this database in many ways. Figure 5.3 shows one example organization. The root node has four group node children, one for the room geometry, one for the table, and one for each chair. The chair group nodes are color-coded red to indicate that they transform their children. There is only one chair leaf node because the two chairs are identical their parent group nodes transform the chair to two different locations to produce the appearance of two chairs. The table group node has a single child, the table leaf node. The room leaf node contains the geometry for the floor, walls, and ceiling.

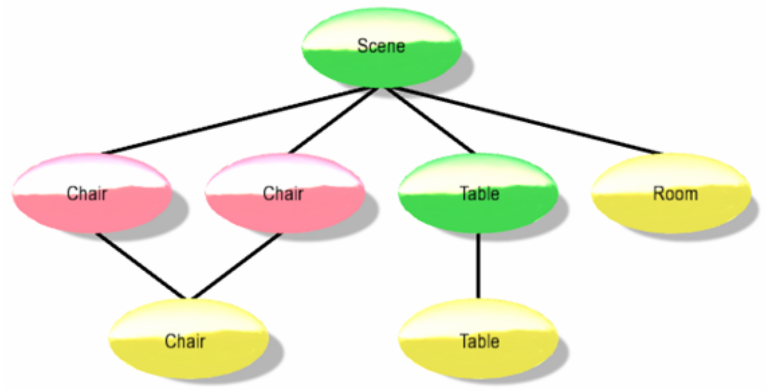

Figure 5.3: A typical scene graph

Scene graphs usually offer a variety of different node types that offer a wide range of functionality, such as switch nodes that enable or disable their children, level of detail (LOD) nodes that select children based on distance

from the viewer, and transform nodes that modify transformation state of child geometry. Object-oriented scene graphs provide this variety using inheritance; all nodes share a common base class with specialized functionality defined in the derived classes.

The large variety of node types and their implicit spatial organization ability provide data storage features that are unavailable in traditional low-level rendering APIs. OpenGL and Direct3D focus primarily on abstracting features found in graphics hardware. Although graphics hardware allows storage of geometric and state data for later execution (such as display lists or buffer objects), low-level API features for spatial organization of that data are generally minimal and primitive in nature, and inadequate for the vast majority of 3D applications. Scene graphs are middleware, which are built on top of low-level APIs to provide spatial organization capabilities and other features typically required by highperformance 3D applications. Figure 5.4 illustrates a typical OSG application stack.

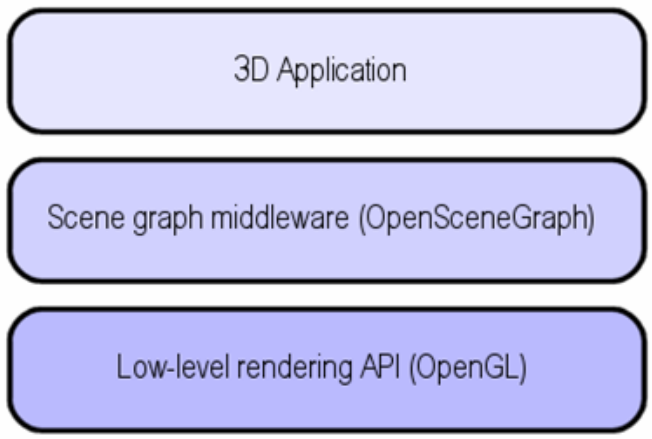

Figure 5.4: The 3D Applications stack

5.2.2 Scene Graph Features

Scene graphs expose the geometry and state management functionality found in lowlevel rendering APIs, and provide additional features and capabilities, such as the following:

- Spatial organization: The scene graph tree structure lends itself naturally to intuitive spatial organization.

- Culling: View frustum and occlusion culling on the host CPU typically reduces overall system load by not processing geometry that doesn't appear in the final rendered image.

- LOD: Viewer-object distance computation on bounding geometry allows objects to efficiently render at varying levels of detail. Furthermore, portions of a scene can load from disk when they are within a specified viewer distance range, and page out when they are beyond that distance.

- Translucent: Correct and efficient rendering of translucent (non-opaque) geometry requires all translucent geometry to render after all opaque geometry. Furthermore, translucent geometry should be sorted by depth and rendered in back-to-front order. These operations are commonly supported by scene graphs.

- State change minimization: To maximize application performance, redundant and unnecessary state changes should be avoided. Scene graphs commonly sort geometry by state to minimize state changes, and OpenSceneGraph's state management facilities eliminate redundant state changes.

- File I/O-Scene graphs are an effective tool for reading and writing 3D data from disk. Once loaded into memory, the internal scene graph data structure allows the application to easily manipulate dynamic 3D data. Scene graphs can be an effective intermediary for converting from one file format to another.

- Additional high-level functionality: Scene graph libraries commonly provide high-level functionality beyond what is typically found in low-level APIs, such as full-featured text support, support for rendering effects (such as particle effects and shadows), rendering optimizations, 3D model file I/O support, and cross-platform access to input devices and render surfaces.

Nearly all 3D applications require some of these features. As a result, developers who build their applications directly on low-level APIs typically resort to implementing many of these features, which increases development costs. Using an off-the-shelf scene graph that already fully supports such features enables rapid application development.

5.2.3 How Scene Graph render

A trivial scene graph implementation allows applications to store geometry and execute a draw traversal, during which all geometry stored in the scene graph is sent to the hardware as OpenGL commands. However, such an implementation lacks many of the features described in the previous section. To allow for dynamic geometry updates, culling, sorting, and efficient rendering, scene graphs typically provide more than a simple draw traversal. In general, there are three types of traversals:

- **Update:** The update traversal (sometimes referred to as the application traversal) allows the application to modify the scene graph, which enables dynamic scenes. Updates are accomplished either directly by the application or with callback functions assigned to nodes within the scene graph. Applications use the update traversal to modify the position of a flying aircraft in a flight simulation, for example, or to allow user interaction using input devices.
- **Cull:** During the cull traversal, the scene graph library tests the bounding volumes of all nodes for inclusion in the scene. If a leaf node is within the view, the scene graph library adds leaf node geometry references to a final rendering list. This list is sorted by opaque versus translucent, and translucent geometry is further sorted by depth.
- **Draw:** In the draw traversal (sometimes referred to as the render traversal), the scene graph traverses the list of geometry created during the cull traversal and issues low-level graphics API calls to render that geometry. OSG includes a fourth traversal, the event traversal, which processes input and other events each frame, just before the update traversal.

Figure 5.5 illustrates these traversals.

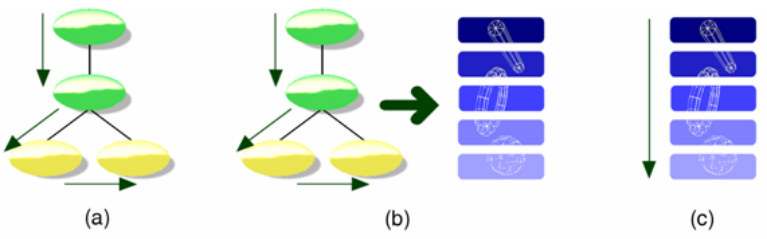

Figure 5.5: Scene graph traversals

Like we can see in the figure above, rendering a scene graph typically requires three traversals. In (a), the update traversal modifies geometry, rendering state, or node parameters to ensure the scene graph is up-to-date for the current frame. In (b), the cull traversal checks for visibility, and places geometry and state references in a new structure (called the render graph in OSG). In (c), the draw traversal traverses the render graph and issues drawing commands to the graphics hardware.

Typically, these three traversals are executed once for each rendered frame. However, some rendering situations require multiple simultaneous views of the same scene. Stereo rendering and multiple display systems are two examples. In these situations, the update traversal is executed once per frame, but the cull and draw traversals execute once per view per frame. (That's twice per frame for simple stereo rendering, and once per graphics

card per frame on multiple display systems.) This allows systems with multiple processors and graphics cards to process the scene graph in parallel. The cull traversal must be a read-only operation to allow for multithreaded access.

5.2.4 Overview of OpenSceneGraph

OSG is a set of open source libraries that primarily provide scene management and graphics rendering optimization functionality to applications. It's written in portable ANSI C++ and uses the industry standard OpenGL low-level graphics API. As a result, OSG is cross platform and runs on Windows, Mac OS X, and most UNIX and Linux (our case) operating systems. Most of OSG operates independently of the native windowing system. However, OSG includes code to support some windowing system specific functionality, such as input devices, window creation, and Buffers. OSG is open source and is available under a modified GNU Lesser General Public License, or Library GPL (LGPL) software license. OSG's open source nature has many benefits:

- Improved quality: OSG is reviewed, tested, and improved by many members of the OSG community. Over 250 developers contributed to OSG v2.0.

- Improved application quality: To produce quality applications, application developers need intimate knowledge of the underlying middleware. If the middleware is closed source, this information is effectively blocked and limited to vendor documentation and customer support. Open source allows application developers to review and debug middleware source code, which allows free access to code internals.

- Reduced cost: Open source is free, eliminating the up-front purchase price.

- No intellectual property issues: There is no way to hide software patent violations in code that is open source and easily read by all.

Design and Architecture

OSG is designed up front for portability and scalability. As a result, it is useful on a wide variety of platforms, and renders efficiently on a large number and variety of graphics hardware. OSG is designed to be both flexible and extensible to allow adaptive development over time. As a result, OSG can meet customer needs as they arise.

To enable these design criteria, OSG is built with the following concepts and tools:

- ANSI standard C++
- C++ Standard Template Library (STL)
- Design patterns [Gamma95]

These tools allow developers using OSG to develop on the platform of their choice and deploy on any platform the customer requires.

Components

The OSG runtime exists as a set of dynamically loaded libraries (or shared objects) and executables. These libraries fall into five conceptual categories:

- The Core OSG libraries provide essential scene graph and rendering functionality, as well as additional functionality that 3D graphics applications typically require.
- NodeKits extend the functionality of core OSG scene graph node classes to provide higher-level node types and special effects.
- OSG plugins are libraries that read and write 2D image and 3D model files.
- The interoperability libraries allow OSG to easily integrate into other environments, including scripting languages such as Python and Lua.
- An extensive collection of applications and examples provide useful functionality and demonstrate correct OSG usage.

Like we can see in the Figure 5.6, The Core OSG libraries provide functionality to both the application and the NodeKits. Together, the Core OSG libraries and NodeKits make up the OSG API. One of the Core OSG libraries, osgDB, provides access to 2D and 3D file I/O by managing the OSG plugins.

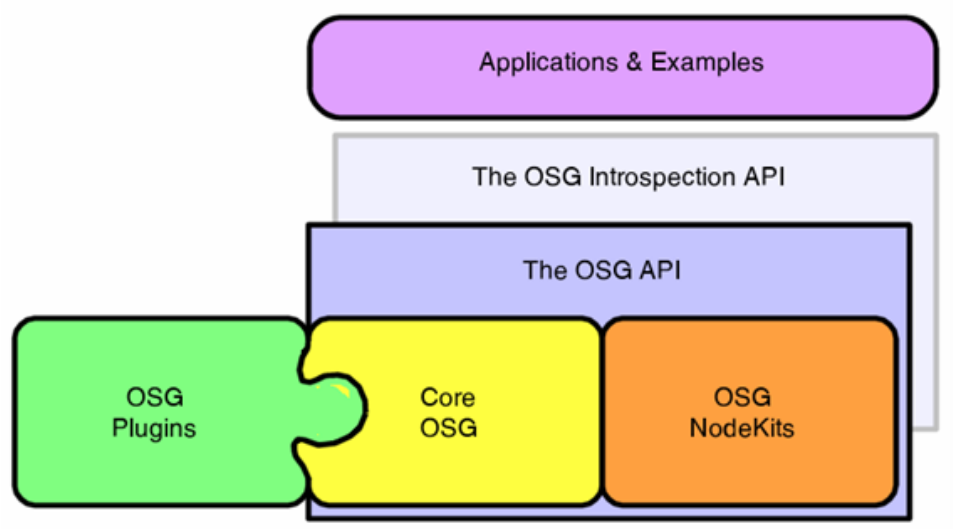

Figure 5.6: OSG architecture

OpenSceneGraph has many more features, but it is impossible to include them in the memory and nor is the goal. In the next section will review all the steps in the development of the simulation station.

5.3 Development of the simulation tool

In this section will see step by step all the development process associated to the simulation tool. With this aim, some features of OpenSceneGraph, that were not exposed before and are necessary to built the tool, will be explained when programming needed.

5.3.1 First steps

The first step was to install OpensceneGraph in the Station PC. The station uses Ubuntu Linux like operating system, so it was necessary to install a compatible version of OpenSceneGraph with Linux. But, before install it, it is necessary to build the dependencies. In Debian GNU/Linux is possible to make use of the pre-packaged binaries and simply type in a console **apt-get build-dep openscenegraph**, to download and install the necessary dependencies. Moreover the dependencies, is necessary to download OpenSeceneGraph standard dataset for use with examples too. In this case OpenSceneGraph-Data-2.8.0.zip was downloaded from http://www.openscenegraph.org/projects/osg/wiki/Downloads/SampleDatasets. With these files we can execute the OSG examples.

After the installation of the dependencies, I downloaded the OpenSceneGraph from http://www.openscenegraph.org/projects/osg/wiki/Downloads. The downloaded openSceneGraph version was OpenSceneGraph-2.8.2 that is a stable version released 28th July 2009. After performing a file decompression (source packet is a .zip), I opened a console, looked for the folder with OSG and entered the following commands:
./configure
Make
Sudo make install

When instructions associated to the last command finished, OSG was successful install in the PC. Next step was to run the example applications. First we need to set up the path to the applications, and it is recommended to add these environmental settings to the station PC login setup such us .bashrc under Unix. The binaries paths to set up are:

- export PATH:/home/caba/OpenSceneGraph2.8.2/bin:/opt/Adobe/Reader8/bin:/home/caba/OpenSceneGraph-2.8.2/lib
- export OSG_FILE_PATH=/home/caba/OpenSceneGraph-Data-2.8.0

Now, it is possible to run a simple a simple application type in a console:

osgviewer cow.osg

Above command result is illustrated in the Figure 5.7:

Osgviewer is an OSG's flexible and powerful model viewing tool and is a fundamental tool in the present project because will allow us to represent the

Figure 5.7: Osgviewer output

3D scene with UAVs. Viewer instantiates an **osgViewer::Viewer** object, attaches a scene graph to it, and allows it to render. In the picture above we loaded a simple model of a cow and display it.

The cow model is in OSG's own .osg file format. However, **osgViewer** supports the same file formats as OSG, many of which are enumerated in the **OSG Plugins** section later in this chapter. **OsgViewer** lets you interact with the model. By default, **osgViewer** exposes a trackball-like interface (there are another interface that will be seen later in the memory). To rotate the cow model, drag with your left mouse button. When you release the mouse button, the model continues to rotate. You can zoom in or out using the right mouse button. Press the space bar to return to the initial view.

Once tested with the OSG examples, is time to start the scheduling of our scene. First aim was to build a scene with a 3D terrain and a plane. For that I was based on the examples of OSG official website. First of all I built the associated scene graph (see Figure 5.8).

Like we can see in this example is very simple. Only need a RootNode and two childs of the RootNode, the Plane and the terrain. Due I only wanted to a static scene the child nodes could be simple osg::Nodes, later will be necessary to do actions over these nodes (like to move the plane around the terrain) and these nodes will be of a different kind in function of these actions. Then, I associated an osgViewer to the Rootnode and when the program was executed the result is illustrated in Figure 5.9.

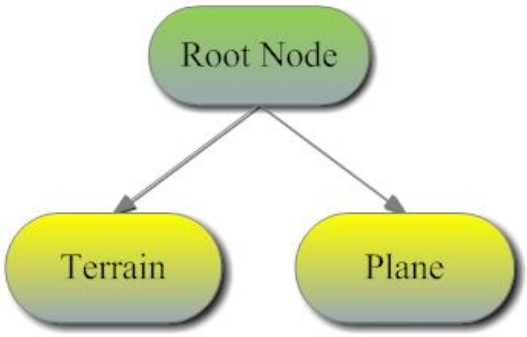

Figure 5.8: Basic Scene Graph

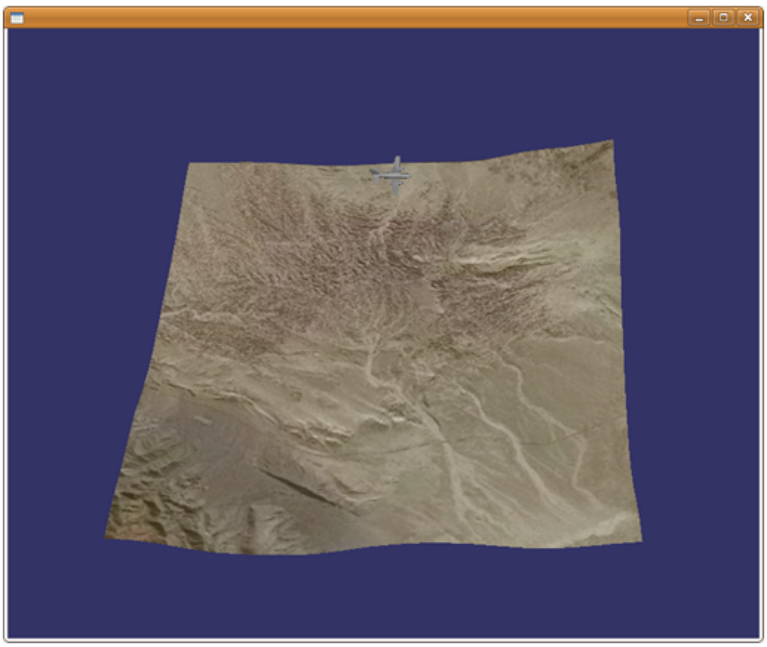

Figure 5.9: Viewer with plane and terrain

And here is the first scene in OSG. The terrain is a Microsoft Flight Simulator file (.flt) and the plane is an .osg file called "Cessna.osg". OSG can load these models thanks to osgDB library.

The osgDB library allows applications to load, use, and write 3D databases. The osgDB plugin architecture provides support for a wide variety of common 2D image and 3D model file formats. The osgDB maintains a registry of and oversees access to the loaded OSG plugins. OSG supports its own file formats. .osg is a plain ASCII text description of a scene graph, and .osga is an archive (or group) of .osg files. The osgDB library contains support code for these file formats. (OSG also supports a binary .ive format.)

Large 3D terrain databases are often created in sections that tile together. In this case, applications require that portions of the database load from file in a background thread without interrupting rendering. The **osgDB::DatabasePager** provides this functionality.

5.3.2 Moving the plane

Once achieved the static scene, next logic step was to get a dynamic scene. Now, the aim is to move the plane around the terrain. For that, we need to define a new node type that can be rotated and translated. Is the time to see the different scene graph classes present in OSG.

Scene Graph Classes

Scene graph classes aid in scene graph construction. All scene graph classes in OSG are derived from **osg::Node**. Conceptually, root, group, and leaf nodes are all different node types. In OSG, these are all ultimately derived from **osg::Node**, and specialized classes provide varying scene graph functionality. Also, the root node in OSG is not a special node type; it's simply an **osg::Node** that does not have a parent.

- **Node**: The **Node** class is the base class for all nodes in the scene graph. It contains methods to facilitate scene graph traversals, culling, application callbacks, and state management.

- **Group**: The **Group** class is the base class for any node that can have children. It is a key class in the spatial organization of scene graphs.

- **Geode**: The **Geode** (or Geometry Node) class corresponds to the leaf node in OSG. It has no children, but contains **osg::Drawable** objects (see below) that contain geometry for rendering.

- **LOD**: The **LOD** class displays its children based on their distance to the view point. This is commonly used to create a varying levels of detail for objects in a scene.

- **MatrixTransform**: The **MatrixTransform** class contains a matrix that transforms the geometry of its children, allowing scene objects to be rotated, translated, scaled, skewed, projected, etc.
- **Switch**: The **Switch** class contains a Boolean mask to enable or disable processing of its children.

This is an incomplete list of OSG node types. Other node types exist, such as **Sequence** and **PositionAttitudeTransform**.

For the system I needed an OSG Node class that allowed me to translate and rotate the plane. I had two choices **MatrixTransfrom** and **PositionAtitudeTransform**. Initially I chose **MatrixTransform**, but now I needed a way to update the **osgviewer** with the moves of the plane. OSG can support dynamical scenes as we saw above. We needed the Callbacks to do it.

CallBacks

OSG allows you to assign callbacks to **Node** and **Drawable** objects. OSG executes **Node** callbacks during the update and cull traversals, and executes **Drawable** callbacks during the cull and draw traversals. This section describes how to dynamically modify a **Node** during the update traversal using an **osg::NodeCallback**. OSG's callback interface is based on the Callback design pattern [Gamma95].

To use a **NodeCallback**, your application should perform the following steps:

- Derive a new class from **NodeCallback**.
- Override the **NodeCallback::operator()()** method. Code this method to perform the dynamic modification on your scene graph.
- Instantiate your new class derived from **NodeCallback**, and attach it to the **Node** that you want to modify using the **Node::setUpdateCallback()** method.

OSG calls the **operator()()** method in your derived class during each update traversal, allowing your application to modify the **Node**.

OSG passes two parameters to your **operator()()** method. The first parameter is the address of the **Node** associated with your callback. This is the **Node** that your callback dynamically modifies within the **operator()()** method. The second parameter is an **osg::NodeVisitor** address. The next section describes the **NodeVisitor** class, and for now you can ignore it.

To attach your **NodeCallback** to a **Node**, use the **Node::setUpdateCallback()** method. **setUpdateCallback()** takes one parameter, the address of a class derived from **NodeCallback**. The following code segment shows how to attach a **NodeCallback** to a node:

```
class RotateCB : public osg::NodeCallback
{
...
};
...
node->setUpdateCallback( new RotateCB );
```

Multiple nodes can share callbacks. **NodeCallback** derives (indirectly) from **Referenced**, and Node keeps a **ref_ptr<>** to its update callback. When the last node referencing a callback is deleted, the **NodeCallback** reference count drops to zero, and it is also deleted. In the code above, your application doesn't keep a pointer to the RotateCB object and doesn't need to.

The book's example code contains a Callback example that demonstrates the use of update callbacks. The code attaches a cow to two **MatrixTransform** nodes. The code derives a class from **NodeCallback** and attaches it to one of the two **MatrixTransform** objects. During the update traversal, the new **NodeCallback** modifies the matrix to rotate one of the cows. Figure 5.10 shows the output of the callback example.

Figure 5.10: Dynamic modification using CallBacks

Now we are going to see the example code, which consists of three main parts. The first part defines a class called **RotateCB**, which derives from **NodeCallback**. The second part is a function called **createScene()**, which creates the scene graph. Note that when this function creates the first **MatrixTransform** object, called **mtLeft**, it assigns an update callback to **mtLeft** with the function call **mtLeft->setUpdateCallback**(new **RotateCB**).

The Callback Example Source Code

This example demonstrates the process of creating a **NodeCallback** to update the scene graph during the update traversal.

```
#include <osgViewer/Viewer>
#include <osgGA/TrackballManipulator>
#include <osg/NodeCallback>
#include <osg/Camera>
#include<osg/Group>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>
// Derive a class from NodeCallback to manipulate a
// MatrixTransform object's matrix.
class RotateCB : public osg::NodeCallback
{
    public: RotateCB() :_angle( 0. ){}
    virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
    {
    // Normally, check to make sure we have an update
    // visitor, not necessary in this simple example.
    osg::MatrixTransform* mtLeft = dynamic_cast<osg::MatrixTransform*>
    (node );
    osg::Matrix mR, mT; mT.makeTranslate( -6., 0., 0. );
    mR.makeRotate(_angle, osg::Vec3( 0., 0., 1. ) );
    mtLeft->setMatrix(mR * mT );
    // Increment the angle for the next from.
    _angle += 0.01;
    // Continue traversing so that OSG can process
    // any other nodes with callbacks.
    traverse( node, nv );
    }
protected:
double _angle;
};
// Create the scene graph. This is a Group root node with two
// MatrixTransform children, which both parent a single
// Geode loaded from the cow.osg model file.

osg::ref_ptr<osg::Node> createScene()
{
    // Load the cow model.
    osg::Node* cow = osgDB::readNodeFile( "cow.osg" );
    // Data variance is STATIC because we won't modify it.
    cow->setDataVariance( osg::Object::STATIC );
    // Create a MatrixTransform to display the cow on the left.
    osg::ref_ptr<osg::MatrixTransform> mtLeft = new
    osg::MatrixTransform; mtLeft->setName( "Left Cow\nDYNAMIC" );
    // Set data variance to DYNAMIC to let OSG know that we
    // will modify this node during the update traversal.
    mtLeft->setDataVariance( osg::Object::DYNAMIC );
    // Set the update callback.
    mtLeft->setUpdateCallback( new RotateCB );
    osg::Matrix m;
```

```
    m.makeTranslate( -6.f, 0.f, 0.f );
    mtLeft->setMatrix( m );
    mtLeft->addChild( cow );
    // Create a MatrixTransform to display the cow on the right.
    osg::ref_ptr<osg::MatrixTransform> mtRight = new
    osg::MatrixTransform;
    mtRight->setName( "Right Cow\nSTATIC" );
    // Data variance is STATIC because we won't modify it.
    mtRight->setDataVariance( osg::Object::STATIC );
    m.makeTranslate( 6.f, 0.f, 0.f );
    mtRight->setMatrix( m );
    mtRight->addChild( cow );
    // Create the Group root node.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->setName( "RootNode" );
    // Data variance is STATIC because we won't modify it.
    root->setDataVariance( osg::Object::STATIC );
    root->addChild( mtLeft.get() );
    root->addChild( mtRight.get() );
    return root.get();
}

int main(int, char **)
{
// Create the viewer and set its scene data to our scene
// graph created above.
osgViewer::Viewer viewer;
viewer.setSceneData( createScene().get());
// Set the clear color to something other than chalky blue.
viewer.getCamera()->setClearColor( osg::Vec4(1.,1.,1.,1.) );
// Loop and render. OSG calls RotateCB::operator()()
// during the update traversal.
viewer.run();
}
```

RotateCB::operator()() contains a call to traverse(). This is a member method of the **osg::NodeCallback** class. This call allows the update traversal (**osgUtil::UpdateVisitor**) to traverse the current group node children. Requiring a call to **traverse()** is a design feature that lets your **NodeCallback** perform either pre- or post-traversal processing, depending on where you place your code relative to the **traverse()** call. Omitting this call prevents OSG from executing child node callbacks. The following section discusses the **NodeVisitor** class in more detail.

Figure 5.11 shows the scene graph that the Callback example creates. The **Group** root node has two child **MatrixTransform** nodes that transform the single cow Geode to two different locations. As the figure shows, one of the two **MatrixTransform** objects has its data variance set to DYNAMIC, and the other uses STATIC data variance because the code never modifies it. The **MatrixTransform** on the left has the update callback attached to it, which dynamically modifies the matrix during the update traversal.

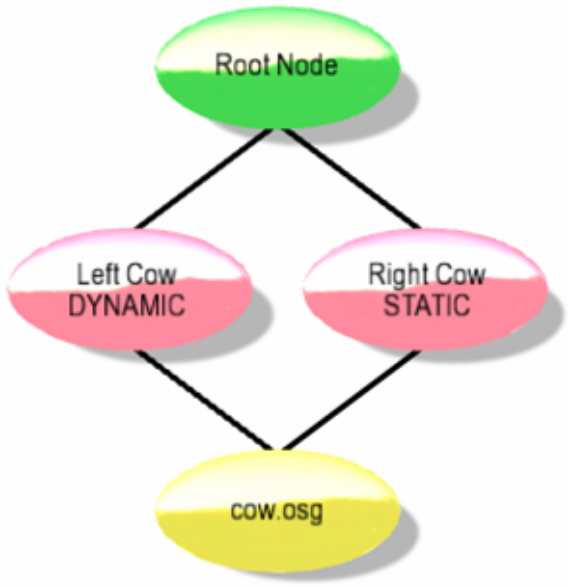

Figure 5.11: The Callback example program scene graph

As this example illustrates, dynamically modifying a node is straightforward because attaching an update callback to a known node is trivial. The problem becomes more complex if your application modifies a node that is buried deep within a scene graph or selected interactively by a user. The next sections describe some methods in OSG for runtime node identification.

NodeVisitors

NodeVisitor is OSG's implementation of the Visitor design pattern [Gamma95]. In essence, **NodeVisitor** traverses a scene graph and calls a function for each visited node. This simple technique exists as a base class for many OSG operations, including the **osgUtil::Optimizer**, the **osgUtil** library's geometry processing classes, and file output. OSG uses the **osgUtil::UpdateVisitor** class (derived from **NodeVisitor**) to perform the update traversal. In the preceding section, **UpdateVisitor** is the **NodeVisitor** that calls the **NodeCallback::operator()()** method. In summary, **NodeVisitor** classes are used throughout OSG.

NodeVisitor is a base class that your application never instantiates directly. Your application can use any **NodeVisitor** supplied by OSG, and you can code your own class derived from **NodeVisitor**. **NodeVisitor** consists of several **apply()** methods overloaded for most major OSG node types. When a **NodeVisitor** traverses a scene graph, it calls the appropriate **apply()** method for each node that it visits. Your custom **NodeVisitor** overrides only the **apply()** methods for the node types requiring processing.

After loading a scene graph from file, applications commonly search the loaded scene graph for nodes of interest. As an example, imagine a model of a robot arm containing articulations modeled with transformations at the joints. After loading this file from disk, an application might use a **NodeVisitor** to locate all the Transform nodes to enable animation. In this case, the application uses a custom NodeVisitor and overrides the **apply(osg::Transform&)** method. As this custom NodeVisitor traverses the scene graph, it executes the **apply()** method for each node that derives from Transform, and the application can perform the operations necessary to enable animation on that node, such as saving the node address in a list.

If your **NodeVisitor** overrides multiple **apply()** methods, OSG calls the most specific **apply()** method for a given node. For example, **Group** derives from **Node**. If your **NodeVisitor** overrides **apply(Node&)** and **apply(Group&)**, OSG calls **apply(Group&)** if it encounters a **Group** or any node derived from **Group** during the traversal. If OSG encounters a **Geode**, it calls **apply(Node&)** in this example, because **Geode** derives from **Node**, not from **Group**.

The **traverse()** method is a member of **NodeVisitor**. This is different from, but similar to, the **traverse()** call in section 5.3.2 **Callbacks**, which is a member of **NodeCallback**. When traversing a scene graph, a **NodeVisitor** uses the following rules:

- A vanilla **NodeVisitor** configured to TRAVERSE_ALL_CHILDREN traverses its children.

- Custom **NodeVisitor** classes that override one or more apply() methods are responsible for calling **NodeVisitor::traverse()** to traverse a node's children. This requirement allows your custom **NodeVisitor** to perform pre- and posttraversal operations, and stop traversal if necessary.

- When using callbacks executed by **NodeVisitor** classes, such as an update callback as described in the previous section, the **NodeVisitor** traverses the children of nodes without a callback. **NodeVisitor** doesn't traverse the children of nodes with callbacks attached. Instead, the callback method **operator()()** is responsible for traversing children with a call to N**odeCallback::traverse()** if the application requires traversal.

To traverse your scene graph with a **NodeVisitor**, pass the **NodeVisitor** as a parameter to **Node::accept()**. You can call **accept()** on any node, and the **NodeVisitor** will traverse the scene graph starting at that node. To search an entire scene graph, call **accept()** on the root node.

Building the dynamic scene

Taking in account all explained before about **CallBacks** and **Nodevisitors** was time to apply all to my program. At this point with each viewer frame using callbacks I could move the plane, because in each frame Callbacks would be executed and if I schedule that callback with instructions to translate or rotate the plane, this translation or rotation would be shown in the viewer. But how could I do it? First solution I found was to use an **osg::AnimationPath** class.

The class **osg::AnimationPath** encapsulates a time varying transformation pathway. It can be used for updating camera position and model object position. **Osg::AnimationPathCallBack** can be attached directly **Transform** nodes to move subgraphs around the scene.

First of all, I needed to define a path that UAV would follow. I decided that the UAV describing a square path over the terrain always at the same height (to eliminate the variable z). So it was needed to define a home position, distance traveled in each frame and when the UAV would have to rotate and change its direction. As I was trying to describe a square it was very easy, only one of the coordinates was changing in each moment. The code to create the **AnimationPath** was the next:

```
osg::AnimationPath* createAnimationPath(const osg::Vec3& home,
doublelooptime)
{
    // set up the animation path
    osg::AnimationPath* animationPath = new osg::AnimationPath;
    // Make sure that it build the loop
    animationPath->setLoopMode(osg::AnimationPath::LOOP);
    // number of samples
    int numSamples = 40;
    float displacement = 0.0f;
    // total displacement will be 800 and we divided that for the
    // number of samples
    float displacement_delta = 800.0f/((float)numSamples-1.0f);
    double time = 0.0f;
    double time_delta = looptime/(double)numSamples;

    // loop "for" of the first path of the square
    for(int i=0;i<10;++i)
    {
        // Position will be the home plus the displacement during
        // the path in the first path we are travelling through y
        // axis from negative values to positive ones
        osg::Vec3 position(home+osg::Vec3(0.0f,displacement,0.0f));
        osg::Quat rotation(osg::Quat(osg::PI,osg::Vec3(0.0,0.0,1.0)));
        // with the token time we introduce the position and
        // orientation of the plane into the animationpath
        animationPath->insert(time,osg::AnimationPath::ControlPoint
        (position,rotation));
```

```
        // Update the variables
        displacement += displacement_delta;
        time += time_delta;

    }
    displacement = 0.0f;
    // auxiliary variable to locate the plane
    osg::Vec3 aux (-100.0f,100.0f,100.0f);
    for(int i=0;i<10;++i)
    {
        // Now we are travelling through x axis, from negatives
        // values to positive ones, therefore the displacement is
        // on x axis.
        osg::Vec3 position(aux+osg::Vec3(displacement,0.0f,0.0f));
        osg::Quat rotation(osg::Quat(0.5f*osg::PI,osg::Vec3(0,0,1)));
        animationPath->insert(time,osg::AnimationPath::ControlPoint
        (position,rotation));

        displacement += displacement_delta;
        time += time_delta;

    }
    displacement = 0.0f;
    osg::Vec3 aux1 (100.0f,100.0f,100.0f);
    for(int i=0;i<10;++i)
    {
        // Now we are travelling through y axis, from positive values
        // to negative ones, therefore the displacement is on y axis.
        osg::Vec3 position(aux1-osg::Vec3(0.0f,displacement,0.0f));
        osg::Quat rotation(osg::Quat(0.0f,osg::Vec3(0.0,1.0,0.0)));
        animationPath->insert(time,osg::AnimationPath::ControlPoint
        (position,rotation));

        displacement += displacement_delta;
        time += time_delta;
    }
    displacement = 0.0f;
    osg::Vec3 aux2 (100.0f,-100.0f,100.0f);
    for(int i=0;i<10;++i)
    {
        // Now we are travelling through x axis, from positive values
        // to negative ones, therefore the displacement is on x axis
        osg::Vec3 position(aux2-osg::Vec3(displacement,0.0f,0.0f));
        osg::Quat rotation(osg::Quat(3.0f*osg::PI/2.0f,osg::Vec3(0,0,1)));
        animationPath->insert(time,osg::AnimationPath::ControlPoint
        (position,rotation));

        displacement += displacement_delta;
        time += time_delta;
    }
    return animationPath;
}
```

It is possible to distinguish four parts in the code, each one associated

to a stretch of the square path. In the code are associated position and orientation of the plane with a timestamp, and in each iteration of any loop "for" is stored this timestamp with its associated values of position and rotation. In that way OSG knows in each moment where the plane is. Like I activated the loop mode, the movement repeats in a periodical way.

But, there is still something to be done. It was needed to call to **Osg::AnimationPathCallBack**, in order to do an update callback when the viewer is refreshed. I had another function that has this mission:

```
osg::Node* createMovingModel(const osg::Vec3& home)
{
    float animationLength = 40.0f;
    osg::AnimationPath* animationPath = createAnimationPath(home,
    animationLength);
    osg::Group* model = new osg::Group;
    osg::Node* cessna=osgDB::readNodeFile("./models/cessna.osg");
    if (cessna)
    {
        osg::MatrixTransform* xform = new osg::MatrixTransform;
        xform->setUpdateCallback(new osg::AnimationPathCallback
        (animationPath,0.0f,2.0));
        xform->addChild(cessna);
        model->addChild(xform);
    }
    return model;
}
```

Like we can see in the code above, this function receive a 3D vector that represents the home position of the plane. It calls to the function we saw before, which is responsible to creating the **AnimationPath**. Then the program loads the 3D model of the plane and creates an **osg::MatrixTransform** Node called ***xform***. Program calls to the **setUpdateCallback** method with a new **osg::AnimationPathCallback** class (this last, with the **AnimationPath** set up before like argument) as argument. Thus, all translations or rotations made to ***xform*** will affect to its children. Then, a child of ***xform*** type **osg::Node** is created and contains the 3D model of the plane. Therefore, all the transformations made to xform will affect our plane. Finally, ***xform*** is added like a child to an **osg::Group Node**, and this node is return to the main program where it will become in a child of the rootNode. This rootNode will be passed like a parameter to the viewer to render the scene.

The result was very satisfactory. It was possible to ascertain that the plane described a square path over the terrain. But there was a mistake, when the plane arrived to the home position did not rotate to face its target but the plane appeared directly facing the goal. But that mistake was not the only, in next sections will see the limitations of this method.

5.3.3 Cameras management

At this point, we had a scene with a plane flying over a terrain. The camera point of view is extern to the scene, that is to say we can see the plane and the terrain simultaneously. But, as I wanted to simulate a camera in the UAV, I needed to change the camera manipulator and associate it with the plane for the camera to be translated and rotated with the plane. In OSG there are many camera manipulators, by default osgViewer have a **TrackBallManipulator**.

We suppose that we have created the scene and the principal node is rootnode (all other nodes on the scene are its children), and now we are going to define the viewer and we want to view the scene which parent node is rootnode, the instructions would be the next:

```
osgViewer::Viewer viewer;
viewer.setScenedata(rootnode);
viewer.setUpViewInWindow( 100, 100,800, 650 );
viewer.realize();
while (!viewer.done())
{
    Viewer.frame(simulationTime);
    simulationTime += 0,001,
}
```

It is easy to understand the code above, is defined a variable **osgViewer:: Viewer** called ***viewer***; with **setScenedata**, that is a method of the **osgViewer::Viewer** class, indicates that the viewer has to represent the scene graph which parent is rootnode. Method **setUpInWindow** define the position and the size of the window of the viewer. And last part is the loop to represent the scene. Figure 5.12 illustrates the output generated for the code above.

Last picture shows the 3D scene in a window (due to **setUpInWindow** method) and as can be seen for default point of view is external to the scene and shows all elements of the scene. But we were looking for a point of view attached to the plane, in this way when the plane moves and rotates, so will the camera. Then, a new definition of the camera manipulator was needed. With the classic camera manipulator, **TrackBallManipulator**, is not possible to get it, for this reason was necessary to find another camera manipulator. Looking for it between OSG camera manipulators I found **osgGA::NodeTrackerManipulator**, this new manipulator has the properties that I was looking for, and it has different members to specify the node to follow and how it rotates.

The **osgGA::NodetrackerManipulator** has two rotation modes:

- TRACKBALL: Use a trackball style manipulation of the view direction with respect to the tracked orientation.

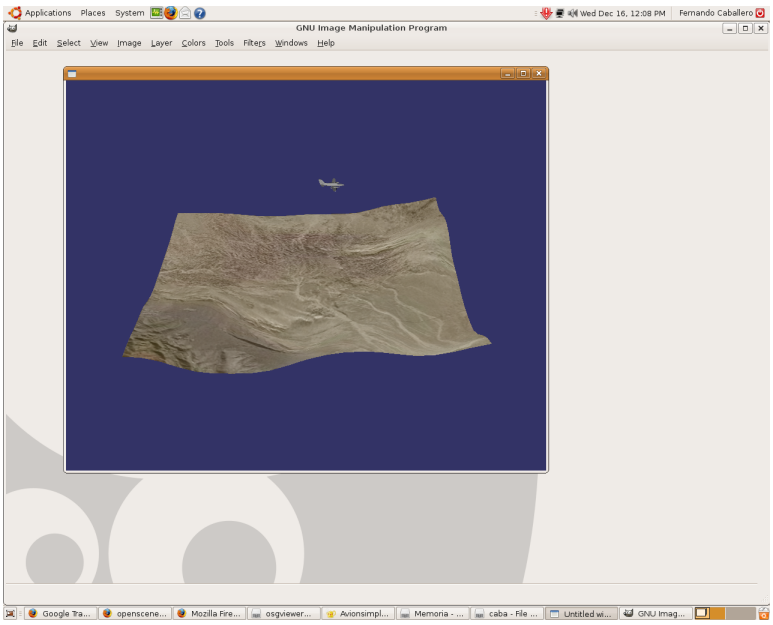

Figure 5.12: Scene view for default

- ELEVATION_AZIM: Allow the elevation and azimuth angles to be adjust with respect the tracked orientation.

  On the other hand, it is necessary to indicate the way that the node is tracked for the new **NodeTrackerManipulator**. There are three different ways of tracking nodes:

- NODE_CENTER: Track the center of the nodet's bounding sphere, but not rotations of the node.

- NODE_CENTER_AND_AZIM: Track the center of the nodet's bounding sphere, and the azimuth rotation (about the z axis of the current coordinate frame).

- NODE_CENTER_AND_ROTATION: Track the center of the nodet's bounding sphere, and the all rotations of the node.

I tested all different ways to track the plane and two ways to track the rotation of the plane. From my point of view, the best possible configuration was to track the plane with NODE_CENTER_AND_ROTATION mode and rotates the view direction with ELEVATION_AZIM mode. The way to track the rotation of the node was chosen because the chosen one (NODE_CENTER_AND_ROTATION) allows to make the same rotations to the camera than those suffered by the plane. In the other hand, the way that camera manipulator rotates was chosen due to this method is more life-

like, simulates a pulp and tilt. To configure a **NodeTrackerManipulator** are only necessary this code lines:

```
osgGA::NodeTrackerMAnipulator* tm = new
osgGA::NodeTrackerManipulator;
tm->setTrackerMode(osgGA::NodeTrackerManipulator::NODE_CENTER
_AND_ROTATION);
tm->setRotationMode(osgGA::NodeTrackerManipulator::ELEVATION
_AZIM);
```

At this point, I needed to attach the **NodeTrackerManipulator** to the plane. In order to do it, we defined a new child for the plane node. This node would be slightly below the plane node and would allow us to attach the **NodeTrackerManipulator** to it. This process is conducted with the following code lines:

```
Osg::positionAttitudetrasform* followerOffset = new
osg::PosititionAttitudeTransform();
followerOffset->setPosition(osg::Vec3(0.0,0.0,-10.0));
followerOffset->setAttitude( osg::Quat(
osg::DegreeToRadians(90.0),osg::Vec3(1, 0, 0)));
Plane.get->addChild(followerOffset);
tm->setTrackNode(followerOffset);
```

With the last command we attach the **NodeTracker** to **followerOffset** that we can see is a child of the plane. With the instruction **setPosition**, we indicate to the system that the node **followerOffset** will be slightly below from the plane; and with the instruction **setAttitude** we indicate that initially the camera will be looking down. But now we needed to indicate to the viewer that the camera manipulator would be tm, for that we will use the Viewer method **setCameraManipulator**:

```
osgViewer:Viewer viewer;
viewer->setCameraManipulator(tm);
```

At this point if we execute the program we would get the shown in the Figure 5.13:

Obviously the obtained result is the terrain. But the aim was to get that the camera moves with the plane and we got it. Like we saw in sections above, the plane is describing a square trajectory and the camera, slightly below of the plane, describes the same trajectory at the same time that the plane.

In that point I had an idea, we can use to views, one like the figure 5.13 illustrates, and another like the figure 5-13 shows. In order to achieve that aim, we needed another new tool, because the **Viewer** did not allow to do it directly. OSG has a tool for this, and this tool is **CompositeViewer**. While Viewer manages a single view into a scene (possibly with a group of Camera objects to support multipipe rendering), **CompositeViewer** supports multiple views into one or more scenes and allows our application to specify their

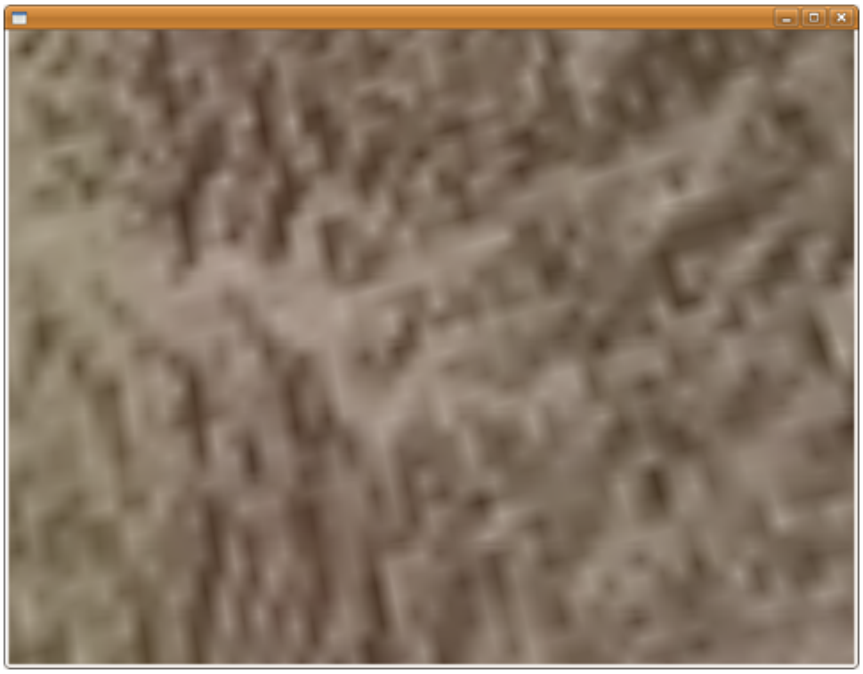

Figure 5.13: Camera view of plane

rendering order. **CompositeViewer** supports render-to-texture (RTT) operations, which allows our application to use the rendered image from one view as a texture map in a subsequent view.

I was looking for the best way to represent two different views at the same time. In OSG examples I found a code **createView.cpp** that let me to show the two views in the same window. The code is shown below:

```
#include <osgViewer/Viewer>
#include <osgDB/ReadFile>
#include<osgSim/MultiSwitch>
#include <osg/MatrixTransform>
#include<osg/PositionAttitudeTransform>
#include<osgGA/TrackballManipulator>
#include <osgViewer/CompositeViewer>
#include <osgGA/StateSetManipulator>
#include<osgGA/NodeTrackerManipulator>

void createView (osgViewer::CompositeViewer *viewer,
                 osg::ref_ptr<osg::Group> scene,
                 osg::ref_ptr<osg::GraphicsContext> gc,
                 osgGA::NodeTrackerManipulator* Tman,
                 int x, int y, int width, int height)
{
    double left,right,top,bottom,near,far, aspectratio;
    double frusht, fruswid, fudge;
    bool gotfrustum;
```

```
    osgViewer::View* view = new osgViewer::View;
    viewer -> addView(view);
    view->setCameraManipulator(Tman);
    view->setSceneData(scene.get());
    view->getCamera()->setViewport(new osg::Viewport
                                        (x,y,width,height));
    view->getCamera()-> getProjectionMatrixAsFrustum(left,right,
                                          bottom,top,near,far);
    if (gotfrustum)
    {
        aspectratio = (double) width/ (double) height;
        frusht = top - bottom;
        fruswid = right - left;
        fudge = frusht*aspectratio/fruswid;
        right = right*fudge;
        left = left*fudge;

        view->getCamera()-> setProjectionMatrixAsFrustum
                           (left,right,bottom,top,near,far);
    }
    view->getCamera()->setGraphicsContext(gc.get());
    // add the state manipulator
    osg::ref_ptr<osgGA::StateSetManipulator>
            statesetManipulator = new osgGA::StateSetManipulator;
    statesetManipulator->setStateSet(view->getCamera()
                                        ->getOrCreateStateSet());
    view->addEventHandler( statesetManipulator.get() );
}
```

As we can see in the last code, it is a function that receives as parameters a **compositeViewer**, the parent node of the scene, a graphic context, the camera manipulator and the position and the size of each view inside the window. I have decided to show the code because finally it was discarded and it is not include in Appendix A. In the main program only will be necessary to define a **compositeViewer** and add two lines:

```
osgViewer::CompositeViewer viewer;
createView (&viewer, rootNode,gc,tm, 0, 0, traits->width, h2);
createView (&viewer, rootNode, gc,tm2, 0, h2, w3, h2);
```

Like we can see above, we define two views. One which camera manipulator is ***tm*** (in this case is the camera attached to the plane) and its width is full screen, and the other which its camera manipulator is ***tm2*** (which correspond to a point above the plane but also moves with it) and its width is one third of the screen. Both have the same height, half the screen. The loop will be the same that in previous examples:

```
While( !viewer.done())
{
    Viewer.frame();
}
```

And the result of the execution of the code is illustrated in the Figure 5.18:

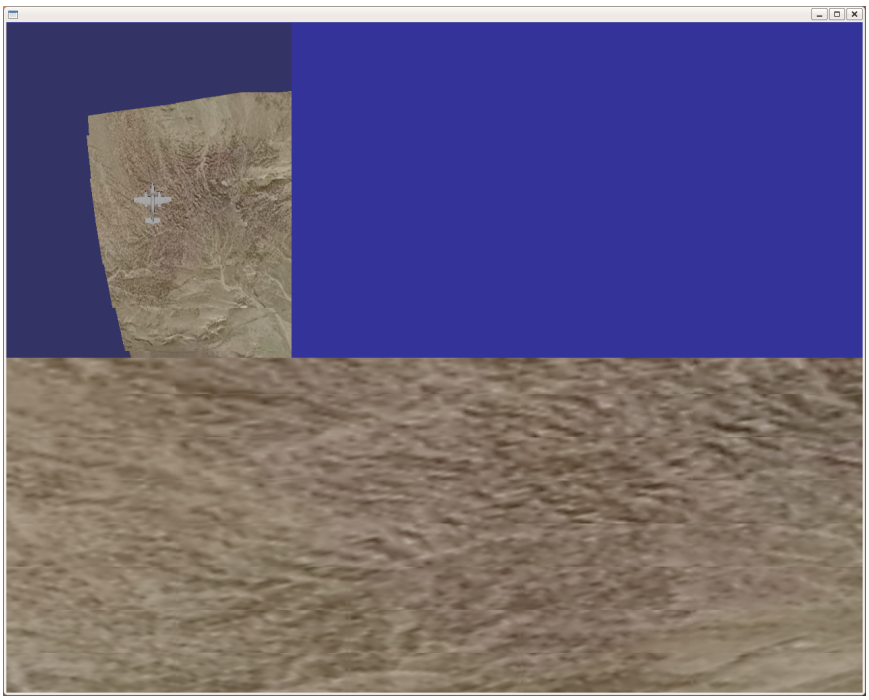

Figure 5.14: Two views in the same window

Resulting aesthetics was not too pretty, so I decided to probe with two different windows. In this case, we can use a **CompositeViewer** method that allows us to add different views. This method is **addView**, and it lets to add different Viewers to a **CompositeViewer**. Then, it is necessary to show these Viewers in different windows but only display one. We can do it using the Viewer method **setUpViewInWindow**, that receives like parameters the position and size of the window. The modification to the code would be:

```
osgViewer::CompositeViewer viewer;

// Create View 0
osgViewer::View* view = new osgViewer::View;
viewer.addView(view);
view->setUpViewInWindow( 10, 680, 400, 270);
view->setSceneData(rootNode);
view->setCameraManipulator(tm2);

// Create View 1

osgViewer::View* view1 = new osgViewer::View;
viewer.addView(view1);
view1->setUpViewInWindow( 10, 680, 400, 270);
view1->setSceneData(rootNode);
view1->setCameraManipulator(tm2);
```

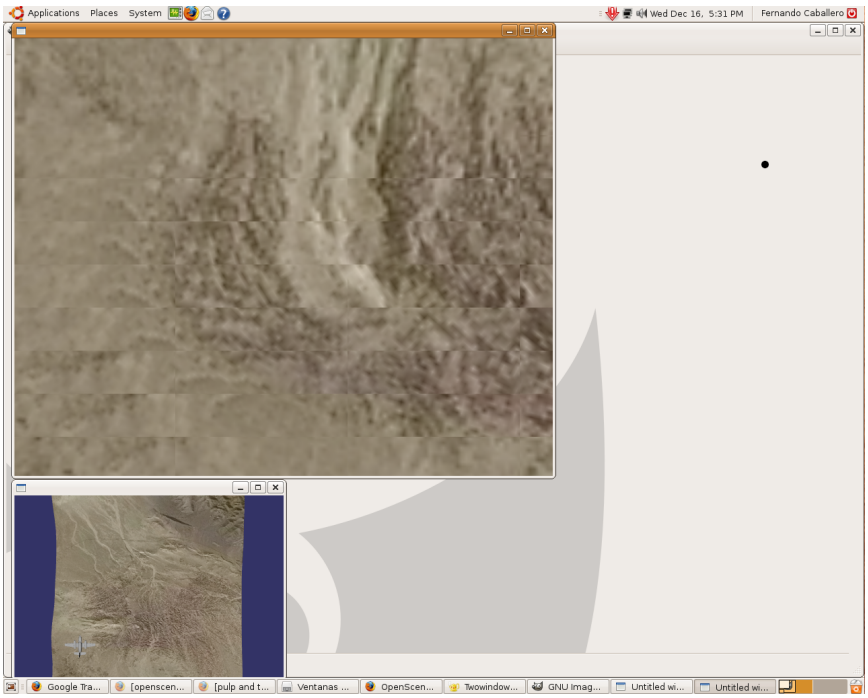

Figure 5.15: Views in different windows

And now we can see the result in the figure 5.15

Comparing the two options, personally I think it is better the second, that is to say, to have one view in different windows. In this way we can move one of the views where we prefer without prejudice to other views. Even considering we have several screens in the station we could move windows between them to our taste. With two views in the same windows that would not be possible.

5.3.4 Introduction of GTK elements

At this point, we have a representation in two windows of a scene with a plane flying around a terrain. But, it would be good that the user, as well as monitor and see what happens windows would interact with them. For this reason was decided to introduce **GTK**, which would allow us to build a graphical interface to interact with the elements in the scene. GTK+ [6] is a highly usable, feature rich toolkit for creating graphical user interfaces which boasts cross platform compatibility and an easy to use API. GTK+ it is written in C, but has bindings to many other popular programming languages such as C++, Python and C# among others. GTK+ is licensed under the GNU LGPL 2.1 allowing development of both free and proprietary software with GTK+ without any license fees or royalties.

Over time GTK+ has been built up to be based on four libraries, also

Figure 5.16: GTK

developed by the GTK+ team:

- GLib, a low-level core library that forms the basis of GTK+. It provides data structure handling for C, portability wrappers and interfaces for such run-time functionality as an event loop, threads, dynamic loading and an object system.

- Pango, a library for layout and rendering of text with an emphasis on internationalization. It forms the core of text and font handling for GTK+ 2.0.

- Cairo, a library for 2D graphics with support for multiple output devices (including the X Window System, Win32) while producing a consistent output on all media while taking advantage of display hardware acceleration when available.

- ATK, a library for a set of interfaces providing accessibility. By supporting the ATK interfaces, an application or toolkit can be used with tools such as screen readers, magnifiers, and alternative input devices.

GTK+ was initially developed for and used by the GIMP, the GNU Image Manipulation Program. It is called the "The GIMP ToolKit" so that the origins of the project are remembered. Today it is more commonly known as GTK+ for short and is used by a large number of applications including the GNU project's GNOME desktop. Looking for a way to apply GTK to OSG, I found **osgGtk**, which is a library of C based Gtk+ and C++ based Gtk widgets to support OpenSceneGraph (OSG) applications. The library also includes several example applications such as osgviewerGtk and osgviewerGtkmm.

Downloaded version was **osgGtk-0.1.4**, last released version. I installed all necessary dependencies and run the application osgviewerGtk. Figure 5.17 illustrates osgviewerGtk:

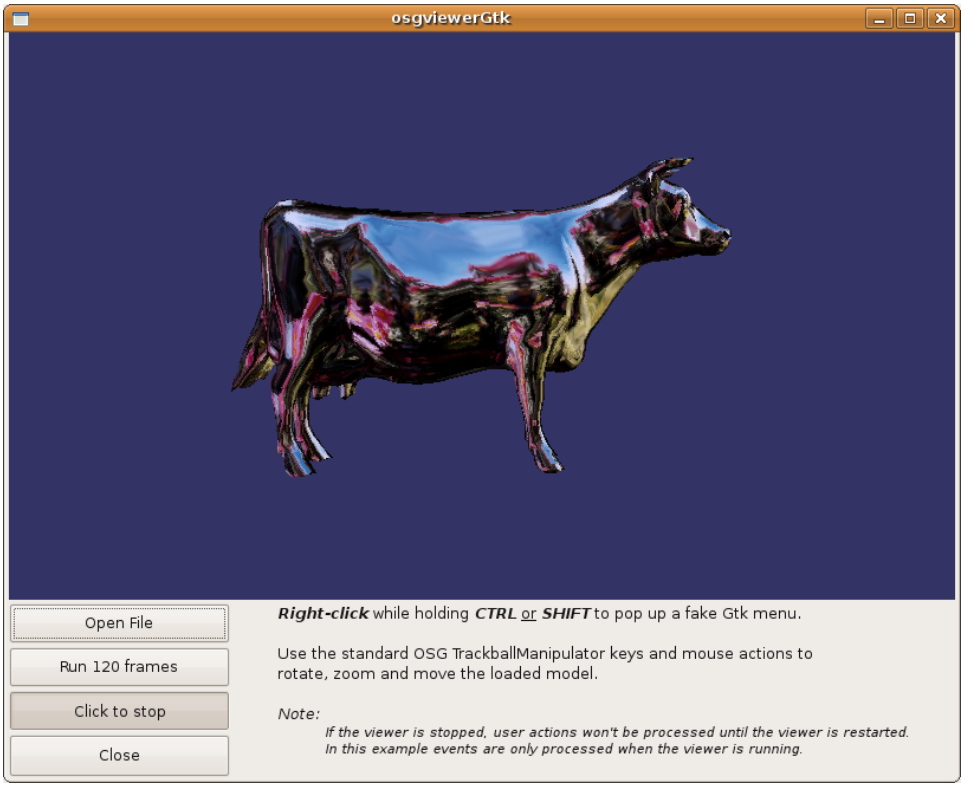

Figure 5.17: OsgViewerGtk

If we compare the two viewers:

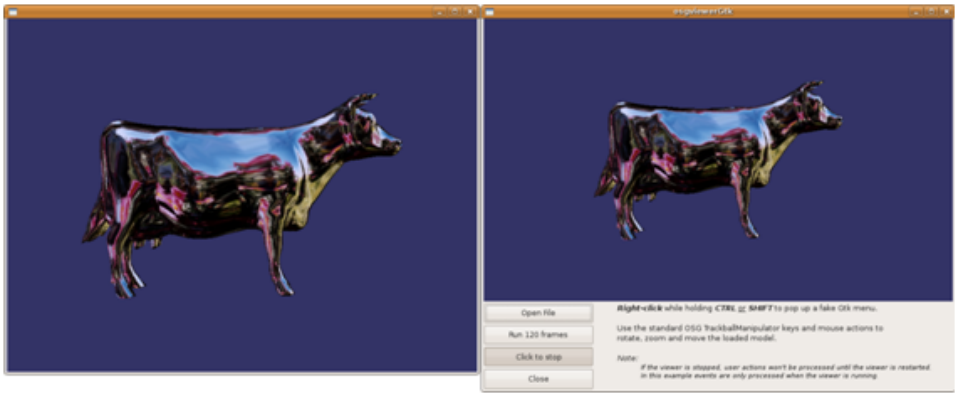

Figure 5.18: Viewers comparison, classic viewer to the left, and Gtkviewer to the right

Now we are going to see the advantages of the use of osgviewerGtk. We look at the buttons on the far left bottom of the screen (Figure 5.19).

For default osgviewerGtk has associated four buttons. If we click with

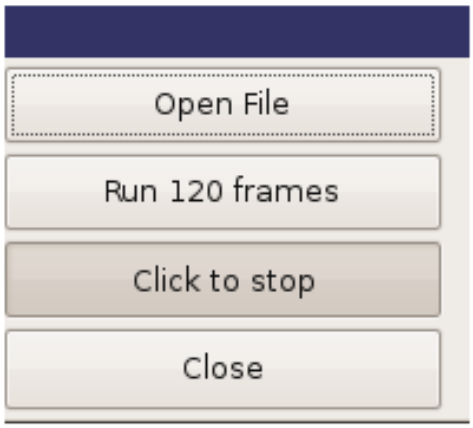

Figure 5.19: Function buttons

the mousse in the buttons or touch the screen in the position of the buttons (remember we are working with touchscreens), a CallBack is executed and there will be an action associated to that button. For example, if we press (or touch) the "Open File" button a new window will open and we can load a 3D model of our computer looking for it. If we press the "Run 120 frames" viewer will be refreshed at 120 frames per second (for default are 60 frames per second). This button has the particularity that changes when is pressed and now the third button "Click to stop" become "Click to run 60 frames", so if we press the button called now "Click to run 60 frames" viewer will be refreshed at 60 frames per second again. The third button "Click to stop" stop the viewer when is pressed, and changes to "Click to run 60 frames", obviously when is pressed again the viewer run. Last button "Close" finish the gtkmain (gtk loop) and close the viewer.

These four buttons offer many possibilities to the station, the functionality can be changed for another more useful with a little change in the function called when the callback is triggered.

osgviewerGtk has scheduled for default a fake Gtk menu that pops up when the user right-click while holding CTRL or SHIFT as illustrates Figure 5.20, and if you look at the bottom of the viewer will can see that osgviewerGtk allows the introduction of explanatory text on it. This fake menu currently does nothing but is a good option to add new functionalities in the future.

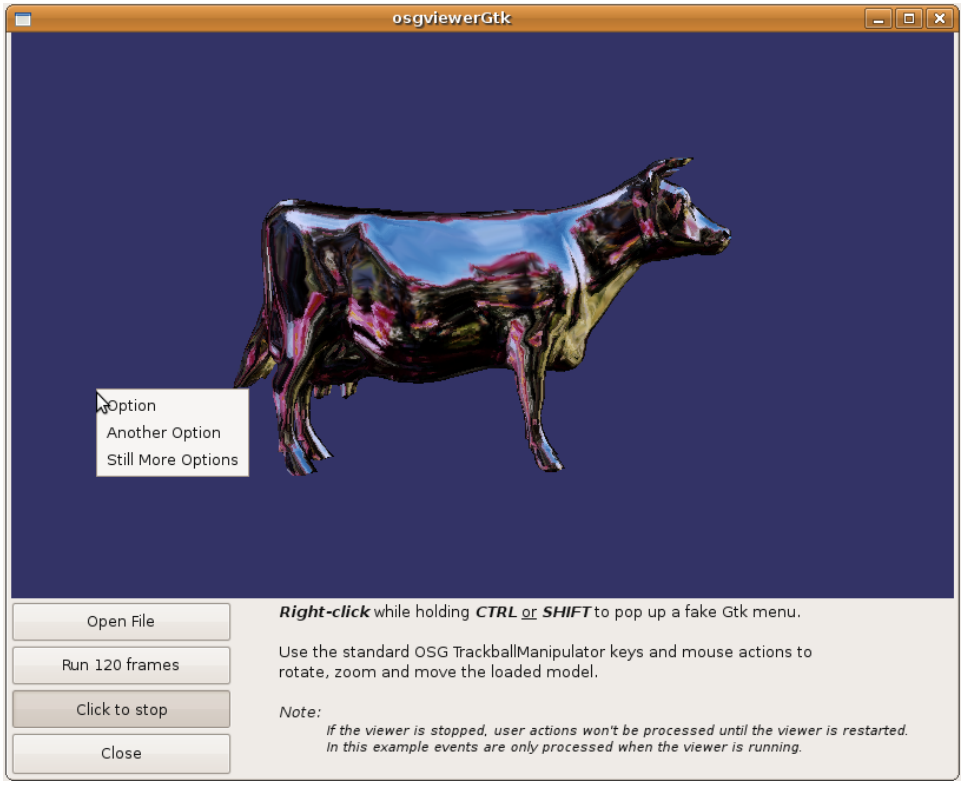

Figure 5.20: Gtkviewer with fake menu

Once I had the Gtkviewer was the moment to prepare the final scene with UAVs.

5.3.5 Creating the final scene

At this point, I provided necessary tools and knowledge to mount the scene. The aim was to have some UAVs flying near an airport, and to be able to control or monitoring one or more UAVs with a camera. We must to be able to use tools like Head-tracking or 3D sound (first is now integrated, second will can be integrated in the future).

Creating the terrain

First of all, I needed a terrain. Terrain used in previous examples is a Microsoft Flight Simulator file (.flt), and it was impossible to modify it. I was looking between OSG examples and I found **osghangglide.cpp** that creates a simple flying site, demonstrating how to create simple terrain, trees and skydomes, and how to implement a simple flight camera manipulator to allow the user to fly around, it is possible to check the code in OSG official website. But, to our station I only needed the part of the code related with

to create the terrain and trees. Codes can be consulted in the appendix A of memory, I will try to reference pages as needed. Code of **Createscene.cpp** A.2 is responsible, among other things, to create the terrain, for it rests the codes of **terrain.cpp** A.11, **trees.cpp** A.13 and **hat.cpp** A.6. First allows us to build the terrain and the other ones to add trees to the scene. It is impossible and it is not my aim to spell out each of these codes, so I will try to explain them superficially. **Terrain.cpp** A.11 uses an array located in **terrain_coords.h** A.12 called ***vertex***, which stores the coordinates of the terrain to build it. The array vertex is constituted for 38 blocks with 39 three dimensions vectors. Each block is a column of the final terrain (for example considering the first dimension of the vectors is the x-axis, each block would contain the points on the plane with the same value of x). The function makeTerrain takes this array and builds the terrain using the OpenGl primitive TRIANGLE_STRIP. Operation is shown in the next figure:

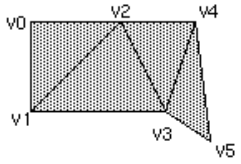

Figure 5.21: GL_TRIANGLE_STRIP

TRIANGLE_STRIP receives coordinates and builds up a mesh of triangles. In the figure we can see its behavior. We have V0 and add to the mesh V1, after that we add V2 and, for add it to the mesh, the primitive joins V0 with V2, then we add V3 and again to join it to the triangle mesh, the primitive joins V3 with V1; and so on. In that way is possible to create any surface.

In our case, V0 would be the first vector of the first block and V1 the first vector of the second block. Then, V2 would be the second vector of the first block, and V3 would be the second vector of the second block and in this way, the surface terrain is built.

Finally to build the terrain is necessary to add a texture. Like we wanted to represent an airport, I was looking for airports satellite images, for that I used Google Earth. I took the images from the airport of Badajoz and around. The images can be seen in Figure 5.22:

Field texture was added with the aim that airport texture did not have to occupy the entire terrain as it would be a problem due to UAVs only could move near to the airport. For this reason and because the resultant terrain was very small, I decided to replicate the terrain and to locate replicas around the airport with the texture of the field, as we can see in figure

Figure 5.22: Airport texture (on the left) and field texture on the right

5.23. This part is carried out for the function **Createterrainaux** presents in **Createscene.cpp** A.2.

Figure 5.23: Terrain structure

In that point was necessary to adjust the height of the terrain, I needed that replicas at their ends coincide, so I needed to change the ***vertex*** array in **terrain_coords.h** A.12 to do that. Besides, I would help pave the terrain in which the airport would be. In order to achieve it, I created a program called **Translator.cpp** A.16 with which is possible to modify the ***vertex*** array.

First part of the code has the aim to create a struct to store the coordinates of each point. Main looks very difficult but only is responsible to get a flat terrain where the airport texture will be and that the ends of each field match, that is to say the coordinate in z-axis must be the same. In order to achieve this must be a .txt file called **prueba.txt** in the same folder that the code above. We make a copy of the ***vertex*** array inside it and run the program. The resulting array is stored in a text file called **pruebaout.txt**.

Now we save the new array in **terrain_cords.h** A.12 and when ***vertex*** array would be called by the function **makeTerrain** we obtain the new stored heights.

But even with this increase was insufficient and had to have escalation in OSG.

At this point was the moment to add the trees to the scene. Trees are mere decorative detail, but it is not difficult to add to the scene and the result is very colorful. There are two .cpp files that are responsible to create and add to the scene the trees. They are **trees.cpp** A.13 and **hat.cpp** A.6. With the first .cpp file is defined the position of the trees in the scene as well as also the height and width of each one, the image .rgba of a tree (Figure 5.24) is associated to it too. At the other hand **hat.cpp** A.6 is responsible to match the coordinates of the terrain with that of the trees, thus placing the trees at the proper height.

It is especially significant the functioning of trees, is a plane with the form and the image of a tree that rotates with the camera seeming a 3D tree. When the point of view of the camera is just above, the behavior is not the most appropriate, but in any other case the impression is pretty good.

Figure 5.24: Tree image

The whole result is illustrated in Figure 5.25 but is not possible to see in the Figure 5.25 the trees, we have a detailed view of the tree in the Figure 5.26

Adding elements to the scene

An important part of the tool is to achieve credibility, for this reason was decided to add different decorative elements to the scene. Because we wanted to simulate an airport it was essential a control tower and hangars. Fortunately it was not necessary to create these 3D models, I had some 3D Studio models and OSG accepts the .3DS format. However, it could not be so easy. When I try to load .3DS files in OSG there were no problems, but in the moment I tried to scale the 3D models the textures were lost. I had to do the scaling with 3D Studio and then to load the models with OSG. From the

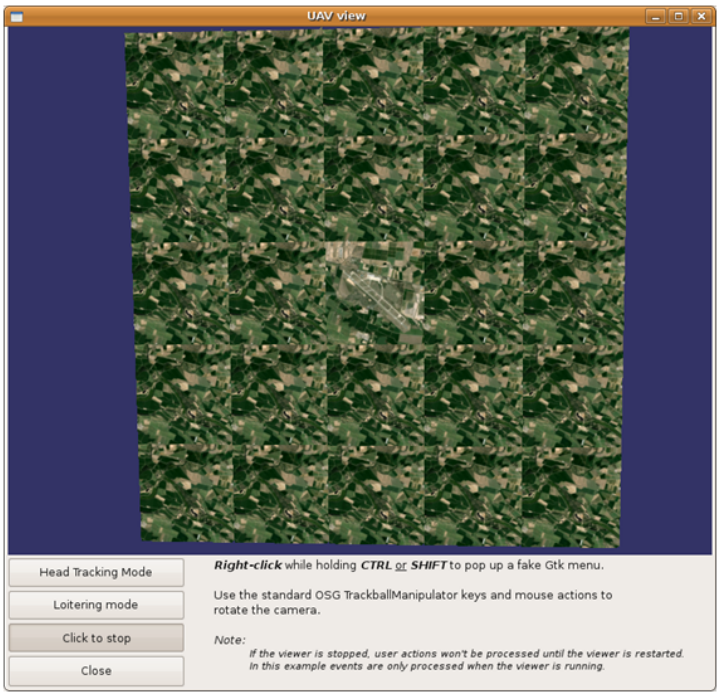

Figure 5.25: Aerial view of the terrain

Figure 5.26: Detailed view of trees

beginning I had the chosen model to the UAV (a helicopter) and hangars, but I did not have a control tower. Then I had to find a 3D model of a control tower. I was looking on the internet for 3D models and I found Google Sketchup 3d warehouse that is a repository of free 3D models and I found a control tower 3D model. But it had a problem, these files only was available in two 3D formats, SketchUp and Collada. Fortunately, Collada is an open standard XML schema for exchanging digital assets among various graphics software applications. This file format provides comprehensive encoding of visual scenes and physics, and even multiple version of the same asset. It also is a very hot topic within the video game industry as it offers quality file format that can somewhat be standardized. With a 3D Studio pluging I was able to change the format of the model from Collada to 3D Studio file. In this way I could load the control tower 3D model in OSG. We can see in the following figure the 3D models presents in the scene:

Figure 5.27: 3D models

Movements of UAVs

Now, is the time to explain like UAVs are going to move by the scene. Like we could see in previous sections, we established a path at the start of the execution and UAV followed it. This first solution was inflexible, it did not allow me to change the movements of the UAV in another way that changing

the whole path code. In the other hand, like the path is introduced at the beginning of the execution was impossible to affect to the UAV movement during the execution. I only could be a viewer. For these reasons was necessary to change the way that UAVs were moved by the scene.

The way I could do it was the same as before (using Callbacks), but instead of to specify the way to go at the beginning of the program, I decided to move the UAV a range increment in every frame of the viewer. In this case, it would be necessary to know the points which were the described path and to define a new variable type called **Movement** that is an **osgNodeCallback** which, as we will see below, will allow us to move the UAVs. So, we would have to enter the coordinates at the beginning of the program. This last part is taking care by the function **readcoordinates** (**Readcoordinates.cpp** A.9), which receives the array of classes **Movement** type (**osg::NodeCallback**) and the number of UAVs, and is responsible for reading the files of coordinates of each UAV (in the same folder as the program) and stored them in their proper place (UAV vector inside **Movement** class). This way the program is independent of the number of points (three coordinates for point) we enter in it, even different UAVs can have a number of different target points. Coordinates files should follow these guidelines:

Coordinate_x1 Coordinate_y1 Coordinate_z1 (intro)
Coordinate_x2 Coordinate_y2 Coordinate_z2 (intro)
.....
Coordinate_xn Coordinate_yn Coordinate_zn

Like we can see above is possible to introduce any number of points (three coordinates each point), but is very important that the last line (with the coordinates of the last point) does not end with an [intro]. Function **readcoordinates** store at the end of the UAV vector inside Movement class the coordinates of the first point.

Once we have stored all coordinates of each point is time to lead the UAV across these points. In order to achieve that, like we saw above, is necessary to create a new class called **Movement**, this class is an **osg:NodeCallback** that will allow us to move the UAV in the three coordinates in every frame of the viewer. This new class is declared and defined in the file **movement.h** A.8. As we saw in the section Callbacks, we use the **Movement()::operator()()** that contains a call to **traverse()** at the end. This is a member method of the **osg::NodeCallback** class. This call allows the update traversal (**osgUtil::UpdateVisitor**) to traverse the current group node children. In our case, the code above the call **traverse()** is executed previous to traverse the current group node children thereby updating the position and orientation of the UAV in each frame of the viewer.

It was decided to create a new parameter called *speed*, this parameter is responsible to control how much progress an UAV in each frame. The higher value of the *speed* greater the speed with which the UAV will move across the

scene. Empirically tested values, 400 seemed the most appropriate. Needless to say that if we want to increase or decrease the speed we can do it with changing the value of the variable *speed*.

We can see the solution to the movement of each UAV in a flowchart in the Figure 5.28:

It is very important to realize is not a conventional loop. Iterations do not follow the normal order, when one ends other begins; but in this case each iteration is performed with each frame of the viewer, that is the time in which the Callback is called.

As shown in Figure 5.28, when the UAV reaches a new point, it rotates on itself to face the next point on the list of coordinates. In earlier versions of the program the UAV arrived at each point and was faced directly with the target, losing any sign of continuity. For this reason I decided to do the UAV rotate over itself to face its next target, but this way to rotate is only possible if the UAV is a helicopter (like in our case). In future implementations, it could be possible to do this orientation with the UAV in movement, allowing in that way that UAV could be a plane.

Last condition, is responsible to know when we are in the first point (or home position). As I said above, last coordinates stored into the UAV vector of the class **Movement** are the coordinates of the first point. Then, when we are in the last member of UAV vector we are in the first point. Saying we are at the first point when we are at the last is like we build the infinite loop. While program is running, UAV describe a path that includes the points that were spent through the txt file, linking the latter with the former, until the window is closed.

At this point, it was only necessary to add new functionality to the scene drawing the **osgGtkviewer** tools. We will see more about this in the next section.

New features

Now, had the terrain and the UAV movement, was the time to take advantage of the use of **osgGtkviewer**. When we were talking about it, I commented that it has some pre-programmed buttons in its interface, and with minor changes we can do many things. Moreover, it is possible to have two or more cameras of the same scene with **osgCompositeViewer** like we saw in previous sections.

I decided to include a global variable that allow to the station user to stop the UAV and look around. For this reason I changed the functionality of one of the buttons of **osgGtkviewer**. *Run 120 frames* button became *Loitering mode* button (Figure 5.29). When the button *Loitering mode* is pressed, the global variable loitering_mode change its value, in that way when its value is 1, the callback in **movement.h** A.8 does not enter the loop and retains the values of the UAV xyz, so the UAV will remain standing. Then, when

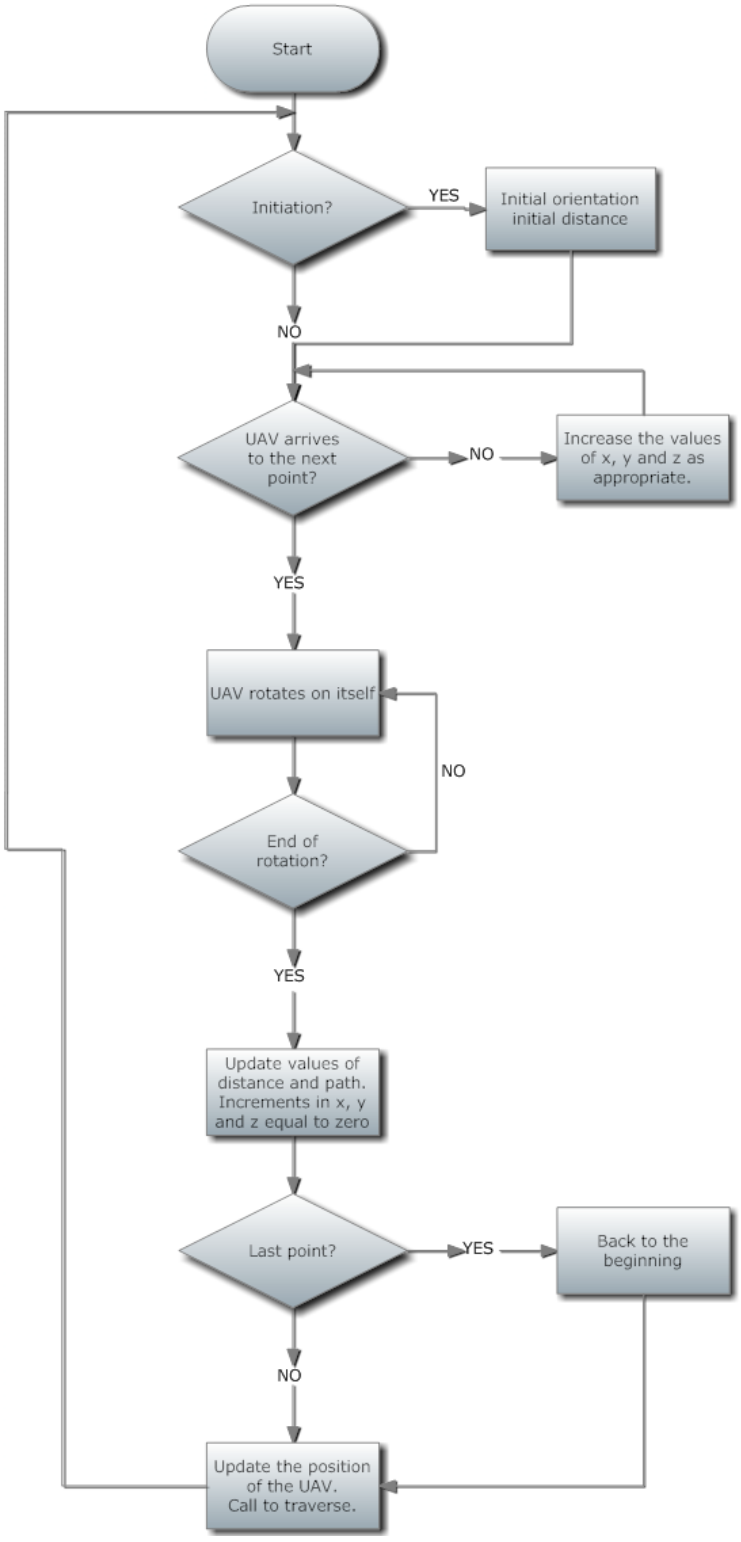

Figure 5.28: Movement flowchart

the button *Loitering mode* is pressed again loitering_mode again be zero and the UAV follows its normal path.

Figure 5.29: Changes in the buttons

We wanted to add the Head-Tracking tool to the 3D simulation. Our aim was to control a camera with the movements of our head. But first of all, we needed a way to communicate the Head-tracking system response with the program in OSG. To solve this problem was used YARP. Now we are going to see an introduction to YARP:

YARP [17] is plumbing for robot software. It is a set of libraries, protocols, and tools to keep modules and devices cleanly decoupled. It is reluctant middleware, with no desire or expectation to be in control of your system. YARP is definitely not an operating system.

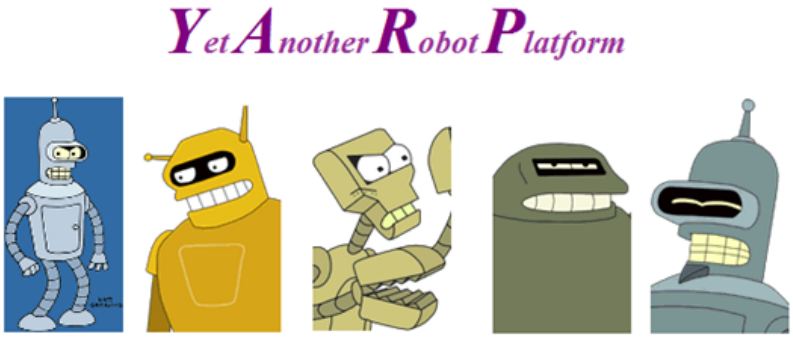

Figure 5.30: YARP

Robot projects are often evolutionary dead ends, with the software and hardware they produce disappearing without trace afterwards. Common causes include dependencies on uncommon or obsolete devices or libraries, and dispersion of an already small group of users. In humanoid robotics, a small field with an avid appetite for novel devices, we experience a great deal of churn of this nature. YARP is our attempt to make robot software

that is more stable and long-lasting, without compromising our ability to constantly change our sensors, actuators, processors, and networks. It helps organize communication between sensors, processors, and actuators so that loose coupling is encouraged, making gradual system evolution much easier. The YARP model of communication is transport-neutral, so that data flow is decoupled from the details of the underlying networks and protocols in use (allowing several to be used simultaneously, key to smooth evolution). YARP uses a methodology for interfacing with devices (sensors, actuators, etc.) that again encourages loose coupling and can make changes in devices less disruptive. At the same time, YARP doesn't expect to be in charge; we want to minimize problem of incompatible "architectures", "frameworks", and "middleware" (also known in this context as "muddleware"). YARP is free and open software. Along with many other benefits, the Free Software social contract can speed software development for small communities with idiosyncratic requirements, such as ourselves. YARP is written by and for researchers in robotics, particularly humanoid robotics, who find themselves with a complicated pile of hardware to control with an equally complicated pile of software. At the time of writing, running decent visual, auditory, and tactile perception while performing elaborate motor control in real-time requires a lot of computation. The easiest and most scalable way to do this right now is to have a cluster of computers. Every year what one machine can do grows, but so do our demands. YARP is a set of tools we have found useful for meeting our computational needs for controlling various humanoid robots. The components of YARP can be broken down into:

- libYARP_OS - interfacing with the operating system(s) to support easy streaming of data across many threads across many machines. YARP is written to be OS neutral, and has been used on Linux, Microsoft Windows, Mac OSX and Solaris. YARP uses the open-source ACE (ADAPTIVE Communication Environment) library, which is portable across a very broad range of environments, and YARP inherits that portability. YARP is written almost entirely in C++.

- libYARP_sig- performing common signal processing tasks (visual, auditory) in an open manner easily interfaced with other commonly used libraries, for example OpenCV.

- libYARP_dev - interfacing with common devices used in robotics: framegrabbers, digital cameras, motor control boards, etc.

These components are maintained separately. The core component is libYARP_OS, which must be available before the other components can be used. For real-time operation, network overhead has to be minimized, so YARP is designed to operate on an isolated network or behind a firewall. For interfacing with hardware, we are at the mercy of which operating systems

particular companies choose to support - few are enlightened enough to provide source. The libYARP_dev library is structured to interface easily with vendor-supplied code, but to shield the rest of your system from that code so that future hardware replacements are possible. Check the requirements imposed by your current hardware; YARP will not reduce these, only make future changes easier. YARP has consequently three levels of configuration: operating system, hardware, and robot level. The first level of configuration should concern you only if you're planning to compile YARP on a new operating system. The second level is the hardware. A new addition on an existing platform or a new platform altogether might require preparing a few YARP device drivers. These are to all effects C++ classes that support the methods for accessing the hardware which is normally implemented through function calls to whatever provided by the hardware vendor. This comes typically in the form of either a DLL or a static library. Finally, you can prepare configuration files for an entirely new robotic platform. With a YARP server we can communicate Head-Tracking tool with the OSG program. But now, we needed to define a new button to activate the Head-Tracking mode in the **osgGtkviewer**. The chosen button was *Open File* which was replaced by *Head Tracking* Mode button (Figure 5.29). In this point needed a way to affect to the camera movements only when we press the *Head Tracking Mode* button. I created a new global variable called **manuallyPlaceCamera** that is a Boolean variable, if is false the Head Tracking mode is deactivated and when it is true the camera should move with the head movements if the program is receiving data form Head Tracking system. But, how I can do it? I was thinking of a solution and I find one, I would create a child of the camera UAV (***followerOffset*** in the program) **HTmode** that it was a pointer to a **MatrixTransform** node, and when I pressed the *Head Tracking Mode* button, I change the rotation of that node (**HTmode**) with the head movements. In the same way as above if we press again the *Head Tracking Mode* button, the Head Tracking system will be deactivated and the system returns to normal. We can see this in the following piece of code of **main.c** A.1:

```
Plane.get()->addChild(followerOffset);

// Now we are going to create a new Node, mLeft which will
// be a son of the followerOffset and will allow us to rotate
// when the Head Tracking mode will be activated.
osg::ref_ptr<osg::MatrixTransform> HTmode = new\\
osg::MatrixTransform;
followerOffset->addChild( HTmode );

// Callback in rotateHt.h that update the rotation of the
// camera when we activate the Head-tracking mode and while
// we are receiving data from the Head_tracking system.
HTmode->setUpdateCallback( new RotateHT );
```

```
// tm will be the nodetracker manipulator asociated to the UAV
// protagonist
osgGA::NodeTrackerManipulator* tm = new
osgGA::NodeTrackerManipulator;
tm->setTrackerMode(
osgGA::NodeTrackerManipulator::NODE_CENTER_AND_ROTATION );
tm->setRotationMode(
osgGA::NodeTrackerManipulator::ELEVATION_AZIM);
tm->setTrackNode(HTmode);
```

Like we can see above, the camera manipulator is associated with **HTmode**, which is a child of ***followerOffset***, which in turn is child of Plane.get (the UAV model). In this way when Head-Tracking mode is deactivated the movements of the camera are associated to the movements of the UAV and mousse or touchscreen camera manipulator. As, **HTmode** is a descendant of Plane.get all movements and rotations affecting the UAV will too **HTmode** and consequently may be seen in the viewer (tm is associated to **HTmode**).

Now, is time to see how we rotate the camera. As in previous cases, it is necessary to use callbacks. Similarly we did before to move the UAV we must define a new class type **osg::NodeCallBack** called in this case **RotateHT**. This class has a method called **operator()()** that allows us to rotate the camera in function of Head Tracking data. The code is in **rotateHT.h** and is very simple:

```
#include <osg/NodeCallback>
#include <osg/MatrixTransform>
#include <iostream>
#include <math.h>
#include "../../../../head_tracking/cyarpheadpos.h"
#include "../../../../head_tracking/cyarpheadposprocessor.h"

using namespace std;

// We define a global variable that will allow us to access to
// the Head-Tracking system.

bool manuallyPlaceCamera = false;

// We define a variable type CyarpHeadPosProcessor that will
// receive data from the Head-Tracking thanks to yarp server.
CYarpHeadPosProcessor headposProcessor;

class RotateHT : public osg::NodeCallback { public:
    // This method will be called in each traverse update.
    virtual void operator()( osg::Node* node,
            osg::NodeVisitor* nv )
    {
        double roll, pitch, yaw, x, y, z;

        // With the next condition we will know if the
        // Head Tracking Button was pressed and we are
```

```
        // receiving data from Head-Tracking system.
        if (manuallyPlaceCamera && headposProcessor.popHeadPos
        (roll, pitch, yaw, x, y, z) == 0)
        {
        osg::MatrixTransform* HTmode =
        dynamic_cast<osg::MatrixTransform*>( node );
        osg::Matrix mR, mT;
        // Translation Matrix will be 0, because we want only
        // the rotation
        mT.makeTranslate( 0., 0., 0. );
        mR.makeRotate(
                osg::DegreesToRadians(0.0), osg::Vec3(0,1,0),
                // roll, deactivated
                osg::DegreesToRadians(-pitch), osg::Vec3(1,0,0) ,
                // pitch
        osg::DegreesToRadians(1.5*(-yaw)-33), osg::Vec3(0,0,1) );
        // yaw

        // now define the transformation matrix
        HTmode->setMatrix( mR * mT );

        // Continue traversing so that OSG can process
        //   any other nodes with callbacks.
        traverse( node, nv );
        }
    }
};
```

Like the code shows, first of all is to check if Head-Tracking mode is activated and that we are receiving data from it. If the condition, we store the pitch and the yaw, (roll is deactivated because we are simulating a pulp and tilt of a camera). And the translation matrix will be zero (move not, only want to rotate). We apply the transformation matrix to HTmode and execute the traverse call.

If we rotate the head left or right much, we can lose sight of the window so we could not turn our head left and right a lot. To avoid this we introduce a scaling in the yaw, thanks to this, we will see a big turn on the screen as a result of a slight twist of the head

Having done all this, it occurred to us include more than one camera in the scene. Taking advantage of **osg::Compositeviewer** seen in previous section, we decided to include an aerial camera that shows the scene viewed from above. Figure 5.31 illustrates the result:

Figure 5.31: Two views

5.4 Simulation tool

And finally we have prepared the simulation tool. The scene will consist of up to 10 UAVs (you can add more if necessary) flying over an airport, we can distinguish the airport control tower and two hangars. We have two cameras, one in the first UAV (UAV protagonist) and the other an aerial camera to view the scene from above. The associated scene graph considering only 3 UAVs for simplicity can be seen in Figure:

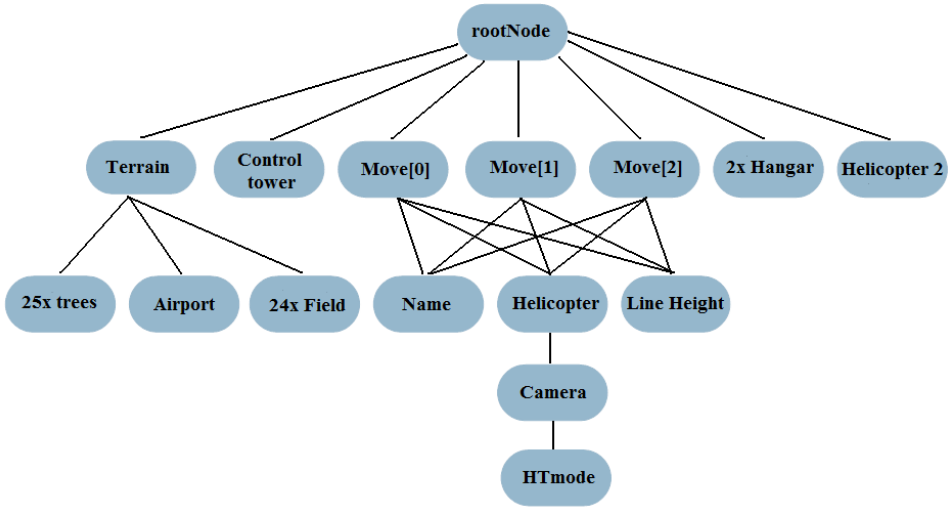

Figure 5.32: Final scene graph

First of all we need to run the YARP server which is responsible for maintaining communications between the program and head-tracking system. To

achieve this, we open a console and type the following:

yarp server

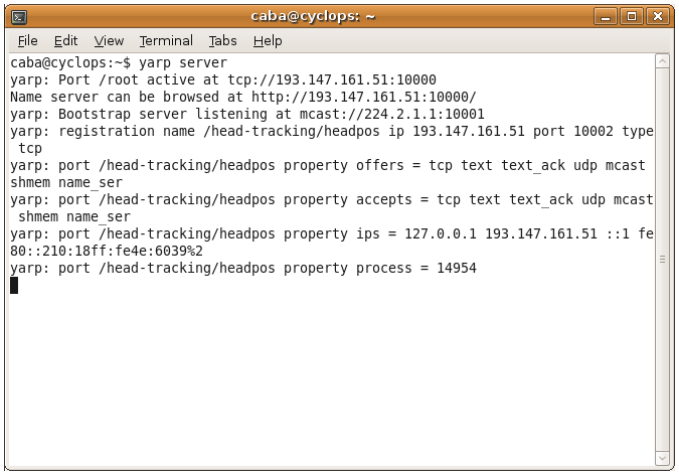

Figure 5.33: YARP server launched

Next step is to boot the Head-Tracking system. We need now to turn on the Headphones (for LEDs to be enlightened), turn on the IMU and its receptor, and start the Head-Tracking program typing in the appropriate folder:

./head_tracking

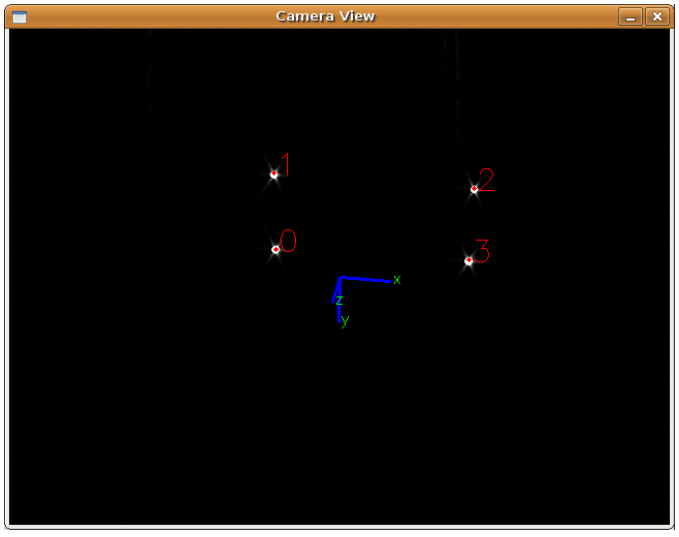

Figure 5.34: Head-Tracking system

In the final scene have been added new functions like that the UAVs take their numbers over and this will move with them, in that way we can identify

them on sight on the screen. Moreover, this text has the property that we see it with the same size regardless of the distance that the UAV is and regardless camera angle. On the other hand, was difficult to distinguish the height of each UAV, so was decided to include a line from UAV to the terrain with the aim to make easy to know the height of each UAV in the scene. Now, next figure illustrates a simplified diagram of the program performance:

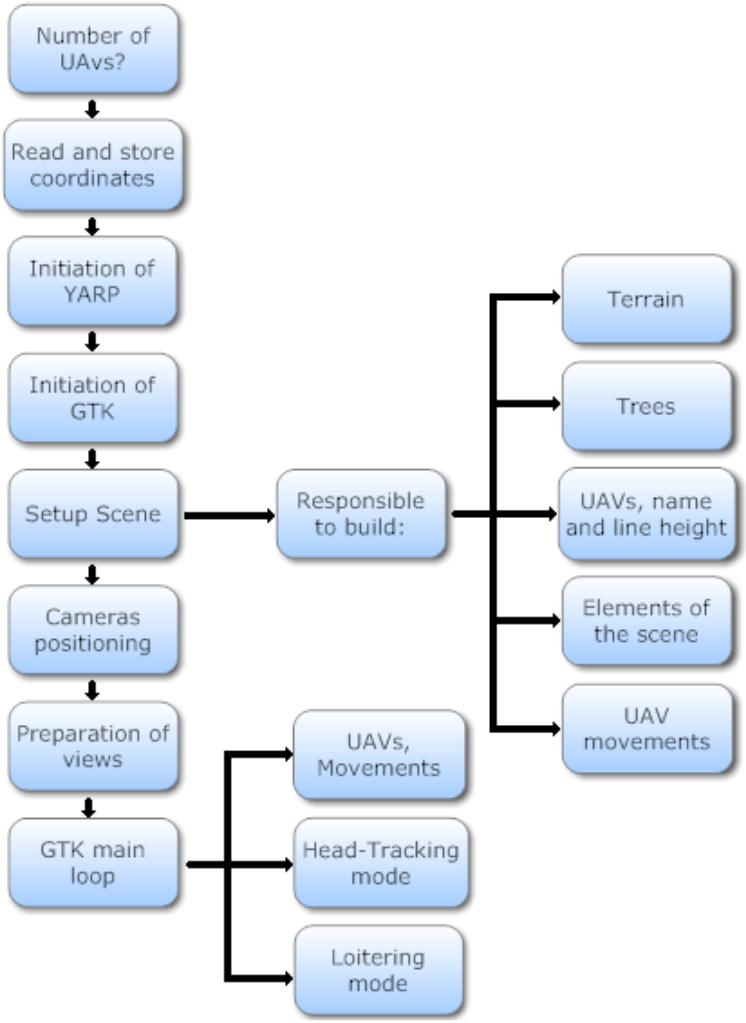

Figure 5.35: Diagram of the final version of the program

By looking at the diagram above I will explain briefly how the program works. First of all, the program will ask for the number of UAVs. Then, if the number is between 1 and 10, the program read the coordinates from coordinates files and store them in the corresponding place. After that,

initialization of YARP is done in order to exchange data with the Head-Tracking system. Also need to initialize GTK. Later, the program will run a group of functions which are responsible to build the whole scene, its mission is to build the terrain, add trees, add the UAVs and, like was said before, the number of each UAV and its height line. Moreover, this group of functions has to add decorative elements on the scene, as are the two hangars, the airport control tower and the helicopter near to hangar. On the other hand, this group of functions is responsible to create the UAV movements too. Once the whole scene is finished, the program will prepare the cameras position (in this version uses two cameras but may be more), one camera aboard the UAV and the other an aerial camera. The next step will be to create two different windows one for each camera. When everything is ready, is time to launch the GTK main that is a loop that allow us to interact with the scene at same time that the UAVs is in movement. Besides, this loop allows us to use the GTK advantages and use them to be able to activate the Head-Tracking system and the "loitering mode". First mode will be activated when we press "Head-Tracking mode" button, and will allow us to move the camera orientation with our head movements. We will need to press the same button again to deactivate the Head-Tracking mode. On the other hand, when we pressed "Loitering mode" button the UAV protagonist will stop, and we will can to explore the scene with the UAV stopped; when we want the UAV to move again we will press the "loitering mode" button again. It is possible to combine the two modes.

The program output can be seen in the following images:

Figure 5.36: UAV view

Figure 5.37: Aerial view

Is important that the new viewer continues to maintain the property of the fake menu we saw in previous sections (Figure 5.38), which can be useful in future for the introduction of new features to the program.

Figure 5.38: Viewer with fake menu