UPDRAFT

THE PROGRAMMERS DOCUMENTATION

Team Leader :

Tomáš Zámečník

Team Members :

Čestmír Houška

Jakub Marek

Bohdan Maslowski

Mária Vámošová

Aleš Zita

Project Supervisor :

Alexander Wilkie

Faculty of Mathematics and Physics,

Charles University,

Prague,

Czech Republic

06-Jun-12

TABLE OF CONTENT

1 INTRODUCTION

There is more to gliding then just jump into the airplane and fly. The ground preparation is nearly as important as the flying itself. And this fact holds not only for commercial or military pilot, but for sport glider pilot too. For instance every competition flight needs to be planned in advance as well as analysed after the flight is done.

The post-flight analysis or so-called debriefing is the process during which the pilot takes the stored flight path recordings and scrutinises the data or re-plays the flight to further analyse the flight or learn from it. As for planning of the competition flight there are sets of pre-defined *turn-points* from which usually the three turn-points are chosen to form a competition triangle. Every turn-point is defined by its world coordinates. During the competition flight, the pilot needs to navigate his/hers airplane through these waypoints in pre-defined order for his flight to be valid. Although this planning can be done by hand, the use of computer program is very convenient. Therefore a need of software that can be used for these tasks has arisen. To our knowledge there are few free and commercial solutions on the market. However the free software doesn't match the quality of the commercial applications and the commercial applications are expensive. One of the widely spread commercial software for this purposes at the present time is a application called *SeeYou* [1] created by the *Naviter* company, which we use as an inspiration.

Our aim is to create similar (in functionality) open-source product under the GNU license and therefore made this product available to wider glider pilot communities. We try to improve on some of the SeeYou features, which we believe can be done in more user friendly manner. We would not like our product to be compared to SeeYou, but rather taken as a free-of-charge alternative. To achieve this uneasy task we have setup the group of skilled programmers and flight enthusiasts, who has spent many a night and day to realize such a project. We call our project **The Updraft**[1].

Figure 1. The Updraft logo.

[1] http://updraft.github.com/

2 THE UPDRAFT PROJECT

2.1 THE TEAM

This subject was originally proposed by our Project Leader *Tomáš Zámečník*. He, as an enthusiastic glider pilot, has pointed out the lack of high-quality non-commercial glider path planning and visualisation software. Team of people fond of flying was established with *Tomáš Zámečník* as a project leader. Finally, the team consists of 6 people.

Team members:

- **Tomáš Zámečník**
- **Čestmír Houška**
- **Jakub Marek**
- **Bohdan Maslowski**
- **Mária Vámošová**
- **Aleš Zita**

See the "*Appendix A – People*" for further details.

Programming of the Updraft project was divided into several partially separable parts with each of the part dedicated to one team member. Obviously it was impossible to split the workload evenly, so most of the people ended up working on parts of the project in groups and tight cooperation between team members proved crucial. However there always was a tendency for having one responsible person for each of the project component. The Table 1 shows every member main responsibilities. The work was well coordinated and project meetings were held on weekly basis. On several occasions the team brain-storming was organised, resulting in marginal work progress or brilliant issue solutions. For e-mail communication we used services of *freelists.org* [2] for its accessibility and transparency. The established group e-mail address is glideplan_swproj@freelists.org.

Developer	Updraft component	Main responsibilities
Tomáš Zámečník	Turnpoints plug-in, TaskDecl plugin	Leadership and management, Turnpoints plug-in, User documentation, TaskDecl plug-in, Importing files
Čestmír Houška	Core, GUI	Updraft core, Settings, TaskDecl. GUI, Mouse picking
Jakub Marek	IGC plug-in, Core	Core, IGC visualisation, IGC parsing, Translations, Bottom panel interface, OpenFile dialog
Bohdan Maslowski	GUI, installation	Mac Updraft version, Installation packages for all platforms, 2D Map mode, Camera manipulation
Mária Vámošová	IGC plug-in, GUI	OSG Earth implementation, Map layers visualisation, left pane interface, IGC graph drawing, IGC statistics, Scene manager
Aleš Zita	Airspaces plug-in, documentation	Airspace plug-in, documentation, turnpoint labels, airfields visualisation

Table 1 : Team members main responsibilities.

2.2 Technical Details of Development

The programming language we choose for the Updraft implementation was C++ for both the object oriented characteristic of the language and overall advancement of this language's features and possibilities. To keep the complexity of the language on a manageable level, we decided to use a Google C++ style guide [3]. Google C++ style guide is one of the widely used open-source set of suggestions and recommendation on how to maintain the readability of the code. The style checker was incorporated to the build of the project to ensure the rules would be followed.

On top of the main programming language we decided to use Nokia Qt framework [4] for its multiplatform user interface capabilities. Nokia Qt is a development framework used for creation of applications for many platforms including Windows, Linux, Mac OS, Symbian and many others. One of the goals we set was to have Updraft usable under not only Windows, but under Mac OS and Linux operating systems as well, because these are 3 most common operating systems nowadays.

The program was developed on Windows and Linux platforms and tested on Mac. As a development environment we used the Microsoft Visual Studio 2008 [5] under student license on Windows and vim-editor on Linux.

CMake [6] was used as a cross-platform project build system.

2.3 Repository Structure

We used the *Git* [7] version control software and hosted the source code on the *GitHub* [8] web hosting service. We have chosen the GitHub, because it offers free accounts and storing space for open-source project, such as ours, and for very convenient possibility of creating our product's web page. The web-page address of our product can be found on http://updraft.github.com. The project repository is accessible via Internet on https://github.com/updraft. The repository structure of our project is as follows:

- updraft.github.com - Project web page files.
- Updraft - Project main repository folder.
  - Docs - Documentation related files
  - Experiments - Project experiments with 3rd part SW installation.
  - SourcesInstallation - Test files for source code installation.
  - Updraft - Source code repository.

2.4 Project Specification

2.4.1 Environment

The goal of this endeavour is to create multiplatform desktop application capable of serving as a glider pilot tool for flight planning and post-flight visualisation. The target platforms are Linux, Microsoft Windows or Mac OS. Connection to the internet is not required, but enables some of the functionality.

2.4.2 Supported Languages

Application is distributed in Czech and English localisations.

2.4.3 Negative Specification

The software is not intended to be used by competition directors or referees or for judging the achievements of any official goals. Application will not be focused on detailed flight planning like calculating optimal speed, water ballast, final glide etc. The application will not run on mobile devices and will not serve as an on-board computer.

2.4.4 List of Features

The features include the visualisation of 3D maps including the terrain height, visualisation of flight relevant data, such as airspace divisions, turn-points, airfields, etc., next to the flight planning and visualisations of recorded IGC files.

The tools we contained within the final product are:

- flight planning (called task declaration)
- airspace visualisation
- turnpoints and airfields visualisation
- flight path visualisation with colour-coded information about
  - speed
  - vertical speed
  - altitude
  - airspeed

2.4.5 GUI PRINCIPLES

Most of the action is done by mouse.

2.5 PROJECT TIMELINE

The project work and team member's personal assignments are thoroughly documented in the *Wiki/Meeting notes* section of project GitHub repository. Following is an abstract of the project timeline.

Time period	Objective
October 2011	Qt introduction, application architecture design
November 2011	Plug-in specification and design, plug-in interface , 3rd party libraries tests, Core design
December 2011	Project Technical Specification, Project timeline, Mac availability, Core creation, osgEarth integration, Plug-in interface update
January 2012	File import, Plug-in creation start, Settings implementation, map visualisation, 3rd party data sources integration, GUI creation
February 2012	Additional map layers, Plug-in development, Plug-in file parsers, Colibri interface testing, 1st alpha version of the application
March 2012	Application improvements, Plug-in development, Application testing and bug fixes, Mouse-picking
April 2012	Installation packages creation, OSGEarth upgrade to v2.2 (contains fixes), documentations writing, application testing, bug fixes
May 2012	Documentation writing, defence preparations, application testing, bug fixing, installation packages for all the platforms creation

Table 2. The Project Timeline.

3 THIRD PARTY SOFTWARE

As this is an uneasy task, we made our work a little bit easier by utilizing several 3rd party frameworks, which help us to create the application. These are :

1. *Nokia Qt* [4] – the multiplatform User Interface
2. *OSG* [9] (the open scene graph) – as a OpenGL interface for 3D geometry visualization
3. *OSG Earth* [10] – as a tool for the world map visualization
4. *GeographicLib* [11] – for exact ellipsoid distance calculation

3.1 NOKIA QT

Nokia Qt [4] is a cross-platform development framework used for creation of applications with graphical user interface (GUI) for many of the operating systems including Windows, Linux, Mac OS, Symbian and others. Nokia Qt is a free and open source software distributed under the terms of the GNU Lesser General Public License. As it has been always our goal for Updraft to be multi platform application, we decided to use the Nokia Qt framework because it is ideal for this task. Moreover, the Nokia Qt is known to be able to incorporate the OSG efficiently. Nokia Qt uses standard C++ but makes extensive use of special code generator called the *Meta Object Compiler*, or *moc* together with several macros to enrich the language. [12]

3.2 OSG

The *OpenSceneGraph* [9] is an open source cross-platform 3D graphics application programming interface used in fields such as visual simulation, games, virtual reality, scientific visualisations and modeling. The OSG is distributed under OpenSceneGraph Public License based on LGPL licence. It uses the *OpenGL* [13] for 3D graphics acceleration interface. We use the OSG as a base for the OSGEarth framework, which we decided to use for the worldwide map visualisation.

3.3 OSG EARTH

The *OSGEarth* [10] is a terrain rendering toolkit written in C++ and distributed under LGPL license. It is developed by Jason Beverage and Glenn Waldron as a visualisation toolkit for the map, elevation and vector data. OSGEarth utilises the OSG framework as the 3D graphics interface.

Figure 2. The OSGEarth framework example.

3.4 GEOGRAPHICLIB

The GeographicLib is a small C++ project for solving geodesic problems. The library is licensed under the MIT/X11 License. In our project we use a small part of the code for very exact computation of distance on the geode surface.

4 THE THIRD PARTY DATA

4.1 INTRODUCTION

So far we described the environment we chose for our project development. OSGEarth is a superb framework for visualising the map imagery and elevation data of various sorts. For map data processing we decided to use the online/caching strategy. As a source we use the data sets which are available online because of minimal necessary installation packaging. We cache the data once downloaded from the internet to limit the bandwidth requirements to minimum level and boost up the speed of the map drawing. In this way the application is capable of displaying the map data from all over the globe without the necessity of all the data being stored locally. This solution also prevents the distribution package to be oversized. Note that these data packages are enormous in size for obvious reasons. Therefore the internet connection during the application run is highly advisable but not essential once the map data are cached. We distribute the application with a basic set of cached map data in reasonable size to resolution ratio for users without internet connection to be able to use the application. Additionally the application allows user to pre-cache map data with the map caching tool for later off-line application usage.

Two types of data sources are necessary for visually pleasing display of the world, the map imagery data and the elevation data. There are several free data sources available online for the map imagery, but not so much for the elevation data.

4.2 MAP IMAGERY SOURCES

In order to display the world map we need to use a free database of map imagery data. This data set needs to be in one of the format supported by the OSGEarth framework. The complete list of the OSGEarth drivers capable of process particular data sources are listed on the OSGEarth web page[2]. The configurations of the particular data sources are written in the `'*.earth'` file stored in `/bin/data` directory.

In our case we wanted to have at least two basic views of the virtual world. We pick the sources in following manner: one to be very simple for the sake of the visualisation clarity and the other to be the "realistic" view, i.e. the photomaps with the elevation displayed as a terrain displacement.

Map imagery data sources used:

- OpenStreet map [14] - free worldwide map tiles database hosted on under open licence
- ArcGIS online map service [15]- free[3] worldwide satellite tiles imagery database

[2] http://osgearth.org/wiki/TileSourcePlugins
[3] http://resources.esri.com/arcgisonlineservices/index.cfm?fa=content

- ArgGIS online map service [16] – free[4] worldwide topographic map tiles database

Figure 3. OpenStreetMap source.

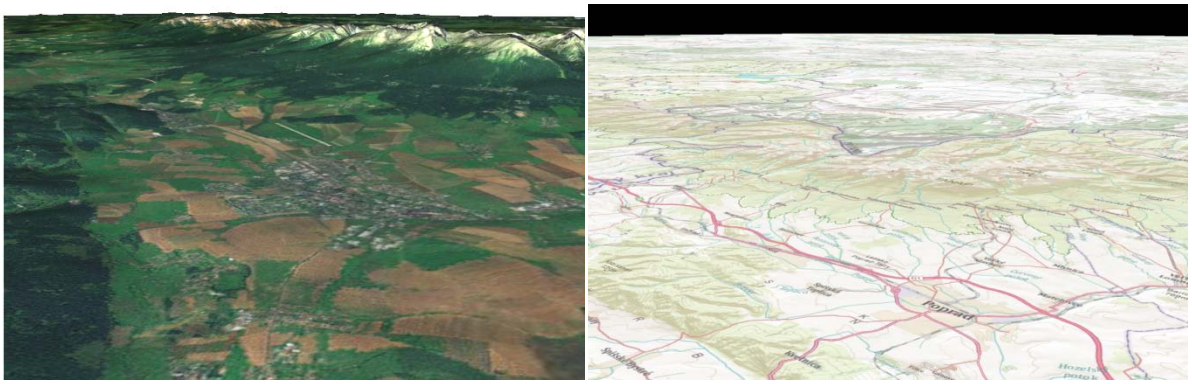

Figure 4. ArcGIS online map service.

4.3 Elevation Data Source

As for the elevation (so-called heightfield) data source we use the Ready Map elevation data source [17]. Because this data source is designated for development purposes only, we asked the author for permission to use this data set for our project. The permission was granted and the corresponding e-mail is attached to this document. See Appendix C – Elevation dataset usage confirmation.

[4] http://resources.esri.com/arcgisonlineservices/index.cfm?fa=content

5 PROGRAM ARCHITECTURE

The application is designed as a core/plug-ins architecture with the core being responsible for user interface, map visualization, settings, file importing and plug-in handling. The plug-ins are linked as a dynamic libraries to the main project and represent the main features functionalities. Plug-ins use the core functions and methods for drawing to the map layer, creating plug-in related menus, mouse click actions, file handling, settings, etc. The core/plug-in communication is represented by a system of Qt signals and back calls between both components. Example scenario is that plug-in needs to react to a user action captured by the user interface (core) such as mouse click on map, item removal or the plug-in map item visibility change.

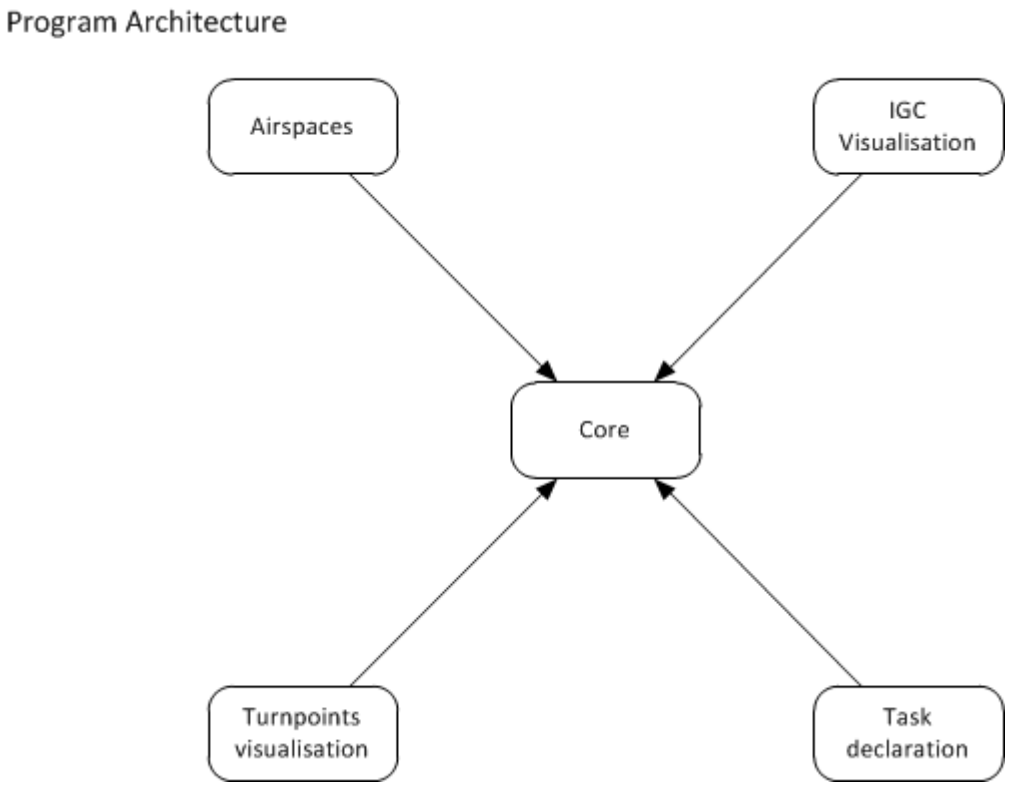

Figure 5. Application architecture diagram.

6 BUILD SCRIPTS

Updraft uses *CMake* [6] to build, test and package the code. *CMake* was chosen because it is a cross-platform tool and is able to use Qt specific compilation steps.

6.1 LAYOUT OF CMAKE BUILD FILES

Directory `/Updraft/` contains the root `CMakeLists.txt` file.

The `/Updraft/src/plugins/` and `/Updraft/src/libraries/` directories both contain a `CMakeLists.txt` that enumerates subdirectories and adds them. Each plug-in and each library then contains a small `CMakeLists.txt` file that sets up the build (using *PLUGIN_BUILD* and *LIBRARY_BUILD* macros) and optionally further adds the directory with tests.

6.1.1 CMAKE MODULES

All *CMake* files in Updraft project use libraries defined in `/Updraft/CMakeModules/`.

Common Build Functions

The most important part of our *CMakeLibraries* is the file `/Updraft/CMakeModules/CommonBuild.cmake`.

The function *GATHER_SOURCES* recursively lists all source files, wraps the MOC and translation files and returns them as follows:

`` `${prefix}_sources` ``	source (`*.cpp`) files
`` `${prefix}_headers` ``	header (`*.h`) files
`` `${prefix}_forms` ``	Qt forms
`` `${prefix}_resources` ``	Qt resources
`` `${prefix}_translations` ``	source translation files (`*.ts`)
`` `${prefix}_translations_wrapped` ``	runtime translation files (`*.qm`)
`` `${prefix}_all_sources` ``	all source files that must be compiled including wrapped MOC objects

GATHER_SOURCES can also skip test directory from the recursive search if it is present.

The macros *LIBRARY_BUILD*, *PLUGIN_BUILD* and *TEST_BUILD* are shortcuts for compiling parts of the project using the common settings.

Locating osgEarth Library

Because *CMake* standard library doesn't contain functions for locating *osgEarth* library, we supplied our own. This code is located in `/Updraft/CMakeModules/FindOsgEarth.cmake`

and matches the *CMake* interface for *FIND_PACKAGE* function, so it seamlessly integrates to the rest of *CMake* code.

Locating osgQt OpenSceneGraph Component

As with *osgEarth*, *CMake 2.8.7* didn't supply find scripts for *osgQt*, a part of *OpenSceneGraph* that takes care of integration into Qt.

`/Updraft/CMakeModules/FindosgQt.cmake` extends the find functions for *OpenSceneGraph*.

Style Checker

Style check is added into *CMake* as a new function defined in `/Updraft/CMakeModules/StyleCheck.cmake`.

File `/Updraft/scripts/cpplint.py` contains Google's *cpplint* with minor modifications to work with our Qt code.

This file sets build paths, adds the directories with source code to processing and handles installation of updraft data.

`/Updraft/src/core/CMakeLists.txt` builds the core.

Improved Wrapping of QObject Header Files

Regular *CMake* code for building *MOC files* from header files requires a list of files as an input. Because we chose to find source files using globs, we can't distinguish which header files contain *QObject* subclasses and therefore must be MOC-ed. To solve this script in `/Updraft/CMakeModules/FilterQOBject.cmake` parses the header files and looks for instances of *Q_OBJECT* macro. If none is found then the file is not passed to *MOC*.

6.2 TESTS

We use *CTest* tool to manage our unit tests. If testing code is enabled, then testing is started by executing the target tests (e.g. *make tests*).

6.3 IMPORTANT CMAKE VARIABLES

Environment variables needed to setup the build:

OSG_DIR	Directory of the *OpenSceneGraph* installation

OSGEARTH_DIR	Directory of the *osgEarth* installation

CMake variables used to customize build:

BUILD_AIRSPACES	Enable building of the specified plug-in.
BUILD_IGCVIEWER	If a new plug-in is added a new variable
BUILD_TASKDECL	appears.
BUILD_TURNPOINTS	
BUILD_TESTS	Enable building of testing code.
SLOPPY_BUILD	If true, don't fail on style checker errors.
QUICK_BUILD	If true, skip style checking entirely.
UPDATE_TRANSLATIONS	If this variable is true, then the `*.ts` translation files are updated on every build.

7 THE CORE

7.1 OVERVIEW

The core provides the basic functionality of Updraft - maps, settings, menus, GUI, mouse actions handling, etc., whereas the plug-ins handle everything else. Therefore, the core has to provide everything that the plug-ins could possibly need. This includes map mouse picking, handling the GUI and menus, saving and loading the plug-in's settings, loading the application data from the correct directory and registering a known file type. All of this functionality can be accessed via core interface. Each plug-in has access to this core interface and can thus use its functions. For loading the plug-ins dynamically at runtime, Updraft uses the Qt's Plug-in system.

7.2 TOP LEVEL STRUCTURE

Top level object of the application is class *Updraft* defined in `Updraft/src/core/updraft.h` and `Updraft/src/core/updraft.cpp`. This class inherits from *QApplication* and is available for the whole core as *updraft* macro. It is responsible for creation of the main application structure.

Below the *Updraft* object there are five managers that handle separate aspects of the application functionality and the main GUI window.

Managers have dependencies between themselves that force the order in which they are created.

The order of initialization is following:

- *Settings Manager* is at the bottom of the hierarchy and needs to be initialized first, because it contains paths to data files and selected language.
- Next the *Translation Manager* is created. It only depends on settings for the current language setting. Translations of plug-ins are loaded at later stage, when plug-ins are loaded.
- Other than the top level objects, there are ellipsoids used for flight statistics and data directory settings that are need initialization at this point.
- *File Type Manager* and main window don't have any initialization-time dependencies between them. So they can be created in any order. However the *OpenGL widget* that is needed in main window is created in the next step. Because of this, the *OpenGL widget* is added after the *Scene Manager* is created at a later stage.
- *Scene Manager* is created next. It depends on the main window, because it needs to add items to the list of map layers.
- Since plug-ins need a core interface to work and core interface uses all of the top level objects, *Plug-in Manager* has to be initialized last. As plug-ins are loaded, their directories are passed to Translation Manager and translations are installed.

- After all top level objects have been installed, command line arguments that have not been processed by Qt are used interpreted as file names and opened.

Apart from initializing the managers, class *Updraft* also provides some application wide values, like path to current data directory and selectable ellipsoid models for distance calculations.

7.3 Plug-in Base Class

Every plug-in in Updraft has to be derived from the *PluginBase* class. This class provides several handler methods that can be overridden by each plug-in to provide the desired behaviour. These methods allow the plug-in to initialize data upon load, de-initialize data when the plug-in is unloaded, and add context-menu items into *MapObject*-related context menus (more information about map objects in section Map Objects), react on left mouse clicks on map objects and also process a file that is being opened by the core. Each plug-in should also provide its name by overriding the method *getName()*.

The *PluginBase* class as a Qt plug-in interface is using *Q_DECLARE_INTERFACE*.

7.4 Core Interface

Interface *CoreInterface* enables the plug-ins to communicate with Core. It is passed to the plug-in during its initialisation phase (every plug-in gets its own instance) and it contains pointer to the plug-in. This makes it possible to attribute all calls to the source plug-in (although this option is currently not used). *CoreInterface* is implemented by a core class *CoreImplementation*, however this class contains only thin wrappers around various parts of the high level managers.

7.5 PluginManager

On the side of Updraft Core, the class *PluginManager* takes care of loading plug-ins. The public interface of plug-in manager is fairly limited, only listing all loaded plug-ins and retrieving plug-in instance or plug-in directory by name.

7.6 Translations

Updraft uses *Qt Linguist* for translations. The source strings in the application are written in English, and every other language gets its own translation file.

7.6.1 Translation Files

Using the *lupdate* tool, Linguist keeps `'*.ts'` files generated from the source code of the application. These files can be edited using the translation tool and during build they are converted to binary `'*.qm'`.

In our application both the updates of `'*.ts'` files and the generation of `` `*.qm` `` files is controlled by CMake scripts.

7.6.2 *LOADING OF TRANSLATIONS*

There is a specialized *QTranslator* instance for every plug-in and for core. During application start-up the translation manager reads the setting that selects the current language and attempts to load the corresponding `` `*.qm` `` files. There is a slight difficulty when registering the language setting, because registration needs a description for the setting and there is no way to translate it at that stage. This problem is avoided by first creating the setting with empty description and setting the translated description after the language is loaded.

7.6.3 *AVAILABLE LANGUAGES*

When the translation manager lists available languages it lists all `` `*.qm` `` files in a directory of each plug-in and in the directory of the main updraft executable. Base names of these files and "e*nglish*" are used as names of languages.

7.6.4 *RUNTIME TRANSLATIONS*

Linguist supports changing translations during runtime using the *languageChange* Qt event. However every string has to be translated as a reaction to this event which is problematic especially for *QAction*`s and for strings generated during initialization of plug-ins. For these reasons Updraft only loads translations on start-up.

7.7 MAP DOWNLOADER

Map downloader is a utility for downloading maps for a user defined area. In fact, it pre-seeds the map-cache used by the program. It is a simple class that provides user interface for defining the area and the level of detail, with an option to fill the values to current view. Current view is got by the intersector.

Cache seeding is implemented by calling *osgEarth::CacheSeed* class.

7.8 FILETYPEMANAGER

The *FileTypeManager* is a part of the Updraft core, which main functionality is a file management. It is represented by the *FileTypeManager* class. Each of the plug-ins registers its file types by file extension name to the *FileTypeManager* using the *registerFileType()* method. From that moment the class remembers which plug-ins are using which file types. One part of this class functionality is the opening of new files.

The *FileTypeManager* is capable of invoking the *OpenFile* dialog upon user request. After user picks a file or more, the *FileTypeManager* copies the designated files to imported files directory. If the directories do not exist, they are created. These directories are located in `/data/` directory. Particular name of the plug-in import directory is given by the file type registration information provided by each of the plug-ins having the file type registered. Additionally, the *openFile()* sub-routine of each of the plug-in that has the file type registered is invoked. Should there be more than one plug-in registering the same file extension, all of the directories are created and the file is copied to all of them, plus all of the plug-in *openFile()* procedures are called.

7.9 The Settings Subsystem

The settings subsystem serves for saving the settings for the core as well as plug-ins. Settings are organized in groups and can be either normal or hidden. They are saved in a XML file and they can be modified using the built-in settings dialog.

The whole settings subsystem is built around the *Model-View-Delegate* software pattern. This pattern is used very prominently in Qt, so we decided to use it as well to be compatible with Qt. The pattern defines three eponymous types of classes. Data models contain the actual data, *Views* display the data from various perspectives and *Delegates* control the way the data is viewed and edited in the *View* classes. In our settings subsystem, these classes correspond to classes *SettingsModel*, *SettingsDialog* with its two *Views* and the *SettingsDelegate* class.

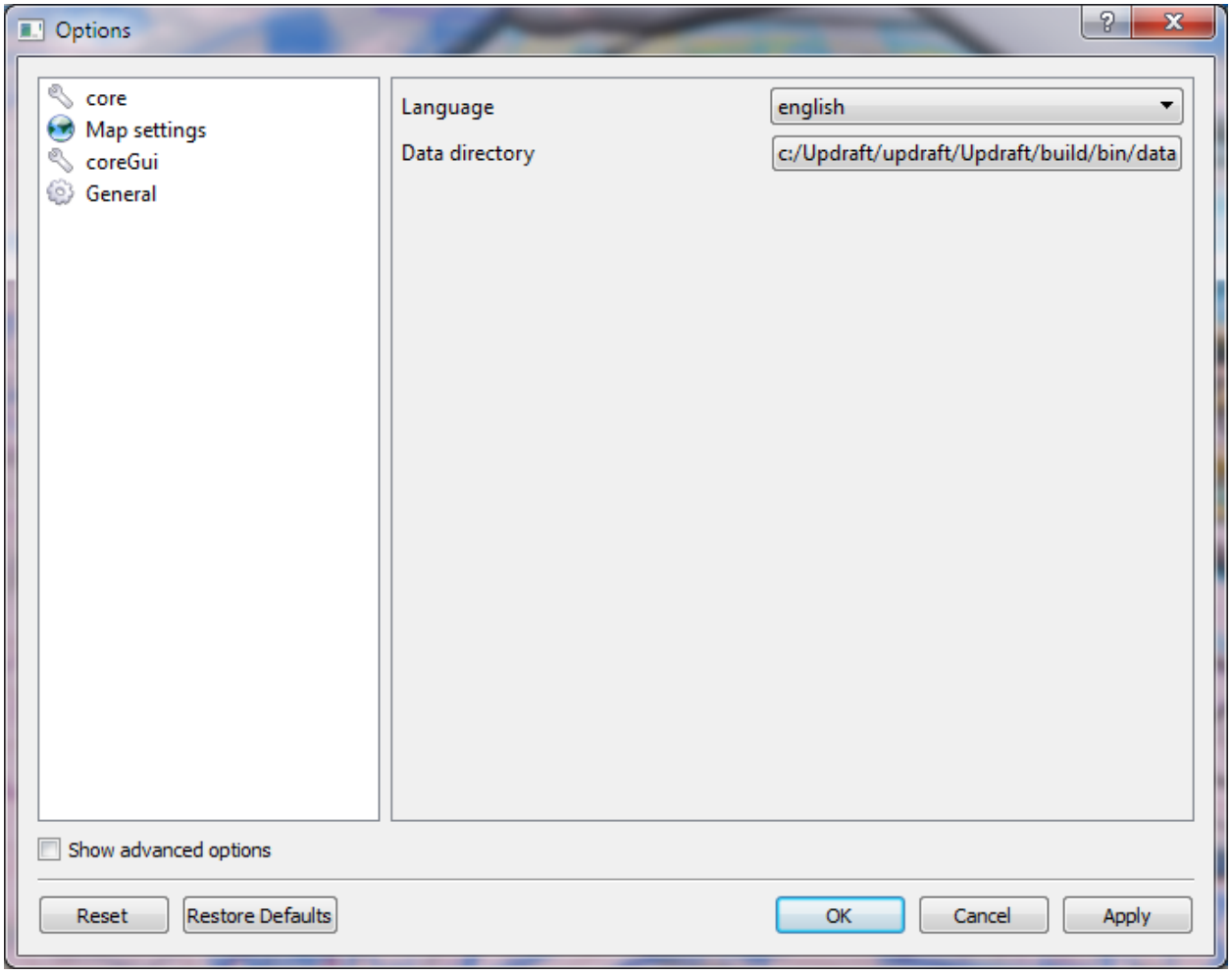

Figure 6. Settings dialog example.

7.9.1 *Settings Manager*

Settings manager is a front-end to the settings subsystem that allows adding settings groups and single settings. It relegates saving and loading of the settings to the settings model, holds the pointer to the settings data model (and also handles the creation and destruction of the model) and also creates the settings dialog, when the corresponding menu action is executed.

When a setting is created multiple times with the same name, all the callers get a reference to the same setting via an instance of class *BasicSetting*. The settings manager holds an associative container of all these instances and whenever a setting value is changed, it uses this container to lookup all these instances and notify them about the change.

7.9.2 *Setting Interface*

BasicSetting is actually just a specific class that implements the *SettingInterface* and whoever wants to get a handle to a setting gets just a pointer to this interface. This is so that we can hide the implementation of the individual settings from the callers. It will be also possible to have multiple implementations of the settings in the future.

The interface allows its owner to get the value of the setting, set the value, set the description for the setting in the settings dialog and set the flag that indicates that changing the setting's value will need Updraft to restart to take effect. It also contains a method that registers a call-back that will be called whenever the value of the setting changes.

7.9.3 SETTINGS MODEL

The settings data model is an implementation of the Qt *QAbstractItemModel*. This model holds the settings in a *QDomDocument*, which is Qt's class that handles *Document Object Model* documents. Our model also contains a hierarchy of *SettingsItem* instances that contain *QDomElement* instances. According to the Qt documentation, these objects internally contain pointers to yet another hierarchy that represents the XML document. However, our data model needs to create model indices and these indices can only contain pointers. These pointers cannot point to the *QDomElement* objects, because no one would be able to delete them. Therefore, our model creates indices that point to our own hierarchy of *SettingsItems*.

Our settings data model represents the setting groups by top-level items and the individual settings by items that are children to these top-level items. The settings can contain various data, representing the setting's name, description, decoration (i.e. icon) or default value. These data roles are a bit different from the Qt's default data roles and we re-defined the data roles so that they have better-suited names.

7.9.4 SETTINGS DIALOG

The settings dialog allows the user to change values of all the settings. The hidden setting groups are initially not shown and can be revealed using a check-box. The dialog contains two views: top view and bottom view. The top view displays only the setting groups and when a setting group is clicked, it sends a signal to the bottom view. The bottom view displays the individual settings and also creates editors that can be used to change the value of each setting.

The dialog contains several buttons at the bottom. All of them are standard buttons from Qt's dialogs. They serve for confirmation/dismissal of the dialog, reset of the settings in the current group, application of changes in the current group and reset of all settings to their default values.

7.9.5 SETTINGS TOP VIEW

SettingsTopView class displays the top-level items in our data model. It displays them using their description and their icon. It can filter the setting groups according to whether they are hidden and/or empty.

7.9.6 SETTINGS BOTTOM VIEW

The bottom view class of the settings displays the individual settings with their editors and changes between the settings groups when they are clicked in the top view. Each editor is created using a *SettingsDelegate* class that overrides editors for some of Qt's types. It contains Qt slots to commit and reset values in editors for settings in the current group. These slots are connected to signals from the corresponding buttons of the settings dialog.

7.9.7 SETTINGS DELEGATE

The settings delegate is used so that we are able to control creation of editors for the different data types of settings. It also contains methods that transfer the data from the editors to their corresponding settings and vice versa.

7.9.8 CUSTOMIZED EDITORS

We use the settings delegate to create editors for colours and filesystem directories. Colours are edited using the *ColorEditor*, which is a colour button that can be clicked to display a colour chooser. Directories are edited using Qt's standard file open dialog.

7.10 SCENE

The main part of the project is the 3D scene. The scene is handled using *OpenSceneGraph* library.

7.10.1 SCENEMANAGER

The main scene is handled by *SceneManager*, which creates, and holds the widget for drawing, and holds current *osg::Camera*, and *osg::Viewer*.

The scene is a degenerated-tree, (? ze proste tam mozu byt diamant-shapes?). The root of the scene has the following children:

- node with the map geometry
- one *osg::Group* node, available with public getter, for adding geometry into the scene
- Upon request, it creates a *osg::Group* node, and adds it to the root. This is used by *MapLayerGroup* instances, so that every map layer group has its own sub-tree.

The *SceneManager* also holds the *osgEarth::Util::ElevationManager* instance. The elevation manager is then available upon request, to compute the elevation at given point. The elevation manager is associated with the first map that has en elevation layer. If there is no map with an elevation, the elevation manager is set as NULL pointer, so that it's clear that no elevation is available.

SceneManager also provides an *osgEarth::Util::ObjectPlacer* instance, that has function which transform an lat-lon-height point to the world coordinates, and vice versa.

The map geometry is handled by having a list of *MapManagers*, each one associated with a map node. The *SceneManager* holds the index of the *activeMap*, and manages switching between the maps. During switching, it also registers, and unregisters the nodes for user clicking.

7.10.2 MAPMANAGER

MapManager is a class that provides an interface to interact with the map. Each map is created from an .earth file that specifies its layers, geometry, path to the cache, etc.

MapManager provides a function to get all layers from the map, in form of a list of *MapLayerInterface* objects. It also can be queried, whether there is an elevation layer in the map, for the *SceneManager* to know if this map can be used for the *elevationManager*.

MapManager also holds the map-manipulator associated with current map.

7.10.3 MAPMANIPULATOR

MapManipulator is a child of *osgEarth::Util::EarthManipulator*, that extends this manipulator with multiple functionalities.

Apart from the mouse and keyboard controls, the manipulator has a method that checks, whether the camera is looking perpendicularly at the terrain (+- some range), and in that case, switches the camera to orthogonal projection. This method is called every frame by the *SceneManager*.

7.10.4 SCENE DESIGN

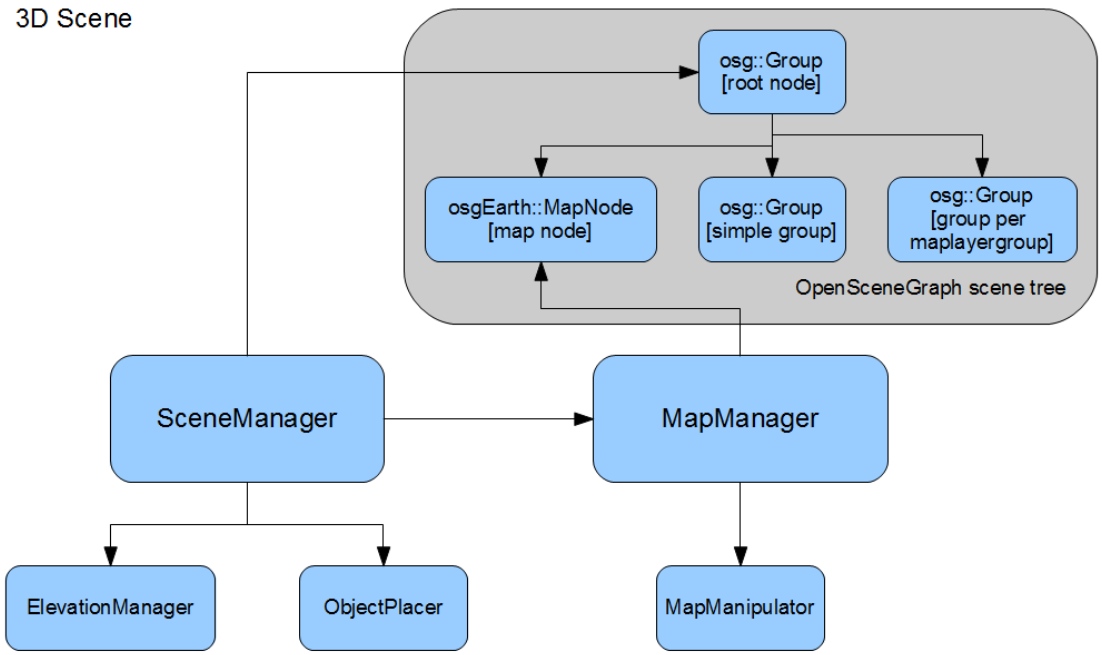

Figure 7. Scene design graphics.

7.11 MAP LAYERS

Class *Maplayer* is an abstract class representing item in the left pane. They hold *OpenSceneGraph* objects that compose the visible objects in the map view and *QTreeWidgetItem*'s that are displayed in the left pane. Map layers are hierarchically organized into groups. The top level group is owned by the main window, this group doesn't have any visible tree widget item.

Map layers are created using methods of map layer group. Map layer will be deleted when the group owning it is deleted, or they can be either deleted manually using operator *delete*.

7.11.1 MAPLAYER SUBCLASSES

There are two subclasses of *MapLayer* that can be instantiated. There is a possibility of extension by adding for example model map layers -- geometry that is automatically generated by *OpenSceneGraph* and placed on terrain based on vector geodetic data.

NodeMapLayer

NodeMapLayer holds a single node of *OpenSceneGraph* geometry.

MapLayerGroup

MapLayerGroup creates the hierarchy of map layers. Each map layer group contains a set of child map layers. *MapLayerGroup* contains the *osg::Group* with the geometry of all child layers.

7.11.2 Interfaces

Plug-ins access map layers using the interfaces *MapLayerInterface* and *MapLayerGroupInterface*. Because a *QObject* subclass is needed to use Qt signals and slots, the interfaces use virtual methods that call connect only on the specific implementation of the object. Similarly for emitting the signals there are virtual methods *emitChecked* and *emitContextMenuRequested*.

7.11.3 Checking and Un-checking Tree Items, Context menu

Other than keeping the root group, main window also has a mapping between tree widget items and map layers. When a map layer's check box is clicked or context menu is requested, main window converts the tree widget item to map layer using this mapping and emits the corresponding map layer's signal.

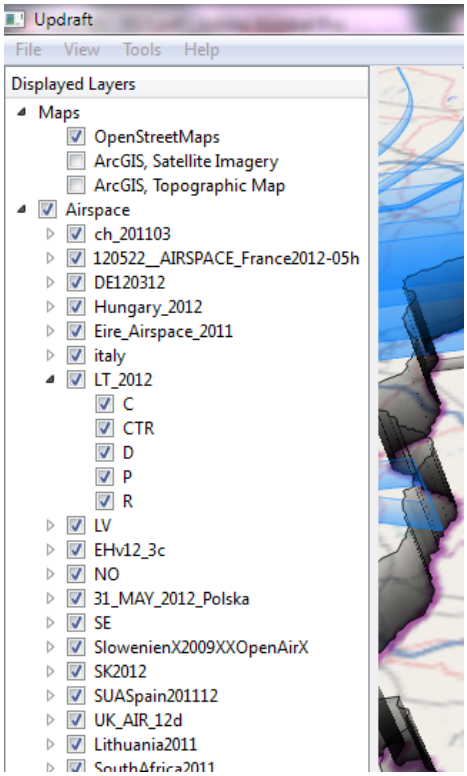

Figure 8. Tree menu example.

7.11.4 Visibility Switching

When a map layer is checked or unchecked in the left pane, it emits a signal *checked()*. This signal must then be handled and a slot *setVisibility()* activated accordingly. This gives the owner of the layer a chance to abort the switch.

However because most map layers the automatic behaviour is enough, there is a method *connectCheckedToVisibility()* that only connects the *checked()* signal to *setVisibility()* slot.

7.11.5 DELETING FILES

Map Layers often belong to an opened file in Updraft. To simplify deleting imported files (see chapter FileTypeManager) map layers define helper method *MapLayer::getDeleteAction()* returning a *QAction* that will remove the file (previously set with *MapLayer::setFilePath()*) from the file system and delete the map layer.

8 THE PLUGINS

8.1 INTRODUCTION

The plug-ins provide most of the Updraft functionality. Each of the plug-in is a dynamic library instantiated by the core. Each contains the *initialize()* and *deinitialize()* routines, which provide plug-in initialisation and de-initialisation. The reason these steps are needed is explained in chapter Top Level Structure in section Core. During the initialisation phase, each plug-in registers its file types and initialises its global variables and loads the cached files if desirable. During this process the signals for plug-in methods calls are connected. During the application run, the plug-in awaits the calls from the core utilising the fore mentioned signals. Before the main program ends the *deinitialize()* method is called to delete its variables and perform a cleanup. For more details please refer to the next chapter.

8.1.1 PLUG-IN CREATION

A new plug-in must contain the main class inheriting from *PluginBase* and *QObject.* Additionally, the class must be marked as *Q_DECL_EXPORT* and *Q_INTERFACES* (*Updraft::PluginBase*), and it's implementation must contain *Q_EXPORT_PLUGIN2* (*pluginName, PluginClass*) outside all functions.

As a convention, we manually add a global variable *g_core* to all plug-ins to simplify access to the core interface in other classes of the plug-in.

8.1.2 MINIMAL PLUG-IN EXAMPLE

This example shows the mandatory steps to be taken for successful plug-in creation.

In file `/Updraft/src/plugins/foo/fooplugin.h`

```
#include "../../pluginbase.h"

class Q_DECL_EXPORT FooPlugin
  : public Updraft::PluginBase, public QObject {
  Q_OBJECT
  Q_INTERFACES(Updraft::PluginBase)

 public:
  QString getName();
  void initialize(Updraft::CoreInterface *coreInterface);
  void deinitialize();
};
```

In file `/Updraft/src/plugins/foo/fooplugin.cpp`

```
#include "fooplugin.h"

Updraft::CoreInterface *g_core = NULL;
```

```
    QString FooPlugin::getName() {
      return "foo";
    }

    QString FooPlugin::initialize(Updraft::CoreInterface
*coreInterface) {
      g_core = coreInterface;
    }

    QString FooPlugin::deinitialize() {}

    Q_EXPORT_PLUGIN2(foo, FooPlugin)
```

In file `/Updraft/src/plugins/foo/CMakeLists.txt`

```
    cmake_minimum_required(VERSION 2.8)
    PLUGIN_BUILD(foo)
```

8.2 AIRSPACES

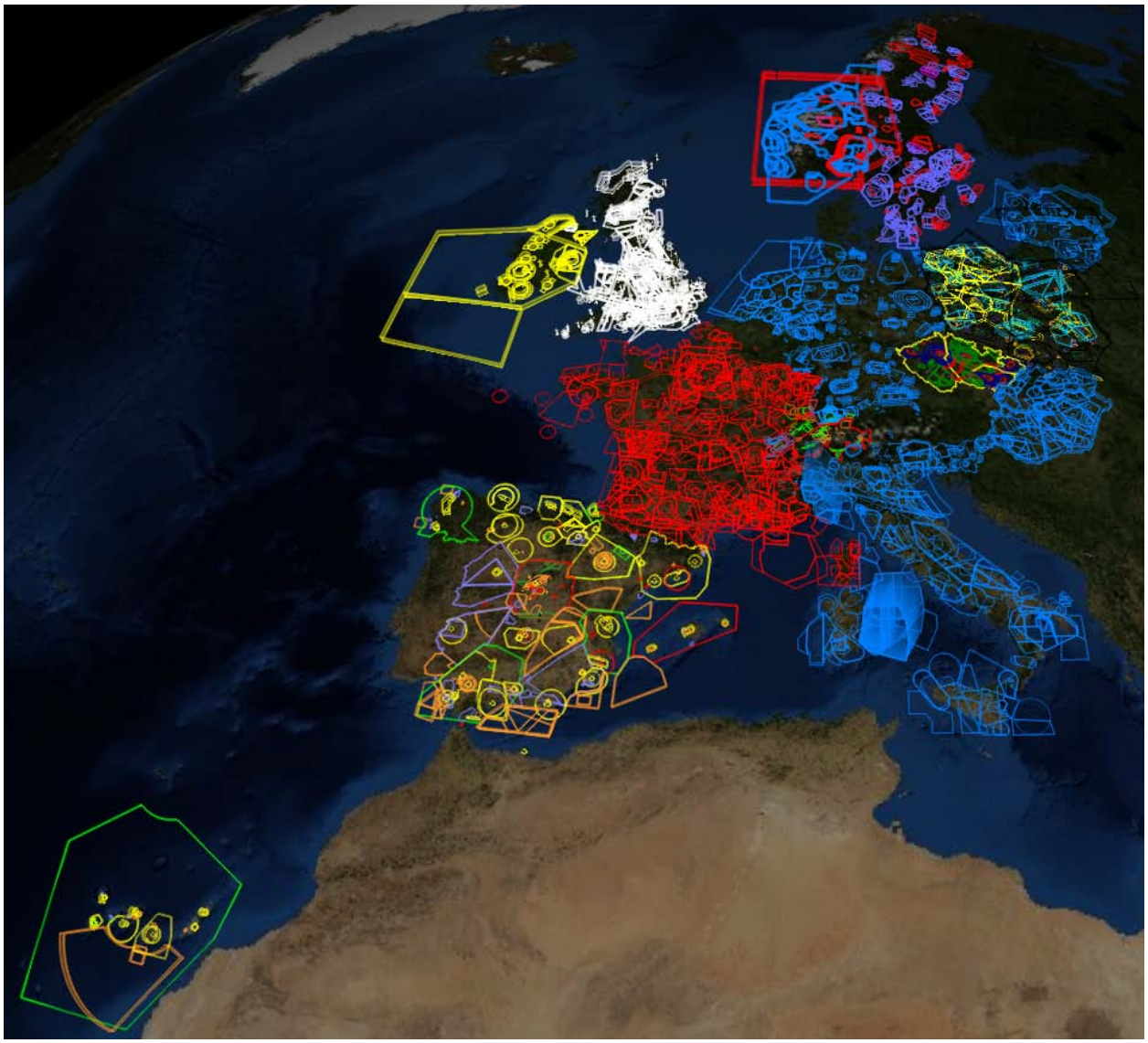

Figure 9. Western European Airspaces.

8.2.1 INTRODUCTION

One the most important information for a pilot planning his flight is to know where he is actually allowed to fly. The air space is divided into several sectors defined by its shape and a top and bottom altitudes. Each such a defined part of the air space belongs to a particular class, which will tell the pilot more about its availability. There are for example military airspaces, no fly zones or restricted areas, the pilot must be aware of, otherwise risking high fine or even worse kind of countermeasures. This type of awareness is vital for every pilot flying. As these information are so crucial, the pilot usually uses the approved maps, which are costly, and cannot rely on open-source

data. On the other hand, if the pilot wishes to use the visualisation tool to check the proximity of any airspace to his flight path after the flight is done, the open-source data are usually sufficient approximation of the reality. These may not replace the approved maps but can rather serve as an eye-candy in post flight visualisation. The files containing such a data are commonly accessible via the Internet. Therefore we like to provide the user with a tool for visualisation of these data. As the most common format used in Europe is the OpenAir(tm)[5] text format, we decided to incorporate the OpenAir(tm) airspace definition format parser into the application as well as the visualisation algorithm to draw these information on the map surface. For more detail please see the Appendix B – OpenAir(tm) airspace format definition.

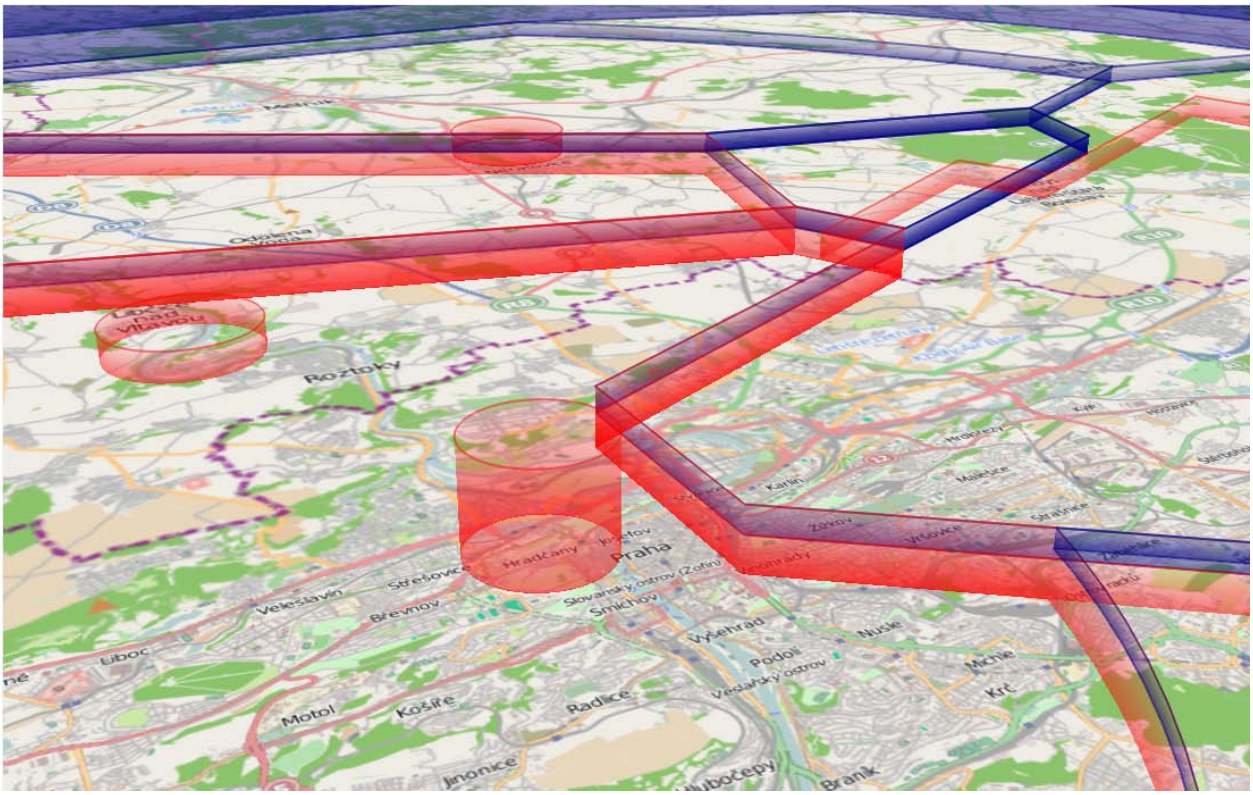

Figure 10. Example of airspace divisions visualisation.

8.2.2 OVERVIEW

The main role of this plug-in is to parse an airspace definition data from the provided file and visualize them. To visualise data, plug-in creates the geometry which is included as a layer to the map. The basic and at the time of the first program release the only format available in our application is the OpenAir(tm) format (see Appendix B – OpenAir(tm) airspace format definition). We use this format to demonstrate the plug-in design and philosophy. More airspace format can be included in future development.

[5] http://www.winpilot.com/UsersGuide/UserAirspace.asp

8.2.3 *DESIGN*

The program is divided in two main parts of the airspace processing. The first part consists of the routines which load and parse the text input file containing the provided airspace information. This information is then handed over to the second part of the algorithm, which will draw the defined airspaces into the map layer. We call the first part the *Parser* and the second part the *Airspaces* plug-in. The plug-in is loaded by the core as a part of the application, whereas the Parser is a library called from within the plug-in otherwise forming independent program. A part of the plug-in is the format-specific drawing engine which is called by the plug-in to draw to a particular map layer. The engine is provided with the name of the file the user wishes to process by the *Core* and calls the parser to load the airspaces which the file contains.

The *Parser* as well as the *Airspaces* library is linked to the main project as a dynamic library.

8.2.4 *OPENAIR(TM) PARSER*

This airspace declaration file is in text format and can contain information about several airspaces usually grouped into one logical or geographical group. Within the file the class, name, altitudes and other information is provided for each of the defined airspace. The airspace itself is defined either as a set of points and arcs forming a polygon or as a set of primitives (i.e. circles). During the start phase of the program, the plug-in parses the information from all the files which are saved in */data/airspaces* folder. Each file typically containing several airspaces within one region (e.g. country, city, etc.).

During the parsing phase of the process the file is opened for reading and the parsing process enters the main parsing loop, which ends when all the file content is processed or if the unexpected error occurs, e.g. the file is of incorrect format. In this loop one by one airspace class member is defined and filled with the data in file. This process is done within the particular Airspace class constructor. As stated above, the airspace is defined by the geometric primitives, such as arcs and points forming a closed polygon or a circle. The special data structures for these primitives were defined in the *geometry.cpp* and *geometry.h* files. These are the *Polygon, ArcI, ArcII* and *Circle* classes.

The parsing process itself is quite straight forward. The only tricky factoid, which the programmer must be aware of, is that some of the variables the airspace definition file contains are state variables. That means that the validity of such a variable reaches beyond the currently parsed airspace hence is valid until the end of the file or until re-defined. One example of such a variable are the pen and brush definitions, which are often defined in the beginning of the airspace definition file only and are valid for all the airspaces contained in the file. Another example could be the definition of the centre of co-centric airspaces. These are typically the airspaces surrounding one airport, therefore its centre is defined once per airport.

The Parser library is contained in following files located in `/Updraft/src/libraries/openairspace/` folder.

a. *openairspace.cpp* – the OpenAir(tm) format parser main procedures
b. *openairspace.*h – the OpenAir(tm) parser class declaration and interface
c. *airspace.cpp* – the single airspace class
d. *airspace.*h – the single airspace class declaration and interface
e. *openairspace_global.h* – the parser library main interface
f. *geometry.cpp* – the airspace geometry primitives class definitions
g. *geometry.h* – the airspace geometry primitives class declarations

Output of the parser is the member of *OpenAirspace::Parser* class containing information about all the airspaces described in particular file. The only private variable of this class is the *allAirspaces*, an array of *OpenAirspace::Airspace* members. Each such a member contains parsed information about single airspace. This concludes the pen and brush colour, name, class, ceiling, floor, geometric primitives forming the airspace boundaries and more.

The OpenAir(tm) *Parser* library is linked to the main project as a dynamic library.

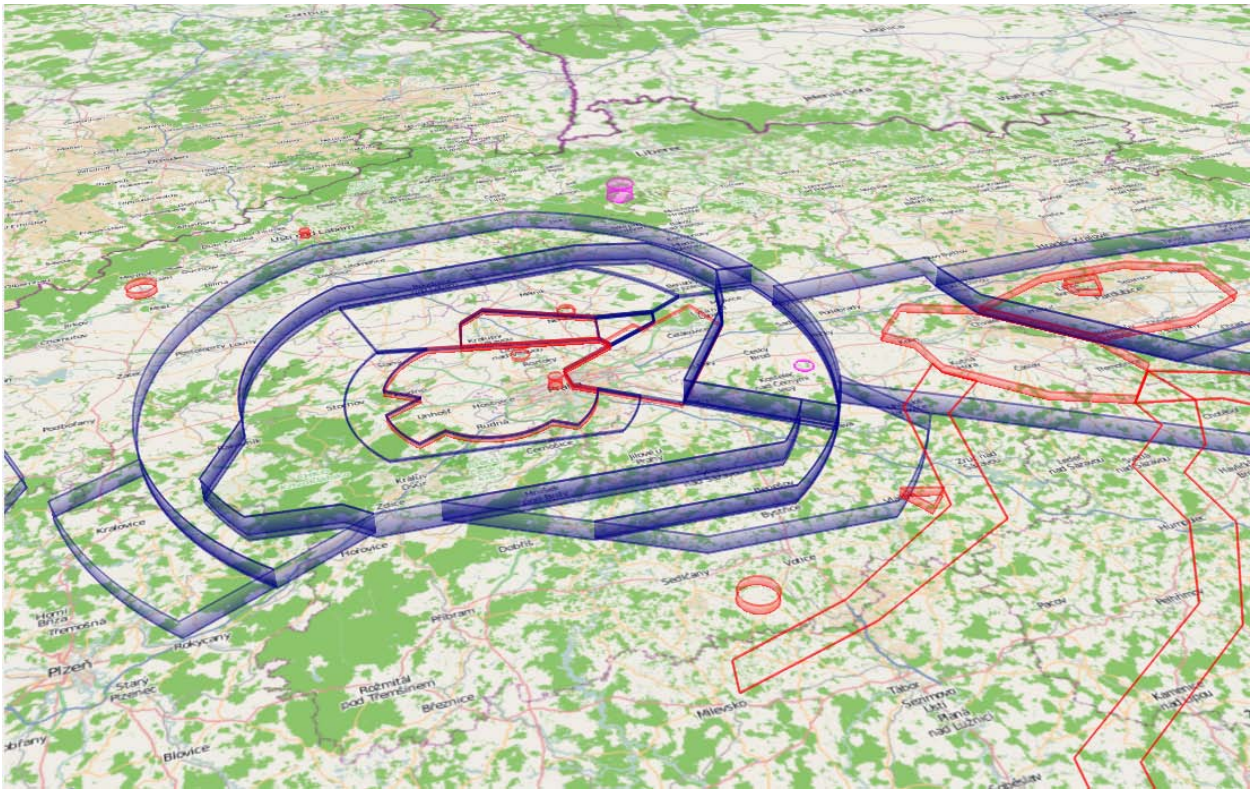

Figure 11. Example of the OpenAir(tm) data visualisations.

8.2.5 *OpenAir(tm) drawing engine*

The main class connected to the *Core* is the *Airspaces* class. This class is responsible for the communication with the core and for calling the airspace drawing engine and parser. In the initial

phase of the plug-in load the *Airspaces* all the files in */data/airspaces* folder are processed. During the program run also the user-imported files are drawn and the file is stored in */data/airspaces* directory. Each of the files, which needs to be processed is given to the drawing engine, specifically the *oaEngine* class constructor. This method is then responsible for calling the parser.

As mentioned above the output of the parser is the member of *OpenAirspace::Parser* class containing information about all the airspaces described in particular file. This member contains in fact an array of parsed airspaces for the given region and is the main outcome of the parsing process. Each of the airspaces in array is then processed in such a way, that first the colour specification is read (this is often common for whole the file or at least for group of airspaces) or the default colour is assigned. Next the geometry is created for each of the airspace's set of points and arcs forming a closed polygon or a circle. The final geometry which is to be drawn consists of bottom polygon and the top polygon. Both the polygons being topologically identical each of them is located at height assigned as a floor and ceiling of the airspace respectively. Data defining the polygons were parsed during the process described above. The array of point pairs (bottom polygon point and corresponding top polygon point) defines the triangle strip OpenGL primitive used to visualize the side of the airspace. As mentioned earlier the OpenAir(tm) format defines the airspace as a circle primitive or as a closed polygon consisting of a set of points and arcs. OpenGL is based on processing the triangle meshes defined as a set of vertices for performance reasons. Thus all the arcs, as well as circles declared must be converted to a set of points allowing the use of OpenGL mesh setup. Granularity of this conversion is defined as a constant within *oaEngine* class. Because all the data defining the airspace are in WGS84 [18] format, many arbitrary algorithms which work with this coordinate system had to be introduced. The main aim of these procedures is to compute the set of points from given airspace geometry. These points can be then used for filling the vertex array of the OpenGL utilising the methods of OSG Earth. These are used to correctly transform the WGS84 coordinates of vertices to the OpenGL world view and for correct evaluation of the terrain height from the height field data provided by OSG Earth.

Such a filled OpenGL vertex array is added to a scene graph node as a set geometry. Each node is now defining single particular airspace. Finally the set of OpenGL switches is used to ensure correct evaluation of transparency effects and visibility of the airspaces. The nodes are accumulated in map nodes array, which is handed back to the *Airspaces* class.

Each of the nodes in the array is added by *Airspaces* class to the scene tree of OSG as a map layer. In the same time the GUI display tree is created and signals for displaying the defined layers connected.

The Parser library is contained in following files located in `/Updraft/src/plugins/airspaces/` folder.

a. *airspaces.cpp* - the main interface of the plug-in
b. *airspaces.h* - the airspaces plug-in interface
c. *oaengine.cpp* - the main airspace drawing engine
d. *oaengine.h* - the OpenAir(tm) drawing engine class and methods declarations

The *Airspaces* plug-in library is linked to the main project as a dynamic library.

8.2.6 3RD PARTY SOFTWARE USED

- Nokia Qt [4] for the GUI framework
- OSG [9] – as a OpenGL framework, for the mesh visualisation
- OSG Earth [10] – for the world-view matrix transformations

8.2.7 THE AIRSPACES PLUG-IN DIAGRAM

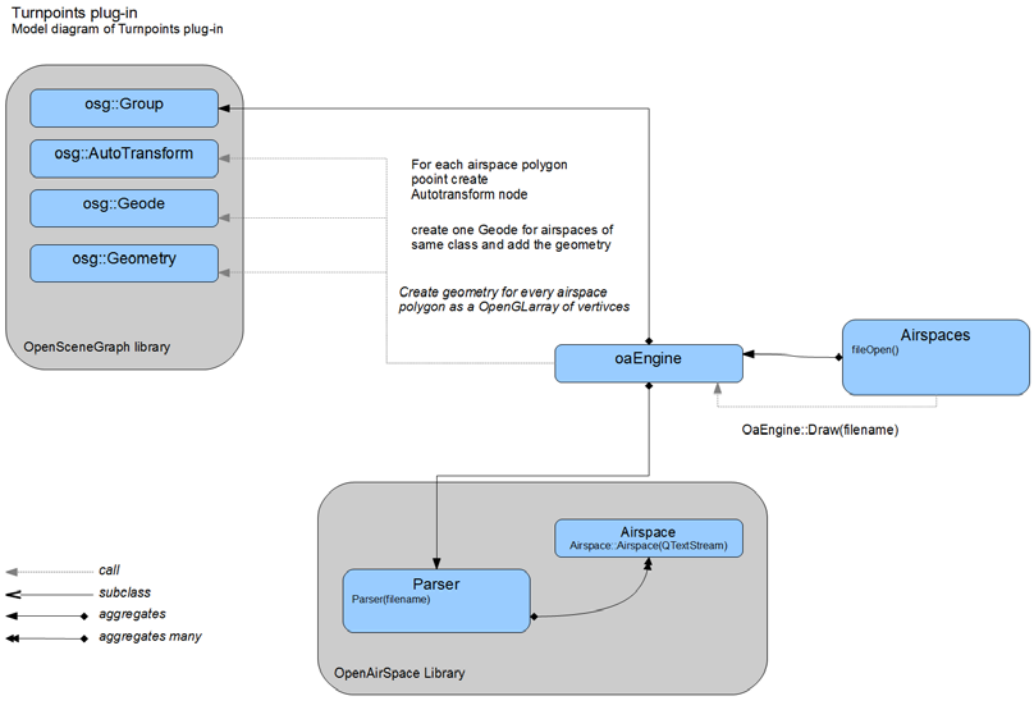

Figure 12 : Airspaces plug-in and OpenAir parser diagram.

8.3 TURNPOINTS

8.3.1 INTRODUCTION

During the glider competition flight the typical task of a pilot is to navigate through the given set of waypoints usually forming a triangle. Each point is defined by its geographical coordinates, typically in WGS84 coordinate system. We will call such a point the *turn-point.* It is common practice (especially during competition) that there is a predefined set of such points. The set is distributed electronically as a file containing all the turn-points (e.g. by competition organizers) concerning the particular area. These turn-points therefore play an important role in flight planning procedure. Hence it is very useful for our software to have the turn-points visualisation capability. It is also very convenient way to visualise the airports whenever this information is provided by the turn-points file.

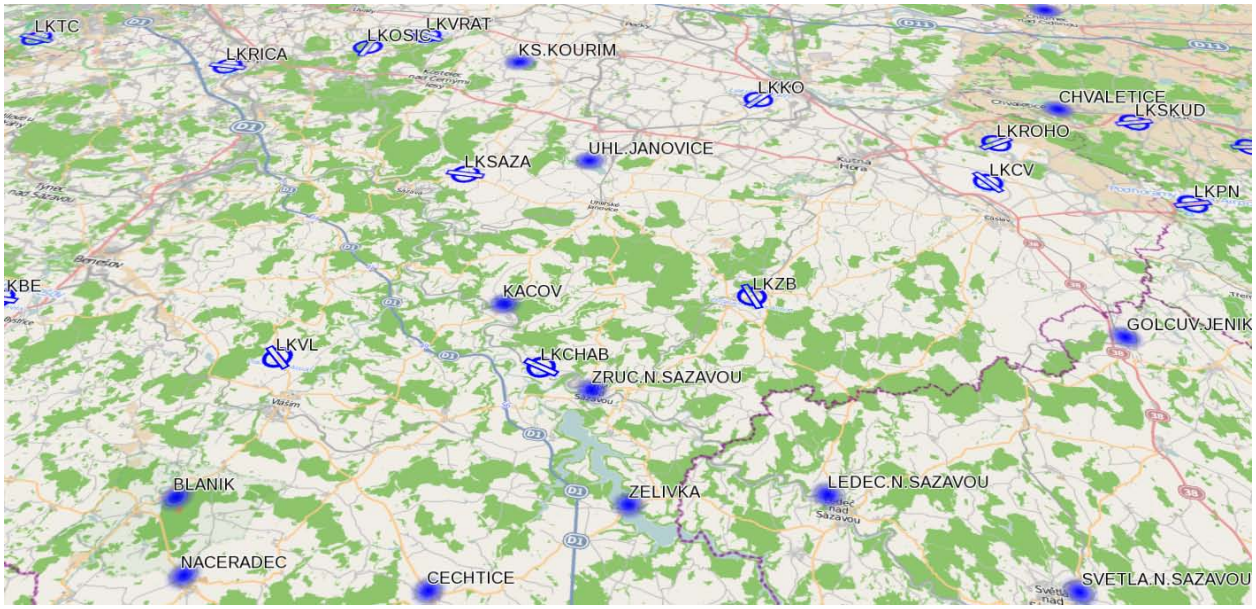

Figure 13. Turnpoints visualisation example.

8.3.2 OVERVIEW

This plug-in is intended to collect and visualize turn-points. It uses Updraft Cup library, which implements parsing and loading turn-points from SeeYou *cup* files. For the cup file format description please see the Appendix D – Cup File Format Description. Loading turn-points from any file format is potentially possible – the plug-in is easily extendable and it doesn't depend on cup file format.

It is a good practise that the definition of the turn-point also contains the information about the turn-point type, one of which is the Airport type. Together with this the detailed information

about the airport, such as main runway heading or airport type, is often provided. In our application we utilise this information for distinguishing the airport from the common turn-point using different icons. In case of airport visualisation, we also use the information about the runway heading to correctly situate the icon.

8.3.3 *Design*

The Turn-points plug-in is loaded by the core the same way any other plug-in is. The plug-in register the file type associated with the plug-in in the initial phase. It also loads all the cached files in the designated turn-points directory. Please see chapter FileTypeManager for more detailed description. For each of the files the cup file parser library is called to extract the information about the turn-points from the files. Subsequently the algorithm for visualising the parsed data is called. It processes each of the turn-points and using the information about the type, geographical position and height it creates the icons, which forms the independent map layer. This map layer is then added to the OSG scene.

8.3.4 *TPLayer Class*

This class is responsible for creating the geometry from the parsed data utilising the OSG methods for creating the OSG nodes and further using the OSG transformation matrices for positioning the data into the scene. It also encapsulates the geometry to the map layer, which is finally added to the OSG scene to be visible in the map window.

8.3.5 *TPFileCupAdapter Class*

This class is a part of the Turnpoints plug-in and serves as an intermediate layer between the CupFile class and the Turnpoints plug-in. It prepares the parsed data in such a way, that these can be used by the TPLayer class for visualisation. This class is responsible for loading the cup files by calling the cup parser.

8.3.6 *Cup File Parser*

Each of the file containing the turn-points needs to be read, data it contains extracted and encapsulated in class data structure to be accessible by the turn-points plug-in. In case of the SeeYou Cup format (see the detailed description in the Appendix D – Cup File Format Description) these data are held in *Updraft::CupFile* class. Data in this format are prepared for use by the Turnpoints plug-in.

8.3.7 *Turnponts Plug-in Diagram*

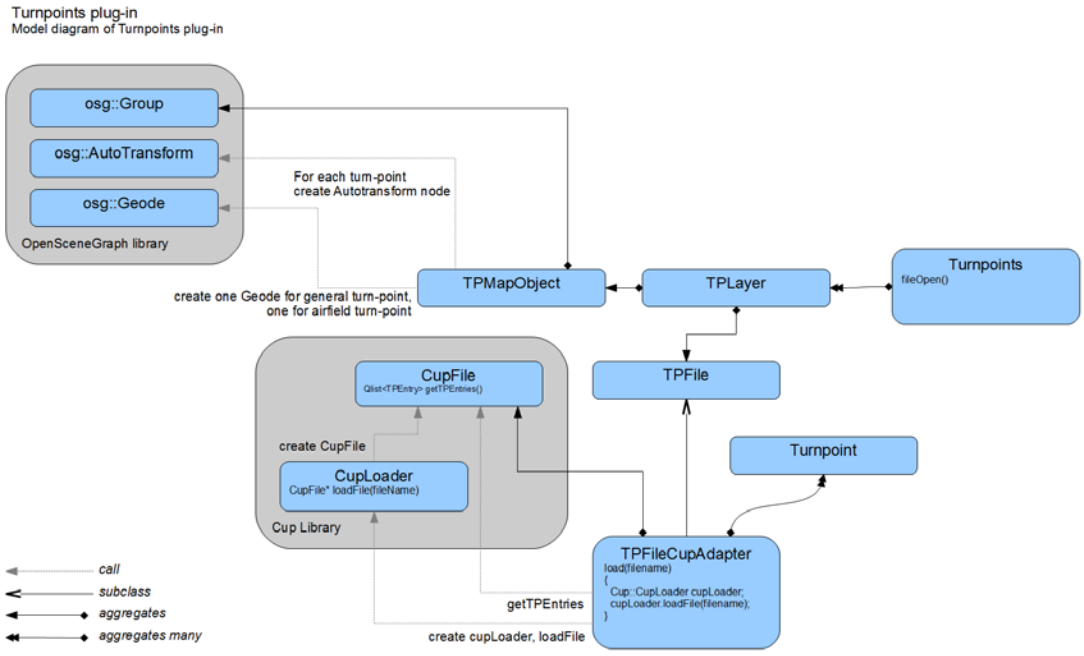

Figure 14. Turn-points plug-in design.

8.4 TASK DECLARATION

8.4.1 INTRODUCTION

Task declaration is a plug-in designed for flight planning. Achieving of pre-declared goal is a basis of cross country flying. The task consists of a few waypoints that pilot must reach. Usually there is a set of turn-points from which the pilot selects. *TaskDeclaration* plug-in is able to define track by both selecting from pre-declared turn-points and also setting arbitrary points. When pilot declares his task, the information he is most interested in is track length and track shape. *TaskDeclaration* plug-in considers both of these capabilities.

Plug-in source files are stored in directory `Updraft/src/plugins/taskdecl`.

8.4.2 PLUG-IN MODEL INTRODUCTION

The core of the plug-in is class *TaskDeclaration* which is sub-class of *PluginBase*. It communicates with core and is responsible for Tasks manipulations. *TaskDeclaration* contains several *TaskLayers*. Each *TaskLayer* corresponds to one opened task file, which may be loaded from disk or just created by application. *TaskLayer* contains *TaskFile* with data and *TaskDeclPanel* with GUI.

8.4.3 DATA ACCESS MODEL

TaskFile

TaskFile provides access to file data and also allows file content changing. All changes are stored and can be undone (using design pattern *Memento*). This behaviour is transparent outside the *TaskFile*, history management is done by its private members.

Data of the task is stored in *TaskData* class. *TaskFile* returns *TaskData* in method *beginRead*. After calling *beginRead* it is impossible to do any changes to *TaskFile* before calling *endRead*. Similarly when task is to be modified, it is performed by calling *TaskFile* method *beginEdit*, which has associated similar end method *endEdit*. Pairs of *beginRead*, *endRead* and *beginEdit*, *endEdit* serves as primitive locking system, although it is not intended to be used by multiple threads (it is not guaranteed to be thread-safe).

Method *beginEdit* has boolean parameter *createNewState*. If is set to *true*, current data are stored and copied so the new changes will be made on the copy. All previous changes are guaranteed to be stored. If the parameter *createNewState* is *false*, then current data are modified (changes cannot be undone).

These are two types of changes that could be performed on task data (undoable and not undoable).

Undo/redo TaskFile changes

TaskFile uses class *DataHistory* for storing *TaskData* and managing its changes. The *DataHistory* stores *TaskData* in a *QLinkedList* container. *DataHistory* has private *iterator* which points to current data. This *iterator* can be moved back and forth by calling *moveForward* and *moveBackward* methods. When the current data is to be stored (undoable changes are made), all items newer than current are deleted from container and copy of the current is appended to the end of the list.

TaskData

This class stores *QVector* container of *TaskPoints*. It is data representation of task itself. Instance of *TaskData* may correspond to state shown in application or to state stored on disk. *Taskdata* can be serialized to the XML formatted text. Reading and writing of XML is processed directly in *TaskData* using Qt DOM classes.

8.4.4 GUI MODEL

TaskLayer GUI is encapsulated in *TaskDeclPanel* class. Panel layout is described in `taskdeclpaneul.ui` file.

GUI consists of caption with task information summary, buttons for saving, undo and redo, *TaskPointButtons* representing single task-points, *addButtons* with green "+" icon for inserting new task-points and *TaskAxis* showing task legs information.

TaskPointButton contains *QLabel* with ordinal number of task-point, another *QLabel* with task-point title and *QPushButton* *quitButton* for task-point deleting.

*QButtonGroup** *addButtons* is a group of *QPushButtons* used for task-point inserting. At most one button from this group is checked at one time. This behaviour is ensured by grouping into *QButtonGroup*.

TaskAxis is a sub-class of *Qwidget*. It handles (overloads) *paintEvent* for printing lengths and

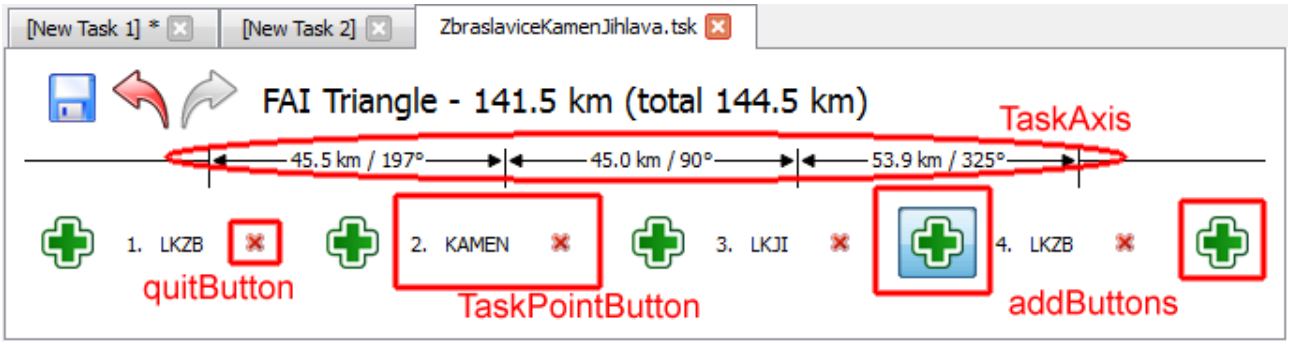

Figure 15. TaskDeclPanel GUI layout.

azimuths of task legs and drawing arrows, pointing to surrounding task-points.

GUI interactions

addButtons and *quitButton* invokes task changes. To perform it, *TaskDeclPanel* takes associated *TaskFile* and with *beginEdit*/*endEdit* modifies current *TaskData*. Undo/redo buttons invoke moving of *TaskData iterator* stored in *DataHistory* of *TaskFile*.

When *TaskFile* is changed, GUI and also rendered geometry needs to be updated. *TaskFile* emits signal *dataChanged* when it has been modified. *TaskDeclPanel* catches this signal and its content can be updated.

When number of task-points changes, set of *TaskPointButtons* and *addButtons* is regenerated.

8.4.5 PLUG-IN SCHEME

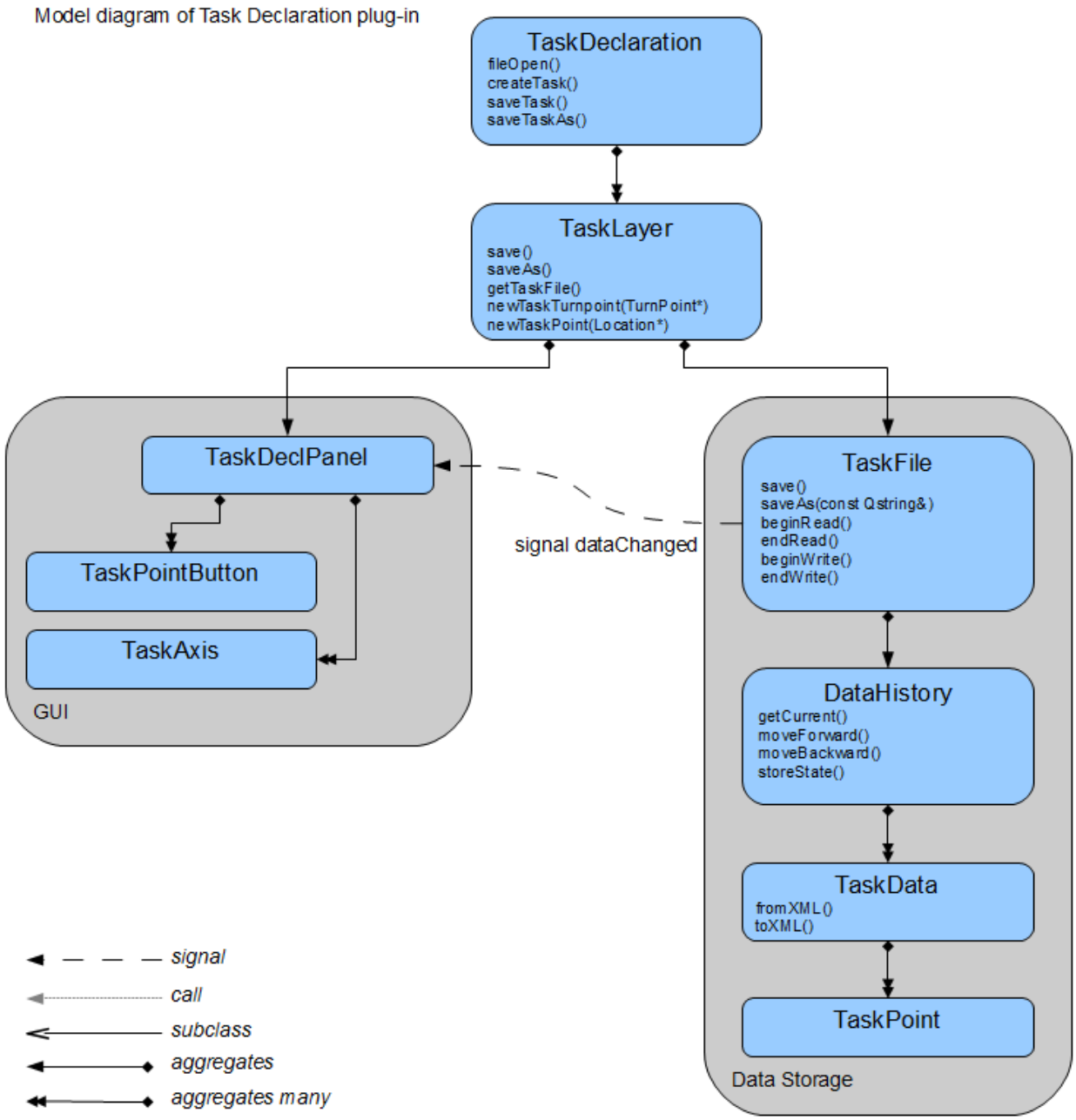

Figure 16. TaskDeclaration plug-in design scheme.

8.5 IGC VISUALISATION

8.5.1 INTRODUCTION

For the purposes of flight debriefing or contest scoring, the flight information is recorded using specialised flight loggers. Loggers export data in a standardized IGC file format[6].

Figure 17. Colibri flight logger. Picture taken from internet[7].

Role of this plug-in is to visualize recorded flight tracks opened from an IGC file, and provide interactive statistics of the flight. These statistics include the colour-coded information about elevation, airspeed, vertical speed, and others.

8.5.2 MODEL

The plug-in functionality is divided into two parts:

- the IGC file parser that gets the raw data from the IGC log
- the visualisation plug-in that visualises the IGC track, and computes the statistics from the data

[6] http://web.archive.org/web/20100723171307/http://www.fai.org/gliding/system/files/tech_spec_gnss.pdf
[7] http://www.lxnavigation.si/avionics/images/avionics/Slike/Colibri_mala.png

8.5.3 IGC PARSER

Igc parser plug-in project is located in the directory `/Updraft/src/libraries/igc`, with several automated tests in the directory `/tests`. The parser itself is a simple top-down parser, reading lines, each one as a record.

Input file is represented by *QIODevice*, but a convenient method that requires only file name as a string is available as well.

All text fields of the IGC file are decoded using selectable codec (Latin 1 by default).

Data items that are allowed only once per IGC file, such as pilot name, glider identification, flight date, etc..., are stored as simple class variables with appropriate getter methods. If a field is absent from the file, then the getter method returns default constructed value.

Values that occur multiple times during the flight (Currently only GPS fix and pilot event) are stored as subclasses of *Event* in a chronological list together with their timestamp. When used, events usually need to be up-casted based on their *type* field.

8.5.4 IGC VIEWER PLUG-IN

Class *IgcViewer* in `/Updraft/src/plugins/igcviewer` directory is an Updraft plug-in for IGC file visualisation and statistics computation that works with the data provided by Igc parser.

It registers the file type `` `*.igc` ``, holds the list of currently opened IGC files, and keeps automatic colourings for IGC files so that different IGC tracks have different colours, to be easily separable.

Each opened file is represented by an *OpenedFile* instance, which is stored in a container indexed by the file name. This container is checked every time a new file is opened to avoid duplicate opening. Opened file creates geometry, and handles the statistics window in the bottom tab bar of the updraft window.

The workflow of opening a file is the following:

1. *IgcViewer* gets an event that a file has been opened, and gets the filename of the file.
2. It creates the *OpenedFile* instance, computes its default colour, and lets it initialize, passing the filename.
3. *OpenedFile* calls the Igc parser on the file, and from the GPS events list creates a *Fix List*.
4. The track geometry is created.
5. *OpenedFile* creates the *FixInfo* instances from the *Fix List* - classes that hold various statistics info per each fix.
6. The tab with the statistics is created.

Fix List

The fix-list is a list of *TrackFix* instances - events from the IGC parser, filtered to contain only GPS fixes and wrapped into a new structure, together with the GPS location coordinates, and coordinates projected to geometry world coordinates (X, Y, Z). Once the fix-list is created, the original events from the parser are not used any more.

Visualisation

Track Geometry

Track of the IGC file visible in the map is created in the method *OpenedFile::createGroup*, which creates the geometry as an OpenSceneGraph *osg::Group* scene node.

This group consists of:

- the actual 3D track - a line strip
- track markers - small circles that mark the locations the statistics are computed on
- the transparent "skirt" to enhance 3D effect and to mark the ground projection of the flight track.

Most of the Geometry is fixed, however position of the marker changes with every click on the map or on the graph (see below) and colours of the track change with user setting.

Track Colouring

Track colouring is selectable from a combo box in the tab GUI.

Tracks can be coloured using:

- automatic colouring
- vertical speed
- ground speed
- altitude
- time

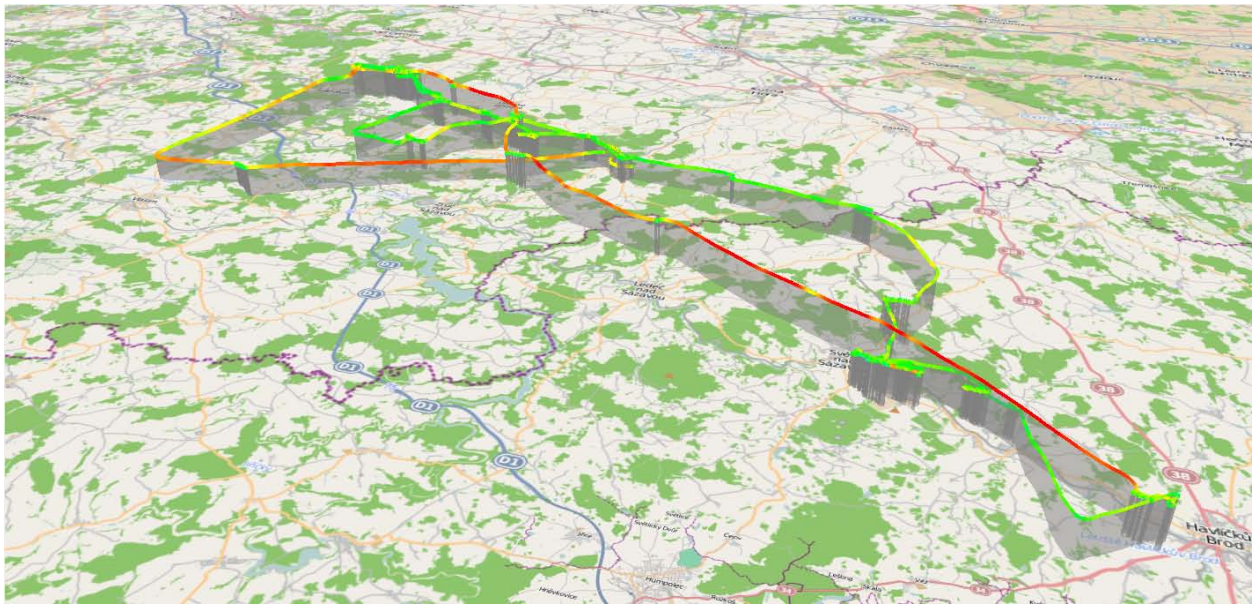

Figure 18. IGC track Ground speed colouring example.

Colourings are represented by the class *Coloring*. It provides colour for given index of the track fix. Subclasses of *Coloring* mostly (with the exception of automatic colour) receive pointer to *FixInfo* and a gradient.

Automatic colouring rotates colours of each opened file from a predefined list. Each colour in the list has a use count and every time a new IGC file is opened, one colour with the least uses is selected. Selection of automatic colour is done in the class *IgcViewer* and the selected colour is passed to the opened file class during initialization.

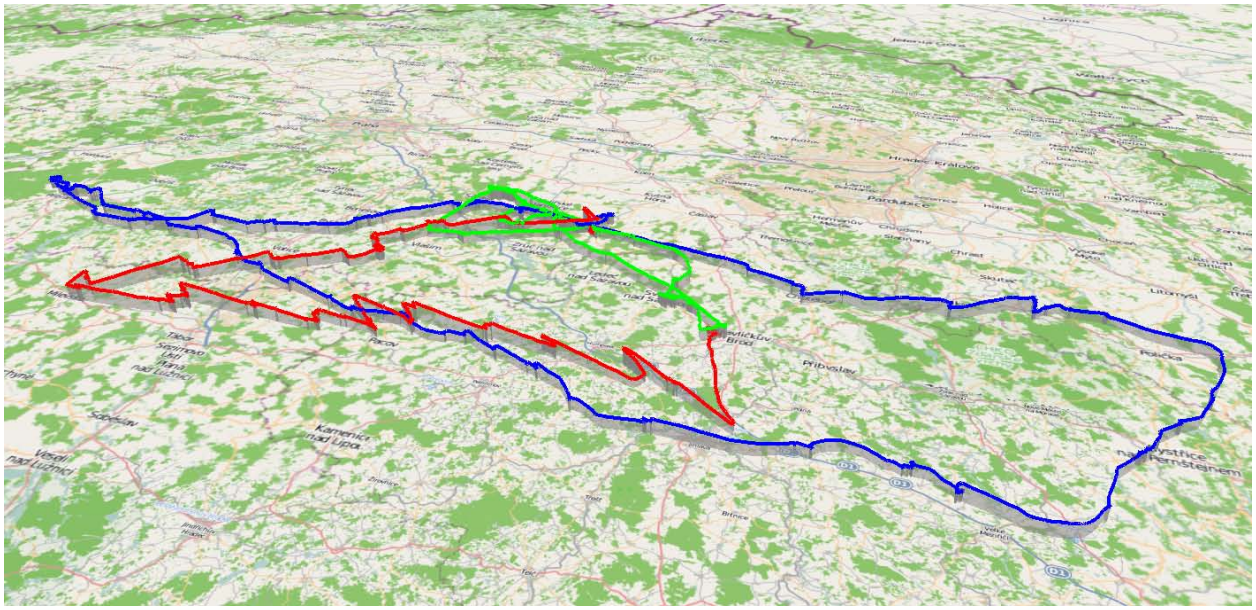

Figure 19. IGC tracks automatic colouring.

Statistics computation

Statistics are visualized in the bottom tab window, allowing user interaction. The basic class for the statistics is the *FixInfo*.

Class FixInfo

Class *FixInfo* is an abstract class that represents values that can be extracted from every fix of the IGC track. Subclasses of this class exist for altitude, vertical speed and ground speed.

FixInfo classes are defined in files `/Updraft/src/plugins/igcviewer/igcinfo.cpp` and `/Updraft/src/plugins/igcviewer/igcinfo.cpp`.

Other than retrieving the value at given point, *FixInfo*`s provide four types of scaling information, which is used in graph plotting. `*min*` & `*max*` are simple minimum and maximum value. Robust versions skip top and bottom 3 % of the values, and global versions cover all opened IGC files.

For example $globalRobustMin = \min_{x \in openedFiles} robustMin_x$.

The global colourings are updated every time an IGC file is opened or closed using method *OpenedFile::updateScales* to provide a visual overview of the track values that are scaled to match all the opened files.

Class SegmentInfo

Class *SegmentInfo* provides statistics for track segments. It holds the *FixList* instance, and provides statistics between 2 fixes.

Statistics visualisation

Statistics are plotted in a graph in the *PlotWidget* window, which also provides user interaction statistics.

PlotWidget class

PlotWidget contains plots of the *FixInfo* values for various statistics, and provides user interaction for picking out points, and segments to compute the statistics from. For better performance, the plots and statistics are not drawn into the widget every *paintEvent*. They are drawn into a *QImage*, and redrawn only at resize events, or when the statistics texts are changed.

Graph plots

- **Axes**
  Basic concept of graph plots are the axes. Axes determine position, size and scaling of the plotted data and draw tick marks. Axes are derived from *QLayoutItem*, this enables us to use Qt's native layouts in the plot widget.

- **Plot Painters**
  Plot painters are classes that are given the axes and the *FixInfo*, and plot the data into the graph. The scale of the plot and the layout in the widget are determined by the axes instance, by calling its methods to compute the drawing positions of given data-value. There are several subclasses of the abstract *PlotPainter* class, each plotting a different statistic in a different style..

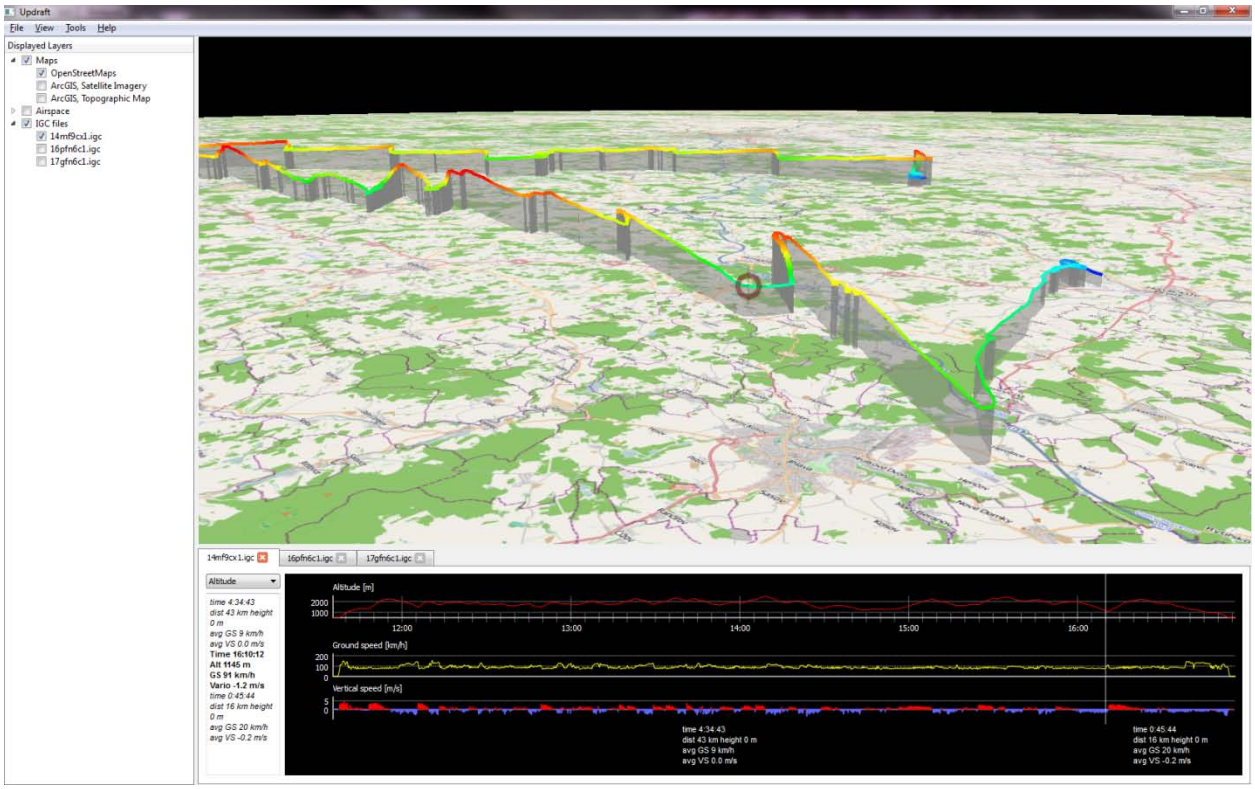

Figure 20. IGC visualisation with altitude colour code. The black pane shows the flight graphs. Mouse picked point for the fix statistics shown as a red circle on the map and a vertical line on the graph plot.

Statistics upon mouse interaction

- **Fix statistics**
  Upon mouse move, or mouse click, the given track fix is computed from the mouse location, and the information of this track-fix are gotten from the *FixInfo* sub-classes. The statistics are stored in text form, and after creating the text string, the string is then passed to *IgcTextWidget* to display. A signal is emitted to set the position of the track marker on the 3D track, on the location of the fix the user has clicked, or is currently pointing.

  A fix can also be picked out when clicking on the 3D track. Then, the *OpenedFile* emits a signal with the fix index the user has clicked on, and *PlotWidget* catches this signal and adds a new picked fix. The statistics for this fix and the segments are then computed.

- **Segment statistics**
  When user picks out a fix (either by clicking on the graph, or clicking on the 3D track), the segment statistics between the points are computed. The segment statistics are saved in a text form, and passed to *IgcTextWidget*, and to *PickedLabel* classes for the display.

8.5.5 *Plug-in Design Scheme*

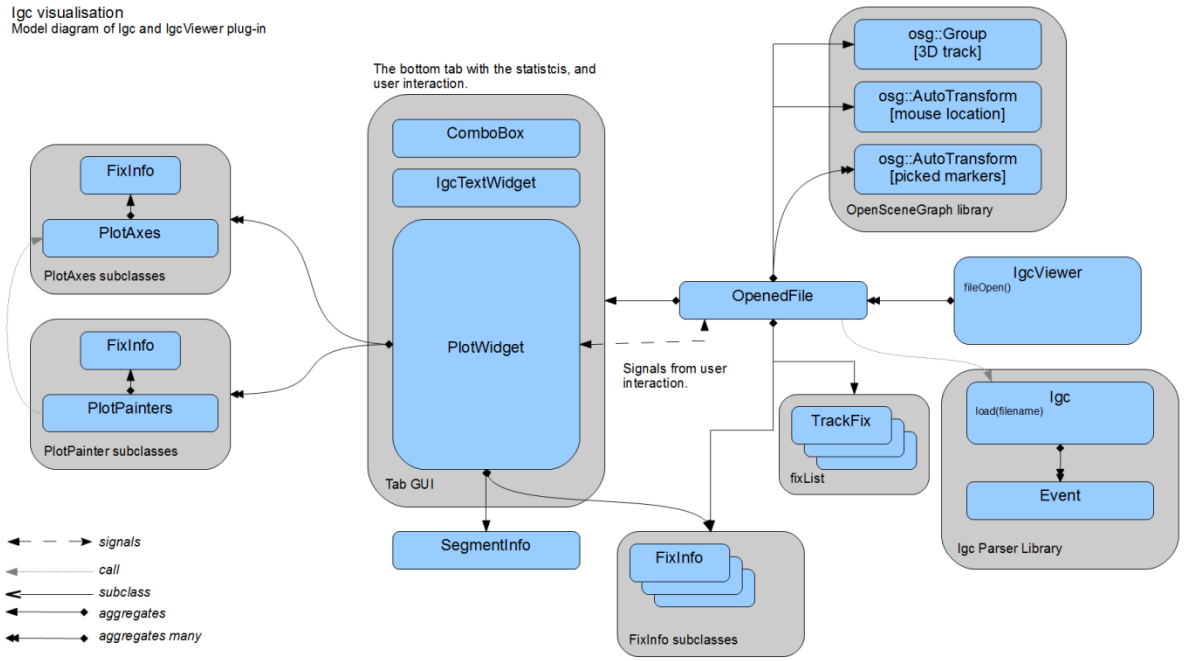

Figure 21. IGC Visualisation plug-in scheme.

9 PROGRAM INSTALLATION

9.1 WINDOWS

9.2 LINUX

9.3 MAC

10 CONCLUSION

We have created this application to serve the glider pilots as a glider path planning and flight visualisation tool. It is a free easy-to use alternative, and we believe even an improvement, to the commercial product *SeeYou*. The main targets we set ourselves as a goal has been all reached. The application has proven to be smoothly running on all of the designated platforms, i.e. Windows, Linux and Mac OS. The implemented features include the *Airspaces* plug-in for airspace division visualisation, the *Turnpoints* plug-in for the waypoints visualisation, the *TaskDeclataion* plug-in for the flight planning and the *IGC visualisation* plug-in for the recorded flight path visualisation. All these features are incorporated to the nice looking 3D map environment including several map types and even the terrain elevation visualisation. We were very unfortunate in having the Colibri logger borrowed for too short period and therefore being unable to implement the direct data retrieval from the device. Also we did not implement the weather forecasting plug-in, as well as some eye-candy features we planned to do. These features were not in the project proposal and were not implemented mainly for the lack of time resources. Because these features are "nice-to-have", we plan to implement them in the future.

11 FUTURE WORK

The application is designed to be expandable. We anticipate the development of the product to continue in the future. The Qt core/plug-in system we have chosen is ideal for addition of new plug-ins such as the weather plug-in. We also plan to implement the interface for flight data download directly from the logger device or some other minor eye-candy features such as compass visualisation. Additionally all the plug-ins working with files are easily extendable to accommodate parsers and data processors for future file formats, e.g. *Airspaces*, *Turnpoints* or *IGC visualisation* plug-ins.

12 REFERENCES

[1] Naviter, "SeeYou," Naviter, [Online]. Available: http://naviter.si/index.php?option=com_content&task=view&id=9&Itemid=213.

[2] "FreeLists," [Online]. Available: http://www.freelists.org.

[3] Google, "Google C++ Style Guide," [Online]. Available: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.

[4] Nokia, "Qt," Nokia, [Online]. Available: http://qt.nokia.com/.

[5] Microsoft, "Visual Studio 2008," Microsoft, [Online]. Available: http://www.microsoft.com/visualstudio/en-us/products/2008-editions.

[6] Open-source, "CMake," [Online]. Available: http://www.cmake.org/.

[7] L. Torvalds. [Online]. Available: http://git-scm.com/.

[8] "GitHub," [Online]. Available: http://www.github.com.

[9] Open-source, "The OpenSceneGraph," [Online]. Available: http://www.openscenegraph.org.

[10] G. W. Jason Beverage, "OSG Earth," Pelican Mapping, [Online]. Available: http://osgearth.org.

[11] C. Karney, "GeographicLib," [Online]. Available: http://geographiclib.sourceforge.net/.

[12] Wikipedia, "Qt framework," [Online]. Available: http://en.wikipedia.org/wiki/Qt_(framework).

[13] K. Gtoup, "OpenGL," [Online]. Available: http://www.opengl.org/.

[14] Open-source, "OpenStreetMap," [Online]. Available: http://tile.openstreetmap.org/.

[15] ESRI, "Imagery World 2D," [Online]. Available: http://server.arcgisonline.com/ArcGIS/rest/services/ESRI_Imagery_World_2D/MapServer.

[16] ESRI, "World Topo Map," ESRI, [Online]. Available: http://server.arcgisonline.com/ArcGIS/rest/services/World_Topo_Map/MapServer.

[17] G. W. Jason Beverage, "Ready map elevation tiles," Pelican Mapping, [Online]. Available: http://readymap.org/readymap/tiles/1.0.0/9/.

[18] Wikipedia, "World Geodetic System," [Online]. Available: http://en.wikipedia.org/wiki/World_Geodetic_System.

13 TABLES

14 FIGURES

15 APPENDIX A – PEOPLE

Alexander Wilkie – Computer Graphics Senior Lecturer on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Tomáš Zámečník – Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Čestmír Houška – Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Jakub Marek – Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Bohdan Maslowski – Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Mária Vámošová - Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

Aleš Zita – Student of Computer Science master degree on Faculty of Mathematics and Physics by Charles University, Prague, Czech Republic;

16 APPENDIX B – OPENAIR(TM) AIRSPACE FORMAT DEFINITION

Following is the definition of the OpenAir(tm) format as provided on the winpilot[8] web page.

OPEN AIR (tm) TERRAIN and AIRSPACE DESCRIPTION LANGUAGE
Version 1.0
December 10, 1998
Updated October 15, 1999
Send comments to jerryp@winpilot.com

AIRSPACE related record types:
==========

AC class ; class = Airspace Class, see below:
R restricted
Q danger
P prohibited
A Class A
B Class B
C Class C
D Class D
GP glider prohibited
CTR CTR
W Wave Window

AN string ; string = Airspace Name
AH string ; string = Airspace Ceiling
AL string ; string = Airspace Floor
AT coordinate ; coordinate = Coordinate of where to place a name label on the map (optional)
; NOTE: there can be multiple AT records for a single airspace segment

TERRAIN related record types (WinPilot version 1.130 and newer):

[8] http://www.winpilot.com/UsersGuide/UserAirspace.asp

```
 ==============================

 TO {string} ; Declares Terrain Open Polygon; string = name (optional)
 TC {string} ; Declares Terrain Closed Polygon; string = name
(optional)
 SP style, width, red, green, blue ; Selects Pen to be used in
drawing
 SB red, green, blue ; Selects Brush to be used in drawing


 Record types common to both TERRAIN and AIRSPACE
 ================================================

 V x=n ; Variable assignment.
 ; Currently the following variables are supported:
 ; D={+|-} sets direction for: DA and DB records
 ; '-' means counterclockwise direction; '+' is the default
 ; automatically reset to '+' at the begining of new airspace segment
 ; X=coordinate : sets the center for the following records: DA, DB,
and DC
 ; W=number : sets the width of an airway in nm (NYI)
 ; Z=number : sets zoom level at which the element becomes visible (WP
version 1.130 and newer)

 DP coordinate ; add polygon pointC
 DA radius, angleStart, angleEnd ; add an arc, angles in
degrees, radius in nm (set center using V X=...)
 DB coordinate1, coordinate2 ; add an arc, from coordinate1 to
coordinate2 (set center using V X=...)
 DC radius ; draw a circle (center taken from the previous V X=...
record, radius in nm
 DY coordinate ; add a segment of an airway (NYI)
```

17 APPENDIX C – CUP FILE FORMAT DESCRIPTION

Each line represents one waypoint with these fields, separated by commas. Here is an example:
"Lesce-Bled","LESCE",SI,4621.666N,01410.332E,505.0m,2,130,1140.0m,"123.50","Home airfield"

1. Name
This is the long name for the waypoint. It is supposed to be ebraced in double quotes to allow any
characters, including a comma in between. This field must not be empty.

2. Code
Also known as short name for a waypoint. Many GPS devicees cannot store long waypoint names, so
this field will store a short name to be used in various GPS types. It is advisable to put it in double
quotes.

3. Country
IANA Top level domain standard is used for the country codes. A complete list is available at
http://www.iana.org/cctld/cctld-whois.htm

4. Latitude
It is a decimal number where
1-2 characters are degrees,
3-4 characters are minutes,
5 decimal point,
6-8 characters are decimal minutes.
The ellipsoid used is WGS-1984

5. Longitude
It is a decimal number where
1-3 characters are degrees,
4-5 characters are minutes,
6 decimal point,
7-9 characters are decimal minutes.
The ellipsoid used is WGS-1984

6. Elevation
It is a string with a number with unit attached. Unit can be either "m" for meters or
"ft" for feet.
Decimal separator must be a point.

7. Waypoint style

It is a digit representing these values:
1 - Normal
2 - AirfieldGrass
3 - Outlanding
4 – GliderSite
5 - AirfieldSolid
6 - MtPass
7 - MtTop
8 - Sender
9 - Vor
10 - Ndb
11 - CoolTower
12 - Dam
13 - Tunnel
14 - Bridge
15 - PowerPlant
16 - Castle
17 - Intersection

8. Runway direction
It is a string in degrees representing heading of the runway. Only used with Waypoint style types 2, 3,
4 and 5

9. Runway length
It is a string for number with unit representing length of the runway. Only used with Waypoint style
types 2, 3, 4 and 5
unit can be either
"m" for meters
"nm" for nautical miles
"ml" for statute miles
Decimal separator must be a point.

10. Airport Frequency
It is a string representing the frequency of the airport. Decimal separator must be a point. It can also
be embraced in double quotes.

11. Description
It is a string field with no limitation in length where anything can be stored in. It should be embraced
with double quotes.

18 APPENDIX D – ELEVATION DATASET USAGE CONFIRMATION

Od: "Glenn Waldron" <gwaldron@pelicanmapping.com>
Komu: "Ales Zita" <ales.zita@volny.cz>
Předmět: Re: ReadyMap elevation data
Datum: 10.12.2011 - 15:53:03

Ales,

You are welcome to use the server for your project and for a freeware app.
Thanks for asking, and if you have any issues or questions, don't hesitate
to email me.

Glenn Waldron / Pelican Mapping / @glennwaldron

On Sat, Dec 10, 2011 at 5:57 AM, Ales Zita <ales.zita@volny.cz> wrote:

> From: Ales Zita <ales.zita@volny.cz>
> Subject: ReadyMap elevation data
>
> Message Body:
> Hello,
>
> I'd like to ask for permission to use the world-wide elevation data for
> > our school project.
>
> We are a group of 6 people developing the the glider flightpath visualisation tool in osgEarth environment as a school project at Faculty
> > of Mathematics and Physics by the Charles University in Prague.
> As this is non commertial project, we cannot afford any paid elevation
> > data sources. Therefore we would very much appreciate the use of your
> > ReadyMap global elevation dataset accessible via the osgEarth api.
>
> Should this project become available as a freeware at some point, can we
> > continue to use your server as a souce of the elevation data?
>
> Thank you
>
> Kind regards
>
> Ales Zita

```
>
> --
> This mail is sent via contact form on Pelican Mapping
> http://pelicanmapping.com
```

19 APPENDIX E – DVD CONTENT