MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Python Language Bindings for 3D Airspace Visualization

BACHELOR THESIS

Tomáš Králíček

Brno, Spring 2009

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Ing. Petr Gotthard

Acknowledgement

I would like to thank to my supervisor Ing. Petr Gotthard for his time spent by helping me and for his suggestions on this thesis.

Abstract

The goal of this thesis was development of the 3D visualization of the airspace. The visualization is implemented as the Python module written in C++ for the Python C API. The application is based on the Earth model from the OssimPlanet and on the OpenSceneGraph graphic library. The implemented application is a working proof of concept of the airspace visualization concept. The project enables implementation of other aeronautics objects for visualization.

Keywords

python, aeronautic, visualization, airspace, osg, ossim

Contents

List of Figures

Chapter 1

Introduction

The aeronautic branch is a very interesting area with application of many modern technologies. The airspace is today full of aircraft and there is a demand for simple and easy visualization of situations in the sky. Currently these situations can be viewed at the air traffic controller (ATC) stations or in the cockpits of modern airplanes with navigation displays. But both views are different and only professionals can understand them. There is a room for new approach to the aeronautical visualization. This thesis intends to provide realistic 3D view for demonstration or for presentations to the general public. The intention of this project isn't replacement of any operationally used visualization. It is intended for non-operational use only.

This project intends to provide a visualization module for a wide area of aeronautic data. The visualization is going to be in the three-dimensional space. This approach to the air traffic visualization is different because at the ATC station or in the aircraft there is today only two-dimensional visualization. The 3D visualization is more understandable to the common user, because the situation can be viewed from almost any angle.

To enable extensibility, the application interface will be provided in the Python [15] programming language. The functions will be implemented as a set of modules for the Python. These modules will be written in the C or C++ programming language with help of the Python C API. The rendering layer will use the OpenSceneGraph (OSG) [12] open source graphic library. For the Earth model and geodesic functions this project will use the Open Source Software Image Map (OSSIM) [14] and OSSIMPlanet libraries. The high level architecture of application modules is on the Figure 2.1.

1.1 About used libraries

This section describes libraries used in my application. There are used three main libraries in my program. These libraries are Python, Open Scene Graph and OSSIM with OSSIMPlanet.

1.1.1 Python

My application is designed as a module for the Python [15] programming language. This language is used as the interface between the user and a Python module which creates airspace and other graphics objects.

The Python is a multi-paradigm high-level programming language. It contains fully dynamic type system and automatic memory management. It is widely used as a powerful scripting language.

Python history begun in late 1980s. Main author of this language is Guido van Rossum and he still designate the course of Python development. The version 1.0 was released in January 1994. Actually last version is 3.0 which was released In December 2008. Python 3.0 was designed to reduce duplicated construction and modules. Thus Python 3.x series isn't compatible with previous Python 2.x series. Although both series are planed to coexists in side by side for several releases. Many of features of Python 3.0 was back ported to Python 2.6 for easier transition from 2.x series.

The Python provides easy extensibility with modules written in C or C++. These modules use Python C API (Application Programmers Interface). It provides access to the most of the Python run-time system. This API is used in my application to encapsulate the whole C++ program functionality into a user friendly Python module. Each module will visualize one of the proposed aeronautical 3D object. The modular architecture is shown on the Figure 2.1.

1.1.2 Open Scene Graph

For displaying 3D objects on computer screen is used the open source graphic library OpenSceneGraph (OSG) [12]. It's a cross-platform graphics toolkit written in C++ and OpenGL.

The OSG development was started in 1998 by Don Burns who is original author of SG (Scene Graph). This project then joined Robert Osfield in 1999. He ported scene graph to Windows. Then this project was turned into open source and Scene Graph became Open Scene Graph.

Development of this library still continues. Currently the last released version is 2.8.0. In my application I used tagged version 2.9.2 [13]. The role of this library is important for 3D scene graph management and for 3D object rendering.

The key concept of the OSG library is scene graph. The scene graph is "a hierarchical tree data structure that organizes spatial data for efficient rendering." [11, p. 13] In the scene graph are stored all geometry objects and their transformations. The scene graph diagram of my application is shown on the Figure 2.1 as a part of the high-level architecture diagram.

1.1.3 OSSIM

The abbreviation OSSIM means Open Source Software Image Map. It's an open source project which is a part of the OSGeo (Open Source Geospatial Foundation). "OSSIM is a high performance software system for remote sensing, image processing, geographical information systems and photogrammetry." [2] This project started in 1996 and it still in active development.

OSSIM is composed from several libraries written in C++. In my application is mainly used the OSSIMPlanet library. This library is build on the top of the OSSIM core library and the OSG. It controls planet node in the scene graph. The schematic view of the scene graph with the planet node is in the Figure 2.1. It also provides access to the computing of a various geodesic functions.

The application is currently in development and there are ongoing changes in some of the application parts. The last stable version is 1.7.15 [14] which I used.

1.2 Document structure

This document has a three main chapters. In the first chapter of this document I described aeronautical objects proposed for the 3D visualization. This study of aeronautical objects I did as a first step. In the second chapter there are described the geodesic related problems which appear to be very important after the study of the Earth visualization. The last chapter of this document deals with the implementation of the visualization of some of the proposed objects.

Chapter 2

Study of the Proposed 3D Objects

This chapter is the study of the future work related to this project. There is a study of aeronautical objects proposed for the 3D visualization of the airspace. There are summarized user expectations and user interface for each of these visualization modules.

In the application for rendering the aeronautical data will be possible to display aeronautical objects in three-dimensional space. Each of the following sections describes the way the user can insert an aeronautical objects to the application and what the application will render.

The user interface is based on Python modules that are controlling individual visualized objects. On the Figure 2.1 is shown the schematic high-level architecture of this application. On the Figure are from the left displayed the visualization output, the scene graph with the OSG root node, planet and two objects. On the right side of the figure is the Python module schema.

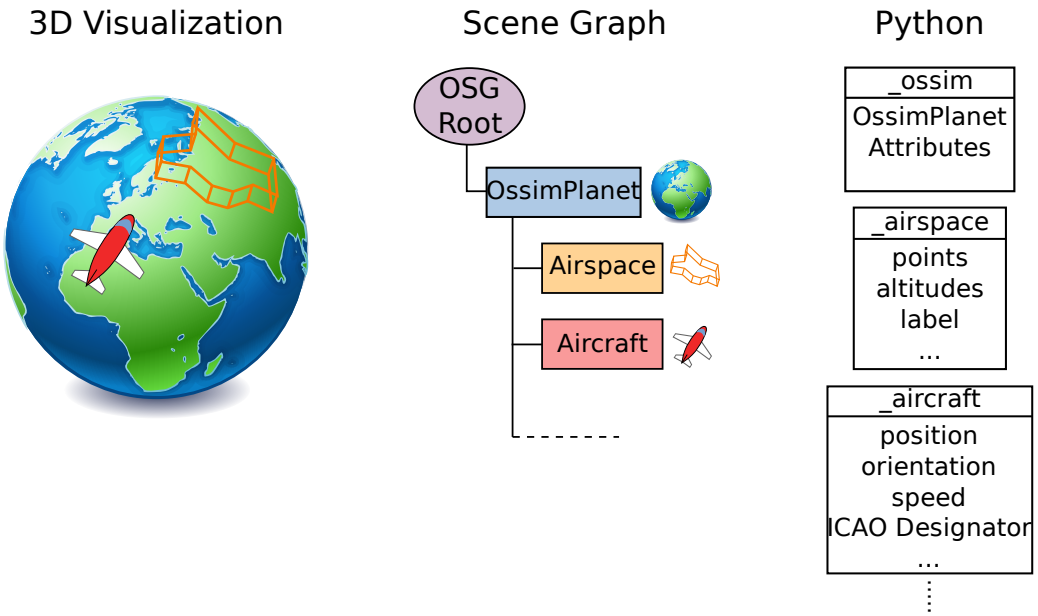


Figure 2.1: Program architecture diagram

2.1 Aircraft

In this section is described rendering of the aircraft[1] in the application. The aircraft will be represented by a static 3D models. Their position will be dynamically actualized by the user side Python script. Following objects will be rendered:

- model of the aircraft located in the airspace
- cylindrical protected airspace zone (PAZ) [19, p. 8] around the aircraft
- line from the aircraft reference point perpendicular to the Earth surface
- label with an ICAO aircraft type designator[2], speed, heading, altitude, vertical speed, world location

The aircraft will be initially created with the user defined properties. These properties are independent on the aircraft position and most of them is contained in the label:

- ICAO aircraft type designator
- optionally, call sign
- optionally, registration of the aircraft
- optionally, safe zone dimensions (radius and height)

The rest of the properties must the user enter each time the aircraft position is changed. This process will be done by the Python aircraft object methods. Data required for the actualization of the position will be:

- world location coordinates
- orientation (pitch, roll and yaw angles)
- optionally, speed and vertical speed

The world location coordinates corresponds to the latitude, longitude and height. The latitude and the longitude values are measured in a decimal degrees. The latitude is positive in the northern hemisphere and the longitude is positive in the eastern hemisphere. This is illustrated on the Figure 3.2 on the page 27.

The height or the altitude is measured in feet or in flight levels (FL)[3]. The altitude can be defined as the altitude above mean sea level (AMSL) or as the altitude above ground level (AGL). These altitude related terms are very important in the aeronautics. The mismatch of these altitudes can lead to the catastrophe so my application provide proper representation of these altitudes.

---

1. "Any machine that can derive support in the atmosphere from the reactions of the air other than the reactions of the air against the earth's surface." [9]
2. Aircraft Type Designators, ICAO Doc 8643, http://www.icao.int/anb/ais/8643es/index.cfm
3. "A surface of constant atmospheric pressure which is related to a specific pressure datum, 1 013.2 hectopascals (hPa), and is separated from other such surfaces by specific pressure intervals." [9]

2.1.1 Example

The example, of the each module, shows the way the user can enter the data to the Python script. These examples show also a parameters of these modules attributes.

For example, to display the aircraft as in Figure 2.2 the user should enter the following data into the Python script:

```
ICAO = 'B744'
PAZradius = 250
PAZheight = 120
```

To position the aircraft in the world or to update its location the user should enter:

```
location = (40.5408, -65.2794, ('FL 280', 'AMSL'))
orientation = (1.1, 0.0, -2.0)
```

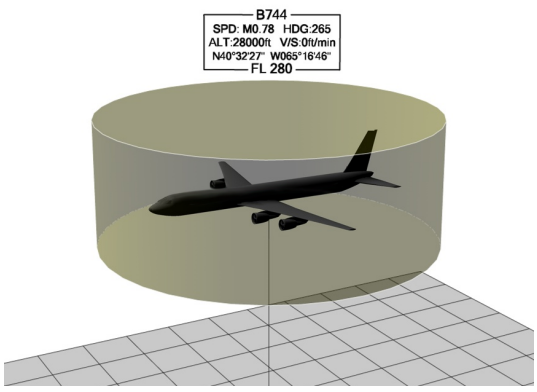


Figure 2.2: 3D render of the aircraft

2.1.2 Python Interface

The Python interface sections provide descriptions of the available module function with their correct parameter types. The functions with more complicated interface are described to more details.

To properly setup the aircraft the user must use following code in the Python. When user wants to use the `_aircraft` module, this module need to be imported first. Then the user creates the `aircraft` object. The Python aircraft object is constructed with the `planet` parameter. Planet is an instance of the module `_ossim` which contains the scene graph and the *OssimPlanet*. Methods of the `aircraft` object are used to set attributes of the visualized 3D object.

```
import _aircraft
aircraft = _aircraft.aircraft(planet)
aircraft.setICAO(<'ICAO DESIGNATOR'>)
aircraft.setPAZradius(<int>)
```

```
aircraft.setPAZheight(<int>)
aircraft.setLocation(<float>, <float>,
  (<int | 'FL int'>, <'AGL' | 'AMSL'>))
aircraft.setOrientation(<float>, <float>, <float>)
aircraft.addToPlanet()
```

After calling previous functions the aircraft will be prepared for adding into the scene graph as a new node. All functions with the set prefix contains only attributes of the newly created 3D object.

The `PAZradius` parameter and the `PAZheight` parameter define the radius and the height of the PAZ. The units used are meters and in the Python it is an integer number.

The function `setOrientation` has three parameters. The first one is a latitude and the second one is a longitude of the aircraft. Units used for the latitude and the longitude are always decimal degrees in the Python expressed as a decimal number. The third parameter is the altitude of the aircraft. The altitude is entered as a height in feet or in flight levels with defined altitude type. The possible types are altitude above mean sea level (AMSL) and altitude above ground level (AGL). This type of the world location entry is used in every module.

The aircraft orientation is set by three values. It is the aircraft pitch, heading and roll. Units used for these values are decimal degrees.

The function which needs to be called last will add the 3D model of the aircraft with the label to the planet. This function is:

```
aircraft.addToPlanet()
```

The aircraft model can be loaded from the FlightGear[4] model library. The type of the aircraft will be recognized from its ICAO aircraft type designator.

Animation of the aircraft movement in the space will be done in a rendering cycle of the scene. For each position actualization will be need to enter data about world location of the aircraft and optionally the orientation of the aircraft. This is done via functions:

```
aircraft.setLocation(*location)
aircraft.setOrientation(*orientation)
```

2.2 Aircraft Intent

The aircraft intent is an "information on the planned future aircraft behavior that can be obtained from the aircraft systems" [6, p. 4]. This information is calculated from the current aircraft state and the predefined route.

The aircraft intent will be represented in the application as shown in Figure 2.3. It will consist of:

4. An open source flight simulator. http://www.flightgear.org/

- semi-transparent tunnel representing the intended trajectory of the aircraft
- radius representing the tolerance of the intended trajectory
- line in the center of the tunnel

This tunnel will be dynamically actualized by the user. User will enter a list of waypoints with a description of turns. Each waypoint will represent a single trajectory change. This means when in the intent is any change of the altitude or of the flight direction there will be this change represented by new entry in the waypoints list. This is shown in the subsection 2.2.1.

For first intent initialization will be required following data from user:

- aircraft intent tolerance

Then every time the user wants to change the intent shape the user must enter:

- trajectory stored as a list of the waypoints with description of the turns

Each waypoint will be defined by a world location coordinates and an altitude. Turns will be defined only by a radius in meters.

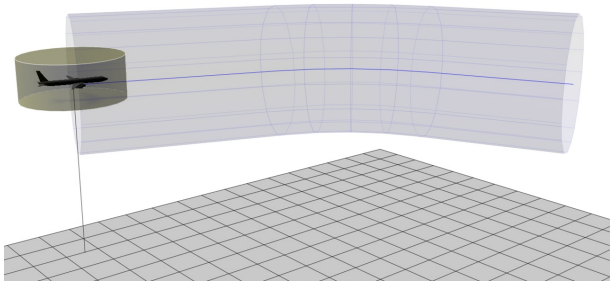


Figure 2.3: 3D render of the aircraft intent

2.2.1 Example

There is the example of an intent initialization and trajectory data entering in the Python:

```
tolerance = 150
waypoints = (
  ((49.2191, 17.0263), (8000, 'AMSL')),
  ((49.1710, 16.7883), (8000, 'AMSL')), # start of the turn
  (8920.0),                             # radius of the turn
  ((49.1813, 16.6023), (8000, 'AMSL')), # end of the turn
  ((49.3083, 16.2363), (8000, 'AMSL'))
```

2.2.2 Python Interface

The interface of the `_intent` module is similar to the aircraft module interface. For use of the intent 3D object in the 3D scene is need to import the `_intent` module and then set the tolerance parameter in meters.

```
import _intent
intent = _intent.intent(planet)
intent.setTolerance(<int>)
intent.addToPlanet()
intent.setTrajectory(
  [(<<float>, <float>, (<int | 'FL int'>, <'AGL' | 'AMSL'>)> |
  <float>),]...
)
intent.create()
```

The function `setTolerance` sets the intent tolerance value. This value is in meters.

The function `setTrajectory` needs to be called every time the intent is changed. This function will recreate the intent model and update this model in the scene graph. The parameter of this function is a list of intent points of trajectory change. The list can contain world location points and entries with turn radius. In the case of the radius entry the point previous in the list is the initial point of the turn and the point which follow immediately after the radius entry is the final point of the turn.

2.3 Airspace

The airspace is a three-dimensional portion of the atmosphere. It covers a specific area of the Earth. Each airspace can be either controlled[5] or uncontrolled airspace. The ICAO classifies the air traffic services (ATS) airspace[6] by classes from A to G. The classes from A to E are controlled and the classes F and G are uncontrolled. The ATS are divided also into a several divisions.

The airspace will be in application rendered as on Figure 2.4 on the page 14. The airspace will be represented by:

- semi-transparent polyhedral boxes, cylinders or a combination of these shapes with a user defined colour
- label with the airspace name, class and airspace height limits

Airspace sides will be perpendicular to the Earth reference ellipsoid model (see the Section 3.1.1. The shape of the airspace can be very various. I divided these shapes into

5. "An airspace of defined dimensions within which air traffic control service is provided in accordance with the airspace classification." [8]
6. "Airspaces of defined dimensions, alphabetically designated, within which specific types of flights may operate and for which air traffic services and rules of operation are specified." [8]

three categories. The polygonal, circular and generic shape airspace are described. All three possible airspace shapes are shown in Figure 2.4.

Airspace will be defined as:

- a list of the geographical coordinates with possible description of arcs in the airspace shape
- lower and upper altitude in feet or flight levels
- optionally the airspace name and colour and other properties

An arc in the airspace shape will be defined by the world location coordinates of the arc centre and by the arc radius. The arc initial point will be defined by the preceding point in the list. The arc end will be determined by the following point in the list. For more details about arc definition see example with *Generic airspace shape* on the page 15 or subsection *setShape(*points)* with illustration on the page 17.

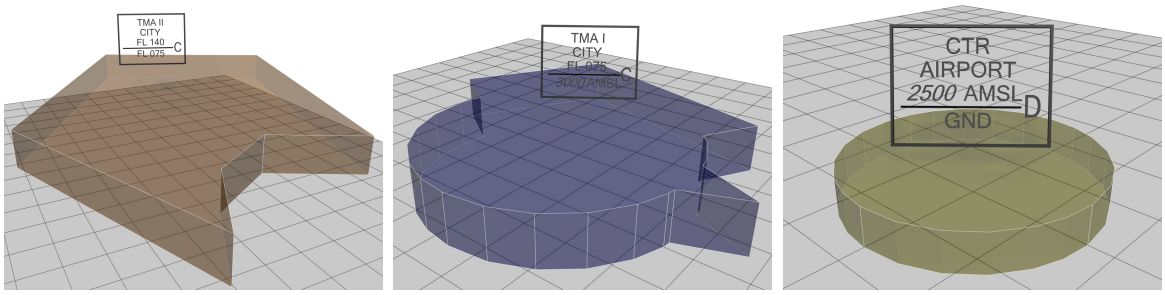
The airspace with the circle ground projection will be defined by the world location coordinates of its centre and by the circle radius in meters. Example is on the page 15 in the subsection *Circle airspace*.

The airspace area height can be entered as an altitude above ground level (AGL) or as an altitude above mean sea level (AMSL). Measuring units used to define the altitude are feet or flight levels. The airspace is then defined from the bottom to the upper altitude.

Optionally user may define the way the airspace will be rendered. It could be for example color or transparency.

2.3.1 Examples

In this subsection are described possible airspace shapes with an example data entry for each shape.



(a) polygonal airspace (b) airspace composed from cylinder and polygon (c) cylindrical airspace

Figure 2.4: 3D renders of possible airspace shapes with label examples

Polygonal Airspace

The polygonal shape is the one which contains only straight lines and no arcs. These lines are in the airspace module computed by the great circle functions (see the Section 3.3).

User will use following style of code entry to create the airspace with a polygonal shape like on Figure 2.4(a).

```
points = (
  (50.7746, 15.0230),
  (50.8528, 15.1570),
  (50.8488, 15.4721),
  (50.8058, 15.3701),
  ...
)
altitudes = (('FL075', 'AMSL'), ('FL 140', 'AMSL'))
label = ('TMA II', 'CITY', 'C')
color = (0.6, 0.3, 0.0, 0.5)
```

Circle Airspace

The circle airspace is composed from only one circular shape. These airspace are usually defined by a radius from a navigation point. This point is then the center of the circle and its coordinates are the only one entry in the `points` list. These types of the airspace are represented by the small circles. These geodesy objects are described in the Section 3.4.

The user has to enter data in the following way to create the airspace with a circle shape:

```
points = ((40.6328, -73.7713, 8000),) # note the trailing comma
altitudes = ((0, 'AGL'), (3000, 'AMSL'))
label = ('CTR', 'AIRPORT', 'D')
color = (1.0, 1.0, 0.0, 0.5)
```

Generic Shape Airspace

Airspace of this type has shape in which are straight segments and also arcs. This is the combination of previous two examples. And also representation of these airspace is composed from combination of the great and small circle geodesy lines. The airspace of this type is on the Figure 2.4(a).

The Python script entry is very similar to the previous ones. There is short code with example of an arc entry in the `points` list.

```
points = (
  (-18.0123, -70.0468),          # start of the arc
  (-18.0578, -70.2763, 14000),   # arc centre and radius
  (-18.1001, -70.0366),          # end of the arc
  (-18.2045, -69.9360),
```

```
  ...
)
altitudes = ((3000, 'AMSL'), ('FL 075', 'AMSL'))
label = ('TMA I', 'CITY', 'C')
color = (0.0, 0.0, 1.0, 0.5)
```

2.3.2 Python Interface

New airspace and areas will be added to the scene graph by the following functions of the Python airspace object. The `_airspace` module must be imported before it can be used in the Python.

```
import _airspace
airspace = _airspace.airspace(planet)
airspace.setShape(
  [(<float>, <float>[, <float>]),]...
)
airspace.setAltitudes(
  (<int | 'FL int'>, <'AGL' | 'AMSL'>),
  (<int | 'FL int'>, <'AGL' | 'AMSL'>)
)
airspace.setLabel(<'TYPE'>, <'Name'>, <'CLASS'>)
airspace.setColor(<float>, <float>, <float>, <float>)

airspace.create()
airspace.addToPlanet()
```

Initialization of the airspace is done via a constructor. In this example it is the line `airspace = _airspace.airspace(planet)`. Where `airspace` is a name of the newly created Python object of the `airspace` type. The `_airspace` is the name of the Python module in which is the airspace type defined.

The parameter `planet` is a pointer to the planet object which contains scene graph. This parameter is there because airspace is then added to the scene graph with utilization of the planet pointer.

The airspace upper and bottom altitude is set by the `setAltitudes(*altitudes)` function. The parameter `altitudes` contains two pairs of altitudes that specifies bottom and upper airspace border. The format still the same as was described in previous sections with visualized objects.

Text attributes that are displayed on the airspace label are set in the `label` parameter. It contains airspace type, name and class. The order of this airspace attributes is the same as is described here. All the values are text strings.

The airspace color can be defined by the function `setColor(*colors)`. This function will set also the transparency. The function has four parameters. The first three

parameters represent RGB color channels. The fourth parameter is the transparency. All parameter values are between 0.0 and 1.0.

setShape(*points)

Parameter of this function is a list of geographical coordinates and optionally there are described an arcs. This coordinates define the shape of the airspace. For example airspace shape could look like on the Figure 2.5. On this figure is shape of the Brno Tuřany CTR.

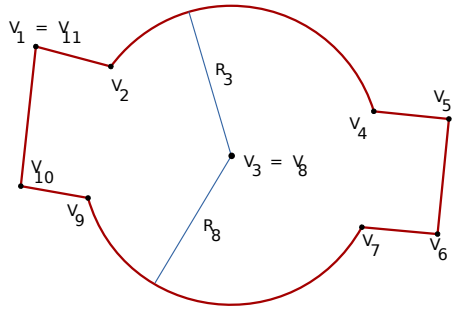


Figure 2.5: Airspace shape

Each geographical coordinate in the list is defined as a latitude and a longitude. These coordinates are for example points V_1, V_3 or V_5 on the figure. Each arc is defined by its centre coordinates and radius. The arc centers are points V_3 and V_8 and radius is defined by values R_3 and R_8.

The arc entry in the `points` list is composed from a geographic coordinates which specify centre of the arc and one more item which is decimal number which specifies the arc radius in meters.

If in a list is defined an arc by specifying its center it must be preceded by a geographic coordinates specifying beginning of the arc and it must be followed by an other geographic coordinates specifying end of the arc. On the figure 2.5 the point V_3 is the center of the arc. So in the `points` list this point must be preceded by the point V_2 and followed by the point V_4. Then the Python script looks like here:

```
points = (
  ...
  (V2),
  (V3,R3),
  (V4),
  ..
)
```

The airspace with the circle shape is defined by only one entry in the `points` list.

The circle is entered same way as an arc is defined. Because in the `points` list is only one item there is need for trailing comma as seen in the example in the subsection *Circle airspace* on the page 15.

When entering an area to Python script there must be at least four entries in the `points` list. This is caused by requirement to begin and finish the list with same coordinates. So airspace created by four coordinates has triangular shape.

The `points` list needs to be sorted. Point coordinates must be sorted clockwise. If points aren't sorted the result is undefined.

create() and addToPlanet()

These functions have no parameters but they do all the work. The function `create()` will make the airspace geometry object and label. And the function `addToPlanet()` will add the created object into scene graph. These functions must be called in the described sequence and after all set functions. The implementation of the `create()` function is described in the last chapter in the Section 4.1.2 on the page 37.

2.4 Navigation Aids

Navigation aids are usually ground based radio stations. These navigation aids help with navigation of the aircraft during the flight. Today many of these navigation aids are defined only by world location coordinates.

Visualization of the navigation aids displays the positions and the information about the navigation aids to user. There is many types of these aids and the visualization should provide simple but precise representation of the various aids for quick recognition of their types. Therefore the style of displaying the navigation aid will respect the ICAO Annex IV standard for the aeronautical charts [8].

All navigation aids will be base on the ground at the user defined world coordinates. The final navigation aid 3D objects will consist of:

- line or cone perpendicular to the Earth surface
- symbol representing navigation aid
- label with information about the navigation aid

Both the label and the symbol depends on the type of the navigation aid. Navigation symbols will respect the navigation aid symbols drawn in *Appendix 2. ICAO Chart Symbols* in table *Radio Navigation Aids* and *Additional Symbols For Use On Paper And Electronic Charts* in ICAO Annex 4 [8].

The proposed symbol design based on the Annex 4 and the study of symbols for electronic displays [21] is in the table 7.1 in the Appendix on the page 49.

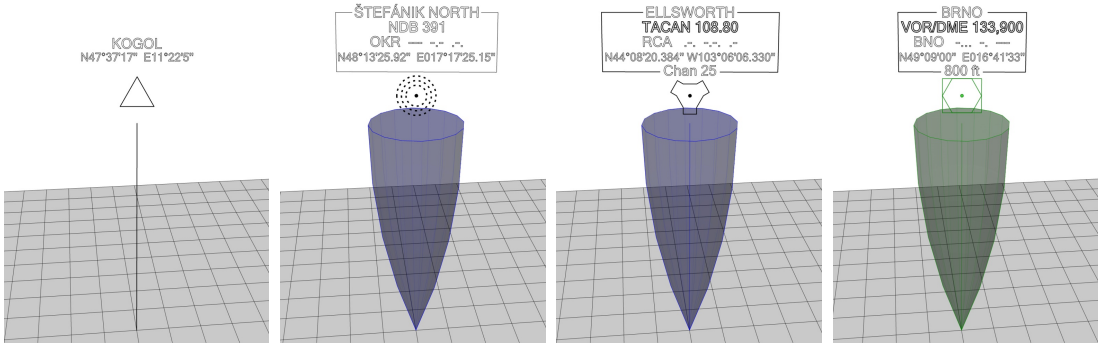
For display the navigation aid in the application the following data will be required:

- world location coordinates of the navigation aid
- navigation aid type
- optionally ICAO identifier, name, frequency and altitude of the navigation aid

2.4.1 Example

To create a VOR/DME radio navigation aid like on Figure 2.6(d) enter the following data:

```
location = (49.1500, 16.6888, (800, 'AMSL'))
type = 'VOR/DME'
label = ('BNO', 'BRNO', '133.9')
```



(a) FIX (b) NDB (c) TACAN (d) VOR/DME

Figure 2.6: Examples of navigation aids

2.4.2 Python Interface

All navigation aid types are represented as 3D objects and can be added to the scene by the following script:

```
import _navaid
navaid = _navaid.navaid(planet)
navaid.setLocation(<float>, <float>,
  (<int | 'FL int'>, <'AGL' | 'AMSL'>))
navaid.setType(<'NAVAID TYPE'>)
navaid.setLabel(<'ICAO ID'>, [<navaid specific data>]...)
navaid.addToPlanet()
```

Position of the navigation aid is set via the function `setLocation(*point)`. There is specified a geographic position of the navigation aid. The syntax is the same as in visualized objects mentioned in previous sections.

The function `setType(*type)` sets the type of the navigation aid. There are several possible types of navigation aids. This module recognize types described in the table 7.1. If the type isn't set the default one is used.

The label is again set by the function `setLabel(*label)`. The `label` parameter describes aeronautical properties of the navigation aid. There are stored all information about the navigation aid. User has to specify here information like the ICAO identifier, name or the radio frequency.

2.5 Airways

Airways, in mean of the ATS routes[7], are used for controlling the traffic flow in the airspace. Aeronautic charts represent airways by colour lines. These lines connect various types of navigation aids. Airways are usually divided into at least two main groups by the local country airspace altitudes. Those are lower routes in the lower airspace and upper routes in the upper airspace. These airways are differed by a colour in the charts and usually by a prefix in the airway ICAO identifier.

The airway will be displayed as on the Figure 2.7. The 3D model will consist of:

- block segments with defined height and tolerance
- label over segments with information about route

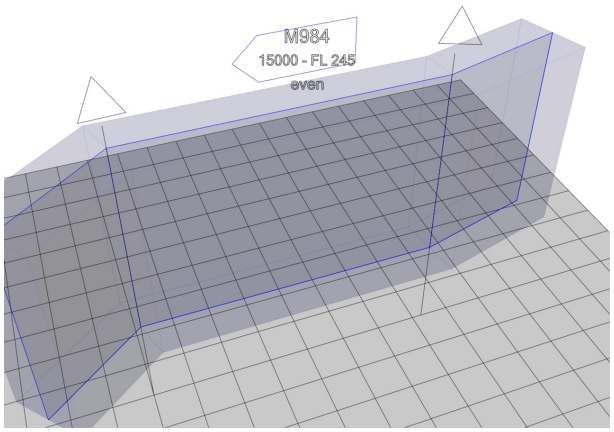


Figure 2.7: 3D render of airway

Data required for the route creation will be:

- tolerance defined for the whole airway
- list of airway waypoints

7. "A specified route designed for channelling the flow of traffic as necessary for the provision of air traffic services." [8]

- lower and upper altitude of the airway at each waypoint
- direction of the traffic flow and even flight levels defined at each segment
- optionally, airway ICAO identifier

The bottom and the upper altitude of the airway will be defined in a similar way as this is defined in the case of an airspace object. There will be a bottom and an upper altitude with a possibility to be above ground level (AGL) or above mean sea level (AMSL).

The direction of the traffic flow of the airway and eventually if the airway is only one way will be defined by special keywords. If the airway is defined in both direction the keyword `'BOTH'` needs to be specified in the point entry. If the airway is one way in the direction of the point entry in the list, then the keyword `'FWD'` as forward is used. In the other case when the airway is one way in the direction reversed to points entry in the list, the keyword `'RWD'` is used.

The airway has an attribute which indicate even flight levels. This is used for separation of an opposite traffic. For example if an aircraft flying to the west has an even flight level then an other aircraft flying in the opposite direction has an odd flight level. This is the semi-circular cruising level system which is also known as NEODD-SWEVEN rule (north-east is odd, south-west is even). There can be exceptions in this rule so it needs to be possible to define this.

The semi-circular cruising level system is defined at each point in the `points` list by one of the four world cardinal directions. The possible keywords are `'NORTH'`, `'EAST'`, `'SOUTH'` and `'WEST'`. This keyword specifies the semi-circle where are flight levels even. So if route is defined with keywords `'WEST'` this means that flight levels heading to the west are even and flight levels in the opposite direction, to the east, are odd.[8]

Labels will be displayed above the airway segment. It will contain the airway name, height restrictions and the airway direction information. This label will be above segments, where data about altitude or direction is other than on the previous one segment.

2.5.1 Example

To create an airway the syntax of attributes in the Python is following:

```
points = (
  ((48.0703, 15.8806), ((9000, 'AMSL'), ('FL 245', 'AMSL')),
  'FWD', 'EAST'),
  ((47.6111, 15.6620), ((15000, 'AMSL'), ('FL 245', 'AMSL')),
  'BOTH', 'WEST'),
  ...
  ((46.8025, 12.2811), ((15000, 'AMSL'), ('FL 245', 'AMSL')),
  'RWD', 'EAST'),
```

8. More information about flight levels are in the Appendix 3 in the ICAO Annex 2 [7].

```
  ...
)
thickness = 9260.0
label = ('M984',)
```

2.5.2 Python Interface

The airway module functions are following. Again, the module needs to be imported first. The `airway` is the name of the instance of type `_airway.airway`.

```
import _airway
airway = _airway.airway(planet)
airway.setCoordinates(
  [(<float>, <float>, (<int | 'FL int'>, <'AGL' | 'AMSL'>),
  <'BOTH' | 'FWD' | 'RWD'>,
  <'NORTH' | 'EAST' | 'SOUTH' | 'WEST'>)]...
)
airway.setTolerance(<int>)
airway.setLabel(<'ICAO ID'>)
airway.addToPlanet()
```

The airway tolerance is set by the function `setTolerance(*thickness)`. The parameter thickness is the tolerance in meters. The tolerance is constant for the whole airway.

The label is created mainly from parameters of other functions. The attributes set by the function `setLabel(*label)` are only the ICAO identifier of the airway and a type of the airway. Type of the airway is the airspace where the airway is situated. This could be upper or lower airspace.

setCoordinates(*points)

This function specifies the shape of the airway. The parameter `points` is a list of geographical coordinates with an airway altitude and direction attributes. Each entry in the list represents one point on the airway. Each point is a pair of two decimal numbers which specifies a latitude and a longitude. This numbers are followed by the item which contains bottom and upper altitude definition. The last two text string items in the entry are the direction and flight levels definitions. Points in the tuple must be sorted as they follow on the route in the user specified direction.

2.6 Airports

Airport or aerodrome is a "defined area on land or water (including any buildings, installations and equipment) intended to be used either wholly or in part for the arrival,

departure and surface movement of aircraft." [8] Airports are represented on aeronautical charts by specified symbols or by runways on an airport.

The airport visualization will consist of:

- reference point with airport symbol
- label with airport aeronautical data

The visualization will not contain runways or an airport equipment. This details are not important for this visualization.

The user will only have to specify a world location of the airport and its properties. In properties of the airport will be specified the ICAO airport identifier, name, city, country and the airport runways count. Then will be possible to specify if the airport is civil or military.

2.6.1 Example

Attributes for create a new airport in the scene are following:

```
location =  (49.1513, 16.6938, (774, 'AMSL'))
label = ('LKTB', 'Turany', 'Brno', 'CZ', 'civil', 2)
```

2.6.2 Python Interface

The interface of the `_airport` module has only two functions for the new airport adding to the scene.

```
import _airport
turany = _airport.airport(planet)
turany.setLocation(
  <float>, <float>, (<int | 'FL int'>, <'AGL' | 'AMSL'>)
)
turany.setLabel(<'ICAO ID'>[, <'Name'>, <'City'>, <'COUNTRY ID'>,
  <'civil' | 'military'>, <int>])
turany.addToPlanet()
```

The position of the airport is set by the function `setLocation(*location)`. The parameter is the world location of the airport together with the airport altitude.

Airport attributes are set by via the label. The `setLabel(*label)` function specifies various data about airport. The first item in the `label` tuple is the ICAO airport identifier. This is the most important value and following values are optional. These text strings are official name of the airport, the airport city, country identifier, specification if the airport is civil or military. The last entry in the tuple is the count of a runways specified as an integer number.

Chapter 3

The Earth Representation

This chapter is focused on the geodesy related topics. The geodesy is a very important for working with the earth model in the OSSIM. The comprehension to the geodesy was very important for me and maybe the most difficult. There are many important things to understand because wrong implementation can lead to non precise representation of the visualized 3D object.

Following three sections describe geodetic data, earth coordinate systems, geodetic curves. In each section is described usage in my application.

3.1 Geodetic Datum

In geodesy, a datum describes a shape and a size of the Earth. It also describes the Earth origin and orientation of the coordinate system. During history many datums have been used to describe position on the Earth and for computing distances. Currently there are used datums varying from flat models for short distances, over spherical models which are used for distance approximation or ellipsoid models to complex geodetic model based on the gravity field.

Due to the amount of a geodetic datums today, there is a need to precise datum selection. Referencing to the wrong geodetic datum can cause significant error in computing. Thus there is need for accurate conversions among geodetic datums.

3.1.1 Ellipsoidal Datum

The ellipsoidal shape of the earth is represented as a rotational oblate spheroid. There are two axis of the ellipsoid. On the earth it is a semi-major axis which is in the equatorial plane and the second is a semi-minor axis which is from the ellipsoid center towards poles. These axis are denoted by the letter a for semi-major axis and for semi-minor axis it is the letter b. Ellipsoid is defined by these two axis or usually by the semi-major axis a and the ellipsoid inverse flattening $\frac{1}{f}$. Inverse flattening is computed as $\frac{1}{f} = \frac{a}{a-b}$.

Ellipsoidal approximation of the earth is better than spherical approximation (Section 3.1.2) and is good for distance measurement. It has maximum error of about 100m from geoid. The ellipsoidal figure of the earth is also used in GPS navigation and in the aeronautic. There is used ellipsoid WGS-84. The ICAO "adopted in 1994 the World Geodetic System — 1984 (WGS-84) as a common horizontal geodetic reference system

for air navigation with an applicability date of 1 January 1998." [10] The parameters of the WGS-84 geoid are in the Appendix in the Table 7.2 on the page 50.

This ellipsoid is used in many parts of my application. Almost all geodesic functions in my application are using the ellipsoid for computing. Especially in conversions from geodetic coordinates to geocentric coordinates is used the ellipsoid datum (see the Section 3.2).

Geoid

The World Geodetic System is used together with geoid. The geoid is the vertical datum used for the ellipsoid. The geoid is "The equipotential surface in the gravity field of the Earth which coincides with the undisturbed mean sea level (MSL) extended continuously through the continents" [8]. It means that in each point of the geoid is the gravity force perpendicular to the geoid surface.

The geoid corresponds to the undisturbed sea level. Thus the geoid is used as the MSL for the computing. The reference among the earth surface, the sea level, the reference ellipsoid and the geoid is on the Figure 3.1. The distance between geoid and ellipsoid is known as the geoid undulation [8].

For WGS-84 was originally used the WGS-84 geoid. This was revised in the 1996 by the EGM-96 (Earth Gravitational Model) geoid which is more accurate.

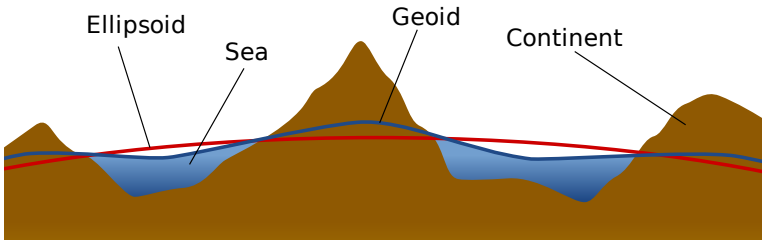


Figure 3.1: Relation among the geoid, the Earth and the reference ellipsoid

3.1.2 Spherical Datum

The easier way to approximate the earth is the spherical datum. This model was firstly used by Aristotle when he made first approximation of the earth shape and size. This model is still used today as a simplifier form than the ellipsoid model. It's good for fast approximation of short distances. For example the DME (Distance measuring equipment) navigation aid uses spherical approximation for computing distance of an aircraft from a DME ground transponder.

Spherical approximation is can be used for local partial substitution of the ellipsoid or for replacement of the whole ellipsoid.

For substitute of the whole oblate ellipsoid is used the sphere with a reference radius. This sphere is used in a simple distance calculations on the earth. The radius

of the reference sphere is derived from the reference ellipsoid parameters. There are more ways to do this. The reference sphere can has the same volume as the ellipsoid or the same surface area as the ellipsoid. Or the reference sphere radius is equal to the arithmetic mean of reference ellipsoid radii.

The radius used in my application in the class `SphericalGC` (see the Section 3.3.2) is computed using the arithmetic mean of the ellipsoid radii [3].

$$R_m = \frac{a + a + b}{3} \tag{3.1}$$

For ellipsoid WGS-84 is the $R_m \approx 6371009m$.

3.2 Geographic Coordinate Systems

The geographic coordinate system allows to accurately address every place on the Earth. Following sub-sections describe coordinates systems used in my application. There are two of many existing coordinate systems.

3.2.1 Latitude, Longitude, Height

This coordinate system is today the most used in the cartography and other branches of geodesy. The latitude and the longitude are angles from a reference planes which is the equatorial plane for the latitude and the prime meridian for the longitude. This is illustrated on the Figure 3.2.

The latitude and the longitude representation in my application is based on the signs of the corresponding values.

The equatorial plane divides the earth into Northern and Southern Hemispheres. The northern latitude is positive number and the southern latitude is negative. The range of the latitude is from $-90°$ to $90°$.

For latitude is the earth divided to Eastern and Western Hemisphere. The eastern latitude is positive and the western is negative. The range of the longitude is from $-180°$ to $180°$.

Geodetic and Geocentric Latitude

Because of the earth approximation to the reference ellipsoid (see the Section 3.1.1) there are two latitude types[1]. The commonly used latitude is the geodetic. This is used on charts. But for work with the coordinates in the three-dimensional space is better to use some geocentric coordinates.

The geodetic latitude in the point P, on the Figure 3.3(a), is the angle φ between the equatorial plane and the surface normal vector of the ellipsoid in the point P. The

1. When the spherical approximation of the earth is used the geocentric and the geodetic latitude is same.

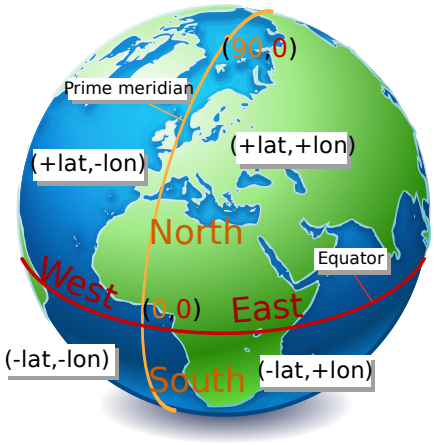


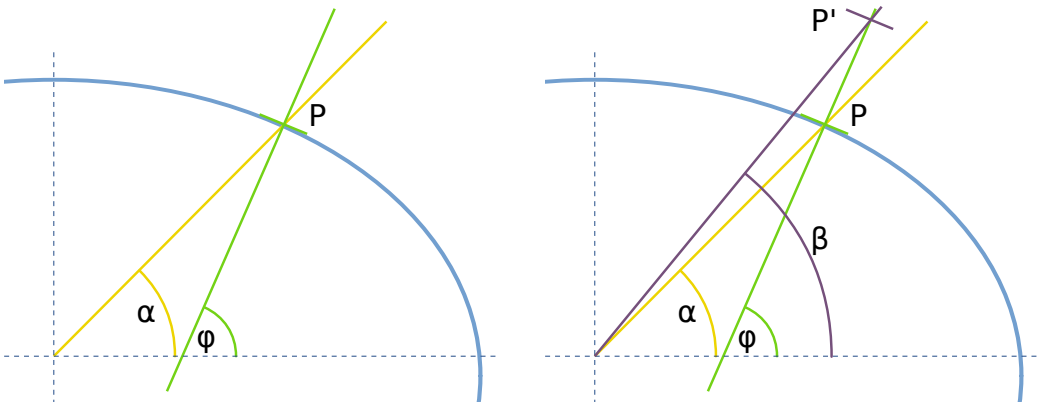
Figure 3.2: The signs used in latitude and longitude over the Earth

geocentric latitude in the point P is the angle α between equatorial plane and the vector connecting center of the ellipsoid with the point P.

The difference between this two types of latitude is the highest for the $45°$ of geodetic latitude. There the geocentric latitude differs from the geodetic by $11.67'$.

The geocentric latitude α can be computed from the geodetic by the equation 3.2.

$$\alpha = \arctan\left(\left(\frac{b}{a}\right)^2 \cdot \tan\varphi\right) \tag{3.2}$$



(a) Geodetic and geocentric latitude

(b) Geocentric latitude and height

Figure 3.3: Geodetic and geocentric latitude

Height

Height is usually represented as a relative vertical distance from some reference surface. It could be distance from the ellipsoid but in geodesy is more common distance from the mean sea level (MSL). Therefor the geoid (see the Section 3.1.1) is used in computing.

With the height measurement is connected problem with geocentric latitude. If I have two points P and P' with different altitude above the ellipsoid as on the Figure 3.3(b). Then these points have the same geodetic latitude φ but they have different geocentric latitudes. The point P has geocentric latitude α which is smaller than the geocentric latitude β of the point P'. The difference between geocentric latitudes is more significant with increasing altitude difference between these points.

3.2.2 Earth-Centered, Earth-Fixed

The ECEF is a three dimensional Cartesian coordinate system. The origin of the ECEF system is in the mass center of the Earth, hence the Earth-Centered. It has Cartesian coordinates $(0, 0, 0)$. The X-axis is from the origin towards the latitude $0°$ and longitude $0°$. This is the intersection of prime meridian and the equatorial plane. The y-axis is from the origin towards the coordinates $N0°, E90°$. The z-axis is the ECEF rotation axis. The z-axis is parallel to the Earth rotational axis and it is heading to the north. The ECEF coordinate system rotates with the Earth, hence the Earth-Fixed. This coordinate system I use for representation of coordinates in the three dimensional space used by the OSG.

3.3 Great Circle

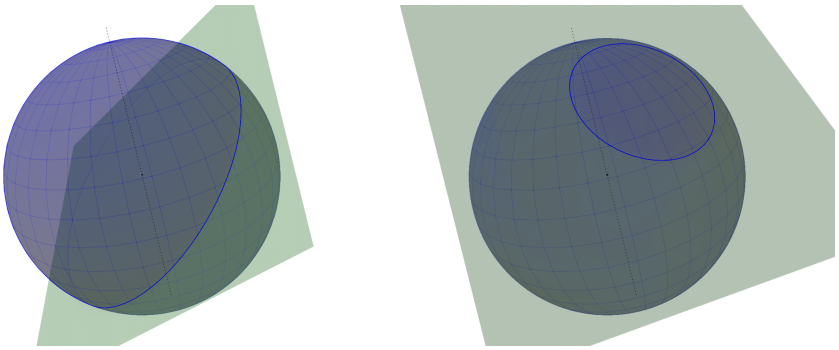
Great circle or *orthodroma* is the intersection of the sphere and a plane passing through the sphere center. The intersection is circle with the sphere radius. The equator is an example of the great circle. The simple visualization of the great circle is on the Figure 3.4(a).

In the following sub-sections are described the way the great circle distance is computed and description of the great circle implementations in the application.

3.3.1 Great Circle Calculation

The great circle segment is the shortest distance between two points. There is an equation 3.4 for finding the distance between two points P_1 and P_2 on the great circle with the spherical approximation of the earth. This equation is a partial solution of the second (inverse) geodetic problem[2]. Each point P is composed from a latitude φ and a longitude λ in radians.

2. Given two points, determine the azimuth and length of the geodesic line that connects them.



(a) Great circle (b) Small circle

Figure 3.4: Great and small circle

$$P_1 = \begin{bmatrix} \varphi_1 \\ \lambda_1 \end{bmatrix} P_2 = \begin{bmatrix} \varphi_2 \\ \lambda_2 \end{bmatrix} \tag{3.3}$$

$$d = R \cdot \arccos(\sin(\varphi_1) \cdot \sin(\varphi_2) + \cos(\varphi_1) \cdot \cos(\varphi_2) \cdot \cos(\lambda_2 - \lambda_1)) \tag{3.4}$$

Where R is a spherical earth radius (see Equation 3.1). The equation 3.4 refers to the section 2.4.2 Ortodroma [3].

The great circle solution for the reference ellipsoid is more complex. My program uses the algorithm known as Vincenty's formulae [17]. This algorithm was developed by Thaddeus Vincenty in 1975. See section 3.3.3 which is devoted to the Vincenty's formulae implementation.

As described in the previous section, the great circle is the shortest distance on the earth. Then each straight segment on the cartography map[3] is in reality the great circle. Straight segments are used in many proposed objects in the chapter 2. So function which will provide possibility of working with great circle is one of the most important.

The ability of computing great circles is located in the `utility` library. There is an abstract class `GreatCircle` which encapsulates the great circle functions. In the library are contained two classes which uses the `GreatCircle` class interface. It is the `SphericalGC` and the `VincentyGC`. When creating module, which will use the great circle, it is possible to use any class for computing the great circle. It is also possible to add new classes, with the same interface, which will provide functionality for the great circle computing.

3. The shape of the great circle is based on the map projection. The ICAO recommends for the most of the aeronautical maps: "A conformal projection on which a straight line approximates a great circle should be used." [8]

Segment Distance

Generally the great circle is a curve on the earth. In computer graphics this curve must be segmented into shorter lines. These lines must then appear as a curved line. I choose the maximum deviation from the optimal great circle curve as d which is $3m$.

$$R = d + d' \tag{3.5}$$

$$l = 2 \cdot \sqrt{R^2 - d'^2} \tag{3.6}$$

$$\varphi = 2 \cdot \cos\left(\frac{d'}{R}\right) \tag{3.7}$$

$$R = 6371009m \quad d = 3m \quad d' = 6371006m \tag{3.8}$$

$$l = 2 \cdot \sqrt{6371009^2 - 6371006^2} = 12365.45187m \tag{3.9}$$

$$\varphi = 2 \cdot \arccos\left(\frac{6371006}{6371009}\right) \approx 1.9409 \cdot 10^{-3} \tag{3.10}$$

This distance l is used in the program in the angular distance of $\varphi \approx 1.9409 * 10^{-3}$ radians. This length is valid on the sphere with the reference radius $R = 6371009m$ (see section 3.1.2) in the $0m$ height above this sphere. With a higher altitude the length is increasing. This distance is constant between every two points on the great circle.

The great circle is defined by its initial and final geodetic coordinates. The altitude of these points isn't important for computing. The great circle is computed at surface of the reference sphere or the reference ellipsoid.

3.3.2 SphericalGC Calculation

This class uses spherical approximation of the earth for the great circle calculations. This is sufficient for short distances. During the creation of the new `SphericalGC` instance, the distance and number of segments of the great circle is computed. For the distance is used algorithm which implements equation 3.4.

After the instance of the `SphericalGC` is created it is possible to call the function `getFinalCoordinates(distance)`. This function returns the geodetic coordinates of the point which is on the great circle at a given angular distance from the great circle initial point. The requested point on the great circle is computed in the following way.

I have defined two geodetic points $G_1 = \begin{bmatrix} \varphi_1 \\ \lambda_1 \end{bmatrix}$ and $G_2 = \begin{bmatrix} \varphi_2 \\ \lambda_2 \end{bmatrix}$ which define the great circle arc. The letters φ and λ correspond to the latitude and the longitude in radians. The great circle angular distance between G_1 and G_2 is σ. This angular distance is computed by the equation 3.4. The point, $G_3 = \begin{bmatrix} \varphi_3 \\ \lambda_3 \end{bmatrix}$, I am searching for is at the angular distance ω from the G_1. I can compute coordinates of the requested point in

the three-dimensional Cartesian coordinate system by the following equations from the Aviation Formulary [20].

$$A = \frac{\sin(\sigma - \omega)}{\sin \sigma} \tag{3.11}$$

$$B = \frac{\sin \omega}{\sin \sigma} \tag{3.12}$$

$$x = A \cdot \cos \varphi_1 \cdot \cos \lambda_1 + B \cdot \cos \varphi_2 \cdot \cos \lambda_2 \tag{3.13}$$

$$y = A \cdot \cos \varphi_1 \cdot \sin \lambda_1 + B \cdot \cos \varphi_2 \cdot \sin \lambda_2 \tag{3.14}$$

$$z = A \cdot \sin \varphi_1 + B \cdot \sin \varphi_2 \tag{3.15}$$

To convert Cartesian coordinates into the geodetic coordinates I can use following equations. The result latitude and longitude is in radians.

$$\varphi_3 = \operatorname{atan2}\left(\frac{z}{\sqrt{x^2 + y^2}}\right) \tag{3.16}$$

$$\lambda_3 = \operatorname{atan2}\left(\frac{y}{x}\right) \tag{3.17}$$

To place the computed point on the earth is needed to correct the height. The height is requested to be above mean sea level (AMSL) or above ground level (AGL). The MSL is represented as a geoid (see Section 3.1.1) and the GL is represented by the earth elevation model. But the computing with geoid or elevation model is much complex for the whole great circle.

So I need to know the offset of the sea level or the ground level from the referenced ellipsoid. These offsets can be get from the OSSIMPlanet functions. So I correct each great circle point height by the local geoid or elevation offset from the ellipsoid and I get the height above ellipsoid.

Now I have the world coordinates of the point with correct altitude above the ellipsoid. This point is then converted into ECEF coordinates (see Section 3.2.2) by `forward(gpt, ecef)` function provided by the OSSIM.

3.3.3 VincentyGC Calculation

This class provides also computing with the great circle but with much higher precision than the `SphericalGC`. There are algorithms for solution of direct[4] and inverse geodesics. These algorithms are implementation of the Vincenty's formulae. This formula works with the reference ellipsoid. The algorithm gives "complete accuracy over lines of any length, from few centimeters to nearly 20000km" [17, 88]. The great circle distance between two points computed by this class "may be in error by up to 0.5mm, which represents $0.000015''$ in the direction of the line." [17, 91]

4. Given a point and the azimuth and distance from that point to a second point, determine the coordinates of the second point.

This class has the same interface as the `SphericalGC` so it provides the same functionality. Inside the constructor are set parameters of the reference ellipsoid. In my application it is the WGS-84 ellipsoid. There is also called the function for the solution of the inverse geodetic problem. The output of this function contains three values. It is the distance of the great circle arc in radians and initial and final azimuth of the great circle.

After the `VincentyGC` instance is created the `getFinalCoordinates(distance)` function can be called. This solve the direct geodesic problem and returns final coordinates in a specified distance from the initial point of the great circle.

3.4 Small Circle

The great circle is used for visualize a straight lines from charts to the great circles on the earth model. And small circle is used for visualize a circular arcs from two-dimensional charts to a three-dimensional small circles on the earth model. This is widely used for modeling airspace in my application.

The small circle is also intersection of a plane and the sphere like in the case of the great circle. But the plane doesn't passing through the center of the sphere. Thus examples of the small circles are all parallels of latitude except the equator. The small circle render is on the Figure 3.4(b) or on the Figure 3.5.

The method used for drawing the small circle is diametrically different from methods used in the great circle drawing. There is again abstract class used as interface for the small circle computing in my application. And there is one class using this interface. It is called `VectorSC` and it provides all functionality for the small circle drawing.

3.4.1 VectorSC

The small circle drawing in this class is based on the vector representation of the point. This vector is rotated around a small circle center. Rotation of this vector around $360°$ creates a complete small circle.

The following two sub-sections describe the creation of a complete 360 degree small circle and the creation of a sector of a small circle. The last sub-section is devoted to the brief description of the height correction on the small circle points.

The earth is approximated by the referenced ellipsoid. It is the WGS-84 ellipsoid (see the Table 7.2 and the Section 3.1.1) as every where else in my application.

Creation of a Complete Small Circle

The approach to the small circle drawing uses the Cartesian coordinate system to represent each point in the space. These xyz coordinates are used to represent the point as

a vector. For small circle, SC, creation is essential the small circle center, $P_c = \begin{bmatrix} \varphi_c \\ \lambda_c \end{bmatrix}$, and its radius, R. Illustration of the small circle SC is on the Figure 3.5. The center is entered in the constructor of the `VectorSC` in geodetic coordinates and then converted to geocentric coordinates in the Cartesian system. Cartesian coordinates of the point P_c then represent the first vector, V_c. The second vector is made by adding the radius R of the small circle as an angular distance α to the geodetic latitude φ_c of the small circle center. Thus the geodetic coordinates of the first small circle point P_s are $\begin{bmatrix} \varphi_c + \alpha \\ \lambda_c \end{bmatrix}$. Next points are constructed by rotating a vector constructed from the P_s around the vector axis V_c.

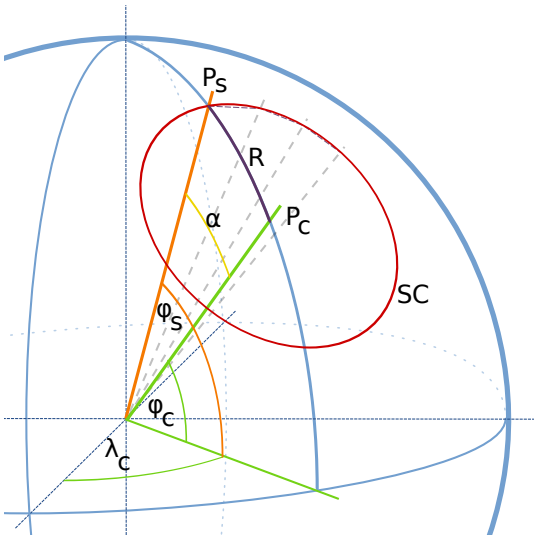


Figure 3.5: VectorSC illustration

For the conversion of the small circle radius R into the arc length α in the meridional direction is needed some computing. The first thing is to compute the *meridional radius of curvature*, M. The meridional radius of curvature "is the radius of a circle that is tangent to the ellipsoid at the latitude and has the same curvature as the ellipsoid in the north-south direction there." [5] The M is computed in the latitude coordinate φ_c of the small circle center.[5]

$$M = \frac{a \cdot (1 - e^2)}{\sqrt{(1 - e^2 \cdot \sin {\varphi_c}^2)^3}} \tag{3.18}$$

Where the term $1 - e^2$ is the ratio of the ellipse axis squared. For axis a and b see the Section 3.1.1.

$$1 - e^2 = \frac{b^2}{a^2} \tag{3.19}$$

5. The equation 3.18 refers to the formula *Radius of Curvature in Meridian* in the document *Radius of the Earth* [5].

With M computed I can evaluate the arc length of the small circle radius, α. For this is used following equation.

$$\alpha = \frac{R}{M} \tag{3.20}$$

Now I can create new point $P_s = \begin{bmatrix} \varphi_s \\ \lambda_s \end{bmatrix}$ where $\varphi_s = \alpha + \varphi_c$ and $\lambda_s = \lambda_c$. This point is then converted to geocentric coordinates in the Cartesian system and in these coordinates is stored as the second vector, V_s.

The instance of the `VectorSC` then provides function `getNextSegmentCoord()`. The value returned by the `getNextSegmentCoord()` is the geodetic coordinate of the next point on the small circle. This is the rotated vector in geocentric Cartesian system converted by a function from the *Geocent*[6] library into geodetic coordinates.

By calling the `getNextSegmentCoord()` function the vector V_s created from the P_s is rotated by a defined step angle around the first vector, V_c made from the P_c. This is done by creating the rotational matrix, RM, in the constructor of the `VectorSC`. This matrix is created by the OSG from the vector V_c and the defined step. The step defined in the class is 5 degrees. The rotation is then only multiplication of the vector and the rotational matrix. The rotation of the vector V_c is illustrated on the Figure 3.5. There are dashed lines from the earth origin to the small circle. These lines are the vector V_s rotated around the vector V_c.

In the constructor is created new vector, V_{tmp} and the vector V_s is assigned into it. This temporal vector represents actual state of the rotation. Every time the function `getNextSegmentCoord()` is called, there is only one multiplication of the V_{tmp}.

$$V_{tmp} = V_{tmp} \cdot RM \tag{3.21}$$

By doing this I always get a new vector which is rotated by the step angle of $5°$. So if the function `getNextSegmentCoord()` is called 72 times, there is created a complete $360°$small circle with defined parameters.

In the last call of the `getNextSegmentCoord()` there is closed the small circle shape. To do this properly and thus to eliminate errors of vector multiplications the last segment is replaced by the original vector V_s.

Creating of a Sector of the Small Circle

This class also allow user to create an circle arc or a sector of the small circle. In this case is not computed the meridional radius of curvature to get the second vector. Because the constructor of a sector of the small circle requires a initial and a final point of the

6. This component provides conversions between Geodetic coordinates (latitude, longitude in radians and height in meters) and Geocentric coordinates (X, Y, Z) in meters.

sector. The initial point is then converted to geocentric coordinates and form a vector V_s. This vector is rotated with the same function `getNextSegmentCoord()` as in the case of creation of the complete small circle.

The vector constructed from the initial point is rotated in a correct direction towards the final point. Because the arc center needn't to be inside the airspace the direction of the vector rotation is found out by following equations. The result of the first equation 3.22 is a vector V_n perpendicular to the vector V_i and V_f. It is a cross product of these vectors. In the second equation 3.23 is made the dot product of the vector V_n and V_c. The result I get is a which is the cosine of the angle between these two vectors. And if the a is smaller than zero the vector V_i need to be rotated in the negative direction. In the case the a is greater than the zero the vector V_i is rotated in the positive direction.

$$V_n = V_i \times V_f \tag{3.22}$$

$$a = V_c \cdot V_n \tag{3.23}$$

The segment of the small circle is used for example in the shape of the airspace in the *Generic shape airspace* example in the Section 2.3.1 (see the page 2.3.1).

Height Correction

During the creation of the small circle in the class `VectorSC`, the height of the small circle is not taken into account. The correction of the height is done in the module which uses this class. The method for the height correction is the same as in the classes used for the great circle (see the Section 3.3.2).

Chapter 4

3D Objects Visualization

This chapter is about the visualization of the proposed 3D objects. Only the airspace visualization is implemented because it features most of the properties of the other proposed 3D objects.

The planet is based on the OSSIM and OssimPlanet libraries and together with the OSG root node of the scene graph forms the module `_ossim`. All other modules with 3D objects depend on the `_ossim` module, that includes the root node of the scene graph. All other geometry nodes and groups are part of the scene graph which root is in the `_ossim` module.

The airspace module is based on the graphics functions provided by the OSG and on implementation of the great and small circle computing in the `utility` library (see the Section 3.3 for the great circle and the section 3.4 for the small circle).

4.1 Module _airspace

Visualization of an airspace, or a controlled area, is one of the most difficult. The shape of the airspace can be very various. The objective of this module is to allow user to define any type of the airspace with any possible shape.

This module uses OSG for creating and displaying models to the user. It also uses OSSIM and OSSIMPlanet for geodesy related functions and for working with elevation data. For access to the OSSIM related functions it need an existing instance of the OSSIMPlanet. Thus this module depends on the `_ossim` Python module.

4.1.1 Initialization

Before creating a new airspace the `_airspace` module need to be imported first. Then a new object of the airspace type can be created in the script. The syntax of the module creation is the following:

```
import _airspace
area = _airspace.airspace(planet)
```

The newly created object name is the `area`. Its type is the `airspace` which is part of the `_airspace` module. The parameter used in the constructor (here it is `planet`) is the pointer to the object of the type `sceneGraph`. This class is inside the `_ossim`

module. The parameter is used, because this module need to have access to the OSSIM-Planet.

In the Python object `airspace` constructor there is only check for existing parameter and there is call to the constructor of the C++ object `Airspace`. The constructor of this C++ object has one parameter of the type `SceneGraph` which is get from the `planet` pointer. During the initialization of this object is done only initialization of protected variables.

4.1.2 Functions of this Module

This Python module provides functionality to create and visualize the airspace. The Python interface and user input examples are described in the Section 2.3 focused on the visualization and interface of the airspace object on the page 13.

All described function with the *set* prefix are only assignment of their parameters to protected variables. The whole functionality is hidden inside the method `create()`.

Method create()

This Python method calls the C++ method `createAirspace` which does all the work. The airspace creation is divided into three parts. In the first part are processes the coordinates of the airspace shape entered by the user. The second part does the triangulation of airspace sides. The last part process the data for the label and this part creates the label geometry. These phase are described in the following sub-sections.

4.1.3 Processing Airspace Coordinates

The processing of the user entered coordinates is made at the beginning of the method `createAirspace`. These coordinates are stored in the variable `mBoundary`. Coordinates in the list are then read item by item by the iterator. Each coordinate entry is stored in the vector inside the list. The vector size could be two or three. By the vector size is decided the way of data processing.

If the vector with coordinates has two items, then values of the vector are a latitude and a longitude of one point inside the boundary. In the second case if the size of the vector is three, then this vector represents an arc center coordinate and its radius.

There is also one special case. When the list has only one item with the vector size three. Then the first two values of the vector are an small circle center coordinates and the last value is a radius of this circle. This way is entered a circular shape airspace described in the Section 2.3.1 in the *Circle airspace* example.

When an airspace with a generic shape is created then the procedure of reading coordinates is divided into two cycles based on the item size. Both algorithms are similar and the main difference is the function used for the processed segment.

When the segment is straight, this means that there are two consecutive points, there is used function for computing the great circle. The included functions are described in the Section 3.3.2 and 3.3.3. In the second case there is an arc segment. Then is used function for the small circle. This function is described in the Section 3.4.1.

For computing the points of the boundary segment is used a `for` cycle. In each step of this cycle there is called the method `getFinalCoordinates()` of the `GreatCircle` instance or respectively the method `getNextSegmentCoord()` of the `SmallCircle` instance. These methods return an coordinate of the point on the great respectively small circle. This point is then converted to geocentric cordinates.

Height Correction and Geodetic Coordinates Conversion

To properly display the point in the OSG is needed the conversion to the Cartesian coordinate system. To place the computed point on the earth is needed to correct the height. The height is requested to be above mean sea level (AMSL) or above ground level (AGL). The MSL is represented as a geoid (see Section 3.1.1) and the ground level is represented by the earth elevation model.

For conversion I need to know the offset of the sea level or the ground level from the referenced ellipsoid WGS-84 (see the Section 3.1.1). These offsets can be get from OSSIMPlanet functions. So I correct each point height by the local geoid or elevation offset from the ellipsoid and I get the height above the reference ellipsoid.

This height correction is done in the following function which returns ECEF coordinates of the given point `gpt`:

```
gpt2Ecef(gpt)
```
[1]

Inside the function is in the first part corrected the height and in the second part is the conversion. The final conversion to geocentric ECEF coordinate system (see the Section 3.2.2) is done by this function:

```
forward(llh, xyz)
```

This function is in the `_ossim` module and calls the adequate function from the OSSIM library. The first parameter contains the latitude, longitude and height above the ellipsoid stored in the vector. The second parameter contains the returned value. This value is the converted point in ECEF Cartesian coordinates.

4.1.4 Airspace Outline Creation

Outlines of the airspace are displayed for better recognition of the airspace shape. The outline is displaed for the upper and the lower boundary of the airspace.

The line representing the outline is made in the following function:

```
pushBoundPointsXYZ(gpt, upper, lower)
```

1. Types of the C++ functions aren't intentionally presented. Types can be read in the source code on the DVD.

This function will convert the given geodesic point `gpt` to geocentric coordinates (see previous sub-section 4.1.3) and this point is then added to the outline.

The conversion is done for two points at the same time. The first one is the `gpt` point form the parameter which has an altitude set to the lower one. The second one is a point with the same geodesic coordinate but with altitude set to the upper airspace altitude. The result of the conversion is then added to the corresponding airspace outline. The result of the first conversion is added to the lower outline and the result of the second conversion is added to the upper outline. The corresponding outline is also used as a constrain in the airspace cap creation (see the Section 4.1.5).

The final geometry model of the boundary is made by *OSG* functions in the method:
`createBoundGeode(points)`

4.1.5 Airspace Polygon Triangulation

The final airspace model is composed from triangle faces. These triangles form the airspace upper and bottom cap and airspace sides. The triangulation is divided to these phases. In the first phase is created the upper cap followed by creation of the lower cap. In the second phase are created sides of the airspace. These phases are made in the function:
`triangulate(upper, lower)`

This function creates new `osg:Group` object `triangulatedSurface` where are stored all airspace polygons.

Triangulation of Airspace Caps

The airspace cap is based on defined altitudes of the airspace. There is bottom and upper airspace altitude and both can be above mean sea level (AMSL) or above ground level (AGL).

In the first case when the cap has AMSL height then the cap is a polygon formed from the geoid shifted to the defined altitude. This is because of the shape of the earth and because the airspace altitude is defined as an AMSL height.

In the second case, when the altitude of the airspace is defined as an AGL height, then the cap should copy the earth elevation model shifted to the defined airspace height.

The only difference between these cases is the correction of the altitude of points inside the cap polygon. The height correction is made in the same way as correction of great and small circle points. This is done using the function `gpt2Ecef()` as was described in the sub-section *Height Correction and Geodetic Coordinates Conversion* on the page 38.

Triangulation of the upper and lower cap is made by one function. The difference

between creating upper or lower cap is in parameters of the following function:
`createBase(bounds,baseAltitude)`

In this function is created whole geometry of the required cap in two phases. In the first phase is created an auxiliary grid. In the second phase are all airspace points connected by the Delaunay triangulation function to form triangles onf the cap polygon.

Creation of the Auxiliary Grid To create a cap of the airspace there is used an auxiliary grid of points that are inside the airspace shape. The grid is created in the bounding rectangle of the airspace.[2] The visualization of the grid is on the Figure 4.1. Where the blue grid is the auxiliary grid and the green grid represents points inside the airspace.

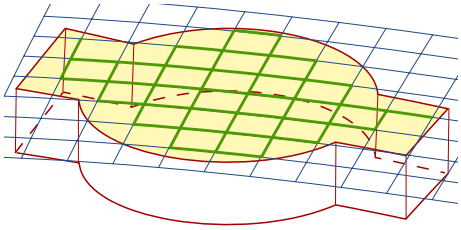


Figure 4.1: Illustration of the auxiliary grid in the airspace cap

Points inside the grid are regularly distributed and form squares. The distance between points in the grid is the same as the distance between great circle segments. This distance is computed using the equation 3.6 and it is around 12 km. For latitude this corresponds to the angular distance of approximately 0.1 degree. For longitude this distance is function of the latitude.

The longitude distance per one degree for the spherical earth (see the Section 3.1.2) can be computed by the equation 4.1. In the application is used the distance of 0.1 degree of the longitude and this can be computed by the equation 4.2. The longitude angular distance is $d\lambda$. The letter φ is the latitude.

$$d\lambda = \cos\varphi \tag{4.1}$$

$$d\lambda = 0.1 * \cos\varphi \tag{4.2}$$

The algorithm for create grid points of the airspace cap is in the function:
`getSphericalGridPoints(gridAltitude, bounds)`

Parameters of the function are requested altitude of the grid and airspace boundary points in the geocentric format. These points are the same as points used for the outline creation. Newly created grid points are added to the copy of the boundary points. The algorithm at the end returns these points back to the function `createBase()`.

2. This rectangle is made during the reading of the boundary points in the `create()` method. There are continuously updated maximal and minimal coordinates of the airspace.

Delaunay Triangulation When the creation of the grid points is finished, these grid points are returned from this function `getSphericalGridPoints()` back to the function `createBase()`. These points are assigned to the `DelaunayTriangulator` object from the OSG. This object provides functionality of the Delaunay triangulation.

This triangulation is used because the Delaunay triangulation creates triangles with good shape. The good shape is a result of the fact that no point of the airspace cap is inside the circumscribed circle of any triangle in the airspace cap. "One of the important uses of Delaunay triangulations is to find a triangulation of the convex hull of a set of points with the property that the triangles have as good a shape as possible. For example, long thin triangles are undesirable in many applications." [1, p. 724]

To the object `DelaunayTriangulator` are also assigned outline points computed during the processing of the list with boundary points. These points are assigned as the constrain for the Delaunay triangulation. This constrain is used for extracting only points which are inside the airspace.

The final result of the Delaunay triangulation is the airspace cap. This cap is returned as the geometry node back to the function `triangulate()`. There is assigned to the geometry group `triangulatedSurface`.

Triangulation of Airspace Sides

Airspace sides are created right after the caps. The creation of the airspace sides is done in a simple way. The entry of the function are two lists of Cartesian coordinates of the airspace boundary outline. The first list contains the upper boundary and the second one contains the lower boundary. These lists are then in one cycle merged together into one list. In this cycle is also computed the normal vector for each point. In this new list are points from the lower boundary at odd positions and points from the upper boundary are at even positions. When the points are in right order they can be triangulated using the OSG function:

```
DrawArrays(mode, first, count, numInstances=0)
```

The first parameter `mode` is set to the required triangulation mode - **TRIANGLE_STRIP**. This feature of the OSG is based in the OpenGL.

Now are all airspace polygons triangulated and hence the process of the triangulation is complete. To the geometry group `triangulatedSurface` where are all triangulated polygons is also assigned color of the airspace and transparency is set. The final geometry model group is assigned to the main geometry group of the airspace `mAirspace`.

4.1.6 Label Creation

The airspace can have an label with aeronautic data about it. The label is created if there are set the label data. This label is positioned above the airspace upper cap and

horizontally in the middle of the airspace bounding rectangle. For the label creation there is special class `AirspaceLabel`. This class takes care for the proper creating of the label. The label has dynamic width. The label width is based on the longest string inside the label.

The label is also scaled by the airspace size. The size is set by the parameter of the function `setScale(airspaceDiameter)`. The parameter value is the airspace dimeter length in decimal degrees. The bigger airspace has also bigger label.

And for better readability the label is always rotated to the user camera. This is done via the OSG object `Billboard`.

4.1.7 Airspace Visualization

The final 3D model of the airspace can be assigned to the scene graph. This is done by the Python function `airspace.addToPlanet()`. The result can look like on the Figure 4.2. There is displayed the part of the Czech Republic airspace around the airport Prague Ruzyně. The displayed scene contains four 3D airspace objects with labels. Other screens from application are in the Appendix on the page 7.4. The script for creation of this scene is on the DVD.

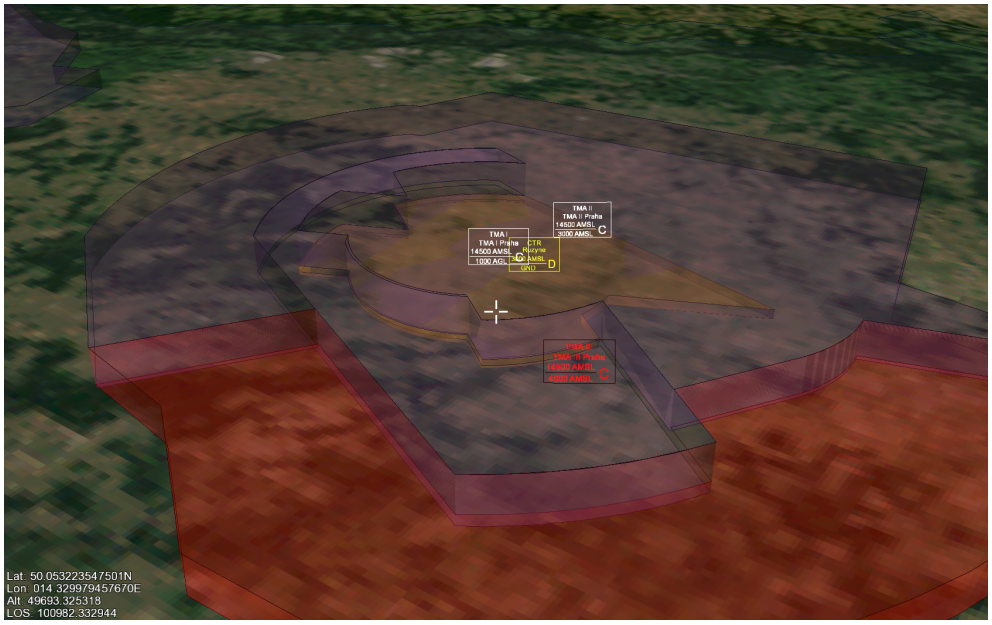


Figure 4.2: Screen shot from running _airspace module script

4.2 Module _ossim

The `_ossim` Python module is the core of the monitoring application. This module is build on the top of the OSSIM and OpenSceneGraph libraries. The OSSIM provides geodetic related function and planet model. The OSG provides scene graph and rendering of all objects in the scene.

4.2.1 Initialization of the Module

The initialization of this module is done in Python by creating new object `sceneGraph`. This is done by following Python code:

```
planet = _ossim.sceneGraph('D:/AeroVis_Data/config.xml')
```

The `planet` is the name of newly created object. The `_ossim` is the module name and the `sceneGraph` is the object type. The constructor of this object requires path to the configuration file as parameter.

This configuration file contains paths to OSSIMPlanet related files. It is in the XML format. There are currently three elements that can contain some value. Here is my configuration file:

```
<VirtualAir>
  <_ossim>
    <earth>
      <elevation>D:/AeroVis_Data/srtm30plus</elevation>
      <geoid>D:/AeroVis_Data/geoid/egm96.grd</geoid>
      <image>D:/AeroVis_Data/image/bmngnewJPEG75.tif</image>
    </earth>
  </_ossim>
</VirtualAir>
```

The element elevation is a path to directory containing files with elevation data for the OSSIMPlanet. This element is optional. If this element isn't present in the configuration file, the _ossim module can't provide elevation data to other modules and the terrain isn't rendered. The geoid element contains path to the geoid file, which is used in the OSSIMPlanet. The geoid is used for computing with the mean sea level.[3]

The element image is the path to the texture of the earth. This image is used only as an image on the earth surface.[4]

In the constructor of the `sceneGraph` inside the module there is called the constructor of the C++ class `SceneGraph`. In the class `sceneGraph` is checked the XML

3. Elevation and geoid can be downloaded from OSSIM server. Available at: <http://ossim.telascience.org/ossimdata/Data/elevation/>

4. Earth texture is available at OSSIM server. There are two world wide textures - BluMarbEarth.tar.gz and earthlowres.tgz with different resolution. Both are available at: <http://ossim.telascience.org/ossimdata/Data/>

configuration file. After this the camera setup is called. This procedure configure the `osgViewer` instance called `mViewer` and assign the camera manipulator to the viewer. There is used manipulator from the OSSIMPlanet. The manipulator used in the visualization is the `ossimPlanetManipulator`. This allows user to rotate the earth, zoom the view and tilt the camera to desired angles.

After setting the view are called constructors of `Space` and `EarthPlanet`. The `EarthPlanet` class is the one using the OSSIMPlanet API and functions. The constructor initialize the `ossimPlanet` by calling `ossimPlanet_init()`. There are also read values from elements of the configuration XML file. For reading XML documents is used class `XpathReaders` contained in the `utility` library. Then there are several calls to API of the OSSIMPlanet. It assign terrain, geoid and texture to the planet if these data were contained in the XML configuration. There is also light setup and creation of the HUD[5]. On the HUD is displayed latitude and longitude in decimal degrees. And there is also altitude and line of sight of the camera.

The node of the planet is then added to the root node of the scene graph (see the scene graph schematic view on the Figure 2.1). After this is the `seceneGraph` object initialized and ready to display 3D scene or for adding new objects to the planet.

4.2.2 Functions of this Module

This module holds the scene graph and the OSSIMPlanet. The Python interface of this module provides some methods related to the OpenSceneGraph. The functions related to the OSSIMPlanet are exposed only via C++ modules. Thus each other module which will be using functions of the OSSIMPlanet needs a pointer to the planet. In my module it is done via its constructor parameter.

In the Python module are only two methods of the scene graph. These methods allow to display the whole scene to the user.

renderFrame()

This method displays one frame to the user. The scene is rendered inside the full screen window of the `osgViewer`. For the rendering of the scene is internally called the method `frame` of the `osgViewer` object. This method render a complete new frame.

To render the scene simultaneously this function must be called in cycle inside the Python script. For example to render 1000 frames the script could contain following code:

```
for i in range(0,1000):
    planet.renderFrame()
```

5. Head-Up Display

renderDone()

The method `renderDone` returns boolean value representing the state of the rendering. In the `osgViewer` is internal flag `done`. This flag is set to true if the rendering is finished. The rendering is finished by pressing the escape key. If this flag was been set to true the render will stop and will not render any other frame.

This method can be used to render infinite number of frames until escape key is pressed. In the Python script this can be done by using the following code:

```
while (not planet.renderDone()):
    planet.renderFrame()
```

4.2.3 Planet Visualization

The visual output of this module is the earth model in the space. This model is displayed in a full screen window. The view can be controlled by the three button mouse with a scroll wheel. The screen with the earth and HUD looks like as on the Figure 4.3 which is taken from the running script with the planet model only.

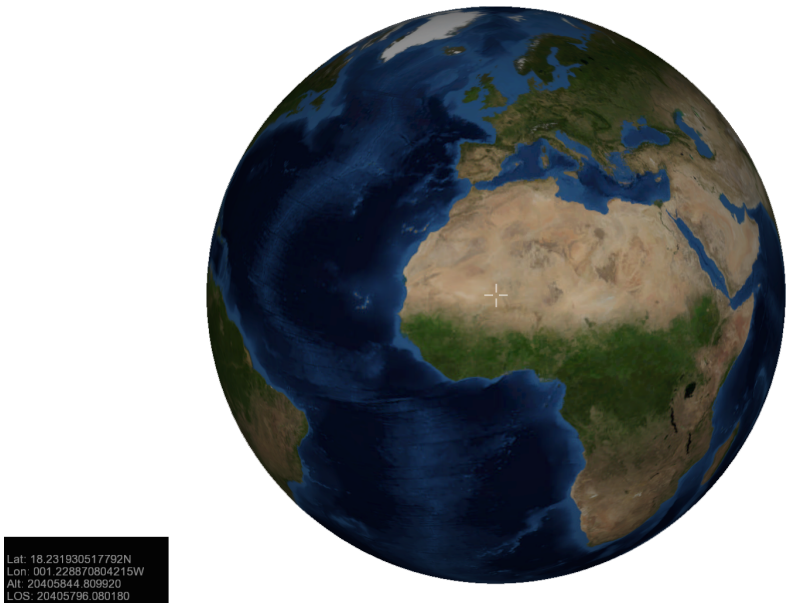


Figure 4.3: Screen shot from the running _ossim module script with removed black background

Chapter 5

Conclusion

The visualization of aeronautical data has been very interesting project. In this thesis I have done the analysis of aeronautic data which are on the aeronautics chart. As a result of this analysis I proposed in the Chapter 2 six aeronautical 3D objects for the visualization. These aeronautical objects are the most important for the visualization of situations in the airspace.

After the analysis I have begun with the implementation of these proposed 3D objects. The first problem that I found, was the representation of the Earth. This is provided by the OSSIM and OSSIMPlanet libraries. Both these libraries provide a lot of geodesic functions. So, in the Chapter 3 I introduced theory about the shape and mathematics representation of the Earth. The work with the Earth model was the most difficult one. The Chapter 3 also describes the visualization of great and small circles. These geodetic arcs are base elements of many 3D objects. Works on these base shapes were longer than I expected so only the airspace visualization has been practicaly implemebted.

Based on the introduced theory, the Chapter 4 describes the visualization of the airspace object. The airspace has been chosen because there is used application of the great and small circle. In this chapter is also described the height conversion. The airspace visualization also required to create spherical polygons. These polygons form the upper and bottom cap of the airspace.

The airspace visualization was success fully implemented to the usable stage. The application is in the stage of the proof of concept but the development still continues. Screens captured from the actual airspace visualization are in the Section 4.1.7 and in the Appendix.

The current version of the project is available on the SourceForge as the *Simulation Monitor* part of the *virtualair* project [18].

This project still has many features that are waiting for the realization. There is a room for new aeronautical objects that can be added to the visualization via the implemented `_ossim` module. This future work is a part of a proposed student project.

Chapter 6

Bibliography

[1] AGOSTON, Max K. *Computer graphics and geometric modeling : implementation and algorithms*. Springer, 2005. ISBN 1852338180.

[2] *awesome image Processing* [Online]. c2009. [cite 2009-05-17]. about ossim. Availableonline at: <http://www.ossim.org/OSSIM/About_OSSIM.html>.

[3] BARANOVÁ, Magdaléna. *Multimediální texty Matematické kartografie* [Online]. Plzeň : ZČU - Fakulta aplikovaných věd, c2009. [cite 2009-05-08]. [Chapter] 2. Základní pojmy. Available online at: <http://gis.zcu.cz/studium/mk2/multimedialni_texty/index_soubory/index_soubory/hlavni_soubory/zaklady.html>.

[4] BARHYDT, Richard and WARREN, Anthony W. *Development of Intent Information Changes to Revised Minimum Aviation System Performance Standards for Automatic Dependent Surveillance Broadcast (RTCA/DO-242A)*. Hampton : NASA Langley Research Center, May 2002 [cite 2008-11-06]. [Chapter] 3, Short and Long-term Intent. Available online at: <http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20020060738_2002099744.pdf>.

[5] CLYNCH, James R. *Radius of the Earth : Radii Used in Geodesy*. February 2006. Available online at: <http://clynchg3c.com/Technote/geodesy/radiigeo.pdf>.

[6] HARRISON, Mike. *Intent or How Do We Get To Trajectory-Based Air Traffic Control and Management?*. May 2003. Available online at: <http://acast.grc.nasa.gov/wp-content/uploads/icns/2003/05_B2/B2-04-Harrison.pdf>.

[7] International Civil Aviation Organization (ICAO). *Rules of the Air : Annex 2 to the Convention on International Civil Aviation*. July 2005. 10th Edition.

[8] International Civil Aviation Organization (ICAO). *Aeronautical Charts : Annex 4 to the Convention on International Civil Aviation*. July 2001. 10th Edition.

[9] International Civil Aviation Organization (ICAO). *Air Traffic Services : Annex 11 to the Convention on International Civil Aviation*. July 2001. 13th Edition.

[10] International Civil Aviation Organization (ICAO). *Doc 9750 : Global Air Navigation Plan* [Online]. 2007. 3rd Edition. Aviable online at: <http://www.icao.int/icaonet/dcs/9750/9750_cons_en.pdf>.

[11] MARTZ, Paul. *OpenSceneGraph Quick Start Guide : A Quick Introduction to the Cross-Platform Open Source Scene Graph API*. Louisville (CO) : Skew Matrix Software LLC, 2007 [cite 2009-05-04]. Available online at: <http://www.lulu.com/content/paperback-book/openscenegraph-quick-start-guide/767629>.

[12] OSFIELD, Robert and BURNS, Don. *Open Scene Graph* [Online]. last revision 18th February 2009 [cite 2009-05-05]. Available online at: <http://www.openscenegraph.org>.

[13] *Open Scene Graph* [Computer software]. Version 2.9.2. March 2009 [cite 2009-03-29]. Available online at: <http://www.openscenegraph.org/svn/osg/OpenSceneGraph/tags/OpenSceneGraph-2.9.2>.

[14] *Open Source Software Image Map (OSSIM)* [Computer software]. Version 1.7.15. January 2009 [cite 2009-02-07]. Available online at: <http://svn.osgeo.org/ossim/tags/v1_7_15>.

[15] *Python* [Computer software]. Version 2.6.2. April 2009 [cite 2009-04-20]. Available online at: <http://python.org/download/releases/2.6.2/>.

[16] *Python v2.6.2 documentation* [Online]. last revision 5th May 2009 [cite 2009-05-05]. Available online at: <http://docs.python.org/>.

[17] VINCENTY, Thaddeus. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review*. Directorate of Overseas Surveys, April 1975, vol. XXIII [cite 2009-04-16]. Aviable online at: <http://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf>.

[18] GOTTHARD, Petr. *Virtual Air* [Computer software]. last revision 12th May 2009 [cite 2009-05-13]. Available online at: <http://sourceforge.net/projects/virtualair>.

[19] WEBER, Rosa. Airborne Separation in Advanced En-Route ATM. April 2008. Aviable online at: <http://ifly.nlr.nl/documents/iFly%20briefing%20to%20AP23%20in%20April%202008.pdf>

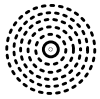
[20] WILLIAMS, Ed. *Aviation Formulary* [Online]. Version 1.44. c1995, Updated November 11, 2008 [cite 2009-05-08]. Available online at: <http://williams.best.vwh.net/avform.htm>.

[21] YEH, Michelle and CHANDRA, Divya C. *Designing and Evaluating Symbols for Electronic Displays of Navigation Information : Symbol-stereotypes and Symbol-feature Rules*. Cambridge (MA) : John A. Volpe National Transportation Systems Center, September 2005 [cite 2008-10-28]. Available online at: <http://www.ntis.gov/search/product.aspx?ABBR=PB2005110722>.

Chapter 7

Appendix

7.1 Navigation aids types

Navigation Type	navType parameter string	Displayed image
Default	empty string	
Distance measuring equipment	dme	
Reporting point	fix	
Non-directional radio beacon	ndb	
UHF tactical air navigation aid	tacan	
VHF omnidirectional radio range	vor	
Collocated VOR and DME radio navigation aids	vordme	
Collocated VOR and TACAN radio navigation aids	vortac	
Waypoint	wpt	

7.2 WGS-84 parameters

WGS-84 parameters	
Semi-major axis a	$6378137.0m$
Semi-minor axis b	$\approx 6356752.314245m$
Inverse flattening $\frac{1}{f}$	298.257223563

7.3 DVD content

The DVD contains following data:

application	This folder contains the empty configuratin XML file - config.xml.
... binaries_W32	There are compiled libraries for the Python – _airspace.pyd and _ossim.pyd and the utility.dll library. Compiled for Windows 32bit.
... source	This folder contains source files of my application
data	There are folders with data for the OSSIMPlanet. These data can be downloaded from . Path to files on the DVD needs to be in the config.xml.
... elevation	Folder with elevation data.
... geoid	Folder with the EGM-96 geoid.
... high-res image	High resolution Earth texture
... low-res image	Small resolution Earth texture.
osg	The OpenSceneGraph 2.9.2 source files and precompiled dependencies for the MS Visual Studio 2008.
ossim	The OSSIM 1.7.15 source files and precompiled dependencies for the MS Visual Studio 2008.
python	The Python 2.6.2 source files and precompiled dependencies for the MS Visual Studio 2008.
scripts	Example Python scripts for the module `_airspace`.
text	The electronic version of this document in the pdf.
... latex	Source files of this document in the LaTeX format with folder containing images.

7.4 Screens taken from application

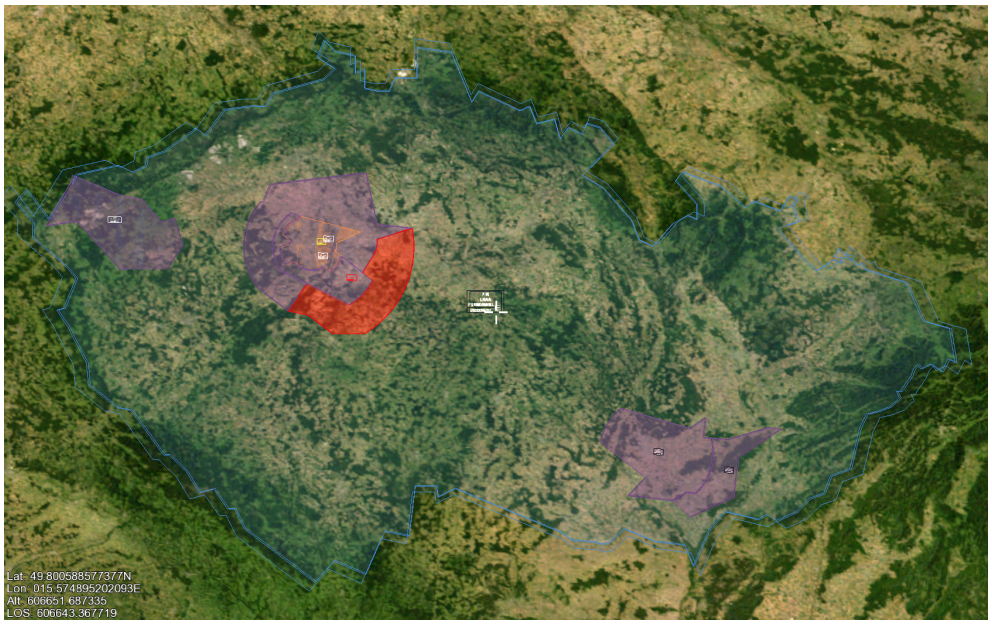


Figure 7.1: Czech Republic

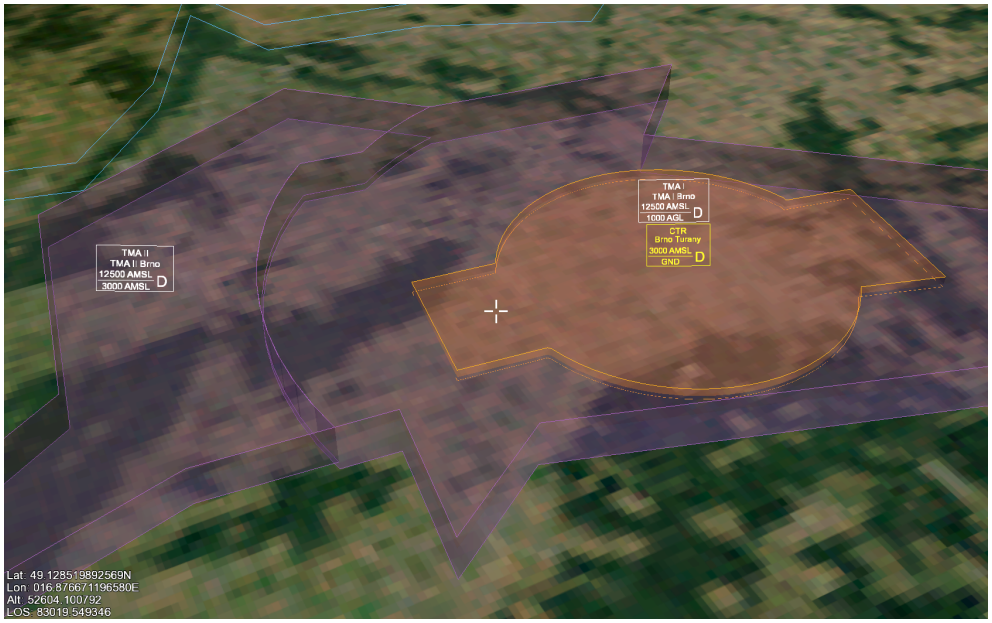


Figure 7.2: Airspace around Brno Tuřany