Visualizing the Air Traffic in 3D

Per Holmqvist

Examensarbete
Master Thesis

Ergonomics och Aerosol Technology
Department of Design Sciences
Lund Institute of Technology
Lund University

ISRN
LUTMDN/TMAT-5056-SE

Abstract

The aim of this thesis was to design and implement a program for visualizing the air traffic within Malmö air space (called Malmö FIR) in 3D, which can be used in the education for both students and experienced air traffic controllers when they are learning a new air space sector. It's based on a previous pilot study and a thesis with an implementation, which was dependent on a high level graphics API (Application Programming Interface). The task was to develop a new implementation, which would be independent of an expensive high level graphics API and with more graphics and functions added. Adds like a height map over the southern part of Sweden, beacons on the ground, reporting points, animation of aircraft routes, some kind of standard steering etc. The project is developed with Visual C++ 6.0 (Microsoft's developing platform for C++), and uses DirectX 8.1 (Microsoft's API to access the hardware directly in Windows), and MFC (Microsoft Foundation Classes for windows programming). With this thesis I show an improved program that is closer the goal than the previous ones and I have done it in a high level API independent way with reusable code for continuing thesis in the future.

Preface

I would like to thank my supervisor Joakim Eriksson at the Department of Design Sciences for leading me through this project and for always being available when I needed help. Also I would like to thank Michael Bartholdson at ATCC Malmö for showing what to include in the program and for giving me the latest updates about Malmö air space. Finally I would like to thank Miro Cepciansky and Victoria Welbert for their study about air traffic control and their program design.

Index

1 Introduction

This thesis was developed under supervision by the Air Traffic Control Centre (ATCC) Malmö, and the Department of Design Sciences at Lund University.

1.1 The use of visualizing the air space in 3D

Today the way of teaching air traffic controllers a new air space sectors is based on a two dimensional representation of the air space. A 2D radar screen visualizes the air space sectors, the coastlines, the aircrafts, beacons, and reporting points etc. To increase the air traffic controller's ability to learn the geometry of the air space would be to visualize it in 3D. So the problem to be solved is to design and implement a program for visualizing the air traffic in 3D, which can be used in the education for both students and experienced air traffic controllers when they are learning a new air space sector.

1.2 Previous approaches

Two previous studies have been made in this area at the department of design and sciences:

- A pilot study by Anders Trollås and Claes Kjällkvist. They made a 3D scenario containing air space sectors and animations of aircraft routes in 3D Studio Max. This kind of visualization is limited and can only be updated by a person who has good acknowledge about 3D Studio Max. And even though it would be very limited and hard to manage.
- A master thesis "Using 3D models for visualizing air space sectors", by Victoria Welbert and Miro Cepciansky [1]. Their work consists mainly of three parts. The first part is a study of how air traffic controllers controls the flow of aircrafts in the air space and how students learn to become an air traffic controller. The second part is a design of a program to be used in the air traffic controller's education. The third part is an implementation in 3D. The program only visualizes combinations of air space sectors from different views. There is no 3D map of the ground, no beacons, no reporting points, no routes, and no animation of aircrafts etc. Their program uses two high level APIs called World Up, and World Toolkit and cannot be executed without them. And traditionally the license is very expensive.

1.3 Objectives

Welbert and Cepciansky [1] suggested some further work in their thesis. They suggested to including "reference marks" (the same as "way points" which will be explained in section 2.2), animation of aircrafts flying through the sectors (sectors will be explained in section 2.1), and aircraft paths (a better name will be "aircraft routes" and it will be explained in section 2.2). These suggestions are taken into account in this thesis. One thing they didn't suggested was to include a height map over the southern part of Sweden in the 3D scene, which I will do. Additionally is to build an implementation that is independent of an expensive high level graphics API (Application Programming Interface). So my way to solve the problem is to design and implement a second prototype for visualizing the air traffic in 3D, which can be used in the education for both students, and experienced air traffic controllers when they are learning a new air space sector. This will be done independent of an expensive high level graphics API and with more graphics and functions added than the

previous thesis. Due to the excluding of the high level graphics API the code will be totally rewritten. The code will be separated in modules, and will be object oriented, which will make it reusable for continuing studies in the future. The name of the program will be Air Traffic Viewer (ATV).

1.4 The report layout

In chapter 2 I will give an introduction to air traffic control by talking about the air traffic control profession and explain terms they are using. In the end of the chapter I will talk about the Traffic Control Centre (ATCC) Malmö, and the Swedish Air Traffic Service Academy (SATSA). In chapter 3 I will talk about graphics APIs (Application Programming Interfaces), and give an overview over DirectX 8.1 (Microsoft's API to access the hardware directly in Windows). In chapter 4 I will talk about windows programming, and give an overview over MFC (Microsoft Foundation Classes for windows programming). In chapter 5 I will present the program design and implementation which will contain: the choice of APIs, the hierarchical structure of the program, an explanation of what frames are, descriptions of the included 3D parts, and in the end sections something about the mouse input and the camera. In chapter 6 I will walk through the user interface's functions one by one. In chapter 7 I will give my final comments of the report. Have I achieved the goal of the thesis? What could have been done better in the thesis? And in the last section something about further work for thesis in the future. At the end of report are the references and an appendix.

2 Air traffic control

The material in this chapter is collected partly from the supervisor at ATCC, partly from Welbert and Cepciansky thesis report [1], and some from LFVs (LuftFartsVerket's) homepage [a].

2.1 Sectors

An air traffic controller's task is to direct and co-ordinate aircrafts in the airspace. Some controllers regulate airport traffic and some regulate traffic between airports. Their work follows international standards, which is the same all over the world. The air space is divided into air space sectors. An air space sector is a three dimensional volume with different vertical heights, which can be subdivided in smaller less complicated parts. Each sector part is a vertical stretched polygon with an upper and a lower height. Figure 1 shows a sector and its sector parts.

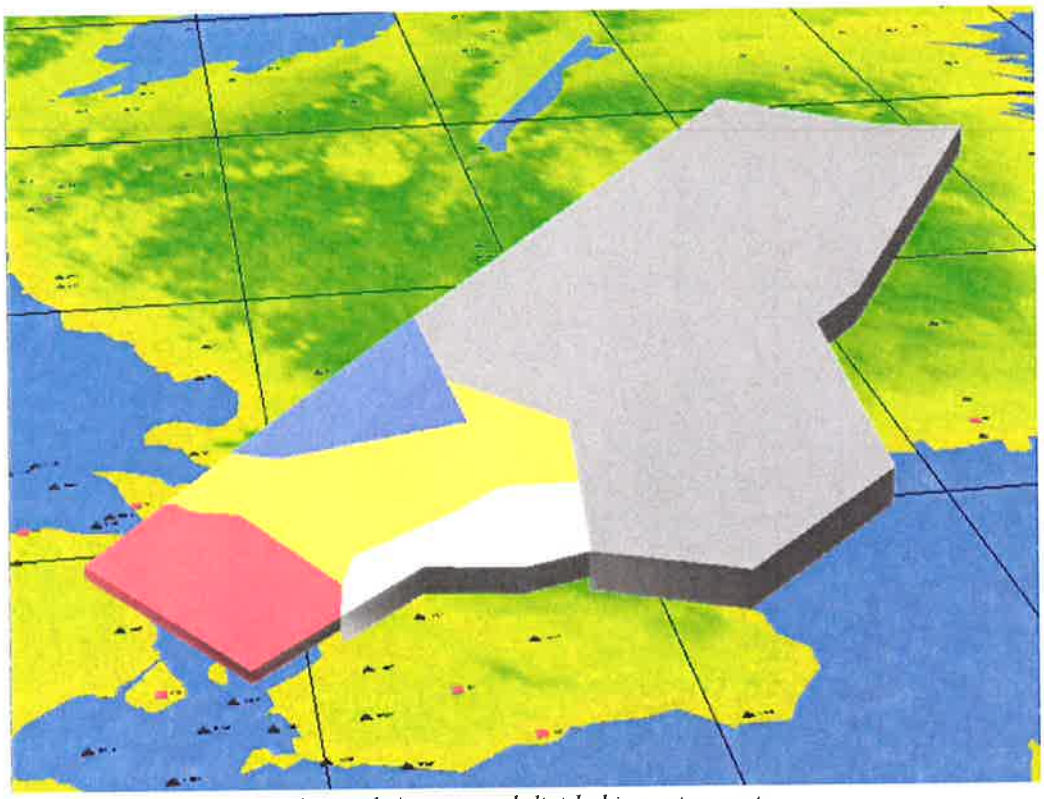

Figure 1 A sector subdivided in sector parts.

Normally an air traffic controller is responsible for one air space sector and must have a good knowledge how it looks like in 3D. They must also have a good knowledge how the adjacent sectors look like. The air traffic controller's main task is to make the aircraft's routes as safe as possible by avoiding collisions with other aircrafts and the ground. They are mainly dealing with delays and are avoiding conflicts by changing flight levels. Usually an air traffic

controller co-operate with other air traffic controllers of adjacent air space sectors. At the end it's always the air traffic controller who decides the aircraft's main direction, not the pilot.

2.2 Waypoints, routes, and flight levels

The aircrafts navigates with help of beacons on the ground, and non-physical reporting points. A commonly name for these points is "way points". Between the way points stretches routes. One route goes between two or more way points and each route part has its own flight level range. The air space that the airline traffic goes through is subdivided in two ranges, the lower, and the upper flight level. The lower flight level ranges from 9500 to 24500 feet (approximately 2900 to 7500 m), and the upper flight level ranges from 24500 to 46000 feet (approximately 7500 to 14000 m). Levels under 9500 feet and above 46000 feet are called the free air space and can be used by others than the ordinary airline traffic. There are many standard routes and the map in figure 2 shows the routes and how they stretch between the way points in the airspace above the southern part of Sweden.

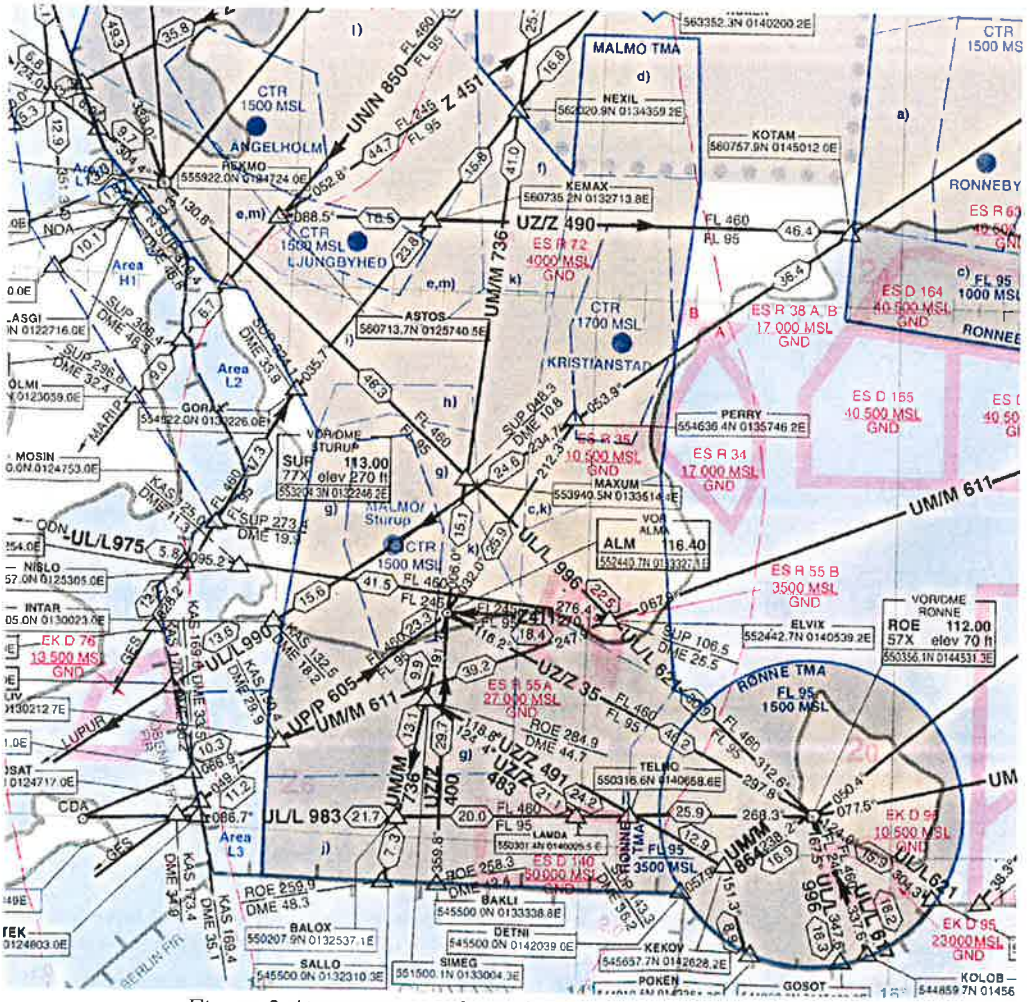

Figure 2 Airspace map above the southern part of Sweden.

The routes are marked with their name/names. For instance the name UM/M 996 says that there are two routes, one in the upper UM 996, and one in the lower M 996. Next to each route part is the flight level range between its two way points written. The beacons is marked by black squares with white circles on them, the reporting points is marked by triangles.

2.3 Radar and communication

Today the air traffic controllers use a 2D radar screen to control the air traffic. The screen visualizes the aircraft traffic within the sector the controller is responsible for and the traffic from the adjacent sectors. It also shows way points, coastlines, standard routes etc. The controller uses a speed vector line to figure out where an aircraft will be in a specified number of minutes. This is helpful to determine if crossing traffic will lead to conflict. In the future satellite navigation technology will have a great impact on the navigation.

Communication between the pilots and the air traffic controller is by radio. The air traffic controller instructs the pilot during takeoff, flight, and landing. Each sector has its own radio frequency. So when a pilot passes from one sector to another the frequency is switched to another air traffic controller. Weather conditions such as wind velocity, and direction, visibility, snow, rain etc. is also reported to the pilot via radio.

2.4 Air Traffic Control Centre (ATCC) and Swedish Air Traffic Service Academy (SATSA)

ATCC Malmö is the control centre for the air space traffic in the southern and western parts of Sweden, and the whole of the upper airspace over the Baltic up to the Finnish FIR (Flight Information Region). The area is called Malmö FIR and is divided in 16 sectors. ATCC is situated at the Malmö-Sturup Airport. Today there are about 150 air traffic controllers working there. Within a couple of years, the ATCC Malmö will also have the responsibility of servicing the whole of the higher airspace over Sweden. Today the number of aircrafts being controlled from ATCC Malmö is about 1700 each day.

There's only one school for air traffic controllers in Sweden. It's called SATSA and is situated at the Malmö-Sturup Airport, next to ATCC Malmö. SATSA is responsible for the training of the Swedish air traffic control personnel. The academy also educates international students. Over the years more than 2000 students from 40 different countries have received their training at SATSA.

3 Graphics programming

3.1 Graphics APIs

This section will give three short overviews of possible graphics APIs to be used when working with ordinary PCs and UNIX machines. Depending of what you're about to do they are all good, but the last two are the most commonly used and therefore most of the time the ones to prefer. The alternatives and their specifics are as follows:

- A combination of World Up, World Toolkit [b], and C++. This is the way the previous thesis program by Welbert and Cepciansky [1] was developed. World Up is high level API that can combine several technologies such as World Toolkit and C/C++ into an integrated, object oriented environment. The user can create visual simulations in which graphical objects have real world properties and behaviours. World Up also provides a graphical user interface (GUI). To add behaviours to objects, there's a possibility to write scipts using the "Basic Script" language or to use property change events to trigger behaviours. The "Basic Script" language is syntactically similar to Visual Basic. World Toolkit can be described as a text version of World Up. It's an API that basically is a library of C functions. World Up and World Toolkit can be combined or used separately. The disadvantages of World Up and World Toolkit are that they are expensive and applications created with them can't be executed without them, you don't have full freedom to create what ever you want and you're bound to the speed of the programs universal implementation.
- OpenGL is a platform independent 2D and 3D graphics API. It's the most widely used and supported graphics API for computers. It's been around for a while (since 1992) and has many advantages such as stability, portability, easy to use, and it is well documented.
- Direct3D the 3D graphics part of DirectX (Microsoft's API to access the hardware directly in Windows) is fairly similar to OpenGL except that it isn't platform independent. I haven't got any experience from using OpenGL but I think OpenGL is to prefer when working with UNIX machines, and DirectX is to prefer when working with PCs. At least all the game developers think so.

Next section will give a more detailed description of DirectX and Direct3D, the alternative that was chosen in this project.

3.2 DirectX version 8.1

The material in this section is principally collected from "The Zen Of Direct3D Game Programming" by Peter Walsh [2], and Microsoft's DirectX 8.1 SDK documentation for C++ [3].

3.2.1 DirectX 8.1 components

DirectX is an API developed by Microsoft that allows programmers to access the hardware directly in Windows. It consists of six different APIs:

- DirectX Graphics (which includes Direct3D and the D3DX library) for two-dimensional and three-dimensional graphics.
- DirectInput for input devices such as keyboard, mouse etc.

- DirectPlay for networked applications.
- DirectX Audio (which includes DirectSound and DirectMusic) for sound effects and music.
- DirectShow for streaming medias such audio and video.
- DirectSetup for one call installation of the DirectX components.

The APIs used in this project are DirectX Graphics and DirectInput.

3.2.2 Interfaces

There are twelve Direct3D interfaces and the inheritance structure can be seen in figure 3.

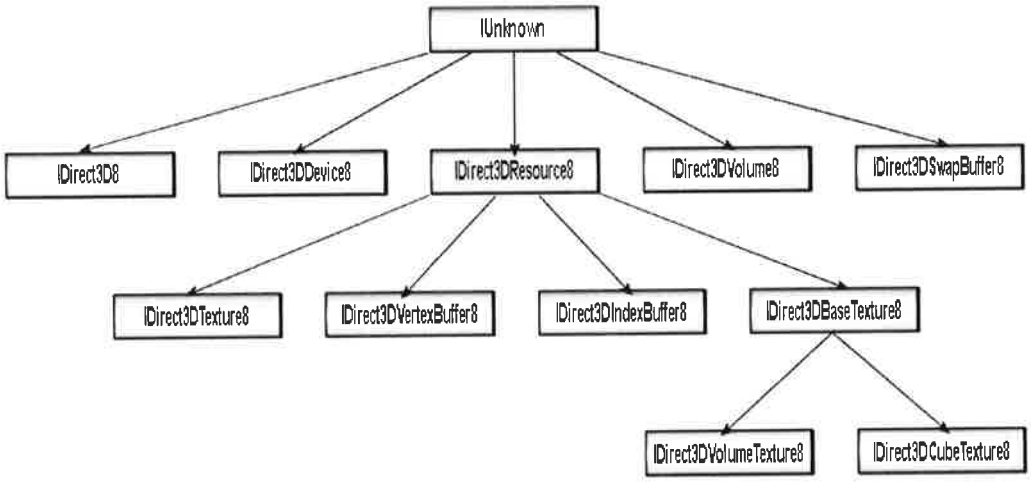

Figure 3 Direct3D interfaces.

To access the interfaces in C++ the following file must be included:

```
#include <d3dx8.h>
```

Then the interfaces are accessed by what is called COM objects (Component Object Model objects). COM objects are used as if the were normal C++ objects. They are created by calling functions. For instance to get the IDirect3D8 interface you write the following C++ code:

```
LPDIRECT3D8 m_pD3D = 0;
m_pD3D = Direct3DCreate8( D3D_SDK_VERSION );
```

The Direct 3D interfaces used in this project are:

- IDIRECT3D8, the main interface to all the Direct3D functionality, by the IDIRECT3D8 object you have access to the IDIRECT3DDEVICE8
- IDIRECT3DDEVICE8, interface for creating what is called the device, this is the direct link to the graphical hardware and all rendering will take place using the device, you also get access to IDIRECT3DVERTEXBUFFER8, IDIRECT3DINDEXBUFFER8 and IDIRECT3DTEXTURE8
- IDIRECT3DVERTEXBUFFER8, interface for creating vertex buffers, a vertex buffer is a group of face vertices.

- IDIRECT3DINDEXBUFFER8, interface for creating index buffers, an index buffer is a group of integer indices that point into the vertex buffer.
- IDIRECT3DTEXTURE8, interface for creating textures

Besides the Direct3D interfaces the DirectInput interface is used. This interface is for input devices and in the project it's used for the mouse. To access the interface in C++ the following file must be included:

```
#include <dinput.h>
```

3.2.3 The transformation pipeline

There are three steps to get the 3D objects from their local coordinates to be visualized on the screen. By local coordinates means that an object is in its own coordinate system. The steps are as follows:

- Local coordinates to world coordinates. This step put all objects to a common coordinate system. It's done by adding the world offset to all objects.
- World coordinates to camera coordinates. The camera is looking down its own positive z-axis and when it starts to move the angels relative to the z-axis start to change. What to do now at this point is to rotate all the objects inversely back to where they were before the movement.
- Camera coordinates to projection coordinates. Transform the objects to have the right perspective.

After the creation of the device it's time to set up the viewport (the viewing volume) and then the projection matrix. Usually the projection matrix is only set up once and that is good because it's the most complex transform in the transformation pipeline. Under the execution of a program the current world matrix is updated every time an object rotates or is changing its position and is about to be rendered. The view matrix is updated every time the camera rotates or is changing its position.

3.2.4 Vertex buffers and index buffers

For optimizing the flow of vertices between the program and the graphical hardware, DirectX uses vertex and index buffers. A vertex buffers is a group of many face vertices, which will be sent to hardware at one call. This will be way quicker than doing them one by one. Each element in the vertex buffer represents a face vertex and contains the position (x, y, z), ambient color, diffuse color, specular color, normal, texture coordinates etc. An even faster method is to combine the vertex buffer with an index buffer. An index buffer is a group of integer indices that point into the vertex buffer. This method is efficient when the faces use the same vertices more than one time. The result will be that you save memory and processing power.

3.2.5 Basic rendering

I will here walk through how to do some basic rendering in Direct3D of an object that changes its position. It will be done with no code and be using an index buffer. At first the following must be done:

- Create an index buffer.

- Lock the index buffer.
- Fill the index buffer with integer indices that point into the vertex buffer.
- Unlock the index buffer.
- Create a vertex buffer.
- Lock the vertex buffer.
- Fill the vertex buffer with the objects vertices.
- Unlock the vertex buffer.

Then every time the object changes its position do as follows:

- Call the device to clear the back buffer. The back buffer is the memory area where the creation of the image first takes place before it's displayed upon the screen.
- Tell the device we are about to start rendering.
- Connect the vertex buffer to a stream.
- Tell Direct3D about the index buffer.
- Update the material.
- Render the stream of vertices using the indices in the index buffer.
- Tell the device we have finished rendering.
- Present the back buffer to the primary surface.

See appendix A for a Direct3D code snippet with the basic rendering procedure.

3.2.6 Materials, textures, and lights

Materials and textures work like state machines. That is you choose one material/texture and render the affected parts then you choose another material/texture and render next parts and so on. Remember to always set a material even when you have set a texture, otherwise the rendering parts will not be shown. Lights work similar to objects and are just enabled or disabled.

4 Windows programming

The material in this section is collected from "Using Visual C++ 6" by Kate Gregory [4], Microsoft's Visual Studio 6.0 Documentation for C++ [5], and Microsoft's homepage for developers [c].

4.1 Windows GUIs

When working with DirectX and C++ under Windows there are two standard ways to program a GUI:

- The fist way is to use the "raw" Windows API (a common name of the WIN16, WIN32, and WIN64 APIs). It allows applications to exploit the Windows family of operating systems, including Windows XP/2000/NT/Me/98/95. Using this API, you can develop applications that run on all the versions while you are taking advantage of the specific features of each version. You can provide your application with a GUI and manage system objects such as memory, files, and processes. With the GUI you can create and use windows to display outputs, handle user inputs, and to take care of other tasks concerning the interaction with the user.
- The second way is to use MFC, which basically is a tool that helps the programmer to generate the Windows API code in an easier way.

In this project MFC is used and in the next section is a short overview of what it is and what it generates.

4.2 Microsoft Foundation Classes

MFC is an "application framework" for windows programming. That is, MFC provides much of the code for managing the Windows API. It makes it easier to set up windows, menus, dialog boxes, input output storing etc. It will shorten the developing time and is good if you don't want to waste too much time on windows programming. The disadvantage is that it makes the application slower due to all the extra code added. If you're in to making a fast 3D game don't choose MFC.

By using what is called the "AppWizard" you create a skeleton program. This program will include the basic features of a windows executable program. Most of the classes created by MFC will inherit from the base class CObject. The classes you get with AppWizard are as follows:

- One class that derives from CWinApp. Normally when you're writing a windows application the WinMain() is the main function that windows calls at the start of an application. In MFC the class that derives from CWinApp is treated as the main function and sets up the application. The truth is, buried inside CWinApp is the WinMain() function.
- One class that derives from CDocument. This is the base class used by any documents that your application creates. Documents represent data and they know how to load and save themselves to files, how to update themselves and how to talk to views. For instance in a text editor is the text the document.

- One class that derives from CView. This is the base view class and its function is to show the contents of documents and graphical data in a window. For instance in a text editor the view shows the text in a window. Usually this class also takes care of the user inputs from keyboard and the mouse.
- One class that derives from CFrameWnd. This is the applications main window and is the window that most people associate with an application. It includes the main caption, the system menu, the application menu, and underlying dialog bars, tool bars, status bars etc.

You also get a resource directory that holds your applications resources. A new project includes some standard resources and new resources can be created and modified by what is called the "AppStudio". Resources are parts of the graphical user interface such as cursors, icons, menus, dialogs, toolbars etc. You can draw cursors and icons and set up the layout of the menus, dialogs, and toolbars with predefined tools.

5 Design and implementation

When programming 3D graphics it's very important to have good knowledge about linear algebra, so you can deal with mathematical problems concerning vectors and planes. During the project I've used "Linjär algebra" by Gunnar Sparr [6] as a reference. A reference book about C++ will also come in handy and the one I've used is "The C++ programming language" by Bjarne Stroustrop [7] (the creator of C++). Two good internet sites you can't live without when you are dealing with 3D programming is GameDev.net [d] and Gamasutra [e] which have many good articles in the area. Some good basic tutorial that I have been studied can be found at "Two kings game development" [f] and at Drunken Hyena [g]. But in this project the main 3D programming literature have been Peter Walsh's book "The Zen Of Direct3D Game Programming" [2].

5.1 Graphics API and GUI

In the beginning of the project I was battled with the problem of which graphics API to use (se chapter 3 for description of graphics APIs). My choice was to use Direct3D. That is, in comparison to World Up and World Toolkit, because I like to have full freedom and then better control when I'm programming and I didn't want my program to be dependent of a high level API. In comparison to OpenGL it is because I'm more used to work with Windows machines. By choosing DirectX it was natural to choose Microsoft's Visual C++ as the developing platform.

Besides the choice of the graphical API, I had also to decide which GUI to use (se chapter 4 for description of GUIs). If I had chosen World Up and World Toolkit the choice would have been the included GUI. But I didn't and then I had two alternatives. The first one was to use the fast "raw" Windows API and the second was to use the high level MFC. As I said before, I like to have full freedom and control when I'm programming, so the natural choice would have been the Windows API. But I chose MFC instead because my task wasn't to get a perfect working GUI and I didn't want to spend too much time on Windows programming. Instead I wanted to concentrate on 3D programming.

5.2 Hierarchical structure

The program consists of three main parts and they are MFC, Engine, and CD3DApplication. MFC is the GUI and is described in the previous chapter. Engine is the programs 3D-engine and is a package with all the 3D code included. It consists of three parts:

- One class called EngineFunctions for initializing Direct3D and DirectInput.
- One part with all the classes used in the program for creating 3D objects such as sectors, outlined sectors, terrain, lights, frames, the camera etc.
- One class for checking mouse events.

CD3DApplication is the applications main class which include functions for:

- Initializing and shutdown the application.
- Initializing and destroying the 3D scene.
- Rendering.
- Animating.

- Reading data files with coordinates and elevations to be used for creating objects.
- Handling mouse inputs.
- Setting predefined views.

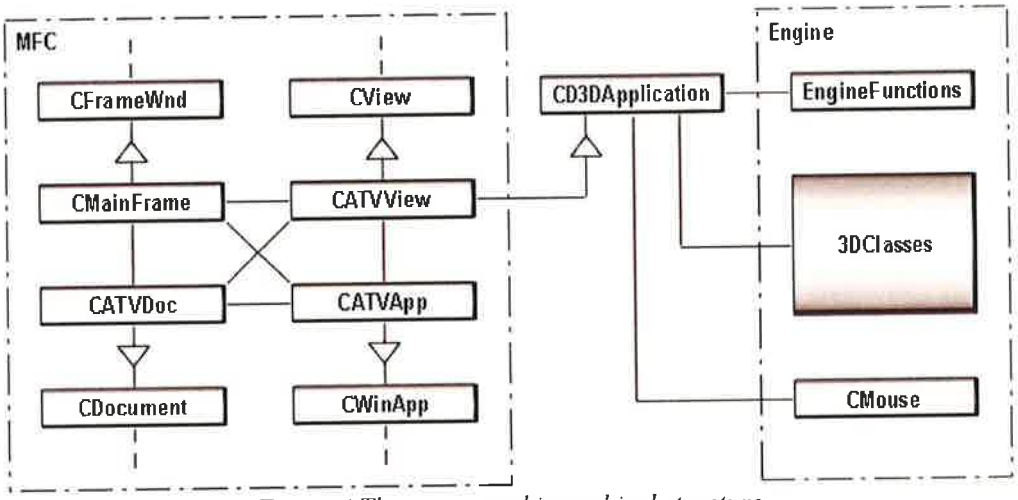

Figure 4 The programs hierarchical structure.

There were two main ideas behind this structure. The first was that I wanted to separate the 3D stuff from the rest of the code and make this part easy to plug-in in any application. To get access to the engine and its classes I just include the engine.h in the CD3DApplication's header file. The second idea was to let CATVView inherit from CD3DApplication, which gives the GUI access to CD3DApplication and then to the engine. The GUI that is started first at an execution of the program will then be able to:

- Initialize the application at start-up.
- Shutdown the application at the termination of the program.
- Affect the 3D objects and then render.
- At every rendering cycle animate the affected 3D objects and then render.
- Handle the mouse input.
- Set one of the predefined views.

Inspiration for how to combine an application with MFC in the code was got from the samples in Microsoft's DirectX 8.1 SDK for C++ [3].

5.3 Frames

I've used something called frames [2] to keep track of the 3D objects in the 3D scene (should not be confused with frame rate, fps etc). All the 3D objects inherit from the same base class CObject2 and is a must when you are about to dealing with frames. Frames are used to make it easier to look after objects in a 3D scene. Every object in the 3D scene could be kept in one or more frames. A frame is an invisible object that has its own position, velocity, and orientation. When objects are added to a frame their position, velocity, and orientation are updated relative to the frames position, velocity, and orientation. The next step is to add this frame to another frame and then get updated and so on. This will create a hierarchical structure with child objects, child frames and one main parent frame, where the objects are the leaves (figure 5).

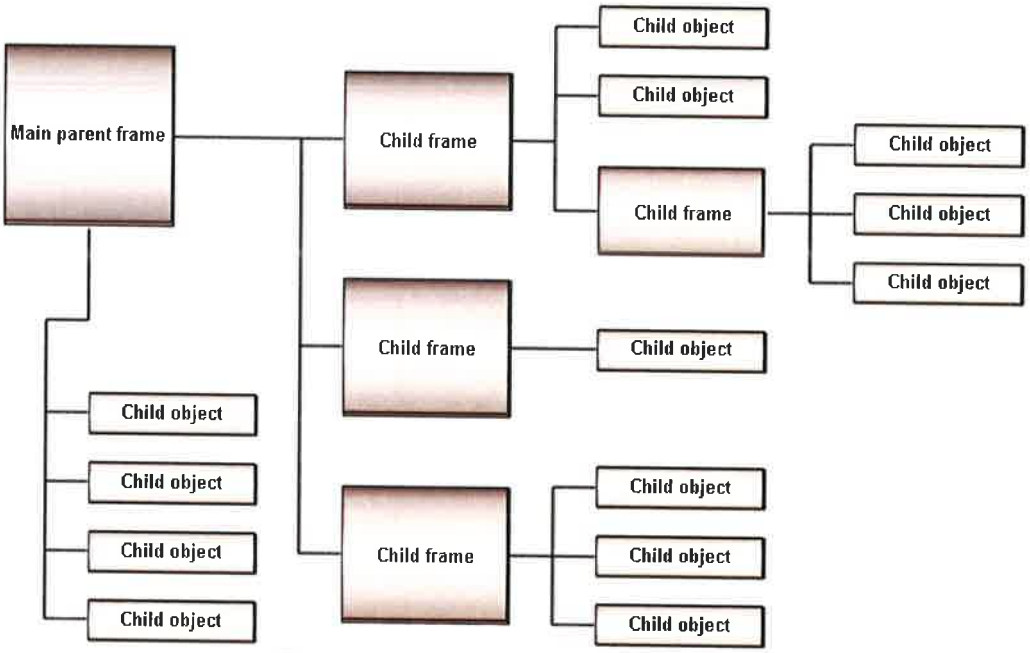

Figure 5 The frames hierarchical structure.

The advantage of this structure is that a single change of a parent frames position, velocity, and orientation will affect all its children. For example, imagine a swarm of birds circulating around some spot on the ground. The birds are randomly changing their position, velocity, and orientation but still staying together in a common circulating trajectory. First you have to create a frame that is circulating around a spot. Then you add a bunch of child frames, each with a bird object added to it. Next step is to program the bird objects to move randomly around their parent frame but in a closer trajectory than previous circulation. This will look realistically and the birds are keeping together in a randomly way. Only one change of the top parent frame is needed to update the whole structure. Without the frame structure it will be a very complex problem to update all the objects position, velocity, and orientation.

5.4 Included 3D parts

5.4.1 Sectors

There are three ways to show the sector parts. The first is as a solid volume, the second is as an outlined volume with a solid bottom, and the third is as an outlined volume with a transparent bottom. The sector parts upper and lower polygons are created by a complex algorithm that basically works like this: First all the vertices are placed in a list. Then the algorithm walks around the polygon vertices three by three and tests if it can do a new subdivision of the polygon into another triangle. As the algorithm runs by the vertices are deleted from the list until there are no vertices left. And the result will be a tessellated polygon of triangles.

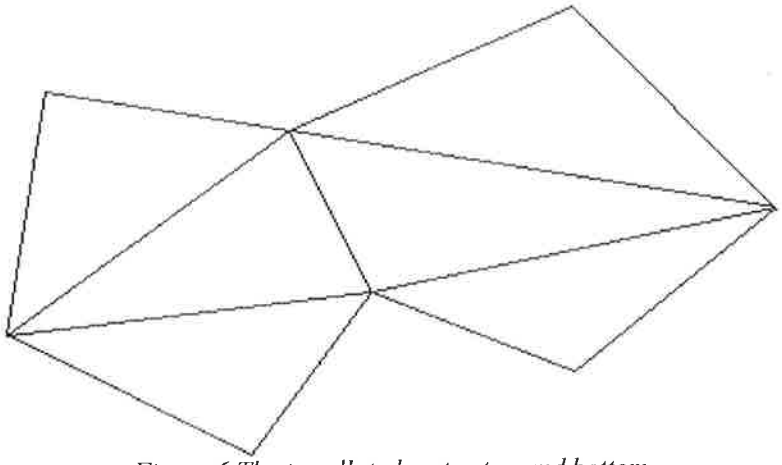

Figure 6 The tessellated sector top and bottom.

The sector parts vertices are created by reading the longitudes, latitudes, and flight levels from a couple of files. These files come from a program called "MapInfo" which they are using at ATCC (see section 6.4 for how to update the files).

5.4.2 The terrain on land

One interesting thing that I've been experiment with was to include real elevation data to the terrain on land. In the beginning I wasn't sure how to do it and soon I would discover that the task wasn't that easy. I started to search for hints on the net. A site that inspired me a lot was the Virtual Terrain Project [h]. Immediately I wanted to do a really good 3D landscape. But neither here nor at another place on the net could I find a description of how to use real world elevation data when making land part meshes with sharp and real world reflected coastlines. The methods are most of the times developed for games and doesn't have to correspond with the real world and therefore are the coastlines only made up (which makes it much easier). So I had to figure it out by myself and came up with two alternatives:

- Use a big map texture over the whole southern part of Sweden and combine it with elevation data.
- Combine vector data over the coastlines with elevation data to make one mesh of each land part.

The first alternative would be the easiest one but the ugliest, due to the coastlines wouldn't look good at a close zoom. By curiosity I first gave this alternative a try and went back to searching on the net. After many hours of searching I realized that it wasn't easy to find neither elevation data nor map bitmaps. But luckily I got some help from GIS centre [i], a department at Lund University working with digital GIS (Geographical Information System) and map data. They showed me an elevation database called GTOPO30 [j] on the net that was freely downloadable. GTOPO30 is a global elevation model resulting from a collaborative effort led by the staff at the U.S. Geological Survey's is Sioux Falls, South Dakota. The elevations are regularly spaced at 30-arc seconds (approximately 1 kilometre) and the elevation values range from -407 to 8752 meters. This was very good but they couldn't help me with the map texture. So again I went back to Internet for some further searching but it turned out to be very hard to get some good map textures, unless you want to spend a lot of money. So after spending too much time on searching on Internet I gave up this alternative.

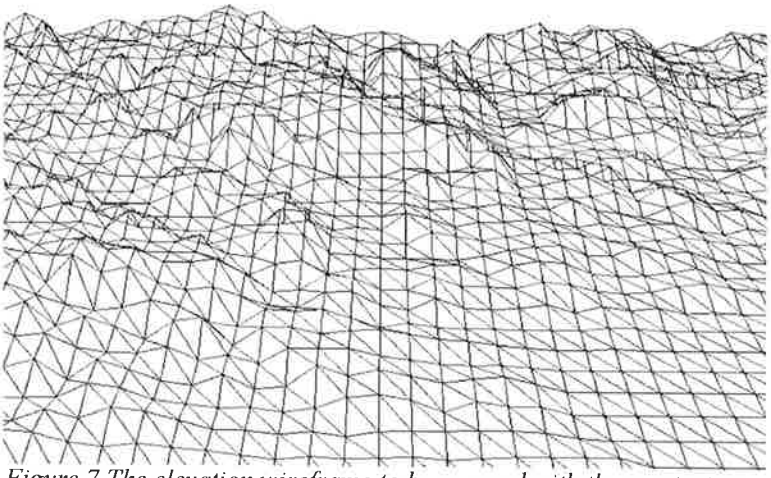

Figure 7 The elevation wireframe to be covered with the maptexture.

The ideal solution was of course to choose the second alternative and now I got the elevation data. So I started to examine the second alternative. I couldn't use the tessellation algorithm that I had made for the sectors because the triangle mesh wasn't equidistance, which it had to be for fitting with equidistance elevation data. I found out that I must make my own clipping algorithm without any help from DirectX, though DirectX doesn't support clipping of polygons with more than six clipping planes. This task is very hard to manage. I figured out how to do it theoretically but it would be very hard to implement and to test before its finish.

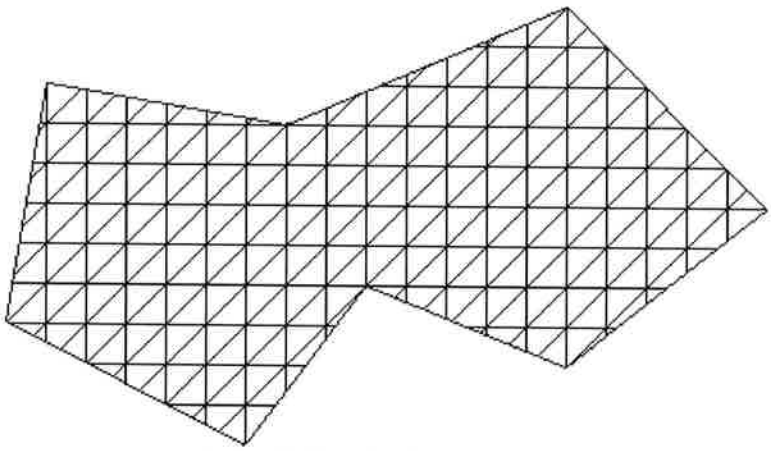

Figure 8 The ideal land part mesh.

Then I came up with a third alternative that was to create the land parts with two meshes. One mesh for the shores, made from vector data over the coastlines, and one mesh with the elevation data (figure 9). So instead I tried this one out and came up with a solution. But unfortunately I didn't like it because of two reasons. The first reason was that it didn't look so good, due to the intolerance of the vector data and the elevation data. The other reason and the most important one was that the 3D scene looked too messy with the elevations. By that I

mean, not to messy for a 3D computer game but to messy for being any help for an air traffic controller. But at least I could use the elevations to colour the land parts and create my own "texture" and so I did. I lowered all the elevations to zero and the result wasn't that bad. And at last I had come up with a solution that was good enough and I kept it this way.

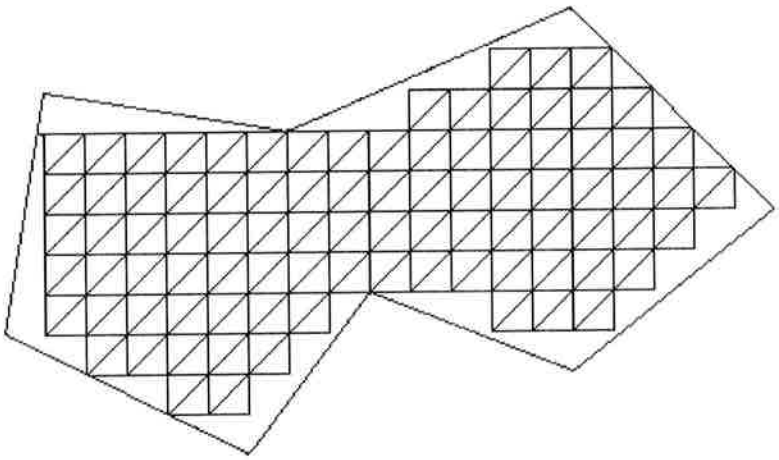

Figure 9 One shore mesh and one elevation mesh on the top.

5.4.3 Grid

I've put in a longitude and latitude grid on the map so that the air controllers could get a better orientation. The longitudes range from 10° to 21° and are regularly spaced with 1° between. The latitudes range from 54° to 60° and are regularly spaced with 1° between.

5.4.4 Way points

Beacons are marked with red squares and reporting points are marked with black triangles. Next to them they are marked with their names. The way points are created by reading the longitudes and latitudes from a file. This file comes from a program called "MapInfo" which they are using at ATCC (see section 6.4 for how to update the file).

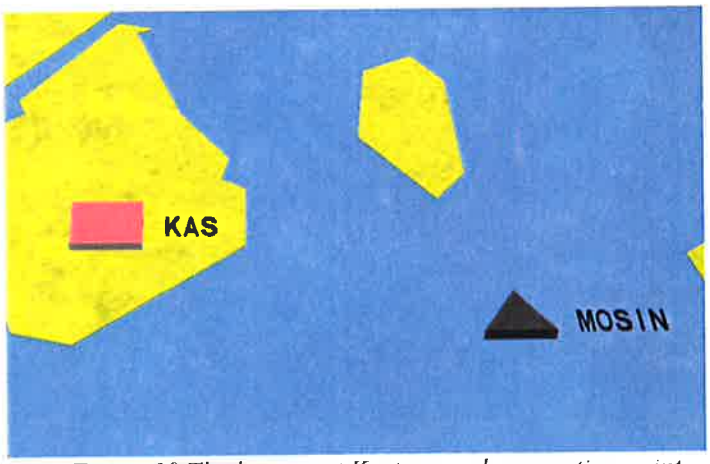

Figure 10 The beacon at Kastrup and a reporting point.

5.4.5 Routes

There are a couple of standard routes included in the program that are visualized with lines between the waypoints. The user is able to change the flight level of the routes in the lower and upper air space. The routes are created by reading the longitudes and latitudes from a couple of files. These files come from a program called "MapInfo" which they are using at ATCC (see section 6.4 for how to update the files).

5.4.6 Aircraft

To visualize an aircrafts flight through the sectors there's animated aircraft that can follow the routes. A timer in MFC controls the animation, which calls a function in CD3DApplication each tick. The aircraft is created in 3D Studio Max (the most commonly used 3D modeller for PCs) and then converted to DirectXs mesh format (x-files).

5.5 Mouse input

To check the mouse movement, its wheel and its buttons, a combination of MFC, and Direct Input is used. Where MFC polls the events and DirectInput checks them. This was done because it was easier to check repeating keystrokes in DirectInput. In early versions of the program, keys were also used to control the cameras movements. Otherwise I think it will work fine to both poll and check the events of the mouse in MFC.

5.6 Camera

The camera represents by an up vector, a right vector, and a look at vector. The up vectors value is <0,1,0>, the right vectors value is <1,0,0>, and the look at vectors value is <0,0,1>. The cameras orientation is controlled by using pitch, yaw, and roll. Where the pitch is the rotation around the x-axis, the yaw is the rotation around the y-axis, and the roll is the rotation around the z-axis.

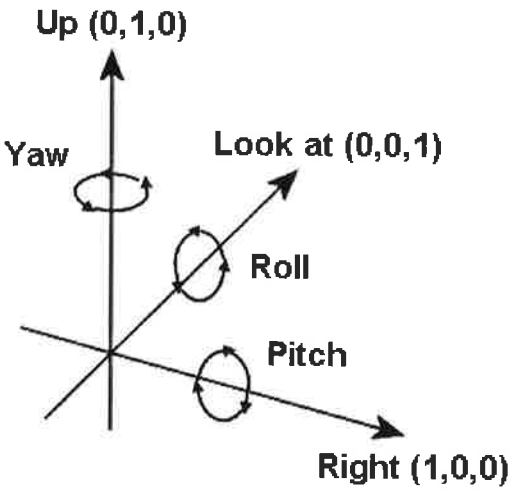

Figure 11 The camera representation.

6 The Air Traffic Viewer

6.1 Main window

The program consists of one main window divided in two parts. The left part is the view area and the right part is the dialog. The dialog is divided in five sections: Sectors, Mode, View points, Routes, and Terrain. Following figure shows the program in action:

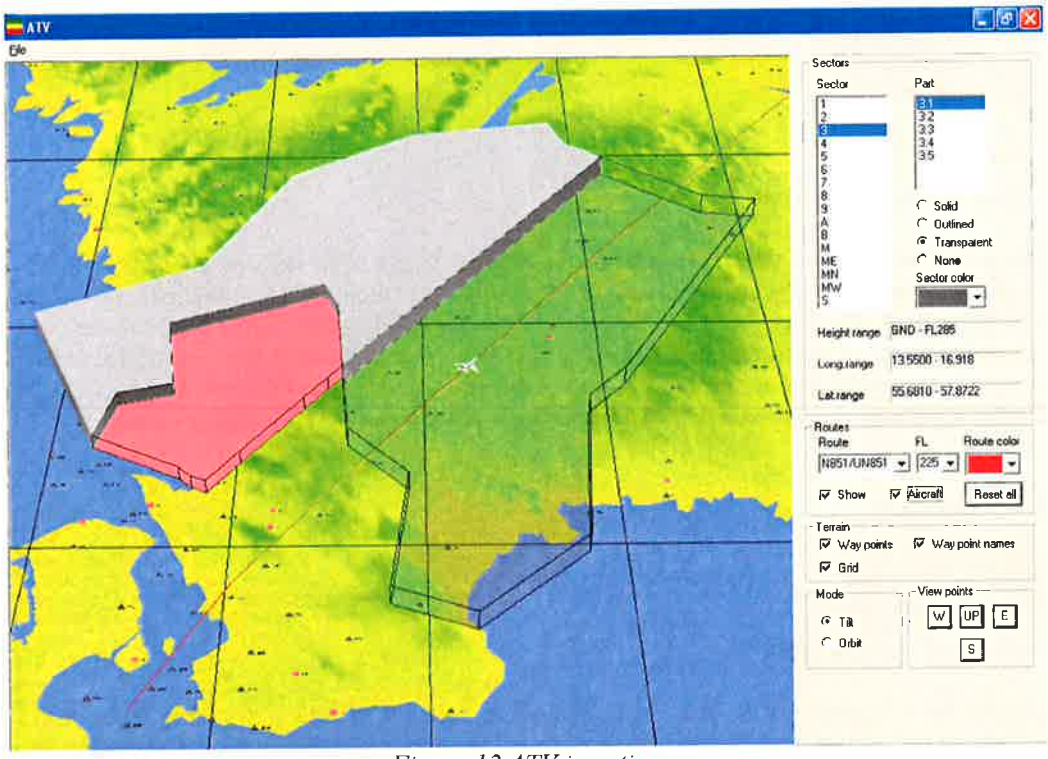

Figure 12 ATV in action.

6.2 Dialog sections

6.2.1 Sectors

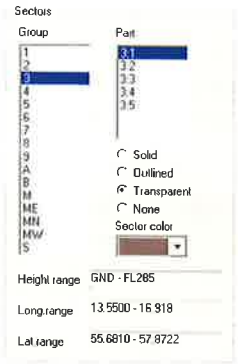

Figure 13 The sectors section.

There are three ways to show the sector parts. The first is as a solid volume, the second is as an outlined volume with a solid bottom, and the third is as an outlined volume with a transparent bottom (see figure 12 in the previous section). Select the current sector in the *Sector* list box and the current part in the *Part* list box. Now you can choose one of the three types of sector parts and in the *Select color* combo box you can select the colour. The sectors height, longitude, and latitude ranges are displayed in the bottom of this dialog section.

6.2.2 Routes

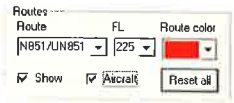

Figure 14 The routes section.

Use the *Route* combo box to select one of the routes. Show it by setting the *Show* check box. Adjust the height with the *FL* combo box. Change the colour with the *Route Color* combo box. Set an aircraft on the current route with the *Aircraft* check box. The *Reset all* button resets all the properties of the routes.

6.2.3 Terrain

Figure 15 The terrain section.

Beacons are marked with red squares and reporting points are marked with black triangles. Next to each way point is its name written. Set the *Way points* check box to show the way points. Set the *Way point names* check box to show the way points names. Set the *Grid* check box to show the grid.

6.2.4 Mode

Figure 16 The mode section.

To navigate around the 3D scene you are using the mouse, its wheel, and its buttons. There are two types of mouse controlling modes that decide the function of the mouse. The modes are the tilt mode (the default mode) and the orbit mode. The modes are almost identical except when only the left button is pressed. The controls are as follows:

Left Button + Mouse	Orbits up and down in tilt mode. Orbits up, down, left, and right in orbit mode.
Right button + Mouse	Moves the map in any horizontal direction.
Left Button + Right Button	Zooms in and out.
Wheel	Zooms in and out.

6.2.5 View points

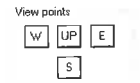

Figure 17 The view points section.

There are four predefined views. They are from the south, west, east, and above.

6.3 System requirement

- Windows 95, Windows 98, Windows Me or Windows XP…
- Geforce 2, 3, 4 or another graphic card with the same capacity…
- Pentium 3 or 4…
- Set the screen resolution to 1024x768 and the colour depth to 32 bits.

6.4 How to update the sectors, the routes, and the way points

- If you are running the program directly from the CD copy all the program files from the CD to the hard disc.
- For updating the sectors you exchange the Accs_mm.mid and Accs_mm.mif in the Sectors library with the new ones. Make sure that the data fields for the sector parts in Accs_mm.mid are written in the same order as in the old file. Otherwise the program will not work.
- For updating the routes you exchange the Cdr.mid, Cdr.mif, Reg.mid, and Reg.mif in the Routes library with the new ones. Make sure that the data fields for the routes in Cdr.mid and Reg.mid are written in the same order as in the old files. Otherwise the program will not work.
- For updating the way points you exchange the points.txt in the Waypoints library with the new one. Make sure that the data fields for the waypoints are written in the same order as in the old file. Otherwise the program will not work.
- Restart the program.

7 Final comments

7.1 What is achieved?

I have achieved the goal to design an implement a program, which is independent of an expensive high level graphics API, for visualizing air space sectors in 3D. And I have done it with a lot of new functions added in comparison to the previous thesis. This program will function as the second prototype in the area, and hopefully it will be used in the education for both students at SATSA and experienced air traffic controllers at ATCC when they are learning a new air space sector. Otherwise I'm sure it will be a very good starting point for further studies in the area. The code is implemented in an object oriented and reusable way where the 3D engine, the GUI and the main application is separated from each other.

7.2 What could have been done better?

The execution speed of the program isn't so good and I think the system requirements are a little too high for a program like this. I know two reasons for this. The first is that MFC is to slow and the other one is that I'm using a MFC timer for the animation. A better way would have been to use the "raw" Windows API instead of MFC and to use a separate thread instead of the MFC timer. Maybe these improvements will be enough to make a fast program.

Another thing that could have been done better is the tessellation of the sector parts polygons. The right way would have been to cut out a polygon from an equidistance mesh, instead of the algorithm that was walking around the polygons vertices three by three (se section 5.4.1). It looks good in the program but if you start to experiment with lights and normals you will soon see that the shading on the irregular tessellated polygons don't look so good.

As mention before in section 5.4.2 the right way to do the land parts was to do it the hard way, which was to combine vector data over the coastlines with elevation data to make only one mesh of each land part and not with two meshes as it's done now. To solve this task wouldn't be so different from the tessellation of the sectors.

7.3 Further work

Just take a look at the map in figure 2 in section 2.2 and you will understand that terrain, sectors, way points, and routes are not all to include in a program that visualizes the air traffic. I think that understanding different maps over the air space will be a good source for future studies in the area of visualizing air space sectors.

I'm not quite sure if the last word is said about the elevations, and if you for an instance look at the screenshots on different landscape meshes at Virtual Terrain Project [h] you start to wonder. Maybe if it's done in the right way as described in section 5.4.2, and with closer spacing between the elevations, and with more accurate elevations, it wouldn't be that messy and then be useful for an air traffic controller. This is also an area to investigate for future studies.

Another thing that can be much more developed is the routes and the aircrafts that follows them. To include real flight times and real velocity, position, and orientation of the aircrafts

would be an interesting further development of the program. The task would be to design some kind of simulator for the air traffic.

Finally, just like Welbert and Cepciansky [1] suggested as their final further development, I like to suggest development of a system that could be used not only during the education of the air traffic controllers and the students, but also as a supplement to the 2D radar in a real working situation.

References

[1] Victoria Welbert and Miro Cepciansky, "Using 3D models for visualizing air space sectors", Master thesis at Department of Design Sciences, Division of Ergonomics and Aerosol at Technology, Lund's Institute of Technology, Lund University, 2001.
[2] Peter Walsh, "The Zen Of Direct3D Game Programming", Prima Tech a division of Prima Publishing, 2001.
[3] Microsoft, DirectX 8.1 SDK
[4] Kate Gregory, "Using Visual C++ 6", Que, first edition 1998.
[5] Microsoft, Visual Studio 6.0 Documentation for C++.
[6] Gunnar Sparr, "Linjär algebra", Studentlitteratur, andra upplagan 1994.
[7] Bjarne Stroustrup, "The C++ programming language", third edition 1997.

Internet

[a] LFVs homepage, http://www.lfv.se/site/air_traffic_control/atcc/index.asp, http://www.lfv.se/site/air_traffic_control/education/index.asp
[b] World up and World toolkit's homepage, http://www.sense8.com
[c] Microsoft's homepage for developers, http://msdn.microsoft.com/visualc, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/compatibility_with_16_bit_windows.asp
[d] GameDev.net, http://www.gamedev.net
[e] Gamasutra, http://www.gamasutra.com
[f] Two kings game development, http://www.riaz.de
[g] Drunken Hyena, http://www.drunkenhyena.com/docs/d3d_tutorial.phtml
[h] Virtual Terrain Project, http://www.vterrain.org
[i] GIS centre, http://www.giscentrum.lu.se
[j] GTOPO30 data base, http://edcdaac.usgs.gov/gtopo30/gtopo30.html

Appendix A

Direct3D code snippet with the basic rendering procedure

```
// ---------------------- Do once --------------------------
// Create an index buffer
pDevice->CreateIndexBuffer( sizeof(short)*nbrOfIndices,
D3DUSAGE_WRITEONLY, D3DFMT_INDEX16, D3DPOOL_DEFAULT, &pIB );
BYTE* pIndexData;
// Lock the index buffer
pIB->Lock( 0, 0, &pIndexData, 0 );
vector<short>::iterator it0;
it0 = indices.begin();
// Fill the index buffer with integer indices
CopyMemory( pIndexData, (CVertex*)it0,
sizeof(short)*nbrOfIndices );
// Unlock the index buffer
pIB->Unlock();
// Create the vertex buffer
pDevice->CreateVertexBuffer( sizeof( CVertex )*nbrOfVertices,
D3DUSAGE_WRITEONLY, VERTEX_TYPE, D3DPOOL_DEFAULT, &pVB );
BYTE* pVertexData = 0;
// Lock the vertex buffer
pVB->Lock( 0, 0, &pVertexData, 0 );
vector<CVertex>::iterator it1;
it1 = vertices.begin();
// Fill the vertex buffer with the objects vertices
CopyMemory( pVertexData, it1, nbrOfVertices*sizeof(CVertex) );
// Unlock the vertex buffer
pVB->Unlock();

// ------------ Do every cycle in the rendering loop ---------
// Call the device to clear the back buffer
pDevice->Clear( 0L, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER |
D3DCLEAR_STENCIL, 0xff0000ff, 1.0f, 0L );
// Tell the device we are about to start rendering
pDevice->BeginScene();
// Connect the vertex buffer to a stream
pDevice->SetStreamSource( 0, pVB, sizeof( CVertex ) );
// Tell Direct3D about the index buffer
pDevice->SetIndices( pIB, 0 );
// Update the material
material.Update();
// Render the stream of vertices
pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0,
nbrOfVertices, 0, nbrOfFaces );
// Tell the device we have finished rendering
pDevice->EndScene();
// Present the back buffer to the primary surface
pDevice->Present( NULL, NULL, NULL, NULL );
```

KFS AB, Lund 2003