Department of Computer Science

Master Thesis
(Revision)

3D Visualization Tool for Automotive Navigation Systems
Design and Implementation

Arnold Adric
Matrikelnumber: 697963
Darmstadt, November 3, 2005

Supervising Professors:
Prof. Dr. Joachim Wietzke
Prof. Dr. Hans-Peter Wiedling
# Table of Contents

1 Preface.......................................................................................................................... 1
  1.1 Motivation.................................................................................................................. 1
  1.2 Thesis Layout.............................................................................................................. 1
  1.3 Contents of the enclosed CD-ROM......................................................................... 2

2 Introduction.................................................................................................................... 3
  2.1 Current State of the Art.......................................................................................... 3
  2.2 3D Visualization Tool Requirements .................................................................... 4
      2.2.1 Memory Requirements..................................................................................... 4
      2.2.2 Computing Requirements............................................................................... 4
      2.2.3 Visualizing 3D Models .................................................................................. 5
      2.2.4 Recognizing 3D File Formats ....................................................................... 5
      2.2.5 Changing the Camera Perspective ............................................................... 5

3 3D Visualization Principles ......................................................................................... 6
  3.1 Graphic Systems ..................................................................................................... 6
      3.1.1 Introduction to OpenGL ............................................................................... 7
      3.1.2 OpenGL for Embedded Systems .................................................................. 7
      3.1.3 OpenGL Architecture ................................................................................ 8
  3.2 Geometric Objects .................................................................................................. 10
      3.2.1 Geometric Primitives .................................................................................. 10
      3.2.2 Complex Primitives .................................................................................... 12
  3.3 Coordinate Systems ............................................................................................... 13
      3.3.1 Graphical Coordinate Systems .................................................................... 13
      3.3.2 Geographic Reference System ..................................................................... 15
      3.3.3 Coordinate Transformation ......................................................................... 17
  3.4 Projections and the Camera Model ......................................................................... 21
      3.4.1 The Synthetic Camera Model ....................................................................... 21
      3.4.2 Perspective Projection .................................................................................. 24
      3.4.3 Orthogonal Projection .................................................................................. 26
# Table of Contents

3.4.4 Projection Normalization .................................................. 28
3.5 Visibility ............................................................................. 29
  3.5.1 Clipping Algorithms ...................................................... 29
  3.5.2 Hidden Surface Removal Algorithms ............................ 33
3.6 Lighting.............................................................................. 34
  3.6.1 Light Sources and Material Properties ........................ 35
  3.6.2 The Phong Lighting Model .............................................. 37
  3.6.3 Shading Techniques ....................................................... 39
3.7 Textures ............................................................................. 41
4 Implementation in C++ with OpenGL ES .......................... 44
  4.1 Intention of Implementation ................................................ 44
  4.2 NAVTEQ ADASRP ............................................................. 44
    4.2.1 Main Components of ADASRP ................................. 46
    4.2.2 Unified Data Access Layer ......................................... 48
    4.2.3 Mobile Map Format .................................................... 51
  4.3 NAVTEQ 3D Specification ................................................. 54
    4.3.1 Generation of 3D Structure Models ............................ 55
  4.4 3D Visualization Concept .................................................. 57
  4.5 Class and Interface Definitions ............................................ 60
    4.5.1 Representation of a 3D Model ................................. 60
    4.5.2 3D Manager ............................................................... 62
    4.5.3 3D Search Box ........................................................ 67
    4.5.4 3D Object Locator .................................................... 68
    4.5.5 3D Object Loader ..................................................... 70
    4.5.6 3D Format Interpreter ................................................. 71
    4.5.7 3D Texture Interpreter .............................................. 73
5 Summary and Prospect .......................................................... 76
  5.1 Summary .......................................................................... 76
  5.2 Prospect ........................................................................... 76
A List of Abbreviations ............................................................ 78
B List of Figures ........................................................................ 80
# Table of Contents

C  List of Tables ................................................................. 82
D  Listings ........................................................................ 83
E  Bibliography .................................................................. 84
F  Statement of Authorship .................................................. 87
1 Preface

1.1 Motivation

The theme of this master thesis belongs to the field of automotive navigation systems. Navigation systems for cars have been around for several years. They use digital maps to perform tasks such as routing, positioning and guidance.

The NAVTEQ Company has spent many years collecting the data for digital maps, and it is expanding the coverage on a yearly basis. However, this data is targeted towards 2D navigation systems.

In recent years, the demand for three-dimensional map data has started to grow and NAVTEQ is in the process of adjusting to this new trend. It has begun to create 3D data for selected inner cities in Europe and the United States and has developed a specification for collecting and storing three-dimensional map data. In addition, a tool was required to visualize the 3D data for demonstration purposes to potential customers.

Through the supervising professor of this master thesis, Prof. Dr. Joachim Wietzke, the possibility arose to work together with NAVTEQ on a project to design and implement such a visualization tool.

1.2 Thesis Layout

This master thesis consists of five chapters. After the preface, a short introduction into the current state-of-the-art regarding 3D modeling and 3D navigation systems is presented. It is followed by an enumeration of the 3D Visualization Tool requirements.
Preface

In Chapter 3, the theoretical background of a graphics library is discussed. In particular, all steps required to generate a three-dimensional image are described, starting from the positioning of a 3D model to the point of projecting the model onto a screen. It also explains how different lighting of a scene and surface textures affect the appearance of a computer-generated image.

In chapter 4, the NAVTEQ Advanced Driver Assistance System Research Platform is introduced. It is used for the development of the 3D Visualization Tool. The description of a digital map format and its accompanying data access library conclude the first part of this chapter. The second part details the design and implementation of the 3D Visualization Tool using vital source code listings.

Chapter 5 summarizes the findings and offers some ideas for further development.

1.3 Contents of the enclosed CD-ROM

Attached to this master thesis is a CD-ROM. It contains the source code that was implemented for the 3D Visualization Tool.

In addition, a demo was put on the CD-ROM that can be used to run the 3D Visualization Tool. Please view the “readme.txt” file on how to install and use the demo. It is located in the root directory of the CD-ROM.
2 Introduction

This chapter provides an introduction into the current state-of-the-art regarding 3D navigation systems. It also outlines the requirements of the 3D Visualization Tool.

2.1 Current State of the Art

3D modeling has been around since the early 1990’s. However, it was mainly deployed for purposes other than navigation. At that time, the process of generating 3D models of real buildings was very time consuming and work intensive. Photographed or laser scanned pictures had to be converted which resulted in models that were not very detailed. For example, roofs and windows were only roughly indicated or not shown at all [EET04].

In recent years, advances in digital photography and digital processing enable a much less time- and work-consuming generation of 3D models with much greater detail. The country that has accelerated in this field the most is Japan. It has created 3D models of eleven major cities, among others Tokyo, Osaka and Nagoya. In addition, companies like Sony, Pioneer and Panasonic have developed navigation systems that include the visualization of these 3D city models [EET04].

Europe and the United States are lagging behind. So far, there has not been much demand for 3D navigation systems. People seem to be satisfied with 2D navigation. Furthermore, the automobile industry has not pushed for it. The problem is that almost no 3D city models exist that can be visualized. In addition, accelerated graphics hardware and microchips for embedded systems are still being developed. According to an online magazine, it is a risk for companies like Nvidia, an American graphic
board producer, to invest in further development if the market does not flourish [EET04].

However, NAVTEQ has started collecting and generating 3D data for a few cities worldwide and meanwhile some rudimentary 3D navigation systems are starting to come on the market. Nevertheless, there is still one problem that needs to be solved. That is the storage of this large amount of 3D data [EET04].

2.2 3D Visualization Tool Requirements

This section outlines the requirements of the 3D Visualization Tool. Section 2.2.1 and 2.2.2 address the system requirements while the other three sections describe the functional requirements.

2.2.1 Memory Requirements

The memory space available for embedded systems varies widely, ranging from PDAs (Personal Digital Assistants) with 64 MB of RAM (Random Access Memory) to mobile phones with 1 MB of RAM. Some more advanced in-vehicle computers may offer even more memory space. The 3D Visualization Tool should require no more memory than a PDA. This can be challenging considering the large amount of memory that a 3D model requires, especially one that represents an entire city. The design of the 3D Visualization Tool should consider this limitation and provide a solution that limits the number of models loaded at one time.

2.2.2 Computing Requirements

It is well known that 3D applications require a large amount of computing power. Hence, the choice of a suitable graphic library is important. Since OpenGL ES (Open Graphics Library for Embedded Systems) is probably the best option because of its increased performance, it is essential to exploit the advantages that this library offers. In addition, some design
choices regarding the level of detail should be considered. The use of accelerated graphics hardware will also lower the required computing power. Nevertheless, the system that is used to run the 3D Visualization Tool should have at least the same amount of computing power like a 400 Mhz PDA.

2.2.3 Visualizing 3D Models
One of the major functional requirements is the visualization of 3D models. This includes buildings and landmarks but it may also include a 3D model of the road network. The 3D Visualization Tool should provide a data structure that can represent both types of models. In addition, this data structure should be able to store the respective material properties.

2.2.4 Recognizing 3D File Formats
Another functional requirement is the recognition of different file formats. Several file formats exist that store 3D information. For example, Autodesk’s 3D Studio stores its 3D models in a binary file, also known as 3DS file. The 3D Visualization Tool should be capable of loading a common 3D file format and be extendable for others.

2.2.5 Changing the Camera Perspective
The camera is an integral part of a 3D application. Hence, the 3D Visualization Tool should be capable of changing the camera perspective. For example, a 3D navigation system may have the option to switch from a cockpit view to a bird’s eye view. This would counteract the phenomenon known as “lost in space”. The capability of zooming may also be useful.
3 3D Visualization Principles

This chapter provides an overview of the internals of a graphic system. In particular, the focus is on a graphic system that is based on a pipeline architecture. In addition, some key concepts regarding 3D visualization are discussed that form the basis for the next chapter.

3.1 Graphic Systems

A graphic system has all the components of a general-purpose computer system including a processor, memory, input and output devices. The purpose of a graphic system is to create computer-generated images [Ang03].

A computer-generated image is created as an array of color elements, also known as pixels. Collectively, they are stored in a part of memory called the frame buffer. The depth of the frame buffer defines the number of bits that are used to store each pixel. The resolution specifies the number of pixels that the frame buffer can hold. Some high-end graphic systems have special types of memory chips such as video random-access memory (VRAM) or dynamic random-access memory (DRAM) to enable a fast redisplay of the contents in the frame buffer [Ang03].

A simple graphic system may only have one processor that does both the normal processing and the graphical processing. However, a more sophisticated graphic system uses special-purpose processors to perform specific graphic functions [Ang03].

A cathode ray tube (CRT), a liquid crystal display (LCD) or a plasma panel may be used as the output device to display the pixels in the frame buffer.
buffer. These so-called raster displays are the most common display devices [Ang03].

The mouse, the joystick and the data tablet are the most common input devices. They provide positional information to a graphic system [Ang03].

At present, many tools exist that allow a programmer to write applications using interactive computer graphics. The most popular are Direct3D [D3D95] and OpenGL [SGI97]. The latter is the focus of this chapter. However, many concepts described in this chapter also apply to other graphic systems.

3.1.1 Introduction to OpenGL

The OpenGL specification is the leading 2D and 3D graphic standard for today’s desktop systems. Since its introduction in 1992, various industrial branches have implemented their 3D applications with the OpenGL API (Application Programming Interface). Motivated by the success of OpenGL, a 3D specification for embedded systems was developed in 2003. This new OpenGL ES specification describes a well-defined subset of OpenGL. It enables companies to realize 3D applications even on the smallest appliances such as portable PDAs and mobile phones. The automotive industry uses OpenGL ES to display three-dimensional terrain data, buildings and POIs (Points of Interest) in their navigation systems [SGI97].

3.1.2 OpenGL for Embedded Systems

OpenGL ES is a low-level, platform-independent interface between 3D software and accelerated graphics hardware. It was developed by the Khronos Group and supports profiles for floating-point and fixed-point systems [Khr00].
The Native Platform Graphics Interface Layer (EGL) works between the OpenGL ES API and the underlying windowing system. The implementation of this layer is platform-specific and may be included as part of an OpenGL ES distribution.

The 3D Visualization Tool utilizes Hybrid's [Hyb05] OpenGL ES implementation for Windows 2000/XP. However, it is assumed that there is no functional difference between OpenGL and OpenGL ES unless stated otherwise.

3.1.3 OpenGL Architecture

With the advent of special-purpose VLSI (Very Large-Scale Integration) circuits, pipeline architectures became increasingly important. The main advantage of a pipeline architecture is the significant improvement in computational speed. However, pipeline architectures are only reasonable if the same operations are performed on many data sets. A graphic system fulfills this requirement because many geometric objects are processed in the same manner. The major disadvantage of a pipeline architecture is that dependencies between subsequent data sets are lost. In a graphic system, this poses a problem when considering global lighting effects (Section 3.6.2).

The Graphic Pipeline: The following paragraphs provide an overview of the individual steps involved in the imaging process of OpenGL. It includes transformations, clipping, projection and rasterization (Figure 3.1) [Ang03]:

![Figure 3.1: Graphic Pipeline](image-url)
1. **Transformations:** Many of the steps in the imaging process can be viewed as transformations between different coordinate systems. For example, in the beginning of the graphic pipeline the geometric objects are converted from the coordinate system in which they are defined to a representation in terms of the coordinate system of OpenGL. In the case of the 3D Visualization Tool, this step requires a transformation between geographic coordinates and Cartesian Coordinates (Section 3.3.3). Another transformation is necessary to convert the geometric objects into camera coordinates (Section 3.4.1). OpenGL performs both transformations using a 4 x 4 matrix called the model-view matrix.

2. **Clipping:** The second fundamental operation in the graphic pipeline is clipping. Clipping is necessary because not all geometric objects may appear on the screen. OpenGL uses a standard view volume (Section 3.4.4) to determine which objects need to be clipped. A clipping algorithm is described in Section 3.5.1.

3. **Projection:** The projection step converts all geometric objects from their original three-dimensional representation to a two-dimensional representation. Similar to the first step, OpenGL uses a 4 x 4 projection matrix to perform this task. OpenGL supports two different types of projections (Section 3.4.2 and 3.4.3).

4. **Rasterization:** Finally, the projected objects must be represented as pixels in the frame buffer. It is the job of the rasterizer to perform this task. Two famous rasterizing algorithms can be found in [Ang03].

**The State Machine:** OpenGL is implemented as a state machine. The functions in OpenGL are of two types: Those to define geometric objects that flow through the graphic pipeline and those to change the state of the
machine. Consequently, most parameters in OpenGL are persistent. In other words, their values remain unchanged unless they are explicitly changed through functions that alter the state. For example, the color of a geometric object is set through a color-altering function that changes the color state. Neglecting to make the necessary state changes or loosing track of the present state can create errors that are hard to find [Ang03].

3.2 Geometric Objects

A scene in a graphic system consists of geometric objects whose shape is only limited by the imagination of the application programmer. Since it is impossible to support all geometric objects, the graphics community has agreed that a graphic system should support a small set of geometric primitives that can be used as the building blocks for other geometric objects. This section describes the geometric primitives supported by OpenGL. It also explains how OpenGL can be used to create primitives that are more complex. Keep in mind that there are variations concerning the primitives supported by other graphic systems.

3.2.1 Geometric Primitives

The geometric primitives supported by OpenGL include points, line segments and polygons, all of which can be generated efficiently in today's graphics hardware. Their descriptions and limitations are the scope of the following paragraphs [Ang03]:

Points: A point is the smallest entity in a graphic system. In computer graphics a point is called “vertex”. It is represented by a set of three floating-point numbers (x, y, z). However, to distinguish between vectors and points, OpenGL uses homogenous coordinates consisting of four floating-point numbers (x, y, z, w). If w equals zero, the three coordinates
(x, y, z) specify a vector from (0, 0, 0) to the point (x, y, z). Otherwise, these coordinates correspond to a three-dimensional point (x/w, y/w, z/w).

**Line Segments:** In contrast to lines that have an infinite length, line segments are delimited by two endpoints. A pair of vertices specifies the endpoints of a line segment. Since line segments are of such importance, they are considered a basic graphical entity. OpenGL allows connecting successive line segments to form a polyline. A suitable polyline can be used to approximate any curve.

**Polygons:** Polygons refer to an area that is bounded by a closed polyline. They are typically drawn with a filled interior. However, OpenGL can also draw just the outline of a polygon. The surrounding edges are defined by a list of vertices. Depending on how these vertices are specified, it may be ambiguous what belongs to the interior of a polygon. Consequently, OpenGL makes some restrictions on the definition of a primitive polygon. Three properties ensure that a polygon is displayed correctly: It must be simple, convex and flat.

The definition of a simple polygon states that no edges cross each other. Simple polygons have well-defined interiors. Since the time to test for simplicity is relatively high, most graphic systems require that an application program perform any necessary testing.

Furthermore, OpenGL guarantees a consistent polygon fill only if the polygon is convex. A polygon is convex if, given any two points in the interior of a polygon, the line segment joining these two points is also in the interior. Like simplicity testing, convexity testing is time-consuming and therefore it is left to the application program.
Finally, polygons must be flat to be drawn correctly. For a flat polygon, all vertices must lie on the same plane. Most graphic systems rely on the fact that the vertices of a triangle are located on the same plane. Hence, using triangles as the building blocks for other geometric objects assures that they are rendered correctly because triangles are flat, convex and simple.

In addition to triangles, OpenGL also supports polygons with any number of vertices. However, it does not guarantee their correct display. In contrast, OpenGL ES only supports triangles.

### 3.2.2 Complex Primitives

Using the geometric primitives described in the previous section, an application programmer can approximate many curves and surfaces. For example, a curve can be created using successive line segments as shown in figure 3.2. Similar, a sphere can be approximated with a set of triangles, which can be generated using a recursive subdivision algorithm on a tetrahedron. The OpenGL utility library (GLU)\(^1\) provides a collection of approximations to some common curved surfaces [Ang03].

![Figure 3.2: Approximation of a Curve and a Sphere](image)

OpenGL also supports the use of “evaluators” to create geometric objects based on their mathematical definition. However, discussing this advanced topic would go beyond the scope of this thesis. Besides, evaluators are not supported by OpenGL ES. For those who are interested, please view the suggested readings in [Woo99].

---

\(^1\) At the writing of this thesis, the GLU library did not exist for OpenGL ES.
3.3 Coordinate Systems

The 3D Visualization Tool utilizes OpenGL ES to project geometric objects onto a screen. Before the objects appear on the screen, they have to be positioned in the coordinate space of the graphic system. This involves the transformation of geographic coordinates into Cartesian coordinates. The following sections provide the theoretical background to perform this task.

3.3.1 Graphical Coordinate Systems

During the process of projecting an object onto a screen, its vertices undergo several coordinate transformations. Figure 3.3 shows the different types of coordinate systems encountered in a graphic system.

**World Coordinate System (WCS):** The world coordinate system allows a programmer to position objects in the three-dimensional space of a graphic system. The programmer can choose the units and the origin of the coordinate system and does not have to worry about the details of input and output devices.
Object Coordinate System (OCS): The object coordinate system is local to each individual object. It is up to the artist creating the object to specify the units and the origin of the object’s coordinate system. The vertices of an object are expressed relative to the local origin.

Camera Coordinate System (VCS): The camera coordinate system defines the position and the orientation of the camera. The vertices of all objects are translated with respect to the origin of the camera coordinate system, so that the generated image portrays a specified point of view. A programmer has the ability to specify the position and the orientation of the VCS within the world coordinate system (Section 3.4.1).

Projection Coordinate System (PCS): The projection coordinate system defines the two-dimensional plane onto which a three-dimensional object is projected before the image is translated into device coordinates. The vertices of all objects are projected towards the center of projection (COP). Usually the COP coincides with the origin of the VCS. However, for some projections these two points may be different.

Device Coordinate System (DCS): The device coordinate system defines the boundaries for input and output devices. The device coordinates specify pixel locations in the frame buffer and are expressed as integer values.

The high number of different coordinate systems provides a great convenience to the user of a graphic system when developing an application. For example, a user does not have to worry about mapping world coordinates into device coordinates. It is the responsibility of the graphic system to perform this task. Having a separate camera coordinate system has the advantage of specifying objects and camera parameters.
independently. This enables a user to change the camera perspective without changing the vertices of the objects [Ang03].

3.3.2 Geographic Reference System

The earth is a three-dimensional object whose shape is represented by the geoid\(^2\). To perform distance calculations commonly used for navigation purposes, a reference system is necessary to mathematically describe the irregular shape of the earth. The best approximation of the geoid is an ellipsoid (Figure 3.4). Another good approximation is a sphere [Dan03].

![Figure 3.4: Geoid, Ellipsoid and Adjusted Ellipsoid](image)

**Ellipsoidal Earth Model:** An ellipsoidal earth model best describes the size and shape of the earth. Together with a description of its position and orientation in space, the ellipsoid forms the geographic reference system also known as geodetic datum. Hundreds of geodetic datums are in use around the world. Most of them are optimized to best-fit local regions. The Global Positioning System (GPS) uses a global reference system, the World Geodetic System 1984 (WGS-84). It is frequently used as the basis for international applications such as navigation [Dan03].

---

\(^2\) A representation of the earth based on the mean sea level surface. The theoretically continuous surface is perpendicular at every point to the direction of gravity.
To define an ellipsoid two parameters are required:

- The semi-major axis (equatorial radius)
- The semi-minor axis (polar radius)

Other parameters such as flattening and eccentricity are computed from these two terms (Figure 3.5).

**Semi-Minor Axis** = Polar Radius = \( b \)
(WGS-84 Value = 6356752.3142 Meters)

**Semi-Major Axis** = Equatorial Radius = \( a \)
(WGS-84 Value = 6378137.0 Meters)

**Flattening** = \( (a - b) / a = f \)
(WGS-84 Value = 1 / 298.257223563)

**First Eccentricity Squared** = \( 2f - f^2 = e^2 \)
(WGS-84 Value = 0.00669437999013)

**Figure 3.5: Ellipsoidal Parameters**

**Geographic Coordinate System:** A geographic coordinate system is the basis for specifying locations on the surface of the earth. There are many different coordinate systems, based on a variety of geodetic datums. The most commonly used coordinate system today is the latitude, longitude and height system. It has its origin at the center of gravity of the earth. The Prime Meridian (P.M.) and the Equator form the reference planes used to define latitude and longitude [Dan03].
The definition of latitude, longitude and height are described as follows (Figure 3.6):

The **geodetic latitude** \( (\Phi) \) of a point is the angle from the equatorial plane to the line passing through the point that is normal to the reference ellipsoid.

The **geodetic longitude** \( (\lambda) \) of a point is the angle between the reference plane of the Prime Meridian and the plane passing through the point and the origin. Both planes are perpendicular to the equatorial plane.

The **geodetic height** \( (h) \) of a point is the distance from the reference ellipsoid to the point in the direction normal to the ellipsoid.

![Figure 3.6: Geodetic Latitude, Longitude and Height](image)

Lines of latitude, also known as parallels, range from zero degrees at the Equator to 90 degrees at the North Pole and minus 90 degrees at the South Pole. The meridians are measured from zero degrees at the Prime Meridian to 180 degrees East and minus 180 degrees West. The intersection of the Prime Meridian and the Equator defines the geographic reference point for all geographic coordinates [Dan03].

### 3.3.3 Coordinate Transformation

This section describes the transformation of latitude and longitude into Cartesian coordinates (world coordinates of the graphic system) and vice versa [Fur96].

The process of transforming a curved reference surface of the earth onto a flat plane is called map projection. Each map projection is defined by a set of equations:

- \( x, y = f(\phi, \lambda) \) **Forward Equation** (3.1)
- \( \phi, \lambda = f(x, y) \) **Inverse Equation** (3.2)
The forward equation is used to transform geographic coordinates such as latitude and longitude into Cartesian coordinates. The inverse equation transforms Cartesian coordinates into geographic coordinates.

While there are many map projections available, all of them have at least some distortions of conformality, distance, direction, scale and area.

The Sinusoidal Projection, also known as the Sanson-Flamsteed Projection or the Mercator Equal-Area Projection translates geographic coordinates into the $x$ and $y$ coordinates of the projection plane with the following equations:

\begin{align*}
    x &= R \times \lambda \times \cos(\phi) \\
    y &= R \times \phi
\end{align*}

The equations are based on a spherical representation of the earth with radius $R$. Latitude and Longitude are expressed with the Greek symbols $\Phi$ and $\lambda$ respectively. Figure 3.7 illustrates the derivation of the above equations.

![Figure 3.7: Geographic to Cartesian Coordinate Transformation](image-url)
The Sinusoidal Projection exploits the fact that all meridians are great circles\(^3\) with radius \(R\). To find the projection of latitude \(\Phi\) along the arc of a great circle, one has to multiply the radius of the earth \(R\) with the angle of latitude \(\Phi\). This yields the projected y-coordinate.

The projection of longitude \(\lambda\) is slightly more difficult. It depends on the radius of the parallel that orbits the sphere at latitude \(\Phi\). To find the radius of the parallel, one has to multiply the radius of the earth \(R\) with the cosine of latitude \(\Phi\). The projected x-coordinate is then calculated by \(R \times \lambda \times \cos(\Phi)\).

![Figure 3.8: Sinusoidal Projection](image)

The map that is produced by the Sinusoidal Projection is pictured in figure 3.8. It shows that the distortion in the Polar Regions is greater than along the Equator. An explanation for this phenomenon is the curvature of the earth’s surface. The projected coordinates far away from the geographic reference point experience more distortion than the ones closer to the reference point (Figure 3.9).

---

\(^3\) A circle on the surface of a sphere that has the same diameter/center as the sphere.
Figure 3.9: Projection Plane tangent to Earth Surface

One way to compensate for these distortions is through convenient recentering where the origin of the projection plane is moved from the original reference point to a new reference point \((\Phi_0, \lambda_0)\) that is close to the region of interest. That minimizes distortions at the new reference point. Consequently, the projection equations have to be revised as follows:

\[
\begin{align*}
\text{1.} & \quad x = R^* (\lambda_i - \lambda_0) \cos(\phi_i) \quad (3.5) \\
\text{2.} & \quad y = R^* (\phi_i - \phi_0) \quad (3.6)
\end{align*}
\]

A simple conversion produces the following inverse equations:

\[
\begin{align*}
\text{1.} & \quad \lambda_i = \lambda_0 + \frac{x}{R^* \cos(\phi_i)} \quad (3.7) \\
\text{2.} & \quad \phi_i = \phi_0 + \frac{y}{R} \quad (3.8)
\end{align*}
\]

The 3D Visualization Tool uses the above equations to position three-dimensional objects with geographic coordinates in the three-dimensional world of the graphic system. Through constant recentering, the 3D Visualization Tool ensures that distances relative to the reference point are kept small. Therefore, the distortions that occur for greater distances can be neglected.
3D Visualization Principles

3.4 Projections and the Camera Model

The process of creating a computer-generated image requires two essential entities: an object and a viewer. The viewer can be visualized as a virtual camera that looks at an object. It forms the image that is later seen on a two-dimensional screen.

In reality, the camera is represented by a coordinate system (Section 3.3.1). Its orientation in space and the definition of a projection type provide the basis for creating a computer-generated image.

The following sections introduce the synthetic camera model and the different types of projections supported by OpenGL [Ang03].

3.4.1 The Synthetic Camera Model

The synthetic camera model emulates the classical imaging systems and it is the basis for a number of graphic APIs. It is synthetic because the objects being imaged do not exist physically but are stored in computer memory. The similarities between classical and computer imaging systems include the basic elements necessary to create an image. Both need objects, a viewer, projectors and a projection plane.

![Figure 3.10: Synthetic Camera Model](image)
In a traditional camera, the center of projection is at the center of the lens whereas in a graphic system, the center of projection is at the origin of the camera coordinate system. Rays emanating from the COP to points on the object are called projectors. The points of projection are located where the projectors intersect the view plane. For convenience, the view plane in a graphic system is moved in front of the camera as shown in figure 3.10.

The combination of the lens and the size of the film determine how much of the world in front of the camera appears on the image. In a graphic system, specifying the type of projection and the viewing parameters makes the equivalent choice.

Characteristic for the synthetic camera model is the independence of specifying objects and the camera parameters. It gives the programmer certain flexibility that the graphic system has to account for. To portray a correct image, the relationship between the camera and the objects is important. Hence, viewing in a graphic system consists of two fundamental operations:

First, the camera has to be positioned in the world coordinate space of the graphic system. This operation prepares the model-view matrix to transform the objects’ vertices from world coordinates into camera coordinates.

Second, the type of projection has to be applied to the projection matrix. It is the basis for transforming the objects’ vertices from camera coordinates into projection coordinates. The OpenGL API provides the functionality to perform these tasks.
Consider the following OpenGL function:

- \texttt{gluLookAt(eye}_X, \texttt{eye}_Y, \texttt{eye}_Z, \texttt{at}_X, \texttt{at}_Y, \texttt{at}_Z, \texttt{up}_X, \texttt{up}_Y, \texttt{up}_Z);\)

It positions the camera and indirectly specifies the coordinate axes of the camera coordinate system.

Figure 3.11 shows a camera located at the “Eye-Point”. The position of the camera is specified in world coordinates. It is oriented toward a second point, called the “At-Point”. Subtracting the At-Point from the Eye-Point produces the view-plane normal \(\mathbf{n}\). Another vector \(\mathbf{v}\), also known as the view-up vector, is necessary to fix the camera. This vector is part of the “Look-At-Function” and specifies the up-direction of the camera. With two axes specified, the third can be found by taking the cross product \(\mathbf{u} = \mathbf{v} \times \mathbf{n}\). This completes the definition of the camera coordinate system having its origin (COP) at the Eye-Point.

As previously stated, the Look-At-Function alters the model-view matrix and consequently the state of the graphic system. In a next step, the projection matrix is modified to apply the specified projection. OpenGL provides the choice of an orthogonal or a perspective projection. The product of the model-view matrix and the projection matrix defines the
3D Visualization Principles

current transformation matrix (CTM) upon which all subsequent vertex operations are based (Figure 3.12).

![Diagram of Model-View and Projection Matrices](image)

Figure 3.12: Model-View and Projection Matrices

### 3.4.2 Perspective Projection

Perspective viewing is characterized by a diminution of size. In other words, objects far from the viewer appear smaller than objects close to the viewer. This gives images their natural appearance.

In a perspective projection, a view volume as shown in figure 3.13 is created to define the boundaries of a three-dimensional scene. All objects within this view volume appear on the image. Objects outside of this view volume are clipped out of the scene.

![Diagram of Perspective View Volume](image)

Figure 3.13: Perspective View Volume

OpenGL provides two methods for specifying a perspective view volume:

- `glFrustum(left, right, bottom, top, near, far);`
- `gluPerspective(fovy, aspect, near, far);`
The first function defines the view volume by specifying the lower left coordinate of the front clipping plane (left, bottom, near) and the upper right coordinate of the back clipping plane (right, top, far). These coordinates are measured from the center of projection. Note that the resulting view volume does not have to be symmetric with respect to the COP.

The second function defines a symmetric view volume. It specifies the field of view in the y direction (fovy), which is the angle between the top and the bottom plane of the view volume. The aspect ratio is the width of the view plane divided by the height of the view plane. Near and Far are the distances of the front and back clipping planes measured from the COP.

The COP in a perspective view is a finite distance away from the objects located in the view volume. To find the projection \((x_p, y_p, z_p)\) of a point, consider figure 3.14.

It shows the camera coordinate system and three different views of the perspective projection of a point \((x, y, z)\). The imaginary camera is located at the origin of the coordinate system. It points in the negative z direction and forms two right triangles as shown in the top view of figure 3.14(b).
Based on this observation, the following equation can be set up:

\[ \frac{x}{z} = \frac{x_p}{d} \]  

(3.9)

Solving for \( x_p \) produces the equation for the projected \( x \)-coordinate:

\[ x_p = \frac{x}{z/d} \]  

(3.10)

A similar result is obtained for \( y_p \) using the side view of figure 3.14(c):

\[ y_p = \frac{y}{z/d} \]  

(3.11)

Since the projection plane is perpendicular to the \( z \)-axis, it follows that:

\[ z_p = d \]  

(3.12)

These projection equations are sufficient for a simple perspective projection with a symmetric view volume. OpenGL uses a more sophisticated approach that also considers asymmetrical view volumes. A technique that is known as projection normalization (Section 3.4.4).

### 3.4.3 Orthogonal Projection

Characteristic for an orthogonal projection is the preservation of distance and size. This makes orthogonal viewing useful for architectural applications where the relationship of distances is important.
The view volume of an orthogonal projection is a parallelepiped as shown in figure 3.15.

![Figure 3.15: Orthogonal View Volume](image)

It is defined by the following OpenGL function:

- `glOrtho(left, right, bottom, top, near, far);`

The parameters are identical to those of `glFrustum()` and specify the lower left coordinate of the front clipping plane and the upper right coordinate of the back clipping plane.

The projectors in an orthogonal projection are perpendicular to the view plane as if the COP is moved to infinity. Rather than moving the COP to infinity, the projection equations can be derived directly.

![Figure 3.16: Orthogonal Projection](image)

Figure 3.16 shows a three-dimensional view of an orthogonal projection. The view plane is located at the COP perpendicular to the z-axis. As the
point \((x, y, z)\) is projected onto the view plane, it retains the \(x\) and \(y\) values. In this picture, the third dimension of the projected point \((x_p, y_p, z_p)\) equals zero.

In general, the projection equations for an orthogonal projection are

\[
\begin{align*}
  x_p &= x & (3.13) \\
  y_p &= y & (3.14) \\
  z_p &= d & (3.15)
\end{align*}
\]

where \(d\) is the distance along the \(z\)-axis from the COP to the view plane.

### 3.4.4 Projection Normalization

As mentioned previously, OpenGL uses a technique called projection normalization to perform perspective and orthogonal projections. After specifying the view volume as described in the previous sections, OpenGL breaks the projection process into two parts. The first part converts the specified view volume into a standard view volume – a cube centered at the origin of the camera coordinate system with sides of length two. The second part uses the orthogonal projection equations derived in section 3.4.3 to create the desired projection.

A series of transformations is necessary to translate the objects’ vertices from the original view volume into vertices within the standard view volume. The math involved to perform these transformations is described in [Ang03].
Converting all projections into an orthogonal projection has several advantages:

- Vertices are independent of any previous coordinate system.
- OpenGL requires only one pipeline for all possible projections.
- Other projections that are not directly supported by OpenGL can be obtained by specifying the transformation matrix directly.
- An orthogonal view volume simplifies the detection of invisible surfaces (Section 3.5).

3.5 Visibility

Part of the projection process is the removal of invisible surfaces that arise if an object is partially outside of the view volume or if the surface is obscured by another object within the view volume. In both cases, there are several algorithms that deal with this problem. Clipping algorithms and Hidden Surface Removal algorithms are the terms used to distinguish between the two and are the focus of the following sections [Ang03].

3.5.1 Clipping Algorithms

The surface of an object consists of polygons with varying shapes and sizes. Each polygon is bounded by its edges, which are the line segments between the vertices of the polygon.

The Cohen-Sutherland algorithm uses these line segments successively to find the parts of an object that are outside of the view volume. It starts by extending the sides of a view volume to form 27 regions as shown in figure 3.17.
A 6-bit binary number, $a_0a_1a_2a_3a_4a_5$, is assigned to each region with the following meaning:

- $a_0 = 1$ if the region is **in front of** the view volume.
- $a_1 = 1$ if the region is **behind** the view volume.
- $a_2 = 1$ if the region is **above** the view volume.
- $a_3 = 1$ if the region is **below** the view volume.
- $a_4 = 1$ if the region is **to the right of** the view volume.
- $a_5 = 1$ if the region is **to the left of** the view volume.

Next, the binary numbers for the two endpoints of a line segment are computed. The need for clipping is based on these two binary numbers. There are four cases that have to be considered:
1. If the two binary numbers equal zero, both endpoints are inside the view volume and no clipping is needed (see segment AB in Figure 3.18).

2. If one of the two binary numbers equals zero and the other is unequal to zero, clipping is needed because one endpoint is outside of the view volume (see segment CD in Figure 3.18).

3. If the bitwise AND of the two binary numbers is unequal to zero, both endpoints are outside of the view volume. No clipping is needed because the line does not intersect with the view volume since both endpoints are on the same side. Thus, the line segment can be discarded (see segment EF in Figure 3.18).

4. If the bitwise AND of the two binary numbers equals zero, both endpoints are outside of the view volume but on different sides. From the binary numbers it cannot be determined if clipping is needed because the line may or may not intersect with the view volume (see segment GH and IJ in Figure 3.18).

For those line segments that have to be clipped (case 2 and case 4) the (x, y, z) coordinates of the intersection between the line segment and the view volume have to be calculated. There are two equations necessary to find this point of intersection. Both are defined in matrix form:

- \[ p(\alpha) = (1 - \alpha)p_1 + \alpha p_2 \quad \text{Line Equation} \]  
- \[ 0 = \vec{n} \cdot (p(\alpha) - p_0) \quad \text{Plane Equation} \]
The line equation specifies a point \( p(\alpha) \) on the line segment bounded by the two endpoints \( p_1 \) and \( p_2 \). As \( \alpha \) varies from zero to one, the point \( p(\alpha) \) moves from \( p_1 \) to \( p_2 \). The plane is defined by the two orthogonal vectors \( n \) and \( (p(\alpha) - p_0) \), where \( n \) is the normal vector of the plane (Figure 3.19). Solving the equations for \( \alpha \) yields:

\[
\alpha = \frac{\vec{n} \cdot (p_0 - p_1)}{\vec{n} \cdot (p_2 - p_1)} \tag{3.18}
\]

Using \( \alpha \) as input for the line equation finds the point of intersection. It replaces the point that was previously outside of the view volume.

The Cohen-Sutherland algorithm only works with an orthogonal view volume and polygons that are convex. Since all primitive polygons in OpenGL are convex, this limitation poses no problem. The technique of projection normalization (Section 3.4.4) satisfies the prerequisite of an orthogonal view volume.

Other clipping algorithms such as the Liang-Barisky and the Sutherland-Hodgeman algorithms are described in [Ang03].
3.5.2 Hidden Surface Removal Algorithms

Once all objects have been clipped, the graphic system has to identify those parts of an object that are visible to the viewer and not obscured by other objects.

OpenGL uses the z-buffer algorithm to perform this task. It is the most widely used hidden surface removal algorithm because it is easy to implement and compatible with the pipeline architecture described in section 3.1.3. The algorithm uses an additional buffer, the z-buffer, which has the same number of elements as the frame buffer. It stores the shortest distance for each pixel to the corresponding point on an object. The cost for the additional memory required to allocate the z-buffer is compensated for by the simplicity of the algorithm. It executes as follows (Figure 3.20) [Ang03]:

![Figure 3.20: z-Buffer Algorithm](image)

Initially, each element in the z-buffer is initialized to the maximum distance an object can have from the view plane. Processing one polygon at a time, the algorithm then calculates the distance from the view plane to a point on the polygon. This point corresponds to the intersection of the polygon with a projection ray. If the distance is less than the distance of a
previous polygon, the element in the z-buffer is updated with this new
distance.

Most graphic systems combine the hidden surface removal algorithm with
the rasterizing process to gain a performance increase [Ang03].

3.6 Lighting
The interaction between light and the surface of an object influences the
color perception of a viewer. Light makes objects visible and produces
different shades of color depending on the material properties of an
object. These shades of color give a two-dimensional image the
appearance of being three-dimensional and thus are an important part of
a graphic system [Ang03].

To get an overview of how a graphic system determines the color of an
image consider figure 3.21.

![Figure 3.21: Light, Surfaces and Computer Imaging](image)

It shows the view plane that is divided into equally sized rectangles. Each
rectangle corresponds to a pixel in the frame buffer. Rays of light
emanating from a light source to points on an object are reflected. The
color of the reflected light that reaches the COP determines the color of
the pixels. If a ray of light intersects directly with the view plane, the color of the light source defines the color at the point of intersection [Ang03].

### 3.6.1 Light Sources and Material Properties

In a graphic system, the rays of light consist of three primary colors - red, green and blue (RGB). When a light ray strikes an object some of the color components are reflected and the others are absorbed. The fraction of the reflected light that reaches the COP defines the color of an object. Hence, what is often referred to as the color of an object, is actually the reflection coefficient for each primary color [Ang03].

A light source creates the light rays that are necessary for the visibility of an object. OpenGL supports four different types of light sources. The characteristics for each light source are described in the following paragraphs [Ang03]:

**Ambient Light**: Ambient light comes from all directions and provides a uniform illumination for the entire scene. It is the result of the secondary interactions of light between the objects in a scene. Every object receives the same amount of ambient light regardless of its position and orientation.

**Point Sources**: A point source radiates light equally in all directions. The illumination that an object receives from a point source depends primarily on its distance from the light source and its orientation. An example of a point source is a light bulb.

**Spotlights**: A spotlight is characterized by a narrow distribution of light. The area of illumination is comparable to a cone having the most light concentrated at the center of the cone. Only objects within this cone of
light interact with the light rays. Examples of a spotlight are the headlights of a car.

**Distant Light Sources:** The direction of the light rays that strike an object from a distant light source is the same for all points on the object. These parallel light rays are comparable to light from the sun. Therefore, distant light sources can be used to simulate outdoor scenes.

As mentioned previously, the material properties of an object determine how light is reflected. OpenGL supports three different types of materials that have the following characteristics [Ang03]:

**Specular Surfaces:** Specular surfaces appear shiny because most of the light is reflected at a single angle. A perfectly specular surface obeys the rule that the angle of incidence of a light ray is equal to the angle of reflection. Mirrors belong to the category of perfectly specular surfaces.

**Diffuse Surfaces:** Diffuse surfaces scatter light in all directions. The diffuse reflection that reaches the COP defines the primary color of an object. Dull or matte surfaces such as walls covered with flat paint are called diffuse reflectors. The brightness of the reflected light depends on the position of the light source and the orientation of the object.

**Translucent Surfaces:** Translucent surfaces allow some light to penetrate an object. This refraction is characteristic for materials such as glass or water. To simulate transparency, OpenGL extends the RGB-color model with an additional color component, the alpha value (RGBA). If the alpha value is set to one, the surface of an object is fully opaque.
Choosing a suitable light source and the appropriate materials for the objects in a scene is crucial to produce the desired appearance of an image. The RGB-color model provides the foundation for all color computations in a graphic system. It matches most of the colors in the visible spectrum and it is consistent with the color model of the human visual system [Ang03].

3.6.2 The Phong Lighting Model

In reality, the interaction of light with objects in a scene is very complex and not practical for the reproduction in a computer environment. Instead, graphic systems have to rely on approximations to create an image with an acceptable appearance [Ang03].

The Phong lighting model provides the most commonly used approximation formula. It calculates the color intensity of a point with the aid of the four vectors shown in figure 3.22.

![Figure 3.22: Vectors used by the Phong Lighting Model](image)

The vector $\mathbf{n}$ is the surface normal at an arbitrary point $\mathbf{p}$. The vector $\mathbf{v}$ points in the direction of the viewer (COP). The vector $\mathbf{l}$ points in the direction of the light source and the vector $\mathbf{r}$ defines the direction that a perfectly reflected ray from $\mathbf{l}$ would take [Ang03].

All these vectors reappear in the equation of the Phong lighting model:

$$I = \frac{1}{a + bd + cd^2} (k_d L_d \mathbf{l} \cdot \mathbf{n} + k_s L_s (r \cdot \mathbf{v})^\alpha) + k_a L_a$$

(3.19)
A detailed derivation of the above equation is described in [Ang03]. However, the basic understanding of the individual terms of the equation is outlined in the following paragraph.

The first term of the equation describes the attenuation factor of the light intensity based on the distance \( d \) from a light source to a point on the surface. \( L_d, L_s \) and \( L_a \) are the diffuse, specular and ambient components of a light source while \( k_d, k_s \) and \( k_a \) refer to the reflection coefficients of the surface. The exponent \( \alpha \) is a shininess coefficient that produces a mirror like surface as \( \alpha \) goes to infinity [Ang03].

To find the color of a point, the equation has to be computed for each primary color and all the light sources in a scene [Ang03].

Nischwitz and Haberäcker [NH04] suggest the values for the reflection coefficients (\( k_a, k_d, k_s \)) and the shininess exponent (\( \alpha \)) for a number of real materials. The following table lists a few examples:

<table>
<thead>
<tr>
<th>Material</th>
<th>( R_a, G_a, B_a, A_a )</th>
<th>( R_d, G_d, B_d, A_d )</th>
<th>( R_s, G_s, B_s, A_s )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black Plastic</td>
<td>.00, .00, .00, 1.0</td>
<td>.01, .01, .01, 1.0</td>
<td>.50, .50, .50, 1.0</td>
<td>32.0</td>
</tr>
<tr>
<td>Brass</td>
<td>.33, .22, .03, 1.0</td>
<td>.78, .57, .11, 1.0</td>
<td>.99, .94, .81, 1.0</td>
<td>27.9</td>
</tr>
<tr>
<td>Bronze</td>
<td>.21, .13, .05, 1.0</td>
<td>.71, .43, .18, 1.0</td>
<td>.39, .27, .17, 1.0</td>
<td>25.6</td>
</tr>
<tr>
<td>Copper</td>
<td>.19, .07, .02, 1.0</td>
<td>.70, .27, .08, 1.0</td>
<td>.26, .14, .09, 1.0</td>
<td>12.8</td>
</tr>
<tr>
<td>Gold</td>
<td>.25, .20, .07, 1.0</td>
<td>.75, .61, .23, 1.0</td>
<td>.63, .56, .37, 1.0</td>
<td>51.2</td>
</tr>
<tr>
<td>Silver</td>
<td>.19, .19, .19, 1.0</td>
<td>.51, .51, .51, 1.0</td>
<td>.51, .51, .51, 1.0</td>
<td>51.2</td>
</tr>
</tbody>
</table>

Table 3.1: Phong Parameters for various Materials

Although the Phong lighting model fits well in the pipeline architecture of OpenGL, it poses some limitations. Global lighting effects such as shadows, reflections on shiny surfaces or the blockage of light cannot be handled by the lighting model due to the independent processing of
polygons in the graphic pipeline (Figure 3.23(b)). Ray tracing is one alternative approach that can handle these global effects (Figure 3.23(a)). However, this algorithm is very computing intensive [Ang03].

\[\text{Figure 3.23: Blockage of Light with (a) Ray Tracing (b) Phong Lighting}\]

3.6.3 Shading Techniques

The shading techniques described in this section all use the Phong lighting model to perform their color computations. They differ in how often they perform the color computations and in how they determine the orientation of a polygon [Ang03].

**Flat Shading**: Flat shading uses the normal of a polygon to determine its orientation. Since flat shading performs the color computation of a polygon only once, all pixels inside the polygon receive the same color. Different polygons of the same object may obtain a different color depending on their orientation and the position of the light source.

The facet-like appearance of an object is characteristic for flat shading. The small differences in color along the edges of adjacent polygons are reinforced by the human visual system that reacts extremely sensitive to these color differences [NH04].

OpenGL uses the following function to enable flat shading:

- `glShadeModel(GL_FLAT);`
**Gouraud Shading:** In contrast to flat shading, the Gouraud shading algorithm performs a color computation for each polygon vertex. Instead of using the polygon normal to compute the color of a vertex, Gouraud shading uses a vertex normal – the normalized average of all polygon normals that share the same vertex (Figure 3.24). The algorithm then interpolates the computed shades of color from one vertex to another and fills the interior points of a polygon using the same linear interpolation.

\[ n = \frac{n_1 + n_2 + n_3 + n_4}{|n_1 + n_2 + n_3 + n_4|} \]  

(Figure 3.24: Calculation of a Vertex Normal)

Gouraud shading eliminates the color differences of adjacent polygons that arise with flat shading. However, highlights in the middle of a polygon are not visible because Gouraud shading only computes the color at each vertex. Nevertheless, due to the low complexity of the algorithm, it is the dominant technique used in today’s graphic systems.

In OpenGL, Gouraud shading is enabled by the following function:

- `glShadeModel(GL_SMOOTH);`

**Phong Shading:** Instead of interpolating the colors across the surface of a polygon, the Phong shading algorithm interpolates the vertex normals. In doing so, a normal for each pixel is generated. Using these normals, the Phong shading algorithm then calculates a color for each pixel.
Phong shading produces smoother renderings than those of Gouraud shading. In addition, it eliminates the highlight problem of Gouraud shading. However, due to the significantly greater computational cost, OpenGL provides no means for specifying this shading technique.

### 3.7 Textures

Texture mapping is another way of defining the colors of an object. The literature distinguishes between two-dimensional and three-dimensional texture mapping. Since OpenGL ES does not support the latter, this section focuses on two-dimensional texture mapping. An illustration of three-dimensional texture mapping can be found in [Wol00].

Two-dimensional texture mapping is a simple way of adding realism to an otherwise sterile-looking scene. During the texture mapping process, a pre-existing image is pasted onto the surface of a geometric object. The image that is usually read from a file provides the color information that is stored in a byte array called the “texture map” [Ang03].

OpenGL uses two coordinates, s and t, to specify the indices of the texture map. These texture coordinates range within the interval zero and one. A value of (0.0, 0.0) corresponds to the first element in the array whereas a value of (1.0, 1.0) corresponds to the last element in the array [Ang03].
Figure 3.25 shows how texture coordinates are mapped to unique pixels in a 512 x 512 image.

![Figure 3.25: Mapping of Texture Coordinates](image)

Similar to the color computations of Gouraud shading, the texture coordinates are specified for each polygon vertex. The difference is that the application programmer has to specify the texture coordinates. OpenGL can also generate texture coordinates automatically but this feature is not supported in OpenGL ES [Ang03].

To find the texture coordinates for the interior points of a polygon, OpenGL interpolates the existing texture coordinates across the surface of a polygon, just as it interpolates the colors in Gouraud shading. The resulting texture coordinates are then used as indices for the texture map to determine the corresponding colors [Ang03].

Unfortunately, two-dimensional texture mapping is subject to a problem known as aliasing. Aliasing is a term used to describe the errors in a computer-generated image (e.g. jagged lines or moiré patterns) that occur during the process of mapping a continuous set of values to a discrete approximation of those values. In the context of texture mapping, these aliasing errors arise during the mapping of texture coordinates to indices of a texture map [Ang03].
Since texture coordinates are continuous floating-point values, they can rarely be mapped directly to a discrete index of a texture map. One option is to use the color at the index that is closest to the texture coordinates. This option is called point sampling but it is most susceptible to aliasing errors. A better alternative is to use a weighted average of a group of colors that surround the index determined by point sampling. This option is known as linear filtering [Ang03].

Another source for aliasing errors is the difference in resolution between the frame buffer and a texture map. The size of a pixel in a texture map may be smaller (minification) or larger (magnification) than a pixel in the frame buffer. In both cases, linear filtering can be used to obtain a less aliased image. A technique known as mipmapping provides an even better way of dealing with the minification problem. For it, a series of texture maps is required at reduced sizes. The application programmer or a utility function of OpenGL can create these additional texture maps. Once the texture maps are set up, OpenGL then automatically uses the appropriate size during the texture mapping process [Ang03].
4 Implementation in C++ with OpenGL ES

This chapter provides an insight into the design and implementation of the 3D Visualization Tool. It describes the research platform and the digital map that was used during the development and highlights NAVTEQ’s current activities regarding the collection of three-dimensional map data.

4.1 Intention of Implementation

The 3D Visualization Tool was developed in cooperation with NAVTEQ to visualize three-dimensional map data. It simulates a journey through a road network with selected POIs placed on both sides of the street. Integrated into the NAVTEQ ADASRP (Advanced Driver Assistance System Research Platform), the 3D Visualization Tool serves mainly for presentation purposes to NAVTEQ customers. It was implemented with the C++ programming language and currently runs on the Windows 2000/XP operating system. However, the design permits an easy migration to an embedded system. With a conformant OpenGL ES graphics library and a suitable data access library (DAL) for digital maps, the 3D Visualization Tool should be able to run on any embedded system if it satisfies the system requirements outlined in section 2.2.

4.2 NAVTEQ ADASRP

The NAVTEQ ADASRP, from now on just called ADASRP, is a Windows-based framework application for prototyping ADAS solutions (Figure 4.1). Their goal is to assist the driver in the driving process and to increase road safety. Since the development of ADAS applications is growing, more ADAS products will become available in the next several years. The following list outlines a few already existing ADAS applications:
Implementation in C++ with OpenGL ES

- Adaptive Cruise Control (ACC)
- Intelligent Speed Advice/Adaptation (ISA)
- Adaptive Front Light Control
- Collision Warning System

ADAS applications use a data model to represent the map data around the current vehicle position. This so-called “ADAS Horizon” is in the process of becoming an open standard. In addition, a standardized interface specification is being developed that enables the access of this map data. These and other activities are outlined in the MAPS&ADAS project definition. NAVTEQ, as a major supplier of digital maps, supports this project [MAA04].

Figure 4.1: NAVTEQ ADASRP
4.2.1 Main Components of ADASRP

This section describes the main components of ADASRP and how they interact with each other. Some of the components can be replaced with other third-party modules if they implement NAVTEQ’s MASSIVE (Map Access Server Specification for In-Vehicle Environments) interface. The interface encapsulates the inter-process-communication by means of the publisher/subscriber model [ARP05].

Unified Data Access Layer: The Unified Data Access Layer (UDAL) provides a high-level interface for other ADASRP components to access digital map data. Its underlying implementation utilizes lower-level data access libraries that currently support NAVTEQ’s Mobile Map and Drive Map Format. However, UDAL can be easily extended to support other Physical Storage Formats (PSFs). The 3D Visualization Tool uses UDAL to gain access to the road network. Hence, it is described in more detail in Section 4.2.2.

Sensor Data Collection Engine: The Sensor Data Collection Engine provides the interface between ADASRP and NAVTEQ’s sensor box. The sensor box consists of a GPS receiver, a gyroscope and other modules that connect to the electronic system of a car (odometer, reverse drive indicator, etc.). It collects the data from the hardware sensors and forwards it to the Vehicle Positioning Engine. ADASRP can also use other third-party collection software if it implements the MASSIVE interface.

Vehicle Positioning Engine: The Vehicle Positioning Engine uses the sensor data and the digital map to calculate the exact car position. A map-matching algorithm compares the raw positional data against the road network thereby increasing the precision of the sensor data. The output generated by the Vehicle Positioning Engine includes the
latitude/longitude coordinates of the current car position and the direction of the car measured as an angle against North. These values provide the input for the 3D Visualization Tool. Again, any other vehicle-positioning module can be used if it implements the MASSIVE interface.

**ADAS Horizon Engine:** The ADAS Horizon Engine integrates the current car position with the digital map data to form a data structure that describes the road network around the current vehicle position. It is generated with a probabilistic algorithm that creates a tree of links. A link represents a street segment between two crossings. Starting at the current root link, each child link is assigned a probability based on user-defined criteria. Continuing in depth-first order, the tree is expanded until the termination criteria of the algorithm are satisfied. The most probable path (MPP) defines the path that starts at the root link and follows the child links with the highest probability. The built-in ADAS Horizon Engine can be replaced with other third-party engines that implement the MASSIVE interface.

**Map Display:** The two-dimensional map display visualizes the road network including the ADAS Horizon. If the Vehicle Positioning Engine is active, the map also shows the current position of the car. Currently, only NAVTEQ’s Drive Map Format can be displayed. However, this will change in a future release. The user interactions supported by the map display are described in [ARP05].

**Plug-In Environment for ADAS Applications:** The plug-in environment of ADASRP is designed for a fast development of ADAS applications. They are built as dynamic linked libraries (DLLs) and are loaded at runtime. Each plug-in occupies one tab in one of the three plug-in windows. Through appropriate listener functions, the plug-ins receive
sensor, vehicle-positioning and ADAS Horizon data. In addition, they can interact with the map display. The ADASRP 3D Plug-In was developed to serve as a client application for the 3D Visualization Tool (Section 4.4).

4.2.2 Unified Data Access Layer

For many years now, the manufacturers of navigation systems create custom PSFs that target the specific requirements of their navigation software. The Mobile Map Format (MMF) developed by NAVTEQ is one example (Section 4.2.3). Since an agreement on a uniform storage format is not in sight, the goal of UDAL is to provide a standardized API for accessing digital map data that is most common to all PSFs. This section describes the UDAL interface and highlights the relevant classes utilized by the 3D Visualization Tool [ARP05].

The UDAL interface consists of an abstract DAL class and various other classes that are grouped into the following three categories:

1. Base Classes: The base classes are virtual C++ or template classes that assist other UDAL classes. The ones outlined here are “DALObject”, “DALObject_var” and “RealArray”:

   The DALObject serves as a base class for most UDAL classes. It supports operators for assignment and equality checking. In addition, DALObjects can be stored in and retrieved from an archive.

   The DALObject_var template class serves as a wrapper around pointers to other UDAL objects. A wrapped object behaves like a regular variable and supports operators for assignment and equality checking.
The `RealArray` template class can be used to create an array of UDAL objects. It supports the access of individual elements in the array.

2. Static Types: The static types are C++ classes that represent spatial entities, which are common to all PSFs. The most frequently used classes are "GeoPosition", "ShapePoint" and "GeoRectangle":

A `GeoPosition` is a simple representation of a latitude/longitude coordinate pair. It can be used to specify a point on the digital map.

A `ShapePoint` is an extension of a `GeoPosition`. It defines a point on a street segment where the direction of the street changes.

A `GeoRectangle` defines a rectangular area. Four `GeoPositions` specify the corners of the rectangle.

3. Dynamic Types: The dynamic types are virtual C++ classes that represent spatial entities, which may be incompatible between PSFs. The classes relevant for the 3D Visualization Tool are "Link", "Poi" and "MapArea":

A `Link` defines a street segment between two nodes. It stores all the attributes of a street segment (Speed Limit, Number of Lanes, etc.). Each link can have several `ShapePoints`.

A `Poi` defines a landmark on the digital map. Its attributes include the name, the type and the geographic position of the landmark.

A `MapArea` specifies a rectangular region on the digital map. It provides the functions for retrieving spatial data as described below.
In addition to the dynamic types specified above, every UDAL adapter must implement the DAL interface. Next to functions that are required for opening and closing a digital map, the DAL interface also provides the functionality for retrieving spatial data. For example, the `newMapArea()` function creates a `MapArea` object based on a user-defined `GeoRectangle` (Listing 4.1).

```cpp
106   /**
107   * Implementation of DAL::newMapArea().
108   * @see DAL::newMapArea()
109   */
110   virtual MapArea* newMapArea(const GeoRect& rect);
```

**Listing 4.1: DAL::newMapArea()**

It in turn provides the two functions `getLinks()` and `getPois()` (Listing 4.2).

```cpp
57   /**
58   * Implementation of MapArea::getLinks() method.
59   * @see MapArea::getLinks()
60   */
61   virtual Sint32 getLinks(Sint32 nLevel, bool bClip,
62     void* pAppData, MapAreaCB& cb);
63
64   /**
65   * Implementation of MapArea::getPois() method.
66   * @see MapArea::getPois()
67   */
68   virtual Sint32 getPois(Sint32 nLevel, Sint32 poiType, bool bClip,
69     void* pAppData, MapAreaCB& cb);
```

**Listing 4.2: MapArea::getLinks() and MapArea::getPois()**

The function `getLinks()` searches the digital map for all links within the map area. Once a link is found, it is passed to a callback function for further processing. Additional data can be passed to the callback function through the `pAppData` parameter. The parameter `nLevel` specifies which map layer should be searched and the parameter `bClip` indicates whether the links that cross the map area boundaries should be clipped or not. An implementation of a callback class must inherit the `MapAreaCB` interface.

The function `getPois()` behaves similar. It searches for POIs that are of the specified `poiType`. 

50
The 3D Visualization Tool implements a callback class to create a three-dimensional representation of the road network via the `getLinks()` function (Section 4.5.4). The `getPois()` function is not being used because the current MMF offers no three-dimensional data for POIs.

### 4.2.3 Mobile Map Format

The MMF was developed by NAVTEQ to create a highly efficient and simple PSF. On the one hand, the PSF should be small in terms of the required storage space. On the other hand, it should permit a fast access to the map content [MMF04].

To achieve these goals, the digital map is divided into layers. The purpose of layering is to group the spatial data in a way that avoids the loading of unnecessary information. For example, lakes and forests are not important to calculate a route, but a map display would show this information. Hence, the MMF distinguishes between road network and carto layers [MMF04].

Table 4.1 shows the different road types as they appear in the road network layers.

<table>
<thead>
<tr>
<th>Road Network Layer</th>
<th>Map Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Interstate Highways</td>
</tr>
<tr>
<td>1</td>
<td>State Highways</td>
</tr>
<tr>
<td>2</td>
<td>County Roads</td>
</tr>
<tr>
<td>3</td>
<td>City Roads</td>
</tr>
<tr>
<td>4</td>
<td>Minor Roads</td>
</tr>
</tbody>
</table>

Table 4.1: MMF Road Network Layers
The information contained in the carto layers is grouped into three categories [MMF04]:

1. Areas: Include lakes, forests, built-up areas, etc.
2. Lines: Include borders, railroad tracks, canals, etc.
3. POIs: Include city centers, restaurants, airports, etc.

Table 4.2 shows the map content of the different carto layers as they exist in the MMF.

<table>
<thead>
<tr>
<th>Carto Layer</th>
<th>Map Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Areas, Lines and POIs displayed at a scale 1:10,000,000</td>
</tr>
<tr>
<td>1</td>
<td>Areas, Lines and POIs displayed at a scale 1:1,000,000</td>
</tr>
<tr>
<td>2</td>
<td>Areas, Lines and POIs displayed at a scale 1:500,000</td>
</tr>
<tr>
<td>3</td>
<td>Areas, Lines and POIs displayed at a scale 1:100,000</td>
</tr>
<tr>
<td>4</td>
<td>Areas, Lines and POIs displayed at a scale 1:50,000</td>
</tr>
<tr>
<td>5</td>
<td>Areas, Lines and POIs displayed at a scale less than 1:50,000</td>
</tr>
<tr>
<td>6</td>
<td>Minor POIs</td>
</tr>
<tr>
<td>10</td>
<td>Overview layer containing: Oceans, Country Borders, State Borders, Major Lakes, Major Roads and City Centers</td>
</tr>
</tbody>
</table>

The MMF further subdivides each layer into equally sized tiles. However, the tiles in the upper layers are usually larger than the ones in the lower layers. The purpose of tiling is to permit a quick data transfer from one location to another. This is useful for distributed architectures and on-the-fly map updates. It also increases the performance when accessing the spatial data [MMF04].
Each tile is stored in a separate physical entity. This can be a row in a database table or a single binary file. To obtain a continuous road network, the tiles in the road network layers keep a reference to street nodes and street segments across tile boundaries (Figure 4.2). In contrast, no references are maintained between tiles in the carto layers [MMF04].

![Figure 4.2: MMF Tile References](image)

Also part of the MMF are gazetteers. They provide the geocoding information and thus can be thought of as a geographical dictionary. Combined with a next character search interface, gazetteers are a useful tool for searching a digital map.
Four different types of gazetteers exist in the MMF. They are ordered hierarchically as described below [MMF04]:

1. Country Gazetteer: Contains the geocoding information for all countries in a digital map. Each country includes a reference to the corresponding town gazetteer.

2. Town Gazetteer: Contains the geocoding information for all cities in a country. Each city includes a reference to the corresponding street gazetteer.

3. Street Gazetteer: Contains the geocoding information for all streets in a city.

4. POI Gazetteer: Contains the geocoding information for all POIs in a city.

4.3 NAVTEQ 3D Specification

Currently, NAVTEQ is developing a specification on how to store three-dimensional structures in the NAVTEQ GDF (Geographic Data Files). It distinguishes between varying levels of detail, which are [GDF05]:

- 2D Polygons
- 2D Polygons with Height (2.5D)
- 2D Polygons with Height and Textures (2.5D)
- 3D Structure Models in two levels of detail with Textures
- 3D Structure Models in two levels of detail with Textures and a 2.5D icon
Implementation in C++ with OpenGL ES

Unfortunately, the current compiler that converts the text-based GDF into the binary MMF does not support these new additions. However, some 3D structure models are available that the 3D Visualization Tool can display. These models are landmarks of major cities in Europe and the United States of America. A 3D artist generated the models based on reference material that was collected by NAVTEQ field employees.

4.3.1 Generation of 3D Structure Models

This section describes the process of collecting the reference material that is necessary to create a 3D structure model. The reference material includes building dimensions and various images [Hop05].

A rangefinder is used to acquire the most important dimensions of a building. These dimensions usually consist of the building’s height and width. However, it may be necessary to capture additional height information for more complex structures, e.g. buildings with domes, arches or more than one rooftop. The objective is to collect enough information so that the 3D artist can determine the remaining dimensions from the digital images.

Digital images are necessary to create the textures for a 3D structure model. They also assist the 3D artist when creating a 3D model. The images are taken with a standard digital camera having a minimum resolution of 1024 x 768 pixels. Several images are required for each building including an image of the entire front of the building, an image of the entire side of the building and a close-up of the building’s primary window. To be useful for texturing, these images should have the following characteristics:
Implementation in C++ with OpenGL ES

- They should lack extraneous information (e.g. cars, trees and people).
- They should be light source independent (i.e. no shadows).
- They should have no or little perspective.

In addition to the images taken with the digital camera, a 3D artist requires aerial images to get an idea of the building’s footprint. These images provide information about the roof design and are useful for determining the building’s width. Several vendors offer aerial images. NAVTEQ obtains their aerial images from GlobeXplorer [GXP05].

Once all the reference material is collected, it is forwarded to a 3D artist who creates the 3D model. Figure 4.3 and 4.4 show some reference material used for a structure in San Francisco.

Figure 4.3: Aerial Image of a Structure in San Francisco
4.4 3D Visualization Concept

The UML (Unified Modeling Language) class diagram outlines the 3D visualization concept by illustrating the most important classes and interfaces along with their relationships (Figure 4.5).

Since the 3D Visualization Tool produces no visual output by itself, a client application is necessary to trigger the rendering of a three-dimensional scene. As mentioned previously, the 3D Plug-In was developed for this purpose. No further details regarding the implementation of the 3D Plug-In are explained here. Instead, please refer to the source code on the enclosed CD-ROM.

The remaining classes and interfaces are described in more detail in section 4.5.
Figure 4.5: UML Class Diagram - 3D Visualization Tool
The UML sequence diagram illustrates the sequence of actions that occur for every vehicle-positioning (VP) message, which is provided by the ADASRP VP Engine (Figure 4.6). Each VP message contains information about the location and the heading of the car. The 3D Plug-In uses this information to set the corresponding member variables of the 3D manager. A call to `drawFrame()` causes a series of subsequent function calls to generate a three-dimensional image of the world in front of the car. This process is described in the following paragraph.

First, a search box (Section 4.5.3) is created. It is used to locate all street segments and landmarks around the current vehicle position. Upon completion, the object locator (Section 4.5.4) returns a linked list of references to the spatial data that was found during the search. The object loader (Section 4.5.5) retrieves this data and returns the resulting 3D models to the 3D manager. The 3D manager (Section 4.5.2) then draws the models to the screen and finally resets the origin of the coordinate system to the current vehicle position.
4.5 Class and Interface Definitions

The following sections describe the definitions of classes that were used to implement the 3D Visualization Tool. The complete source code of these classes can be found on the enclosed CD-ROM.

Several of the classes shown in figure 4.5 were derived from base classes to ensure the extensibility of the 3D Visualization Tool. Each base class contains a static factory method to create the appropriate instance of a class. Listing 4.3 shows a sample implementation for the creation of a CSimpleLocator instance.

```cpp
15 /**
16   * Creates an object locator.
17   *
18   * @param locator Type of locator to create.
19   *
20   * @returns Pointer to object locator or 0 on error.
21   */
22 I3DObjectLocator* I3DObjectLocator::create3DObjectLocator( 23 const char* locator) 24 { 25     if (strcmp(locator,"SimpleLocator") == 0) 26         { 27             return new CSimpleLocator; 28         } 29     return 0; 30 }

Listing 4.3: Implementation of a Static Factory Method
```

4.5.1 Representation of a 3D Model

One of the functional requirements specified in Section 2.2 is the visualization of 3D models. A data structure was developed to store the information of these models (Listing 4.4). It contains several member variables including: An array of vertices, an array of vertex normals, an array of texture coordinates and an array of vertex colors. They are necessary so that OpenGL can draw a textured or a colored 3D model onto the screen.
A 3D model may be composed of several groups if, for example, the model has more than one texture. An array of groups is stored in the t3DModel class. A t3DGroup contains an array of indices to the vertex array and an index to the respective material. A t3DMaterial holds the color information of a group and a pointer to an optional texture map.

Several utility functions are part of the t3DModel class. For example, it provides a function to calculate the vertex normals used for shading and a function to create an array of vertex colors. Another function calculates the lower-left corner and the upper-right corner of the model's bounding box.

During the implementation of the 3D Visualization Tool, it turned out that the data structure described above is able to store the information of a road network and other 3D structures. Further investigation is necessary to determine if the data structure is sufficient for three-dimensional terrain data.
4.5.2 3D Manager

The C3DManager class is the core of the 3D Visualization Tool and at the same time it serves as the interface for a client application (Listing 4.5). Before a client application can start using the interface, it must initialize the 3D manager via the init3DManager() function. This will set up the
rendering context and initialize the OpenGL state machine. At present, default values for shading and texturing are used that generate an image with the highest quality. However, if performance is an issue it should be possible to change these values interactively. Currently, the 3D Visualization Tool does not support this feature.

Several member variables are part of the 3D manager. The most important are the EGL resources (Section 3.1.2), a pointer to the UDAL interface (Section 4.2.2) and pointers to the classes shown in the UML class diagram (\texttt{I3DSearchBox}, \texttt{I3DObjectLocator} and \texttt{C3DObjectLoader}). The latter are explained in more detail in the following sections.

In addition, the \texttt{C3DManager} class maintains member variables for the camera parameters. A client application can modify these parameters with the two functions: \texttt{zoom()} and \texttt{setFirstPerson()}. Depending on the \texttt{zoomFactor}, the first function zooms either away from or towards the car. The second function switches from a bird’s eye view to a cockpit view and vice versa.

The current vehicle position and its orientation is modified via the \texttt{setCurrentPosition()} and \texttt{setCurrentOrientation()} functions. A client application should call these functions prior to calling \texttt{drawFrame()} to simulate a driving car. A coordinate transformation is necessary to represent the position of the car in Cartesian coordinates. The equations 3.5 and 3.6 perform this calculation (Section 3.3.3).

The 3D manager also keeps a draw count for every call to \texttt{drawFrame()}. It uses the draw count to search for 3D models only every tenth time. This was a design choice to increase performance. The client application
should have an option to change this setting. However, this feature is currently not supported.

```
24 /**<
25  * Utilizes OpenGL ES to display 3D Data.
26  */
27 class C3DManager
28 {
29
30     friend class CSearchBoxAroundFov;
31     friend class CSearchBoxAroundCar;
32
33 private:
34
35     /** Display on which graphics are drawn. */
36     EGLDisplay  m_EGLDisplay;
37     /** Configuration for EGL Surface. */
38     EGLConfig   m_EGLConfig;
39     /** Rendering context. */
40     EGLContext  m_EGLContext;
41     /** Drawing window. */
42     EGLSurface  m_EGLWindowSurface;
43
44     /** Counter for draw events. */
45     int m_DrawCount;
46     /** True if zooming. */
47     bool m_bZooming;
48     /** True if in cockpit view. */
49     bool m_bFirstPerson;
50     /** Elevation from the ground for cockpit view. */
51     float m_EyePosition;
52     /** Field of view angle in the y-direction. */
53     float m_Fovy;
54     /** Distance to clipping planes. */
55     float m_NearPlane, m_FarPlane;
56     /** Width and height of window. */
57     int m_WidthWindow, m_HeigthWindow;
58     /** Current orientation against north. */
59     float m_CurrentOrientation;
60     /** Current latitude / longitude. */
61     int m_CurrentLat, m_CurrentLong;
62     /** Latitude / Longitude for the coordinate system's origin. */
63     int m_OriginLat, m_OriginLong;
64     /** XYZ camera position. */
65     float m_CameraX, m_CameraY, m_CameraZ;
66     /** XYZ car position. */
67     float m_CarX, m_CarY, m_CarZ;
68     /** XYZ lookAt position. */
69     float m_LookAtX, m_LookAtY, m_LookAtZ;
70     /** Camera distance to car in meters. */
71     float m_CameraDistanceToCar;
72     /** XYZ unit vector of current orientation. */
73     float m_DeltaX, m_DeltaY, m_DeltaZ;
74
75     /** Pointer to DAL object. */
76     UDAL::DAL* m_pDAL;
77     /** Pointer to 3D Visualizer PlugIn. */
78     C3DVisualizer* m_p3DVisualizer;
79
80     /** Pointer to 3D street model. */
81     t3DModel* m_p3DStreets;
82     /** Pointer to 3D MPP (most probable path) model. */
83     t3DModel* m_p3DMPP;
84
85     /** Pointer to 3D car model. */
86
87 }
```
t3DModel* m_p3DCar;

/** Pointer to searchbox creator. */
I3DSearchBox* m_pSearchBoxCreator;

/** Pointer to 3D object loader. */
C3DObjectLoader* m_pObjectLoader;

/** Pointer to 3D object locator. */
I3DObjectLocator* m_pObjectLocator;

/** Pointer to current list element of 3D models. */
tListElement* m_pFirstListElement;

private:

/**
 * Initializes OpenGL state machine.
 */
void initGL(void);

/**
 * Initializes EGL resources.
 * @param hWnd Properly initialized window handler.
 */
void initEGLResources(void* hWnd);

/**
 * Destroys EGL resources.
 */
void destroyEGLResources(void);

/**
 * Draws 3D model to the current rendering context.
 * @param p3DModel Properly initialized 3D model.
 * @param posX    X-Position of 3D model.
 * @param posY    Y-Position of 3D model.
 * @param posZ    Z-Position of 3D model.
 * @param rotX    X-Rotation of 3D model.
 * @param rotY    Y-Rotation of 3D model.
 * @param rotZ    Z-Rotation of 3D model.
 */
void draw3DModel(t3DModel* p3DModel, float posX, float posY, float posZ, float rotX, float rotY, float rotZ);

/**
 * Resets the origin of the coordinate system
 * to the current latitude / longitude values.
 * Will adjust the car, camera, and lookAt position accordingly.
 */
void resetCoordinateSystem(void);

public:

/**
 * Default constructor.
 * Object is invalid until init3DManager() is called.
 */
C3DManager(void);

/**
 * Default destructor.
 * destroy3DManager() should be called before object is deleted.
 */
virtual ~C3DManager(void);

/**
 * Initializes 3D manager with default values.
 * Initializes OpenGL state machine and EGL resources.
 *
Implementation in C++ with OpenGL ES

```c
    154     * @param hWnd         Properly initialized window handler.
    155     * @param widthWindow  Width of the drawing window.
    156     * @param heigthWindow Heigth of the drawing window.
    157     * @param plugIn       Pointer to 3D Visualizer PlugIn.
    158     */
    159     void init3DManager(void* hWnd, int widthWindow, int heigthWindow,
                            void* plugIn);

    160
    161     /**
    162     * Deletes all 3D models and destroys EGL resources.
    163     */
    164     void destroy3DManager(void);

    165
    166     /**
    167     * Adjusts the projection matrix for new window dimensions.
    168     * @param widthWindow  Width of the drawing window.
    169     * @param heigthWindow Heigth of the drawing window.
    170     */
    171     void resize(int widthWindow, int heigthWindow);

    172
    173     /**
    174     * Creates a searchbox to locate 3D models.
    175     * Draws all models to the rendering context.
    176     */
    177     void drawFrame(void);

    178
    179     /**
    180     * Adjusts the camera position to zoom away/towards the car.
    181     * Only works if not in first person view mode.
    182     * @param zoomFactor Zoom towards the car if positive.
    183     *                 Zoom away from the car if negative.
    184     */
    185     void zoom(int zoomFactor);

    186
    187     /**
    188     * Creates 3D model for MPP (most probable path).
    189     * @param path List of GeoPositions for MPP.
    190     */
    191     void createMPP(std::vector<UDAL::GeoPosition>& path);

    192
    193     /**
    194     * Sets the current position's latitude / longitude values.
    195     * @param latitude  Latitude in 10^7 degrees.
    196     * @param longitude Longitude in 10^7 degrees.
    197     */
    198     void setCurrentPosition(int latitude, int longitude);

    199
    200     /**
    201     * Get the current orientation.
    202     * @returns The current orientation against north.
    203     */
    204     float getCurrentOrientation(void);

    205
    206     /**
    207     * Set the current orientation.
    208     * @param angle Angle against north.
    209     */
    210     void setCurrentOrientation(float angle);

    211
    212     /**
    213     * Check if in first person view mode.
    214     * @returns True if in cockpit view mode, false otherwise.
    215     */
    216     void setCurrentOrientation(float angle);
```

66
Implementation in C++ with OpenGL ES

Listing 4.5: Definition of "C3DManager" Class

```cpp
222     */
223     bool isFirstPerson(void);
224
225     /**
226     * Set first person view mode.
227     *
228     * @param firstPersonStatus If true, cockpit view mode will be active.
229     */
230     void setFirstPerson(bool firstPersonStatus);
231
232     /**
233     * Check zoom status.
234     *
235     * @returns True if currently zooming, false otherwise.
236     */
237     bool isZooming(void);
238
239     /**
240     * Set zoom status.
241     *
242     * @param zoomStatus If true, zooming will be active.
243     */
244     void setZooming(bool zoomStatus);
245 ```

4.5.3 3D Search Box

The search box was developed to create a rectangular region around the current vehicle position. It provides the basis for locating street segments and landmarks within the specified area. Latitude and longitude coordinates specify the corners of a search box.

The I3DSearchBox interface can be used to create various search boxes. The definition of the CSearchBoxAroundCar class is shown in Listing 4.6. It is a simple example of a rectangular region covering a square kilometer around the current vehicle position. A more complex search box can be found on the CD-ROM. It covers an area around the current field of view.

During the implementation, it was determined that a region covering the field of view is not useful because no street segments or landmarks are retrieved behind the car. This causes a problem in a bird's eye view where the area behind the car is shown.
However, the concept of a search box turned out to be a good design choice because only the necessary data is loaded. In addition, its dimensions can be modified to accommodate different requirements.

```
/**
 * Creates a 3D Searchbox covering a Square Kilometer around the Car.
 */
class CSearchBoxAroundCar : public I3DSearchBox
{
 public:
  /**
   * Default constructor. Does nothing.
   */
  CSearchBoxAroundCar(void);

  /**
   * Default destructor. Does nothing.
   */
  virtual ~CSearchBoxAroundCar(void);

  /**
   * Creates a searchbox covering a square kilometer around the car.
   * @param searchBox Reference to searchbox.
   * @param pData Pointer to 3D manager to get access to private variables.
   */
  virtual void createSearchBox(UDAL::GeoRect& searchBox, 
    const void* pData);
};
```

Listing 4.6: Definition of "CSearchBoxAroundCar" Class

### 4.5.4 3D Object Locator

The object locator finds the street segments and landmarks within the specified search area. The function `locate3DModels()` was implemented to perform this task (Listing 4.7). It returns a pointer to the first element of a linked list that references the spatial data specified by `TYPE`.

As mentioned previously, the MMF does not contain any spatial data for 3D landmarks. Thus, a Microsoft Access database is used to store the positional information for selected POIs. The `CSimpleLocator` class connects to this database and loads all the information into a set of records. Next, the geographic position of all POIs is compared against the
boundaries of the search box. If one lies within the search area, it is added to the linked list.

In contrast to loading POIs, street segments are retrieved directly from the digital map. The UDAL interface is used for this purpose (Section 4.2.2). The same data structure that holds the vertices of a POI also stores the nodes and shape points of a street segment. However, these points first have to be converted into Cartesian coordinates. The equations 3.5 and 3.6 perform this calculation (Section 3.3.3).

Since the MMF does not contain enough information to represent a three-dimensional street segment, a procedure is necessary to create additional points. The callback class that was implemented for the UDAL getLinks() function performs this task. Based on a predefined lane width, it uses the bisecting line of three consecutive shape points to determine the additional points.

```cpp
/*
 * Simple 3D Object Locator.
 * Uses Access as a lookaside database to locate 3D POIs.
 */

class CSimpleLocator : public I3DObjectLocator
{
  public:
    /*
     * Default constructor.
     * Opens a connection to the lookaside database and creates a recordset.
     */
    CSimpleLocator(void);

    /*
     * Default destructor.
     * Closes previously opened recordset and database connection.
     */
    virtual ~CSimpleLocator(void);

    /*
     * Locates all 3D models within searchbox specified by TYPE.
     */
    virtual void locate3DModels(tListElement* &pFirstListElement,
                                const UDAL::GeoRect& searchBox,
                                const int TYPE);
};
```
4.5.5 3D Object Loader

The object loader provides the functionality for loading 3D models that are stored in a file (Listing 4.8). This is the case for all POIs returned by the object locator. It creates an instance of a format interpreter (Section 4.5.6) and an instance of a texture interpreter (Section 4.5.7) to interpret the file format of the 3D model and its associated texture(s).

In addition, the object loader creates an instance of an object transport. The object transport hides the details on how to access a file by offering a virtual function that reads a sequence of bytes from a file.

This design choice was made because it could be possible that in the future 3D models are stored in a database or some other location. In that case, the implementation of a different object transport ensures a quick adjustment.

```cpp
/**
 * Loads 3D Models into "t3DModel" Structures.
 */
class C3DObjectLoader
{
    public:
    /**
    * Default constructor. Does nothing.
    */
    C3DObjectLoader(void);
    /**
    * Default destructor. Does nothing.
    */
    virtual ~C3DObjectLoader(void);
    /**
    */
};
```
Implementation in C++ with OpenGL ES

```cpp
31   * Loads 3D model into "t3DModel" structure.
32   *
33   * @param pModel Properly initialized "t3DModel".
34   * Filename of pModel has to be set.
35   */
36 void load3DModel(t3DModel* pModel);
37 }
```

Listing 4.8: Definition of "C3DObjectLoader" Class

### 4.5.6 3D Format Interpreter

The format interpreter loads a 3D model from a file and stores it in a `t3DModel` data structure. Currently, the 3D Visualization Tool supports Autodesk’s 3DS file format. However, other interpreters that inherit the `I3DFormatInterpreter` interface could be implemented in the future.

A 3DS file consists of data chunks that are structured hierarchically with the top-level chunk specifying the file version. Each data chunk contains a six-byte header that stores the chunk’s ID and its length. The function `readChunkHeader()` is used to read the header of a chunk (Listing 4.9).

The group and material chunks are stored at the next level. Each group is further subdivided into chunks that hold the information of the group’s vertices, the group’s texture coordinates and the group’s material name. A material chunk stores the color information and the name of an optional texture file.

The `C3DSMaxInterpreter` class uses three recursive functions that read the data from a 3DS file. Their function names are `processNextChunk()`, `processNextGroupChunk()` and `processNextMaterialChunk()` (Listing 4.9).

```cpp
79 /**
80  * 3D Studio Max Interpreter.
81 */
82 class C3DSMaxInterpreter : public I3DFormatInterpreter
83 {
84 public:
85 86 /**
```
Implementation in C++ with OpenGL ES

```cpp
87     * Default constructor.
88     */
89     C3DSMaxInterpreter(void);
90
91     /**
92     * Default destructor.
93     */
94     virtual ~C3DSMaxInterpreter(void);
95
96     /**
97     * Creates a 3D model based on the 3D Studio Max format.
98     *
99     * @param pOT    Pointer to properly initialized object transport.
100    * @param pModel Will hold 3D model data after successful execution.
101    */
102     virtual void create3DModel(I3DObjectTransport* pOT, t3DModel* pModel);
103
104 private:
105
106     /** Pointer to object transport for the 3D Studio Max file. */
107     I3DObjectTransport* m_pOT;
108     /** Pointer to the current chunk read. */
109     tChunk* m_CurrentChunk;
110     /** Pointer to a temporary chunk. */
111     tChunk* m_TempChunk;
112
113     /**
114     * Reads a string and sets the pointer of the string.
115     *
116     * @param name Reference to a NULL string pointer.
117     *
118     * @returns Number of bytes read including the null character.
119     */
120     unsigned int getString(char* &name);
121
122     /**
123     * Reads a chunk header.
124     *
125     * @param pChunk Pointer to a properly initialized chunk.
126     */
127     void readChunkHeader(tChunk* pChunk);
128
129     /**
130     * Reads and processes main chunks.
131     *
132     * @param pModel         Pointer to the current 3D model.
133     * @param pPreviousChunk Pointer to the previous chunk.
134     */
135     void processNextChunk(t3DModel* pModel, tChunk* pPreviousChunk);
136
137     /**
138     * Reads and processes group chunks.
139     *
140     * @param pModel         Pointer to the current 3D model.
141     * @param pGroup         Pointer to the current 3D group.
142     * @param pPreviousChunk Pointer to the previous chunk.
143     */
144     void processNextGroupChunk(t3DModel* pModel, t3DGroup* pGroup,
145                                  tChunk* pPreviousChunk);
146
147     /**
148     * Reads and processes material chunks.
149     *
150     * @param pModel         Pointer to the current 3D model.
151     * @param pPreviousChunk Pointer to the previous chunk.
152     */
153     void processNextMaterialChunk(t3DModel* pModel, tChunk* pPreviousChunk);
154```
void readColorChunk(t3DMaterial* pMaterial, tChunk *pChunk);

/**
 * Reads and processes the group's vertices.
 * @param pModel Pointer to the current 3D model.
 * @param pGroup Pointer to the current 3D group.
 * @param pPreviousChunk Pointer to the previous chunk.
 */
void readVertices(t3DModel* pModel, t3DGroup* pGroup, tChunk* pPreviousChunk);

/**
 * Reads and processes the group's face information.
 * @param pModel Pointer to the current 3D model.
 * @param pGroup Pointer to the current 3D group.
 * @param pPreviousChunk Pointer to the previous chunk.
 */
void readVertexIndices(t3DModel* pModel, t3DGroup* pGroup, tChunk* pPreviousChunk);

/**
 * Reads and processes the group's texture coordinates.
 * @param pModel Pointer to the current 3D model.
 * @param pPreviousChunk Pointer to the previous chunk.
 */
void readTexCoordinates(t3DModel* pModel, tChunk* pPreviousChunk);

/**
 * Reads and processes the group's material index.
 * @param pModel Pointer to the current 3D model.
 * @param pGroup Pointer to the current 3D group.
 * @param pPreviousChunk Pointer to the previous chunk.
 */
void readGroupMaterial(t3DModel* pModel, t3DGroup* pGroup, tChunk* pPreviousChunk);

4.5.7 3D Texture Interpreter

The texture interpreter loads an image from a texture file and stores the pixels in a texture map. Currently, the 3D Visualization Tool supports the TGA file format. However, other interpreters that inherit the I3DTextureInterpreter interface could be implemented in the future.

A TGA file consists of a header and a body that stores the pixel data. The information stored in the header is the width and the height of the image.
Implementation in C++ with OpenGL ES

specified in number of pixels, the number of bits per pixel and the orientation of the image, i.e. whether it is flipped horizontally or vertically.

It also stores whether the image is compressed or not. A compressed TGA file is run-length encoded. Run-length encoding is a common algorithm used for compressing digital images. Instead of storing each pixel separately, it combines the same consecutive pixels into one pixel and adds a count specifying the number of consecutive pixels.

TGA files are unique because each pixel is stored in the opposite order. The blue component comes first, followed by the green component, followed by the red component.

The CTGAInterpreter class uses the loadTGA() function to retrieve the pixel data from a TGA file (Listing 4.10). It then flips the image horizontally or vertically if necessary.

```cpp
24 /**
25  * TGA Interpreter.
26  */
27 class CTGAInterpreter : public I3DTextureInterpreter
28 {
29 public:
30
31     /**
32     * Default constructor. Does nothing.
33     */
34     CTGAInterpreter(void);
35
36     /**
37     * Default destructor. Does nothing.
38     */
39     virtual ~CTGAInterpreter(void);
40
41     /**
42     * Creates an OpenGL ES texture based on the TGA format.
43     *
44     * @param pOT Pointer to a properly initialized object transport.
45     * @param pMaterial Will hold texture data after successful execution.
46     */
47     virtual void create3DTexture(I3DObjectTransport* pOT,
48         t3DMaterial* pMaterial);
49
private:
50
51     /** Pointer to object transport for the TGA file. */
52     I3DObjectTransport* m_pOT;
```
Implementation in C++ with OpenGL ES

```cpp
/**
 * Loads a TGA texture file into a "tTextureImage" structure.
 * @param pTexImage Pointer to a properly initialized texture image.
 */
void loadTGA(tTextureImage* pTexImage);

/**
 * Flips the pixels of the texture image vertically.
 * @param pTexImage Pointer to a loaded texture image.
 */
void flipVertically(tTextureImage* pTexImage);

/**
 * Flips the pixels of the texture image horizontally.
 * @param pTexImage Pointer to a loaded texture image.
 */
void flipHorizontally(tTextureImage* pTexImage);
```

Listing 4.10: Definition of "CTGAInterpreter" Class
5 Summary and Prospect

This chapter provides a summary of the previous chapters and offers some ideas for further development.

5.1 Summary

The goal of this master thesis was the design and implementation of a 3D Visualization Tool for automotive navigation systems. It was developed in cooperation with NAVTEQ to simulate a journey through a road network with selected POIs placed on both sides of the street.

A system running the 3D Visualization Tool should have the minimum system requirements of 64 MB of RAM and 400 Mhz of computing power. This is comparable to a standard PDA. The functional requirements were specified by NAVTEQ. They included the visualization of 3D models, the recognition of a common 3D file format and the changing of the camera perspective.

As the graphics library, OpenGL was chosen because it is based on a pipeline architecture and as such has performance advantages. The NAVTEQ ADASRP was used as the development platform for the 3D Visualization Tool.

5.2 Prospect

A new era of navigation systems seems to be closer than one might think. Although the current 2D navigation systems are still up to date, companies including NAVTEQ are starting to push for 3D navigation systems. The cooperation between NAVTEQ and the University of Applied Sciences in Darmstadt is a strong indicator for this.
Meanwhile, the advancement in technology is continuously progressing and chips that support accelerated graphic hardware are starting to capture the market of embedded systems.

However, there are still barriers to overcome. For example, no area-wide collection of 3D map data exists that is necessary for such a system. Another problem is the storage for this large amount of data. Imagine driving down the road and constantly having to change the CD because additional data has to be loaded. An in-vehicle computer with enough secondary storage could solve this problem.

The 3D Visualization Tool is an attempt to simulate a 3D navigation system. However, some aspects could not be realized in the time frame given. They are outlined below and may provide inspiration for further enhancements.

The following additions regarding the functionality of the 3D Visualization Tool came to mind during the preparation of this master thesis:

- The visualization of a horizon
- The visualization of a cockpit with a speedometer and a rearview mirror
- The visualization of environmental influences (e.g. fog, rain, snow)

Another task would be the migration from the Windows platform to an embedded system platform. However, this would require making UDAL or a similar map interface part of the 3D Visualization Tool. In addition, an OpenGL ES conformant graphics library would have to exist for the target platform.
### A List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>Adaptive Cruise Control</td>
</tr>
<tr>
<td>ADASRP</td>
<td>Advanced Driver Assistance System Research Platform</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>COP</td>
<td>Center of Projection</td>
</tr>
<tr>
<td>CRT</td>
<td>Cathode Ray Tube</td>
</tr>
<tr>
<td>CTM</td>
<td>Current Transformation Matrix</td>
</tr>
<tr>
<td>DAL</td>
<td>Data Access Library</td>
</tr>
<tr>
<td>DCS</td>
<td>Device Coordinate System</td>
</tr>
<tr>
<td>DLL</td>
<td>Dynamic Linked Library</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory</td>
</tr>
<tr>
<td>EGL</td>
<td>Native Platform Graphics Interface Layer</td>
</tr>
<tr>
<td>GDF</td>
<td>Geographic Data Files</td>
</tr>
<tr>
<td>GLU</td>
<td>OpenGL Utility Library</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>ISA</td>
<td>Intelligent Speed Advice/Adaptation</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
</tr>
<tr>
<td>MASSIVE</td>
<td>Map Access Server Specification for In-Vehicle Environments</td>
</tr>
<tr>
<td>MMF</td>
<td>Mobile Map Format</td>
</tr>
</tbody>
</table>
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPP</td>
<td>Most Probable Path</td>
</tr>
<tr>
<td>OCS</td>
<td>Object Coordinate System</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>P.M.</td>
<td>Prime Meridian</td>
</tr>
<tr>
<td>PCS</td>
<td>Projection Coordinate System</td>
</tr>
<tr>
<td>PDA</td>
<td>Personal Digital Assistant</td>
</tr>
<tr>
<td>POI</td>
<td>Point of Interest</td>
</tr>
<tr>
<td>PSF</td>
<td>Physical Storage Format</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, Green, Blue</td>
</tr>
<tr>
<td>RGBA</td>
<td>Red, Green, Blue, Alpha</td>
</tr>
<tr>
<td>UDAL</td>
<td>Unified Data Access Layer</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VCS</td>
<td>Camera Coordinate System</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>VP</td>
<td>Vehicle Positioning</td>
</tr>
<tr>
<td>VRAM</td>
<td>Video Random Access Memory</td>
</tr>
<tr>
<td>WCS</td>
<td>World Coordinate System</td>
</tr>
<tr>
<td>WGS-84</td>
<td>World Geodetic System 1984</td>
</tr>
</tbody>
</table>
# List of Figures

Figure 3.1: Graphic Pipeline .......................................................... 8  
Figure 3.2: Approximation of a Curve and a Sphere ......................... 12  
Figure 3.3: Different Graphical Coordinate Systems ............................. 13  
Figure 3.4: Geoid, Ellipsoid and Adjusted Ellipsoid ............................ 15  
Figure 3.5: Ellipsoidal Parameters .................................................. 16  
Figure 3.6: Geodetic Latitude, Longitude and Height .......................... 17  
Figure 3.7: Geographic to Cartesian Coordinate Transformation ............ 18  
Figure 3.8: Sinusoidal Projection .................................................... 19  
Figure 3.9: Projection Plane tangent to Earth Surface ......................... 20  
Figure 3.10: Synthetic Camera Model ............................................. 21  
Figure 3.11: Look-At-Function ...................................................... 23  
Figure 3.12: Model-View and Projection Matrices ............................. 24  
Figure 3.13: Perspective View Volume ......................................... 24  
Figure 3.14: Three Views of Perspective Projection ............................. 25  
Figure 3.15: Orthogonal View Volume .......................................... 27  
Figure 3.16: Orthogonal Projection ................................................. 27  
Figure 3.17: Cohen-Sutherland Regions ....................................... 30  
Figure 3.18: Cases in Cohen-Sutherland Algorithm ............................ 30  
Figure 3.19: Plane-Line Intersection ............................................. 32  
Figure 3.20: z-Buffer Algorithm ....................................................... 33  
Figure 3.21: Light, Surfaces and Computer Imaging .............................. 34  
Figure 3.22: Vectors used by the Phong Lighting Model ....................... 37  
Figure 3.23: Blockage of Light with (a) Ray Tracing (b) Phong Lighting . 39  
Figure 3.24: Calculation of a Vertex Normal .................................... 40  
Figure 3.25: Mapping of Texture Coordinates ................................... 42  
Figure 4.1: NAVTEQ ADASRP ......................................................... 45
List of Figures

Figure 4.2: MMF Tile References .................................................................53
Figure 4.3: Aerial Image of a Structure in San Francisco .......................56
Figure 4.4: Digital Image of a Structure in San Francisco .......................57
Figure 4.5: UML Class Diagram - 3D Visualization Tool ......................58
Figure 4.6: UML Sequence Diagram - 3D Visualization Tool ...............59
C List of Tables

Table 3.1: Phong Parameters for various Materials ........................................... 38
Table 4.1: MMF Road Network Layers .............................................................. 51
Table 4.2: MMF Carto Layers ......................................................................... 52
D Listings

Listing 4.1: DAL::newMapArea() .............................................................. 50
Listing 4.2: MapArea::getLinks() and MapArea::getPois() ....................... 50
Listing 4.3: Implementation of a Static Factory Method ......................... 60
Listing 4.4: Definition of "t3DModel" Class .............................................. 62
Listing 4.5: Definition of "C3DManager" Class ........................................ 67
Listing 4.6: Definition of "CSearchBoxAroundCar" Class ...................... 68
Listing 4.7: Definition of "CSimpleLocator" Class ............................... 70
Listing 4.8: Definition of "C3DObjectLoader" Class ............................... 71
Listing 4.9: Definition of "C3DSMaxInterpreter" Class ......................... 73
Listing 4.10: Definition of "CTGAInterpreter" Class ............................ 75
E Bibliography


F Statement of Authorship

I hereby certify that this master thesis is entirely my own work. I also confirm that any auxiliary material that was used for the preparation of this master thesis has been properly cited.

Darmstadt, November 3, 2005

Arnold Adric