MAINTAINING REALITY

Modelling 3D spatial objects in a Geo-DBMS using a 3D primitive

Master thesis by C.A. Arens
E-mail: calin_arens@hotmail.com
Delft, March 2003

Professor:	prof.dr.ir. P.J.M. van Oosterom
Supervisors:	drs. J.E. Stoter (general)
	drs. C.W. Quak (programming)

Section GIS-technology
Department of Geodesy
Faculty of Civil Engineering and Geosciences

TU Delft

Preface

The title of this report is ‘Maintaining Reality’. This title seems a little odd, but refers to the goal and final result of the research that is described in this report. The subtitle is ‘Modelling 3D spatial objects in a Geo-DBMS using a 3D primitive’ and says more about the contents of this report. The goal of the research in this report is to enable more realistic (3D) spatial applications, while improving the maintainability of spatial data. This goal is met, which means that reality is maintained. Hence, I can speak of ‘Maintaining Reality’.

This report is written as partial fulfilment of my Master thesis in Geodetic Engineering at the Faculty of Civil Engineering and Geosciences at Delft University of Technology. The research in this report concentrates on two main research subjects at the Section GIS-technology: ‘Spatial DBMSs (Geo-DBMSs)’ and ‘3D-GIS and Visualisation’.

I would like to thank my supervisor drs. J.E. Stoter with whom I worked together on this topic. She has been a great help in all areas. Furthermore, I would like to thank drs. C.W. Quak; especially for all his help on implementation issues. Then, I would like to thank prof.dr.ir. P.J.M. van Oosterom for all his feedback on and ideas for this research. Finally, I would like to thank Susanne, Maarten and Friso for their help and feedback on my thesis.

Delft, March 2003
C.A. Arens

Summary

More and more applications depend on 3D spatial data. These data are stored in Geo-DBMSs. The present Geo-DBMSs do not support 3D primitives, but 3D spatial objects can be modelled by using 2D primitives such as polygons in 3D space. This is possible by using 3D coordinates, which are supported by the Geo-DBMSs. In this way, several 2D polygons bound a 3D object. These 2D polygons can be stored in one (multi-polygon) or multiple records.

The absence of a real 3D primitive in the Geo-DBMSs however, results into two problems:

- The Geo-DBMSs do not recognize 3D objects, because they do not have a 3D primitive to model these objects. This results into DBMS functions not working properly (e.g. there is no validation for the 3D object as a whole and functions only work with the projection of these objects, because the third dimension is ignored [3]).
- In some cases the 2D objects, that bound a 3D object, are stored in multiple records; a better administration of these large datasets requires a 1:1 relationship between objects in reality and objects in the database, because then there is a clear connection between the object in the database and the object in reality.

Geo-DBMSs were chosen to store spatial data, because they could guarantee the safety of the data (in 2D). But with the arrival of applications depending upon correct 3D data, the present techniques do not suffice. The solution for this problem is to implement a real 3D primitive, including validation functions and functions that e.g. return the volume or the distance between objects in 3D. This improves the maintainability of 3D geo-datasets [2] and opens the door to more realistic applications [6], [12].

Therefore, the objective of this thesis is answering the following question:

How can 3D spatial objects be modelled (i.e. stored, validated, queried) in a Geo-DBMS using 3D primitives and how can these objects be visualised?

To answer this question the theory from various literature is used to create a prototype implementation of a 3D primitive in a Geo-DBMS.

3D Spatial objects are stored with the polyhedron as (3D) primitive. This primitive is easy for users to model objects, can fairly easily be validated, because the algorithms are not too difficult to implement and still result in realistic objects. Each polyhedron has a set of faces, which consist of a set of ordered nodes. These nodes point to a vertex (x,y,z). This means that the data model is geometric with internal topology. The polyhedron is stored within the original Oracle Spatial geometry data model.

The validation occurs by checking if the polyhedra are stored correctly and after that checking each characteristic of the polyhedra. These characteristics are: flat faces, should bound one volume, simplicit faces and orientable.

To improve the performance of queries, a spatial index should be made on a table with polyhedra. The standard Oracle Spatial indices can be used, because of the way the polyhedra are stored in the Oracle Spatial geometry data model. A bounding box is constructed around the 3D line or its projection in case of a 2D spatial index. A test shows that it is preferable to create a 3D spatial index (3D R-tree) rather than a 2D spatial index, to get maximal query performance.

Using functions that are part of Oracle Spatial, is not suitable for 3D objects, because these functions work with the 2D projection of the 3D objects. Instead, some of the most commonly used functions (e.g. area, volume, point-in-polyhedron and bounding box) are implemented in 3D, so that functions return a realistic value.

The polyhedra can be visualised in GIS and CAD programs that can make a DBMS connection. To do this, the polyhedra have to be exported to 3D multi-polygons. This export function is implemented, as is the import function that makes a polyhedron from a 3D multi-polygon. To visualise polyhedra in a VRML viewer, the objects in the database can be exported to a VRML file. This function is implemented, as is the function to make a polyhedron from a VRML object.

These conclusions together satisfy the goal to implement a 3D primitive in a Geo-DBMS in a way that improves the maintainability of 3D spatial data and opens the door is to more realistic applications.

Samenvatting

Steeds meer applicaties zijn afhankelijk van 3D geografische informatie. Deze informatie is opgeslagen in Geo-DBMSs. De huidige Geo-DBMSs ondersteunen geen 3D primitieven, maar 3D ruimtelijk objecten kunnen worden gemodelleerd door 2D primitieven zoals polygonen in 3D ruimte te gebruiken. Dit is mogelijk doordat 3D coördinaten *wel* worden ondersteund door de Geo-DBMSs. Zo bakenen een aantal 2D polygonen een 3D object af. Deze 2D polygonen kunnen worden opgeslagen in één (multi-polygoon) of meerdere records.

De afwezigheid van een echte 3D primitieve in de Geo-DBMSs zorgt voor een tweetal problemen:

- De Geo-DBMSs herkennen 3D objecten niet. Het resultaat is dat DBMS functies niet voldoende werken (bv. er is geen validatie voor het 3D object als geheel en andere functies werken alleen met de projectie van deze objecten, omdat de 3^e dimensie genegeerd wordt).
- In sommige gevallen worden 2D polygonen, die samen een 3D object vormen, opgeslagen een meerdere records. Een beter beheersbaarheid van grote ruimtelijke datasets hebben een 1:1 relatie nodig tussen de objecten in de database en in de werkelijkheid.

Geo-DBMSs worden gebruikt om geografische informatie op te slaan, omdat ze de veiligheid van deze data konden garanderen. Met de komst van applicaties die afhankelijk zijn van correcte 3D data moeten er nieuwe technieken worden ontwikkeld. De oplossing voor dit probleem is om een echte 3D primitieve te implementeren, inclusief validatie functie en functies die bv. het volume of de afstand tussen twee objecten in 3D teruggeven. Dit verbetert de beheersbaarheid van geo-databases en opent de deur naar meer realistische applicaties.

Het doel van deze scriptie is om de volgende vraag te beantwoorden:

Hoe kunnen 3D ruimtelijk objecten worden gemodelleerd (i.e. opgeslagen, gevalideerd, bevraagd) met een 3D primitieve in een Geo-DBMS en hoe kunnen deze objecten worden gevisualiseerd?

Om deze vraag te beantwoorden is theorie uit verschillende stukken literatuur gebruikt om een prototype van een 3D primitieve te implementeren in een Geo-DBMS.

3D ruimtelijke objecten zijn opgeslagen met de polyhedron als 3D primitieve. Deze primitieve is gemakkelijk voor een gebruiker te modelleren, kan vrij gemakkelijk worden gevalideerd, de algoritmes zijn niet te moeilijk te implementeren en geven een realistische abstractie van de werkelijkheid. Elke polyhedron heeft een set zijvlakken die bestaan uit een geordende set van hoekpunten. Deze hoekpunten verwijzen naar coördinaten (x,y,z). Dit vormt een geometrisch data model met interne topologie. De polyhedron is opgeslagen binnen het originele Oracle Spatial geometrische data model.

De validatie vindt plaats door te controleren of de polyhedra correct zijn opgeslagen. Vervolgens wordt elke eigenschap van de polyhedra gecontroleerd. Deze eigenschappen zijn: platte vlakken, mogen maar één volume afbakenen, simpele zijvlakken en oriënteerbaar.

Om de prestaties van bevragingen te verbeteren moet er een ruimtelijke index worden gemaakt op een tabel met polyhedra. De standaard Oracle Spatial indices kunnen worden gebruikt door de manier van opslag in het Oracle Spatial geometrische data model. Het omhullende volume is om de 3D lijn door alle coördinaten gemaakt of door de projectie hiervan als het om een 2D index gaat. Een test wijst uit dat het de voorkeur heeft om een 3D ruimtelijke index (3D R-tree) te gebruiken boven een 2D index. Dit geeft maximale prestaties.

Het gebruik van de standaard Oracle Spatial functies is niet geschikt voor 3D objecten, omdat deze functies alleen werken met de 2D projectie van de 3D objecten. In plaats daarvan zijn enige van de meest gebruikte functies (bv. oppervlakte, volume, punt-in-polyhedron en bounding box) in 3D geïmplementeerd.

De polyhedra kunnen worden gevisualiseerd in GIS and CAD programma's die een DBMS verbinding kunnen maken. Hiervoor moeten de polyhedra geëxporteerd worden naar 3D multi-polygonen. Deze export functie is geïmplementeerd, evenals de import functie om een polyhedron te maken van een multi-polygoon. Om polyhedra te visualiseren met behulp van een VRML-viewer, moeten de objecten in de database geëxporteerd worden naar een VRML-bestand. Deze functie is geïmplementeerd, evenals de vice versa functie.

Deze conclusies samen zorgen ervoor dat het doel behaald is, om een 3D primitieve in een Geo-DBMS te implementeren, zodat de beheersbaarheid van 3D ruimtelijke data verbeterd en de deur naar meer realistische applicaties geopend wordt.

Table of contents

1 Introduction

Geo-DBMSs make it possible to manage large spatial datasets in databases that can be accessed by multiple users at the same time. These spatial datasets usually contain 2D data, while more and more applications depend on 3D data. Some examples are 3D cadastres [1], telecommunications [6] and town planning [12]. These applications mainly come from the ever-growing tendency of using living space multifunctional by building in the vertical direction, e.g. apartments, buildings over spanning a road, tunnels and bridges [1]. 2D Spatial data can be modelled in the Geo-DBMS with 2D primitives. However, the present Geo-DBMSs do not support 3D primitives, but 3D spatial objects can be modelled by using 2D primitives such as polygons in 3D space. This is possible by using 3D coordinates, which are supported by the Geo-DBMSs. In this way, several 2D polygons bound a 3D object. These 2D polygons can be stored in one (multi-polygon) or multiple records.

The absence of a real 3D primitive in the Geo-DBMSs however, results into two problems:

- The Geo-DBMSs do not recognize 3D objects, because they do not have a 3D primitive to model these objects. This results into DBMS functions not working properly (e.g. there is no validation for the 3D object as a whole and functions only work with the projection of these objects, because the third dimension is ignored [3]).
- In some cases the 2D objects, that bound a 3D object, are stored in multiple records; a better administration of these large datasets requires a 1:1 relationship between objects in reality and objects in the database, because then there is a clear connection between the object in the database and the object in reality.

Geo-DBMSs were chosen to store spatial data, because they could guarantee the safety of the data (in 2D). But with the arrival of applications depending upon correct 3D data, the present techniques do not suffice. The solution for this problem is to implement a real 3D primitive, including validation functions and functions that e.g. return the volume or the distance between objects in 3D. This improves the maintainability of 3D geo-datasets [2] and opens the door to more realistic applications [6], [12].

Therefore, the objective of this thesis is answering the following question:

How can 3D spatial objects be modelled (i.e. stored, validated, queried) in a Geo-DBMS using 3D primitives and how can these objects be visualised?

To answer this question the theory from various literature is used to create a prototype implementation of a 3D primitive in a Geo-DBMS. Many concepts have been developed in the area of 3D modelling [5], [6], [17], [18], [19], [20]. The innovation of this research is that the developed concepts have been translated into prototype implementations of a true 3D primitive in a DBMS-environment (Oracle 9i Spatial). As far as known, this is the first time ever that a Geo-DBMS directly supports a 3D primitive. Oracle Spatial 9i will be used as Geo-DBMS, because of the good knowledge and availability at the department, but the research is applicable to all Geo-DBMSs. Furthermore, it is important that the 3D data in the database can be visualised.

Chapter 2 presents the choice of a 3D primitive to model the 3D objects with and describes its implementation. The validation that ensures the correctness of the 3D objects is defined in chapter 3. Chapter 4 continues with the spatial index that is used to speed up the 3D functions. The implementation of the most commonly used functions (e.g. area, volume, point-in-polyhedron and bounding box) in 3D is presented in chapter 5. Chapter 6 contains methods to visualise 3D spatial objects that are stored as 3D primitives. Some test data is created in chapter 7 to test the prototype implementation and chapter 8 concludes this research and has recommendations for further research.

2 3D Primitive

At the moment, Geo-DBMSs are able to store, validate and query spatial data in 2D coordinate space. 2D spatial objects are stored as 2D primitives (polygons). To store 3D spatial objects, without the problems mentioned in the introduction, a 3D primitive is necessary.

In §2.1 a 3D primitive is chosen and defined in §2.2. This chapter continues with the data model for storing this 3D primitive in §2.3 and discusses this model in §2.4. The chapter ends in §2.5 with the actual implementation.

2.1 Choosing a 3D primitive

Stoter and Van Oosterom [4] propose a number of 3D primitives to model 3D spatial objects with:

- Tetrahedron: This is the simplest 3D primitive (3-simplex) and consists of 4 triangles that form a closed object in 3D coordinate space (Fig 1). It is relatively easy to create functions that work on this primitive. The object is well defined, because the three points of each triangle always lie in the same plane. A disadvantage is that it could take many tetrahedra to construct one factual object; this does not solve the problem of not having a 1:1 relationship between the factual object and the object's representation in the database (see chapter 1). Note that a tetrahedron is a special case of a polyhedron that is described below.

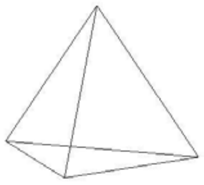

Fig 1 Tetrahedron.

- Polyhedron: This is the 3D equivalent of a polygon. It is made up by several flat faces (Fig 2). An advantage is that one polyhedron equals one factual object. Because a polyhedron can have holes in the shell, it can already model many objects. A disadvantage (that is shared by the tetrahedron primitive) is that the buffer operation results into a non-polyhedral object, because it will contain spherical or cylindrical patches, which cannot be represented by the polyhedron primitive. The solution is to approximate the result of the buffer operation by several flat faces [21].

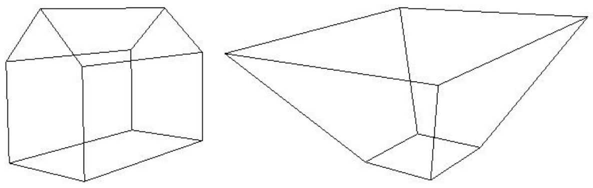

Fig 2 Collection of polyhedra.

- Polyhedron combined with spherical and cylindrical patches: This is the equivalent of the current 2D geometry data model of Oracle Spatial (i.e. straight lines and circular arcs). This possibility makes it possible to model 3D objects even more realistic (Fig 3). The result of the buffer operation is still not closed in all cases, because some parts of the buffer boundary can be very complex curves that can only be modelled by parametric curved elements. Modelling with this primitive is complex, because the user has to make a choice between polyhedral and curved elements. This will undoubtedly lead to different users modelling the same object in a different way.

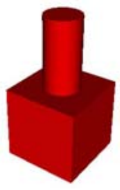

Fig 3 Polyhedron combined with a cylinder.

- CAD objects: There are many possibilities [16], e.g. Constructive Solid Geometry (CSG, Fig 4), cell decomposition, octree [12] and objects with curved faces. These objects either do not fit with the present (OpenGIS/ISO) 2D geometry data model or are very complex to model without an advanced graphics user interface.

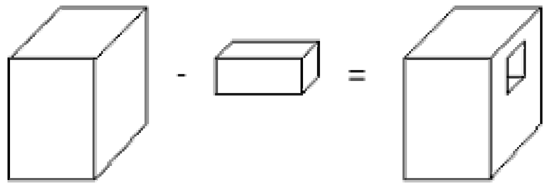

Fig 4 Example of Constructive Solid Geometry.

To choose a suitable 3D primitive some criteria have to be evaluated [16]. The implementation should lead to valid objects (see chapter 3). And once an object is modelled, there cannot be any ambiguities. A representation of an object should make clear how the object looks like in reality. It should be easy to create and enable efficient algorithms. Furthermore, the size and redundancy of storage (conciseness) should be taken in consideration. These criteria are evaluated for the 4 possible 3D primitives and listed in Table 1.

	Validation	Realism	Ease of modelling	Algorithms
Tetrahedron	++	+	-	++
Polyhedron	+	+	++	+
Polyhedron combined with spherical and cylindrical patches	-	++	-	+/-
CAD objects	-	++	+/-	--

Table 1 Evaluation of the possible 3D primitives.

The tetrahedron is not suitable, because there are several primitives necessary to model one object and that was one of the problems. CAD objects with curved faces can model a spatial object very realistic, but are complex to model without an advanced graphics user interface and other CAD objects do not fit within the present 2D geometry data model. That leaves the polyhedron option with and without the cylindrical/spherical patches. The one with spherical and cylindrical patches would fit better to the present 2D geometry data model, but ease of creation and implementation favour the polyhedron without spherical and cylindrical patches. Therefore, the polyhedron is chosen as the 3D primitive in this research to start with. If needed, spherical and cylindrical patches can be approximated by several flat faces (Fig 5). It is also expected that choosing a relatively simple primitive will give more insight into the problems that occur when implementing more complex primitives in the future.

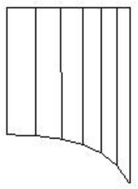

Fig 5 Approximation of cylindrical patch by several flat faces.

2.2 Definition of a polyhedron

A polyhedron is the 3D equivalent of a polygon (in 2D space) and can be defined as a bounded subset of 3D coordinate space enclosed by a finite set of flat polygons (called faces) such that every edge of a polygon is shared by exactly one other polygon [16]. Note that the polyhedron should bound a single volume, i.e. from every point (can be on boundary), every other point (can be on boundary) can be reached via the interior. The characteristics of the polyhedron primitive are (see chapter 3):

- Flatness: The polygons that make up the polyhedron have to be flat. This means that all points that make up the polygon must be in the same plane. For three points this is always true, but for more than three points this is not always true [2], because of the geodetic measuring and processing methods and the finite representation of coordinates in a digital computer. Furthermore, inner rings (hole in polygon) of a face have to be in the same plane as the outer ring that it belongs to [4].

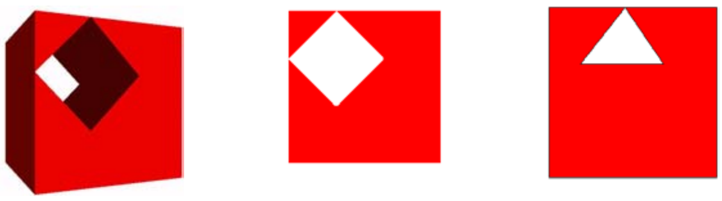

Fig 6 Left: Invalid polyhedron, because the hole divides the polyhedron in two volumes. Note that this object is stored as one volume minus a hole (2 connected inner rings) and not as two separate volumes. Centre: Invalid polygon (bounds two areas). Right: Valid polygon with inner ring that touches boundary.

- 2-Manifold: This characteristic looks at a polyhedron as a whole; it should bound only one volume. This means that from every point on the boundary, you should be able to reach every other point on the boundary via the interior (Fig 6). For the object to be valid, the faces where the hole starts and ends have to be modelled as a face with one or more inner rings. The edges and vertices should be 2-manifold [16]. This means an edge is adjacent to exactly two faces and a vertex is the apex of only one cone of faces (i.e. two or more shells do not touch in one vertex, Fig 19 on page 18).

- Simplicity: This characteristic looks at the faces of a polyhedron. The polyhedron has to be composed of simple features [22]. These are closed polygons that are not self-intersecting and have no inner rings [11], [16]. The faces of a polyhedron however, are allowed to have inner rings, as long as the faces together form a closed polyhedron. That is the reason this characteristic is called simplicity and not just simple. The inner rings of faces are not allowed to interact with the outer ring, except for touching boundaries. Furthermore, the vertices that span a face are not allowed to lie all on a straight line, i.e. the face has to have an area. A face has exactly one outer ring and zero or more inner rings. Finally, each edge has exactly 2 vertices [16]. Only straight line segments are allowed, so there is no necessity for an edge to have more than 2 vertices. Note that two or more (but not all) edges are allowed to lie on a straight line, if this is more convenient for modelling an object (Fig 7).

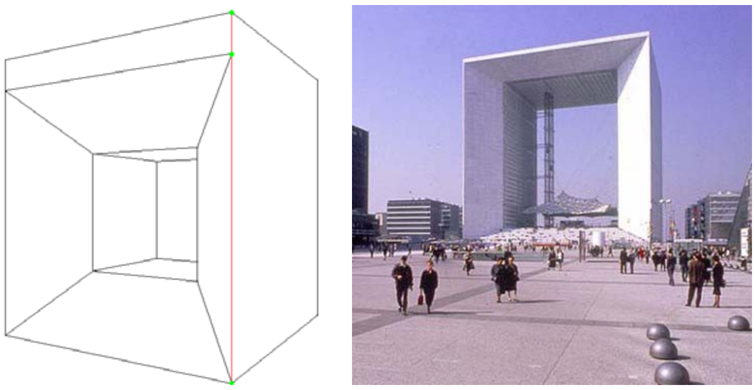

Fig 7 Two or more edges (red edges between green vertices) are allowed to lie on a straight line. The object modelled here (left) is a simplified version of ‘La grande arche’ in La Défense, Paris (right).

- Orientable: There has to be a clear outside and inside of the polyhedron. In the field of computer graphics [10] the normal vectors of faces point from inside to outside. This means that the vertices in a face must be specified in counter-clockwise order seen from the outside of the object. Note that the vertices in inner rings of faces need to be ordered in opposite direction (clockwise).

All polyhedra need to fulfil these characteristics. The validation function (chapter 3) is able to check if these characteristics are met.

2.3 Data model

The polyhedron can be stored by storing the vertices explicitly (x,y,z) and describing the arrangement of these vertices in the faces of the polyhedron (Fig 8 shows an UML diagram). This yields a hierarchical boundary representation [5], [16]. Note that edges are not stored explicitly in this model. There are tags that describe if the face description is an outer or inner boundary (of a polyhedron) or an outer or inner ring (of a face). With these elements it is already possible to model complex objects, e.g. objects with through-holes or objects that are hollow inside. This set of elements is enough for the functions to understand what the 3D spatial objects look like.

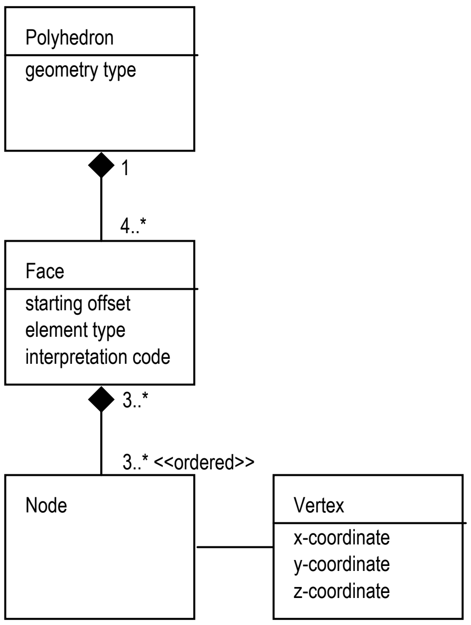

Fig 8 UML diagram of polyhedron storage.

The 3D primitive is implemented in a geometrical model with internal topology. This means that topology between objects is not maintained. Internal topology (topology within 3D objects) is maintained since the vertices for one object will be stored only once: faces are defined by internal references to the nodes and nodes are shared between faces.

There is a special geometry type in the object-relational model in Oracle Spatial 9i. This type is called sdo_geometry and is defined as:

```
CREATE TYPE sdo_geometry AS OBJECT (
sdo_gtype NUMBER,
sdo_srid NUMBER,
sdo_point SDO_POINT_TYPE,
sdo_elem_info MDSYS.SDO_ELEM_INFO_ARRAY,
sdo_ordinates MDSYS.SDO_ORDINATE_ARRAY);
```

This type is stored in the MDSYS scheme. The meaning of the elements of sdo_geometry is [15]:

- sdo_gtype: This indicates the type of geometry (point, linestring, polygon, multipoint, multilinestring, multipolygon) and the dimension (0D, 1D, 2D, 3D) of its embedding space. Each geometry type has its own code, e.g. a 2D polygon has sdo_gtype = 2003. The first digit is the dimension and the last digit is the geometry type.
- sdo_srid: This is a reference to the spatial reference system used by the coordinates. In this research local (Cartesian-)coordinates are used, so no sdo_srid is specified (NULL). Non-projected reference systems have to be converted to Cartesian coordinates first.
- sdo_point: This element is used when only points are stored as single object or when a point is stored in addition to the other geometry. The SDO_POINT_TYPE has an x-, y- and z-element.
- sdo_elem_info: This specifies the elements of the geometry with references to the coordinates (starting_offset), information about the element itself (e_type) and an interpretation code (e.g. straight line, rectangle, circle) on how to interpret the coordinates. This is stored in a variable array of numbers. A rectangular polygon specified by two coordinates is e.g. stored as sdo_elem_info_array = (1,1003,3).
- sdo_ordinates: This is a variable array of numbers and contains the coordinates.

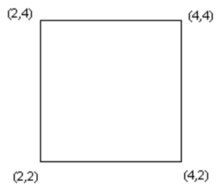

Fig 9 Square with its coordinates.

Fig 9 shows a square with its coordinates. The SQL to insert this geometry in Oracle Spatial 9i is:

```
INSERT INTO table (id, geometry) VALUES (1,
mdsys.sdo_geometry(2003, NULL, NULL,
mdsys.sdo_elem_info_array(1,1003,3),
mdsys.sdo_ordinate_array(2,2,4,4)));
```

This means that elements of sdo_geometry are:

- sdo_gtype = 2003 (2D polygon)
- sdo_srid = NULL (no spatial reference system)
- sdo_point = NULL (no point type)
- sdo_elem_info = 1,1003,3 (coordinates start at position 1, outer polygon ring, rectangle)
- sdo_ordinates = 2,2,4,4 (southwest and northeast coordinates)

To extend Oracle Spatial 9i with a polyhedron geometry, a new set of codes is necessary. The proposal for these codes as described in [4] is the starting point for this research. The data model is a geometric model, defined with internal topology; the vertices are stored only once per polyhedron. It is important not to use any existing codes for new features. The following extensions are applied:

- sdo_gtype: A 3D polygon already exists in Oracle Spatial 9i (3003). The 3D polyhedron is 3008, with the 3 standing for 3-dimensional and the 8 for polyhedron.
- sdo_srid: No extensions.
- sdo_point: No extensions.
- sdo_elem_info: The starting offset is now referenced to where the face description in sdo_ordinates begins, not to the coordinates itself. Four new element types (e_type) are added:
    - 1006: Outer ring of exterior polyhedron boundary (face).
    - 1106: Inner ring of exterior polyhedron boundary (hole in face).
    - 2006: Outer ring of interior polyhedron boundary (face).
    - 2106: Inner ring of interior polyhedron boundary (hole in face).

  In §2.4 there is a discussion on among others the use of some more interpretation codes.
- sdo_ordinates: The coordinate list is extended with face descriptions. First all the vertices are listed once; they are implicitly numbered from 1 to the number of vertices. After that each face of the polyhedron is described with a reference to the point number of the vertices.

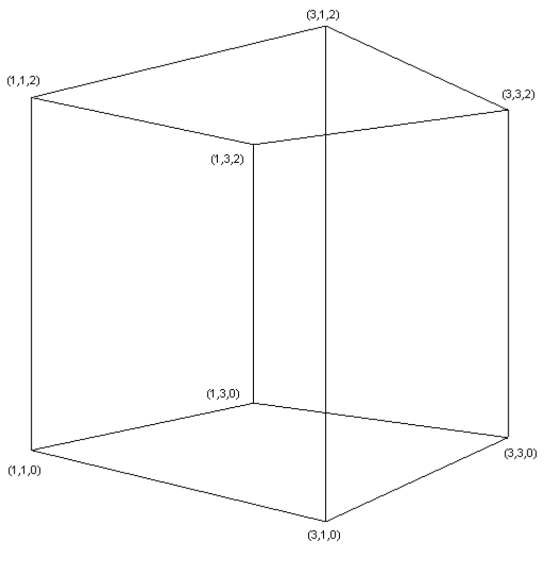

Fig 10 Cube with its coordinates.

Fig 10 shows a cube with its coordinates. The SQL to insert this geometry in Oracle Spatial 9i, as a polyhedron primitive by means of the proposed codes, is:

```
INSERT INTO table (id, geometry) VALUES (2,
mdsys.sdo_geometry(3008, NULL, NULL,
mdsys.sdo_elem_info_array(
25,1006,1, 29,1006,1, 33,1006,1, 37,1006,1, 41,1006,1, 45,1006,1),
-- 25 is the first face, the first 24 are used by the coordinates
mdsys.sdo_ordinate_array(
1,1,0, 1,3,0, 3,3,0, 3,1,0, 1,1,2, 1,3,2, 3,1,2, 3,3,2, -- the coordinates
1,2,3,4, -- bottom face starts at index 25
8,7,6,5, -- top face starts at index 29
1,4,8,5, -- front face starts at index 33
2,6,7,3, -- back face starts at index 37
1,5,6,2, -- left face starts at index 41
4,3,7,8 -- right face starts at index 45
)));
```

This means that elements of sdo_geometry are:

- sdo_gtype = 3008 (3D polyhedron)
- sdo_srid = NULL (no spatial reference system)
- sdo_point = NULL (no point type)
- sdo_elem_info = 6 times x,1006,1 (exterior polyhedron boundary, x is where the face starts)
- sdo_ordinates = 8 coordinate triplets and 6 face descriptions

2.4 Design issues

The geometry data model from §2.3 is somewhat simplified. It fits with the present 3D geometry data model, but some elements are missing, e.g. there is no multipolyhedron variant (sdo_gtype = 3009 in [4]). And Oracle Spatial 9i supports a range of interpretation codes for rectangles, circles, etc. In §2.1 a 3D primitive without any curved elements was chosen, so the interpretation codes for curved elements are not necessary now. In the future, the system can be extended with these codes. The rectangle interpretation code offers the functionality to specify geometry faster. However, this code raises an additional question. Interpretation code 3 (rectangle in 2D) constructs a box shaped polyhedron, but how should rectangles for the faces of the polyhedron be constructed? A polygon in 3D is not defined with only two coordinates (Fig 11).

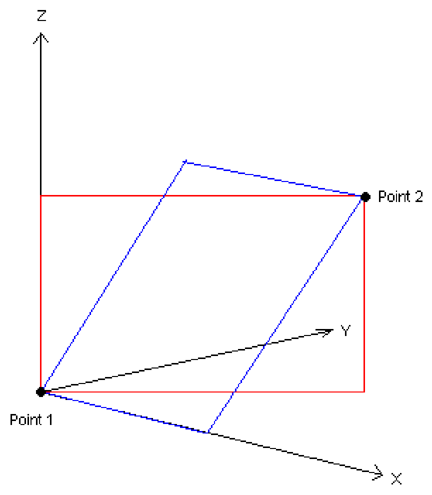

Fig 11 Two of the infinite (all rectangles rotated around the line through point 1 and 2) possibilities (red and blue) to span a 3D polygon with 2 points.

The solution would be to add even more interpretation codes, but errors are easily made. That is also why in this research only interpretation code 1 (face defined by an ordered set of vertices connected by straight lines) is used.

The next topic of discussion is the ordering of the vertices. They can be ordered either clockwise or counter-clockwise, seen from the outside of the object. For most internal DBMS functions (chapter 5) the orientation does not matter, but for some functions and especially for visualisation (chapter 6) the orientation is important. In the field of computer graphics [10] it is a custom to order all the vertices of outer rings counter-clockwise, seen from the outside of an object, and the vertices of inner rings clockwise. In this thesis the same is done for consistency and clarity.

The extensions on Oracle Spatial 9i are not recognised by the standard spatial index (chapter 4). Of course it is possible to implement a new spatial index interface with the existing spatial indices, but it is more convenient to use an existing one. Therefore, the implementation in §2.5 is altered somewhat from the geometry data model in §2.3.

Because of the finite representation of coordinates in a digital computer, values that should be zero in validation and other functions can deviate a little from this zero value. To solve this problem a tolerance value is introduced. This value is related to the size of the domain (e.g. the value could be kilometres or micrometres depending on the data). The validation function and some of the 3D functions have this tolerance value as input. It is important for these functions that this value is not equal to zero, because this will introduce errors in the functions if there are any deviations in floating point computations. This tolerance value should also not be too large, otherwise invalid objects will be accepted as valid. A good value for the tolerance is the standard deviation of the geodetic measurements. This value should be set in Oracle's metadata table (user_sdo_geom_metadata) or used as an input parameter in functions.

2.5 Implementation

Oracle Spatial ignores all elements with sdo_gtype or e_type = 0 (sdo_gtype and e_type are explained in §2.3). If the sdo_gtype = 0, the object is ignored by the spatial index. On the other hand, sdo_gtype = 3008 is not recognized and therefore it is also not possible to create a spatial index on that sdo_gtype. Therefore, an existing sdo_gtype = 3002 is chosen. This is a 3-dimensional polyline going through all the coordinates of the defined polyhedron. When creating a 3D spatial index (which is possible in Oracle), a bounding box is created around this line. This bounding box is equal to the bounding volume around the polyhedron. The drawback of using an existing sdo_gtype is that application will be confused, because there is no difference between a 3D polyline and a polyhedron.

In order to store the line, an entry in the sdo_elem_info is necessary. If the cube from §2.3 is taken (Fig 10), it will look like this:

```
INSERT INTO table (id, geometry) VALUES (2,
mdsys.sdo_geometry(3002, NULL, NULL, -- 3002 = 3D line
mdsys.sdo_elem_info_array(1,2,1, 25,0,1006, 29,0,1006, 33,0,1006, 37,0,1006,
41,0,1006, 45,0,1006), -- first triplet is line, then the faces
mdsys.sdo_ordinate_array(1,1,0, etc., 1,2,3,4, etc.)));
```

This means that elements of sdo_geometry are:

- sdo_gtype = 3002 (3D line)
- sdo_srid = NULL (no spatial reference system)
- sdo_point = NULL (no point data)
- sdo_elem_info = 1,2,1 (straight line) – x,0,1006 (6 times a exterior polyhedron boundary, x is where the face starts)
- sdo_ordinates = (8 coordinate triplets and 6 face descriptions)

E_type = 0 is necessary in this implementation for Oracle Spatial to ignore this element. The interpretation code is free to choose and thus takes the role that the e_type had. This is why the information about the element is moved to this position. The rest of the implementation is the same as described in the §2.3. Table 2 shows an overview of the storage options.

Sdo_gtype		3002: 3D line to create index on
Sdo_elem_info	startingOffset	Points to the starting offset of a face in sdo_ordinates
	e_type	Is always 0, to ensure proper working
	Interpretation- code	1006: Outer ring of exterior polyhedron boundary (face) 1106: Inner ring in exterior polyhedron boundary (hole in face) 2006: Outer ring of interior polyhedron boundary (face) 2106: Inner ring in interior polyhedron boundary (hole in face)
Sdo_ordinates		Then ordinate triplets that store the vertices Then face descriptions that point to the ordinate triplets

Table 2 Overview of storage options in the implementation of 3D primitive.

3 Validation

Large-scale spatial data is very valuable, because of the expense of labour intensive methods (designing, surveying and processing) it took to create these data. The DBMS protects the data integrity in a multi-user environment [6]. It is important that the spatial data is checked when it is inserted in the DBMS or when it is changed in the DBMS. This check on the geometry of the spatial objects is called validation. Valid objects are necessary to make sure the objects can be manipulated in a correct way, e.g. it is impossible to compute the volume of a cube when the top face is omitted; this would be an open box without a volume. Validating seams quite easy for the human eye, but a computer needs a large set of rules to check the spatial data.

This chapter describes the rules and the implementation to validate the 3D primitive. In §3.1 the correct storage is enforced. §3.2 through §3.5 contain the implementation of the validation rules for each characteristic of a polyhedron (§2.2). All the rules together enforce the correctness of the spatial data.

A polyhedron is valid when:

- It is stored correct.
- It has flat faces.
- It is 2-manifold (it bounds a single volume).
- Its faces are simplicit.
- It is orientable.

These rules are all implemented in Oracle, so it is presently possible to validate the polyhedron data type. The objects are validated in the order as in the ordering of characteristics of the polyhedron, because all functions are dependent on each other (Table 3). All functions need correctly stored objects (§3.1), they also need to know if the faces are flat (§3.2). Then the 2-manifold characteristic (§3.3) is tested. Hereafter follows the simplicity test (§3.4) that depends on the 2-manifold test. The orientation test (§3.5) is tested last, because this expects valid objects, except that the orientation is either completely correct or completely incorrect.

Function	Depends on
Correct storage	-
Flatness characteristic	Correct storage
2-Manifold characteristic	Correct storage, flatness
Simplicity characteristic	Correct storage, flatness, 2-manifold
Orientable characteristic	Correct storage, flatness, 2-manifold, simplicity

Table 3 Function dependencies.

3.1 Correct storage

Functions only work correctly if objects are stored in the way described in chapter 2. This is also true for the validation functions. This means the validation should start with validating correct storage. Correct storage is described in Table 2. Starting with the element info:

- The starting offset of the faces should be larger than the number of ordinate triplets in the sdo_ordinates and should be less than the total length of the sdo_ordinates.
- The e_type of the faces should equal 0.
- The interpretation code of the faces must be in {1006,1106,2006,2106}.
- The element info should not start with interpretation codes {1106,2106}, because these are inner rings and should always follow an outer ring.
- Interpretation code 1106 should follow 1006 or 1106, and 2106 should follow a 2006 or 2106. Note that 1106/2106 can follow themselves; this is the case when there are multiple inner rings in one face.

For the sdo_ordinates:

- The vertices listed in face descriptions must exist, that means that the reference must be smaller or equal to the amount of vertices.

To enforce these rules, one should retrieve the geometry from the database and look at the values of this geometry to see if these rules are met.

Example
This example shows geometry where the values for e_type (0) and interpretationCode (1006) for the first face are switched:

```
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,19,1006,0, --e_type and interpretationCode
--switched
22,0,1006, 25,0,1006, 29,0,1006, 33,0,1006),
mdsys.sdo_ordinate_array(-1,-1,1, 0,-1,-1, 1,-1,1,
0,1,-1, 1,1,1, 1,1,1,
1,3,2, 4,6,5, 1,5,6,3, 2,3,6,4, 1,2,4,5))
```

The following SQL statement runs the validation function on column geom. In table test with a tolerance value (§2.4) of 0.05:

```
SELECT validate_polyhedron(geom,0.05) VALID from test;
```

The result:

```
VALID
----------------------------------------------------------------
Storage error
```

The validation function recognises the error as a storage error.

3.2 Flatness characteristic

From the definition in §2.2 we can derive a set of validation rules that enforce this characteristic:

- The polygons that make up the polyhedron have to be flat.
- The inner ring of a face has to be in the same plane as the outer ring that it belongs to.

All vertices in a polygon should be in the same (flat) plane. This function has to check for every face description if it is flat. The inner rings of a face are to be checked together with the belonging outer ring of the face, because they always need to be in the same plane. The vertices of a face (from the outer plus inner rings) and a tolerance value are the input (Fig 12). The output is a Boolean value representing if the face is flat (planar) or not flat. A least squares plane [23] is estimated through the average coordinate of all vertices:

$$x_c = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$y_c = \frac{1}{n}\sum_{i=1}^{n} y_i$$

$$z_c = \frac{1}{n}\sum_{i=1}^{n} z_i$$

The derived plane equation is used to compute the distances between the vertices and the plane. The distance between the least squares plane and a vertex is computed by filling out:

$$Ax + By + Cz + D$$

for the vertex, this results in a distance. If one of these distances is larger than a certain tolerance value, then the vertices do not span a flat plane and are thus invalid.

A least squares plane minimises:

$$\sum_{i=1}^{n}\left(Ax_i + By_i + Cz_i - D\right)^2$$

where A, B and C are the components of the normal vector, D is the distance to the origin, x_i, y_i and z_i are the vertices and n is the number of vertices. If the average coordinate is subtracted from the vertices, the plane goes through the origin, which results in d=0. The components of the normal vector are now the unknowns and are solved as in [23].

To retrieve the plane equation, D can be computed by:

$$Ax_c + By_c + Cz_c + D = 0$$

where xc, yc and zc are the average coordinates of all vertices.

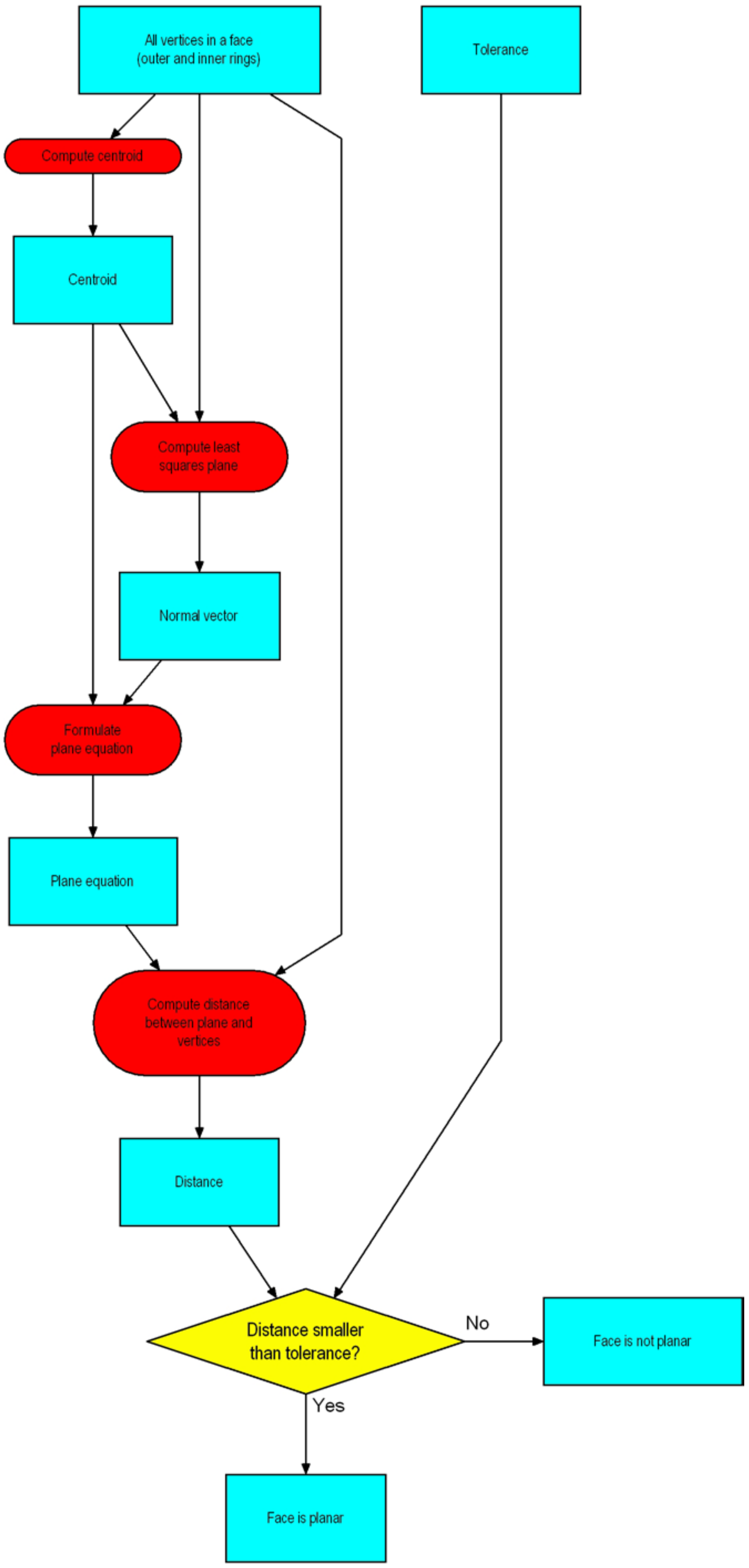

Fig 12 Determining if a face is planar.

Example
This example shows a cube with a non-flat face for tolerance 0.05:

```
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1, 25,0,1006, 29,0,1006, 33,0,1006, 37,0,1006,
41,0,1006, 45,0,1006),
mdsys.sdo_ordinate_array(
0,-1,0, -- y = -1 should be y = 0
3,0,0, 3,0,3, 0,0,3, 0,3,0, 3,3,0, 3,3,3, 0,3,3,
1,2,3,4, 4,3,7,8, 5,8,7,6, 1,5,6,2, 2,6,7,3, 4,8,5,1))
```

The following SQL statement runs the validation function on column geom. In table test with a tolerance value of 0.05:

```
SELECT validate_polyhedron(geom,0.05) VALID from test;
```

The result:

```
VALID
---------------------------------------------------------------------
Face not planar
```

The validation function detects the error and returns ‘Face not planar’.

3.3 2-Manifold characteristic

From the definition in §2.2 we can derive a set of validation rules that enforce this characteristic:

- The edges (derived out of 2 vertices) should be 2-manifold.
- There are no intersecting faces, because this will result in a polyhedron that bounds more than one volume. Note that an object that is stored as one volume minus one or more holes, which results in two or more separate volumes, is not allowed (e.g. Fig 6, see §2.2 for details).
- A polyhedron can only contain one object, e.g. two separate cubes should be stored as two polyhedra.
- The vertices should be 2-manifold.

2-Manifold edges

If the edges are 2-manifold (i.e. an edge is used in exactly 2 faces), there are no missing or dangling components (nodes, edges and faces) [11] (Fig 13). If there is a cut line in the objects, this object needs to be modelled as two separate objects (Fig 14).

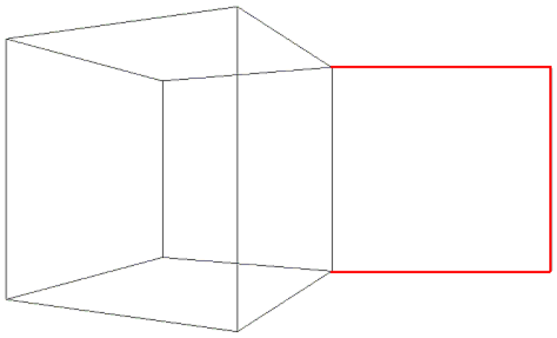

Fig 13 Invalid polyhedron, because of dangling face (red).

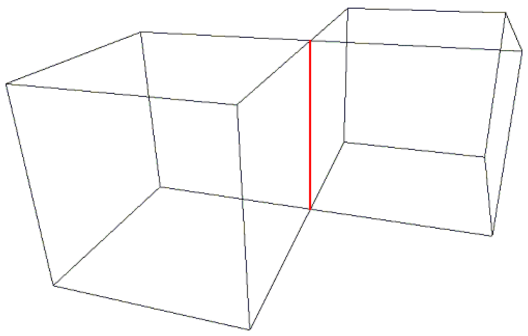

Fig 14 Invalid polyhedron, because of cut line (red).

For polyhedra without holes the topology of an object can be validated by Euler's law:

$$v - e + f = 2$$

This detects a variety of invalid objects. There is an extension on Euler's law called Euler-Poincaré's law [16] that does work with polyhedra containing holes of all kinds:

$$v - e + f = 2 \cdot (s - h) + r$$

where:

- v is the number of vertices
- e is the number of edges
- f is the number of faces
- s is the number of shell bodies (an internal hole is also a shell body)
- h is the number of through holes (also called the genus)
- r is the number of inner rings on the faces

In the data model however, the edges are not explicitly stored. The number of edges can be computed from Euler's law, but then Euler-Poincaré's law would lose some of its detection capabilities to find invalid objects, because the law is set up to count vertices, edges and faces independently. Some errors cannot be detected, because the vertices, edges and faces are really dependent. So, Euler-Poincaré's law cannot be used in this research.

There is another way to validate the internal topology of the objects [11] that can be used. It overlaps for a big part with the detection capabilities of the Euler-Poincaré's law and together with the other validation rules it gives a failsafe validation of the objects in the DBMS. It looks at the ordering of two following vertices in each face. A combination of vertices (implicit edge) is only allowed two times in a polyhedron and they have to be in the opposite direction (Fig 15). This will implement the validation rule for 2-manifold edges, so the object could be valid if this specific function returns true. If it returns false the object is certainly invalid.

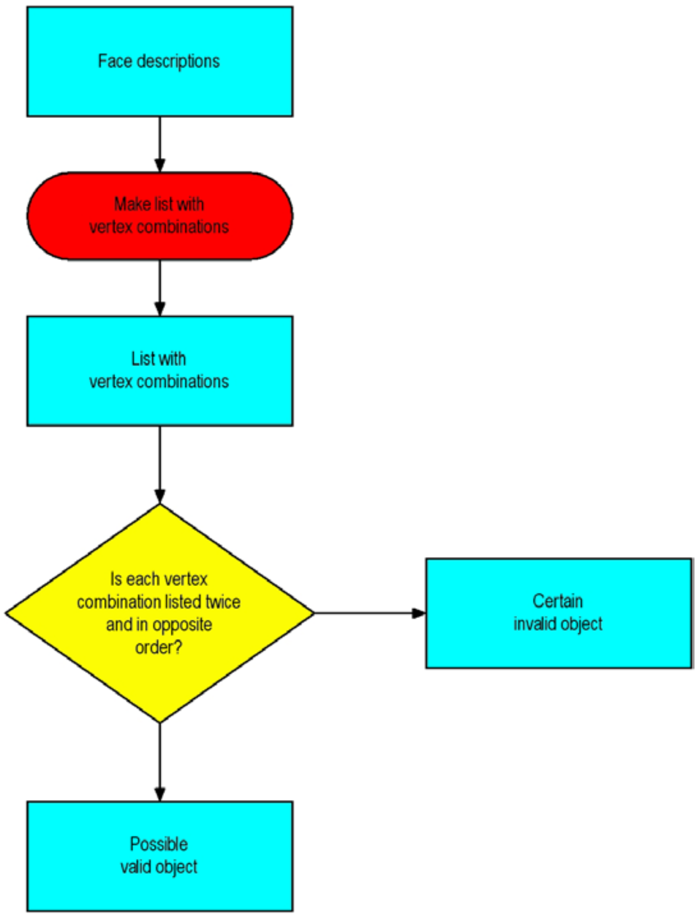
Face descriptions
Make list with vertex combinations
List with vertex combinations
Is each vertex combination listed twice and in opposite order?
Certain invalid object
Possible valid object

Fig 15 Determining if each vertex combination is listed twice and in opposite order in the face descriptions.

No intersecting faces

Faces are not allowed to intersect, except on edges (Fig 20). They can touch if this is the result of connecting two inner rings of faces (Fig 21). Note that this is not allowed if two or more separate volumes are created by this (Fig 6). This is a quite complex task to test. Each combination of faces has to be checked. To avoid computing an intersection for each combination, we look at the relative position of the two faces. If all vertices of the one face are on one side of the plane through the other face in the combination, then there is certainly no intersection. If this is not true, we have to do an intersection computation. If there are vertices on this plane through the face, the test should see if these are vertices. If there seems to be an intersection, we intersect all the edges of one face with the plane through the other face. If one of the intersection points is inside the face then the object is invalid. If it is outside the face, than this combination of faces does not intersect. If all the vertices of one face are on the plane through the other face, we only have to check if these vertices are inside the other face. If all vertices of the one face coincide with the other face, than two faces are equal and the object is invalid. If there are vertices of the one face coincident with the vertices of the other face, but not all, then the faces touch and the object could still be valid. If the vertices of the one face are inside the other face and do not coincide with its vertices then the object is invalid, because then the faces overlap. The (simplified) flow diagram of the intersection test is in Fig 18. An example where this test is needed is shown in Fig 16.

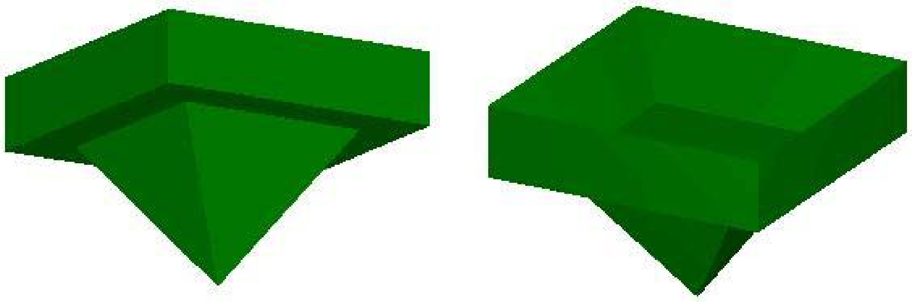
Fig 16 Invalid polyhedron with the top of the cone pointing down instead of up. The faces of the cone intersect with the square bottom of the object.

Note that testing face A with face B is not enough. Fig 17 shows that the edges of face A do not intersect with face B, but the face as a whole does intersect. To solve this the intersection test needs to be performed in opposite way too, face B with face A. The edges of face B *do* intersect with face A.

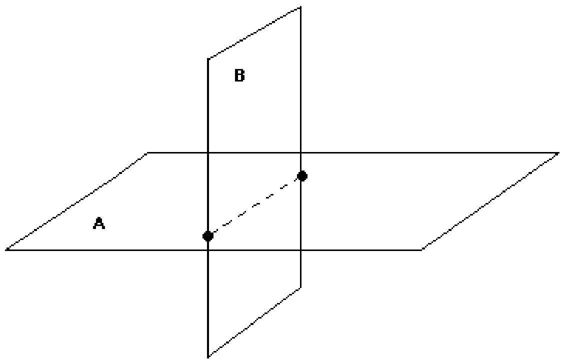

Fig 17 The edges of face A do not intersect with face B, but the face A as a whole does.

To test if there are through holes (genus) that separate the polyhedron in multiple volumes, we look if this through hole intersects two faces in a line. If this is the case, there are multiple volumes and the polyhedron will be invalid.

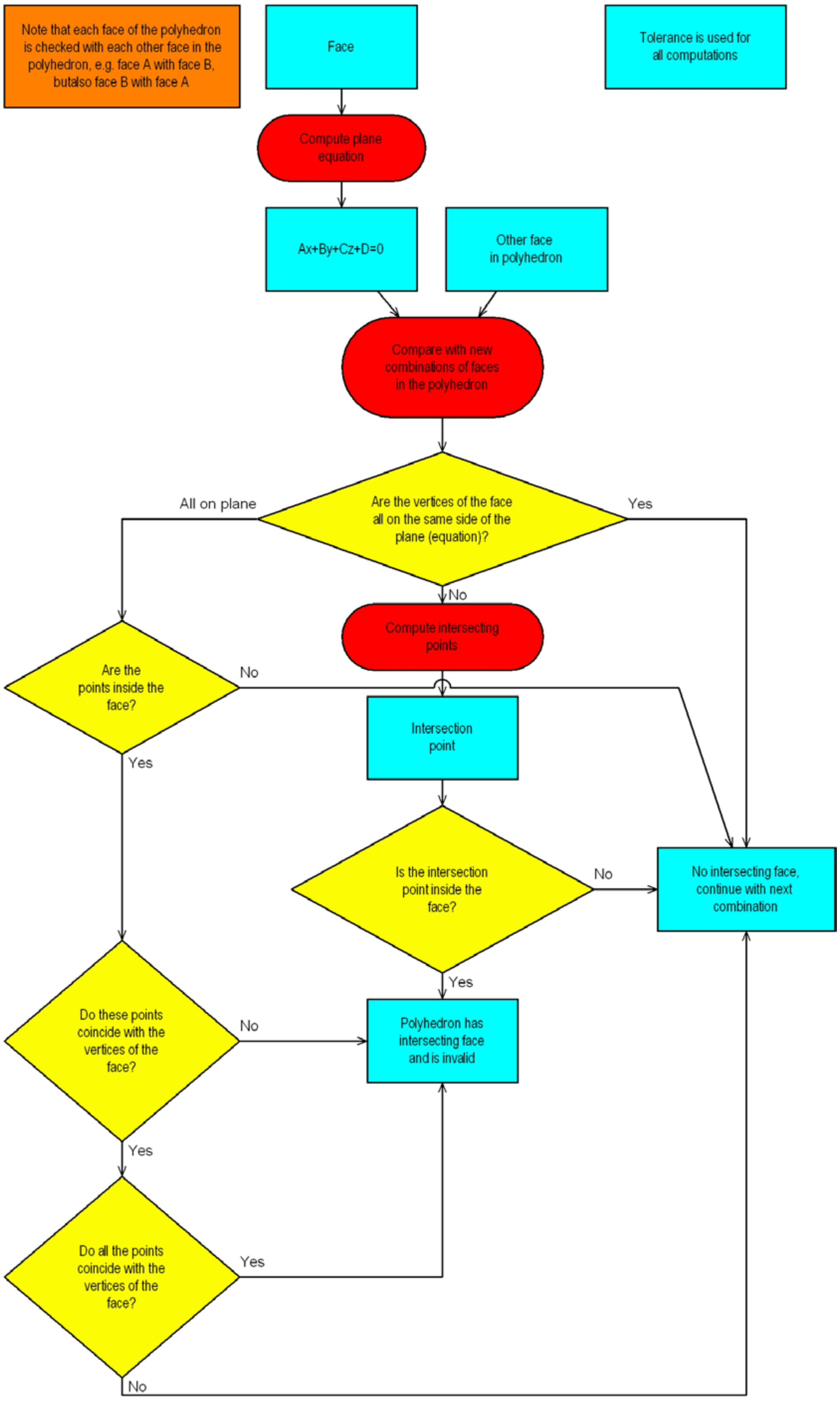
Note that each face of the polyhedron is checked with each other face in the polyhedron, e.g. face A with face B, butalso face B with face A
Face
Tolerance is used for all computations
Compute plane equation
Ax+By+Cz+D=0
Other face in polyhedron
Compare with new combinations of faces in the polyhedron
All on plane
Are the vertices of the face all on the same side of the plane (equation)?
Yes
No
Compute intersecting points
Are the points inside the face?
No
Yes
Intersection point
Is the intersection point inside the face?
No
No intersecting face, continue with next combination
Yes
Do these points coincide with the vertices of the face?
No
Polyhedron has intersecting face and is invalid
Yes
Do all the points coincide with the vertices of the face?
Yes
No

Fig 18 Determining if faces of a polyhedron intersect.

No separate objects as one polyhedron

Now that the edges are 2-manifold, Euler-Poincaré's law [16] can be used to check if there are no separate objects stored as one polyhedron. This is done by only looking at the outer boundary of the polyhedron (interpretation code 1006) and the inner rings that are in this outer boundary (interpretation code 1106). Note that if vertices are reused in inner rings then these vertices need to be counted twice. We only need to collect the necessary information and see if Euler-Poincaré's law holds:

$$v - e + f = 2 \cdot (s - h) + r$$

where:

- v is the number of unique nodes in the outer boundary.
- e is the number of edges in the outer boundary and its inner rings divided by two.
- f is the number of faces in the outer boundary.
- s is 1, because there should only be one shell body, inner boundary is ignored.
- h is the number of through holes (also called the genus)
- r is the number of inner rings on the faces of the outer boundary

Example

Fig 19 has 15 vertices, 24 edges and 12 faces. There are 2 shells (two volumes), no through holes and no inner rings:

$$v - e + f = 15 - 24 + 12 = 3$$
$$2 \cdot (s - h) + r = 2 \cdot (2 - 0) + 0 = 4$$

If the two cubes were not sharing a vertex, there would be a vertex more and the formula would return true. This is undesirable, because separate volumes are not allowed. To test this, we keep the number of shells at 1 (for the outer boundary). When Euler-Poincaré's law holds the polyhedron is still valid, otherwise it is invalid. Some more examples (including an example with a three-way hole) are in [28].

2-Manifold vertices

2-Manifold vertices make sure that two parts of a polyhedron do not touch in only one vertex (Fig 19). Objects like the one in Fig 19 need to be modelled as two separate polyhedra. This is a special case of two separate objects that are represented as one polyhedron. The Euler-Poincaré test detects this.

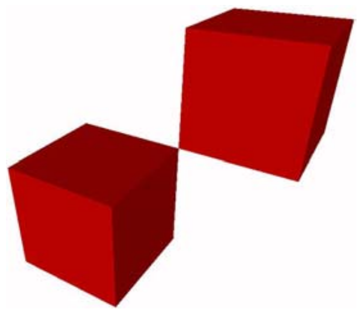

Fig 19 Invalid polyhedron, because of the vertex, that is not 2-manifold where the cubes touch.

Example
This example has two geometries, the first is the one from Fig 13 and the second is the one from Fig 20:

```
mdsys.sdo_geometry(3002,null,null, -- Fig 13
mdsys.sdo_elem_info_array(1,2,1, 31,0,1006, 35,0,1006, 39,0,1006, 43,0,1006,
47,0,1006, 51,0,1006, 55,0,1006),
mdsys.sdo_ordinate_array(0,0,0, 3,0,0, 3,0,3, 0,0,3, 0,3,0, 3,3,0, 3,3,3,
0,3,3, 6,3,0, 6,3,3,
1,2,3,4, 4,3,7,8, 5,8,7,6, 1,5,6,2, 2,6,7,3, 4,8,5,1, 6,7,9,10))
-- dangling face
```

```
mdsys.sdo_geometry(3002,null,null, -- Fig 20
mdsys.sdo_elem_info_array(1,2,1, 25,0,1006, 29,0,1006, 33,0,1006, 37,0,1006,
41,0,1006, 45,0,1006),
mdsys.sdo_ordinate_array(0,0,0, 3,0,0, 3,0,3, 0,0,3, 0,3,0, 3,3,0, 3,3,3,
0,3,3,
1,2,3,4, 4,3,8,7, 8,5,6,7, 1,6,5,2, 2,5,8,3, 6,1,4,7))
-- intersecting faces
```

The following SQL statement runs the validation function on column geom. In table test with a tolerance value of 0.05:

```
SELECT validate_polyhedron(geom,0.05) VALID from test;
```

The result:

```
VALID
--------------------------------------------------------------------
Not a 2-manifold object
Not a 2-manifold object
```

The validation function detects the errors and returns 'Not a 2-manifold object'.

3.4 Simplicity characteristic

From the definition of a polyhedron in §2.2 we can derive a set of validation rules that enforce this characteristic:

- Each edge has exactly 2 vertices.
- The starting point of a polygon is the same as the ending point.
- The vertices that span a face are not allowed to lie all on a straight line, i.e. the face has to have an area.
- The faces are not self-intersecting.
- The inner rings of faces are not allowed to interact with the outer ring (except for touching boundaries).

The method of storing the polyhedron is of influence on the validation functions. Because the edges are not explicitly stored, but formed by connecting two vertices in the face description (chapter 2), the rule that each edge has exactly 2 vertices is always true. Secondly, the starting point is not repeated as ending point in a face description. This means that the last point is always connected to the first point, i.e. the starting point always equals the ending point.

Faces must have an area

A face has to have at least 3 edges to be able to span an area. This means there are at least three vertices in the face description of each face in the polyhedron. This is implemented by looking at the number of vertices per face in the object's record in the database. If all these vertices of a face lie on a straight line, then the face is useless. To test this, the absolute value of the area should be greater than a certain epsilon (tolerance value). This function is also supported by the flatness characteristic (§3.2), because it is nearly impossible to compute the supporting plane with least squares if it does not have an area. The area function in 3D is explained in §5.1.

Faces are not self-intersecting

If a face is self-intersecting and there are no intersecting faces in the polyhedron, then this is detected by the 2-manifold edges validation function (§3.3). If a face is self-intersecting and there are intersecting faces (Fig 20), then this is detected by the 2-manifold intersection validation function (§3.3). Thus there is no need to implement this validation rule.

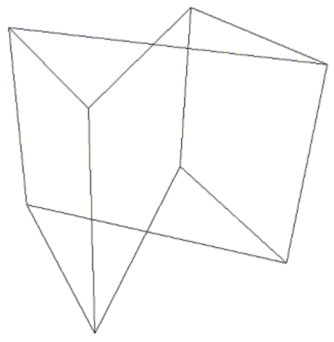

Fig 20 Invalid polyhedron with self-intersecting top and bottom.

Inner rings do not interact with their outer ring

Inner rings are not allowed to interact with its outer ring (Fig 21), except that touching is allowed if it does not subdivide the polyhedron. This is simply tested by seeing if any of the edges of the inner ring intersect with one of the edges of the outer rings.

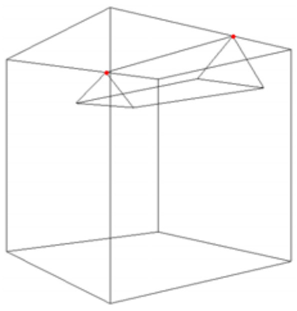

Fig 21 Valid polyhedron, because of inner rings that touch (red dots) with their outer rings. Intersection is not allowed. Note that still one volume is bounded.

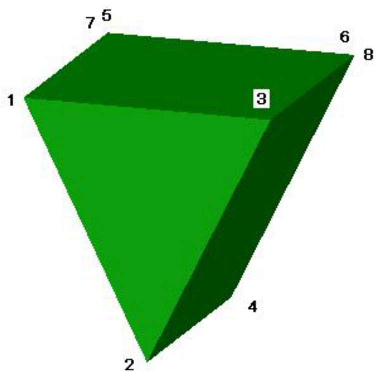

Fig 22 Invalid polyhedron, because face 5,7,8,6 has no area (vertices 5,7 and 6,8 are the same).

Example
This example shows a prism-shaped geometry where two unnecessary vertices form a face without an area (Fig 22), because these two vertices (7 and 8) are the same as the other two vertices (5 and 6) in the face:

```
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1, 25,0,1006, 28,0,1006, 31,0,1006, 35,0,1006,
39,0,1006, 44,0,1006),
mdsys.sdo_ordinate_array(-1,-1,1, 0,-1,-1, 1,-1,1, 0,1,-1, -1,1,1, 1,1,1, -
1,1,1, 1,1,1, -- unnecessary vertices
1,2,3, 4,5,6, 1,3,8,7, 6,5,7,8, 2,4,6,8,3, 1,7,5,4,2))
```

The following SQL statement runs the validation function on column geom. In table test with a tolerance value of 0.05:

```
SELECT validate_polyhedron(geom,0.05) VALID from test;
```

The result:

```
VALID
---------------------------------------------------------------------
Face not simplicit
```

The validation function detects the errors and returns ‘Face not simplicit’.

3.5 Orientable characteristic

From the definition in §2.2 we can derive a set of validation rules that enforce this characteristic:

- The vertices in the outer and inner rings of faces need to be ordered in opposite direction. In this thesis: the vertices of an outer ring are ordered counter-clockwise and the vertices of an inner ring clockwise seen from the outside of the object (Fig 23).

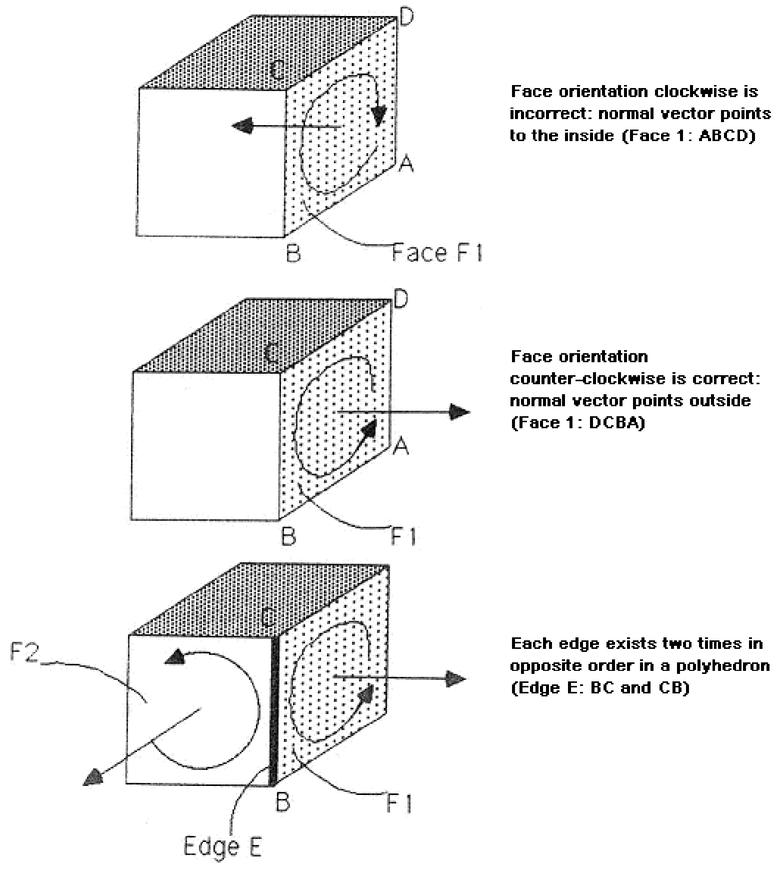
D
C
A
B
Face F1
Face orientation clockwise is incorrect: normal vector points to the inside (Face 1: ABCD)
D
C
A
B
F1
Face orientation counter-clockwise is correct: normal vector points outside (Face 1: DCBA)
C
F2
B
F1
Edge E
Each edge exists two times in opposite order in a polyhedron (Edge E: BC and CB)

Fig 23 Orientable characteristic.

As stated in §2.4 the order of vertices is important for visualisation. A clear inside and outside is received by ordering the vertices counter-clockwise looking from the outside of the object to a visible face. This means e.g. that parallel faces of a cube have a different ordering seen from the same side of the object, because one has to look from a different viewpoint to make the face visible. The 2-manifold edge function (§3.3) implicitly checks the ordering of the vertices, because each edge (combination of two vertices) has to be in a different direction. The only mistake that can be made is that all vertices in a boundary/shell are ordered exactly the other way around (clockwise for outer rings). A consequence is that the object is not visible when exported to VRML, because of back face culling [10] (unless the viewpoint is inside the object), or visualised incorrectly in GIS/CAD programs, especially if there are inner rings.

To check the orientation of the vertices, we only have to check the orientation of one edge, because the orientation of all other edges is checked by the 2-manifold edge function (§3.3). To check this:

- Find the point with the lowest z-ordinate. If there are more points with the same lowest z-ordinate, then find the one with the lowest y-ordinate and if there are more points with the same combination of z- and y-ordinate find the one with the lowest x-ordinate. This makes sure that the point is part of a convex part of the boundary of the face.

- Find the faces that have the point as one of its vertices. Compute the normal vectors of these faces and choose the face with the largest absolute normalised z-component in the normal vector This will yield the most flat face in the vertical direction, because this face's normal vector is closest to (0,0,-1). The components in x- and y-direction are not important, because the angle between (0,0,-1) and the normal vector of the face is only dependent on the z-component: $\cos\alpha = a \cdot b$, with *a* = (0,0,-1) and *b* is the normal. This results in the dot product always equalling 0 in the x- and y-component, because a_x and a_y equal 0. This is the most flat face in the vertical direction and because it has the lowest z-ordinate, this is the bottom of the object (which should have a normal vector pointing downwards, i.e. the z-component of the normal vector is negative).

- Compute the normal vector of the face by taking the cross product of the vectors between the point and its predecessor and successor To compute the z-component of this normal vector:

$$\vec{A} = P_i - P_{i-1}$$
$$\vec{B} = P_{i+1} - P_i$$
$$N_z = A_x B_y - A_y B_x$$

  If the vertices are ordered correctly, this normal vector has a negative z-component. If not, then the orientation test failed.

If there is an inner boundary (e-type=2006) then it needs to be checked for orientation too. The steps are the same as above, except for that you only look at the coordinates of inner boundary vertices.

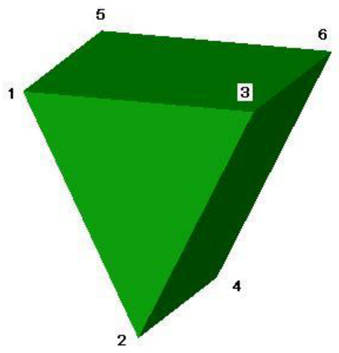

Fig 24 Polyhedron from the example above. The first face should be defined as 1,2,3 (counter-clockwise) and not as 1,3,2 like the example shows.

Example
This example has a prism-shaped geometry that is correct (Fig 24), except that all the vertices are ordered clockwise seen from the outside of the object:

```
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1, 19,0,1006, 22,0,1006, 25,0,1006, 29,0,1006,
33,0,1006),
mdsys.sdo_ordinate_array(-1,-1,1, 0,-1,-1, 1,-1,1, 0,1,-1, -1,1,1, 1,1,1, --
coordinates
1,3,2, 4,6,5, 1,5,6,3, 2,3,6,4, 1,2,4,5)) -- face descriptions
```

The following SQL statement runs the validation function on column geom. In table test with a tolerance value of 0.05:

```
SELECT validate_polyhedron(geom,0.05) VALID from test;
```

The result:

```
VALID
--------------------------------------------------------------------
Orientation incorrect
```

The validation function detects the errors and returns ‘Orientation incorrect’. The orientation can be corrected by using the function fix_orientation (Appendix B).

4 Spatial index

A spatial dataset often contains many objects; this causes spatial queries like: "Find all objects within this rectangle", to execute slowly, because all the objects need evaluation in this computational complex query, even objects that are not remotely close to the query window. The solution is to use a 'two-tier' query model [15]. In this model, the query is solved in two steps. The first step quickly returns a number of candidate objects with the help of a spatial index and the second step does the exact computation on these candidates, instead of on all the objects in the dataset. This solves the query much faster when there are many objects in the database. So the spatial index provides a way to quickly select a number of candidate objects, just like an index in a book quickly shows a number of candidate pages by looking at a certain keyword.

This chapter describes the use of a spatial index in this research. The possible spatial indices for the 3D primitive from chapter 2 are described in §4.1. §4.2 specifies an implementation of a spatial index to be used by certain 3D functions in chapter 5. The choice between a 2D or 3D spatial index is discussed in §4.3

4.1 Possible spatial indices for the 3D primitive

The two most commonly used spatial indices are the R-tree [14] and the quadtree [24]. These are both implemented in Oracle Spatial. These indices are able to index spatial data in 2D coordinate space. The 3D primitive can be indexed by the 2D R-tree and the quadtree by taking its 2D projection on the x,y-plane. Together with the 3D variants of the R-tree and the quadtree (respectively called 3D R-tree and octree), these spatial indices form the possibilities to index the 3D primitive:

2D R-tree

An R-tree index stores the Minimum Bounding Rectangle (MBR) that encloses each geometry in a spatial dataset (Fig 25). This MBR is used to reduce the computational complexity in spatial queries and is defined along the axes.

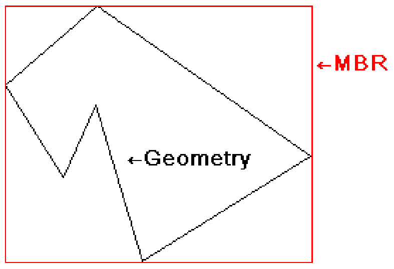

Fig 25 Minimum Bounding Rectangle (MBR) encloses geometry.

The MBR that encloses all the objects in a spatial dataset forms the root of the R-tree. This area is then subdivided in two or more nodes that each contains a MBR of one or more objects. This subdivision continues until all the objects have their own MBR (Fig 26). The nodes that are not subdivided in an R-tree are called leaf nodes and contain, besides the MBR, also a reference to the geometries in the spatial dataset. The nodes in higher levels are called non-leaf nodes.

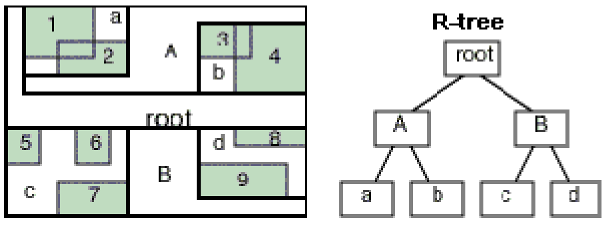

Fig 26 Concept of R-tree.

The advantage of using an R-tree index is that the irregular sized MBRs can fit the objects in the real world (in this case: footprints of buildings), in contrary to the subdivision of space in the quadtree. The disadvantage is that the MBRs can be much larger than the objects itself. This causes the R-tree index to select more candidate objects, because empty parts of the MBRs will fall within the query window (Fig 27). This increases the load in the exact computation (the second step in solving a query), because more objects need to be processed.

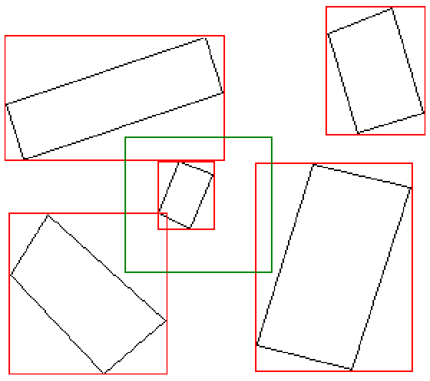
Fig 27 The query window (green) returns 4 objects when using the MBRs (red), while there is only one object inside the query window.

The solution to this problem is to allow oblique MBRs that fit the objects better, but this increases the computational complexity [6] and thus is not a very useful solution for a spatial index.

3D R-tree

The 3D R-tree uses the same concepts as the 2D R-tree. The only difference is that the space is subdivided in irregular shaped boxes instead of rectangles (Fig 28). Hence, the MBRs are replaced by MBBs (Minimum Bounding Boxes).

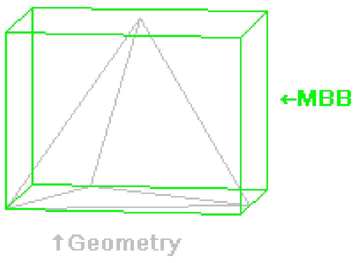

Fig 28 3D geometry (gray) enclosed by its MBB (green).

The 3D R-tree can also be used to represent objects with a lower level of detail (also possible with 2D R-tree). This comes in hands with rendering 3D scenes where the far-away objects do not have to be displayed in full detail. Instead, they are replaced by their approximation in the 3D R-tree [5], [6], [7]. For example, for far away objects a bounding box that bounds several objects is shown, for less far away objects a bounding box is shows that bounds a single object and for close by objects the complete objects is shown.

Quadtree (2D)

The quadtree is also a tree structure. In the R-tree the objects are organised, but the quadtree organises space by subdividing it in tiles of the same shape. A node is always subdivided in four (2^2) new nodes (hence the name quadtree). A node is subdivided if it still contains more than the allowed number of objects per node or until a specified level (number of subdivisions or tile size) is reached (Fig 29). At this level the nodes point to the geometries.

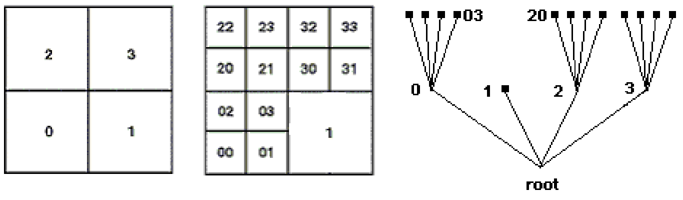

Fig 29 Concept of quadtree.

Octree (3D)

This 3D index uses boxes too. Just like the tiles in the quadtree, the boxes all have the same shape and all the boxes on a certain level are of the same size. Because of the third dimension each node is subdivided in eight (2^3) new nodes (hence the name octree). It works the same as the quadtree (Fig 30).

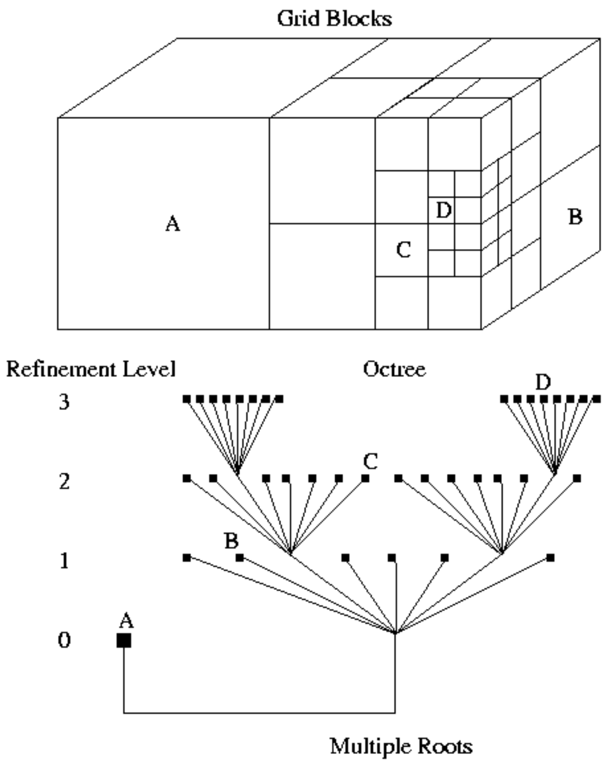

Fig 30 Concept of octree.

4.2 Implementing a spatial index for the 3D primitive

No new spatial index interface is implemented in this research; instead the existing Oracle spatial indices are used. Oracle spatial supports R-trees indices up to 4 dimensions and the (2D) quadtree (no support for octree). Using the Oracle spatial index is made possible by storing the 3D objects in a special way, i.e. Oracle Spatial ignores all elements with sdo_gtype or e_type = 0 (sdo_gtype and e_type are explained in §2.3). If the sdo_gtype = 0, the object is ignored by the spatial index. Therefore, an existing sdo_gtype = 3002 is chosen. This is a 3-dimensional polyline going through all the coordinates of the defined polyhedron. When creating a 3D R-tree in Oracle (R-trees up to 4 dimensions are supported), a bounding box is created around this line. This bounding box is equal to the bounding box around the polyhedron. The drawback of choosing an existing sdo_gtype is that applications will be confused whether the object is a 3D polyline or a polyhedron. Note that the octree is not implemented by Oracle and therefore not used in this research.

To create a spatial index on a geometry table in Oracle, first a record has to be inserted into the metadata table containing the domain of the coordinate space that the objects are in and the tolerances for the coordinates. This is done by the following SQL-statements:

```
-- creating table:
CREATE TABLE testtable (
id NUMBER,
geom MDSYS.SDO_GEOMETRY);

-- inserting cube geometry
INSERT INTO testtable VALUES (1,--id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1, 25,0,1006, 29,0,1006, 33,0,1006, 37,0,1006,
41,0,1006, 45,0,1006),
mdsys.sdo_ordinate_array(
0,0,0, 3,0,0, 3,0,3, 0,0,3, 0,3,0, 3,3,0, 3,3,3, 0,3,3, --coords
1,2,3,4, 4,3,7,8, 5,8,7,6, 1,5,6,2, 2,6,7,3, 4,8,5,1))); --faces
```

```
-- inserting metadata
INSERT INTO user_sdo_geom_metadata VALUES (
'TESTTABLE','GEOM', -- name of table and geometry column
mdsys.sdo_dim_array(
mdsys.sdo_dim_element('X',-100,100,0.001), -- domain for x and tolerance
mdsys.sdo_dim_element('Y',-100,100,0.001), -- domain for y and tolerance
mdsys.sdo_dim_element('Z',-100,100,0.001)), -- domain for z and tolerance
NULL);
```

Then the 3D R-tree can be created by the following SQL-statement:

```
CREATE INDEX index_name
ON testtable(geom) – table_name(geometry_column)
INDEXTYPE IS mdsys.spatial_index
parameters('sdo_indx_dims=3'); -- 3D R-tree
```

The index is then managed by Oracle. The spatial index can be used in spatial functions by including the SDO_FILTER function in the WHERE-clause of SQL queries.

4.3 Discussion

In many spatial applications the dimensions and the variations of the values in the x,y-plane are larger than in the z-direction. For example, a city plan typically covers an area of 5x5 kilometres with buildings up to 50 meters tall. This, plus the fact that queries usually try to find all the objects in a specific (x,y)-region (with possibly objects that are on top of each other), may make a 3D spatial index less useful in this kind of application [12]. In short, the x- and y-coordinate are more selective than the z-coordinate. This means a 2D spatial index might work just as good or better than a 3D spatial index, because it is a little more compact (2 2D points in stead of 2 3D points). A 2D R-tree is created by projecting the line through all the object's coordinates onto z=0. It is created by:

```
CREATE INDEX index_name
ON table_name(geometry_column)
INDEXTYPE IS mdsys.spatial_index; -- no parameters needed
```

A test is performed to see if one might just as well use a 2D spatial index and not a 3D spatial index. A dataset that contains a part of the buildings (polyhedra) in a city (Delft, The Netherlands) is created (chapter 7) and a number of query windows have been created in the city area. The query windows are boxes of varying size. There are two sets, both of 11 boxes (Fig 31 and Fig 32):

- Boxes with a height from 0 to 50m. NAP (Netherlands National Ordnance Datum)
- Boxes with a height from 20 to 50m. NAP

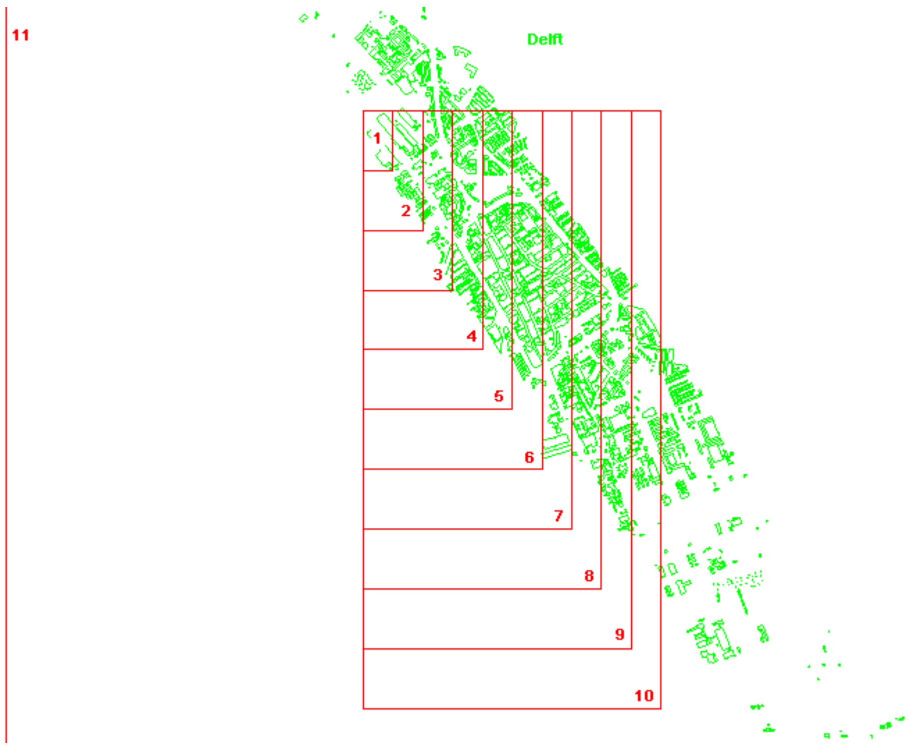

Fig 31 Top view of the area that is covered by both sets of the 11 query boxes. Note that box 11 is a big box containing all the buildings. The area on the picture is about 6x6km.

For each of the boxes is queried which buildings are (partially) inside the box. This results in the number of buildings in the dataset that intersect with each box. In the introduction can be read that a spatial index selects a set of candidates that have to be queried to retrieve the exact result. A 2D and a 3D spatial index (both R-trees) are created on the dataset. For each of the query boxes these spatial indices return a set of candidates. If the number of candidates is closer to the actual number of intersections the spatial index filter is more efficient. This means that the ratio between the actual number of intersections and the number of candidates is the efficiency of the spatial index filter.

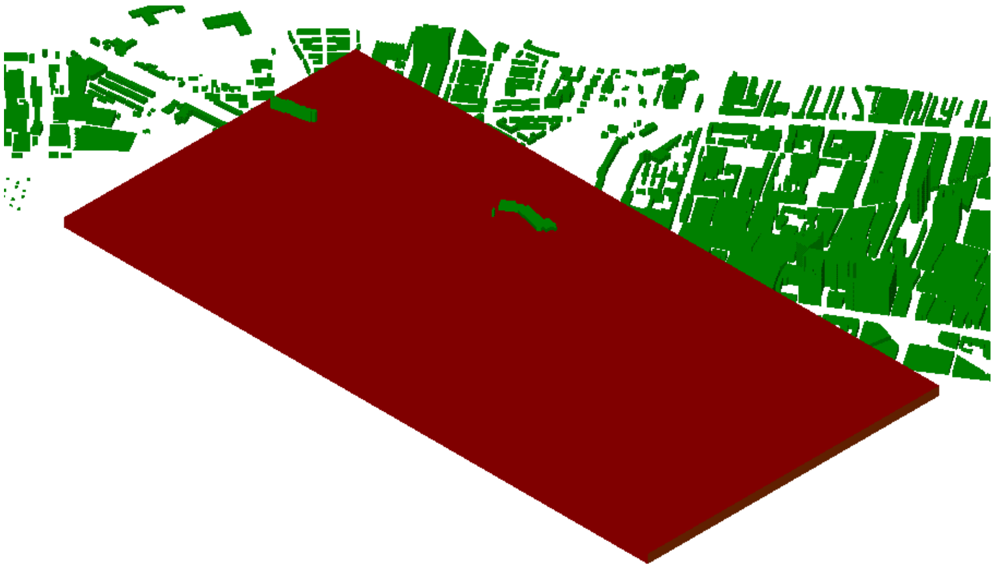

Fig 32 The buildings in the dataset (green) and a query box (red).

The tables used in these test look like this:

```
CREATE TABLE buildings_table (
id NUMBER,
geometry MDSYS.SDO_GEOMETRY);

CREATE TABLE querywindow (
id NUMBER,
geometry MDSYS.SDO_GEOMETRY);
```

The first table contains 1348 polyhedra representing buildings and the second table is defined twice, first containing 11 boxes from 0-50m and secondly containing 11 boxes from 20-50m.

SDO_FILTER is the Oracle Spatial function that uses the spatial index to select candidates for spatial queries. It is the only Oracle Spatial function that works in 3D (in connection with the 3D R-tree). The following SQL-statement shows how to use this filter to retrieve the number of candidates (5th and 7th column in Table 4 and Table 5):

```
SELECT COUNT(id) FROM buildings_table WHERE
SDO_FILTER(geometry, ,(SELECT geometry FROM querywindow WHERE id=1), 'querytype
= WINDOW')='TRUE';
```

To retrieve the number of actual intersections (2nd column in Table 4 and Table 5), a 3D Boolean intersection function is implemented (§5.2). The function can be used in an SQL-statement as follows:

```
SELECT COUNT(id) FROM buildings_table WHERE
intersection(geometry,(SELECT geometry FROM querywindow WHERE id=1),0.05)=1;
```

In a normal query you would combine the spatial filter with the intersection function like this (§7.2):

```
SELECT COUNT(id) FROM buildings_table WHERE
SDO_FILTER(geometry, ,(SELECT geometry FROM querywindow WHERE id=1), 'querytype
= WINDOW')='TRUE'
AND intersection(geometry,(SELECT geometry FROM querywindow WHERE
id=1),0.05)=1;
```

Table 4 shows the results of the test with the first set of boxes (0 – 50m):

Query box	Number of actual intersections	No spatial index		2D R-tree		3D R-tree	
		Number of candidates	Efficiency	Number of candidates	Efficiency	Number of candidates	Efficiency
1	7	1348	0,52%	7	100,00%	7	100,00%
2	71	1348	5,27%	71	100,00%	71	100,00%
3	180	1348	13,35%	180	100,00%	180	100,00%
4	281	1348	20,85%	281	100,00%	281	100,00%
5	395	1348	29,30%	395	100,00%	395	100,00%
6	509	1348	37,76%	510	99,80%	510	99,80%
7	614	1348	45,55%	615	99,84%	615	99,84%
8	740	1348	54,90%	741	99,87%	741	99,87%
9	849	1348	62,98%	851	99,76%	851	99,76%
10	910	1348	67,51%	912	99,78%	912	99,78%
11	1324	1348	98,22%	1348	98,22%	1324	100,00%

Table 4 Efficiency of a spatial index (intersection with boxes 0 – 50m).

Table 4 shows that the efficiency of a 2D spatial index and a 3D spatial index is equally high. The small differences between the two indices are because some (24) buildings are entirely below NAP (negative height), while the 3D query boxes only select the buildings above NAP-level. A spatial index is less efficient when the query window is larger, i.e. more buildings of the dataset are inside the box. Note that in this test the overhead of using the spatial filter was negligible compared to the time it took to do the intersections. This is usually the case, that is why spatial indices are invented; they improve performance, e.g. in this test the query for box 7 using the 2D R-tree would be more than 2 times faster, because only 615 instead of 1348 objects have to be intersected (Table 5).

Table 5 shows the results of the test with the second set of boxes (20 – 50m):

Query box	Number of actual intersections	No spatial index		2D R-tree		3D R-tree	
		Number of candidates	Efficiency	Number of candidates	Efficiency	Number of candidates	Efficiency
1	0	1348	0%	7	99,99%	0	100%
2	8	1348	0%	71	11,27%	8	100%
3	20	1348	0,01%	180	11,11%	20	100%
4	39	1348	0,03%	281	13,88%	39	100%
5	49	1348	0,04%	395	12,41%	49	100%
6	59	1348	0,04%	510	11,57%	59	100%
7	62	1348	0,05%	615	10,08%	62	100%
8	71	1348	0,05%	741	10,00%	71	100%
9	85	1348	0,06%	851	0,10%	85	100%
10	93	1348	0,07%	912	0,10%	93	100%
11	144	1348	0,11%	1348	0,11%	144	100%

Table 5 Efficiency of a spatial index (intersection with boxes 20 – 50m).

Table 5 shows a different result. This query basically selects all buildings higher than 20m NAP in a certain area. This results in less actual intersections. The consequence is that not using a spatial index is very inefficient, because an intersection has to be performed on all of the buildings in the dataset. There is a large difference between the 2D and 3D R-tree now. The 2D filter returns buildings of all heights in a certain area, while the 3D filter only returns the desirable buildings that are higher than 20m NAP. This means that when the query window gets larger, eventually all buildings are returned as candidates by the 2D filter. In this case the 3D filter performs so good that all of the candidates are actual intersections, which results in the efficiency being 100% for each box. Of course, this efficiency is not always attained, especially not in cases such as in Fig 27.

With the knowledge that the overhead of a 2D R-tree and a 3D R-tree are both relatively small, there is no reason to build a 2D R-tree on the dataset. The 3D R-tree performs as well as the 2D R-tree in case the query window contains the ground level height (because this is the height where the 2D R-tree works on), but it performs a lot better when this query window does not contain the ground level height.

5 3D functions

The standard functions in Oracle, just as in most Geo-DBMSs, only work with the projection of these objects on 2D coordinate space, because the third dimension is ignored. This is illustrated in the following example where a polygon in 3D coordinate space is defined (Fig 33).

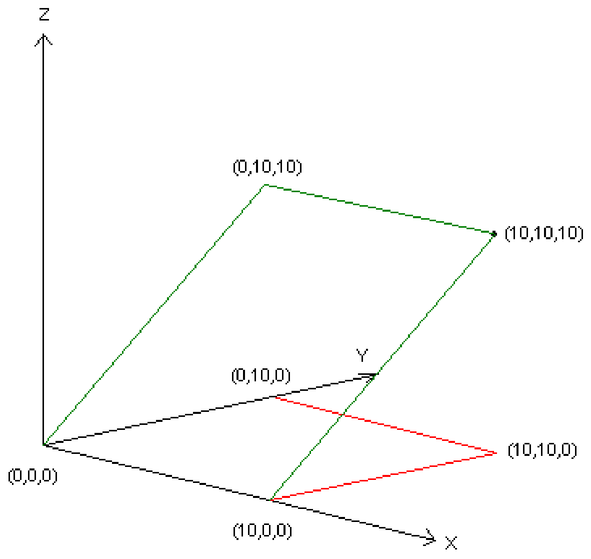

Fig 33 Polygon (green) in 3D space and its projection (red).

```
INSERT INTO test(id, geom) VALUES (1,
SDO_GEOMETRY(3003,NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1),
SDO_ORDINATE_ARRAY(0, 0, 0, 10, 0, 0, 10, 10, 10, 0, 10, 10)));
```

The polygon has an area of:

$$\left(10\sqrt{2}\right)*10 = 100\sqrt{2} \approx 141$$

and a perimeter of:

$$2*\left(10\sqrt{2}\right)+2*10 = 20\sqrt{2}+20 \approx 48$$

The result of the following query shows something different:

```
SELECT sdo_geom.sdo_area(geom,0.05), sdo_geom.sdo_length(geom,0.05) FROM test;

SDO_GEOM.SDO_AREA(GEOM,0.05) SDO_GEOM.SDO_LENGTH(GEOM,0.05)
---------------------------- ------------------------------
                 100                 40
```

This shows that the computations are done on the projection of the polygon on 2D coordinate space. Therefore some new 3D functions are implemented that *do* work with the 3D coordinates. Most of these functions only work on the polyhedron primitive, but some functions (e.g. area) also work on 3D polygons. It is clear that functions in 3D require more complex algorithms than 2D functions. This also has a big influence on the computational complexity. To maintain good performance, the emphasis should be on keeping the algorithms as efficient as possible. Spatial datasets can contain many objects, so a slightly more efficient algorithm will already yield noticeable better performance when querying all these objects.

In order to get high performance and avoid unnecessary conversions and data communication between DBMS and client, the data should be queried in the Geo-DBMS itself. This can be done by storing procedures or functions as part of the database. These stored procedures and functions can be written in PL/SQL or Java, both of them using SQL to access the data. With the help of the spatial index (chapter 4) this leads to good performance. The spatial index can be used in spatial queries by adding the SDO_FILTER function in the WHERE-clause of an SQL-statement (example in §4.3).

There are many functions possible. This chapter describes some basic functions that are often used with spatial objects in GIS applications [4]. The functions are ordered by their input parameters (one or two objects) and their return type. Table 6 has an overview of the types of functions, in which paragraph they are described, which functions are implemented and which functions are not implemented, but nice to have:

Function type	Paragraph	Implemented functions	Not implemented functions
Functions that are used in the conversion	§5.1	Multi-polygon conversion and vice-versa VRML-file conversion and vice-versa Footprint + height data conversion Topology conversion Validation Orientation fix	
Functions that return a Boolean	§5.2	Point-in-polyhedron Interaction test	More specific interaction test
Unary functions that return a scalar	§5.3	Area Volume Perimeter	
Binary functions that return a scalar	§5.4	Distance between average coordinates	Minimum distance between objects Maximum distance between objects
Unary functions that return simple geometry	§5.5	Bounding box Average coordinate Footprint Transformation (scaling, translation, rotation)	Buffer Centre of mass/gravity Circumscribed sphere Convex hull Inscribed sphere Point that is certainly inside polyhedron
Binary functions that return simple geometry	§5.6	Line segment between average coordinates of two polyhedra	Set operations
Functions that return complex geometry	§5.7	-	Tetrahedrisation Skeletonisation Shortest path

Table 6 Overview of (not) implemented functions.

Note that examples of the implementation of these functions in this chapter can be found in chapter 7 and a manual in Appendix B. Appendix C has an example Java source (volume computation).

5.1 Functions used in the conversion

These functions are used when converting spatial data to the polyhedron type and back. There are a number of conversion functions implemented. These are the present possibilities to convert spatial data in other formats to the polyhedron type and vice versa:

- It is possible to manually insert a polyhedron into a record of a spatial table in the database. This option requires basic knowledge of SQL and is a time-consuming job subject to many errors. Therefore, it is recommended to use this option just for testing. The vice versa function simply consist of the SELECT statement in SQL.

- The second option is to convert multi-polygons (standard type in Geo-DBMS) that together form a polyhedron to the polyhedron type itself. This means that if spatial data is available in this format or if spatial data can be stored in this format, these data can be converted to a real 3D primitive. The vice versa function works exactly opposite and is especially useful to visualise the polyhedra. An advantage is that data can be inserted by GIS/CAD front-ends. This option is described in §6.1.

- The third option is to create a polyhedron table in the database from a VRML-file. The vice versa function can create a VRML-file from the polyhedra. This is especially useful for visualisation on the Internet. This option is extensively described in §6.2.

- Then there is a function that converts a body, face and node table (topology) to the polyhedron type.

- From the footprint (2D polygon) and height data, volumes can be created and added to a database. There is no direct vice versa function. The footprints can e.g. come from a base map and the height data from laser scanning [13].

Once the conversion to the polyhedron type has taken place, the user can decide to validate the polyhedra to see if they are correctly modelled. This is recommended, because all other DBMS functions expect the polyhedra to be valid. The validation function is described in chapter 3. Related to the validation function is the function to correct the orientation of the faces of a polyhedron (fix_orientation, Appendix B).

5.2 Functions that return a Boolean

These functions return a Boolean, i.e. true or false. A well-known Boolean function is the point-in-polyhedron function. This function determines whether a point is inside a polyhedron or not. For the implementation of this function an algorithm in [26] is used:

- Generate a random unit vector. The point to test plus the direction of this vector form a random ray away from the point. The choice for a random vector is made, because if a fixed vector is chosen, there is a chance that the vector will intersect with the boundary of the polyhedron. This results in undesirable results in the function.
- Test for each plane if the ray intersects with it.
- If the number of intersections is even, then the point is outside the polyhedron, if this number is odd, then the point is inside the polyhedron.

A problem arises with this algorithm. If the ray hits the boundary of one or more of the faces, it is undetermined if the point is inside or outside the polyhedron. The solution is that if the ray hits a boundary of a face, then the algorithm is started over with a different random ray. If the ray intersects with the boundary again, then another random ray is generated and so on until a good ray is found.

Other functions that return a Boolean are topological relationship functions. These functions return true or false reflecting if a certain topological relation exists between two objects. The 9-intersection model [9] shows what relationships can exist between objects of different (or the same) dimensions, e.g. two polyhedra can have the following topological relationships (Fig 34):

- Intersect
- Disjoint (two polyhedra do not interact)
- Equals (special case of intersect where the two polyhedra are exactly the same)
- Touches (special case of intersect where the intersection is a point, line or plane)
- Within (one object is totally contained by the other)
- Contains (one object contains the other, opposite of within)

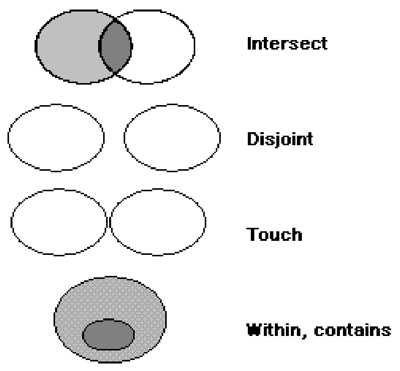

Fig 34 Topological relationships.

In Oracle Spatial there is a 2D function called SDO_RELATE. This function implements the 9-intersection model. With this function, topological relationships can be found. The parameter of SDO_RELATE used most often is ‘ANYINTERACT’, which test if two geometries interact, in any kind of way. For the polyhedron type in the DBMS a similar function has been implemented. Note that this function only supports the equivalent of the ‘ANYINTERACT’-parameter and does not allow searching for special cases of interaction. The other limitation is that the function only works on two polyhedra and not on a polyhedron in relation with a 0D, 1D or 2D object.

The first test is to see if one of the edges of the first polyhedron intersects with the second polyhedron (query window). As soon as an edge is found to intersect one of the faces of the second polyhedron, the function returns true and ends:

```
FOR EACH edge OF polyhedron_A
{
   FOR EACH face OF polyhedron_B
   {
      IF intersects(edge,face)
         RETURN TRUE
      ELSE
         CONTINUE
   }
}
RETURN FALSE
```

If none of the edges intersect with the second polyhedron, then the first polyhedron is either completely outside or completely within the second polyhedron. If one of the vertices of the first polyhedron is inside the second polyhedron (the point-in-polyhedron function is used here) then the program returns true and ends. There is one more problem, if the query window is the first parameter in the function and the to be tested polyhedron the second, the algorithm above does not work, because the edges of the query window may not intersect any of the faces of the polyhedron and no points of the query window may be inside the polyhedron. Therefore, the function is run a second time, but then with the input polyhedra switched. This second run is only performed, if no intersection was detected in the first run. This algorithm is shown in Fig 35.

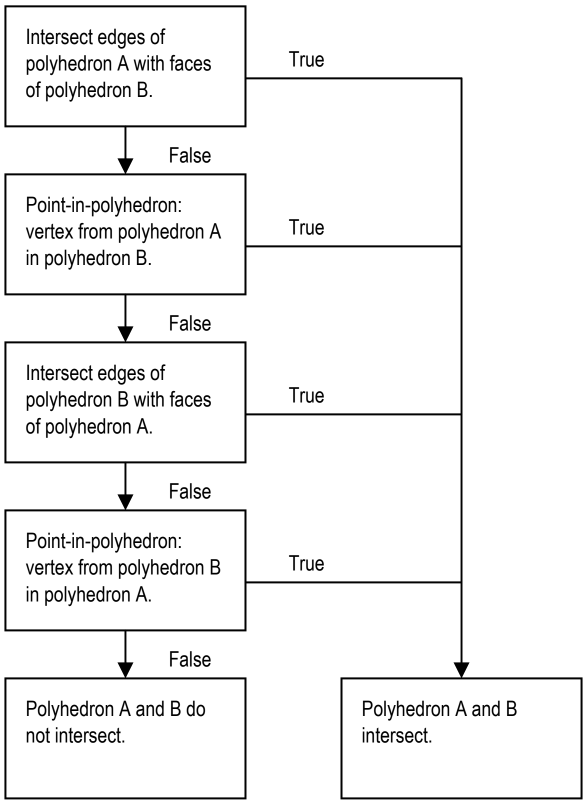
Intersect edges of polyhedron A with faces of polyhedron B.
True
False
Point-in-polyhedron: vertex from polyhedron A in polyhedron B.
True
False
Intersect edges of polyhedron B with faces of polyhedron A.
True
False
Point-in-polyhedron: vertex from polyhedron B in polyhedron A.
True
False
Polyhedron A and B do not intersect.
Polyhedron A and B intersect.

Fig 35 Interaction test.

The topological relationship function works a lot faster when using a spatial index (chapter 4). To use the spatial index include the SDO_FILTER function in the query. This function can only be used in a SQL 'WHERE'-clause. By specifying SDO_FILTER as the first element in the 'WHERE'-clause, this function will make a quick selection of objects that will be tested by the relationship function.

5.3 Unary functions that return a scalar

These are functions that work on a single polyhedron (unary) that return a number (scalar). Three of these kinds of functions are implemented:

Area

This function returns the true 3D surface area for a 3D polygon or the summation of the area of all faces that span a polyhedron. The area of a face is computed by projecting the face on 2D coordinate space. This projection takes place on the largest component of the normal vector n of the face. This evades numerical problems [26]. The formula of the area of a face [26] then is:

$$Area(\mathrm{P}) = \frac{1}{2}\begin{cases} \frac{1}{|n_z|}\sum_{i=0}^{n-1} x_i\left(y_{i+1} - y_{i-1}\right), if |n_z| = \max_i |n_i| \\ \frac{1}{|n_y|}\sum_{i=0}^{n-1} x_i\left(z_{i+1} - z_{i-1}\right), if |n_y| = \max_i |n_i| \\ \frac{1}{|n_x|}\sum_{i=0}^{n-1} y_i\left(z_{i+1} - z_{i-1}\right), if |n_x| = \max_i |n_i| \end{cases}$$

Note that the area of an outer ring (e-type = 1006 or 2006) is added to the accumulated area and that the area of an inner ring (e-type = 1106 or 2106) is subtracted from the accumulated area.

Volume

This function returns the volume of a polyhedron (Appendix C). The general idea of the algorithm that is used here to compute the volume is to multiply the area of each face by a depth, just like one would compute the volume of a box. With a polyhedron this results in computing overlapping boxes for each face, but by using the right orientation of vertices (§2.2), these volumes are either positive or negative. By summing these volumes, the overlapping volumes disappear and the result is the right volume of the polyhedron as a whole. More details can be found in [26]. The function [26] is:

$$Volume(R) = \frac{1}{3}\sum_{i=0}^{n-1}\left((\hat{n}_i \cdot P_{0,i})Area(S_i)\right)$$

where:

- R is the polyhedron
- S is a face
- $\hat{n}_i$ is the unit normal vector of face i
- $P_{0,i}$ is the first vertex of face i (this point spans the plane together with the normal vector)
- n is the number of faces

Perimeter

This function returns the true 3D perimeter of a polyhedron. Summing the length of each edge for all faces does this. Because each edge is used twice, the result from the latter has to be divided by 2. The length of an edge is:

$$Length(E) = \sqrt{\left(P_e - P_s\right)^2} = \sqrt{(\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2}$$

i.e the square root of the dot product of the vector, between the ending point and the starting point, with itself.

5.4 Binary functions that return a scalar

Binary functions are functions that operate on two objects. The most common binary functions that return a scalar are distance functions. This allows us e.g. to find all objects within a certain distance of another object. There are several ways to define the distance between two objects, e.g.:

- Distance between the average coordinates
- Minimum distance between two objects (this is the true distance)
- Maximum distance between two objects

For the latter two this can be further defined into only looking at the distance between vertices, or also looking at the edges and ultimately at the faces themselves. It is clear that the last option is more difficult to implement than the first two.

The distance between the average coordinates of two objects is not as refined as the minimum distance between two objects, but already provides some functionality. Therefore, this type of distance function is implemented. It simply computes the average of the coordinates of the two objects and then computes the length of the vector between these two points. Note that these points might be outside the polyhedron.

5.5 Unary functions that return simple geometry

There are many unary functions possible that return simple geometry. With simple geometry is meant, geometry that represents single simple objects (like single points or a cube). There are many functions like this possible; these are implemented:

- Bounding box: This is the smallest possible orthogonal box around a polyhedron. This function is implemented by searching for the smallest x-, y- and z-coordinate and the largest x-, y- and z-coordinate. The first triplet forms the lower left front vertex and the second triplet forms the upper right back vertex of the box, looking to positive y and the x,z-plane. The other 6 vertices can simply be constructed from these two. The bounding box can be used as a simplified model of the polyhedron (Fig 28 on page 24).

- Average coordinate: The average coordinate is the average of all coordinates. This function is implemented by taking the average of all x-coordinates, all y-coordinates and all z-coordinates and constructs the average coordinate point from these three values.

- Footprint: This function returns a single polygon (with or without inner ring) defined in 2D coordinate space and forms the footprint of the polyhedron. This can be used to make 2D maps. The footprint is constructed by taking all the faces with a positive z-component of the normal, i.e. the faces that can be seen by looking straight down from the sky. The edges of these faces that form a boundary are selected and construct the 2D footprint (a polygon). If there are any inner rings then these are also added to the footprint. If the footprints of two polyhedra intersect they are drawn on top of eachother.

- Transformation: This function should return geometry as a result of scaling, translating or rotating. If the object is valid, just the vertices need to be changed; the face descriptions stay the same. The transformation is implemented as three functions:

    - Scaling: This function multiplies each vertex with a scalar. It is possible to scale x, y and z separately. The polyhedron (average coordinate) is first translated to the origin (0,0,0), so that the scaling takes place in all directions. After the scaling it is translated back to its original position.

    - Translation: This function translates each vertex with a translation vector.

    - Rotation: This function has two parameters, the rotation angle (θ) and the rotation axis. First the object (average coordinate) is moved to the origin (0,0,0), then it is rotated and then it is moved back to its original position. The rotation multiplies every vertex with a rotation matrix. The rotation axis defines the elements of the rotation matrix. There are three rotation matrices [10]:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

There are more examples; the following functions are not yet implemented:

- Buffer: This function creates a buffer around the polyhedron. Any spherical or cylindrical patches have to be approximated by a certain amount of flat faces.

- Centre of mass: This function returns the centre of mass (or centre of gravity) of the 3D primitive. This is a point geometry defined in 3D coordinate space. The average coordinate function provides an approximation of the centre of mass.

- Circumscribed sphere: The sphere around the polyhedron through its vertices.

- Convex hull: This function removes concavities in the polyhedron, e.g. through holes. This will result in a convex polyhedron.

- Inscribed sphere: The largest possible sphere inside the polyhedron.

- Point that is inside the polyhedron: Returns a point that is certainly inside the polyhedron. This can be used as label point in applications.

5.6 Binary functions that return simple geometry

A relatively simple function is the one that returns the line segment representing the distance between two objects. This function has been implemented; it is simply the line between the average coordinates of two polyhedra.

The most well known binary functions that return simple geometry are set operations. Set operations are often used to create new geometry from other geometry, e.g.:

- Intersection: Returns geometry that two spatial objects have in common.
- Difference: Returns geometry from two spatial objects without the part that they have in common.
- Subtraction: Returns the part of one polyhedron that does not overlap with another polyhedron.
- Union: Returns the geometry of two spatial objects as one geometry.

These are complex functions in case of general (non-convex) polyhedra and can be the sole subject of a Master thesis. They are therefore not implemented in this research. The intersection function for example can be implemented by intersecting all the faces of the first polyhedron with the second polyhedron and vice versa. This yields in two sets of faces that have to be connected in some way. This is a difficult task in case of general polyhedra. There is more literature available about these functions in case of simplexes and convex polyhedra (e.g [26]). [27] could be the step to the solution for general polyhedra.

5.7 Functions that return complex geometry

There is one more set of functions left. These functions work on a single polyhedron, but return geometry that has multiple parts. These functions are, just like set operations, complex functions in case of general polyhedra. These functions too, can be the topic of a sole Master thesis and are thus not implemented here. Three examples of functions that return complex geometry are:

- Tetrahedrisation: This will divide the polyhedron in multiple tetrahedra. This is the 3D equivalent of triangulation.

- Skeletonisation: This will return the skeleton of a polyhedron. The skeleton of a polyhedron consists of the heart lanes of the polyhedron, e.g. the skeleton of a tunnel is a 3D line along the driving direction.

- Shortest path: The shortest path between two points on the boundary of a polyhedron.

6 Visualisation

To visualise 3D objects it is necessary to use programs that actually can show the third dimension. It is possible to make a viewer, but easier and better to use existing programs and convert the 3D data to a format readable for these programs. There are basically two options:

- GIS/CAD programs that can make a DBMS connection like Microstation and ArcScene. These programs can only handle 3D objects that consist of multiple 2D objects (the present situation described in the introduction). The 3D data stored as a 3D type needs a conversion before it can be visualised, e.g. splitting up the 3D object in multiple 2D polygons.

- VRML (Virtual Reality Modelling Language). When using VRML, there needs to be a translation between the 3D type in the database and the VRML syntax.

The first option is described in §6.1 and the second in §6.2.

6.1 GIS/CAD software

There are numerous GIS- and CAD programs (e.g. Microstation and ArcScene) that can make a DBMS connection to display geometry that is stored in a spatial database. These programs can only handle 3D objects that consist of multiple 2D objects (the present situation described in the introduction). The 3D data stored as polyhedron needs a conversion before it can be visualised. This conversion has to split up the polyhedron in multiple (2D) polygons (with 3D coordinates).

There is a multi-polygon geometry type in Oracle. The gType is either 3004 or 3007 (3007 is a specialisation of 3004, because 3004 can contain any geometry and 3007 can only contain polygons). These types can store multiple 3D-polygons in one record. The difference with the polyhedron type is that there is no separation between coordinates and face descriptions, i.e. there are only face descriptions that are built up by listing the coordinates. Besides the fact that no validation can be performed, the main disadvantage is that the same coordinates are listed multiple times and there is no information about outer or inner boundaries of the polyhedron.

The conversion function from the multi-polygon type to the polyhedron type converts the integrated coordinate/face descriptions to one that first lists the coordinates and then the face descriptions with references to these coordinates. Furthermore, the information about the elements need to be set to the Oracle standard. The vice-versa conversion function does the reverse, i.e. from the polyhedron type to the multi-polygon type.

Example
This example shows a cube with a ‘hole’ stored as multi-polygon type:

```
mdsys.sdo_geometry(3007,null,null,--3007=multi-polygon
mdsys.sdo_elem_info_array(
1,1003,1, 16,1003,1, 31,1003,1, 46,1003,1,--1003=outer ring
61,1003,1, 76,2003,1, 91,1003,1, 106,2003,1,--2003=inner ring
121,1003,1, 136,1003,1, 151,1003,1, 166,1003,1),
mdsys.sdo_ordinate_array(
0,0,0, 3,0,0, 3,0,3, 0,0,3, 0,0,0,--face 1
0,0,3, 3,0,3, 3,3,3, 0,3,3, 0,0,3,--face 2
0,3,0, 0,3,3, 3,3,3, 3,3,0, 0,3,0,-- etc.
0,0,0, 0,3,0, 3,3,0, 3,0,0, 0,0,0,
3,0,0, 3,3,0, 3,3,3, 3,0,3, 3,0,0,
3,1,1, 3,1,2, 3,2,2, 3,2,1, 3,1,1,
0,0,3, 0,3,3, 0,3,0, 0,0,0, 0,0,3,
0,2,1, 0,2,2, 0,1,2, 0,1,1, 0,2,1,
0,1,1, 0,1,2, 3,1,2, 3,1,1, 0,1,1,
0,1,2, 0,2,2, 3,2,2, 3,1,2, 0,1,2,
0,2,2, 0,2,1, 3,2,1, 3,2,2, 0,2,2,
0,2,1, 0,1,1, 3,1,1, 3,2,1, 0,2,1))
```

The following SQL statement returns the polyhedron type from the multi-polygon type on column geom. in table test:

```
SELECT return_polyhedron(geom) from test;
```

The result:

```
RETURN_POLYHEDRON(GEOM)
---------------------------------------------------------------------
SDO_GEOMETRY(3002, NULL, NULL,
SDO_ELEM_INFO_ARRAY(1,2,1, 49,0,1006, 53,0,1006, 57,0,1006,
61,0,1006, 65,0,1006, 69,0,1106, 73,0,1006, 77,0,1106, 81, 0, 1006,
85,0,1006, 89,0,1006, 93,0,1006), -- 1106 is hole in faces
SDO_ORDINATE_ARRAY(0,0,0, 3,0,0, 3,0,3, 0,0,3, 3,3,3, 0,3,3,
0,3,0, 3,3,0, 3,1,1, 3,1,2, 3,2,2, 3,2,1, 0,2,1, 0,2,2, 0,1,2,
0,1,1, -- coordinates
1,2,3,4, 4,3,5,6, 7,6,5,8, 1,7,8,2, 2,8,5,3, 9,10,11,12,
4,6,7,1, 13,14,15,16, 16,15,10,9, 15,14,11,10, 14,13,12,11,
13,16,9,12 – faces
))
```

The conversion function returns the polyhedron type and this can be used in other SQL-statements. The vice-versa conversion function has the same functionality.

GIS- and CAD-programs can read from a table created using the conversion functions and stored multi-polygon data can be converted to the polyhedron type. The object from the example above is visualised in Fig 36 using Microstation.

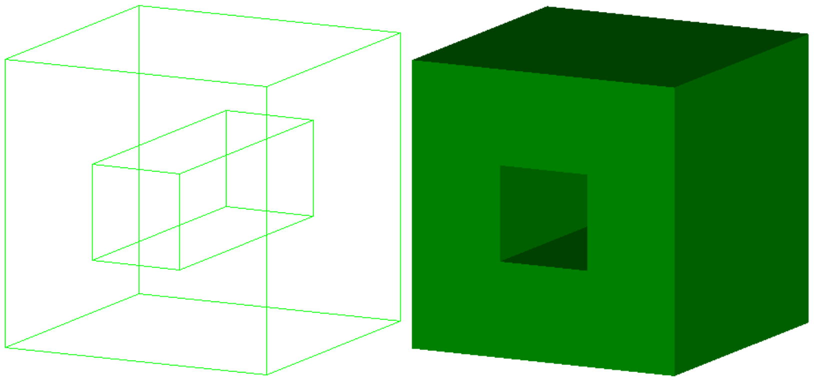

Fig 36 Object from the example above visualised in Microstation GeoGraphics: wireframe model left and rendered right.

6.2 VRML

VRML is a language that is used to make 3D virtual worlds that can be browsed on the Internet. A VRML-file (file-extension: .wrl) can contain different kinds of geometry. The type of geometry that is useful in this thesis is the IndexedFaceSet [8]. An IndexedFaceSet is closely related to the storage of the polyhedron type, because it also has a list of coordinates and face descriptions pointing to these coordinates. It has less information though, because inner rings are not explicitly recognisable. Inner rings can be specified by creating an edge from and to the outer ring. It does not matter if one of these edges intersects with another inner ring; VRML handles this.

There is an extra step to convert the VRML-file to the polyhedron type: first the VRML-file is stored as an SQL-loader file. With the Oracle tool SQL-loader this file can be loaded into a database to construct a table from all the geometries listed in this file. This extra step is taken, because it gives the possibility to convert VRML-files without a DBMS connection and it is more efficient to load all geometries in one run into the database than one by one.

Each IndexedFaceSet in the VRML-file corresponds to one polyhedron object in a database. To retrieve the IndexedFaceSets from a VRML-file, a Java package called CyberVRML97 for Java [25] is used. From here, the coordinates and the face descriptions have to be formed to the storage model of the polyhedron type and exported to the SQL-loader file.

The vice-versa function does not use the CyberVRML97 for Java package. Here the data from the database is written to a VRML file directly, because each geometry in the database has to be retrieved anyway.

Note that both functions work outside the DBMS, because the VRML-files are not inside the DBMS. The object used in this chapter is visualised in VRML in Fig 37.

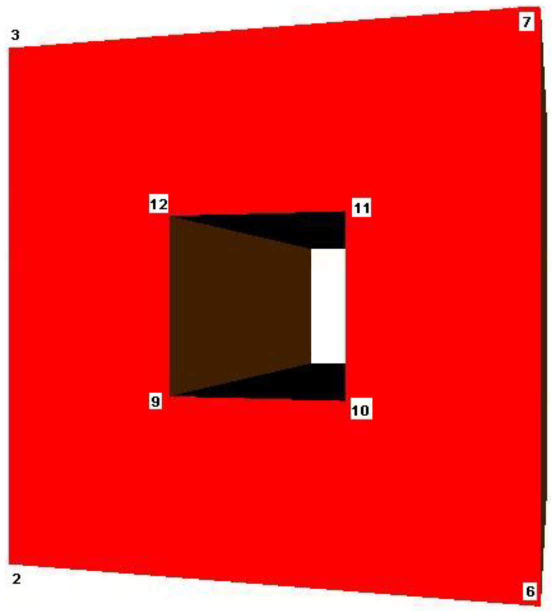

Fig 37 Visualisation of the object used as example in this chapter in VRML.

Example

This examples shows how the object from Fig 37 is converted from polyhedron to VRML-file. The SQL-statements to create this object are:

```
create table fig(
id NUMBER,
geom MDSYS.SDO_GEOMETRY);
```

```
insert into fig values (5, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
49,0,1006, 53,0,1006, 57,0,1006, 61,0,1006,
65,0,1006, 69,0,1106, --one side through hole
73,0,1006, 77,0,1106, --other side through hole
81,0,1006, 85,0,1006, 89,0,1006, 93,0,1006),
mdsys.sdo_ordinate_array(
10,-4,0, 13,-4,0, 13,-4,3, 10,-4,3, 10,-1,0, 13,-1,0, 13,-1,3, 10,-1,3,
13,-3,1, 13,-2,1, 13,-2,2, 13,-3,2, 10,-2,1, 10,-3,1, 10,-3,2, 10,-2,2,
1,2,3,4, 4,3,7,8, 5,8,7,6, 1,5,6,2, 2,6,7,3, 9,12,11,10, 4,8,5,1, 13,16,15,14,
14,15,12,9, 15,16,11,12, 16,13,10,11, 13,14,9,10)));
```

To convert this object to a VRML-file:

```
java geom3d.SdoToVRML fig geom fig.wrl
```

The result is the VRML-file fig.wrl:

```
#VRML V2.0 utf8
Background {
    skyColor [
        0.0 0.2 0.7,
        0.0 0.5 1.0,
        1.0 1.0 1.0
    ]
    skyAngle [1.009, 1.571]
}
Transform {
    rotation 1 0 0 -1.571
    children [
Shape {
    appearance Appearance {
        material Material {
            diffuseColor 1 0 0
    }}
```

```
        geometry IndexedFaceSet {
            convex FALSE
        coord Coordinate {
            point [
    10.0 -4.0 0.0, 13.0 -4.0 0.0, 13.0 -4.0 3.0, 10.0 -4.0 3.0, 10.0 -1.0 0.0,
    13.0 -1.0 0.0, 13.0 -1.0 3.0, 10.0 -1.0 3.0, 13.0 -3.0 1.0, 13.0 -2.0 1.0,
    13.0 -2.0 2.0, 13.0 -3.0 2.0, 10.0 -2.0 1.0, 10.0 -3.0 1.0, 10.0 -3.0 2.0,
    10.0 -2.0 2.0,
            ]}
        coordIndex [
    0, 1, 2, 3, 0, -1,
    3, 2, 6, 7, 3, -1,
    4, 7, 6, 5, 4, -1,
    0, 4, 5, 1, 0, -1,

    1, 5, 6, 2, 1, 8, 11, 10, 9, 8, 1, -1,
    3, 7, 4, 0, 3, 12, 15, 14, 13, 12, 3, -1,
    13, 14, 11, 8, 13, -1,
    14, 15, 10, 11, 14, -1,
    15, 12, 9, 10, 15, -1,
    12, 13, 8, 9, 12, -1,
            ]}
    }
    ]}
```

After the file header, the geometry is specified, in this case an IndexedFaceSet (bold in example file). Just like in the SQL-statement, first the coordinates are specified and then the faces. Note that the end of a face is specified with –1, the first vertex is repeated at the end and that the nodes go from 0 to n-1 instead of 1-n. Special cases are the inner rings in two faces. In SQL these are interpretation code 1006 followed by interpretation code 1106, i.e. two elements. In VRML this is converted to one element.

In SQL:

```
65,0,1006, 69,0,1106, → 2,6,7,3, 9,12,11,10,
73,0,1006, 77,0,1106, → 4,8,5,1, 13,16,15,14
```

In VRML (if every node is added by 1):

```
2, 6, 7, 3, 2, 9, 12, 11, 10, 9, 2, -1,
4, 8, 5, 1, 4, 13, 16, 15, 14, 13, 4, -1,
```

Note that the inner ring is pasted after the repeated first node of its belonging face. The last node of the inner ring is also repeated and followed by the first node of its belonging face. This way a double edge is formed between the inner ring and its belonging face (edge 2-9 and 4-12). This causes the object to be displayed correctly.

7 Test case

The prototype that is described in this research needs testing to see if it works satisfactory. Therefore, some test objects are modelled and stored in the database (§7.1) to test the variety of storage options. Furthermore, real data is inserted in the database (§7.2) to test larger datasets. All data are then tested extensively to make sure that all the functions work properly. The manual for the functions is in Appendix B.

7.1 Test objects

The test objects are chosen to use a variety of the storage possibilities. There are 5 different objects (Fig 38):

- A tetrahedron (1)
- A cube (2)
- A cube with a dent in one of the faces (3)
- A hollow cube (4)
- A cube with a through hole (5)

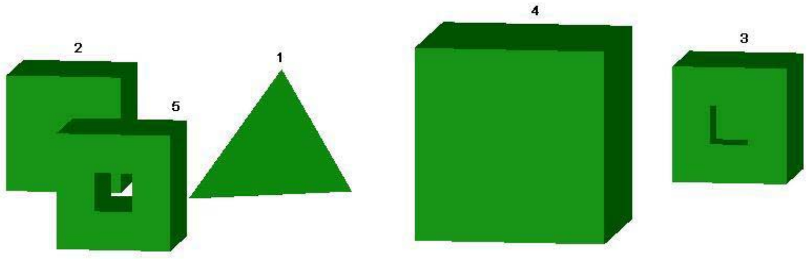

Fig 38 The test objects. Note that test object 4 is hollow, but this is not visible.

The SQL-statements to insert these test objects are in Appendix A. The tetrahedron is situated in the origin of the coordinate system. The other 4 objects are each situated in a separate quadrant. In all functions, tolerance value 0.01 is used.

We are now ready to validate the objects. The objects are stored in a table with the name ‘testobjects’. This table has an id column (id) and a geometry column (geom):

```
CREATE TABLE testobjects (
id NUMBER,
geom MDSYS.SDO_GEOMETRY);
```

The following SQL-statement validates these 5 objects:

```
SELECT id, validate_polyhedron(geom,0.01) FROM testobjects;
```

The result:

```
ID    VALIDATE_POLYHEDRON(GEOM,0.01)
-----------------------------------------------------------------------------
1     True
2     True
3     True
4     True
5     True
```

All the test objects are valid objects (examples of invalid objects are in chapter 3).

The next function tests if a point is inside a polyhedron. The point to be tested is just above the origin of the coordinate system (0,0,1). The point-in-polyhedron function should return true (1) for the tetrahedron and false (0) for all other objects. The SQL-statement:

```
SELECT id, point_in_polyhedron(geom, 0,0,1, 0.01) from testobjects;
```

The result:

```
        ID POINT_IN_POLYHEDRON(GEOM,0,0,1,0.01)
---------- ------------------------------------
         1                  1
         2                  0
         3                  0
         4                  0
         5                  0
```

Object 1, the tetrahedron, returns 1, which means the point is contained in that object. What if we choose a point in the through hole of the object 5? The centre point is (11.5,-2.5,1.5). The point-in-polyhedron function should return false (0) for all objects. The SQL-statement:

```
SELECT id, point_in_polyhedron(geom, 11.5,-2.5,1.5, 0.01) from testobjects;
```

The result:

```
    ID POINT_IN_POLYHEDRON(GEOM,11.5,-2.5,1.5,0.01)
---------- ------------------------------------
     1                 0
     2                 0
     3                 0
     4                 0
     5                 0
```

Indeed, it works.

Now the unary functions that return a scalar are going to be tested, i.e. area, volume and perimeter. The expected results can be computed by hand out of the coordinates in Appendix A. The SQL-statement to retrieve the answers from the database objects is:

```
SELECT id, area3d(geom), volume(geom), perimeter(geom) from testobjects;
```

The result:

```
     ID AREA3D(GEOM) VOLUME(GEOM) PERIMETER(GEOM)
------- ------------ ------------ ---------------
     1   22.9530689     5.5         22.0723224
     2   54             27          36
     3   58             26          48
     4   204            98          96
     5   64             24          56
```

The functions work properly: inner holes are subtracted from the total volume and concavities are not a problem.

It is also possible to compute the distance from the average coordinate of the tetrahedron to the average coordinates of the test objects. The following query also returns the average coordinate:

```
SELECT id, average coordinate(geom) AVERAGE COORDINATE,
distance(geom,(select geom from testobjects where id=1)) DISTANCE
FROM testobjects;
```

The result:

```
    ID      AVERAGE COORDINATE
DISTANCE
-------------------------------------------------------------------------------
     1      (0.5, -0.25, 0.75)                                         0
     2      (-6.5, -6.5, 0.5)                                          9.38749168
     3      (-3.090909090909091, 11.5, 1.5)                            12.2831185
     4      (7.5, 7.5, 2.5)                                            10.5889093
     5      (11.5, -2.5, 1.5)                                          11.2527774
```

Note that the distance is the 3D distance between the average coordinates, not the projected distance. The values returned by the function are the distance in 3D space.

The next SQL-statement creates a table named testobjects_bb with the bounding boxes for each test object. To visualise these objects in MicroStation the bounding boxes must be returned as multi-polygons. The function for this is also included in the SQL-statement (note the nested functions):

```
CREATE TABLE testobjects_bb AS
SELECT id ID, return_multipolygon(bounding_box(geom)) GEOM FROM testobjects;
```

The result is visualised in Fig 39:

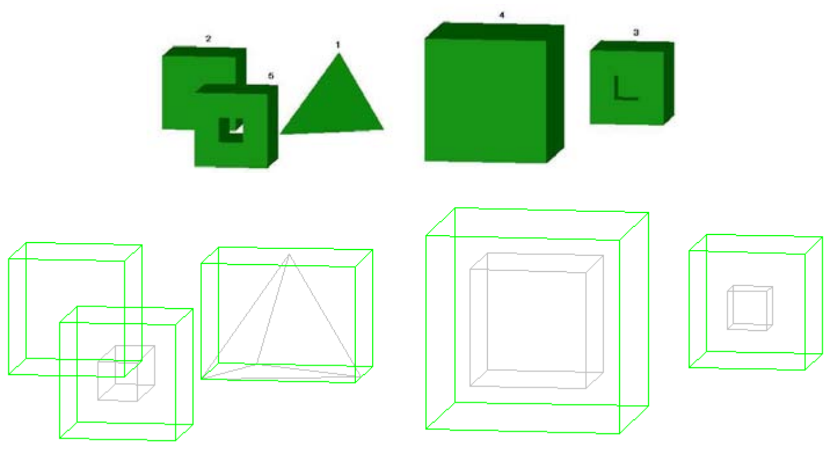

Fig 39 The test objects (above and grey in bottom) and their bounding boxes (bottom green).

Note that most of the bounding boxes overlap with the test object, but the tetrahedron gives a good example of the correctness of this function. The conversion function to multi-polygons works flawless too.

The next functions to be tested are the translation and rotation function. The test objects are rotated over 45 degrees (about 0,7 radials) along the x-axis. They are also translated by (25,25,0). The results are saved as multi-polygon tables (rotated and trans) for visualisation. The SQL-statements are:

```
CREATE TABLE rotated AS
SELECT id ID, return_multipolygon(rotation(geom,'x',0.7)) GEOM FROM
testobjects;

CREATE TABLE trans AS
SELECT id ID, return_multipolygon(translation(geom,25,25,0)) GEOM FROM
testobjects;
```

The result can be seen in Fig 40:

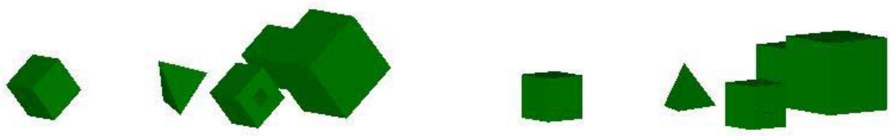

Fig 40 The rotated test objects left and the translated test objects right.

The rotated test objects are still situated at their original position and it can be seen in Fig 40 that the translation function works too. The scaling function is not tested here.

7.2 Real data

In this paragraph real data is used. The emphasis is more on the amount of data than on the variety of modelling and storage possibilities. Most of the objects are rather simple: they do not have any inner rings in the faces. However, some of the objects have through-holes.

First, some 3D data is made. This is done by combining the buildings of a 2D base map (TOP10) with data (3D points) from an airborne laser scanner (density 1 point/4m^2). These data cover 1348 buildings in Delft (Dutch city). The emphasis is on creating useful data, not to model the city as accurate as possible. Each building polygon in the 2D base map is assigned with the highest point from the laser data in the area covered by the polygon. This spatial overlay is performed by ArcInfo and then exported to a text file containing the height and coordinates of the building polygon.

The text file with the height and coordinates of the buildings can be converted to polyhedra by extruding the polygon by its height. This results in two polygons above each other (floor and roof) connected with walls that have the height specified in the text file. The following function (outside the DBMS) does this:

```
java geom3d.TOP10toSQL [baseheight] [inputfile] [outputfile]
```

The base height is the ground level height of the dataset (Delft: -0.5m NAP, NAP is Netherlands National Ordnance Datum). Note that only buildings on the surface can be converted in this way, not subsurface constructions. In the ideal case we would have had the bottom and top height of a building. The input file is the text file with the 2D buildings and the output file is the name of the Oracle SQL-loader file that is created by the conversion function. This SQL-loader file contains the polyhedra. These polyhedra can be loaded by:

```
sqlldr userid=[user]/[pass], control=[outputfile]
```

By providing the username and password for the database and the output file this command loads all data into a database table. Working with an SQL-loader file is faster than inserting the polyhedra one by one into the database (chapter 6).

The buildings are inserted in table top10 and are now ready to be validated. This time we only want to know which buildings are not valid polyhedra:

```
SELECT id FROM top10 WHERE NOT validate_polyhedron(geom,0.01)='True';
```

The result:

```
        ID
----------
       244
       368
       481
      1112
      1117
      1118
      1120
      1160
      1177
      1282
      1286
      1292
      1312
      1321
      1341
      1342
```

The result shows some invalid polyhedra. Errors arise where the roofs of objects are totally under NAP ground level. This is likely due to buildings in the 2D base map that do not exist anymore (the 2D base map is a couple of years older than the laser data). The top then equals the bottom and results in a polyhedron without a volume. They can be corrected using the SQL UPDATE command.

Now that all objects are valid, it is time to visualise the data. This time the buildings are exported to a VRML file. This function works outside the DBMS:

```
java geom3d.SdoToVRML [table name] [outputfile]
```

The table name in this case is top10 and the output file can be any file with the file extension .wrl. The result is shown in Fig 41.

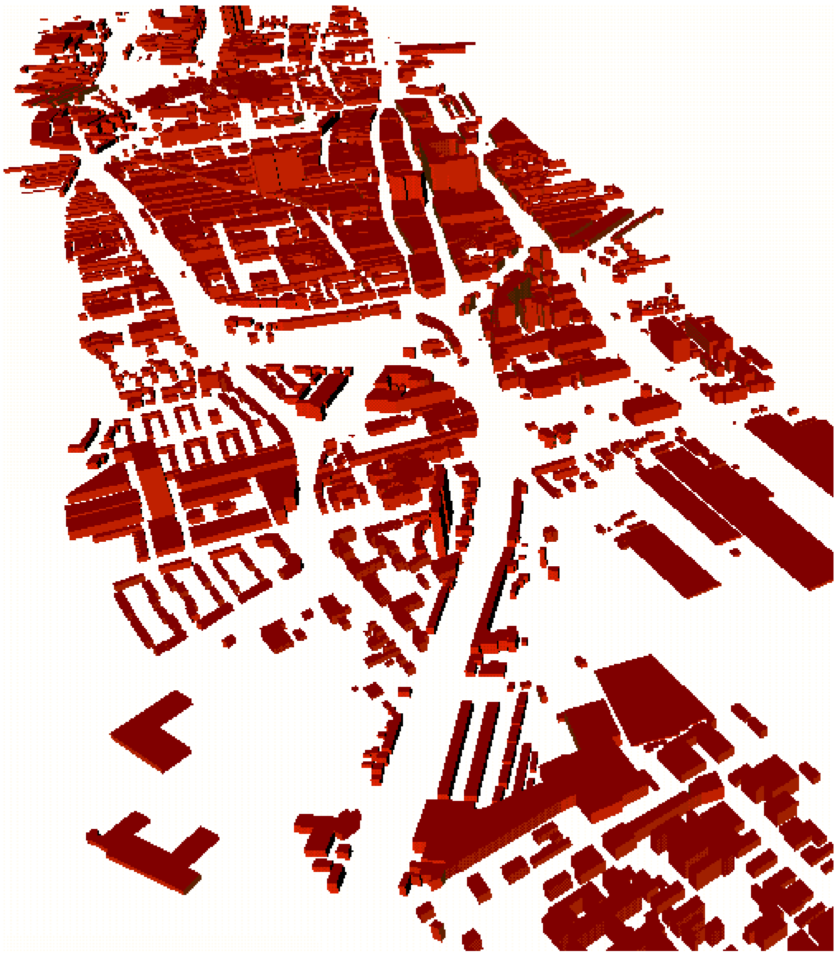

Fig 41 VRML file showing the buildings that are stored in the database. The city is about 5km north - south, the view is from the north.

Fig 41 shows a good result. If, for any reason, the 2D footprint of the buildings is necessary, then there is a function that can convert the polyhedra to 2D polygons. The result of this function should of course be equal to the 2D base map that was the source of the 3D buildings. The SQL-statement to compute the footprint and insert these in a table named footprint is:

```
CREATE TABLE footprint AS
SELECT footprint(geom.,0.01) FROM top10;
```

The result is shown in Fig 42 together with the original 2D base map.

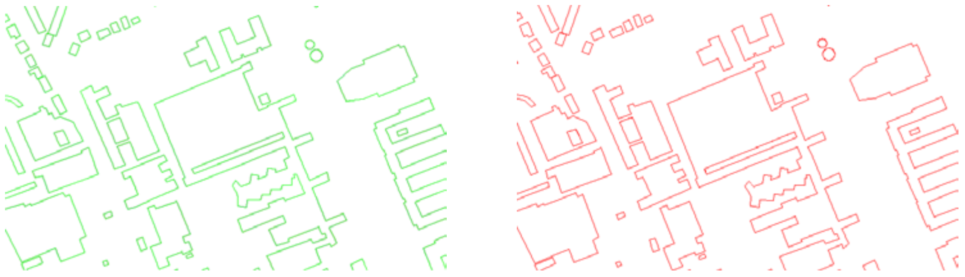

Fig 42 2D footprints derived from polyhedra (left, green) and 2D base map (right, red).

The maps in Fig 42 are exactly the same, which means that the footprint function is doing its job properly.

The power of GIS lies in its capabilities to perform spatial analyses. We might for example want to select all large (in volume) university buildings. This means we have to do an intersection with a query window that covers the university campus. This query window is stored in table querywindow, record #12. Then all buildings that are larger than a certain volume (here: 20000 m^3) have to be selected. If we want to visualise the result in Microstation, multi-polygons have to be created. The SQL-statement is:

```
CREATE TABLE result AS
SELECT id ID, return_multipolygon(geom) GEOM FROM top10 WHERE
volume(geom) > 20000
AND SDO_FILTER(geom,(select geom from querywindow where id=12),'querytype =
WINDOW')='TRUE'
AND intersection(geom,(select geom from querywindow where id=12),0.05)=1;
```

Note that the SQL-statement includes using the spatial index (SDO_FILTER, see chapter 4). It does not matter in which sequence the WHERE conditions are specified, this is optimized by Oracle. The result contains 67 buildings. They are shown in Fig 43.

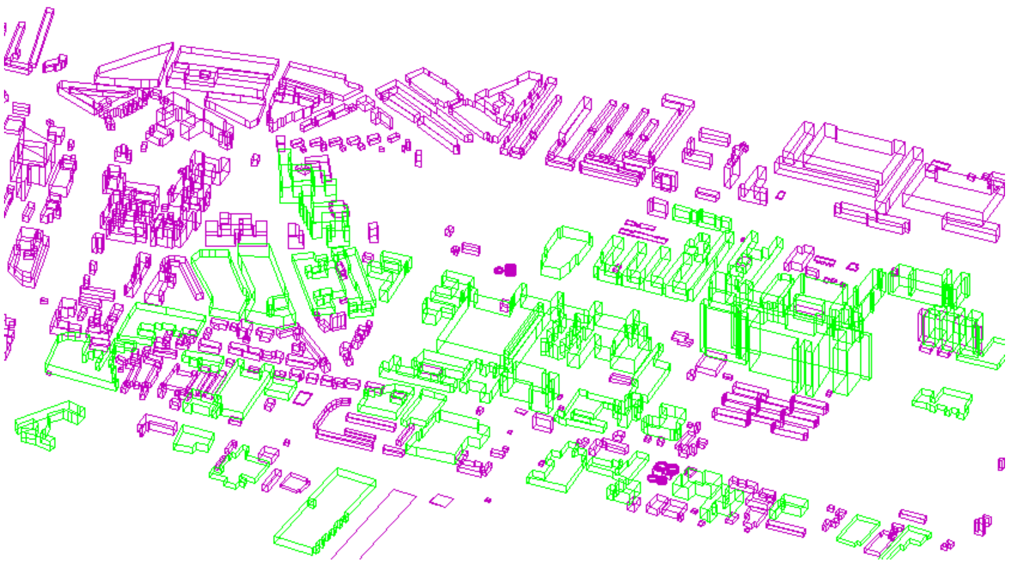

Fig 43 The result of querying the large building on the university campus (green). The purple buildings are not selected, because they are either too small or outside the university campus.

The last test compares the storage requirements for the polyhedron table (TOP10_3D) and the multi-polygon table (TOP_3D_MP) containing the 1348 buildings in Delft.

```
SELECT segment_name, COUNT(*) extents, SUM(bytes/(1024*1024)) size_in_mb,
sum(blocks) "Blocks" FROM user_extents;
```

Result:

```
SEGMENT_NAME                      EXTENTS SIZE_IN_MB  Blocks
------------------------------ ---------- ---------- -------
TOP10_3D                                2       1.00     128
TOP10_3D_MP                             4       2.00     256
```

It shows that the multi-polygon variant takes up twice as much space.

This paragraph has shown some possibilities on how to use the prototype system described in this thesis and proves that the system is indeed operational.

8 Conclusions and recommendations

8.1 Conclusions

There has already been a lot of research on the concepts of 3D data models. This research is a first attempt to implement a true 3D primitive in the Geo-DBMS including validation, conversion, indexing and spatial functions in 3D. The implementation described in this thesis enables users of Geo-DBMSs to add their 3D data and perform 3D queries on them. The added value above 3D CAD software is that Geo-DBMSs can store and manage information on objects and that this information can be queried by numerous other applications, while 3D CAD software focuses more on drawing and visualisation. The objective stated in chapter 1 was answering the following question:

How can 3D spatial objects be modelled (i.e. stored, validated, queried) in a Geo-DBMS using 3D primitives and how can these objects be visualised?

The answer to this question can be found in chapters 2 to 6:

- 3D Spatial objects are stored with the polyhedron as 3D primitive. This primitive is easy for a user to model, can fairly easily be validated, the algorithms for the geometric operations are not too difficult to implement and result in realistic objects (§2.1).

- The polyhedron is stored as a hierarchical boundary representation (§2.3), which means that the edges are not stored explicitly and vertices only need to be stored once. For each polyhedron is stored the set of faces, which consist of a set of ordered nodes. These nodes point to a vertex (x,y,z).

- To avoid errors in functions with deviations, because of floating point computations, a tolerance value is introduced for these functions. It is important not to choose this value equal to zero. A good value for the tolerance is the standard deviation of the geodetic measurements. This value is an input parameter in many of the functions and is also stored in the metadata table in Oracle (user_sdo_geom_metadata).

- The polyhedron is stored within the original Oracle Spatial geometry data model (§2.5). By setting some elements equal to zero, Oracle and other applications will think the polyhedron is a 3D line through all vertices of the polyhedron. The 3D functions however, recognize the object as a polyhedron.

- The validation occurs by checking if the polyhedra are stored correctly (§3.1) and after that checking each characteristic of the polyhedra (§2.2). These characteristics are: flat faces (§3.2), should bound one volume (§3.3), simplicit faces (§3.4) and orientable (§3.5). This order is chosen, because of the dependencies of the functions.

- To improve the performance of queries, a spatial index should be made on a table with polyhedra. The standard Oracle Spatial indices can be used, because of the way the polyhedra are stored in the Oracle Spatial geometry data model (§4.2). A bounding volume is constructed around the 3D line or its projection in case of a 2D spatial index.

- A test (§4.3) shows that it is preferable to create a 3D spatial index (3D R-tree) rather than a 2D spatial index, to get maximal query performance. In queries where the ground level is included, the indices have the same performance, but if the ground level is not included, a 3D spatial index is faster, because less candidates are selected.

- Using functions that are part of Oracle Spatial is not suitable for 3D objects, because these functions work with the 2D projection of the 3D objects. Instead, some of the most commonly used functions are implemented in 3D (chapter 5). Most of these work on the polyhedron primitive, but e.g. the area function also works on 3D polygons.

- The polyhedra can be visualised in GIS and CAD programs that can make a DBMS connection. To do this, the polyhedra have to be exported to 3D multi-polygons. This export function is implemented, as is the import function that makes a polyhedron from a 3D multi-polygon (§6.1).

- To visualise polyhedra in a VRML viewer, the objects in the database can be exported to a VRML file. This function is implemented, as is the function to make a polyhedron from a VRML object (§6.2).

These conclusions together satisfy the goal to implement a 3D primitive in a Geo-DBMS in a way that the maintainability of 3D spatial data improves and that the door is opened to more realistic applications (chapter 7).

8.2 Recommendations

Since this research was a first attempt to implement true 3D functionality including a 3D primitive, some recommendations for further research can be made:

- It is now clear what it takes to implement a 3D primitive. Instead of the polyhedron primitive, an even more realistic primitive can be researched to implement, e.g. the polyhedron with spherical and cylindrical patches.

- To implement the polyhedron primitive in Oracle Spatial, some not so logical constructions had to be made, e.g. using the 3D polyline sdo_gtype (3002) which confuses other applications, setting the e-type to zero and storing the vertices and face descriptions in one array. It is recommended that Oracle extends its geometry data model to 3D and includes an array that e.g. can hold shells. This will add support for a multi-polyhedron type (which can also be implemented now by increasing the number of interpretation codes). It is also important to add support for internal topology (vertices should only need to be specified once), so that storage requirements can be decreased.

- It is also interesting to know how to maintain a topological complex of 3D volumes. The materialisation function is already implemented, but more research needs to be done on e.g. integrity and performance of this complex data structure.

- In this research, local coordinates (x,y,z) are used. There are also applications where other coordinate systems are used (e.g. ETRS together with a height). These coordinates need to be transformed to local coordinates and then it should be possible to use the function described in this research. A closer look might also reveal that it is necessary to add nodes, edges or faces if the distances are very large compared to the tolerance value.

- This research does not contain any benchmarks of the system. This can be the topic of a case study, e.g. by using very large datasets to test the spatial index and the performance difference between 2D and 3D systems.

- Some more 3D functions can be implemented. This research has implemented some of the most common functions. There might also be faster algorithms available that increase overall performance. Examples of useful functions are tetrahedrisation and skeletonisation.

- A very interesting function could be the complete implementation of the 9-intersection model in 3D. This model is in principle the same as in other dimensions, but there also needs to be determined whether all relations are useful in 3D or if there are any interesting relations that are not covered by the 9-intersection model.

- It has been shown that objects in the database can be visualised in GIS/CAD programs. It is also possible to store (simple) objects in the database. It is interesting to know more about the ability of these programs to store complex geometry in a database. Theoretically this is possible, but a technical case study could show the practical possibilities.

References

[1] Stoter, J.E., Needs, possibilities and constraints to develop a 3D cadastral registration system, In: E.M. Fendel (eds.), Proceedings of UDMS 2000, UDMS 2000, 22nd Urban Data Management Symposium 'Urban and Rural Data Management Common Problems - Common Solutions' (Delft, 13-09-2000), UDMS/TU Delft, Delft, 2000, p. III.43-III.58, 2000.

[2] Stoter, J.E. and M. Salzmann, Towards a 3D cadastre: where do cadastral needs and technical possibilities meet?, In: Proceedings Interantional Workshop on 3D Cadastres, Registration of properties in strata, Delft, The Netherlands, 28-30 November 2001.

[3] Stoter, J.E. et al., Towards a 3D cadastre, In proceedings: FIG, ACSM/ASPRS, April 19-26-2002, Washington D.C. USA, 2002.

[4] Stoter, J.E. and P.J.M. van Oosterom, Incorporating 3D geo-objects into a 2D geo-DBMS, In proceedings: FIG, ACSM/ASPRS, April 19-26-2002, Washington D.C. USA, 2002.

[5] Zlatanova, S., 3D GIS for urban development, PhD thesis, ITC publication 69, Enschede, the Netherlands, 222 p., 2000.

[6] Kofler, R-trees for the Visualisation of Large 3D GIS Database, Ph.D. Thesis, Technical University, Graz, Austria, 1998.

[7] Coors, 3D-GIS in Networking Environments, In: Proceedings Workshop on 3D Cadastres, Delft, November, 2001.

[8] Andrea L. Ames, David R. Nadeau and John L. Moreland, The VRML 2.0 Sourcebook ,1997.

[9] OpenGIS Consortium, OpenGIS® Simple Features Specification for SQL, Revision 1.1, 1999.

[10] Nieuwenhuizen, van and Jansen, Computer graphics lecture notes, TU Delft, 2000.

[11] Teunissen and Van Oosterom, The creation and display of arbitray polyhedra in HIRASP, Rapportno. 88-20, Department of Computer Science, Leiden University, 1988.

[12] Cambray, Three-dimensional modelling in a geographical database, In: Auto-Carto'11: 11th International Conference on Computer Assisted Cartography, pages 338-347, Oct 1993.

[13] Zlatanova, S. and F.A. van den Heuvel, 3D city modelling for mobile augmented reality, 2002

[14] Guttman, A., R-Trees: A dynamic index structure for spatial searching, In: Proceedings of the 1984 ACM SIGMOD international conference on management of data, pp. 45-57, 1984.

[15] Oracle, Oracle 9i Spatial User Guide and Reference, Release 9.0.1, Part Number A88805-01, June 2001.

[16] Aguilera, A., Orthogonal polyhedra: study and application, PhD thesis, Barcelona, Spain, 2001.

[17] Molenaar, M., A Formal Data Structure for 3D Vector Maps, In: Proceedings of EGIS'90, Vol. 2, Amsterdam, The Netherlands, pp. 770-781, 1990.

[18] Pigot, S., A Topological Model for a 3-Dimensional Spatial Information System, PhD Thesis, University of Tasmania, Australia, 228 p., 1990.

[19] Pilouk, M., Integrated Modelling for 3D GIS, PhD thesis, ITC, the Netherlands, 200 p., 1996.

[20] Saadi Mesgari, M., Topological Cell-Tuple Structures for Three-Dimensional Spatial Data, PhD thesis, University of Twente and ITC, ITC Dissertation Number 74, Enschede, the Netherlands, 200 p., 2000.

[21] De Vries, J., 3D GIS en grootschalige toepassingen, De opslag en analyse in een geintegreerde driemensionale GIS, MSc thesis (in Dutch) GIS-technology, TU Delft, 2001.

[22] OpenGIS Consortium, The OpenGIS™ Abstract Specification, Topic 1: Feature Geometry (ISO 19107 Spatial Schema) Version 5, 2001.

[23] Li, Y. and J.X. Zhang, Least squares plane, http://www.infogoaround.org/JBook/LSQ_Plane.html, 2001.

[24] Rosenfeld, A., Tree structures for region representation, in Freeman, H. and G.G. Pierioni (eds), Map data processing, New York: Academic Press, pp. 127-150, 1980.

[25] Konno, S., CyberVRML97 for Java, http://www.cybergarage.org/vrml/cv97/cv97java/, 2003.

[26] Schneider, P. and D. Eberly, Geometric tools for computer graphics, San Francisco: Morgan Kaufmann, 2003.

[27] Franklin, W.R., Efficient polyhedron intersection and union, Graphics Interface May 1982, Toronto, Canada, 1982.

[28] Shene, C-K., http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/euler.html, Lecture notes for computing with geometry, Michigan Technological University, 2003.

Appendix A: SQL-statements to insert test objects

The SQL-statements to insert the test objects from §7.1 are:

Tetrahedron

```
insert into testobjects values (1, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
13,0,1006, 16,0,1006, 19,0,1006, 22,0,1006),
mdsys.sdo_ordinate_array(
-1,-1,0, 1,2,0, 2,-2,0, 0,0,3, --coords
1,2,3, 1,3,4, 3,2,4, 2,1,4 –faces
)));
```

Cube

```
insert into testobjects values (2, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
25,0,1006, 29,0,1006, 33,0,1006, 37,0,1006, 41,0,1006, 45,0,1006),
mdsys.sdo_ordinate_array(
-8,-8,-1, -8,-5,-1, -5,-5,-1, -5,-8,-1, -8,-8,2, -8,-5,2, -5,-5,2, -5,-8,2,
1,2,3,4, 5,8,7,6, 1,4,8,5, 2,6,7,3, 1,5,6,2, 4,3,7,8 –faces
)));
```

Cube with dent

```
insert into testobjects values ( 3, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
49,0,1006, 53,0,1006, 57,0,1006, 61,0,1006, 65,0,1006,
69,0,1006, 73,0,1106, --1106 is inner ring where dent is located
77,0,1006, 81,0,1006, 85,0,1006, 89,0,1006, 93,0,1006),
mdsys.sdo_ordinate_array(
-5,10,0, -5,13,0, -2,13,0, -2,10,0, -5,10,3, -5,13,3, -2,13,3, -2,10,3,
-3,11,1, -3,12,1, -2,12,1, -2,11,1, -3,11,2, -3,12,2, -2,12,2, -2,11,2,
1,2,3,4, 5,8,7,6, 1,4,8,5, 2,6,7,3, 1,5,6,2, 4,3,7,8, 12,16,15,11,
9,12,11,10, 13,14,15,16, 9,10,14,13, 10,11,15,14, 9,13,16,12
)));
```

Hollow cube

```
insert into testobjects values (4, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
49,0,1006, 53,0,1006, 57,0,1006, 61,0,1006, 65,0,1006, 69,0,1006,
73,0,2006, 77,0,2006, 81,0,2006, 85,0,2006, 89,0,2006, 93,0,2006),
-- 2006 is the inner boundary that shells the hole
mdsys.sdo_ordinate_array(
5,5,0, 5,10,0, 10,10,0, 10,5,0, 5,5,5, 5,10,5, 10,10,5, 10,5,5,
6,6,1, 6,9,1, 9,9,1, 9,6,1, 6,6,4, 6,9,4, 9,9,4, 9,6,4,
1,2,3,4, 5,8,7,6, 1,4,8,5, 2,6,7,3, 1,5,6,2, 4,3,7,8,
9,12,11,10, 13,14,15,16, 9,13,16,12, 10,11,15,14, 9,10,14,13, 12,16,15,11
)));
```

Cube with through hole

```
insert into testobjects values (5, --id
mdsys.sdo_geometry(3002,null,null,
mdsys.sdo_elem_info_array(1,2,1,
49,0,1006, 53,0,1006, 57,0,1006, 61,0,1006,
65,0,1006, 69,0,1106, --one side of the through hole
73,0,1006, 77,0,1106, --other side of the through hole
81,0,1006, 85,0,1006, 89,0,1006, 93,0,1006),
mdsys.sdo_ordinate_array(
10,-4,0, 13,-4,0, 13,-4,3, 10,-4,3, 10,-1,0, 13,-1,0, 13,-1,3, 10,-1,3,
13,-3,1, 13,-2,1, 13,-2,2, 13,-3,2, 10,-2,1, 10,-3,1, 10,-3,2, 10,-2,2,
1,2,3,4, 4,3,7,8, 5,8,7,6, 1,5,6,2, 2,6,7,3, 9,12,11,10, 4,8,5,1,
13,16,15,14, 14,15,12,9, 15,16,11,12, 16,13,10,11, 13,14,9,10
)));
```

Appendix B: User manual

Using the 3D primitive in Oracle Spatial requires some installation. The polyhedra can be stored in the standard Oracle Spatial schema, so there is no installation needed here. The validation and 3D functions are written in Java. These functions need to be loaded in Oracle first. The syntax is:

```
loadjava -v -u [username]/[password] [Java class]
```

Note that the packages sdoapi and Jama also need to be loaded in the DBMS.

The next step is to create PL/SQL functions that can call the Java functions. These can all be gathered in a file with extension sql. To create a single function:

```
CREATE OR REPLACE FUNCTION function_name (var1 TYPE, var2 TYPE, etc.) RETURN
TYPE AS LANGUAGE JAVA
NAME 'geom3d.JavaClass.method(java_type, java_type, etc.) RETURN java_type';
/
```

The functions can now be used. A short overview of the functions from chapter 5:

Function	Description
validate_polyhedron	Determines if a polyhedron is valid
fix_orientation	Returns a correctly oriented polyhedron
area3d	Returns the area of a polyhedron or 3D polygon
volume	Returns the volume of a polyhedron
perimeter	Returns the perimeter of a polyhedron (sum of all edges)
average coordinate	Returns the average coordinate of a polyhedron
bounding_box	Returns the bounding box of a polyhedron
footprint	Returns the footprint of a polyhedron
rotation	Rotates a polyhedron
translation	Translates a polyhedron
scale	Scales a polyhedron
distance	Returns the distance between the average coordinates of two polyhedra
point_in_polyhedron	Determines if a point is inside a polyhedron
intersection	Determines if there is an intersection between two polyhedra
return_multipolygon	Converts a polyhedron to a multipolygon
return_polyhedron	Converts a multi-polygon to a polyhedron
distance_line	Returns the line between the average coordinates of two polyhedra

Table 7 Short overview of the 3D functions.

Now the complete specifications for each function:

Validate_polyhedron

Format	validate_polyhedron (geom MDSYS.SDO_GEOMETRY, tol NUMBER) RETURN VARCHAR2;
Parameters	geom: Polyhedron geometry tol: Tolerance value
Returns	'True' if the polyhedron is valid or an error message when it is invalid.
Example	SELECT validate_polyhedron(geom,0.05) FROM table;

Fix_orientation

Format	fix_orientation (geom MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry
Returns	Polyhedron with correctly oriented faces.
Example	UPDATE table SET geom. = fix_orientation(geom);

Area3d

Format	area3d (geom MDSYS.SDO_GEOMETRY) RETURN NUMBER;
Parameters	geom: Polyhedron geometry
Returns	The area as a number.
Example	SELECT area3d(geom) FROM table;

Volume

Format	volume (geom MDSYS.SDO_GEOMETRY) RETURN NUMBER;
Parameters	geom: Polyhedron geometry
Returns	The volume as a number.
Example	SELECT volume(geom) FROM table;

Perimeter

Format	perimeter (geom MDSYS.SDO_GEOMETRY) RETURN NUMBER;
Parameters	geom: Polyhedron geometry
Returns	The perimeter as a number.
Example	SELECT perimeter(geom) FROM table;

Average coordinate (average of all vertices)

Format	average coordinate (geom MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry
Returns	The average coordinate as text (function should be converted to return geometry).
Example	SELECT average coordinate(geom) FROM table;

Bounding_box

Format	bounding_box (geom MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry
Returns	Polyhedron geometry.
Example	SELECT bounding_box(geom) FROM table;

Footprint

Format	footprint (geom MDSYS.SDO_GEOMETRY, tol NUMBER) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry tol: Tolerance value
Returns	2D polygon geometry (2003).
Example	SELECT footprint(geom,0.05) FROM table;

Rotation

Format	rotation (geom MDSYS.SDO_GEOMETRY, rotaxis VARCHAR2, theta NUMBER) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry rotaxis: Rotation axis (‘x’, ‘y’ or ‘z’) theta: Rotation angle in radials
Returns	Polyhedron geometry.
Example	SELECT rotation(geom,’x’,0.7) FROM table;

Translation

Format	translation (geom MDSYS.SDO_GEOMETRY, px NUMBER, py NUMBER, pz NUMBER) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry px: Number to translate X-coordinate py: Number to translate Y-coordinate pz: Number to translate Z-coordinate
Returns	Polyhedron geometry.
Example	SELECT translation(geom,10,15,0) FROM table;

Scale

Format	scale (geom MDSYS.SDO_GEOMETRY, px NUMBER, py NUMBER, pz NUMBER) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry px: Number to scale X-coordinate py: Number to scale Y-coordinate pz: Number to scale Z-coordinate
Returns	Polyhedron geometry.
Example	SELECT scale(geom,200,100,50) FROM table;

Distance

Format	distance (geom1 MDSYS.SDO_GEOMETRY, geom2 MDSYS.SDO_GEOMETRY) RETURN NUMBER;
Parameters	geom1: Polyhedron geometry 1 geom2: Polyhedron geometry 2
Returns	Distance between average coordinates as number.
Example	SELECT distance(geom1,geom2) FROM table;

Point_in_polyhedron

Format	point_in_polyhedron (geom MDSYS.SDO_GEOMETRY, p MDSYS.SDO_GEOMETRY, tol NUMBER) RETURN NUMBER;
Parameters	geom: Polyhedron geometry p: Point geometry tol: Tolerance value
Returns	Number 1 when true or number 0 when false.
Example	SELECT point_in_polyhedron(geom,p,0.05) FROM table;

Intersection

Format	intersection (geom1 MDSYS.SDO_GEOMETRY, geom2 MDSYS.SDO_GEOMETRY, tol NUMBER) RETURN NUMBER;
Parameters	geom1: Polyhedron geometry 1 geom2: Polyhedron geometry 2 tol: Tolerance value
Returns	Number 1 when objects intersect and 0 when no intersection occurs.
Example	SELECT intersection(geom1,geom2,0.05) FROM table;

Return_multipolygon

Format	return_multipolygon (geom MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Polyhedron geometry
Returns	Multi-polygon geometry (3004 or 3007).
Example	SELECT return_multipolygon(geom) FROM table;

Return_polyhedron

Format	return_polyhedron (geom MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom: Multi-polygon geometry
Returns	Polyhedron geometry.
Example	SELECT return_polyhedron(geom) FROM table;

Distance_line

Format	distance_line (geom1 MDSYS.SDO_GEOMETRY geom2 MDSYS.SDO_GEOMETRY) RETURN MDSYS.SDO_GEOMETRY;
Parameters	geom1: Polyhedron geometry 1 geom2: Polyhedron geometry 2
Returns	The line segment geometry (3002) between the average coordinates.
Example	SELECT distance_line (geom1,geom2) FROM table;

Then there are some functions that work outside the DBMS:

VRML to SQL

Format	java geom3d.VRMLtoSQL [inputfile] [outputfile]
Parameters	inputfile: VRML file outputfile: SQL Loader file
Returns	SQL loader file with polyhedra.
Example	java geom3d.VRMLtoSQL 3Dscene.wrl 3DsceneSQL.ldr

SDO to VRML

Format	java geom3d.SdoToVRML [tablename] [outputfile]
Parameters	tablename: Table containing polyhedra outputfile: VRML file
Returns	VRML file with the polyhedra from the database.
Example	java geom3d.SdoToVRML pol_table 3Dscene.wrl

TOP10 to SQL

Format	java geom3d.TOP10toSQL [baseheight] [inputfile] [outputfile]
Parameters	baseheight: ground level height inputfile: ArcInfo export file including heights outputfile: SQL Loader file
Returns	SQL loader file with polyhedra.
Example	java geom3d.TOP10toSQL –0.5 objects_z.txtl 3DsceneSQL.ldr

For visualisation in Microstation an extra table needs to be created. This can be done by executing the mscat.sql file located in the Microstation directory:
Bentley\Program\MicroStation\database\oracle\

Appendix C: Java source code for volume computation

This appendix contains an example Java-file. The example here is the source code to compute the volume (filename: Volume.java). This source code is compiled first to a class-file (filename: Volume.class). The next step is to load this class-file into the DBMS:

```
loadjava -v -u –resolve [username]/[password] Volume.class
```

Note that dependent functions already should be loaded in the DBMS.

The next step is to create PL/SQL functions that can call the Java functions:

```
CREATE OR REPLACE FUNCTION volume (geom MDSYS.SDO_GEOMETRY) RETURN NUMBER AS
LANGUAGE JAVA
NAME 'geom3d.Volume.volume(oracle.sql.STRUCT) RETURN double';
/
```

The Java source code:

```
// geom3d.Volume
// Computes the volume of a polyhedron
// Calin Arens, 2003
// E-mail: calin_arens@hotmail.com

package geom3d;

import java.sql.*;
import oracle.jdbc.*;
import oracle.sql.*;


public class Volume
{

   public static double volume(oracle.sql.STRUCT s) throws
   java.sql.SQLException
   {

   // Convert MDSYS.SDO_GEOMETRY to Java
   SdoGeometry sdoGeom = new SdoGeometry(s);

   // Store volume
   double vol = 0.0;

   // Iterate over faces
   for(int i=0;i<sdoGeom.faceArray.length;i++)
   {

      // Get pointlist
      Point3D[] pointlist = sdoGeom.faceAsPointList(i);

      // Compute area of face
      double area = Area.area(pointlist);

      // Compute unit normal vector
      double[] n = sdoGeom.unitNormalFace(i);

      // Compute c = normal dot P0
      Point3D p0 = pointlist[0];
      double c = p0.dotProduct(n);

      // Contribution of face
      vol += (c*area);

   }

   // 1/3 of the vol is the volume
   double volume = ( (1.0/3.0)*vol );

   if (Double.isNaN(volume))
      return -1;
   else
      return volume;
}
```

Glossary

2D
2-Dimensional; 2D objects are flat, e.g. a polygon; objects in 2D space are spanned with 2 coordinates (usually length and width).

2-Manifold
A 2-manifold object is an object that bounds a single volume in space.

3D
3-dimensional; 3D objects have a volume, e.g. a polyhedron; objects in 3D space are spanned with 3 coordinates (usually length, width and height).

Base map
Map that contains basic topographic elements, like buildings and roads.

Binary functions
Functions that have two objects as input parameters.

Boundary
The separation between the inside and outside of an object.

CAD
Computer Aided Design; system used for designing e.g. buildings and infrastructure.

Average coordinate
Average of all object coordinates.

Computational complexity
The complexity of computations in a computer as in the number of computing steps in the worst case. More complex computations result in lower performance.

Conversion
Translation of data to another format to enable different programs to use these data.

Convex geometry
Geometry is convex if with every pair of points that belong to the geometry, it contains the whole straight line segment connecting the two points.

Coordinate
Unit to specify the position of a point.

Data model
The specification on how to store data.

Database
A collection of related data organised for efficient retrieval of information.

DBMS
DataBase Management System; collection of programs to maintain the data in databases.

Edge
Line between two nodes.

Face
Ordered collection of edges that form a closed curve.

Front-end
Program that allows end-users to work with the data in the database (back-end).

Genus
Number of holes that go completely through an object.

Geo-DBMS
DBMS that supports the management of geographical data.

Geographical data
Data that is dependent on a certain location on the earth.

Geometry
A related set of coordinates that form a shape.

GIS
Geographical Information System; system used for storing, maintaining, querying, analysing and visualising geographical data.

Hierarchical boundary representation
Data model to store geometry using ordered vertices, so that edges do not have to be stored and vertices only are stored once by using references to these vertices.

Inner ring
A hole inside a face or polygon.

Integrity
Characteristic of a DBMS to keep data valid at all times.

Intersection
Two objects intersect when they interact (e.g. touch or are completely within) with each other.

Java
Object oriented programming language that can be used to create functions (that can be stored and used in a DBMS, e.g. Oracle).

Line segment
Part of a line between two points.

MBB
Minimum Bounding Box; box around a 3D object that is parallel with the coordinate system axes.

MBR
Minimum Bounding Rectangle; rectangle around a 2D object that is parallel with the coordinate system axes.

Metadata
Information about data, e.g. precision and domain.

Modelling
Modelling is the process of describing an object or scene so that we can construct an image of it.

Multi-polygon
Collection of polygons that can be stored in one record in a database table.

NAP
Normaal Amsterdams Peil (Netherlands National Ordnance Datum); Dutch height system.

Node
Reference to a point (x,y,z).

Normal vector
Vector pointing to the outside of a face/polygon (in the context of a polyhedron).

Octree
3D spatial index that tiles up 3D space.

Orientable
Characteristic that makes it possible to discriminate the inside from the outside of an object.

Outer ring
The outer boundary of a face/polygon.

Plane
An infinite separation of space.

PL/SQL
Procedural Language/Structured Query Language; an Oracle extension to allow procedures in SQL.

Polygon
Flat geometry spanned up by connecting a set of vertices.

Polyhedron
Bounded subset of 3D coordinate space enclosed by a finite set of flat polygons such that every edge of a polygon is shared by exactly one other polygon. Each point can reach every other point through its interior.

Primitive
Building block to describe an object in a (database) system.

Quadtree
2D spatial index that tiles up space.

Query
A question or request to select a number of objects.

Query window
A question or request to select a number of objects within a certain area.

Projection
Lowering the number of dimensions by one, e.g. retrieving the footprint of 3D objects.

R-tree
2D or 3D spatial index that tiles up objects.

Record
Row in a table in a database to store data in.

Scalar
A single number.

Shell
Boundary that encloses a single volume in space. This can be the boundary of solid space or the boundary of 'empty' space (exterior) inside a solid object, e.g. a hollow object has two shells, one containing the solid object and one containing the empty hole inside the object.

Simple geometry
Closed geometry that is not self-intersecting and contains no inner rings.

Skeleton
A lower dimensional representation of an object, e.g. the centre line of a tunnel.

Spatial data
See: geographical data.

Spatial dataset
Set containing spatial data.

Spatial index
A spatial index is created to provide a way to improve spatial queries.

SQL
Structured Query Language; language to query data in a database.

Stored procedure/function
Procedure/function that is stored within the DBMS to enable better performance.

Tetrahedron
3-Simplex consisting of 4 triangles that form a closed object in 3D space.

Tolerance
Close to zero value to accommodate for numerical problems with floating point computations.

Topological relationship
Description of the relationship that exists between two objects (e.g. disjoint or completely within).

Topology
The spatial relationship between features (e.g. faces consist of edges consist of nodes). Also used for topological relationships.

Two-tier query
Query where first a set of candidates is quickly selected and then the complete query takes place on these candidates; this to improve performance.

Unary functions
Functions that operate on a single object.

Validation
Process of checking objects for correctness.

Vertex
Point that consists out of a number of coordinates (e.g. x,y,z).

VRML
Virtual Reality Modelling Language; language to model 3D objects for visualisation on the Internet.