New 3D data type and topological operations for Geo-DBMS

Chen Tet Khuan and Alias Abdul-Rahman
Department of Geoinformatics, Faculty of Geoinformation Science and Engineering, Universiti Teknologi Malaysia, Skudai, Malaysia

Sisi Zlatanova
Section GIS Technology, OTB Research Institute for Housing, Urban and Mobility Studies, Delft University of Technology, The Netherlands.

ABSTRACT: DBMS becomes very important for GIS as it used to handle large volume of spatial data and could ensure the stability of dataset handling. Next generation of GIS software would highly depend on DBMS in both geometrical modeling and analysis. One of the desired components in such future software or system is the geometric modeling that works with 3D spatial operations. This paper presents a portion of the problems, which are 3D topological operations for DBMS. These operations are very important for 3D spatial analysis. This paper discusses implementation of a 3D data type (Polyhedron) and eight 3D topological relationships between polyhedrons. The relationships are compliant with the 4-intersection model, and it could be extended to 9-intersection model if the exterior component is considered. The implementations are tested for PostgreSQL.

1 INTRODUCTION

GIS has become a sophisticated system for handling spatial and thematic information of real world spatial objects. DBMS are evolving to a core component of GIS architecture used to maintain geographic data (Breuning and Zlatanova, 2006, Zlatanova and Stoter, 2006). Therefore we believe much GIS functionality can and have to be implemented within DBMS environment. The DBMS then becomes an important medium for spatial data maintenance, spatial operations and integration purposes (e.g. data access from different front-ends). The two major aspects of DBMS functionality that define a Geo-DBMS are then spatial data and spatial operations and functions built on these data types.

Many researches have been working for providing support to 2D spatial data types and operations in mainstream DBMS. In the last year almost all DBMS vendors implemented the geometry models as defined by Open Geospatial Consortium (OGC). For example, Informix (2006) supports three basic spatial data types: point, line and polygon; Ingres (2006) supports one more data type: circle, beside the three basic types; Oracle Spatial (2006) not only able to handle points, lines, polygons and circles, but also gives further support to arc strings and compound polygons. These DBMSs manage spatial objects, together with their 3D coordinates, i.e. 2D objects (2D polygon + z-coordinate) are embedded in 3D space (Zlatanova 2006). Beside spatial data types, the spatial DBMS also implement operators and functions on the spatial data types, and thus spatial queries are possible at DBMS level. There are also functions, which can return list with coordinates, complete basic geometric transformations, maintain geometry validity, etc. Although coordinates in the data types can be 3D, the functions on spatial data types are still 2D, the z-values are hardly considered. Actually the lack of “true” 3D data type (i.e. 3D tetrahedron, 3D polyhedron) results in the unavailability of 3D operations in DBMS.

The ideas on OGC Abstract Specifications, which are identical with ISO 19107 'Spatial Schema', consider representing 3D geometry according to the well-known *Boundary Representation* (Foley *et al*. 1995). The geometry of spatial features is described by a combination of a geometry and a coordinate reference system. The Abstract Specifications suggest the use of other volumetric shapes as cones and spheres, and even freeform shapes, such as NURBS. The Implementation Specifications discuss only 2D objects, however.

The first attempt of 3D spatial data type and corresponding operations in a spatial DBMS has been investigated quite successfully by Arens 2003, Arens et al 2005. The basic idea is that a 3D polyhedron could be defined as a bounded subset of 3D space enclosed by a finite set of flat polygons, such that every edge of a polygon is shared by exactly one other polygon. Here, the polygons are in 3D space because they are represented by vertices, which can be 3D points in a spatial DBMS. This research has been studied and some concepts are taken over in the new 3D data type implemented in Oracle Spatial 11.

Yet, another attempt to define 3D object is reported by Penninga, 2005. The 3D object, i.e. tetrahedron is used for representing volumetric shapes. The tetrahedron is the simplest possible geometry in 3D domain. The conceptual design is intended for implementation for both geometry and topology model (Penninga et al, 2006).

Some other related research is reported by Pu (2005), who has created complex geometry types, i.e. freeform curves and surfaces. Although freeform shapes can be simulated by tiny line segments/triangles/polygons, it is quite unrealistic and inefficient to store all these line segments/triangles/polygons into a DBMS especially when shapes are rather huge or complex. Therefore, Pu and Zlatanova (2005) stored freeform shapes directly in DBMS. Corresponding spatial operators and functions that manage these shapes are also tested.

Quite some research has been competed on exchange of 3D models. In this respect it is worth mentioning 3D CityGML (Kolbe *et al.* 2006). It is an open data model and XML-based format for the storage and exchange of virtual 3D city models. It is an application schema for the Geography Markup Language 3 (GML3). CityGML defines the concept of Levels of Detail (LOD) for real-world volumetric objects and suggests 5 LOD, i.e. LOD0 denotes 2.5D Digital Terrain Model, LOD1 denotes the blocks model comprising prismatic buildings with flat roofs, LOD2 has differentiated roof structures and thematically differentiated surfaces, LOD3 denotes architectural models with detailed wall and roof structures, balconies, bays and projections. High-resolution textures can be mapped onto these structures. In addition, detailed vegetation and transportation objects are components of a LOD3 model. LOD4 completes a LOD3 model by adding interior structures for 3D objects. CityGML is basically a sematic model and uses the OGC geometry data types, but more complex geometries can be considered as well..

With these efforts, the possibility of developing 3D data types in DBMS is obvious. Currently missing are the topological operations for 3D data type. Current DBMS only provides 2D operations, which the z-value is not considered. 3D topological operations are also rather limited in typical GIS software. In this paper, we focus on simple but complete strategy in creating new datatype, i.e. polyhedron. Furthermore, the 3D datatype is used to test the developed topological operations that based on Spatial 9i model for 3D GIS. The algorithm fully covers the third dimension and able to be applied in 3D situation. The 3D topological operations are tested for PostgreSQL and can theoretically become a part of the PostGIS.

The paper is organized in the following order: first, short discussion for the 3D objects construction in three-dimension, i.e. polyhedron (see section 2). Then, the method to create new 3D datatype described in Section 3. The rule for developing 3D topological operations for DBMS is discussed in Section 4. The experiment and discussions are presented in Section 5 and the research is concluded with some future work remarks in Section 6.

2 CHARACTERISTIC OF POLYHEDRON

Polyhedron is a 3D equivalent of a set of polygon that bounds a solid object. It is made up by connecting all faces, sharing a common edge between two adjacent polygons. The most

important constraint is all the polygons that make up the polyhedron have to be flat. This means that all points used to construct a polygon must be in the same plane. Figure 1 denotes a sample of a planar and non-planar polygon. The characteristics of a valid polyhedron should have the following rules (Aguilera & Ayala (1997), Aguilera (1998)):

- Flatness – all polygons that bound a single volume of polyhedron must be flat. This means all vertices involve in constructing a polygon should be in the same plane. The flatness of a polygon can be verified by plane equation as follow:

  $$Ax + By + Cz + D = 0 \quad \text{Eq. (1)}$$

- Polyhedron must be single volume object – a set of polygons that make up a polyhedron should be bounded as a single volume. In order to create a single volume of polyhedron, some rules must be followed:
  - Each edge (derived out of 2 vertices) should be shared by only 2 polygons. This rule will result in a simple polyhedron, i.e. outer ring will not touch the boundary of the polyhedron. On the other hand, if an edge is shared by more than 2 polygons, the polyhedron may consist at least 2 volumes.

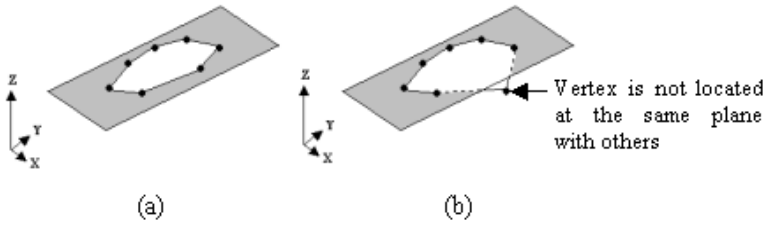


Figure 1: (a) Planar polygon, and (b) non-planar polygon

- Simplicity characteristic – as discussed by Arens (2003). However, this condition could be simplified by enforcing the construction of a polygon as follow:
  - Each edge has exactly 2 vertices only.
  - The starting and ending points of a polygon is same, and will only be stored once. E.g. a polygon consists 4 points (a, b, c, d), thus the polygon will be stored as (a, b, c, d, a), instead of (a, b, c, d, e), although a = e. Any point(s) with same location will be stored only once.
  - Polygon must have an area.
  - Lines from a polygon must not self-intersecting.
  - Singularity of polyhedron is not allowed, i.e. lower dimension object must not exist in the interior of higher dimension. E.g. point will not exist in the interior of line or polygon or polyhedron, line will not exist in the interior of polygon or polyhedron. However, lower dimension object may exist at the border of higher dimension object. This rule may directly avoid polygon intersects with other polygon(s) (see Figure 2). Any polygons that intersect with other polygon(s) will not be stored as a part of polyhedron.

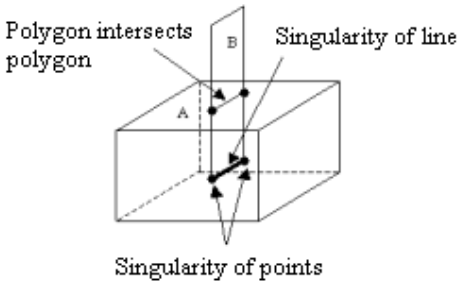


Figure 2: Polygon intersection causes the singularity of points and line

3 3D TOPOLOGICAL OPERATIONS

The topological operations presented here are based on the 4-intersection model and extends to 3D. Typically, the results given by this operation are in Boolean type, i.e. either TRUE or FALSE. The related operations include Overlap, Meet, Disjoint, Inside, Covers, CoveredBy, Contain, and Equal (see Figure 4).

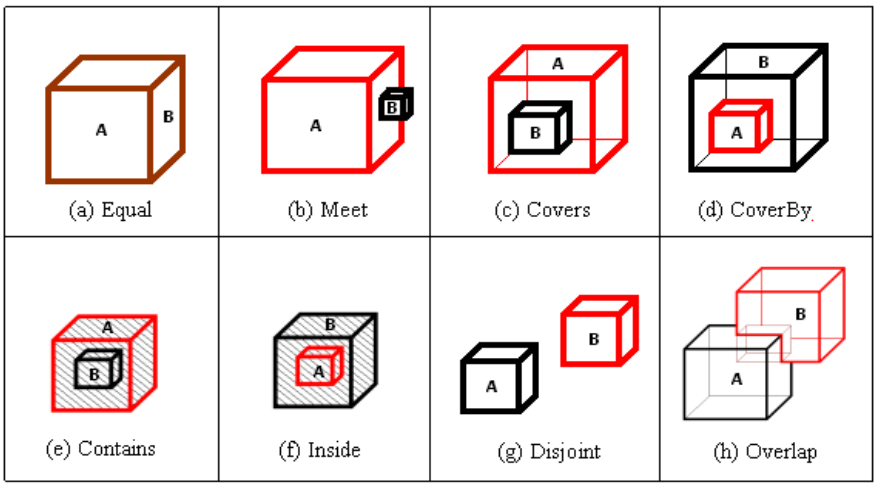


Figure 4: Body and body relation (after Zlatanova, 2000)

For topological operation in geometrical model, coordinate triplet of vertex will be discussed. Similar to computational-geometry operation from previous section, the binary operation is divided into base and target object. However, the vertices from base object and polygons from target object will be discussed (see Figure 5).

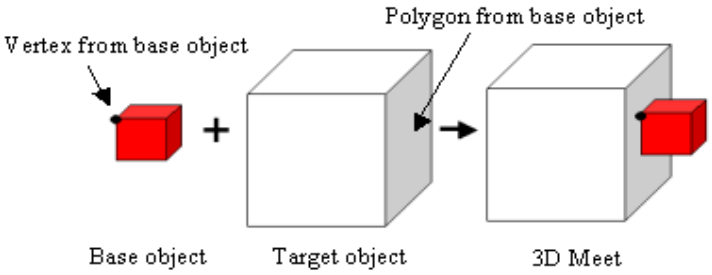
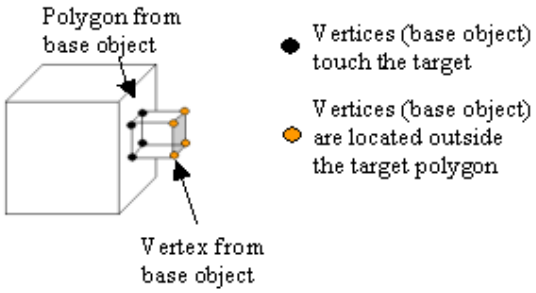


Figure 5: Base and target object for 3D operation

This topological operation involves vertices (from base object) and polygon (from target object). Therefore, the relation between these 2 objects will be examined. The location of base vertices relative to target polygon will be either outside, touch, or inside. The implementation was discussed in Chen and Abdul-Rahman (2006). These relations will be used to determine how these 2 polyhedrons intersect each other as shown in Figure 4. For example (see Figure 6), vertices from base object are either touch the target polyhedron or located outside from target object.



3D topological operations	Inside	Outside	Touch
Meet	NO	YES	YES

Figure 6: Vertices (base) are located and touch the target polygon

The following Table 1 denotes the complete relation between base and target object.

3D topological operations	Inside	Outside	Touch
Equal	X	X	✓
Meet	X	✓	✓
Covers	✓	X	✓
CoveredBy	✓	X	✓
Contains	✓	X	X
Inside	✓	X	X
Disjoint	X	✓	X
Overlap	✓	✓	✓

Table 1: Conditions for topological operations

The relationship of Covers and CoveredBy are different due to the role of base and target objects between these two relationships are different. For Covers, the base object covers the entire target object, whereas for CoveredBy, the target object covers the entire base object. The similar approach implemented between Contains and Inside.

4 IMPLEMENTATION IN DBMS

Most of the commercial DBMS enable a user to create a new user-defined data type and functions. This user-defined datatype and functions can be written in C, C++ or Java. Data types can be started also using high-level language PL/SQL but usually these implementations have a bad performance. In this research, we have used C. In general, a user-defined type is defined as a class and must always have *input* and *output* functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If users want to do anything more with the type than merely store it, they must provide additional functions to implement whatever operations they'd like to have for the type.

The following three sections will illustrate how a new data type and a new function can be designed in C, compiled and used in PostgreSQL

4.1 *Polyhedron data type*

Suppose user wants to define a type complex that represents complex numbers. A natural way to represent a complex number in memory would be the following C structure:

```
typedef struct {
   char buf[200];
}POLYHEDRON;
```

As the external string representation of the type, a string of the form (POLYHEDRON) is chosen. The input and output functions are usually not hard to write especially the output function. But when defining the external string representation of the type, remember that users must eventually write a complete and robust parser for that representation as their input function. For instance:

```
PG_FUNCTION_INFO_V1(Polyhedron_in);

Datum
Polyhedron_in(PG_FUNCTION_ARGS)
{
    //== POLYHEDRON input class ==//
}
```

The output function can simply be:

```
PG_FUNCTION_INFO_V1(Polyhedron_out);

Datum
Polyhedron_out(PG_FUNCTION_ARGS)
{
    //== POLYHEDRON output class ==//
 }
```

To define the complex data type, user needs to create the user-defined I/O functions within PostgreSQL environment before creating the type:

```
CREATE FUNCTION Polyhedron_in(cstring)
    RETURNS POLYHEDRON
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION Polyhedron_out(POLYHEDRON)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

Notice that the declarations of the input and output functions must reference the not-yet-defined type. Although this is allowed, but it will draw warning messages that could be be ignored. The input function must appear first. Finally, the data type will be declared:

```
CREATE TYPE POLYHEDRON (
   internallength = 100,
   input = Polyhedron_in,
   output = Polyhedron_out,
   alignment = double
);
```

4.2 *User-defined function/operation*

There are two different calling conventions are currently used for C functions (PostgreSQL, 2006). To create new user-defined functions/operations, the calling convention must be “version 1”, due to ‘version 0” is not applicable for creating user-defined functions. The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The PG_FUNCTION_INFO_V1( ) macro is used in calling for the function. Within the function, each actual argument is fetched using a PG_GETARG_xxx() macro that corresponds to the argument’s data type, and the result is returned using a PG_RETURN_xxx() macro for the return type. PG_GETARG_xxx() takes as its argument the number of the function argument to fetch, where the count starts at 0. PG_RETURN_xxx() takes as its argument the actual value to return. The C declaration of a version-1 function is:

```
PG_FUNCTION_INFO_V1(OVERLAP3D);

Datum OVERLAP3D(PG_FUNCTION_ARGS)
{
    int32 arg = PG_GETARG_xxx(0);
    //== 3D OVERLAP function class ==//
    PG_RETURN_xxx();
}
```

In addition, the macro call must appear in the same source file. (Conventionally, it's written just before the function itself.) This macro call is not needed for internal-language functions, since PostgreSQL assumes that all internal functions use the version-1 convention. It is, however, required for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a PG_GETARG_xxx() macro that corresponds to the argument's data type, and the result is returned using a PG_RETURN_xxx() macro for the return type. PG_GETARG_xxx() takes as its argument the number of the function argument to fetch, where the count starts at 0. PG_RETURN_xxx() takes as its argument the actual value to return. The `CREATE FUNCTION` commands are the same as for the version-0 equivalents.

4.3 *Compiling and Linking Dynamically-Loaded Functions*

Before the implementation of PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precised, a *shared library* needs to be created. Creating shared libraries is generally analogous to linking executables. First, the source files are compiled into object files, then the object files are linked together. The object files need to be created as *position-independent code* (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. The dynamic loading feature involves 2 processes:

- *Dynamic loading* is what PostgreSQL does to an object file. The object file is copied into the running PostgreSQL server and the functions and variables within the file are made available to the functions within the PostgreSQL process. PostgreSQL does this using the dynamic loading mechanism provided by the operating system. The syntax that runs in Linux platform is to produce an object file called *Polyhedron.o* that can then be dynamically loaded into PostgreSQL.

  ```
  gcc -fpic -c Polyhedron.c
  ```

- *Loading and link editing* is what user does to an object file in order to produce another kind of object file (e.g., an executable program or a shared library). User performs this using the link-editing program. This share library will be registered within PostgreSQL environment.

```
gcc -shared -o Polyhedron.so Polyhedron.o
```

The methodology of creating user-defined datatype and function/operation are presented in flowchart as follows: (see Figure 3)

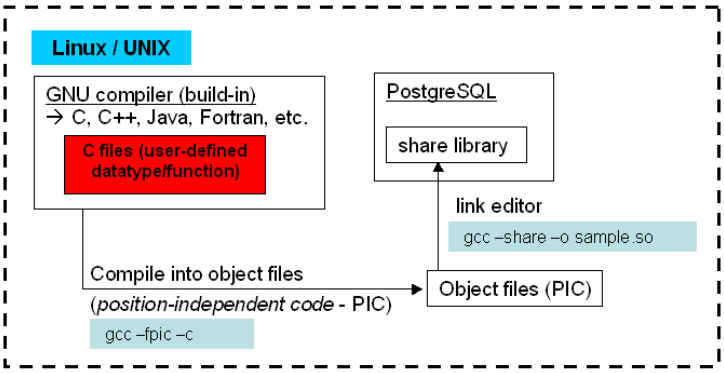


Figure 3: Workflow of creating user-defined datatype/function in PostgreSQL

5 EXPERIMENT AND DISCUSSIONS

As mentioned above, the 3D topological operations were tested within PostgreSQL environment. A 3D data type, i.e. POLYHEDRON was created and several 3D functions were implemented. The methodology for the complete implementation is given in Figure 7.

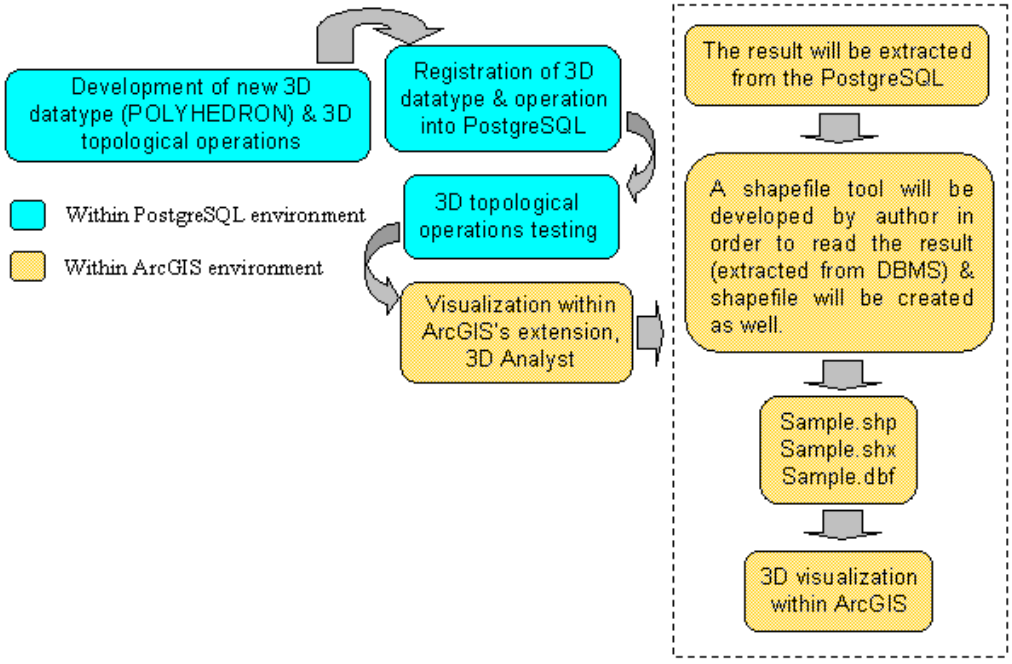


Figure 7: The implementation of new 3D datatype & operations for DBMS

The following SQL line denotes a sample of a polyhedron will be defined in PostgreSQL:

```
SELECT * FROM BODYTABLE WHERE PID = 1;
```

```
1,POLYHEDRON(PolygonInfo(6,24),SumVertexList(8),SumPolygon
List(4,4,4,4,4,4),VertexList(100.0,100.0,100.0,400.0,100.0,
100.0,400.0,400.0,100.0,100.0,400.0,100.0,100.0,100.0,400.0
,400.0,100.0,400.0,400.0,400.0,400.0,100.0,400.0,400.0),Pol
ygonList(1,2,6,5,2,3,7,6,3,4,8,7,4,1,5,8,5,6,7,8,1,4,3,2))
```

1). `PolygonInfo(6,24)` denotes 6 polygons and 24 IDs in `PolygonList`,
2). `SumVertexList(8)` denotes the total vertices,
3). `SumPolygonList(4,4,4,4,4,4)` denotes total vertices for each of polygon (total polygon is 6, referred to (1)),
4). `VertexList()` denotes the list of coordinate-values for all vertices (with no redundant), and
5). `PolygonList()` denotes the information about each polygon from sets of ID.

The graphical representation of the sample polyhedron stated above is given as follows (see Figure 8):

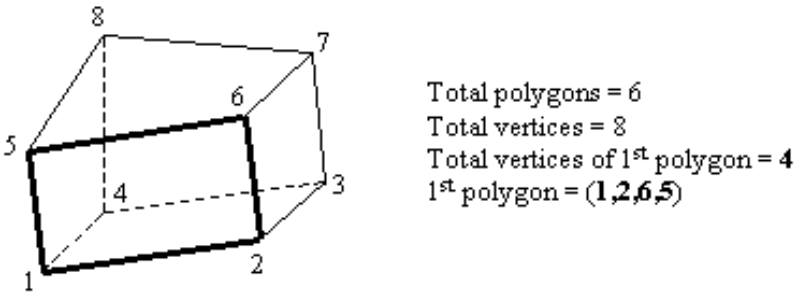


Figure 8: Sample structure of a polyhedron

The following examples are implemented within PostgreSQL environment. 2 polyhedrons are inserted into a table, `test`, as follows:

```
INSERT INTO test(PID,POLYHEDRON) VALUES
(1,'POLYHEDRON(PolygonInfo(6,24),SumVertexList(8),SumPolygonList(4,4,4
,4,4,4),VertexList(100.0,100.0,100.0,400.0,100.0,100.0,400.0,400.0,100
.0,100.0,400.0,100.0,100.0,100.0,400.0,400.0,100.0,400.0,400.0,400.0,4
00.0,100.0,400.0,400.0),PolygonList(1,2,6,5,2,3,7,6,3,4,8,7,4,1,5,8,5,
6,7,8,1,4,3,2))');

INSERT INTO test(PID,POLYHEDRON) VALUES
(2,'POLYHEDRON(PolygonInfo(6,24),SumVertexList(8),SumPolygonList(4,4,4
,4,4,4),VertexList(300.0,300.0,300.0,600.0,300.0,300.0,600.0,600.0,300
.0,300.0,600.0,300.0,300.0,300.0,600.0,600.0,300.0,600.0,600.0,600.0,6
00.0,300.0,600.0,600.0),PolygonList(1,2,6,5,2,3,7,6,3,4,8,7,4,1,5,8,5,
6,7,8,1,4,3,2))');
```

The following SQL statement runs the 3D Overlap (see Figure 9):

```
SELECT GMOVERLAP3D(a.POLYHEDRON,b.POLYHEDRON) AS
GM_OVERLAP3D FROM test a, test b where a.PID=1
and b.PID=2;
```

The result:

```
GM_OVERLAP3D
----------
(TRUE)
```

For visualization purposes, ArcGIS's extension, 3D Analyst is used to verify the result. Although PostGIS provides a function pgsql2shp for export to shape files, it cannot be used since it works only with the natively supported data types of PostGIS. Therefore we have implemented our own function. The integration between PostgreSQL and ArcGIS is beyond the scope of this paper. ArcGIS is used here only to illustrate the implementation of the new data type and the corresponding functions (Figure 9 and 10).

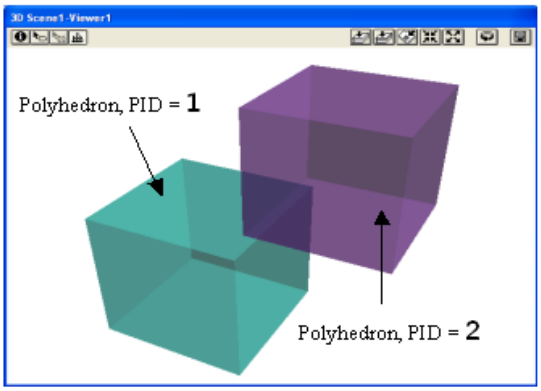


Figure 9: 3D Overlap

The same implementation was done for the rest of 3D topological operations, i.e. 3D Meet, 3D Disjoint, 3D Inside/Contains, 3D Covers/CoveredBy, and 3D Equal (see Figure 10a to 10e).

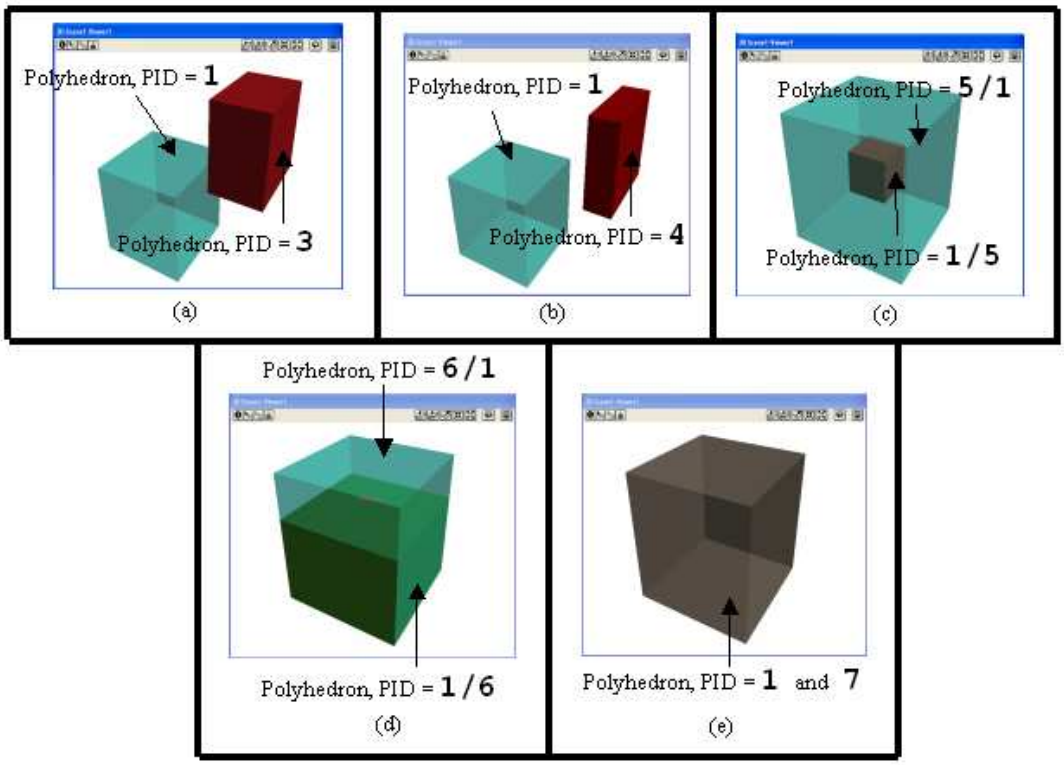


Figure 10: (a) 3D Meet, (b) 3D Disjoint, (c) 3D Inside/Contains, (d) 3D Covers/CoveredBy, and (e) 3D Equal

6 CONCLUDING REMARKS

We have implemented an approach for 3D topological operations of geometrical model in Geo-DBMS. The results have shown that implementation of a 3D data type and functions allowing 3D GIS analysis are possible. Our concept was tested within PostgreSQL computing environment and has provided a promising outcome with respect to the developed algorithms.

The 3D topological operations for DBMS could be implemented using different approaches such as using other programming language, i.e. PL/PGSQL, PL/TCL, PL/Perl, and SQL within PostgreSQL environment. However, since the PostgreSQL was developed mostly using C language, an implementation using procedural languages could result in less efficiency and low performances.

New data types can also be implemented in other DBMS, e.g. Oracle Spatial, similar to the work by Pu (2005) that had been done using free-from objects. The reason of using PostgreSQL in this research is that PostgreSQL follows the specifications of Open GIS Consortium (OGC, 2006). The most important for user is the commercial issues, which it is an open source technology and suitable for educational purposes. Some of the comparisons among DBMSs could be found at Konrad *et al.* (2006).

The test data set consists currently of very simple objects as it can be seen from the example, but some more experiments with real data sets are planned and will be completed soon. It is also interesting to compare our implementation with the shortly coming implementation of Oracle Spatial 11.

Currently, 3D topological operation between only polyhedron and polyhedron where implemented. Next important step will be the implementation of 3D spatial operations between polyhedron and polygon, polyhedron and line and polyhedron and point. As polygon, line and point data types the existing data types of PostGIS can be used.

Future research will concentrate on computational-geometry operations such as 3D intersection, 3D difference, and etc. A very important issue still need to be addressed is visualization of the result of 3D queries. Appropriate graphical visualization is especially important for 3D in order to get a better perception of the result of the query. Some topics to be considered are: 1) direct access to the new data type from GIS, avoiding first export to a shape file, 2) direct connection with CAD/CAM software, e.g. Microstation and Autodesk Map 3D to be able not only to visualize but also edit, 3) user-defined environment, where user develops display tool that manage to retrieve and visualize data from DBMS, or 4) access via Internet, using e.g. WFS.

We believe this research effort towards realizing a fully 3D spatial analysis tools within Geo DBMS environment would be beneficial to 3D GIS research community. This is because major GIS task involves DBMS (except 3D visualization), i.e. dataset handling, spatial operations, etc. It is our aim to move further in addressing this issue of spatial data modeling and geometrical modeling for 3D GIS.

REFERENCES

Aguilera, A, and Ayala, D. 1997. Orthogonal polyhedra as geometric bounds in constructive solid geometry. In C. Hoffman, and W. Bronsvort, (ed.), *Fourth ACM Siggraph Symposium on Solid Modeling and Applications*, Vol.4: 56-67.

Aguilera, A 1998, Orthogonal polyhedra: study and application. *Ph.D. Thesis*, LSI-Universitat Politècnica de Catalunya.

Arens, C. A. 2003. Modelling 3D spatial objects in a geo-DBMS using a 3D primitives. *Msc thesis*, TU Delft, The Netherlands. 76 p.

Arens, C., Stoter, J.E., and van Oosterom, P.J.M. 2005. Modelling 3D spatial objects in a geo-DBMS using a 3D primitive. In *Computers & Geosciences*, volume 31, 2. pp. 165-177

Breuning, M. and Zlatanova, S. 2005. 3D Geo-DBMS, Chapter 4. In S. Zlatanova, & D. Prosperi (ed.), *Large-scale 3D data integration: challenges and opportunities*, Taylor & Francis, A CRC press book. pp. 88-113

Chen T. K., and A. Abdul-Rahman, A. 2006. 0-D feature in 3D planar polygon testing for 3D spatial analysis. In A. Abdul-Rahman, S. Zlatanova, and V. Coors (ed.), *Lecture Note on geoinformation and cartography – innovations in 3D Geo information systems*, Springer-Verlag. pp. 169-183.

CityGML, http://www.citygml.org/

Foley, J., van Dam, A., Feiner, S., and Hughes, J. 1995. *Computer graphics: principles and practice*. Addison Wesley, 2nd Ed.

Geodata Infrastructure North-Rhine Westphalia (GDI NRW). http://www.gdi-nrw.org/index.php?id=34&lang=eng

Informix (2000). http://www.ibm.com/software/data/informix/

Ingres (2006). http://www.ingres.com/

Kolbe, T., Groeger, G. and Czerwinski, A. 2006. City Geography Markup Language (CityGML). In *OGC, OpenGIS Consortium, Discussion Papers*, Version 0.3.0, 120 p.

Konrad, B., Maciej, C., Micha, J., Dawid, J., Piotr, M., Marcin, M.,, Miko, O., Wiktor, S. P., Sylwester R., Piotr, S., Tomasz, T., Dominik, T., Jacek, W. 2006. Comparison of Oracle, MySQL and Postgres DBMS. http://dcdbappl1.cern.ch:8080/dcdb/archive/ttraczyk/db_compare/db_compare.html

OpenGIS Consortium (OGC). 2001. The OpenGIS abstract specification, Topic 1: Feature geometry (ISO 19107 Spatial Schema), Version 5, edited by J.H. Herring, OpenGIS Project Document Number 01-101, Wayland, Mass., USA.

Oracle Spatial 10g. http://www.oracle.com/database/index.html

Penninga, F. 2005. 3D topographic data modelling: why rigidity is preferable to pragmatism. In *Spatial Information Theory, Cosit'05, Vol. 3693 of Lecture Notes on Computer Science*, Springer. pp. 409-425.

Penninga, F., van Oosterom, P.J.M, and Kazar, B. M. 2006. A TEN-based DBMS approach for 3D topographic data modelling. In *Spatial Data Handling 2006*.

Pu, S. 2005. Managing freeform curves and surfaces in a spatial DBMS. *Msc Thesis*, TU Delft. 77 p.

Pu, S. & Zlatanova, S. 2006. Integration of GIS and CAD at DBMS level. In E. Fendel, M. Rumor (ed), *Proceedings of UDMS'06 Aalborg, Denmark, TU Delft*. pp. 9.61-9.71.

PostGIS. http://www.postgis.org/

PostgreSQL. http://www.postgresql.org/
Special Interest Group 3D (SIG 3D), http://www.ikg.uni-bonn.de/sig3d/
Zlatanova, S. 2006. 3D geometries in DBMS. In A. Abdul-Rahman, S. Zlatanova & V. Coors (ed.), *Innovations in 3D geoinformation systems,* Springer, Berlin. pp. 1-14.
Zlatanova, S. & Stoter, J. 2006. The role of DBMS in the new generation GIS architecture, Chapter 8 in S., Rana & J. Sharma (ed.), *Frontiers of Geographic Information Technology*. Springer. pp. 155-180.