Managing Freeform Curves and Surfaces in a Spatial DBMS

Master thesis by Shi Pu

July, 2005

Summary

The GIS (geographic information system), which implements an integration of semantic, geometric data and spatial relationships, seems to be the most appropriate system ensuring a large scope of analysis and thus serving many applications and daily activities. Therefore many CAD applications have been trying to provide some GIS functionality for years. On the other hand, current GIS applications support only a limited number of geometry types (point, linestring, polygon, etc.), and they are trying to support more complex geometry types, such as freeform curves and surfaces, which are already supported in CAD applications. Freeform curves and surfaces are everywhere in the real world and they are also essential to GIS fields. A logical consequence of attempts from both sides has achieved agreements (OGC specifications) on the manner for representing, accessing and disseminating spatial information in a central DBMS.

After several years' development, nowadays there are many mainstream DBMSs (database management system) such as Oracle, Postgres, Informix and Ingres, which are able to manage simple geometries. Examples of the supported geometries are: point, linestring, polygon, and collection geometry types which are made up of the first three basic geometries. However, these supported geometry types in current DBMSs are still limited. Complex geometry types like freeform curves and surface are not yet supported. Freeform curves and surfaces are essential shapes in CAD applications, therefore DBMSs cannot work fully collaborative with CAD applications without supporting freeform shapes. 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. A more direct solution is to create freeform spatial data types based on mathematical representations, and store the required parameters with the attributes of the freeform spatial data types. Operators and functions on freeform spatial data types are also necessary to manage (access, transform, validate, etc.) freeform curves and surfaces at DBMS level.

This thesis presents my MSc research which aims at managing freeform shapes such as Bézier curve, B-spline curve, and NURBS curve/surface, in a well-known spatial DBMS: Oracle Spatial. This is done by implementing user-defined data types for these freeform geometries. Each instance of a user-defined data types represents a piece of freeform curve/surface, and parameters of this freeform curve/surface are stored in the attributes of the user-defined data type. The next step is to create 3D functions on these types to make Oracle Spatial able to manage the freeform data types. Finally, data exchange of freeform shapes between CAD applications (MicroStation and AutoCAD) and Oracle Spatial is also addressed in this research, which means that freeform geometry models can be transferred from CAD applications to Oracle Spatial, and the other way around.

Acknowledgements

This thesis is the result of my MSc research project for the Media Knowledge Engineering program of the Faculty of Electrical Engineering, Mathematics and Computer Science at Delft University of Technology. The research was done at the GIS Technology Section of OTB Research Institute for Housing, Urban and Mobility studies at Delft University of Technology between January 2005 and June 2005. During the 6 months' period, I have been very lucky to receive supervision from two supervisors: Sisi Zlatanova from the GIS Technology Section of the OTB Research Institute and Wim Bronsvoort from the Computer Graphics and CAD/CAM Group of the Faculty of Electrical Engineering, Mathematics and Computer Science.

I have been working happily, not only because this was an interesting research topic, but also because of the great help from a lot of people. I would like to thank them all, and here I mention the people who have contributed most.

First of all, I would like to thank Sisi Zlatanova and Wim Bronsvoort, for their strict and patient supervision of my research, as well as their concern about my life in the Netherlands. I have to mention Dan Tartaglia from Bentley's technical support team, who has given me accurate and fast help for the visualization part of this research. Thanks to Wilko Quak for joining the research discussion and giving me so many nice suggestions. Special thanks to my colleagues and room mates: Evelien van Rij and Demetrio Munoz-Gielen. They have been so warm-hearted in helping me to get involved into the Dutch working environment: coffee, lunch, weather, etc. Finally I would like to thank Lingxiao Zhao. As my predecessor at Delft University of Technology, he has been giving me all kinds of suggestions on study, life, and future, which greatly encouraged me to focus on my target of the future.

This MSc thesis research topic is the first attempt on implementing data types for complex geometries in a spatial DBMS. Hopefully it can be a good reference for relevant research in the future.

Delft, July 2005
Shi Pu

Table of Contents

Chapter 1

Introduction

The border between CAD (Computer-aided design) and GIS (Geographic information system) is fading. Software for CAD was primarily designed to deal with large-scale models (but relatively small in size), without maintenance of attributes and geographic coordinates systems. In contrast, GIS was able to manage small-scale models (but very large in size), maintain attributes and a variety of different geographic coordinate systems [1]. However, the GIS, which implements an integration of semantic, geometric data and spatial relationships, seems to be the most appropriate system ensuring a large scope of analysis and thus serving many applications and daily activities, according to [14]. Therefore many CAD applications, such as AutoCAD and MicroStation, have been trying to provide some GIS functionality for years. On the other hand, current GIS applications support only a limited number of geometry types (point, linestring, polygon, etc.), and they are trying to support more complex geometry types, such as freeform curves and surfaces, which are already supported in CAD applications. Freeform curves and surfaces are everywhere in the real world and they are also essential to GIS fields. For example, a large number of roads are freeform curves, and more and more modern buildings' surfaces are freeform surfaces. A logical consequence of attempts from both sides has achieved agreements (OGC specifications) on the manner for representing, accessing and disseminating spatial information in a central DBMS (database management system): a system in which spatial (geometry) data and attribute (semantic) data are maintained in one integrated environment, as shown in Figure 1.1. Thus several DBMS vendors (Oracle, Postgres, Ingres, etc.) have developed spatial DBMSs based on traditional DBMSs. A spatial DBMS differs from a traditional DBMS in that a spatial DBMS is able to maintain spatial data types (point, linestring, polygons, etc.) beside the traditional data types (number, varchar, date, etc.), and there are also a number of functions on these spatial data types, which can do operations like return geometric information, do basic geometric transformations, maintain geometry validity, etc.

A lot of research has been done since the increasing requirement for representing, accessing and disseminating spatial data from CAD and GIS fields [13], and more and more DBMSs give support to spatial data types and operations. For example, **Informix** supports three basic spatial data types: point, line and polygon; **Ingres** supports one more type: circle, beside the three basic types; **Oracle Spatial** not only has points, lines, polygons and circles, but gives further support to arc strings and compound polygons. Beside spatial data types, all these spatial DBMS also implemented operators and functions on the spatial data types, so that geometric queries and operations are possible at DBMS level.

However, the currently supported spatial data types in DBMS are rather limited, mostly to 2D space. Although the points in most spatial DBMS can be 3D, the functions on spatial data types are actually still based on 2D, which means the z values are not considered. The first attempt of 3D spatial data type and operations in a spatial DBMS has been done successfully at the Section GIS Technology, Technical University of Delft [2], and 3D polyhedrons can be

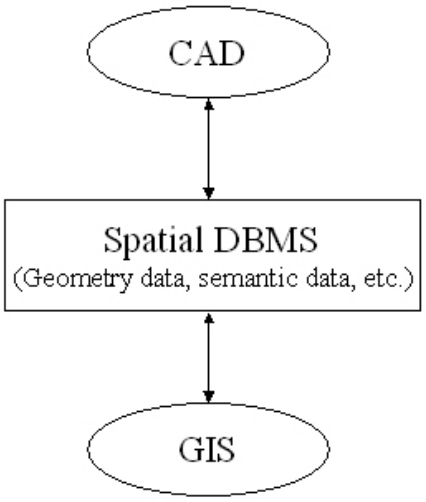

Figure 1.1: Spatial DBMS as a central system

stored and manipulated in Oracle Spatial after this attempt. The basic idea of [2] is that a 3D polyhedron can 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. The polygons are in 3D space because they are represented by vertices, which can be 3D points in a spatial DBMS. Based on this idea, a true 3D polyhedron in the spatial DBMS, including functions such as validation, volume, 3D transformation, etc., has been implemented.

After this attempt of 3D spatial data types and operations in a spatial DBMS, more complex geometry types such as freeform curves and surfaces, can be researched to implement. Many shapes in real world are freeform, i.e. not only contain points, linestrings and polygons, but also curves and curved surfaces. Examples of these shapes include roads, territory surfaces, building surfaces, etc. Furthermore, freeform shapes have been widely supported in mainstream CAD applications (AutoCAD, MicroStation). As the integration system between CAD and GIS, a spatial DBMS cannot work fully collaboratively with CAD/GIS applications before supporting freeform data types. 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. A more efficient solution is to implement freeform spatial data types which are able to store freeform shapes directly in DBMS, and implement corresponding spatial operators and functions to manage these shapes. The objectives of this thesis can be expressed in one main question:

How can freeform curves and surfaces be managed in a spatial DBMS?

This question can be subdivided into four smaller questions:

1. Which kinds of freeform curves and surfaces should be chosen to be supported in a spatial DBMS? Which parameters do they need to define a freeform curve/surface?

2. How can freeform curves and surfaces be stored in the spatial DBMS?

3. How can freeform curves and surface be manipulated in the spatial DBMS?

4. How can freeform curves and surfaces be transferred from the spatial DBMS to CAD/GIS applications, and the other way around?.

In order to answer these questions, theories and technologies from several fields (geometric modeling, CAD, GIS, DBMS) were studied. Three popular mathematical representations for freeform shapes: Bézier, B-spline and NURBS, are used as the freeform data types in the spatial DBMS. As one of the most powerful spatial DBMS on market, Oracle Spatial 10i is chosen as the spatial DBMS to build the freeform data types on. Another reason that Oracle Spatial is chosen is the strong support for user-defined data types, which means that creation of new data types is possible. A well-known CAD application, MicroStation v8, is used to visualize freeform spatial data from Oracle Spatial and store freeform spatial data into Oracle Spatial, because it is easy to get support from MicroStation's developers. The freeform spatial data transfer between MicroStation and Oracle Spatial can be done by writing JMDL code, which is an extension of Java language by MicroStation, and using certain functions to access Oracle and visualize freeform geometries [28].

Chapter 2 gives the background relevant to this research, including the mathematical background for the three freeform shape models, and an introduction to spatial DBMSs. Chapter 3 designs the conceptual models of freeform data types and spatial functions by analyzing several important aspects. Based on the conceptual models, implementations of freeform data types are explained in Chapter 4, and implementation of spatial functions are explained in Chapter 5. Chapter 6 presents the approach to exchange freeform shape data between Oracle Spatial 10i and MicroStation v8. Some test cases are created to demonstrate the result of the research, and these are illustrated in Chapter 7. The report is closed with conclusions and recommendations for future research in Chapter 8.

Chapter 2

Background

Before exploring how to manage freeform curves and surfaces in a spatial DBMS, familiarity with mathematical representation of freeform curves and surfaces and spatial DBMS is required. This chapter first shows the study of three widely used mathematical representations for freeform curves and surfaces: Bézier, B-spline and NURBS, and describes which parameters are required by these representations. Then spatial DBMSs are investigated by introducing OGC specifications and a mainstream spatial DBMS: Oracle spatial.

Section 3.1 gives the mathematical background of Bézier, B-spline and NURBS. Section 3.2 explains the spatial DBMS.

2.1 Freeform curves and surfaces

Implicit functions and parametric functions are the two most common forms of representing curves and surfaces in geometric modelling [11].

An implicit function has the form:

$$f(x, y, z) = 0 \tag{2.1}$$

A parametric function has the form:

$$p(u) = [x(u), y(u), z(u)] \qquad (u_{min} < u < u_{max}) \tag{2.2}$$

Parametric functions hold a number of advantages over implicit functions. Some of the most important ones include [3]:

- Points can be evaluated reasonably fast by numerically stable and accurate algorithms.
- There are more degrees of freedom than implicit functions have.

Represented with parametric functions, Bézier, B-spline and NURBS are three widely used mathematical models for freeform curves and surfaces in current CAD applications. Bézier, B-spline and NURBS all belong to an important geometric class called splines [6]. This section gives a glance of the three mathematical models.

2.1.1 Bézier

Consider n+1 control points P_k (k=0..n) in 3D space. The Bézier parametric curve (Figure 2.1) function is of the form:

$$C(u) = \sum_{k=0}^{n} P_k B_k^n(u) \qquad (0 \leq u \leq 1) \tag{2.3}$$

Figure 2.1: A cubic (degree 3) Bézier curve

B(u) is a Bernstein polynomial[1] and it is defined by:

$$B_k^n(u) = \frac{n!}{k!(n-k!)} u^k (1-u)^{n-k} \tag{2.4}$$

The extension of Bézier curves to surfaces is called the Bézier patch (see Figure 2.2). The patch is constructed from an (n+1)×(m+1) array of control points $\{P_{i,j} : 0 \leq i \leq n, 0 \leq j \leq m.\}$. The resulting surface, which is now parameterized by two variables, is given by the equation

$$P(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_{i,n}(u) B_{j,m}(v) \tag{2.5}$$

Most of the methods for patches are direct extensions of those for curves.

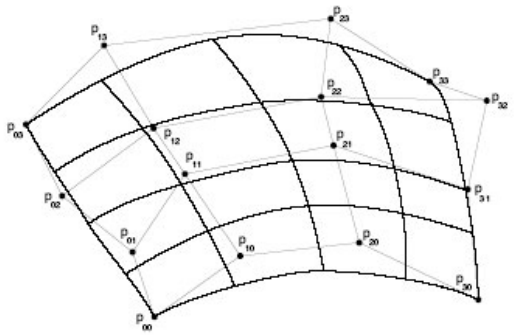

Figure 2.2: A bi-cubic (degree 3*3) Bézier surface

Some important properties of Bézier curves/surfaces are listed below:

- **Endpoint interpolation**: Bézier curves interpolate the first and the last control points: $P_0 = C(0)$ and $P_n = C(1)$; Bézier surfaces interpolate the four corner control points.
- **Smoothness**: The smoothness of a shape, which are made up of several curves, can be represented with the continuity at connections [12]. Figure 2.3 shows the smoothness of a shape with continuity 0, 1 and 2 at the connection. C^0 continuity means connected; C^1 continuity requires the same tangency at the the connection point; C^2 continuity is commonly referred to as geometric continuity that can be visually recognized as something "very smooth"; it is very difficult to visualize the difference of smoothness for parts with continuity bigger than 2. With a Bézier curve, we can only get a smooth

[1] A polynomial is a mathematical expression involving a sum of powers in one or more variables multiplied by coefficients.

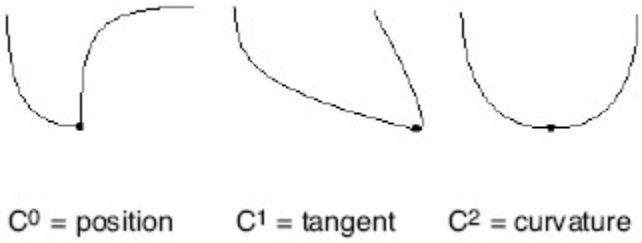

Figure 2.3: Continuity

curve of a given degree, with C^1 continuity, by defining a sequence of curves, each curve defined by (degree+1) points. The directions of $P_1 - P_0$ and $P_n - P_{n-1}$ are always tangent to the Bézier curves at the two endpoints P_0 and P_n respectively. Hence, in order to make two Bézier curves join smoothly, we need to put the last two control points of the first curve and the first two of the second curve in line.

- **Convex hull**: the curves are contained in the convex hulls of their defining control points.
- **Affine invariance**: one can apply to Bézier curves/surfaces the usual transformations, such as rotations, translations, and scalings, by just applying them to the control polygon/net.

2.1.2 B-spline

A B-spline (Figure 2.4) is a generalization of the Bézier curve. A B-spline curve of degree p

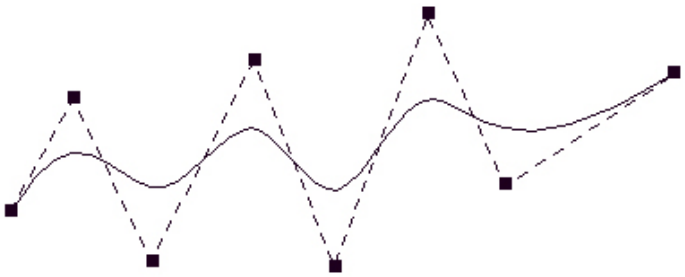

Figure 2.4: A B-spline curve with 8 control points

is defined by n+1 control points $P_0, ..., P_n$ and a knot vector of m+1 knots:

$$U = \{u_0, u_1, ..., u_m\}$$

where U is a nondecreasing sequence with $u_i \in [0, 1]$, and n, m and p must satisfy:

$$p \equiv m - n - 1 \tag{2.6}$$

The B-spline parametric curve function is of the form:

$$C(u) = \sum_{i=0}^{n} P_i N_{i,p}(u) \tag{2.7}$$

$N_{i,p}(u)$ are the basis functions of B-splines, defined by:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \tag{2.8}$$

A B-Spline surface is an expansion of B-spline curves in two directions, with corresponding control points, knot vectors, and univariate B-spline functions. The surface is defined by

$$S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} N_{i,k}(u) N_{j,l}(v) \tag{2.9}$$

where the k,l are the orders (degree+1) of the B-spline surface in both directions. The $N_{i,k}(u)$ are the polynomial B-spline basis functions of degree k-1 in the u parameter direction, and $N_{j,l}(v)$ are the basis functions of degree l-1 in the v direction.

If the knot vector of a B-spline curve is just (p+1) zeros followed by (p+1) ones, this B-spline curve reduces to a Bézier curve. Therefore actually B-spline curves are a generalization of Bézier curves, and the same applies for surfaces.

Here are some important properties of B-spline curves/surfaces:

- **Smoothness**: A p-degree B-spline curve/surface has continuity of (p-k) at the knots of multiplicity k (k knots clamped together). Figure 2.5 shows a 4-degree B-spline curve. It is easy to calculate that the curve at the three multiple knots has continuity of 2, 1, 0 respectively. For continuity 1, the corresponding point lies on the control polygon, while the curve pass through a control point for continuity 0, which results in a visual discontinuity.

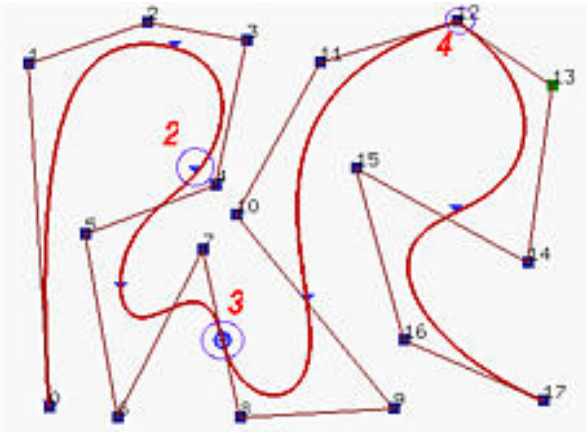

Figure 2.5: A B-spline curve with different continuity. The multiple knots are marked with circles; the numbers beside the circles are the multiplicities of the knots.

- **Strong convex hull**: The curves/surfaces are contained in the convex hulls of their defining control points. Especially, for a B-spline curve, if $u \in [u_{i_0}, u_{i_0+1}), p \leq i_0 \leq m-p-1$, then the curve C(u) is in the convex hull of the control points P_i, $i_0-p \leq i \leq i_0$. For a B-spline surface with a knot vector $\{u_0, u_1, ..., u_m\}$ in u direction and a knot vector $\{v_0, v_1, ..., v_n\}$ in v direction, if $(u, v) \in [u_{i_0}, u_{i_0+1}) \times [v_{j_0}, v_{j_0+1})$, then the surface S(u,v) is in the convex hull of the control points $P_{i,j}$, $i_0 - p \leq i \leq i_0$ and $j_0 - q \leq j \leq j_0$.

- **Affine invariance**: An affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation). In general, an affine transformation is a composition of rotations, translations, dilations, and shears, and they are all subclasses of projective transformations [6]. For a B-spline curve/surface, one can apply the affine transformations by just applying them to the control polygon/net.

- **Local modification scheme**: Moving P_i changes a B-spline curve only in the interval $[u_i, u_{i+p+1}]$, because $N_{i,p}(u) = 0$ for $u \notin [u_i, u_{i+p+1}]$; similarly, moving $P_{i,j}$ only affects a B-spline surface in rectangle $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$, because $N_{i,p}(u)N_{j,q}(v)$ is zero if (u,v) is outside of the this rectangle.

2.1.3 NURBS

NURBS(Nonuniform Rational B-Splines) are nearly inevitable for computer-aided design, manufacturing and engineering (CAD, CAM, CAE) and are part of numerous industry wide used standards, e.g. IGES, STEP, and PHIGS [3]. Different from Bézier and B-spline, NURBS support more freeform to modeling with the weights of control points, and only NURBS are able to define the exact conic sections.

A NURBS curve C(u) (Figure 2.6), which is a vector-valued piecewise rational polynomial function, is defined as:

$$C(u) = \frac{\sum_{i=0}^{n} w_i P_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)} \tag{2.10}$$

where

w_i: weights

P_i: control points (vector)

$N_{i,k}$: normalized B-spline basis functions of degree k, defined by Equation 2.8.

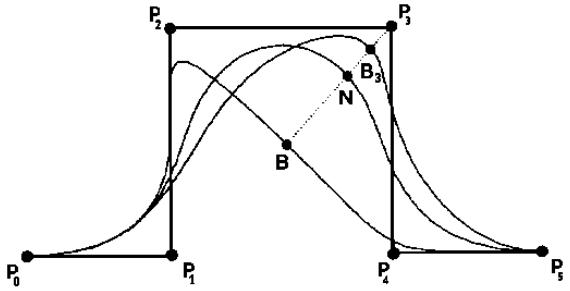

Figure 2.6: A NURBS curve with 6 control points. B, N, and B_3 illustrate the change of a point's position (same u) when the weight of P_3 is 0/1/(bigger than 1) respectively. [9]

Optional but important parameters for NURBS are the trim values. The lower and upper trim values: t_0 and t_1, extract a subcurve from a whole NURBS curve, and the restriction for trim values is that $u_0 \leq t_0 \leq t_1 \leq u_1$.

A NURBS surface is defined in a similar way:

$$S(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} N_{i,k}(u) N_{j,l}(v) P_{ij}}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j} N_{i,k}(u) N_{j,l}(v)} \tag{2.11}$$

where the k,l are the orders (degree+1) of the NURBS surface in both directions. The $N_{i,k}(u)$ are the polynomial NURBS basis functions of degree k-1 in the u parameter direction, and $N_{j,l}(v)$ are the basis functions of degree l-1 in the v direction.

NURBS differ from B-splines mainly in that the control points of NURBS have weight values. Specially, when all weights of a NURBS curve are equal, this NURBS curve becomes a B-spline curve. Therefore actually NURBS curves are a generalization of B-spline curves, and the same applies for surfaces.

Below are some important properties of NURBS curves/surfaces.

- **Endpoint interpolation**: NURBS curves interpolate the first and the last control points: $P_0 = C(0)$ and $P_n = C(1)$; NURBS surfaces interpolate the four corner control points.
- **Smoothness**: A p-degree NURBS curve/surface has continuity of p-k at the knot of multiplicity k. The NURBS curves/surfaces are smooth with continuity larger than 1, and are less smooth with continuity 1 and 0. This property is similar to the same property of B-splines.
- **Strong convex hull**: The curves/surfaces are contained in the convex hulls of their defining control points. Especially, for a NURBS curve, if $u \in [u_{i_0}, u_{i_0+1}), p \leq i_0 \leq m-p-1$, then C(u) is in the convex hull of the control points P_i, $i_0 - p \leq i \leq i_0$. For a NURBS surface, if $(u, v) \in [u_{i_0}, u_{i_0+1}) \times [v_{i_0}, v_{i_0+1})$, then S(u,v) is in the convex hull of the control points $P_{i,j}$, $i_0 - p \leq i \leq i_0$ and $j_0 - q \leq j \leq j_0$;
- **Projective invariance**: One can apply to curves/surfaces not only the affine transformations, but also the projective transformations, by just applying them to the control polygon/net;
- **Local modification scheme**: Moving P_i or changing its weight changes a NURBS curve only in the interval $[u_i, u_{i+p+1}]$; and moving $P_{i,j}$ or changing its weight only affect a NURBS surface in the rectangle $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$.
- **Conic section**: Only NURBS can represent exactly the conic sections, i.e. circles, ellipses, cones, which are the curves generated by the intersections of a plane with one or two nappes of a cone [6]. Such curves and surfaces occur very frequently in CAD/GIS applications, where several shapes and geometric constructions are based on such geometric primitives [3].

2.1.4 Summary

Bézier, B-spline and NURBS are three widely used mathematical models for representing freeform curves and surfaces. From the introduction in this section, we know that various helpful properties hold for Bézier, B-spline and NURBS. For example: define curves/surface using control points; able to produce smooth shapes; affine (projective under NURBS) invariance; modify curves/surfaces locally instead of globally (except Bézier); etc. Bézier curves/surfaces are special cases of B-spline curves/surfaces, and Bézier and B-spline curves/surfaces are special cases of NURBS curves/surfaces.

The required parameters for the three models can be summarized as:

- All the three models use control points and degree to define a curve/surface.
- Bézier doesn't require other parameters than control points and degree.
- B-spline require an additional parameter: knot vector.
- NURBS also require weight values in addition to the parameters required by a B-spline.
- Surfaces require knot vector and degree in both u and v direction.

2.2 Spatial DBMS

According to [13], the GIS, i.e. the integration of semantic, geometric data and spatial relationships, seems to be the most appropriate system ensuring a large scope of analysis and thus serving many applications and daily activities. Both CAD applications and DBMSs are trying to represent geometric data together with semantic data. Several years' attempts have achieved some agreements on the manner for representing, accessing and disseminating spatial information, i.e. the OGC specifications [15][18], and there are already a number of DBMSs (Oracle Spatial, PostGIS, Informix, etc.) which follow the OGC specifications.

This section gives an overall introduction to spatial DBMS by first explaining the OGC specifications, and then demonstrating one representative spatial DBMS on the market: Oracle Spatial.

2.2.1 OGC Specifications

The OGC (Open GeoSpatial Consortium) is a non-profit organization dedicated to open systems geoprocessing [16], and they give directions and recommendations to GIS researchers. Much geospatial data is available via the Web and in off-line repositories, but most of these data are stored in different data formats, using different data models, coordinate reference systems, geometry models, etc. Thus, sharing spatial data has required considerable time, expertise and special software. OGC specifications define a series of common software interfaces and encodings, which enable users to carry out their research on the same data format and concentrate on the same research direction. Nowadays OGC specifications have been widely accepted by GIS researchers.

Two main categories of OGC specifications are the **Abstract Specifications** and **Implementation Specifications**.

Abstract Specifications

The purpose of **Abstract Specifications**[1] is to create and document a conceptual model sufficient enough to allow for the creation of Implementation Specifications. One OGC abstract specification, the **Spatial Schema**, gives a standard for geographic information by specifying geometry and topology separately as a UML (Unified Modeling Language)[2] package dependency tree. The geometry package (Figure 2.7) has several internal packages that separate primitive geometric objects, aggregates and complexes, which have a more elaborate internal structure than simple aggregates [7]. Two representations of freeform curves and surfaces, i.e. Bézier curve/surface and B-spline curve/surface, have been standardized in the geometric package.

Figure 2.8 shows the spline curves packages defined in **Spatial Schema**, where Bézier curve and B-spline curve are specified in the **GM_BSplineCurve** package. Different from the parameters explained in Section 2.1, four parameters: 'degree', 'curveForm', 'knotSpec' and 'isPolynomial', are required by this package. Their explanations are as follows [7]:

- The '**degree**' attribute specifies the algebraic degree of the basis functions;
- The '**curveForm**' attribute is used to identify particular types of curve which this spline is being used to approximate, or set to 'NULL' if no such approximation is intended;
- The '**knotSpec**' attribute gives the type of knot distribution used in defining this spline;

[1] This abstract specification has also been accepted by ISO and is referred as **ISO 19107**.
[2] See [17] for an explanation of UML.

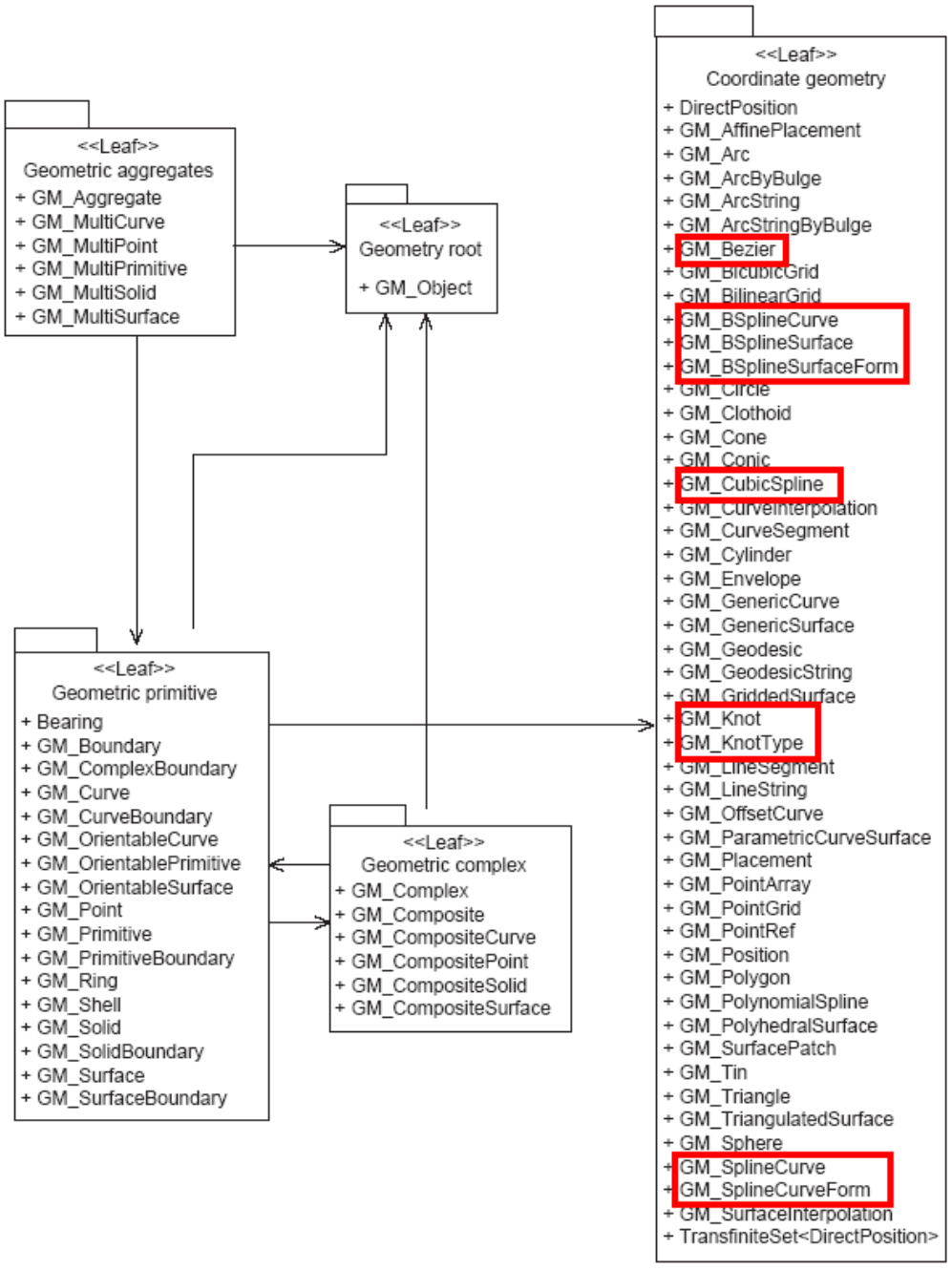

Figure 2.7: Geometric package in the **Spatial Schema** [7]. The Bézier type, B-spline type and their related types are marked with red boxes.

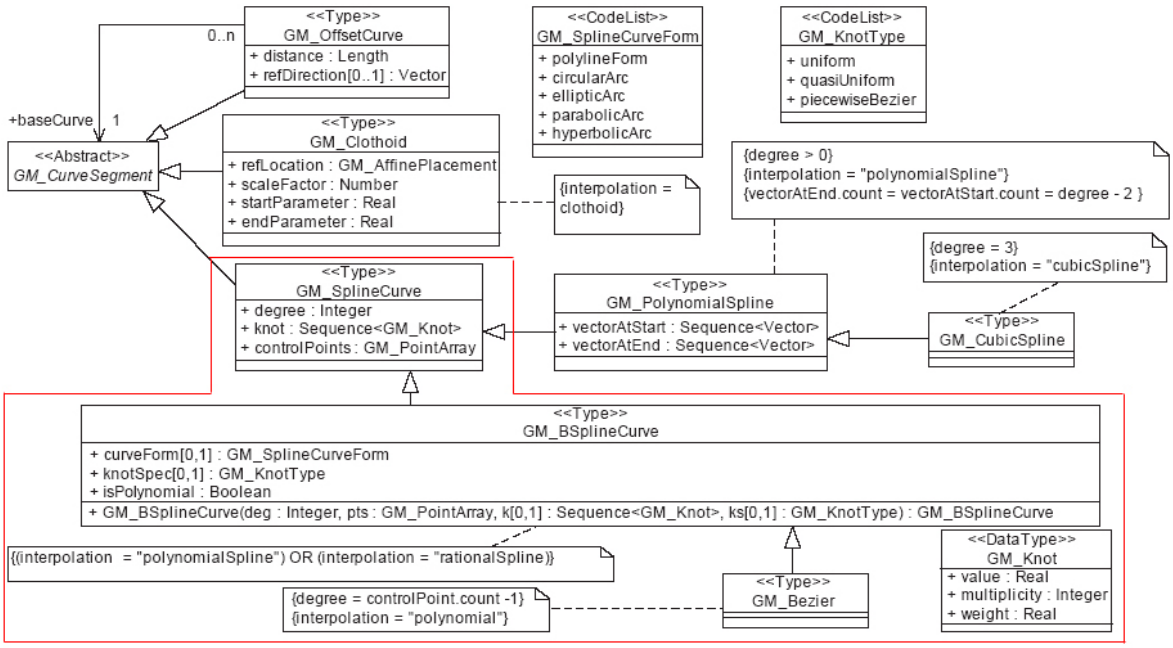

Figure 2.8: Spline packages in **Spatial Schema**

- The '**isPolynomial**' attribute is set to 'True' if this is a polynomial spline. The difference between a polynomial spline and a normal B-spline in this package is that a polynomial spline passes through the control points, while a normal B-spline doesn't.
- The class constructor '**GM_BSplineCurve**' takes four parameters (degree, control points, knots, knot types) to construct a B-spline curve.

NURBS is not explicitly standardized in **Spatial Schema**. However, as one of the most powerful mathematical representations for freeform curves and surfaces, NURBS have already been included in many geometric standards, such as OpenGL, IGES, STEP and PHIGS, and supported by mainstream CAD applications (AutoCAD, MicroStation). For example, function '*gluNurbsCurve()*' in OpenGL is used to describe a NURBS curve with the following parameters [19]:

nknots Specifies the number of knots.

knot Specifies an array of knot values.

stride Specifies the offset (as a number of single-precision floating-point values) between successive curve control points.

ctlarray Specifies a pointer to an array of control points.

order Specifies the order of the NURBS curve.

Implementation Specifications

The **Implementation Specifications** are unambiguous technology platform specifications for implementation of industry-standard, software application programming interfaces [15]. One OGC implementation specification that we should pay attention to is the **OpenGIS Simple Features Specification for SQL (SFS)**[18], because this specification defines a

standard SQL schema for simple geospatial features (geometric primitives) which are based on OGC **Abstract Specifications**.

According to SFS, a spatial SQL environment should contains the following aspects.

Spatial data type A spatial DBMS should support the following spatial data types :

- Geometry
- Point
- Curve
- LineString, Line, LinearRing
- Polygon
- Surface
- GeometryCollection
- MultiPoint
- MultiCurve
- MultiLineString
- MultiPolygon
- MultiSurface

The difference between Multi type and non-Multi type is that the first type is a geometric collection of the latter type.

Exchange formats The Well-Known Text (WKT) representation and the Well-Known Binary (WKB) representation are recommended to be used to exchange (import/export) spatial data. WKT/WKB provides standard textual/binary representations for spatial reference system information. The nine geometries above, including simple andnon-simple, closed and non-closed, can be represented accurately using WKT/WKB. For example:

a Polygon with one exterior ring and one interior ring represented with WKT:

```
POLYGON((0 0,10 0,10 10,0 10,0 0),(5 5,7 5,7 7,5 7, 5 5))
```

Spatial operators/functions A spatial database cannot 'manage' spatial data types without spatial operators/functions. SFS also defines a lot of spatial functions, including functions for constructing a geometry value given its WKT/WKB, functions that test spatial relationships, functions for distance relationships, functions that implement spatial operators and spatial functions on each spatial data type. SFS doesn't define any validation function. However, validity of spatial data should be checked before insertion.

Metadata Metadata, or in other words, the descriptions of the table columns with spatial data types, should be stored in a separate metadata table. The required descriptions include the name of the column, the name of the table containing this column, dimension, spatial type, **Spatial Reference System**, etc.

Spatial Reference System A spatial DBMS should also develop its **Spatial Reference System**, which identifies the coordinate system for all geometries stored in the geometric columns, and gives meaning to the numeric coordinate values for any geometry instance stored in the columns.

Although OGC **Abstract Specifications** also defines standards for complex features (topological primitives), they are still missing in SFS and other OGC **Implementation Specifications**. Freeform shapes like Bézier, B-spline or NURBS curve/surface are not mentioned in the OGC **Implementation Specifications**.

2.2.2 Oracle Spatial

Oracle Spatial is one of the most powerful spatial DBMSs on the market. Oracle series began to support spatial data in its option **Oracle Spatial** since Oracle 8i. Partially compliant with **Simple Features Specification for SQL**, Oracle Spatial supports several spatial types specified in SFS. Oracle Spatial is considered partially compliant with OGC specifications but not completely compliant because in Oracle Spatial there is no separate data types for point, linestring, polygon, etc., but there is uniform data type: SDO_GEOMETRY to represent all spatial data types. Beside spatial data types, a large number of spatial functions are available in Oracle Spatial as well.

Spatial types

Unlike other spatial DBMSs in which different spatial types represent different geometries, there's only one spatial type: SDO_Geometry. Before explaining the geometry types that can be represented with SDO_Geometry, the Entity Relationship diagram in Figure 2.9 would be helpful to understand how SDO_Geometry works.

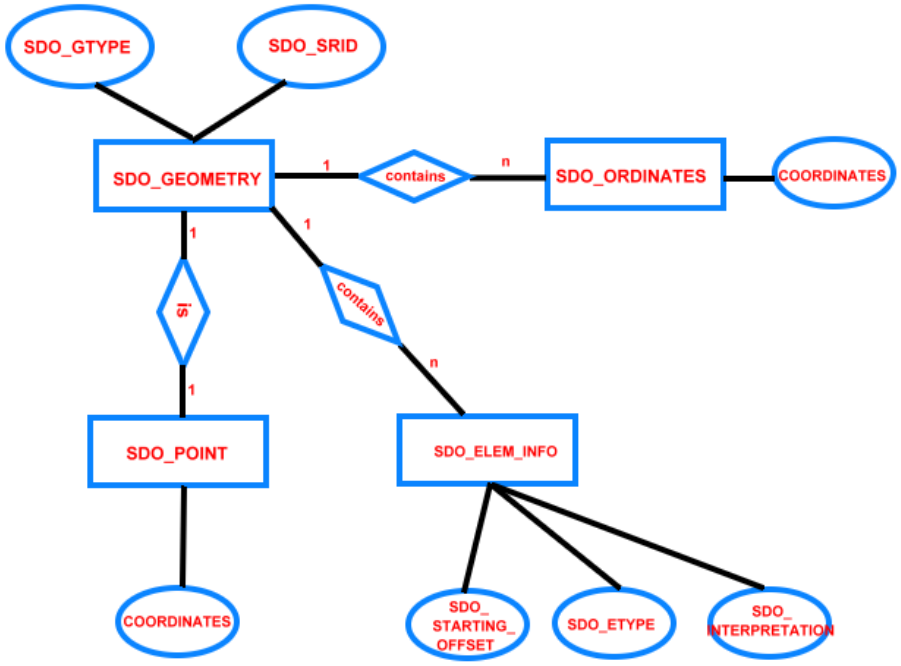

Figure 2.9: ER diagram for SDO_Geometry

Oracle enables users to define new data types (user-defined data types) which are made up of several attributes. These attributes can be of basic data types such as numbers, varchar2, date, or other existing user-defined data types. SDO_GEOMETRY is created as a user-defined data type. Several kinds of geometries can be represented by setting attributes of SDO_GEOMETRY. Oracle Spatial defines the object type SDO_GEOMETRY as [20]:

```
CREATE TYPE sdo_geometry AS OBJECT (
SDO_GTYPE NUMBER,
SDO_SRID NUMBER,
SDO_POINT SDO_POINT_TYPE,
SDO_ELEM_INFO SDO_ELEM_INFO_ARRAY,
```

```
SDO_ORDINATES SDO_ORDINATE_ARRAY);
```

SDO_GTYPE The SDO_GTYPE (geometry type) value is 4 digits in the format dltt. d is the number of dimension (2, 3, or 4); l specifies which dimension (3 or 4) contains the measure value[3]; tt represents geometry types, for example, dl01 represents point, dl02 represents line/curve, dl03 represents polygon, etc. Supported geometry types are:

- UNKNOWN_GEOMETRY
- POINT
- LINE/CURVE
- POLYGON(with/without holes)
- COLLECTION
- MULTIPOINT
- MULTILINE/MULTICURVE
- MULTIPOLYGON.

SDO_SRID This attribute is used to relate a spatial reference system to this geometry. "null" values means this geometry doesn't relate to any reference system.

SDO_POINT When SDO_POINT value is not null, this geometry is just a single point, otherwise it is one of other geometries. Using SDO_POINT is convenient when there're only point geometries in a layer.

SDO_ELEM_INFO This attribute indicates the format of coordinates in SDO_ORDINATES by repeating the following attributes:

SDO_STARTING_OFFSET This attribute indicates the starting offset of current element's first coordinate in SDO_ORDINATES;

SDO_ETYPE Type of current element, can be simple elements or compound elements. Especially, geometries which are not supported by Oracle Spatial can be represented by setting SDO_ETYPE to zero. Geometries with type 0 elements must contain at least one nonzero element, which should be an approximation of the unsupported geometry.

SDO_INTERPRETATION Interpretation for SDO_ETYPE. Especially, when this is a zero type element, the SDO_INTERPRETATION value for the type zero element can be any numeric value, and applications are responsible for determining the validity and significance of the value [20].

SDO_ORDINATES Arrays of coordinates are stored here with the format specified in SDO_ELEM_INFO.

The SQL statements below create a spatial table with a spatial column, and then insert a polygon in into this table.

```
CREATE TABLE example_table(
id NUMBER,
shape SDO_GEOMETRY);

INSERT INTO example_table VALUES(
```

[3]Measure value is used to reflect the relation between world coordinate and local coordinate [20].

```
1,
SDO_GEOMETRY(
2003, --two-dimensional polygon
NULL,
NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),

      --1003 indicates simple polygon, 1 indicates vertices are connected by
                                                          straight line segments

SDO_ORDINATE_ARRAY(-1,-1, -1,1, 1,1, 1,-1)

                                                --coordinates for four vertices

));
```

Notice that the SQL statement above is not the only possibility to represent this polygon. We can also set SDO_ETYPE to 1003 and SDO_INTERPRETATION to 3, which indicates a rectangle. In this case, only coordinates of top-right and bottom-left vertices should be specified in SDO_ORDINATE_ARRAY.

Spatial operators and functions

Oracle Spatial support a large number of operators and functions for spatial data types. Spatial operators in Oracle Spatial provide optimum performance because they use the spatial index, which is an R-tree of MBR (minimum bounding rectangle) around geometry shapes[4], on spatial columns. Spatial operators must be used in the WHERE clause of a query. Spatial functions in Oracle Spatial differ from spatial operators in that they do not require that a spatial index be defined, and they do not use a spatial index if it is defined. These spatial functions can be used in the WHERE clause or in a subquery. Both spatial operators and spatial functions are included in the **Spatial PL/SQL application programming interface** (API).

Spatial query and operations using the existing operators and functions in Oracle is quite convenient. These spatial operators and functions can be mainly categorized into:

- Relationship (True/False) between two objects: RELATE, WITHIN_DISTANCE
- Validation: VALIDATE_GEOMETRY_WITH_CONTEXT, VALIDATE_LAYER_ WITH _CONTEXT
- Single-object operations: SDO_ARC_DENSIFY, SDO_AREA, SDO_BUFFER, SDO_CENTROID, SDO_CONVEXHULL, SDO_LENGTH, SDO_MAX_MBR_ORDINATE, SDO_MIN_MBR_ORDINATE, SDO_MBR, SDO_POINTONSURFACE
- Two-object operations: SDO_DISTANCE, SDO_DIFFERENCE, SDO_INTERSECTION, SDO_UNION, SDO_XOR

The following example shows how to get all the geometries which contain a rectangle with vertices at (-0.5,-0.5),(-0.5,0.5),(0.5,0.5),(0.5,-0.5) using spatial operators.

```
select a.id from example_table a
where SDO_CONTAINS(a.shape,
SDO_GEOMETRY(2003,NULL,NULL,
SDO_ELEM_INFO_ARRAY(1,1003,3),
```

[4]See [20] for more information of spatial index in Oracle Spatial

```
SDO_ORDINATE_ARRAY(-0.5,-0.5, -0.5,0.5, 0.5,0.5, 0.5,-0.5))
)='TRUE';
```

The result is '1', which means the spatial column 'shape' of the row with id '1' contains the rectangle.

The following example shows to how to get the centroid of the geometry with id '1' using spatial function.

```
select id, SDO_GEOM.SDO_CENTROID(shape, 0.005)
from example_table where id=1;
```

The result is 'SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(0, 0, NULL), NULL, NULL)', which means point (0,0) is the centroid of the geometry with id '1' (a rectangle with vertices(-1,-1), (-1,1), (1,1), (1,-1)).

Although these operators and functions are quite powerful and convenient, they cannot be applied to new geometry types because the algorithms are based on existing geometry types. In order to make the spatial database fully supporting new geometry types, for example freeform curves and surfaces, we have to create the operators and functions by writing PL/SQL (Procedural Language extensions to SQL) codes, JAVA codes or C codes.

2.2.3 Other spatial DBMSs

There are several spatial DBMSs on the market, such as Ingres, Informix, PostGIS, and Oracle Spatial. All of them have spatial data types and spatial functions, but the differences in their spatial functionalities are significant, which can be revealed from the following comparison. Oracle Spatial is also included in this comparison as a reference spatial DBMS.

- Informix, PostGIS and Oracle Spatial support OGC specifications, whereas Ingres doesn't.
- The spatial types can only be 2D in Ingres, but they can be 3D or 4D (with measure values) in other ones.
- Supported spatial types are different. Oracle Spatial supports most of the data types specified in SFS, and unknown geometry types can be stored; PostGIS supports the same spatial data types as Oracle Spatial, but unknown geometries cannot be stored; Informix doesn't have Multi-Geometry type, which is a collection of different basic geometry types; Ingres doesn't support any Multi- types, like MultiPoint, MultiPolygon, etc.
- Numbers of spatial functions are different. Oracle Spatial and PostGIS give more useful spatial functions than Informix and Ingres.

2.2.4 Summary

Current spatial DBMSs are able to support simple geometries like point, line string, polygon, and etc., Which and how geometries should be supported has been specified in some standards: OGC specifications. Among the three popular freeform geometries: Bézier, B-spline and NURBS, only the first two are mentioned in the **Abstract Specification**. None of the three is mentioned in the **Implementation Specification**.

As a spatial DBMS, Oracle Spatial gives support to simple geometries with a uniform spatial data type: SDO_Geometry. Different kinds of simple geometries are stored with

SDO_Geometry by specifying different attribute values. Freeform geometries are not supported by Oracle Spatial yet.

2.3 Concluding remarks

After the study in this chapter, the following conclusions can be stated:

- Bezier, B-spline and NURBS can be used to represent freeform curves and surface. Bezier require the degree and control points as parameters; B-spline require degree, control points, and knot vector; NURBS require degree, control points, knot vector and weight values. Freeform surfaces require degree and knot vector in two directions.
- OGC specifications mention Bezier and B-splines curve/surface in Abstract Specifications. NURBS is still missing, although it is a powerful representation for freeform curves and surfaces, considering its helpful properties.
- Oracle Spatial is able to store some simple geometry types with a uniform data type: SDO_Geometry, and manipulate them with a number of functions and operators on SDO_Geometry.
- New data types can be created in Oracle by creating user-defined data types. Therefore creation of data types for freeform curves and surfaces is possible in Oracle.

The study and investigation in this chapter has answered the first sub-question in Chapter 1. In the next chapter, conceptual models based on the background in this chapter will be designed.

Chapter 3

Conceptual design

From Section 2.2 we know that geometries can be managed in spatial DBMSs with spatial data types and spatial functions, as specified in OGC Abstract and Implementation Specifications. OGC Abstract Specification: **Spatial Schema** specified a number of geometries which should be supported in a spatial DBMSs, and some simple geometries are chosen and specified in OGC Implementation Specification: **SFS**. Most of the simple geometries specified in SFS have been implemented in current spatial DBMSs. A lot of spatial functions on these geometries, for example, translation, rotation and intersection, are also available in Oracle Spatial.

This chapter discusses the conceptual design for each DBMS module in Figure 3.2, i.e. freeform data types and functions. In each section, motivations and possibilities are discussed first, then the determined conceptual models are given and explained. Section 3.1 discusses the conceptual models of freeform data types. Section 3.2 discusses the potential functions on freeform data types.

3.1 Freeform data types

From Section 2.1 we know that freeform curves and surfaces can be represented with three powerful mathematical models: Bézier, B-spline and NURBS.

For **freeform curves**, there are two possibilities for the conceptual model. The first is to create data types for all of Bézier curve, B-spline curve and NURBS curve. An alternative approach is to just create a data type for NURBS curves, and represent Bézier curves and B-spline curves by leaving some parameters of NURBS curve empty, because NURBS curve is actually the generalization of Bézier curve and B-spline curve. I didn't adopt the second approach mainly because:

- Leaving the empty values for some parameters will decrease system efficiency, as many unnecessary empty values should be specified and stored.
- OGC specification: **Spatial Schema** recommends different data types for different shapes.
- Some geometry algorithms are different for Bézier curve, B-spline curve and NURBS curve.

Similar to freeform curves, conceptual models for **freeform surface** can be a model for generalized NURBS surface, or models for all of Bézier surface, B-spline surface and NURBS surface. I decided to only create data type for NURBS surface instead of for all three kinds of surfaces. This is mainly because the whole procedure of creating Bézier surface, B-spline surface and NURBS surface will be very similar, and Bézier surface, B-spline surface can be represented in NURBS form by leaving **knots** parameter or **weight** parameter empty. In the research stage, it is wise and more representative to only implement the most complex type:

NURBS surface, instead of repeating the similar procedure for all three, hence time can be saved for more creative research.

The mathematical background in Chapter 1 summarized that:

- A Bézier curve is defined by several **control points**.
- A B-Spline curve is defined by several **control points** and a sequence of **knots**.
- A NURBS curve is defined by several **control points** and a sequence of **knots**; each control point has a **weight** value; sometimes **trim** values are required to represent part of a whole NURBS curve.
- A NURBS surface is defined by a control points net and knots in both **u and v** directions. Each control point has a weight value; trim values in both u and v directions are optional.
- **Degree** is an important factor for freeform curves and surfaces.
- B-Spline is generalization of Bézier, and NURBS is generalization of B-Spline. Bézier curve, B-Spline curve and NURBS curve are all Spline curves.

All the parameters mentioned above are also included in the specification of freeform geometries in **Spatial Schema**, except that NURBS is still absent in **Spatial Schema**. However, **Spatial Schema** defines more parameters for each data type than the basic required parameters.

According to Figure 2.8, the GM_BSplineCurve in **Spatial Schema** requires three more parameters:

- *curveForm*. The *curveForm* attribute is used to identify particular types of curve which this spline is being used to approximate, for example: circular arc, elliptic arc, parabolic arc, etc. *curveForm* is an optional attribute of *GM_BSplineCurve* package, and it will not be included in my conceptual models for freeform curve. This is mainly because different CAD/GIS applications have different classifications for the curve forms, and fixed curve form classification inside spatial DBMS should be avoided. My conceptual model should include the common parameters of all front-ends. Additional and optional parameters like *curveForm* can be stored outside the instance of conceptual models, for example, other columns besides the column of freeform data type.
- *knotSpec*. *knotSpec* is another optional attribute which is used to identify particular types of knot vector, for example: uniform, quasiUniform, and etc. This attribute will not be included in my conceptual models either. This is because a knot vector is just a sequence of values, and whether this sequence is uniform or not can be easily checked by simple algorithms. Storage of such a parameter is a unnecessary data redundancy.
- *isPolynomial*. *isPolynomial* is an obligatory attribute which is used to identify whether this is a polynomial spline or a normal spline, according to [7]. In other words, *isPolynomial* is used to identify whether the spline interpolates or approximates the control points. *isPolynomial* will be included in my conceptual model. This is because, first, it is an obligatory attribute in **Spatial Schema**, and second, some CAD applications (AutoCAD and MicroStation) do have splines in both forms, and there are translation tools to convert one to another. *isPolynomial* attribute will be used in most cases instead of being assigned with the empty value.

The *GM_Knot* package in **Spatial Schema** is also different from common definition of knot vector. Actually, **Spatial Schema** defines a knot vector with a sequence of *GM_Knot*. Each *GM_Knot* requires the following parameters:

- *value*. This is the value of the knot.
- *multiplicity*. How many times this knot is repeated.
- *weight*. The "importance" of this knot in the whole knot vector. **Spatial Schema** doesn't give much mathematical explanation to this parameter, and there're a few publications which mention knot weight. According to [10], the *knot weight* is a temporary parameter which is used in a knot removal algorithm, where the *knot weight* values give an importance rank to all knots, and the least important knot will be removed.

The usage of *multiplicity* can save storage when most knots are repeated. For example, a knot vector {0,0,0,0,1,1,1,1} can be considered as only two *GM_Knot*: one with value 0 and multiplicity 4 and another one with value 1 and multiplicity 4. However, the disadvantage of this model is significant when some knot are not multiplied. For example, a knot vector {0, 0.2, 0.4, 0.6, 0.8, 1}, will have to store five *GM_Knot*, all of which have the multiplicity 1. This obviously results in data redundancy. In practice, the latter situation is more common, so therefore the *GM_Knot* package in **Spatial Schema** will not be adopted as the conceptual model. Instead, a model (as in Figure 3.1) of knot vector, which contains a value sequence attribute and a weight sequence attribute, would be more efficient than the *GM_Knot* package in **Spatial Schema**. In this approach, a knot vector is defined by a *GM_KnotVector* model. The knot values are listed in the *knots* attribute, and the weight values are listed in the *weights* attribute. Multiplicity of knots can be easily checked with algorithms. Normally a knot vector is not very long, therefore such checking should not be heavy.

Based on the mathematical definitions, the specifications in OGC specifications, considerations on practical purpose, and discussions above, conceptual models for Bézier curve, B-spline curve, NURBS curve and NURBS surface can be created as in Figure 3.1.

Following are some explanations for Figure 3.1.

- GM_SplineCurve is the basic class for GM_BezierCurve, GM_BSplineCurve and GM_NURBSCurve; control points, knots and degree are the parameters of GM_SplineCurve, and they are inherited by the other three curves.
- GM_NURBSSurface is a freeform surface class using NURBS surface representation;
- GM_Knot is the class for knot vector, which is used in GM_BSplineCurve, GM_NURBSCurve and GM_NURBSSurface as parameters;
- GM_PointArray and GM_WeightArray are simple Array types which enumerate real values, and they are used to represent control points and weight values in all the freeform curves and surface types;
- GM_Trim represents the trim values for GM_NURBSCurve and GM_NURBSSurface.

3.2 Functions on freeform data types

As this is the first attempt of managing freeform data types in a DBMS, there is no standard of which functions should be implemented for these types. OGC specifications didn't mention anything about the spatial function on freeform data types, either. In this case, standards and implementations of existing spatial types can be referenced. On the other hand, there should also be spatial functions which are special for freeform data types. Generally, three main factors are considered for the determination:

- Standards of spatial functions on existing spatial data types in OGC Implementation Specifications.

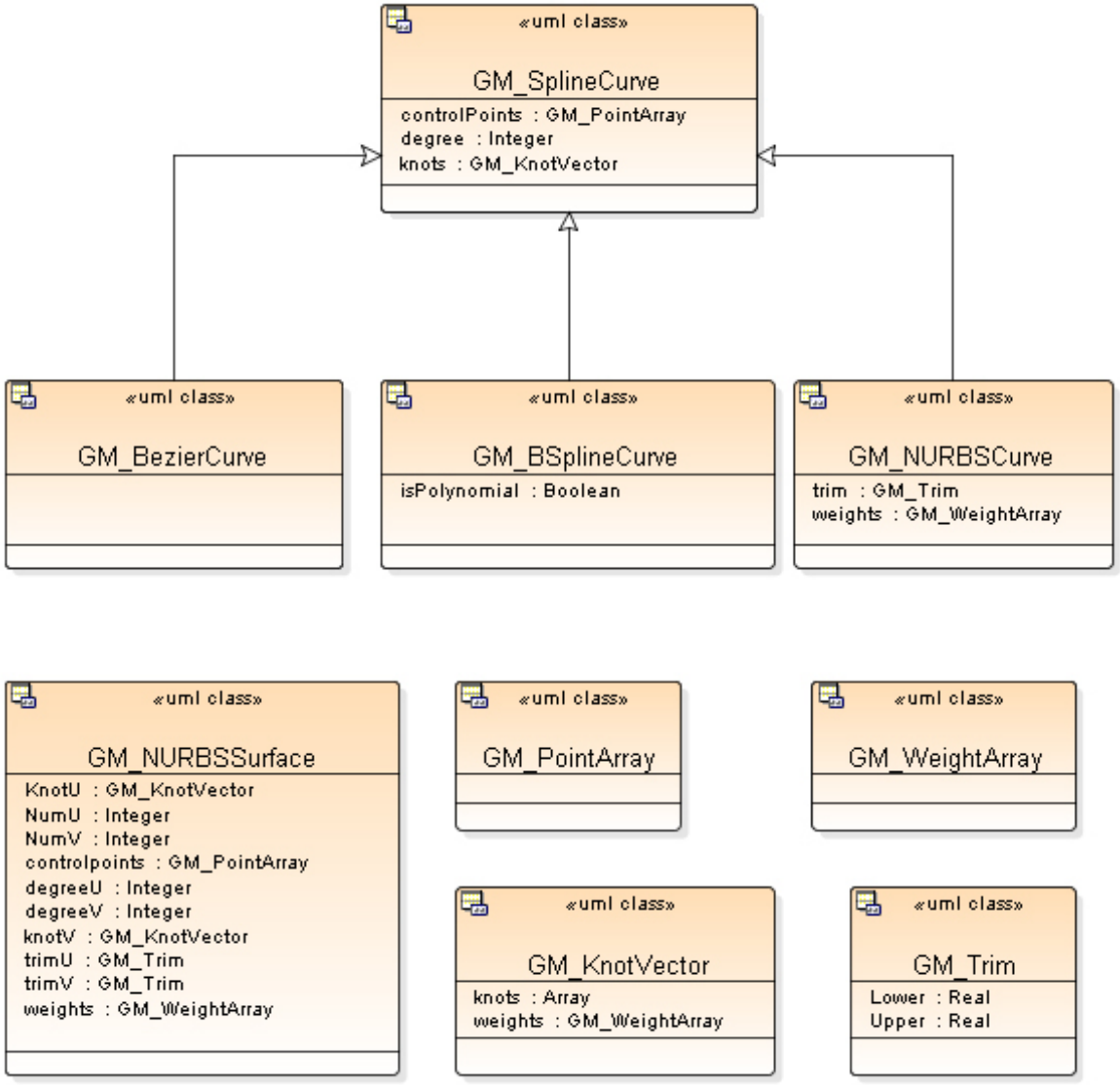

Figure 3.1: Conceptual models of freeform curves and surfaces

- Implemented spatial functions on existing spatial data types in current spatial DBMSs.
- Potential functions that would be useful for CAD applications.

SFS defines a number of spatial functions on existing spatial data types: constructor functions, functions that test spatial relationships, functions for distance relationships and functions on each spatial data type:

- The functions that test spatial relationships include "*Equals, Disjoint, Touches, Within, Overlaps, Crosses, Intersects, Contains, Relate*".
- The functions for distance relationships include "*Distance*".
- The functions that implement spatial operators include "*Intersection, Difference, Union, SymDifference, Buffer, ConvexHull* ".
- There should also be spatial functions for each spatial data type. For example, the spatial functions for 'curve' include "*StartPoint, EndPoint, IsClosed, IsRing, Length*".

Beside SFS, two mainstream spatial DBMSs: Oracle Spatial and PostGIS have been analyzed for their implemented spatial functions on existing spatial types. Besides the ones already mentioned in SFS, they both give several extra functions, including:

- `Centroid(g1 Geometry) : Double`
  Returns the geometry center of this geometry.
- `Buffer(g1 Geometry, d Double): Geometry`
  Returns a geometry that represents all points whose distance from this geometry is less than or equal to d.
- `GeomUnion(g1 Geometry, g2 Geometry): Double`
  Returns a geometry that represents the point set union of this geometry with another geometry.
- `MBR(g1 Geometry): Geometry`
  Returns the minimum bounding rectangle of this geometry.
- Etc.

From the above functions, the ones which are also meaningful to freeform curves and surface, have been chosen to be implemented for freeform data types.

Another consideration is whether certain functions should be available at DBMS level. DBMSs are famous for the speed and efficiency of data storage and processing, but they are not good at complex and heavy computation. On the other hand, a number of complex computation functions are already available in CAD/GIS applications. It would be convenient to have some basic function in DBMS, but it would be more logical to not include any complex geometry computation at DBMS level. For example, the curve evaluation functions are not considered because the algorithms are reasonably complex and are available in nearly all CAD/GIS APIs (AutoCAD, MicroStation). Furthermore, some access functions are required to return the basic geometry information, too. Considering the nice invariance property (See section 2.1) of Bézier, B-spline and NURBS, the basic geometry transformation functions (translation, rotation, scaling) should also be implemented. Conversion functions between Bézier curve, B-spline curve, and NURBS curve will be useful and required because different CAD/GIS applications give different levels of support to freeform data types.

From the considerations above, the functions required by freeform curves types include:

- `N(c Curve): Integer`
  Returns the number of control points
- `ConvexHull(c Curve) : Polyhedron`
  Returns the convex hull of this curve.
- `IsClosed(c Curve) : Boolean`
  Returns whether this curve is closed or open.
- `Centroid(c Curve): Point`
  Returns the geometry center of this curve.
- `cPolygon(c Curve): Polygon`
  Returns the control polygon
- `BoundingCube(c Curve): Cube`
  Returns the minimum bounding cube of this curve.
- `RotateX(c Curve, a Double): Curve`
  Returns a rotated curve by rotating the curve c along x axis by degree a.
- `RotateY(c Curve, a Double): Curve`
  Returns a rotated curve by rotating the curve c along y axis by degree a.
- `RotateZ(c Curve, a Double): Curve`
  Returns a rotated curve by rotating the curve c along y axis by degree a.
- `Translation(c Curve, x Double, y Double, z Double): Curve`
  Returns a translated curve by translating the curve c with (x,y,z).
- `Scale(c Curve, x Double, y Double, z Double): Curve`
  Returns a scaled curve by scaling the curve c with (x,y,z).
- `Distance(c1 Curve, c2 Curve): Double`
  Returns the approximated distance between two curves.
- `AnyIntersect(c1 Curve, c2 Curve): Boolean`
  Checks whether two curves intersect with each other.

Considering from the mathematical analogous between freeform curves and freeform surfaces, the functions required by freeform surface types are considered analogous to curve types, too.

Besides freeform curve and surface types, there should also be a few functions on the supplementary type: GM_KnotVector, because a knot vector contains much mathematical meaning, which can be returned and manipulated by functions. These functions are:

- `N(k KnotVector): Integer`
  Returns the number of knots
- `regulation(k KnotVector): KnotVector`
  Reconstruct the knot vector by recomputing all the value within [0..1].
- `validation(k KnotVector): number`

Other important functions are the validation functions. The spatial data is checked when it is inserted into the DBMS or when it is changed in the DBMS; this check on the geometry of the spatial objects is called validation. It is important because valid objects are necessary to

make sure the objects can be manipulated in a correct way, e.g. it is impossible to construct a freeform curves with degree -1. Validation functions need a set of rules to check the columns with freeform data types.

Currently there are no existing geometry validity rules for a freeform geometry, therefore I formulated several rules for valid freeform geometry from the mathematical background of freeform geometry and validation rules for simple geometries.

A freeform geometry is valid when:

- It is storage valid, which means there're enough parameters to construct this shape.
- It is geometry valid, which means it is possible to construct a freeform shape with the stored parameters. A geometry valid freeform geometry has to be a storage valid freeform geometry first.

Storage validity

Validation rules of storage validity varies for different freeform types. However, they mainly ensure that the parameters are enough to construct the geometry.

For example, a NURBS curve is storage valid when it satisfies the conditions:

- Coordinates for control points are not missing.
- Degree value is not missing.
- Weight values are not missing.
- Knot vector is not missing.

The storage validity rules for NURBS surface are analogous:

- Control points are not missing.
- Degree value in u direction is not missing.
- Degree value in v direction is not missing.
- Number of control points in u direction is not missing.
- Number of control points in v direction is not missing.
- Weight values are not missing.
- Knot vector in u direction is not missing.
- Knot vector in v direction is not missing.

Geometry validity

If a freeform geometry satisfies all the storage validation rules, it will be checked with several geometry validation rules, to ensure that this is a completely valid freeform geometry. The geometry validation rules are more complex than the storage validation rules, and these geometry validation rules mainly come from the mathematical definition of geometries. For example, the mathematical definition of a knot vector is: **a nondecreasing sequence.**

We can derive two geometry validation rules from this definition:

1. Any knot (except the first one) value should be larger or equal to the previous knot value.

2. The length of this knot vector should be larger or equal to 2, otherwise it is not a sequence.

Analogously, we are able to derive geometry validation rules for all freeform geometries from their definitions. Here we list the rules for NURBS curves:

1. degree > 1;
2. number of control points ≥ 3;
3. Degree = Number of knots - Number of control points - 1;
4. The number of weight values is equal to the number of control points.
5. Each weight value > 0;
6. Knot vector is non-decreasing and has more than 1 knot;
7. upper trimming value > lower trimming values.

Validation functions on freeform data types can be created by checking the validity rules above one by one, and return different values for storage invalid, geometry invalid and valid freeform data.

3.3 Concluding remarks

After the discussion in this chapter, conceptual models for freeform data types and functions are designed. The overall architecture for managing freeform data types in a spatial DBMS can be drawn in Figure 3.2. As illustrated, several freeform data types and functions on freeform data types need be designed in a spatial DBMS. Freeform spatial data can be retrieved or stored by users via either CAD front ends or SQL Plus. CAD front ends can be used to visualize the freeform spatial data from DBMS, or model freeform geometries and store them into DBMS. An alternative method to access freeform spatial data in DBMS is to use SQL Plus, where users store or retrieve freeform spatial data by writing SQL statements manually. Functions on freeform data types can be invoked by either SQL Plus or CAD front ends to return requested information.

Considered from mathematical background of freeform geometry and OGC specifications, the freeform data types to be created include:

- Three curve types: GM_BezierCurve, GM_BSplineCurve and GM_NURBSCurve.
- One surface type: GM_NURBSSurface.
- Four supplementary types: GM_PointArray, GM_WeightArray, GM_KnotVector and GM_Trim.

Considered from OGC specifications and functions on existing spatial data types in current spatial DBMSs, the functions to be created on freeform data types include:

- Access functions. For example, the function to check whether a freeform curve is closed or not.
- Geometry relationship functions. For example, the function to check whether two freeform geometry may intersect with each other.

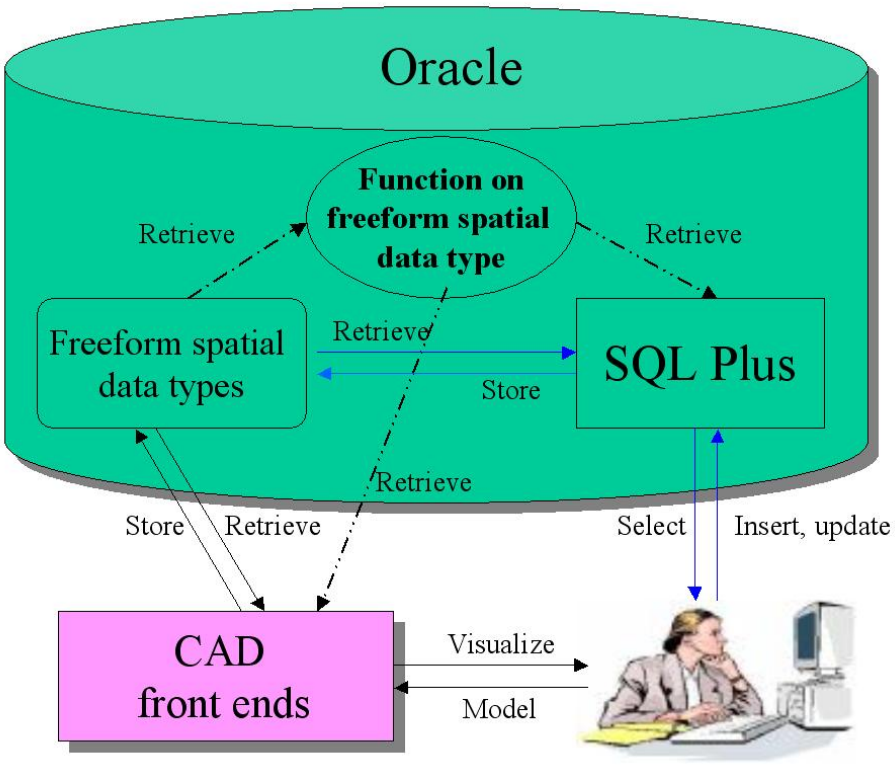

Figure 3.2: Overall architecture for conceptual design. Green blocks are DBMS modules; purple blocks are interface modules between user and DBMS; blue block represents user.

- Geometry transformation functions. For example, the function to rotate a freeform geometry.
- Conversion functions. For example, the function to convert a B-spline curve to a NURBS curve.
- Validation functions. For example, the function to check whether a record stores a valid freeform curve.

Chapter 4

Freeform data types

This chapter gives a detailed explanation of how freeform data types are implemented from the designed conceptual models in Chapter 3, and explains how freeform curves and surfaces can be stored in a spatial DBMS. Oracle Spatial is chosen to build the freeform data types on, instead of other spatial DBMSs, because:

- Oracle enables creating user-defined data types, which is convenient for creating new data types.
- I'm familiar with Oracle more than other spatial DBMSs.
- Oracle gives very detailed technical support. Nearly all aspects of Oracle and Oracle Spatial have been well documented, and these documentation can be easily accessed from Internet.

Section 4.1 discusses the possible implementation approaches, makes a choice, and then explains how to create freeform data types with this approach. Sections 4.2 gives the implementation of freeform curve data types. Section 4.3 gives the implementation of freeform surface data types. Section 4.4 shows some examples of how to manipulate the freeform data types.

4.1 Implementation approach

Conceptual models can be implemented in Oracle Spatial by different approaches. This section gives and compares the possible approaches, makes a choice, and then explains how this approach practically works in Oracle Spatial.

4.1.1 Possible approaches

From Section 3.2 we know that geometries are stored using SDO_Geometry by setting attribute values. Different kinds of geometries can be represented with this uniform data type, and their differences are reflected in the different combinations of attribute values. For example, the combination of SDO_ETYPE 2 and SDO_INTERPRETATION 1 indicate a linestring whose vertices are connected by straight line segments; while the combination of SDO_ETYPE 2 and SDO_INTERPRETATION 2 indicates a line string made up of a connected sequence of circular arcs. Storage of vertex coordinates and other relevant information should be according to certain rules determined by each combination. These rules can be found in [20].

Similarly, freeform geometries can be stored with SDO_GEOMETRY, too. The attribute SDO_GTYPE of SDO_GEOMETRY indicates the type of this geometry, for example: dl01 represents point, dl02 represents line/curve, dl03 represents polygon, etc. Currently number

00 to 07 have been used by some geometries, while 08 to 99 are still reserved for future use, which means they be can used by the freeform geometry types as well.

A possible implementation of the NURBS curve using SDO_Geometry is as follows:

```
SDO_GEOMETRY(
3009, --9 is GTYPE value for NURBS type
NULL, NULL,
SDO_ELEM_INFO_ARRAY(
   1,
   9, –ETYPE value for NURBS curve type
   d), – SDO_INTERPRETATION: degree value
SDO_ORDINATE_ARRAY(
   n, – number of control points
   m, – number of knots
   cpoints, – coordinates of n weighted control points (x,y,z,w)
   knots, – m knot vector values, for example (0, 0.2, 0.4, 0.6, 0.8, 1)
   s0, s1, – trimming values
));
```

The advantages of using SDO_Geometry to represent freeform data types include:

- Easy to integrate with existing supported geometry types. Because both freeform types and other geometry types are stored with the same data type, they can be stored in the same column of tables. This is sometimes rather convenient for data insertion, spatial queries, spatial index and other operations.
- Easy to implement. The required steps just include specifying type number (for example, 09 for NURBS curve, 10 for B-Spline curve, and etc.), and set up rules for SDO_ELEM_INFO.

However, there're also several significant disadvantages, including:

- The storage rules are quite complex. Unlike simple feature such as points, line strings which have very simple data structures, Bézier, B-spline and NURBS curves/surfaces have rather complex structures. Each record requires 3-5 parameters, and some of these parameters are an enumeration of point coordinates or real values. Manual input of freeform shapes is very difficult and confusion can be easily made in both insertion and retrieval, because it is hard to remember all the complex storage rules.
- Much redundant information is stored. SDO_Geometry requires that formats of data are specified in the parameter SDO_ELEM_INFO first by recording a pointer to the head of each data segment. This pointer information are redundant information, and the amounts are quite considerable.
- All the existing spatial functions on SDO_Geometry cannot be used for freeform data. This is mainly due to: 1. mathematical algorithms (intersection, convexHull, length, etc.) for simple features are completely different from the ones for freeform geometries; 2. all these functions are 2D functions, while we require 3D freeform geometries to be stored.

An alterative solution is to represent each geometry type with a separate data type in Oracle. For example, data type NURBSCurve for NURBS curve geometry, data type BsplineCurve for B-spline curve geometry, and so on. This approach is quite popular in other spatial DBMSs, including PostGIS, Informix, etc. The following example shows how to insert a

polygon into PostGIS:

```
INSERT INTO example_table(id, shape) VALUES
(1, GeometryFromText('POLYGON(-1 -1, -1 1, 1 1, 1 -1)', 128));
```

Note that "`POLYGON`" is the data type for polygon geometry. The advantages of creating several freeform data types include:

- Data structures are very clear. Types of freeform geometry are explicit, and the parameters can be accessed directly. This differs from SDO_GEOMETRY where certain analysis and traversal through SDO_ORDINATE_ARRAY is required.
- Few redundant information is stored. No reference information is required.
- Freeform data types can be extended and adapted to other applications easily. The structures of user-defined data types in Oracle are object-oriented. The data structures of freeform shapes and required parameters are various in different applications. Adapted data types can be easily inherited from existing prototypes, and functions on prototype types will be also operational for inherited types. This object-orient mechanism is rather important for practical purposes. More explanation about the user-defined data type in Oracle is given in the next section.

Disadvantages of this method include:

- Different kinds of geometries have to be stored in different table columns. This may lead to inconvenience for practical purposes, where a model normally consists of different geometry types, and all information of a model is preferably stored in the same column.
- Certain programming codes are required to create new data types. For example, Java or C or PL/SQL codes are required in Oracle Spatial for user-defined data types. Detailed explanations can be found in the next section.

From the comparison above we know that there are more advantages of the latter method: creating several freeform data types. Although different kinds of geometries cannot be stored in the same table column which leads to certain inconvenience, this problem can be solved by storing different kinds of geometries in different tables. Then a meta-data table which acts as an interface between DBMS and CAD/GIS applications, is used to record all the table names, spatial column names, dimension information, etc. Furthermore, the method of creating several freeform data types is recommended in the OGC specifications. Therefore this method is adopted for the implementation. The next section will zoom into the procedure of creating user-defined data types in Oracle.

4.1.2 Implementation by creating several user-defined data types

User-defined data types in Oracle use Oracle built-in data types and other user-defined data types as the building blocks of object types that model the structure and behavior of data in applications. Four kinds of data types: Object types, REFs, VARRAYs, and Nested tables can be created [18]. Here we are only interested in object types and VARRAYs.

Object types Object types are abstractions of real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A name, which identifies the object type uniquely within that schema
- Attributes, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity.

- Methods, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

Oracle's object type is **object-oriented**, which means new object types (subtype) can be created by inheriting from another object type (supertype). Derived subtypes inherit the features of the parent object type but extend the parent type definition. The specialized types can add new attributes or methods, or redefine methods inherited from the parent. The resulting type hierarchy provides a higher level of abstraction for managing the complexity of a model [23].

VARRAY An array is an ordered set of data elements. All elements of a given array are of the same data type. Each element has an index, which is a number corresponding to the position of the element in the array. The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called varrays. A maximum size is required when declaring the varray.

User-defined data types in Oracle can be declared using the SQL statement "*CREATE TYPE*". The implementation of the declaration can be:

- **PL/SQL** code. The implementation codes are written with PL/SQL language codes under another SQL statement "*CREATE TYPE BODY*".
- **Java** class. Each user-defined data type is mapped to a Java class, with each attribute of the user-defined type corresponding to a variable of the Java class, and each method of the user-defined type corresponding to a function of the Java class, as illustrated in Figure 4.1.
- **C** file. Each user-defined data type is mapped to a C language file, with each attribute of the user-defined type corresponding to a variable of the C file, and each method of the user-defined type corresponding to a function of the C file.

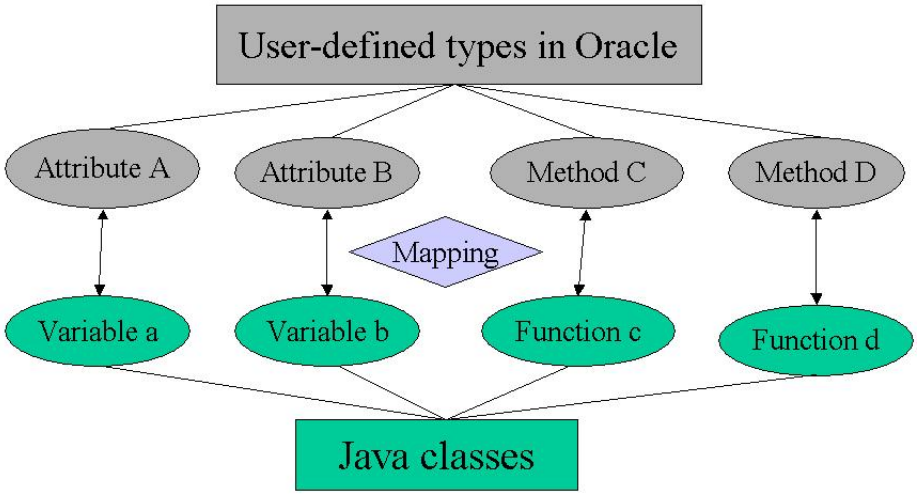

Figure 4.1: Mapping between user-defined data types and Java classes in Oracle

In this research, the Java approach is chosen because of the good support of Java by Oracle and familiarity of Java-SQL operations. PL/SQL is also well supported, but it is not

chosen because of the complexity and low efficiency of PL/SQL in geometry processing (see [27] p 32).

The overall procedure of creating user-defined data types (with the Java approach) can be divided into three main steps: a Java class which implements the conceptual model is created first; then this class will be loaded to the Oracle server; finally the type in Oracle is declared with "*CREATE TYPE*" statement.

Create Java class

There are two requirements for the Java class to be mapped with the user-defined SQL type.

The first requirement is that the Java class must implement either interface *java.sql.SQLData* or interface *oracle.sql.ORAData*. *java.sql.SQLData* supports methods (readSQL() and writeSQL()) to automatically convert Java types to standard SQL types, and the other way around. *oracle.sql.ORAData* extends *java.sql.SQLData* in the way that Oracle extended SQL types can also be converted to/from Java types.

The second requirement is that type mapping must follow several mapping rules, some of which are listed in the table below. The Java data types are different from SQL data types in Oracle, therefore mapping rules are required. For example, Java type *String* corresponds to SQL type *Varchar2*; Java type *int* corresponds to SQL type *number*; etc.

Java Type	**Oracle SQL Datatype**
boolean	NUMBER
byte	NUMBER
short	NUMBER
int	NUMBER
float	NUMBER
double	NUMBER
java.lang.String	CHAR VARCHAR2 LONG
java.sql.Struct	object types
java.sql.Ref	reference types
java.sql.Array	collection types
custom object classes implementing java.sql.SQLData	object types

Below is an example for a Java class implementing *java.sql.SQLData* interface.

```
public class GM_SplineCurve implements SQLData
         //SQL Data is the interface to mapping between SQL Data and Java Data
{
public String sql_type = "GM_SplineCurve";
public ARRAY cpt;  //mapping attribute "controlpoints"
public GM_KnotVector knots; //mapping attribute "knots"
public int degree;  //mapping attribute degree
public GM_SplineCurve()
{}
public void readSQL(SQLInput stream, String typeName) throws SQLException
    //output from Java to SQL
{
  sql_type = typeName;
  degree=stream.readInt();
  cpt=(ARRAY)stream.readArray();
  knots=(GM_KnotVector)stream.readObject();
}
```

```
public void writeSQL(SQLOutput stream) throws SQLException   //input from SQL to Java
{
stream.writeInt(degree);
stream.writeArray(cpt);
stream.writeObject(knots);
}
}
```

Load Java class

After a Java file is complied using *javac* compiler, the .class file is still in the local machine, and it needs to be loaded to the Oracle server using the *loadjava* tool. The *loadjava* tool is an operating system command-line utility that is able to uploads Java related files (.java, .class, .jar, etc.) into the database.

Here is the syntax:

```
loadjava {-user | -u} username/password[database] [-option_name
[-option_name] ...]  filename [filename ]...
```

An important option is *-resolve*. Specifying this option will compile a Java source file, resolve class dependencies, and then load it to the Oracle server if everything is all right. If this option is not specified, files are loaded but not compiled or resolved until runtime.

The following example shows how to load a Java .jar package *geometry.jar* to an Oracle server called *myserver* with username *usr* and password *pwd*.

```
loadjava -user usr/pwd@myserver geometry.jar
```

The following example shows how to load a Java source file GM_SplineCurve.java to the Oracle server.

```
loadjava -user usr/pwd@myserver -resolve GM_SplineCurve.java
```

Create type

After Java class files are loaded in Oracle server, finally user-defined data type can be declared using the *CREATE TYPE* statement. The *CREATE TYPE* statement specifies the name of the user-defined type, its attributes, methods and other properties. Figure 4.2 and Figure 4.3 show a simplified syntax of *CREATE TYPE* statement.

Following are explanations of some clauses of Figure 4.2 and Figure 4.3. Clauses that are not relevant to this research will not be explained, but they can be obtained from [22].

type_name Specify the name of an object type, or a varray type.

UNDER Specify *UNDER supertype* to create a subtype of an existing object type. The subtype inherits the attributes and methods of its supertype. It must either override some of those attributes/methods, or add new attributes/methods to distinguish it from the supertype.

sqlj_object_type Specify information of the mapped Java class.

attribute Specify the name of an attribute for this user-defined type. If this is a subtype, then the attribute name cannot be the same as any attribute or method declared in the supertype chain.

datatype Specify data type of this attribute. The allowed data type includes Oracle built-in datatypes (VARCHAR2, NUMBER, DATE, etc.) or another user-defined type.

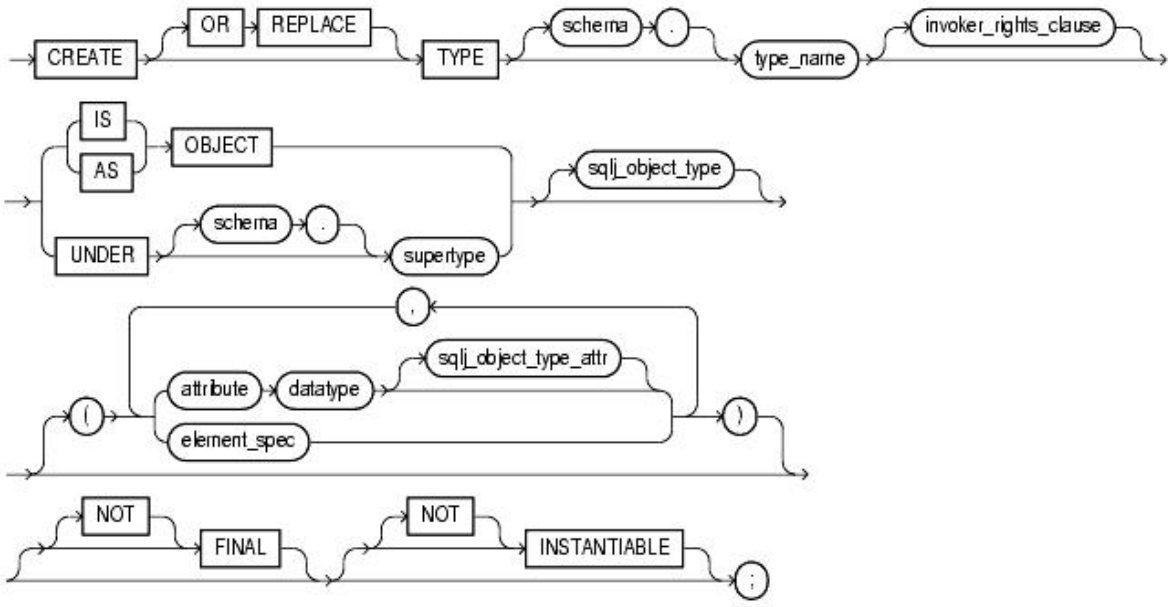

Figure 4.2: Syntax of *CREATE TYPE*. Clauses in main lines are required, while those in branches are optional; strings in rectangles are constant, while those in rounded rectangles are user-specified. [22]

Figure 4.3: sqlj_object_type [22]

NOT FINAL Specify *FINAL* if no further subtypes can be created for this type; specify *NOT FINAL* if further subtypes can be created under this type.

SQLData, OraData Specify the mapping strategy. Use SQLData to map standard SQL data types; use OraData to map Oracle extended data types.

Following is an example of creating a user-defined data type *GM_SplineCurve* which maps Java class *GM_SplineCurve.class*. As shown in Figure 3.1, *GM_SplineCurve* is the super type of *GM_BezierCurve*, *GM_BSplineCurve* and *GM_NURBSCurve*.

```
SQL> CREATE or REPLACE type GM_SplineCurve as object
external name 'GM_SplineCurve' language java using SQLData(
degree number external name 'degree',
controlPoints GM_PointArray external name 'cpt',
knots GM_KnotVector external name 'knots'
);

SQL> /

Type created.
```

4.2 Freeform curve types

Following the procedures explained in the previous section, the four freeform curve types in the conceptual model are implemented. They are GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve and GM_NURBSCurve. GM_SplineCurve is the super type of the other three. Although GM_SplineCurve itself is just an abstract geometry class, it is created for several important reasons:

1. Because of the object-oriented structure of Oracle objects, common attributes and methods of GM_BezierCurve, GM_BSplineCurve and GM_NURBSCurve can be specified as GM_SplineCurve's attributes and methods, which will be easy for creation and modification;

2. GM_SplineCurve package is mentioned in OGC Abstract specifications.

Following are the descriptions of implemented freeform data types in Oracle.

GM_SplineCurve

```
SQL> desc GM_SplineCurve
GM_SplineCurve is NOT FINAL
Name                                      Type
----------------------------------------  --------------------
DEGREE                                    NUMBER
CONTROLPOINTS                             GM_POINTARRAY
KNOTS                                     GM_KNOTVECTOR
```

Explanations for the parameters:

DEGREE The degree value of this curve.

CONTROLPOINTS x,y and z coordinates of the control points.

KNOTS Knot vector of this curve.

GM_BezierCurve

```
SQL> desc GM_BezierCurve
GM_BezierCurve extends GM_SPLINECURVE
Name                                      Type
----------------------------------------  ----------------------------
DEGREE                                    NUMBER
CONTROLPOINTS                             GM_POINTARRAY
KNOTS                                     GM_KNOTVECTOR
```

Explanations for the parameters:

DEGREE The degree value of this Bézier curve.

CONTROLPOINTS x,y and z coordinates of the control points.

KNOTS Knot vector should be specified 'NULL' for Bézier curve.

GM_BSplineCurve

```
SQL> desc GM_BSplineCurve
GM_BSplineCurve extends GM_SPLINECURVE
Name                                      Type
----------------------------------------- ----------------------------
DEGREE                                    NUMBER
CONTROLPOINTS                             GM_POINTARRAY
KNOTS                                     GM_KNOTVECTOR
ISPOLYNOMIAL                              NUMBER
```

Explanations for the parameters:

DEGREE The degree value of this B-spline curve.

CONTROLPOINTS x,y and z coordinates of the control points.

KNOTS Knot vector of this B-spline curve.

ISPOLYNOMIAL If 'True', then this is a polynomial B-spline, otherwise this is a normal B-spline curve. The major difference between the polynomial B-splines and normal B-splines is that polynomial B-splines pass through their control points.

GM_NURBSCurve

```
SQL> desc GM_NURBSCurve
GM_NURBSCurve extends GM_SPLINECURVE
Name                                      Type
----------------------------------------- ----------------------------
DEGREE                                    NUMBER
CONTROLPOINTS                             GM_POINTARRAY
KNOTS                                     GM_KNOTVECTOR
WEIGHTS                                   GM_WEIGHTARRAY
TRIM                                      GM_TRIM
```

Explanations for the parameters:

DEGREE The degree value of this NURBS curve.

CONTROLPOINTS x,y and z coordinates of the control points.

KNOTS Knot vector of this NURBS curve.

WEIGHTS Weight values of the control points. The length of WEIGHTS must be equal to 1/3 of the length of CONTROLPOINTS.

TRIM Trimming values of this NURBS curve. Specify 'NULL' if no trimming.

Besides the freeform curves data types, several supplementary data types are also created according to the conceptual model. The freeform data types require some parameters such as knot vector, weight values, etc. Different from the simple attributes such as degree which can be represented with existing data types in Oracle, these complex parameters can only be represented with separate user-defined data types. GM_PointArray, GM_WeightArray, GM_Trim and VECTOR are Varray types, which are enumerations of basic data type (number) in Oracle. GM_KnotVector is an object type with two attributes: knots and weights. Following are description of these types:

GM_PointArray:

```
GM_PointArray VARRAY(1048576) OF NUMBER

VECTOR VARRAY(1048576) OF NUMBER
```

GM_WeightArray:

```
GM_WeightArray VARRAY(1048576) OF NUMBER
```

GM_Trim:

```
GM_Trim VARRAY(2) OF NUMBER
```

VECTOR:

GM_KnotVector:

Name	Type
KNOTS	VECTOR
WEIGHTS	GM_WEIGHTARRAY

Explanations for the parameters of **GM_KnotVector**:

KNOTS Non-decreasing knot values.

WEIGHTS The value of the averaging weight used for this knot of the spline.

4.3 Freeform surface type

Only one freeform surface type: GM_NURBSSurface is created, instead of implementing all of Bézier surface, B-spline surface and NURBS surface. This is mainly because:

- B-spline surfaces can be represented with GM_NURBSSurface by specifying *NULL* value to *weights* attribute; Bézier surfaces can be represented with GM_NURBSSurface by specifying *NULL* value to *weights* attribute and *knot* attribute.
- NURBS surfaces are more popular in CAD/GIS applications than B-spline surfaces or Bézier surfaces.
- The implementation of Bézier surface, B-spline surface and NURBS surface will be very similar. Therefore as a research topic, the most complex and generalized surface geometry: NURBS surface, is chosen to be implemented, instead of repeating the similar procedure for all three.

GM_NURBSSurface:

Name	Type
NUMU	NUMBER
NUMV	NUMBER
DEGREEU	NUMBER
DEGREEV	NUMBER
CONTROLPOINTS	GM_POINTARRAY
WEIGHTS	GM_WEIGHTARRAY
KNOTU	GM_KNOTVECTOR
KNOTV	GM_KNOTVECTOR
TRIMU	GM_TRIM
TRIMV	GM_TRIM

Explanations for the parameters of **GM_NURBSSurface**.

NUMU The number of control points in u direction.

NUMV The number of control points in v direction.

DEGREEU Degree value in u direction.

DEGREEV Degree value in v direction.

CONTROLPOINTS x,y and z coordinate of the control points.

WEIGHTS Weight values of the control points. The length of WEIGHTS must be equal to 1/3 of the length of CONTROLPOINTS.

KNOTU Knot vector in u direction.

KNOTV Knot vector in v direction.

TRIMU Trimming values in u direction.

TRIMV Trimming values in v direction.

4.4 Examples

The following examples show basic SQL operations with the created freeform type: GM_BSplineCurve. A table *test* with a *GM_BSplineCurve* will be created, then 2 B-spline curves will be inserted, and finally these curves are queried.

The following SQL statement creates a table *test* with 2 columns. The column *col* is of the freeform data type: *GM_BSplineCurve*.

```
SQL> create table test(id number,col GM_BSplineCurve);

Table created.
```

The following SQL statement inserts a normal (not polynomial) B-spline curve (as in Figure 4.4) with parameters:

- degree 2;
- 5 control points;
- equally spaced (uniform) knot vector;

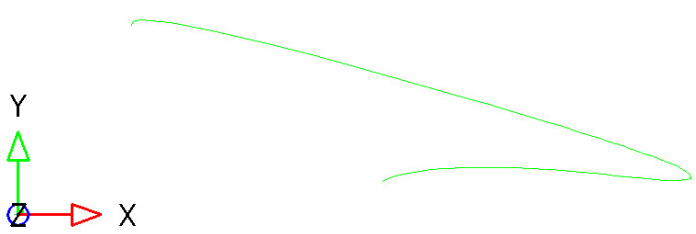

Figure 4.4: A B-spline curve

```
SQL> insert into test
values(2,GM_BSplineCurve(2,GM_PointArray(135,225,346,127,256,336,945,20,30,504
,70,698,434,40,4),GM_KnotVector(Vector(-0.5,0,0.5,1.5,2,2.5,3,3.5),NULL),NULL));

1 row created.
```

The following SQL statement inserts another B-spline curve (as in Figure 4.5) into table *test*.

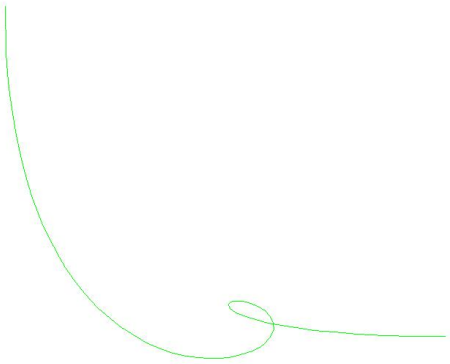

Figure 4.5: A B-spline curve

```
SQL> insert into test
values(3,GM_BSplineCurve(4,GM_PointArray(1,2,10,1,2,3,9,2,3,5,7,6,4,4,4,9,0,4)
,GM_KnotVector(Vector(0,0,0,0,0,0.5,1,1,1,1,1),NULL),NULL));

1 row created.
```

The following SQL statement selects all B-spline curve(s) from table *test*.

```
SQL> select * from test;

        ID
----------
COL(DEGREE, CONTROLPOINTS, KNOTS(KNOTS, WEIGHTS), ISPOLYNOMIAL)
--------------------------------------------------------------------------------
         2
GM_BSPLINECURVE(2, GM_POINTARRAY(135, 225, 346, 127, 256, 336, 945,
20, 30, 504, 70, 698, 434, 40, 4),
GM_KNOTVECTOR(VECTOR(-0.5,0,0.5,1.5,2,2.5,3,3.5), NULL), N ULL)

         3
GM_BSPLINECURVE(4, GM_POINTARRAY(1, 2, 10, 1, 2, 3, 9, 2, 3, 5, 7,
6, 4, 4, 4, 9 , 0, 4), GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, 0, .5, 1, 1,
1, 1, 1), NULL), NULL)
```

The following SQL statement selects all B-spline curve(s) with degree 4 from table *test*.

```
SQL> select * from test a where a.col.degree=4;

        ID
```

```
----------
COL(DEGREE, CONTROLPOINTS, KNOTS(KNOTS, WEIGHTS), ISPOLYNOMIAL)
--------------------------------------------------------------------------
         3
GM_BSPLINECURVE(4, GM_POINTARRAY(1, 2, 10, 1, 2, 3, 9, 2, 3, 5, 7,
6, 4, 4, 4, 9 , 0, 4), GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, 0, .5, 1, 1,
1, 1, 1), NULL), NULL)
```

The following SQL statement selects the knot vector of the B-spline curve with row id 3.

```
SQL> select a.col.knots from test a where a.id=3;

COL.KNOTS(KNOTS, WEIGHTS)
--------------------------------------------------------------------------
GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, 0, .5, 1, 1, 1, 1, 1), NULL)
```

More examples can be found in Chapter 7.

The following examples show basic SQL operations with the created freeform surface type: GM_NURBSSurface. A table *test2* will be created, then one NURBS surface is inserted, and this surface is finally queried.

The following SQL statement creates a table *test2* with 2 columns. The column *col* is of the freeform surface type: GM_NURBSSurface.

```
SQL> create table test2(id number, col GM_NURBSSurface);

Table created.
```

The following SQL statement inserts a NURBS surface (as in Figure 4.6) with parameter:

- 3 control points in u direction; 5 control points in v direction; 3*5 control points in total.
- Degree 2 in u direction; degree 2 in v direction.
- Knot vector in both u and v directions.
- 15 weight values.
- No trimming value.

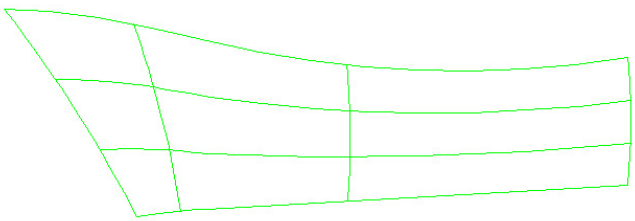

Figure 4.6: A NURBS surface

```
SQL> insert into test2 values(1, GM_NURBSSURFACE(3, 5, 2, 2,
GM_POINTARRAY(5.1469375, 1.83903125, 1. 744375, 5.14721875,
```

```
2.24558333, 1.722375, 5.1475, 2.421, 1.98245833, 5.40390625,
1.83903125,1.744375 , 5.371125, 2.2166875, 1.759125, 5.33834375,
2.421, 1.9854375, 5.660875,1.83903125, 1.744375, 5.5950 3125,
2.18780208, 1.795875, 5.5291875, 2.421, 1.98841667, 5.910875,
1.83903125, 1.744375, 5.81545833 , 2.15890625, 1.832625,
5.72003125, 2.421, 1.99139583, 5.910875, 1.83903125, 1.994375,
5.910875, 2.1 3002083, 1.994375, 5.910875, 2.421, 1.994375),
GM_WEIGHTARRAY(1, 1, 1, 1, 1, 1, 1, 1, 1, .707106781, .853553391,
1, 1, 1, 1), GM_KNOTVECTOR(VECTOR(0, 0, 0, 1, 1, 1), NULL),
GM_KNOTVECTOR(VECTOR(0, 0, 0, .5, .5, 1, 1, 1), NULL),NULL,NULL));

1 row created.
```

The following SQL statement selects the number of control points in u directions of all NURBS surface(s) from table *test2*.

```
SQL> select a.col.NumU from test2 a;

  COL.NUMU
----------
         3
```

The following SQL statement selects the NURBS surface(s) with degree 2 in v directions from table *test2*.

```
SQL> select a.id from test2 a where a.col.degreeV=2;

        ID
----------
         1
```

More examples can be found in Chapter 7.

4.5 Concluding remarks

In this chapter, the freeform data types are implemented by creating user-defined data types in Oracle. Parameters required by a freeform geometry are represented with the attributes of a user-defined date type. An alternative approach is to store freeform geometries with SDO_Geometry, however, this approach is not adopted mainly because of the complex storage rules and storage redundancy.

User-defined types in Oracle can be declared using SQL statement: "*CREATE TYPE*". The implementation of the declaration can be written in PL/SQL, Java or C. In Java approach, each user-defined data type maps a Java class, with each attribute of the data type mapping a variable of the Java class, and each method of the data type mapping a function of the Java class.

All the freeform data types, which are defined in the conceptual model in Chapter 3, have been implemented in Oracle. They are GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface, and some supplementary data types.

Chapter 5

Functions on freeform data types

A data type cannot be "managed" by a spatial DBMS without functions on this data type. Based on the created freeform data types in the previous chapter, this chapter will explain the implementation of functions on freeform data types.

Section 5.1 introduces the approaches of how conceptual models can be implemented. Sections 5.2-5.6 present the implemented functions by categories. Section 5.7 gives some examples of how these functions work in practice.

5.1 Implementation approach

As was explained in Section 2.2.2, two kinds of functions can be distinguished in Oracle:

Standalone functions which are called by name. The parameters and return data can be any data type. A standalone function is called as $function(p1, ..., p_n)$.

Methods which are called along with its object type. These functions are created along with a certain object type, and they are only operational within this object type. The parameters and return data can be any data type, too. A method is called as $dataType.method1$ $(p1, ...p_n)$.

The conceptual functions can be implemented with either standalone functions or methods. The main difference is only in the way of usage. Generally speaking, there is no strict rule of whether a function should be implemented as standalone function or method. They don't conflict with each other, and a function can be both implemented as standalone function and as method. However, considering from practical usage, it is more appropriate to implement functions which take only one parameter as methods. This is because the name of a standalone function should be unique in the DBMS environment, and hence if a conceptual function can be applied to more than one data type, the implemented standalone functions have to use different names. For example, conceptual function 'isClosed()' need to be implemented for all of GM_BezierCurve, GM_BSplineCurve and GM_NURBSCurve. If 'isClosed()' is implemented as standalone function, there have to be three different functions such as 'isClosed(GM_BezierCurve)', 'isClosed1(GM_BSplineCurve)' and 'isClosed2(GM_NURBSCurve)', which will cause significant inconvenience to users. If 'isClosed()' is implemented as a method, then it can be invoked as 'GM_BezierCurve.isClosed()', 'GM_BSplineCurve.isClosed()' and 'GM_NURBSCurve.isClosed()'.

However, implementation for these two kinds of functions is somewhat different. This is explained below:

5.1.1 Standalone functions

Standalone functions are declared using the *CREATE FUNCTION* statement, which has a similar procedure as *CREATE TYPE*. The implementation of the declaration can also be

PL/SQL codes, Java class or C file. To be consistent with data type implementation, the Java approach will also be used to create standalone functions. Figure 5.1 shows a simplified syntax of *CREATE FUNCTION*.

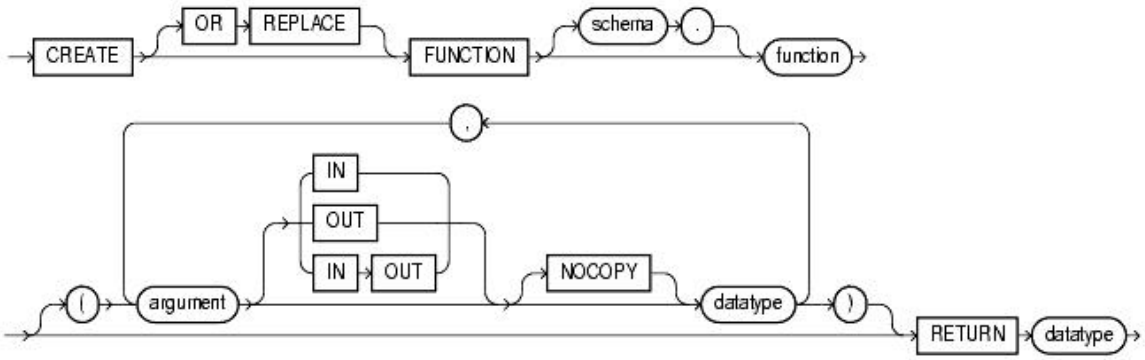

Figure 5.1: Syntax of *CREATE FUNCTION*

Following is an example of creating a standalone function: *Translation*, which maps the Java function *translation* within *functions.class*. *Translation* takes 4 parameters: the column of *GM_SplineCurve* data type, and the translation distances in x, y and z directions.

```
SQL> CREATE FUNCTION translation
(s IN GM_SplineCurve,x IN number,y IN number,z IN number)
RETURN GM_SplineCurve AS LANGUAGE JAVA NAME
'functions.translation(GM_SplineCurve,float,float,float)
return GM_SplineCurve';

Function created.
```

5.1.2 Methods

Methods are declared along with object types. As mentioned in the previous chapter, the object types are declared using *CREATE TYPE*. The *MEMBER FUNCTION* parameter of *CREATE TYPE* is the part to declare methods of this type. Each member function maps to a function within a Java class, as illustrated in Figure 4.1.

Following is an example of creating a user-defined type: *GM_BSplineCurve*, under an existing user-defined type: *GM_SplineCurve*. *GM_BSplineCurve* has one more attribute: *isPolynomial*, and two more methods *toNURBS* and *validation* than *GM_SplineCurve*. Note that the method *validation* in this user-defined data type maps the function *validation* within *GM_BSplineCurve.class*:

```
SQL> CREATE OR REPLACE TYPE GM_BSplineCurve under GM_SplineCurve
external name 'GM_BSplineCurve' language java using SQLDATA(
isPolynomial number external name 'isPolynomial',
member function toNURBS return GM_NURBSCurve external name
'toNURBS() return GM_NURBSCurve',
member function validation return number external name 'validation()
return int')

Type created.
```

According to their functionalities, we categorize all the implemented functions in five groups: access functions, geometry relationship functions, geometric transformation functions, conversion functions and validation functions.

5.2 Access functions

The access functions return geometric information of a freeform curve or surface.

N

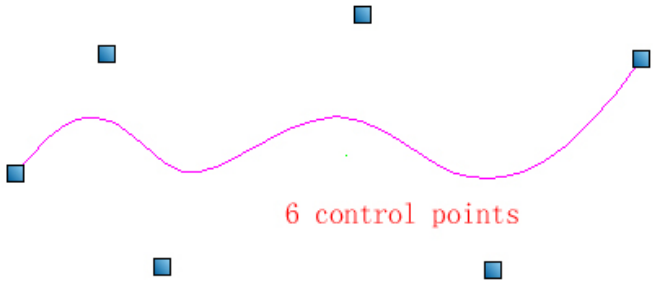

Figure 5.2: Spatial function: N()

This function returns the number of control points or the number of knots.

Type	Method
Format	N() RETURN NUMBER
Parameters	None
Returns	The number of control points; the number of knots.
Member of	GM_KnotVecotor, GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.N() FROM table a;

ISCLOSED

This function returns whether this curve is closed.

Type	Method
Format	ISCLOSED() RETURN NUMBER
Parameters	None
Returns	Whether this curve is closed.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve
Example	SELECT a.col.ISCLOSED() FROM table a;

CENTROID

This function returns the approximated centroid of a curve/surface with the center point of its control polygon, which is the polygon defined by the control points.

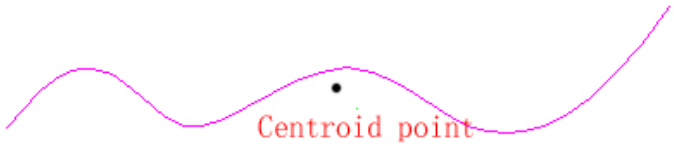

Figure 5.3: Spatial function: Centroid()

Type	Method
Format	CENTROID() RETURN SDO_Geometry
Parameters	None
Returns	The center point of this curve/surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.CENTROID() FROM table a;

CPOLYGON

This function returns the control polygon/polyhedron of this curve/surface.

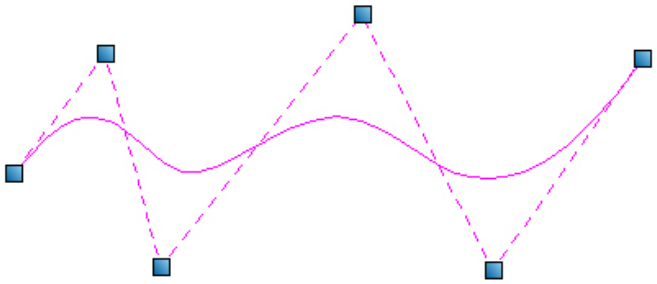

Figure 5.4: Spatial function: CPolygon()

Type	Method
Format	CPOLYGON() RETURN SDO_Geometry
Parameters	None
Returns	The control polygon/polyhedron of this curve/surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.CPOLYGON() FROM table a;

CONVEXHULL

This function returns the convex hull of this curve/surface. The convex hull of a spline curve/surface is defined by its control points [3].

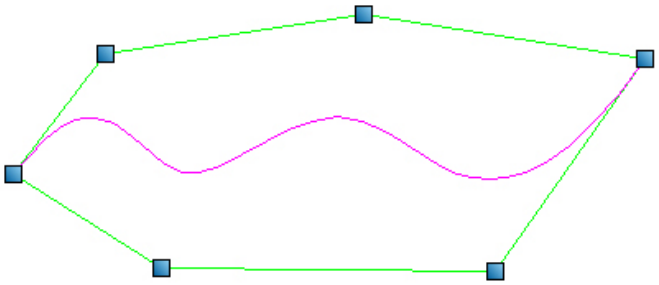

Figure 5.5: Spatial function: ConvexHull()

Type	Method
Format	CONVEXHULL() RETURN SDO_Geometry
Parameters	None
Returns	The convex hull of this curve/surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.CONVEXHULL() FROM table a;

BOUNDING_BOX

This function returns a bounding box of this curve/surface. This bounding box is defined by the maximum and minimum x, y and z coordinates of all control points.

Type	Method
Format	BOUNDING_BOX() RETURN SDO_Geometry
Parameters	None
Returns	A bounding box of this curve/surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.BOUNDING_BOX() FROM table a;

5.3 Geometry relationship functions

Geometry relationship functions return a relationship between freeform curves, freeform surfaces and simple geometries in SDO_Geometry.

DISTANCE

This function returns the approximated distance between two freeform curves. The distance between the centroids of two freeform curves is calculated and returned. This differs from distance functions of points, linestrings and polygons that return the shortest distance between two geometries, because shortest distance computations between freeform curves are rather complex, and accurate geometry computation is better done in CAD/GIS front ends, not at DBMS level.

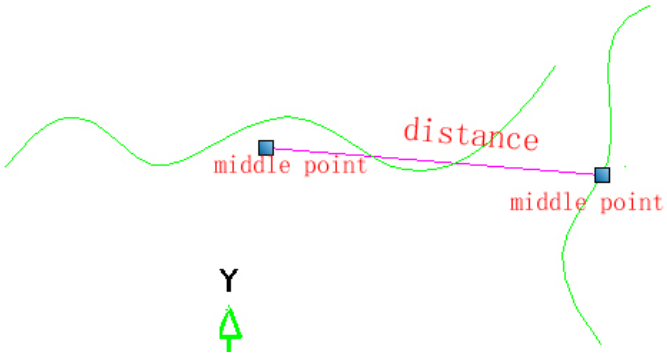

Figure 5.6: Spatial function: Distance()

Type	Standalone function
Format	DISTANCE(G1 GM_SplineCurve, G2 GM_SplineCurve) RETURN NUMBER
Parameters	G1: The first Bézier/B-spline/NURBS curve. G2: The second Bézier/B-spline/NURBS curve.
Returns	The distance between two freeform curves.
Example	SELECT DISTANCE(a.col, b.col) FROM table1 a, table2 b;

DISTANCE_C2S

This function returns the distance between a freeform curve and a freeform surface. Centroids of the curve and surface are used to compute the distance.

Type	Standalone function
Format	DISTANCE_C2S (G1 GM_SplineCurve, G2 GM_NURBSSurface) RETURN NUMBER
Parameters	G1: A Bézier/B-spline/NURBS curve. G2: A NURBS surface.
Returns	The distance between a Beizer/B-spline/NURBS curve and a NURBS surface.
Example	SELECT DISTANCE_C2S(a.col, b.col) FROM table1 a, table2 b;

DISTANCE_C2G

This function returns the distance between the centroid of a freeform curve and the centroid of a simple geometry defined with SDO_Geometry. The centroid of the simple geometry in SDO_Geometry can be obtained with Oracle Spatial built in function: *SDO_GEOM.SDO_CENTROID*.

Type	Standalone function
Format	DISTANCE_C2G (G1 GM_SplineCurve, G2 SDO_Geometry) RETURN NUMBER
Parameters	G1: A Bézier/B-spline/NURBS curve. G2: A simple geometry.
Returns	The distance between a Bezier/B-spline/NURBS curve and a simple geometry in SDO_Geometry.
Example	SELECT DISTANCE_C2G(a.col, b.col) FROM table1 a, table2 b;

ANYINTERSECT

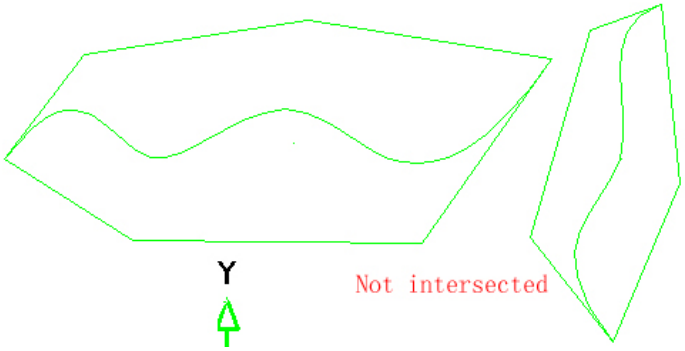

Figure 5.7: Spatial function: AnyIntersect()

This function returns whether two freeform curve may intersect each other. This is actually done by checking the intersection between their convex hulls, which are approximation polygons/polyhedrons of freeform curves.

Type	Standalone function
Format	ANYINTERSECT (G1 GM_SplineCurve, G2 GM_SplineCurve) RETURN NUMBER
Parameters	G1: The first Bézier/B-spline/NURBS curve. G2: The second Bézier/B-spline/NURBS curve.
Returns	Whether two freeform curves may intersect with each other.
Example	SELECT ANYINTERSECT(a.col, b.col) FROM table1 a, table2 b;

5.4 Geometry transformation functions

Geometry processing functions apply basic geometric transformations such as translation, rotation and scaling to freeform curve and surface types.

TRANSLATION
This standalone function returns the translated freeform curve by (X,Y,Z).

Type	Standalone function
Format	TRANSLATION (G GM_SplineCurve, X NUMBER, Y NUMBER, Z NUMBER) RETURN GM_SplineCurve
Parameters	G: The Bézier/B-spline/NURBS curve to be translated. X: The translation distance in x direction. Y: The translation distance in y direction. Z: The translation distance in z direction.
Returns	The translated Bézier/B-spline/NURBS curve.
Example	SELECT TRANSLATION(a.col, 1, 2, 3) FROM table a;

TRANSLATION
This method returns the translated freeform curve/surface by (X,Y,Z).

Type	Method
Format	TRANSLATION (X NUMBER, Y NUMBER, Z NUMBER) RETURN GM_SplineCurve/GM_BezierCurve/ GM_NURBSCurve/GM_NURBSSurface
Parameters	X: The translation distance in x direction. Y: The translation distance in y direction. Z: The translation distance in z direction.
Returns	The translated Bézier/B-spline/NURBS curve or NURBS surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.TRANSLATION(1, 2, 3) FROM table a;

TRANSLATION_S
Standalone function to translate a GM_NURBSSurface, analogous to **TRANSLATION**, which translates freeform curves.

SCALE
This standalone function returns the scaled freeform curve by (x,y,z).

Type	Standalone function
Format	SCALE (G GM_SplineCurve, X NUMBER, Y NUMBER, Z NUMBER) RETURN GM_SplineCurve
Parameters	G: The Bézier/B-spline/NURBS curve to be scaled. X: The scale extent in x direction. Y: The scale extent in y direction. Z: The scale extent in z direction.
Returns	The scaled Bézier/B-spline/NURBS curve.
Example	SELECT SCALE(a.col, 1, 2, 3) FROM table a;

SCALE
This method returns the scaled freeform curve/surface by (x,y,z).

Type	Method
Format	SCALE (X NUMBER, Y NUMBER, Z NUMBER) RETURN GM_SplineCurve / GM_BezierCurve / GM_BSplineCurve / GM_NURBSCurve / GM_NURBSSurface
Parameters	X: The scale extent in x direction. Y: The scale extent in y direction. Z: The scale extent in z direction.
Returns	The scaled Bézier/B-spline/NURBS curve or NURBS surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.SCALE(1, 2, 3) FROM table a;

SCALE2_S
Standalone function to scale GM_NURBSSurface, analogous to **SCALE**.

ROTATEX
This standalone function returns the rotated freeform curve along x axis by A.

Type	Standalone function
Format	ROTATEX (G GM_SplineCurve, A NUMBER) RETURN GM_SplineCurve
Parameters	G: The Bézier/B-spline/NURBS curve to be rotated. A: The angle to be rotated.
Returns	The rotated Bézier/B-spline/NURBS curve along X axis by A.
Example	SELECT ROTATEX(a.col, 3.1415926) FROM table a;

ROTATEX
This method returns the rotated freeform curve/surface along x axis by A.

Type	Method
Format	ROTATEX (A NUMBER) RETURN GM_SplineCurve / GM_BezierCurve / GM_BSplineCurve / GM_NURBSCurve / GM_NURBSSurface
Parameters	A: The angle to be rotated.
Returns	The rotated Bézier/B-spline/NURBS curve or NURBS surface.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, GM_NURBSCurve, GM_NURBSSurface
Example	SELECT a.col.ROTATEX(3.1415926) FROM table a;

ROTATEX_S
Standalone function to rotate GM_NURBSSurface along X axis, analogous to **ROTATEX**.

ROTATEY
Standalone function and method to rotate GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, and GM_NURBSCurve along Y axis, analogous to **ROTATEX**.

ROTATEY_S
Standalone function to rotate GM_NURBSSurface along Y axis, analogous to **ROTATEY**.

ROTATEZ
Standalone function and method to rotate GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve, and GM_NURBSCurve along Z axis, analogous to **ROTATEX**.

ROTATEZ_S
Standalone function to rotate GM_NURBSSurface along Z axis, analogous to **ROTATEZ**.

REGULATION
This functions returns a 'stretched' knot vector, in which the minimum knot value is 0 and the maximum knot value is 1, of the original knot vector. The original knots are linearly recomputed to a new value within 0 and 1.

Type	Method
Format	Regulation() RETURN GM_KnotVector
Parameters	None.
Returns	The regulated knot vector. All knot values are within 0 to 1.
Member of	GM_KnotVector
Example	SELECT a.col.knots.REGULATION() FROM table a;

5.5 Conversion functions

The conversion functions make conversions between freeform curves.

TOBSPLINE
This function converts a Bezier curve into B-Spline form. This is done by giving the Bezier curve a *knot* parameter, which starts with *degree*+1 zeros and ends with *degree*+1 ones.

Type	Method
Format	TOBSPLINE() RETURN GM_BSplineCurve
Parameters	None
Returns	The BSpline form of a Bézier curve.
Member of	GM_BezierCurve
Example	SELECT a.col.TOBSPLINE() FROM table a;

TONURBS

This function converts a B-spline curve into NURBS form. This is done by giving the B-spline curve a *weight* parameter, which has a length equal to 1/3 of the *controlPoint* and all elements with equal value one.

Type	Method
Format	TONURBS() RETURN GM_NURBSCurve
Parameters	None
Returns	The NURBS form of a BSpline curve.
Member of	GM_BSplineCurve
Example	SELECT a.col.TONURBS() FROM table a;

5.6 Validation functions

Based on the conceptual validation function in Section 3.2, the validation functions in Oracle are all implemented as method of each freeform types. For example, validation of NURBS curve is dones by *NURBSCurve_COLUMN.validation()*, validation of NURBS surface is done by *NURBSSurface_COLUMN.validation()*, etc. The validation functions are implemented as methods, and they return -1 for a storage invalid geometry, return 0 for a geometry invalid geometry, and return 1 when found a valid freeform geometry. The storage validity and geometry validity rules are defined in Section 3.2.

VALIDATION

Type	Method
Format	VALIDATION() RETURN NUMBER
Parameters	None
Returns	The validity information of this geometry.
Member of	GM_SplineCurve, GM_BezierCurve, GM_BSplineCurve GM_NURBSCurve, GM_NURBSSurface, GM_KnotVector
Example	SELECT a.col.VALIDATION() FROM table a;

5.7 Examples

The following SQL statements show how to use the functions explained in this chapter on freeform data types.

Let's create a table with GM_NURBSCurve column first.

```
SQL> create table test(id number,col GM_NURBSCurve);

Table created.
```

Insert two NURBS curves; they actually have the same shape as the B-spline curves in Figure 4.4 and Figure 4.5.

```
SQL> insert into test
values(3,GM_NURBSCurve(4,GM_PointArray(1,2,10,1,2,3,9,2,3,5,7,6,4,4,4,9,0,4),G
M_KnotVector(Vector(0,0,0,0,0,0.5,1,1,1,1,1,1),NULL),GM_WeightArray(1,1,1,1,1,1),
NULL));

1 row created.
```

```
SQL> insert into test
values(5,GM_NURBSCurve(2,GM_PointArray(135,225,346,127,256,336,945,20,30,504,7
0,698,434,40,4),GM_KnotVector(Vector(-0.5,0,0.5,1,2,3,4,5),NULL),GM_WeightArray
(1,1,1,1,1),NULL));

1 row created.
```

Check their validity.

```
SQL> select a.col.validation() from test a;

A.COL.VALIDATION()
------------------
                 1
                 1
```

This means they are all valid NURBS curve.
The following SQL statement returns the centroid of the first NURBS curve.

```
SQL>  select a.col.centroid() from test a where a.id=3;

A.COL.CENTOID()(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z),
SDO_ELEM_INFO, SDO_ORDI
--------------------------------------------------------------------------------
SDO_GEOMETRY(3001, NULL, SDO_POINT_TYPE(4.4212963, 2.76851852,
5.36111111), NULL , NULL)
```

The following SQL statement returns the distance between the two NURBS curves.

```
SQL> select distance(a.col, b.col) from test a, test b where
a.id=3 and b.id=5;

DISTANCE(A.COL,B.COL)
---------------------
           558.859116
```

Translate the first NURBS curve by (2,2,2):

```
SQL> update test a set
a.col.controlpoints=translation(a.col,2,2,2).controlpoints where
a.id=3;

1 row updated.

SQL> select * from test where id=3;

        ID
----------
COL(DEGREE, CONTROLPOINTS, KNOTS(KNOTS, WEIGHTS), WEIGHTS, TRIM)
--------------------------------------------------------------------------------
         3
GM_NURBSCURVE(4, GM_POINTARRAY(3, 4, 12, 3, 4, 5, 11, 4, 5, 7, 9,
8, 6, 6, 6, 11 , 2, 6), GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, 0, .5, 1,
1, 1, 1, 1), NULL), GM_WEIGH TARRAY(1, 1, 1, 1, 1, 1), NULL)
```

Regulate all NURBS curves' knots.

```
SQL> update test a set a.col.knots=a.col.knots.regulation();

2 rows updated.

SQL> select a.col.knots from test a;

COL.KNOTS(KNOTS, WEIGHTS)
--------------------------------------------------------------------------------
GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, 0, .5, 1, 1, 1, 1, 1), NULL)
GM_KNOTVECTOR(VECTOR(0, .090909091, .181818182, .272727273,
.454545455, .6363636 36, .818181818, 1), NULL)
```

The following example shows how a validation function checks the storage validity of some GM_NURBSCurve rows.

```
SQL> insert into test
values(4,GM_NURBSCurve(6,Null,Null,Null,NULL));

1 row created.

SQL> insert into test
values(1,GM_NURBSCurve(4,GM_PointArray(1,2,3,1,2,3,9,2,3,5,7,6,4,4,4,9,0,4),
GM_KnotVector(Vector(0,1),NULL),GM_WeightArray(1,1,1,1),NULL));

1 row created.

SQL> select id, a.col.validation() from test a;

        ID A.COL.VALIDATION()
---------- ------------------
         4                 -1
         3                 -1
```

The first NURBS curve is storage invalid because not enough parameters are stored; the second is storage invalid because the number of weights is not equal to the number of control points ((length of GM_PointArray)/3).

The following example shows how a validation function checks the geometry validity of some GM_NURBSCurve rows.

```
SQL> insert into test values(1,GM_NURBSCURVE(3,
GM_POINTARRAY(-1.5043915, 1.89224826, 0, -.91495325, 2.91036891, 0,
-.14332497, 1.86009708, 0, .59615213, 2.51383771, 0, 1.42136571,
1.86009708 , 0, 2.3215987, 3.31761717, 0, 2.88960285, 1.92439944,
0), GM_KNOTVECTOR(VECTOR( 0, 0, 0, 0, .25, .5, .75, 1, 1, 1, 1),
NULL), GM_WEIGHTARRAY(1, 1, 1, 1, 1, 1, 1 ), NULL));

1 row created.

SQL> insert into test
values(3,GM_NURBSCurve(3,GM_PointArray(1,2,10,1,2,3,9,2,3,5,7,6,4,4,4,9,0,4),G
M_KnotVector(Vector(0,0,0,0,0,0.5,1,1,1,1,1),NULL),GM_WeightArray(1,1,1,1,1,1)
```

```
,NULL));

1 row created.

SQL> select id,a.col.validation() from test a;

        ID A.COL.VALIDATION()
---------- ------------------
         1                  1
         3                  0
```

The first NURBS curve is valid. The second is a geometry invalid NURBS curve, because it violates the third geometry validation rule for NURBS curve.

5.8 Concluding remarks

In this chapter, functions on freeform data types are implemented in Oracle. These functions can be either implemented as standalone functions or as methods of user-defined data types. Standalone functions are created using "*CREATE FUNCTION*" SQL statement, and methods is created when creating user-defined data types.

All these functions are implementations of the conceptual models in Chapter 3. According to difference of functionalities, they can be categorized into five groups: access functions, geometry relationship functions, geometry transformation functions, conversion functions, and validation functions, which follows the same categorization with conceptual model.

Chapter 6

Freeform data exchange

There are several CAD applications which are able to visualize and model freeform geometries. Normally the freeform geometries are stored/retrieved in/from files, but nowadays some applications already implemented interfaces to exchange geometry data with spatial DBMSs. For example, MicroStation GeoSpatial supports a small program: *SpatialViewer*, which is able to query simple geometries from *SDO_Geometry* columns in Oracle Spatial to MicroStation. These simple geometries can be visualized, modified, and stored back to Oracle Spatial. Due to the absence of freeform geometries in spatial DBMSs, no interface exists which is able to exchange freeform geometries between CAD/GIS applications and spatial DBMS.

In order to show the possibility of exchanging freeform geometries between spatial DBMS and CAD/GIS applications, freeform data exchange between Oracle Spatial and CAD applications (MicroStation and AutoCAD) has been explored. Section 7.1 presents how to exchange freeform geometries between Oracle and MicroStation. Section 7.2 presents the similar procedure between Oracle and AutoCAD.

6.1 MicroStation

In MicroStation, there are two possibilities to exchange freeform geometries with Oracle.

The first possibility is to convert freeform geometries to line strings or polygon patches. This operation can be done inside Oracle with certain conversion functions, and store the line strings/polygon patches in another table, then visualize them using *SpatialViewer*. The disadvantages of this approach are significant, including:

- Visualization is approximated, and the more accurate the approximation, the heavier the computation.
- Modeling of freeform shapes becomes nearly impossible. The output geometries from DBMS to Microstation are not freeform curves or surfaces any more, but a large number of linestrings or polygons. Therefore the modeling methods for freeform shapes cannot be applied.
- Data exchange is one-way instead of two-way. After the freeform shapes are converted into linestrings or polygons, they can hardly be stored back to DBMS in freeform geometry form. Although freeform fitting algorithms can be applied, the results will probably be different freeform geometries, with different degree, control points, knot vector, and weights.

Therefore this approximation approach will not be adopted.

Another possibility is to write an interface manually with MicroStation programming environments. MicroStation includes several programming environments, such as MDL (MicroStation Development Language), JMDL (MDL's Java version), VBA (Visual Basic for

Applications). Programs can be written and embedded inside MicroStation with these environments.

Figure 6.1 illustrates the procedure of how we managed to exchange freeform geometries between MicroStation and Oracle using JMDL.

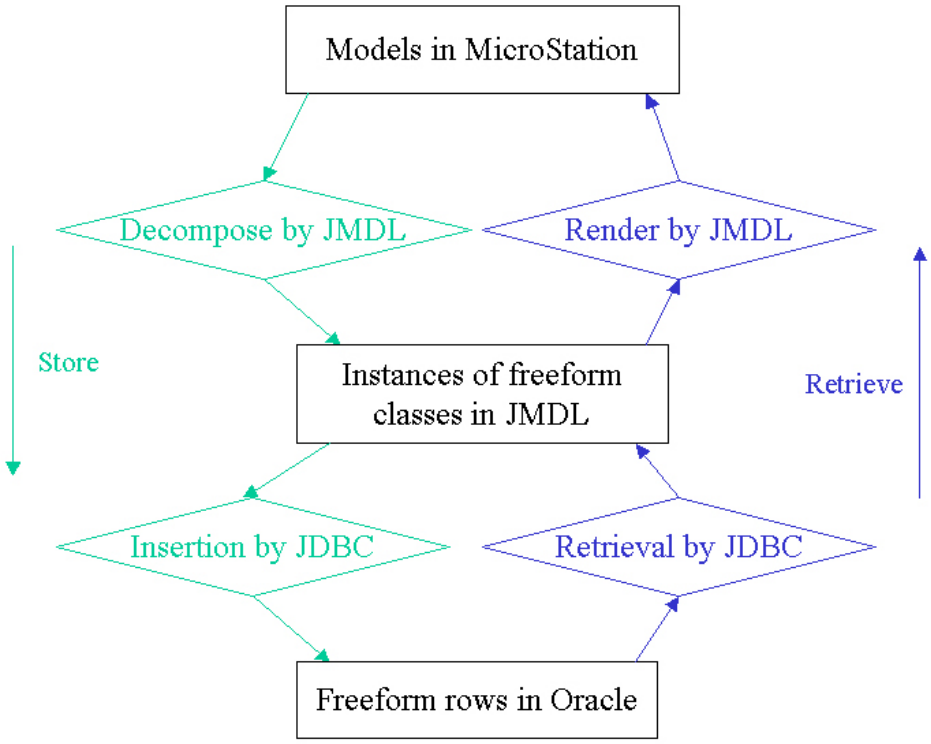

Figure 6.1: Freeform data exchange with JMDL

The storage procedure includes the following steps:

1. Create freeform shapes using MicroStation.
2. There're freeform classes in JMDL, such as *BsplineCurveElement*, and *BsplineSurfaceElement*. JMDL codes will check all the objects in current model, then construct an instance of the corresponding freeform class if a freeform geometry is found, and then finally insert this instance into an Array.
3. Instances of freeform classes in JMDL are reorganized according to the format of freeform types in Oracle, and they are inserted into Oracle using JDBC (Java Database Connectivity) bridge. JDBC is an API included in the Java related environment that provides cross-DBMS connectivity to a wide range of SQL databases, including Oracle. SQL statements can be executed with JDBC funtions, and then both SQL return values and SQL statements can be translated from/to Java variables. Detailed explanation of JDBC can be found in [25].

The retrieval procedure is just the other way around:

1. Freeform rows are selected from Oracle by JDBC.
2. Instances of freeform classes will be constructed from the selected information by JDBC.
3. Constructed instances of freeform classes are rendered in the view windows in MicroStation, using rendering functions of JDML.

During both storage and retrieval procedures, the compatibility between the freeform data types in Oracle and JMDL should be taken care of. The freeform data types in Oracle require the most basic parameters for freeform geometries, while JMDL also require some extra parameters for visualization. For example, JMDL's *BsplineSurfaceElement* also requires parameters such as number of rule lines in both direction, boundary points, etc. The extra parameters can be stored in Oracle in the same table, but additional columns than the column of freeform data type. All applications, which support freeform geometries, can store the freeform geometries in Oracle by storing the basic geometric information in one column with freeform data type, and storing the extra parameters in additional columns of the same table.

The following texts are description of table *test_NURBSSurface*, which is used to store NURBS surfaces from MicroStation. Note that the first column is of type *GM_NURBSSurface*, and the basic geometric information for NURBS surfaces is stored here. The other columns store the extra parameters returned by JMDL. They will be used to reconstruct the NURBS surfaces in MicroStation.

```
SQL> desc test_NURBSSurface;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 COL                                                GM_NURBSSURFACE
 URULES                                             NUMBER
 VRULES                                             NUMBER
 OBOUNDARYPOINTS                                    GM_POINTARRAY
 NUMBOUNDARY                                        NUMBER
 ISCLOSEDU                                          NUMBER
 ISCLOSEDV                                          NUMBER
 BOUNDARYOUT                                        NUMBER
```

Figure 6.2 shows 3 B-spline curves created in MicroStation. They can be stored into Oracle and retrieved back without data loss. More complex examples can be found in Chapter 7.

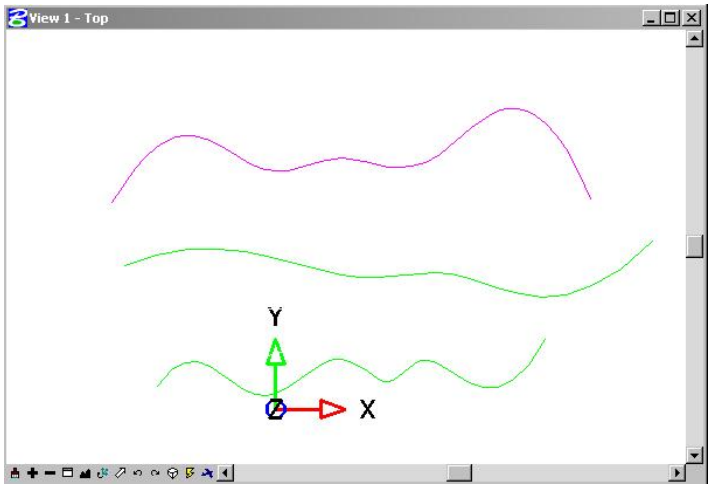

Figure 6.2: 3 B-spline curves created in MicroStation

6.2 AutoCAD

Similar to MicroStation, there is no direct tool which is able to exchange freeform geometries between AutoCAD and Oracle either. AutoCAD supports several runtime extension pro-

gramming languages, such as AutoLISP, Visual LISP, VBA, ObjectARX, ObjectDBX, which can be used to develop AutoCAD applications.

ObjectARX is a C++ based programming language. An application programmed with ObjectARX is a dynamic link library (DLL) that shares the address space of AutoCAD and makes direct function calls to AutoCAD. The development of an ObjectARX application contains the following steps [29]:

1. The ObjectARX application needs to be programmed in Visual C++ environment first. After installing the ObjectARX add-on to Visual C++, an ObjectARX project can be created with wizards easily, as shown in Figure 6.3. This add-on supports many helpful tools for ObjectARX developers, such as function dictionary, realtime debugger, etc. The complied ObjectARX application will be a .ARX file and some configuration files.

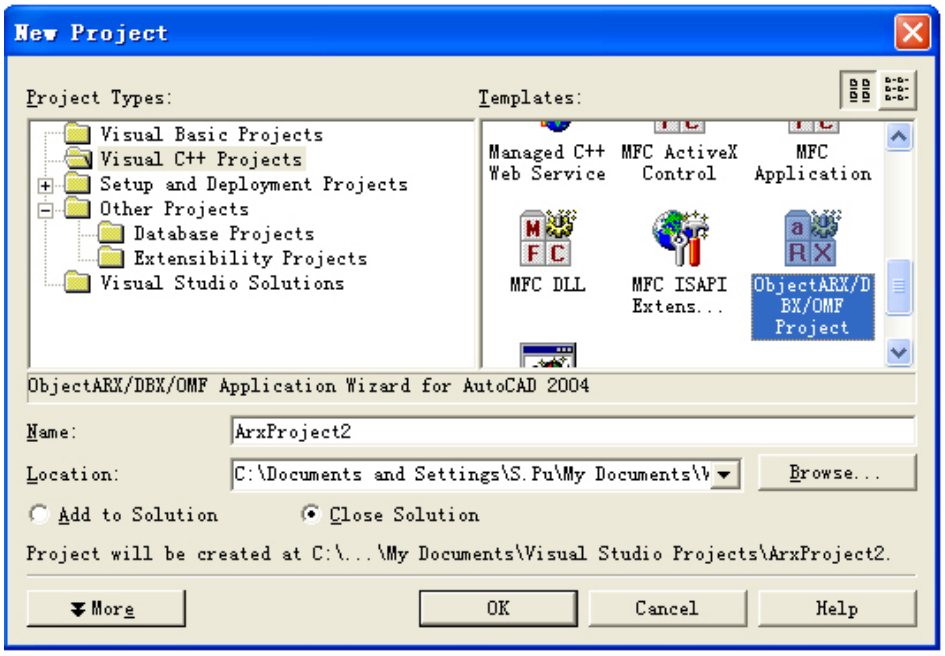

Figure 6.3: Create an ObjectARX project in Visual C++

2. The .ARX file is loaded and registered in AutoCAD. This can be done with a tool called 'Load Application' in AutoCAD. All developer needs to do is just locate the .ARX file and click 'OK'.

3. The loaded ObjectARX application is finally executed by inputting commands (Figure 6.4). The command's name doesn't need to be the same as the .ARX file name. The relationship between a .ARX file and its command name is specified in the configuration files.

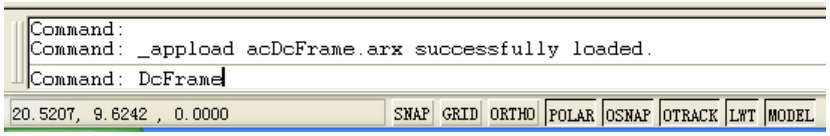

Figure 6.4: Execute an ObjectARX application in AutoCAD

Figure 6.5 illustrates the procedure of visualizing freeform spatial data from Oracle in AutoCAD.

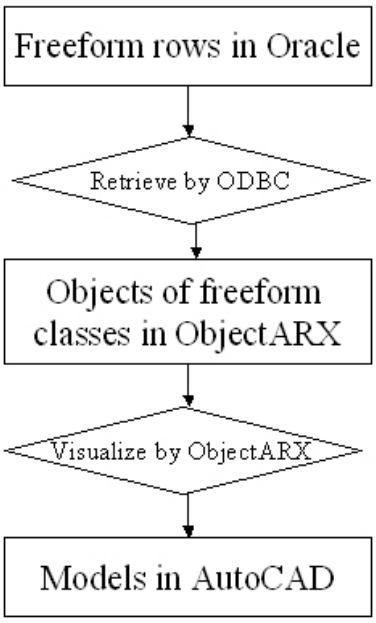

Figure 6.5: The procedure of visualizing freeform spatial data from Oracle in AutoCAD

This is mainly done in the following steps:

1. Retrieve freeform spatial data from Oracle using ODBC (Open Database Connectivity) bridge. ODBC is able to make connections to DBMS with some functions, and connection information such as database name, user name and password are specified as parameters of these functions. Then database data can be retrieved by calling functions which take SQL statements as parameters. The return value will be in the formats of local variables, and hence database data are translated to local formats, i.e. C++ variables. Detailed explanation of ODBC can be found in [26].

2. There are freeform classes in ObjectARX, such as *AcGeNurbSurface*, *AcGeNurbCurve3d*, *AcGeNurbCurve2d*. Objects of these classes can be constructed from the translated database data in the first step.

3. Objects of the freeform classes in ObjectARX are then visualized on an AutoCAD window using function 'acedRedraw()' [29].

Due to time limit, the storage procedure from AutoCAD to Oracle hasn't been researched. Considering the visualization procedure of AutoCAD and storage procedure of MicroStation, it might be done by analyzing the current model with ObjectARX code, extracting freeform geometries, and then inserting these to Oracle using ODBC.

An example of visualizing freeform spatial data from Oracle in AutoCAD will be in Figure 7.6.

6.3 Conclusion remarks

In this chapter, freeform data exchange between two CAD applications (AutoCAD, MicroStation) and Oracle is achieved.

For storage procedure from MicroStation to Oracle, first, freeform shapes in MicroStation are extracted to instances of Java classes with JMDL codes, and then parameters are retrieved from these instances to construct SQL statements which insert freeform geometries into Oracle. The SQL statements are executed with JDBC bridge. The retrieval procedure is just the other way around from storage procedure.

For retrieval procedure from Oracle to AutoCAD, first, freeform spatial data is retrieved from Oracle using ODBC, then objects of freeform classes in ObjectARX are constructed using these freeform spatial data, and then they are visualized in AutoCAD using ObjectARX visualization function.

Chapter 7

Test cases

In this chapter, three freeform curves examples and a freeform surface example will be illustrated. They are all created in MicroStation, and can be transferred between MicroStation and Oracle without data loss.

Section 7.1 gives the freeform curve examples. Section 7.2 gives the freeform surface example.

7.1 Freeform curves (MicroStation)

Three NURBS curves: a circle, an ellipse and a normal NURBS curve are created in MicroStation, as shown in Figure 7.1.

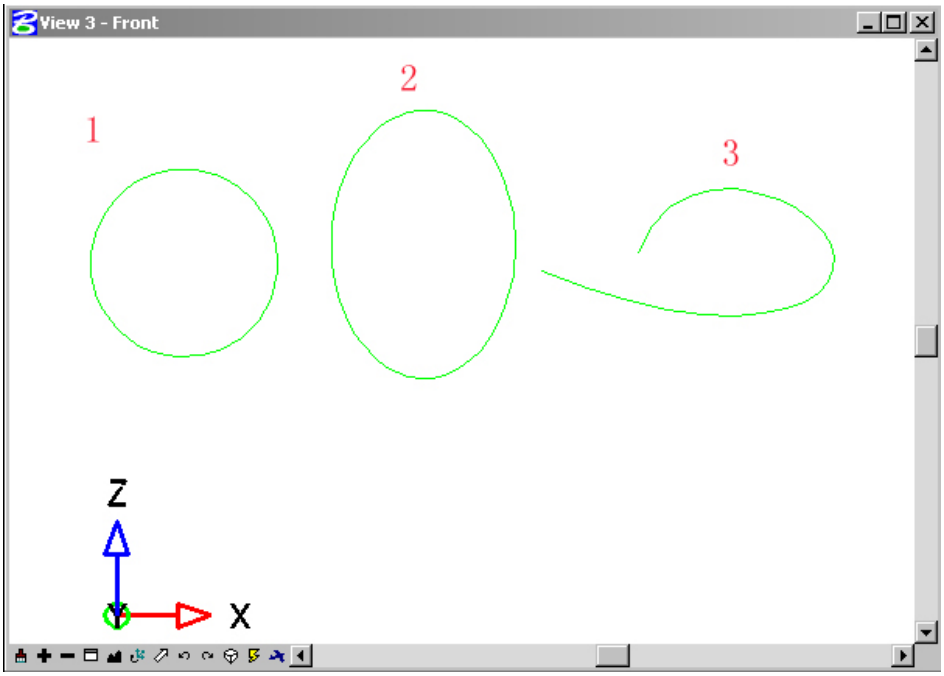

Figure 7.1: 3 NURBS curves

They can be inserted into Oracle with a pre-defined JMDL program: *SelectionTest2*. Execution of *SelectionTest2* can simply be done by typing command: **java example.dgn.SeletionTest2**, in the key-in window of MicroStation, as shown in Figure 7.2.

In Oracle they are stored as:

```
SQL> select col from test_NURBS;
```

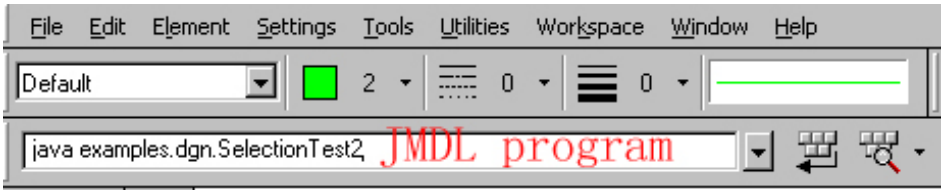

Figure 7.2: JMDL program to insert freeform curves

```
COL(DEGREE, CONTROLPOINTS, KNOTS(KNOTS, WEIGHTS), WEIGHTS, TRIM)
--------------------------------------------------------------------------------
GM_NURBSCURVE(2, GM_POINTARRAY(.885744519, 0, 3.12600375,
-.07769652, 0, 3.81417 592, -.19195058, 0, 2.63572543, -.30620464,
0, 1.45727493, .771490457, 0, 1.9475 5326, 1.84918556, 0,
2.43783158, .885744519, 0, 3.12600375), GM_KNOTVECTOR(VECTO R(0, 0,
0, .333333333, .333333333, .666666667, .666666667, 1, 1, 1), NULL),
GM_WEIGHTARRAY(1, .5, 1, .5, 1, .5, 1), NULL)

GM_NURBSCURVE(2, GM_POINTARRAY(2.90828926, 0, 2.70598357,
2.90828926, 0, 4.39692 09, 1.90078572, 0, 3.55145224, .893282179, 0,
2.70598357, 1.90078572, 0, 1.86051 491, 2.90828926, 0, 1.01504624,
2.90828926, 0, 2.70598357), GM_KNOTVECTOR(VECTOR (0, 0, 0,
.333333333, .333333333, .666666667, .666666667, 1, 1, 1), NULL),
GM_WEIGHTARRAY(1, .5, 1, .5, 1, .5, 1), NULL)

COL(DEGREE, CONTROLPOINTS, KNOTS(KNOTS, WEIGHTS), WEIGHTS, TRIM)
--------------------------------------------------------------------------------

GM_NURBSCURVE(3, GM_POINTARRAY(3.80318199, 0, 2.64922409,
4.01886803, 0, 3.23952 272, 5.23352098, 0, 3.09194806, 5.39401765,
0, 2.12896805, 3.99616423, 0, 2.1289 6805, 3.09930089, 0,
2.51461793), GM_KNOTVECTOR(VECTOR(0, 0, 0, 0, .333333333, .
666666667, 1, 1, 1, 1), NULL), GM_WEIGHTARRAY(1, 1, 1, 1, 1, 1),
NULL)
```

Now let's check their validity:

```
SQL> select a.col.validation() from test_NURBS a;

A.COL.VALIDATION()
------------------
                 1
                 1
                 1
```

This means they are all geometry valid.
Now let's rotate all the NURBS curves by 180 degrees along Y axis.

```
SQL> update test_NURBS a set a.col=a.col.rotateY(3.1415926);

3 rows updated.
```

Now let's retrieve them back to MicroStation and visualize as Figure 7.3.

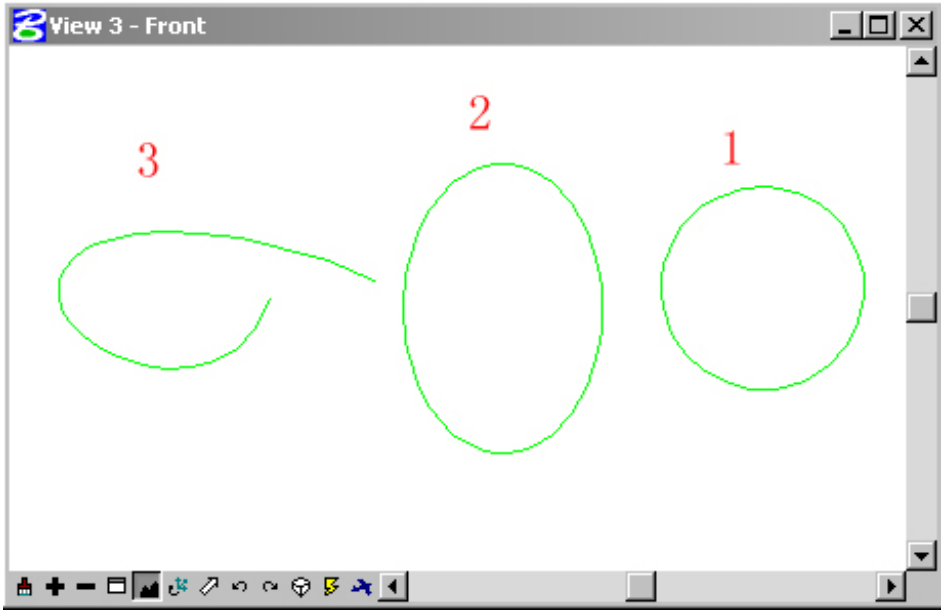

Figure 7.3: Rotated NURBS curves

The visualization is done by executing another JMDL program: *NURBSCurveCommand*, in the key-in window of MicroStation, as shown in Figure 7.4.

Figure 7.4: JMDL program to visualize freeform curves

7.2 Freeform surface (MicroStation and AutoCAD)

In order to show that spatial DBMS can be a good intermediate system between CAD applications, freeform data exchange between MicroStation and AutoCAD via Oracle, and direct exchange between them two, are tested in the following examples.

First let's test the first case: freeform data exchange via Oracle. The sport car in Figure 7.5 is made up of 306 NURBS surfaces. This example will show how to store this sport car from MicroStation to Oracle, then retrieve from Oracle to AutoCAD, and then visualize in AutoCAD (as in Figure 7.6).

The storage from MicroStation to Oracle is done by executing key-in command: *jmdl examples.dgn.NURBSSurfaceCommand*, in MicroStation.

The visualization in AutoCAD is done by executing ObjectARX command: *NURBS_LOAD*, in AutoCAD.

From comparison between Figure 7.5 and Figure 7.6, it is clear that the sport car is maintained perfect.

Now let's test the second case: direct freeform data exchange between MicroStation and AutoCAD. Different CAD applications have their own file formats: the .dgn format is a

standard MicroStation file format, and the .dxf format is a standard AutoCAD file format. The definitions for CAD models are different in the two file formats. The sport car in Figure 7.5 is originally in .dgn format. Using MicroStation's exporting tool, we can convert the car from .dgn file to .dxf format. The converted car in .dxf format looks as Figure 7.7 in MicroStation and Figure 7.8 in AutoCAD.

It is clear from Figure 7.7 and Figure 7.8 that the converted model lost many shapes, probably due to the difference between .dgn and .dxf formats.

7.3 Concluding remarks

From the test cases in this chapter, the following conclusions can be stated:

- Freeform geometries can be exchanged between Oracle and CAD applications (MicroStation and AutoCAD).
- Freeform geometries can be geometric transformed in Oracle first, then visualized in CAD applications.
- DBMS can be a good intermediate system between CAD applications. Models in different CAD applications are in different formats. If all follow the same formats in DBMS and stored in DBMS, they can be exchanged between CAD applications without data loss.

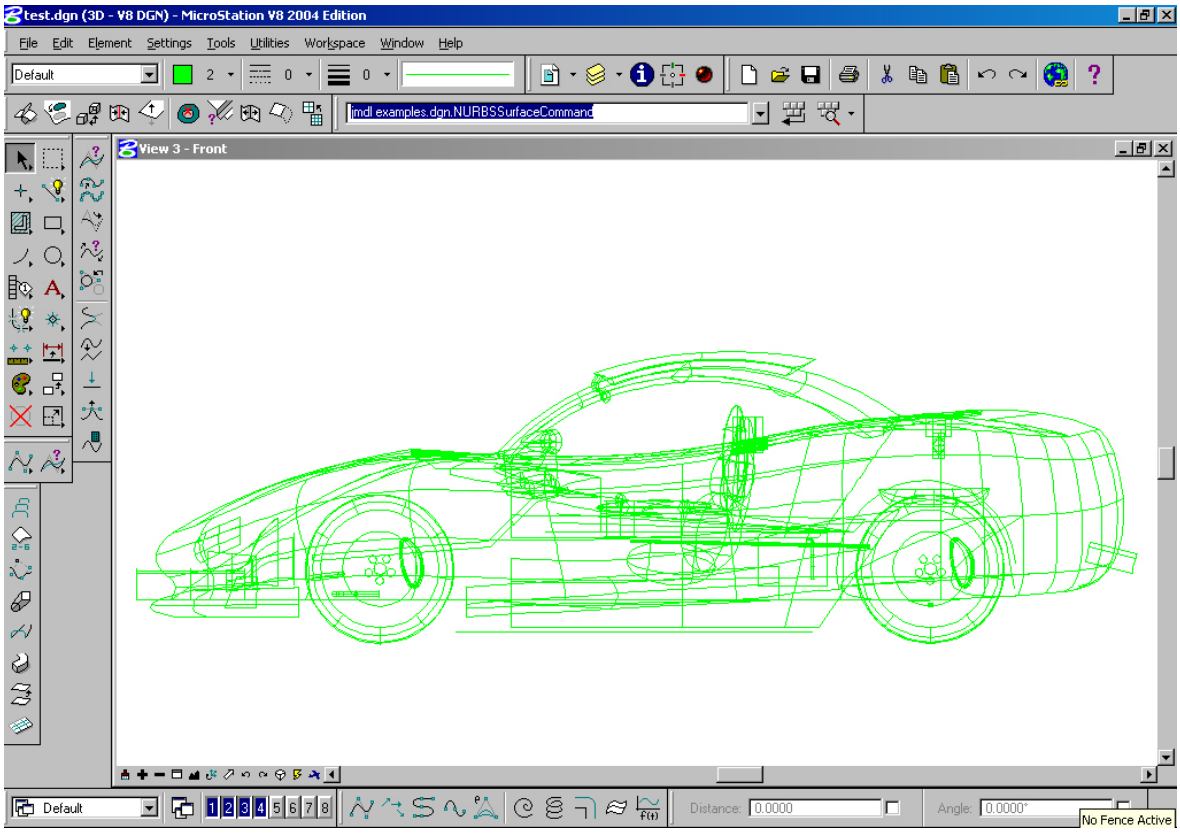

Figure 7.5: A sport car with 306 NURBS surfaces in Microstation v8.

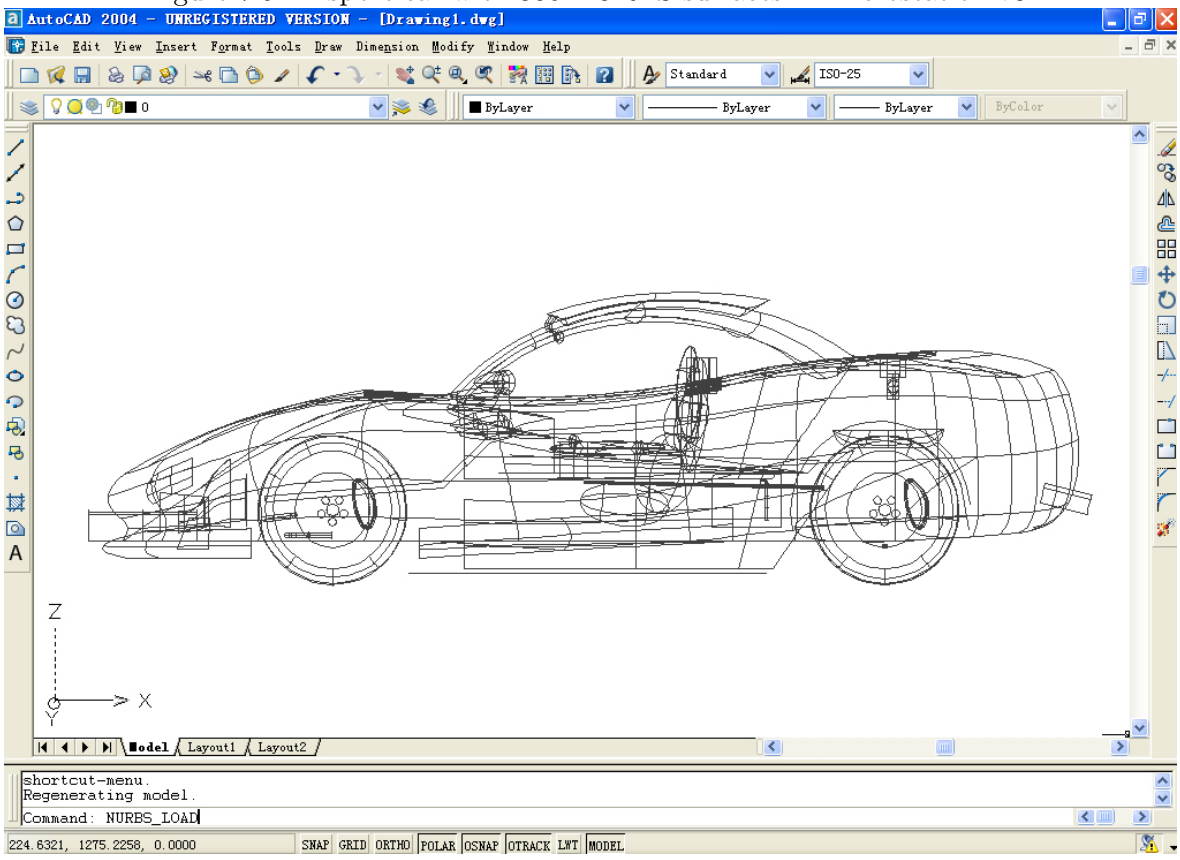

Figure 7.6: The same sport car in AutoCAD 2004

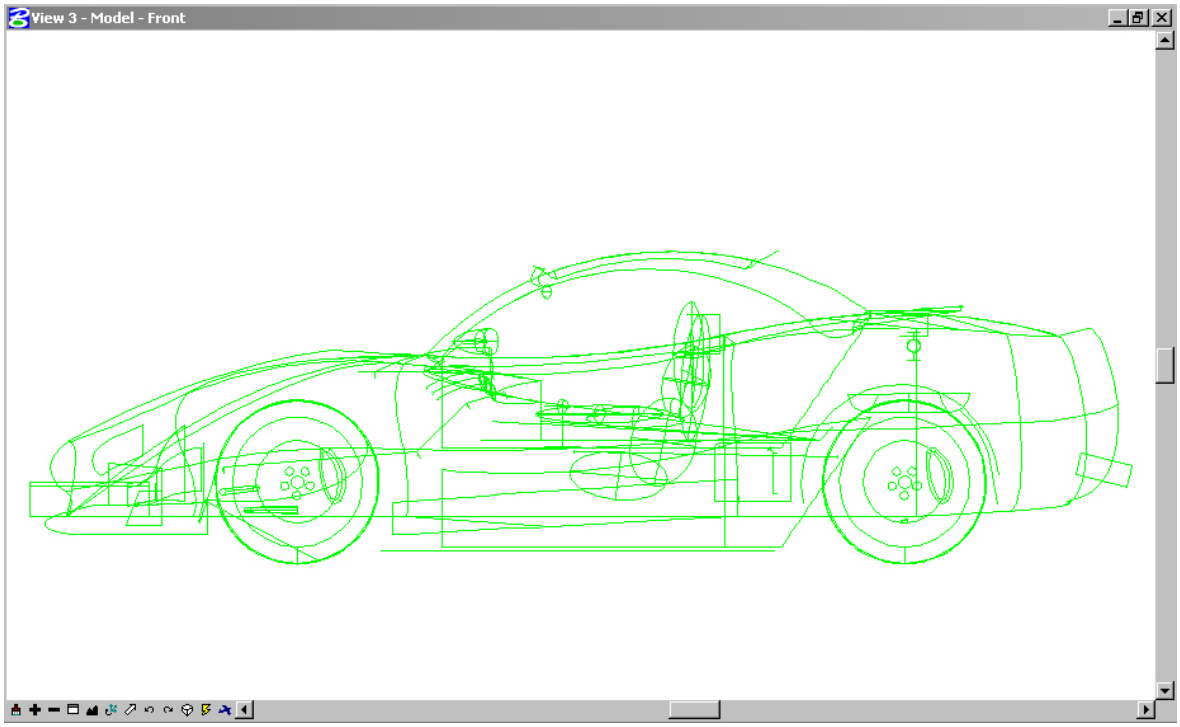

Figure 7.7: The converted sport car visualized in MicroStation v8

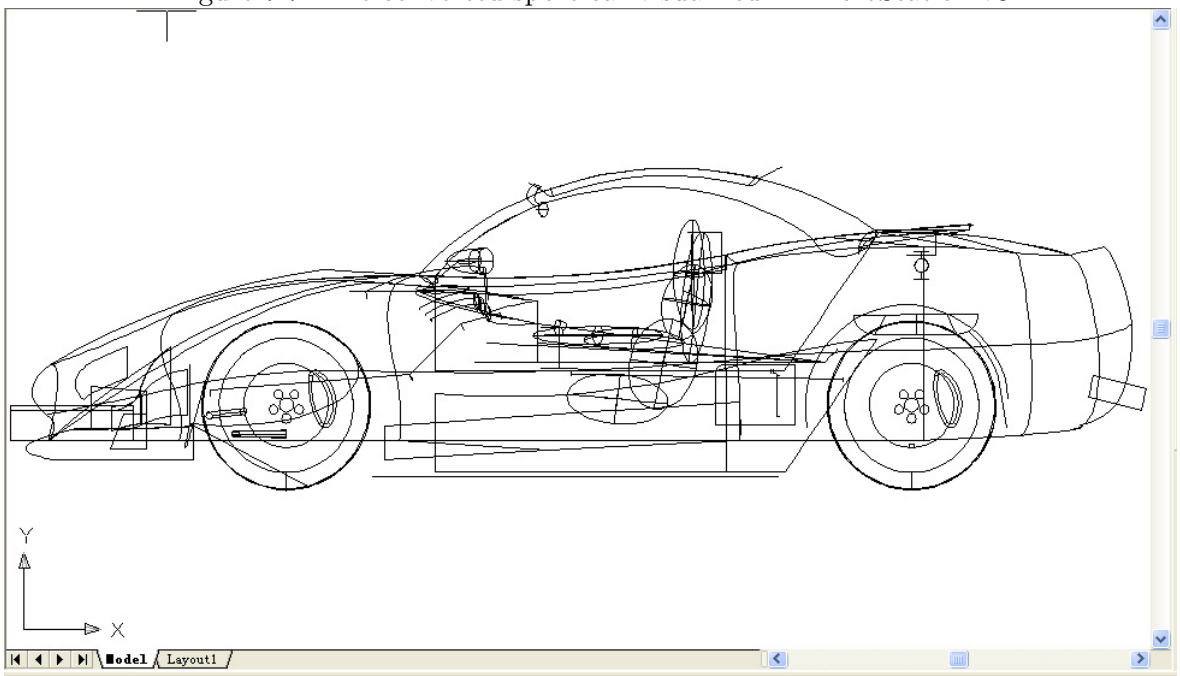

Figure 7.8: The converted sport car visualized in AutoCAD 2004

Chapter 8

Conclusions and recommendations

8.1 Conclusions

This research is, as far as we know, the first attempt of managing freeform geometry types in a spatial DBMS. Three popular mathematical representation of freeform geometries: Bézier, B-spline and NURBS curve/surface have been used to design the conceptual models of freeform data types, then user-defined data types, which are based on conceptual models, have been created in Oracle, and finally several useful spatial functions have been created on these data types. Two CAD applications: MicroStation and AutoCAD, are able to visualize/store freeform data from/to Oracle using the implemented freeform data types.

The following conclusions can be drawn from:

- Spatial DBMS is able to manage complex geometries like freeform curves and surfaces. Freeform geometries can be implemented as user-defined data types in Oracle. An alternative approach is to use the existing spatial type: *SDO_Geometry*. This approach was not adopted because of the complexity and data redundancy compared to the user-defined data type approach.

- Spatial functions are very helpful to manage the freeform data types. They can be used to return geometric information of the stored freeform geometries, do basic geometric transformations, maintain geometry validity, etc.

- Some simple geometric operations can be done in DBMS level with spatial functions, especially when the target data sets are huge. Current CAD/GIS applications usually store geometries in files. DBMSs have extraordinary abilities in data storage efficiency and fast data processing, and it would be faster to retrieve/store the spatial data from/to DBMS than files. But we also need to be aware that a DBMS is not good at heavy computations. Therefore complex geometry operations are still better to be done in CAD/GIS applications.

- Validation functions are required to maintain the validity of freeform geometries. Currently storage validity and geometric validity are checked by the validation function.

- Since the conceptual models for freeform data types are generic and OGC Abstract Specifications are considered, they can be readily implemented in any other DBMSs as well.

- CAD/GIS applications which support Bézier, B-spline or NURBS curve/surface can store/retrieve the freeform geometries in/from Oracle using the implemented freeform data types.

- Incompatibility between applications' freeform type definition and DBMS' freeform type definition can be solved by storing extra attributes of the first definition in additional columns of the same table with the freeform geometry column in DBMS.
- Spatial DBMS can be a good central system system between CAD/GIS applications, because format incompatibility can be well solved by following the same format in the central DBMS system, as proven in Figure 7.5, Figure 7.6 and Figure 7.7.

8.2 Recommendations

Since this research is the first attempt to manage freeform data types in a spatial DBMS, recommendations for future research are given:

- OGC implementation specifications and mainstream spatial DBMSs should consider involving freeform data types. Several simple geometries have been standardized in OGC implementation specifications and implemented in mainstream spatial DBMSs, but freeform data types are still absent. Bézier, B-spline and NURBS curve/surface can represent freeform geometries well, because of their nice mathematical and geometric properties, therefore they are recommended to be involved in the future OGC specifications.
- Validation rules for freeform curves and surfaces can be further investigated, and specified in relevant standards.
- Because freeform geometries are stored in other data types than *SDO_Geometry*, they cannot be stored in the same column with simple geometries. This can lead to inconvenience under certain circumstances. Two approaches can be attempted to implement. The first approach is to store different kinds of geometries in different tables, and use a meta-data table to organize and manage all these tables. Another approach is to implement a super data type for all user-defined data types. Which approach is more suitable and efficient should be analyzed in future research.
- A spatial index can make spatial queries much more efficient. Now the spatial index in Oracle Spatial works only on existing simple geometries. A spatial index which also works on freeform data types should be researched to implement in the future.
- Data types for more geometry types can be made in Oracle. The currently supported geometry types in Oracle Spatial are still limited. Following the procedure of creating data type for freeform geometries, data types of more geometry types, such as cone, cylinder and sphere, can be implemented.
- Some tests would be necessary to determine which functions should stay in DBMS level and which ones should be in CAD/GIS level. Comparing to CAD/GIS front ends, DBMSs are good at data processing, but lack in the ability of heavy computation. Therefore the time spent on the same geometric operation in both DBMSs and CAD/GIS fronts end should be compared. The size of target data sets should also be considered as an important factor, because some geometric operations may be faster in DBMSs for small amounts of spatial data, but may be slower for huge amounts of spatial data.
- A separate data type for error messages would be of great help after more geometry data types are created. At least an error number attribute and an error message attribute should be included in this type.

- It would be interesting to implement data types for freeform curves and surface in other spatial DBMSs than Oracle Spatial.
- It would also be interesting to know how to exchange freeform geometries between Oracle and other CAD/GIS applications than MicroStation and AutoCAD.

Bibliography

[1] Sisi Zlatanova, *Large-scale 3D data integration - An Introduction to the Challenges for CAD and GIS Integration*, Directions Magazine, available at http://www.directionsmag.com, 2004

[2] C.A.Arens, *Modelling 3D spatial objects in a Geo-DBMS using a 3D primitives*, Msc thesis, 2003

[3] Les Piegl and Wayne Tiller, *The NURBS book 2nd edition*, Springer, 1997

[4] Bentley, *MicroStation v8 user's manual*, 2004

[5] AutoDesk, *AutoDesk 2005 user's mauual*,2005

[6] Mathworld, *Mathworld website*, http://mathworld.wolfram.com/, 2005

[7] ISO, *ISO standard 19107-Geographic information*, Internet Organization for Standardization, 2002

[8] Les Piegl and Wayne Tiller, *Computing offsets of NURBS curves and surfaces*, Computer-Aided Design Vol. 31 (p 147-156), 1999

[9] Les Piegl, *On NURBS: a survey*, IEEE Computer Graphics and Applications, Vol. 11, No.1 (p 55-71), 1991

[10] T.Lyche and K.Morken, *Knot removal for parametric B-spline curves and surfaces*, research report No. 109, Institute of informatics, University of Oslo, 1987

[11] W.Bronsvoort and F.Post, *Slides of geometric modelling course*, TUDelft, 2005

[12] Beeker E., *Smoothing of shapes designed with freeform surfaces*, Computer-Aided design Vol. 18 (p 224-232), 1985

[13] Siyka Zlatanova, Alias Abdul Rahman, Morakot Pilouk, *3D GIS: current status and perspectives*, Proceedings of ISPRS/8-12 July/Ottawa Canada/CDROM/8p, 2002

[14] Jantien Stoter and Siyka Zlatanova, *Visualisation and editing of 3D objects organised in a DBMS*,Proceedings of the EuroSDR Com V. Workshop on Visualisation and Rendering, 2003

[15] OGC, *Abstract specifications overview*, available at http://www.opengis.org, 1999

[16] OGC, *OGC official website*, http://www.opengis.org, 2005

[17] OMG, *Introduction to OMG's Unified Modeling Lanuage*, available at http://www.omg.org, 2004

[18] OGC, *OpenGIS simple features specification for SQL*, available at http://www.opengis.org, 1999

[19] OpenGL, *OpenGL online documentation*, available at http://www.opengl.com, 2005

[20] Oracle, *Oracle Spatial user's guide and reference*, available at http://www.oracle.com/technology/documentation, 2003

[21] Oracle, *Oracle Spatial 10g data sheet*, http://www.oracle.com/technology/products/spatial/

[22] Oracle, *Oracle SQL guide 10g release 1*, available at http://www.oracle.com/technology/documentation, 2003

[23] Oracle, *Oracle database application developer's guide - object-relational features 10g release 1*, available at http://www.oracle.com/technology/documentation, 2003

[24] Oracle, *Oracle JDBC developer's guide 10g release 1*, available at http://www.oracle.com/technology/documentation, 2003

[25] Oracle, *Oracle ODBC developer's guide 10g release 1*, available at http://www.oracle.com/technology/documentation, 2003

[26] Microsoft, *MSDN online*, available at http://www.microsoft.com/data/, 2005

[27] Oracle, *Oracle PL/SQL user's guide and reference 10g release 1*, available at http://www.oracle.com/technology/documentation, 2003

[28] Bentley, *JMDL Developer's Guild*, available at http://www.bentley.com/support, 2005

[29] AutoDesk, *ObjectARX Developer's Reference*, available at http://www.autodesk.com, 2005