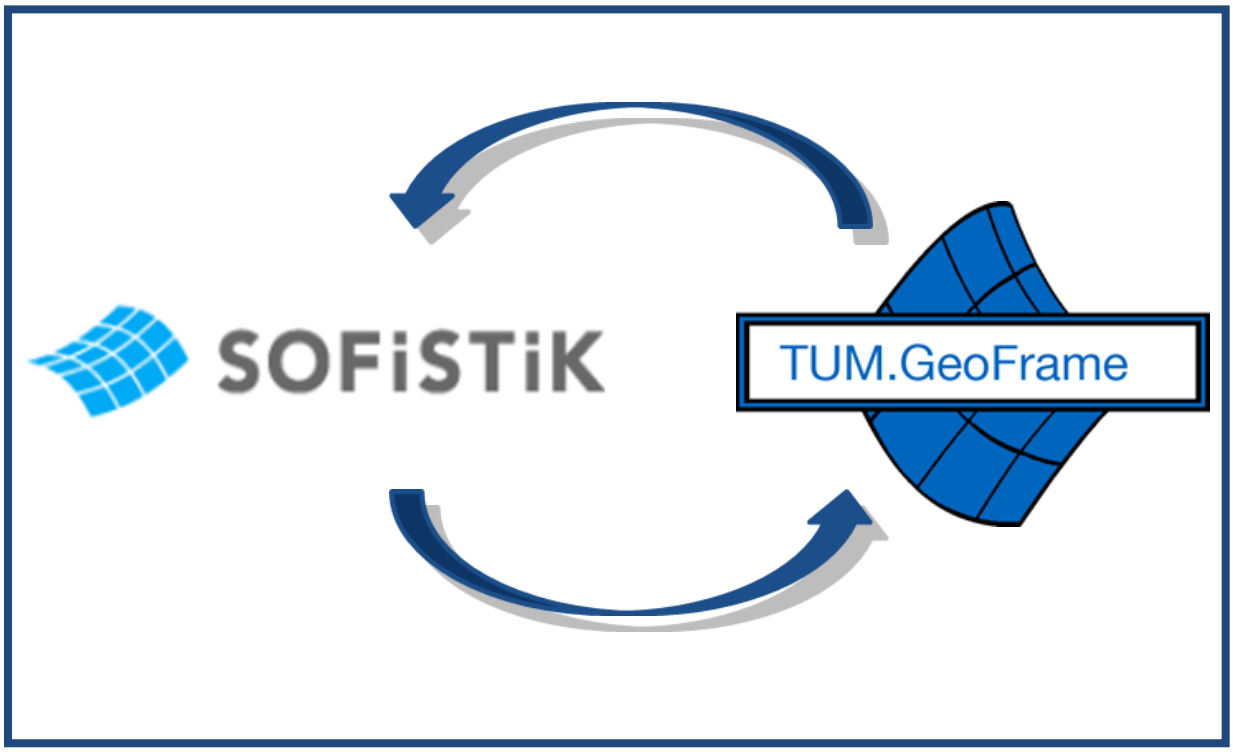
BACHELOR THESIS

Import und Export der Datenbasis CDBase der SOFiSTiK AG in TUM.GeoFrame



Lehrstuhl:
Technische Universität München
Lehrstuhl für Computation in Engineering
Univ.-Prof. Dr. rer. nat. Ernst Rank

Betreuer:

Dipl.-Ing. Christian Sorger M.Sc.

Verfasser:

Cornelius Preidel cand. ing.
Matr.Nr.: 3603721

Adresse:

Rosenheimer Straße 115
81667 München

Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

München, den

Ort / Datum	**Unterschrift**

Ich danke meinem Betreuer, Herrn Dipl. Ing. Christian Sorger MSc., für die Anregung zu dieser Arbeit, sein Interesse daran und seine stets sehr gute Betreuung.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die im Jahre 1987 gegründete SOFiSTiK AG stellt im Bereich des konstruktiven Ingenieurbaus Anwendern basierend auf den Plattformen AutoCAD und AutoDesk Architectural Desktop Finite-Elemente- und CAD-Software zur Verfügung. [6]
Von Prof. Dr. Casimir Katz, einem Gründungsmitglied der SOFiSTiK AG („Software für Statik und Konstruktion“), wurde schon vor der Firmengründung ein geschlossenes Datenbasiskonzept mit dem Namen CDBase („Casimir´s Data Base“) erstellt, welches es erlaubt, sämtliche Daten zu einer Geometrie inklusive konstruktiver Daten, wie z.B. Berechnungen und Lastfälle, in nur einer Datei zu speichern.[1]
Parallel wird an dem Lehrstuhl für „Computation in Engineering“ an der Technischen Universität München seit dem Jahr 2007 TUM.GeoFrame entwickelt. Dieses Projekt hat zum Ziel, die „high-order“ Element-Erstellung für die Finite Elemente Methode (FEM) zu automatisieren, was bisher noch nicht durch Software realisiert wurde.[7] Es ist von besonderem Vorteil, eine große Anzahl von verschiedenen Datenbasen in dieses Projekt zu integrieren, um dem Anwender eine möglichst umfangreiche Funktionenvielfalt zur Verfügung zu stellen und so das Programm, welches auf den Zugriff auch auf externe Datenbasen angewiesen ist, zu vervollständigen.

1.2. Aufgabenstellung

Ziel der vorliegenden Bachelorarbeit ist es, die Datenbasis CDBase der SOFiSTiK AG und der Dateiendung *.cdb zunächst in Struktur und Funktionsweise zu erklären und anschließend die Erstellung eines Parsers zum Auslesen dieses Dateiformats in der Programmiersprache C++ zu dokumentieren.
Der Parser wird in das Framework TUM.GeoFrame implementiert, um mit diesem ausgelesene Daten darstellen zu können und vorhandene Daten wiederum zu exportieren. Dabei beschränkt sich der Parser auf ausgesuchte Inhalte der Datenbasis.

1.3. Verwendete Software

Voraussetzung für den Zugriff auf SOFiSTiK-Datenbanken ist eine Installation des SOFiSTiK-Software-Pakets, welches momentan für Studierende in der Version 2012 bzw. Analysis 27 verfügbar ist.[5]

Weiterhin werden von der SOFiSTiK AG zusätzlich Tools zur Verfügung gestellt, mit denen Konsolen-Zugriffe auf die Datenbanken ermöglicht werden. In der vorliegenden Arbeit wird im Besonderen auf das Tool dbinfo, welches ebenfalls in dem SOFiSTiK-Software-Paket enthalten ist, eingegangen. Für die Erstellung des Parsers in C++ wird als Arbeitsumgebung Microsoft Visual Studio 2008 verwendet.

Für die Implementierung des Parsers wurde die aktuellste Version des Projekts TUM.GeoFrame von dem Lehrstuhl „Computation in Engineering“ der Technischen Universität München zur Verfügung gestellt. Des Weiteren sind hierzu die Open-Source-Klassenbibliotheken Qt der Nokia Corporation in der Version 4.7.1 [2] und VTK in der Version 5.8.0 [8] notwendig, da das Programm auf diesen Bibliotheken basiert.

2. Struktur und Funktionsweise der CDBase

2.1. Aufbau der CDBase

Eine CDBase-Datenbank der SOFiSTiK AG ist mit der Endung *.cdb als einzelne Datei gespeichert. In diesem Dateiformat können sämtliche geometrische und konstruktive Daten gesichert werden.

Die CDBase-Datenbasis besitzt eine index-sequentielle Struktur [4], was bedeutet, dass sie mittels sogenannter Datenschlüssel aufgegliedert wird. Um beim Auslesen einer CDB-Datenbasis gewünschte Daten zu erhalten, ist jedem Datensatz ein definierter Schlüssel bestehend aus einem Kwh- und Kwl-Wert zugewiesen.

Vorteil dieses Datenkonzepts ist es, dass dem Anwender ein schneller Zugriff ermöglicht wird. Anstatt eine komplette Datei zu durchsuchen, was bei großen Datenmengen lange Wartezeiten zur Folge haben kann, kann mittels des Schlüssels direkt auf den gewünschten Datensatz zugegriffen und so der Zeitaufwand minimiert werden.

Eine entsprechende Übersicht und Erklärung dieser Schlüssel ist in der Hilfsdatei cdbase.chm [3] enthalten, die im SOFiSTiK-Software-Paket den Anwendern zur Verfügung gestellt wird.

2.2. Zugriff-Schlüssel

2.2.1. Kwh-Schlüssel

Der Kwh-Schlüssel ist eine maximal dreistellige Ziffer, mit der die gesamte CDB-Datenbasis zunächst in ihre Grundbestandteile aufgegliedert wird.[4] Die folgende Gliederung ist der cdbase.chm [3] entnommen, wurde allerdings auf ihre Grundbestandteile gekürzt (eine vollständige Gliederung ist in der cdbase.ch unter dem Unterpunkt „SOFiSTiK Data“ zu finden):

```
CTRL :      kwh: 0          Control data
MATE :      kwh: 1          Material
BORE :      kwh: 2          Soilprofiles
AXIS :      kwh: 3          Geometric Axis
TEND :      kwh: 4          Prestressing Schemes
AREA :      kwh: 5          Geometric Area
CON :       kwh: 8          Steel connections
SECT :      kwh: 9          Cross sections
```

```
SYST :      kwh: 10 to 11    Controlinfo
LC :        kwh: 12 to 14    Loadcaseinfo
            kwh:  18 to 19   Viewing info
NODE :      kwh: 20 to 29    Nodes
G??? :      kwh: 30 to 39    Structural Elements
TEND :      kwh: 40 to 49    Tendons
            kwh: 50 to 59    reserved
            kwh: 60 to 69    reserved
            kwh: 70 to 79    reserved
HIST        kwh: 80 to 89    Time History
EIGE        kwh: 90 to 99    Eigenvalues
BEAM :      kwh: 100 to 109 Beamelements
BSCT :      kwh: 140 to 149 Sections
TRUS :      kwh: 150 to 159 Truss elements
CABL :      kwh: 160 to 169 Cable elements
SPRI :      kwh: 170 to 179 Springs, Dampers
BOUN :      kwh: 180 to 189 Boundary elements
PIPE :      kwh: 190 to 199 Pipe elements
QUAD :      kwh: 200 to 299 QUAD elements
BRIC :      kwh: 300 to 399 BRIC elements
            kwh: 400 to 499 Substructures
            kwh: 900 to 999 Bridge Segments
```

Code 1: Kwh-Schlüssel der CDB-Datenbasis [3]

Anhand dieser Gliederung können nun die gewünschten Daten und deren zugehörige Kwh-Schlüssel abgelesen werden. Beispielsweise werden die Daten der Netz-Knoten mit den Kwh-Werten von 20 bis 29 ausgelesen.
An diesem Punkt zeigt sich eine Grundidee hinter dem Datenkonzept der CDBase. Den Netzknoten ist nicht nur ein Datensatz, sondern mehrere zugewiesen, in die neben den geometrischen Daten wiederum weitere, mit den Netz-Knoten in Verbindung stehende, konstruktive Daten gespeichert werden können. Die Datensätze für die Netzknoten gliedern sich folgendermaßen auf:

```
20/* Nodes
21/* Kinematic constraints
23/* Nodal loads
24/* Nodal results
25/* Dynamic nodal results
26/* Nonlinear nodal results
27/* Nodal coordinate offsets
29/* Nodal scalar physical values
```

Code 2: Kwh-Schlüssel für die Netz-Knoten (Kwh = 20 – 29) [3]

Hier sind neben den geometrischen Parametern (Kwh =20) noch weitere konstruktive Daten zu finden, wie z.B. kinematische Zwangsbedingungen (Kwh = 21) und Knotenlasten (Kwh = 23) oder aber auch Ergebnisse von abgeschlossenen Rechnungen mit den Knoten (Kwh = 24). So kann auf jeden gewünschten Datensatz mittels eines Schlüssels einzeln zugegriffen werden.

2.2.2. Kwl-Schlüssel

Weiterhin gibt die CDBase Anwendern die Möglichkeit, ihre eigenen Angaben zu den gewünschten Daten zu präzisieren. Dieses geschieht mittels des optionalen Kwl-Schlüssels, einem maximal fünfstelligen Wert.

Für jeden Kwh-Wert - also jeden angesprochenen Datensatz - stehen mehrere Kwl-Schlüssel zur Verfügung, die jeweils eine eigene Datenstruktur zum Auslesen gewünschter Daten vorgeben.

Da dieses zur Folge hat, dass eine sehr große Gliederungsstruktur entsteht, soll hier im Folgenden beispielhaft eine Datenstruktur für Netz-Knoten - in diesem Falle mit den Werten Kwl = 20 und Kwh = 00 - aufgezeigt werden. Zu finden ist diese Auflistung wiederum in der cdbase.chm [3] unter SOFiSTiK Data => Kwh-Wert => Auflistung der Kwl-Werte.

```
@Rec: 020/00 NODE | Nodes :V200501
@0= NR [int]     | node-number          |Knotennummer
@1# INR [int]    |internal node-number |interne Knotennummer
@2# KFIX [int]   |degree of freedoms    |Freiheitsgrade
```

```
@3# NCOD [int]   |additional bit code  |Zusatz bit code

@1: XYZ 3[1001] |X-Y-Z-ordinates      |XYZ-Ordinaten
```

Code 3: Vorgegebene Datenstruktur der Netzknoten für Kwh/Kwl = 20/00 [3]

In den unter Code 3 aufgeführten Zeilen ist nun eine Datenstruktur beschrieben, mit deren Hilfe die aufgeführten Daten ausgelesen werden können. Auf diese Datenstrukturen wird im folgenden Kapitel 2.3 näher eingegangen.
Insgesamt ergibt sich durch Kombination von Kwh- und Kwl-Schlüssel die von der SOFiSTiK AG definierte Schreibweise: Kwh/Kwl (xxx/xxxxx).
Das Prinzip der Kwl-Schlüssel kann jedoch auch anderweitig verwendet werden. Mittels des zweiten Schlüssels ist es möglich, Daten innerhalb eines Kwh-Datensatzes individuell zu markieren. Dieses wird im Besonderen bei den Geometrie-Datensätzen genutzt.
Bei den Geometrie-Daten übernehmen die Kwl-Werte aufgrund der deutlich geringeren Elemente-Anzahl die Aufgabe, die einzelnen Geometrieelemente zu nummerieren. Dies bedeutet, dass jedes Geometrieelement einen eigenen Kwl-Wert erhält und somit einzeln angesprochen werden kann. Auf diese Kennzeichnung wird in der Folge im Kapitel 4.4.1.2 näher eingegangen.

2.3. Datenstrukturen

Wie bereits in Kapitel 2.2 beschrieben, werden die Daten aus der CDBase in vorgegebene Datenstrukturen eingelesen.
Aus der in Kapitel 2.2.2 dargestellten Datenstruktur für die Netz-Knoten-Basisdaten (20/0) lassen sich folgende Variablen mit den zugehörigen Datentypen auflisten:

Name	Datentyp	Erklärung (engl.)	Erklärung (deut.)
NR	[int]	node-number	Knotennummer
INR	[int]	internal node-number	interne Knotennummer
KFIX	[int]	degree of freedoms	Freiheitsgrade
NCOD	[int]	additional bit code	Zusatz bit code
XYZ [3]	[float]	X-Y-Z-ordinates	XYZ-Ordinaten

Tabelle 1: Liste der Variablen in der Datenstruktur 20/0 [Eigene Anfertigung]

Für die Programmiersprache C++ können diese Datenstrukturen nun aufgegriffen werden, sie müssen jedoch entsprechend umformuliert werden. So lässt sich für das obige Beispiel die Struktur folgendermaßen definieren:

```
// ---- Netz-Knoten-Datenbasis 20/0 ----
typedef struct CNET_NODE
      {
       int  m_NR;                  //    |node-number
       int  m_INR;                 //    |internal node-number
       int  m_KFIX;                //    |Freedom-Degrees
       int  m_NCOD;                //    |additional Bit-Code
       float  m_XYZ[3];            //    |X-Y-Z-ordinates
       } CNET_NODE;
```

Code 4: Umformulierte Datenstruktur der Netzknoten (20/00) für C++ [Eigene Anfertigung]

Die oben dargestellte Datenstruktur wurde der Datei SOFiSTiK_Data.h entnommen, die im Zuge dieser Arbeit für den Parser angefertigt wurde. In dieser Header-Datei sind einige weitere Datenstrukturen, die im Rahmen der vorliegenden Arbeit verwendet wurden, aufgeführt.
Von der SOFiSTiK AG wird parallel das Header-File cdbtypegeo.h angeboten, in welchem alle Datenstrukturen, die benötigt werden, enthalten sind. Das File SOFiSTiK_Data.h dient in diesem Kontext nur zu Anschauungszwecken.

2.4. Besonderheiten im Umgang mit der CDBase

Die index-sequentielle Struktur der CDBase birgt jedoch einige Konsequenzen hinsichtlich des Umgangs mit den Datenstrukturen.
Aus einer CDBase lassen sich keine Einzeldaten, wie z.B. eine bestimmte Koordinate für einen Knoten, sondern immer nur eine komplette Datenstruktur im sequentiellen Verfahren auslesen. Des Weiteren kann nicht direkt auf einzelne Teile der Datenbank zugegriffen werden. So können z.B. die Daten eines Knotens 2 erst nach den Daten des Knotens 1 in die Datenstruktur eingelesen werden. Dieses muss für entsprechende Auslese-Funktionen und vor allem für Zugriffszeiten bedacht werden.

3. CDBase-Zugriff

Um auf eine CBDase-Datenbasis zuzugreifen, sind im Wesentlichen drei Schritte notwendig: Zunächst muss die Datei initialisiert - also aufgerufen bzw. geöffnet werden -, um diese danach auszulesen. Schließlich muss die Datenbasis geschlossen werden, um einen eventuellen Datenverlust vorzubeugen.
Hierzu werden von der SOFiSTiK AG grundlegende Funktionen und Tools im angebotenen Software-Paket zur Verfügung gestellt, die diesen Zugriff ermöglichen.

3.1. Tool dbinfo

Für den manuellen Konsolenzugriff auf eine CDBase gibt es das Tool dbinfo.
Dieses Tool bietet grundlegende Funktionen für den Umgang mit *.cdb-Dateien an, welche in Abbildung 1 dargestellt sind.

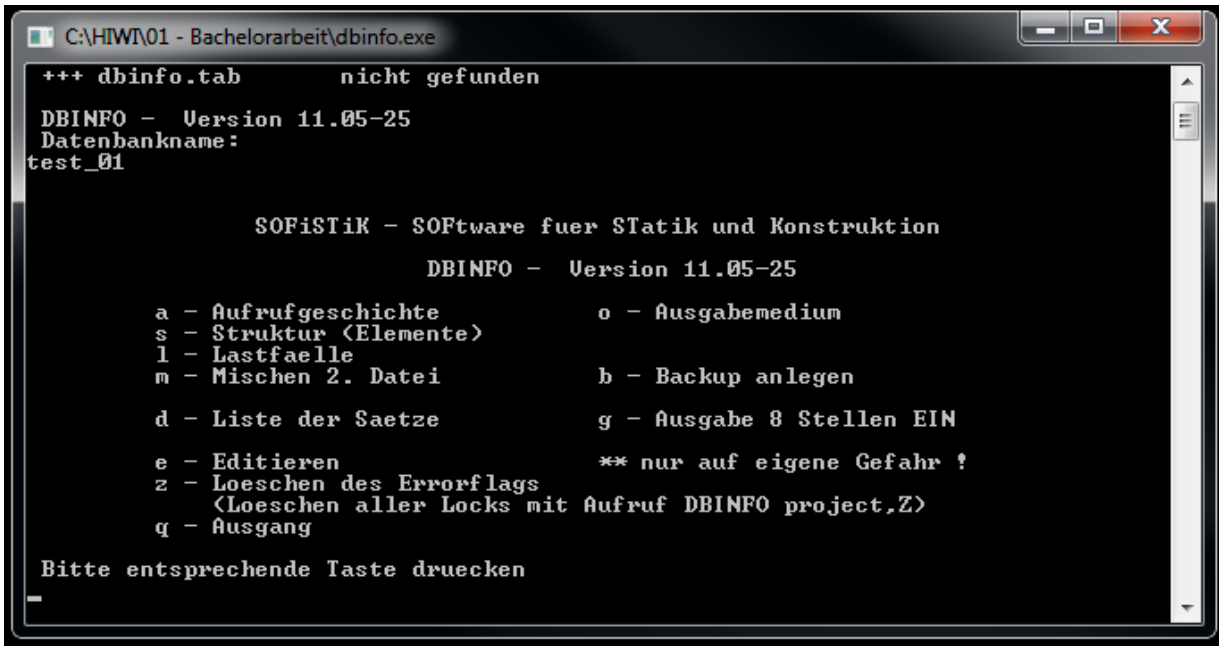


Abbildung 1: Konsolenansicht nach Aufruf einer *.cdb mit dbinfo [Eigene Darstellung]

Mittels dieses Tools kann unter anderem eine Liste der Datensätze, welche in der aufgerufenen CDBase enthalten sind, angezeigt werden. Nach dem Drücken der Taste „d“ erhält man beispielhaft die Konsolenansicht in Abbildung 2.

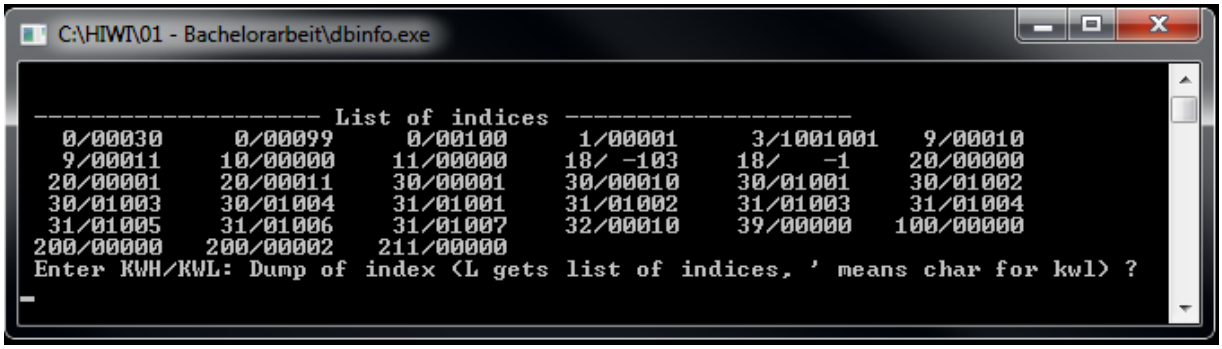


Abbildung 2: Konsolenansicht nach Aufruf der Listensätze mit "dbinfo" [Eigene Darstellung]

In der in Abbildung 2 dargestellten Konsolenansicht sind die in der Datei vorhandenen Datensätze durch Kwl/Kwh Schlüssel angegeben.

Die Funktionsweise dieser Datenschlüssel wurde bereits in Kapitel 2.2 erklärt.

In der Folge werden nach Eingabe eines gewünschten Datenschlüssels in die Konsole die entsprechenden Daten angezeigt (siehe Abbildung 3).

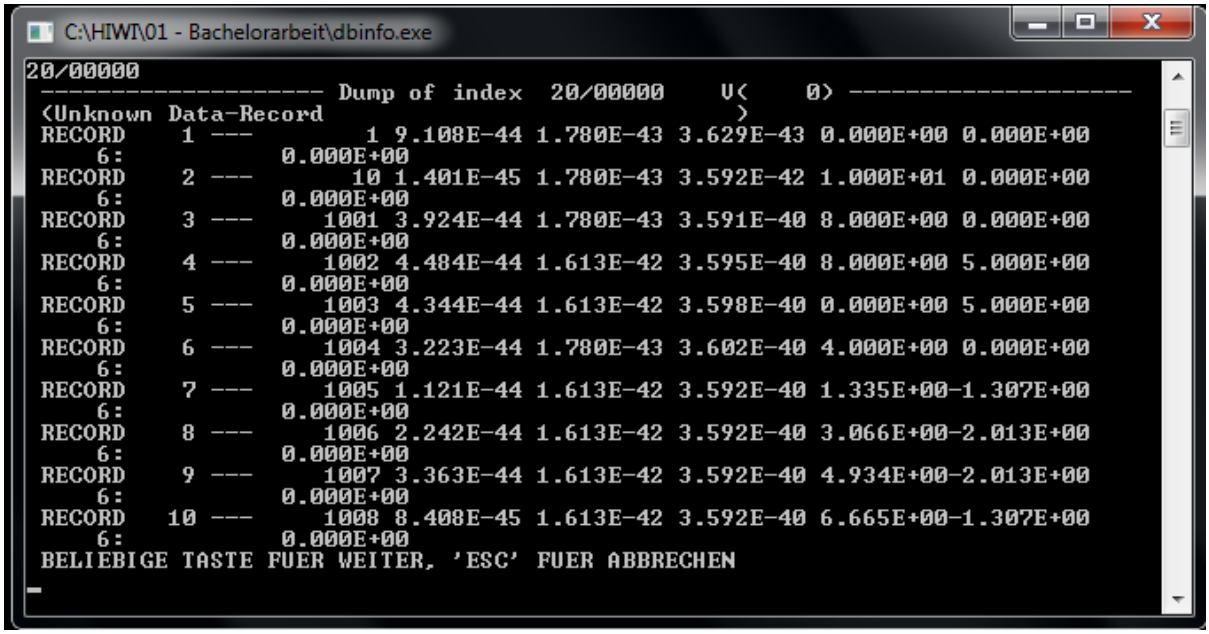


Abbildung 3: Netz-Knoten-Datensatz in der Konsolenansicht mit dbinfo [Eigene Darstellung]

Beim Drücken der Taste „s“ im Hauptfenster erscheint die folgende Auflistung in Abbildung 4.

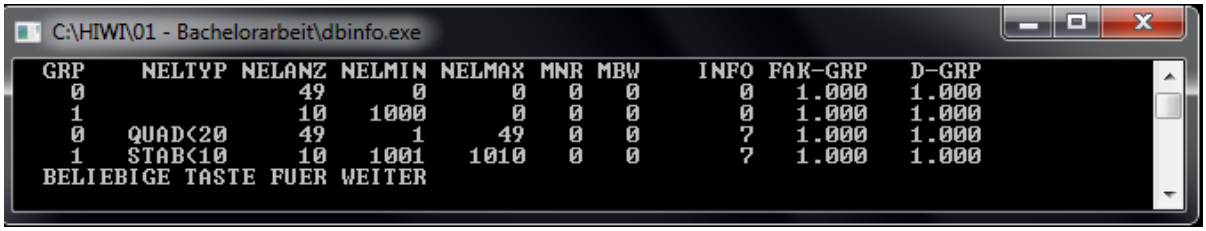


Abbildung 4: Struktur einer CDBase in der Konsolenansicht mit dbinfo [Eigene Darstellung]

Hier sind die vorhandenen Elementtypen einer CDBase mit ihrer Zahl, ihrer minimalen und maximalen Anzahl oder aber ihren zugehörigen Gruppennummern aufgelistet.

dbinfo stellt somit ein hilfreiches Tool dar, mit welchem man schon vor der Funktionsfähigkeit eines Programms bzw. eines Programm-Codes Zugriff auf eine CDBase erhält und so manuell Funktions- und Datenkontrollen durchführen kann.

3.2. Zugriff-Funktionen

Für den tiefer gehenden Zugriff auf die CDBase werden von der SOFiSTiK AG Funktionen zur Verfügung gestellt, mit welchen der Anwender in der eigenen Programmstruktur Zugang zu der Datenbasis erhält. [5]
Im Folgenden sollen diese in der Reihenfolge ihrer Anwendung unter Angabe ihrer Übergabe- und Rückgabeparameter erläutert werden:

3.2.1. sof_cdb_init(Name, InitType)

Die Funktion `sof_cdb_init()` initialisiert eine Datenbank der Endung *.cdb und bereitet diese zum Lesen vor.

Übergabe-Parameter:

Name	[char]	Name der CDBase, die ausgelesen werden soll.
InitType	[int]	Initialisierungs-Art:

- 0: initialisiert die CDBASE.
- >0: öffnet eine CDBase mit dem angegebenen Index.
- 95: öffnet die Datenbasis im “Read-Only“-Modus.
- 94: erstellt eine neue Datenbasis.
- 99: prüft, ob der Name der Datenbasis schon existiert und öffnet diese, falls vorhanden. Andernfalls wird eine neue Datenbasis angelegt.

Rückgabewert:

Icd	[int]	Mit Hilfe des Index kann geprüft werden, ob die CDB erfolgreich angesprochen wurde. < 0 : Fehler 0 : *.cdb erfolgreich initialisiert.

3.2.2. sof_cdb_get(Index, Kwh, Kwl, Data, DataLen, Pos)

Mit der Funktion `sof_cdb_get()` können die gewünschten Daten aus der CDBase in eine vorher erstellte Datenstruktur eingelesen und somit gesichert werden.

Übergabeparameter

Index	[int]	beim Initialisieren zurückgegebener Index.
Kwh/Kwl	[int]	Datenschlüssel (siehe Kap. 2.2)
Data		Datenstruktur, in der die gelesenen Daten gespeichert werden.
DataLen	[int]	ermöglicht Überprüfung, ob die verwendete Datenstruktur, die passende Länge – und somit Größe - hat, um alle Daten auszulesen.
Pos	[int]	Position des aktuell zu lesenden Eintrags.

Rückgabewert

Ie	[int]	ermöglicht die Überprüfung des Datensatzes, der ausgelesen werden soll. • 0: kein Fehler • 1: Daten passen nicht in die Datenstruktur. • 2: das Ende der Datei wurde erreicht. • 3: Fehlermeldung

3.2.3. sof_cdb_put(Index, Kwh, Kwl, Data, RecLen, Pos)

Mit der Funktion `sof_cdb_put()` können die Daten aus einer Datenstruktur in eine CDBase eingelesen und so in dieser gespeichert werden. Damit werden die Daten für die Tools des SOFiSTiK-Software-Pakets zugänglich.

Übergabeparameter

Index	[int]	beim Initialisieren erhaltener Index.

Kwh/Kwl	[int]	Datenschlüssel (siehe Kap. 2.2)
Data		Datenstruktur, in der die gelesenen Daten gespeichert werden.
RecLen	[int]	ermöglicht Überprüfung, ob die verwendete Datenstruktur die passende Länge für das Auslesen der Daten besitzt.
Pos	[int]	Position des aktuell zu lesenden Eintrags.

Rückgabewert

Ie	[int]	ermöglicht die Überprüfung des Datensatzes, der ausgelesen werden soll. • 0: kein Fehler • 1: Daten passen nicht in die Datenstruktur. • 2: das Ende der Datei wurde erreicht. • 3: Fehlermeldung

3.2.4. sof_cdb_close(Index)

Abschließend wird die ausgelesene CDBase mit der Funktion `sof_cdb_close()` geschlossen, so dass es zu keinen Datenverlusten kommen kann. Diese Funktion besitzt keinen Rückgabewert.

Übergabeparameter:

Index	[int]	mit dem Index = 0 wird das System geschlossen.

4. Parser-Funktion in TUM.GeoFrame

Um die in einer CDBase enthaltenen Daten zugänglich zu machen, wurde im Rahmen dieser Arbeit ein Parser in der Programmiersprache C++ erstellt. Im Folgenden sollen zunächst das Projekt TUM.GeoFrame und anschließend Aufbau und Funktionsweise des CDBase-Parsers erklärt werden.

4.1. TUM.GeoFrame

Wie schon im Kapitel 1.1 beschrieben, handelt es sich bei dem Programm TUM.GeoFrame um ein automatisiertes Tool im Bereich der Finite-Elemente-Berechnungen, mit dem Elemente im high-order-Bereich erstellt werden können.[7] Zunächst soll nun näher auf Struktur und Aufbau des Programmes eingegangen werden.

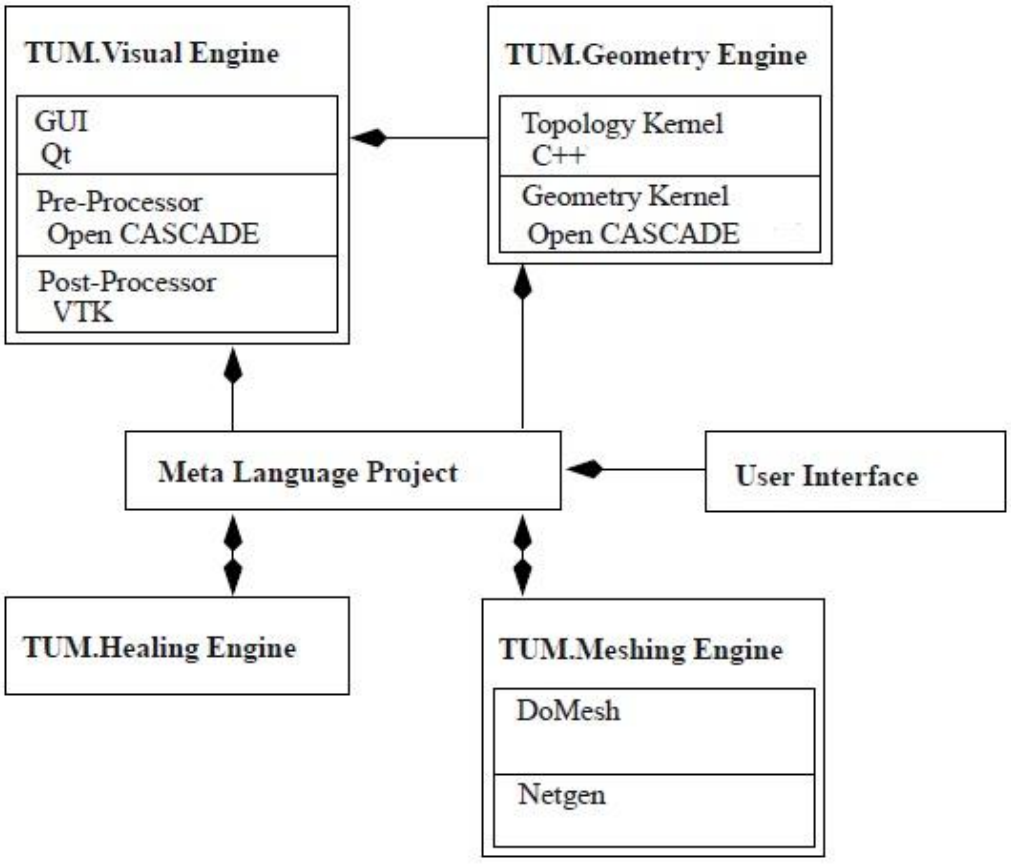


Abbildung 5: Aufbau und Struktur von TUM.GeoFrame [7]

Grundsätzlich enthält TUM.GeoFrame – wie in Abbildung 5 dargestellt - drei für den Parser relevante Teile: die VisualEngine (GUI), die MeshingEngine (Netzerstellung) und die GeometryEngine (Topologie- und Geometry-Kern).

Einsatzbereich des Parsers soll es sein, dem Benutzer in der VisualEngine eine Import- und Export-Funktion zu Verfügung stellen (TUM.VisualEngine), eine CDBase über den Geometrie- und Topologiekern einzulesen (TUM. GeometryEngine), um schließlich eigene Netz-Berechnungen im „high-order“-Bereich anzufertigen (TUM. MeshingEngine).
Der Parser arbeitet hierbei vorwiegend im Bereich der GeometryEngine, und so soll dieser Teil nachfolgend näher betrachtet werden.
Grundlegend besteht die TUM.GeometryEngine aus einem an dem CiE-Lehrstuhl entwickelten Topologie- und einem auf der Open-Source-Geometriebibliothek OpenCASCADE basierenden Geometrie-Kern.

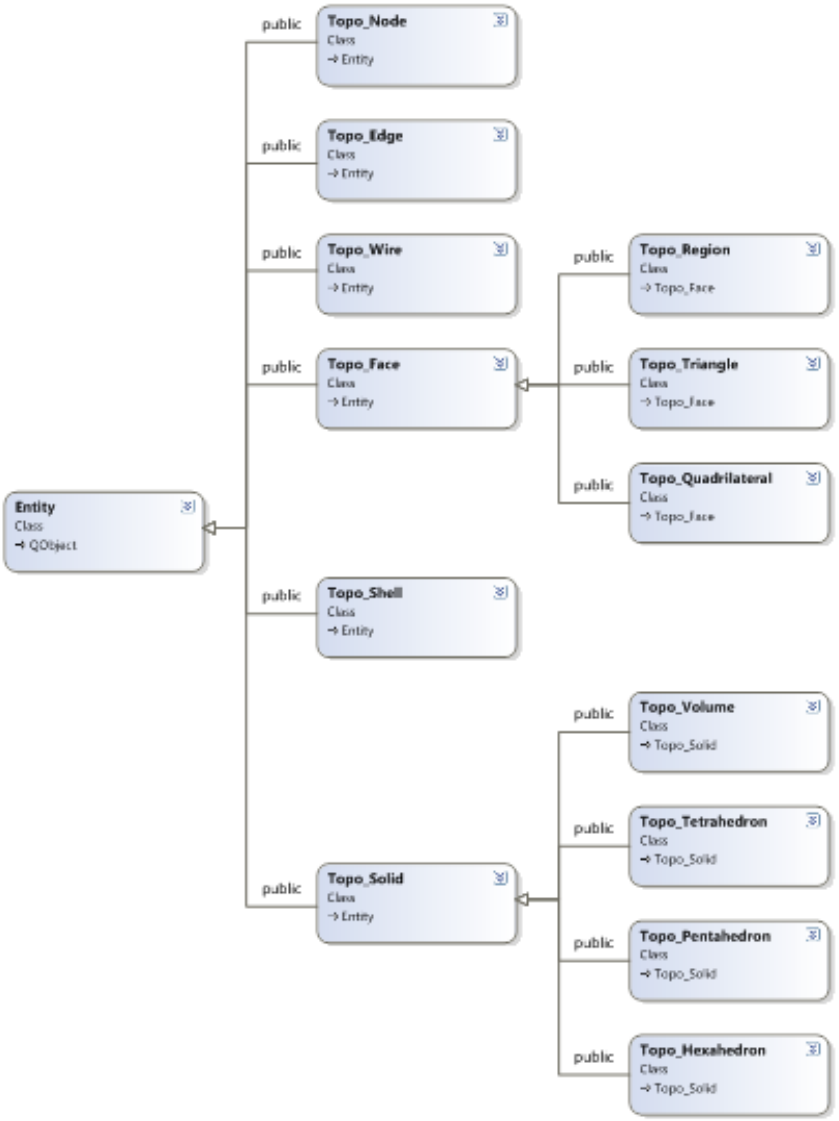


Abbildung 6: Aufbau des Topologiekerns von TUM.GeoFrame [Eigene Anfertigung]

In Abbildung 6 ist die Klassenhierarchie des Topologiekerns der TUM.GeometryEngine aufgeführt. Sämtliche dieser Klassen, die je nach Geometrie bestimmte Parameter besitzen, gehen auf die Klasse Entity zurück, die wiederum Basis-Parameter zur Verfügung hat. Im Prinzip muss der Parser nun lediglich die Daten aus der CDBase an diese Klassen des Topologiekerns weitergeben.

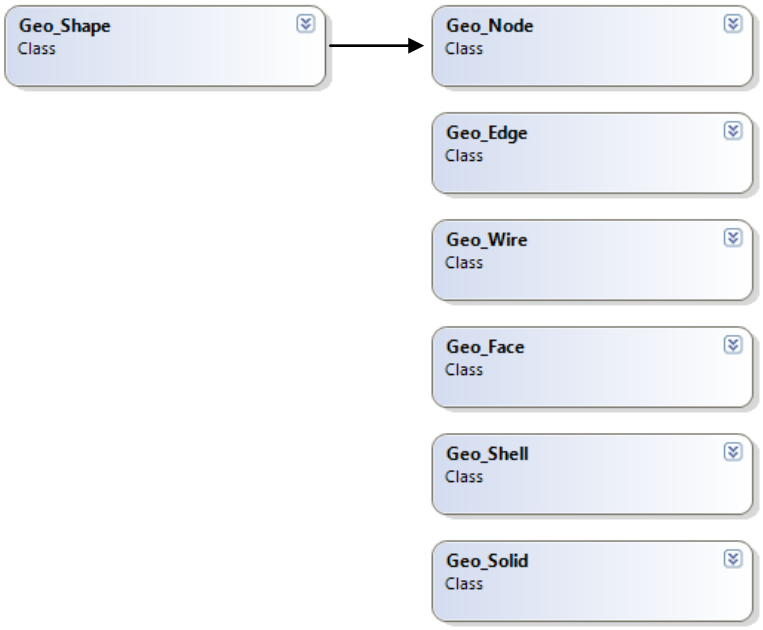


Abbildung 7: Aufbau der Geometrie von TUM.GeoFrame [Eigene Anfertigung]

Mit der Übergabe der Daten an die Topologienklassen befinden sich nun die Daten im Topologiekern von TUM.GeoFrame. Um diese Daten zu visualisieren, müssen sie in den Geometrie-Kern gelangen. Dafür ist im Geometriekern eine weitere Klassenhierarchie angelegt.

Sämtliche Geometrie-Klassen gehen auf die Klasse Geo_Shape zurück (siehe Abbildung 7) und erben somit deren Eigenschaften. Je nachdem, um welche Geometrie es sich handelt, müssen die Topologie-Klassen nun in passende Geometrien-Klassen gewandelt werden, damit sie abschließend mittels des Geometrie-Kerns OpenCASCADE dargestellt werden können. Hierfür sind in TUM.GeoFrame entsprechende Funktionen eingebaut, die zwischen Geometrie- und Topologiekern interagieren. Bei gekrümmten Geometrien, wie z.B. Kreisbögen oder NURBS (Non-Rational Uniform B-Splines), bedient sich der Topologiekern sogenannter Attribute. Ist also beispielweise eine Kante gekrümmt, so ist sie zunächst

grundsätzlich in ihrer Geometrie - durch Start- und Endpunkt definiert – eine gerade Kante. Die Information, dass eine Krümmung vorliegt muss der Topologieklasse zusätzlich übergeben werden. An dieser Stelle muss der Geometrie ein Attribut hinzugefügt werden, welches die Kante als gekrümmt kennzeichnet.

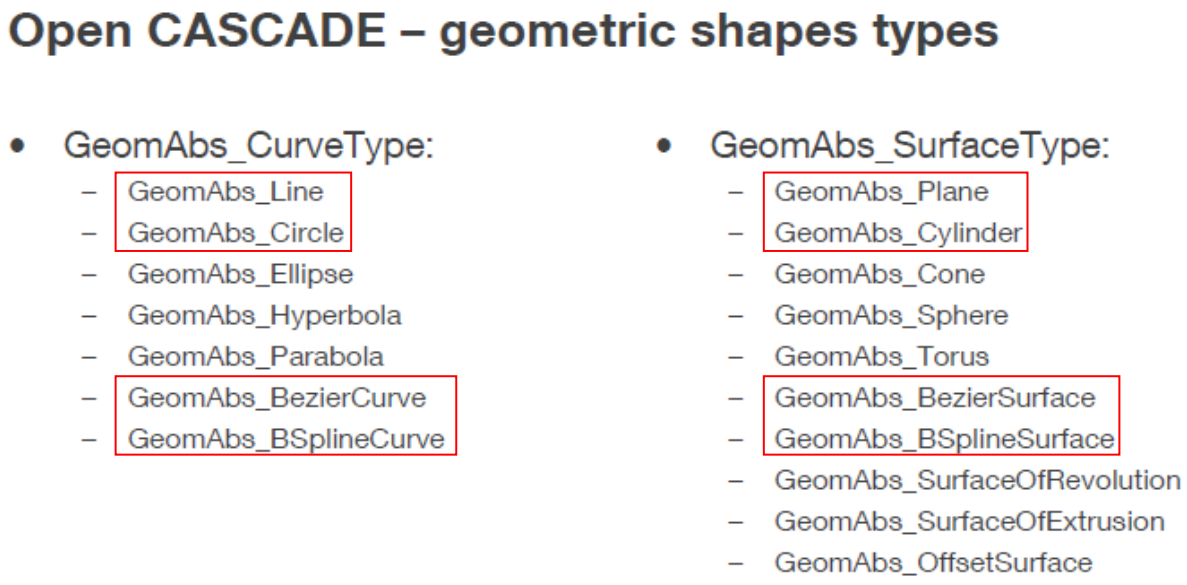


Abbildung 8: Darstellungsarten von OpenCASCADE*

In Abbildung 8 sind die verschiedenen Darstellungstypen und –klassen aufgeführt, welche OpenCASCADE nutzt, um Geometrien zu visualisieren. Mittels der Geometrie-Klassen in TUM.GeoFrame, welche schließlich die Daten aus der CDBase enthalten, und der aufgeführten Visualisierungsmöglichkeiten ist OpenCASCADE in der Lage eine Geometrie darzustellen. Diese Darstellung wird abschließend an das Projekt-Fenster von TUM.GeoFrame weitergeleitet.

Der Parser wird letztlich die in der Abbildung markierten Klassen von OpenCASCADE verwenden. Da diese Implementierung sehr aufwändig ist, werden weitere Klassen im Rahmen der vorliegenden Arbeit nicht behandelt. Allerdings lassen sich über die Geometrie-Klassen BSpline- und Bezier-Kurve bzw. –Fläche viele weitere Formen genau bestimmen, was für eine Visualisierung von komplexen Geometrien ebenfalls ausreicht und auf diesem Weg ermöglicht.

* internes Diagramm des Lehrstuhls „Computation in Engineering“ der TU München

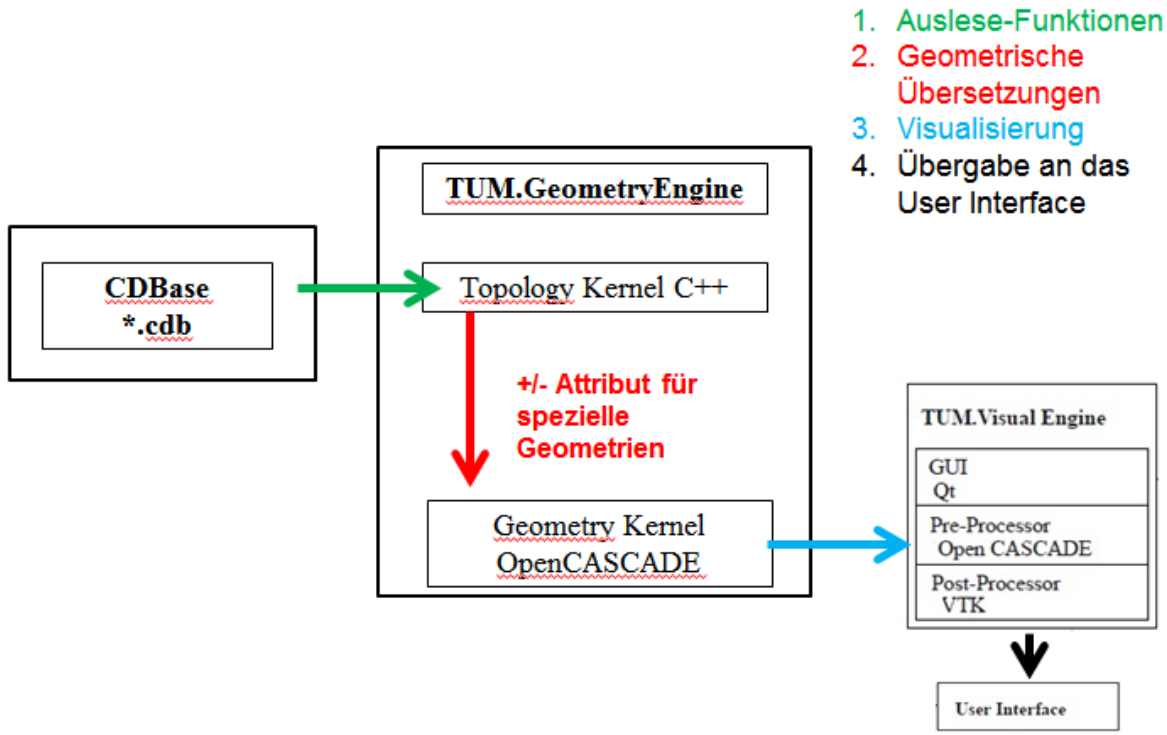


Abbildung 9: Grundidee für das Auslesen der CDBase [Eigene Darstellung]

Abschließend lässt sich der „Weg“ der CDBase-Daten wie in Abbildung 9 darstellen.

4.2. Voraussetzungen

Für die Programmierung des Parsers für die Datenbasis *.cdb ist es notwendig, von der SOFiSTiK AG bereitgestellte Bibliotheken zu implementieren. In das Projekt TUM.GeoFrame wurden folgende Bibliotheken inkludiert:

***.h-Dateien:**	***.lib-Dateien:**	***.dll-Dateien:**
`cdbase.h`	`QtCore_sofistik27_d4.lib`	`QtCore_sofistik27_d4.dll`
`cdbtypeall.h`	`cdbdc27.lib`	`sofdc27w.dll`
`cdbtypegeo.h`		

Tabelle 2: Auflistung der inkludierten Dateien [Eigene Anfertigung]

Zunächst ist eine allgemeine Implementierung der SOFiSTiK-Software in das Framework erforderlich, was mittels der `sofdc27w.dll` geschieht. Die Funktionsweisen und Deklarationen der CDBase werden durch die statische Bibliothek `cdbdc27.lib` inkludiert.

Das Framework TUM.GeoFrame ist mit Hilfe der beiden Klassenbibliotheken Qt (Nokia) und VTK (Visualization Toolkit) programmiert worden, was eine Anpassung der SOFiSTiK-Software erforderlich macht. Die Kompatibilität wird mittels der statischen und dynamischen Bibliotheksdateien „`QtCore_sofistik27_d4`“ (.lib und.dll) hergestellt.
In den weiteren Header-Dateien, insbesondere in der `cdbase.h`, sind die in Kapitel 3.2 aufgeführten Zugriffsfunktionen und deren notwendige Typenspezifikationen enthalten. In den Header-Dateien `cdbtypegeo.h` und `cdbtypeall.h` sind vorgefertigte Datenstrukturen und -typen gespeichert, auf die bereits im Kapitel 2.3 eingegangen wurde. [4, 5]

4.3. Funktionsweise

Grundlegend wurde der Parser in Form von Import- und Export-Funktionen in TUM.GeoFrame integriert. Dieses hat den Hintergrund, dass für Parser-Funktionen eine entsprechende Programmierung im Projekt bereits vorhanden ist und der Parser somit eingebettet werden kann.
Bei Im- und Export der CDBase in TUM.GeoFrame wird im Folgenden zwischen Netz- und Geometriedaten unterschieden. Die deutlich komplexere Datenstruktur bei der Speicherung von Geometrie-Daten macht eine solche Unterscheidung erforderlich.

4.4. Datenstrukturen

Wie im Kapitel 2.3 beschrieben ist es für den Zugriff auf die CDBase zunächst notwendig, Datenstrukturen zu definieren. Zur Veranschaulichung wurden in der Header-Datei SOFiSTiK_data.h eigenangefertigte Datenstrukturen gespeichert und so in die DataBase von TUM.GeoFrame integriert. Diese beschränken sich deshalb stark auf das Auslesen von Netz-Daten.
Alle weiteren Datenstrukturen werden im Folgenden aus dem Header-File `cdbtypegeo.h` bezogen. (siehe Kapitel 2.3)

4.4.1. Import einer CDBase

4.4.1.1. Auslesen des Netzes

Zunächst soll die Funktionsweise einer Ausleseschleife für die CDBase an einem Grundmodell erklärt werden, bevor auf weitere Details eingegangen wird.

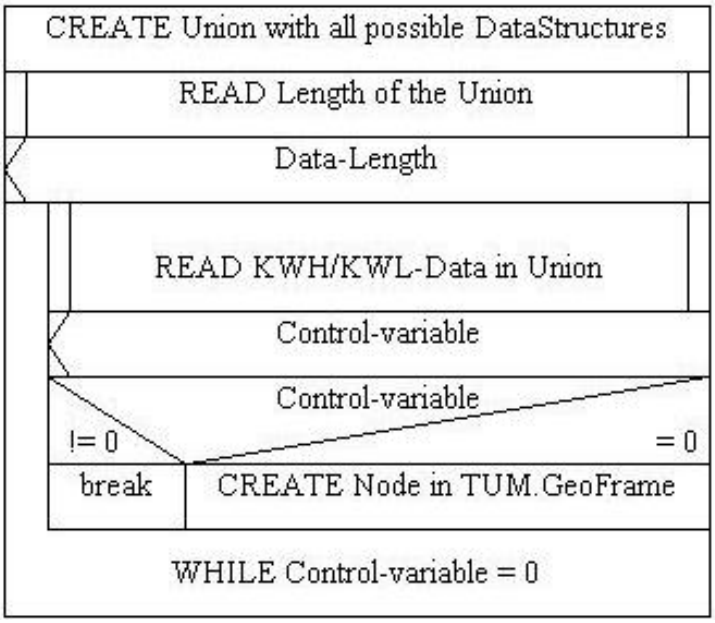


Abbildung 10: Struktogramm der Einlese-Schleife [Eigene Anfertigung]

Wie in Abbildung 10 dargestellt, wird zunächst eine Union erstellt, welche sämtliche Datenstrukturen enthält, die Netz-Knotenpunkte beschreiben können. Wahl und Funktionsweise der Datenstrukturen wurden ausführlich in Kapitel 2 beschrieben, darüber hinaus kann eine Auflistung der Datenstrukturen der cdbase.chm [3] entnommen werden.
Anschließend wird mit dem Befehl „sizeof“ die Datenlänge der größten Datenstruktur in der Union ermittelt. Die erhaltene Datenlänge wird bei der folgenden Auslesefunktion zur Kontrolle weiterverwendet.
Mittels der Funktion sof_cdb_get() kann nun der gewählte Kwh/Kwl-Datensatz in die Union eingelesen werden. Das bedeutet, dass jede der enthaltenen Datenstrukturen auf der jeweiligen Position den gleichen Wert erhält. Ausgabeparameter ist eine Kontrollvariable, mittels welcher der Ausleseprozess kontrolliert werden kann. Die Kontrollvariable kann die Werte von 0 bis 3 annehmen und kann somit in einer Do-While-Schleife überprüft werden. Je nachdem, welchen Wert die Kontrollvariable annimmt, gibt sie Auskunft über den Erfolg oder Fehler des Auslesens der CDBase (siehe Kapitel 3.2.2).
Nimmt nun diese Kontrollvariable nicht den Wert 0 an, so muss der Prozess unterbrochen werden, und die Schleife wird verlassen. Erfolgt der Prozess jedoch ohne Einschränkungen, kann nun mit den gegebenen Daten eine Netz-Topologie in TUM.GeoFrame erstellt werden.

Im Prinzip muss vor dem Erstellen der Topologieklasse eine Verzweigung erfolgen, da noch nicht geklärt ist, aus welcher Datenstruktur die Daten für die Geometrie bezogen werden

sollen. Da jedoch sämtliche Netz-Daten aufgrund ihrer einfachen und linearen Struktur mittlerweile mit nur einer Datenstruktur ausgelesen werden können und es nur bei sehr alten Daten zu Komplikationen kommen kann, darf an dieser Stelle auf die Verzweigung verzichtet werden. Bei den später folgenden Geometrie-Daten wird diese Unterscheidung zwischen den Datenstrukturen jedoch zwingend benötigt, weshalb diese bei den Netzknoten, um Vollständigkeit zu gewährleisten, ebenfalls aufgeführt wird.

Abschließend wird in der Schleife nun die Netz-Topologie in TUM.GeoFrame erstellt. Hierzu werden die geometrischen Parameter aus der Datenstruktur ausgelesen und in eine neu erstellte Topologieklasse von TUM.GeoFrame eingelesen.

4.4.1.2. Auslesen der Geometrie

Bei dem Auslesen der Geometrie muss das oben verwendete Grundmodell geändert werden, da die Geometrie-Datenspeicherung in der CDBase abweicht. Im Folgenden soll das Auslesen an Hand eines weiteren Struktogrammes (s. Abbildung 11) erläutert werden.

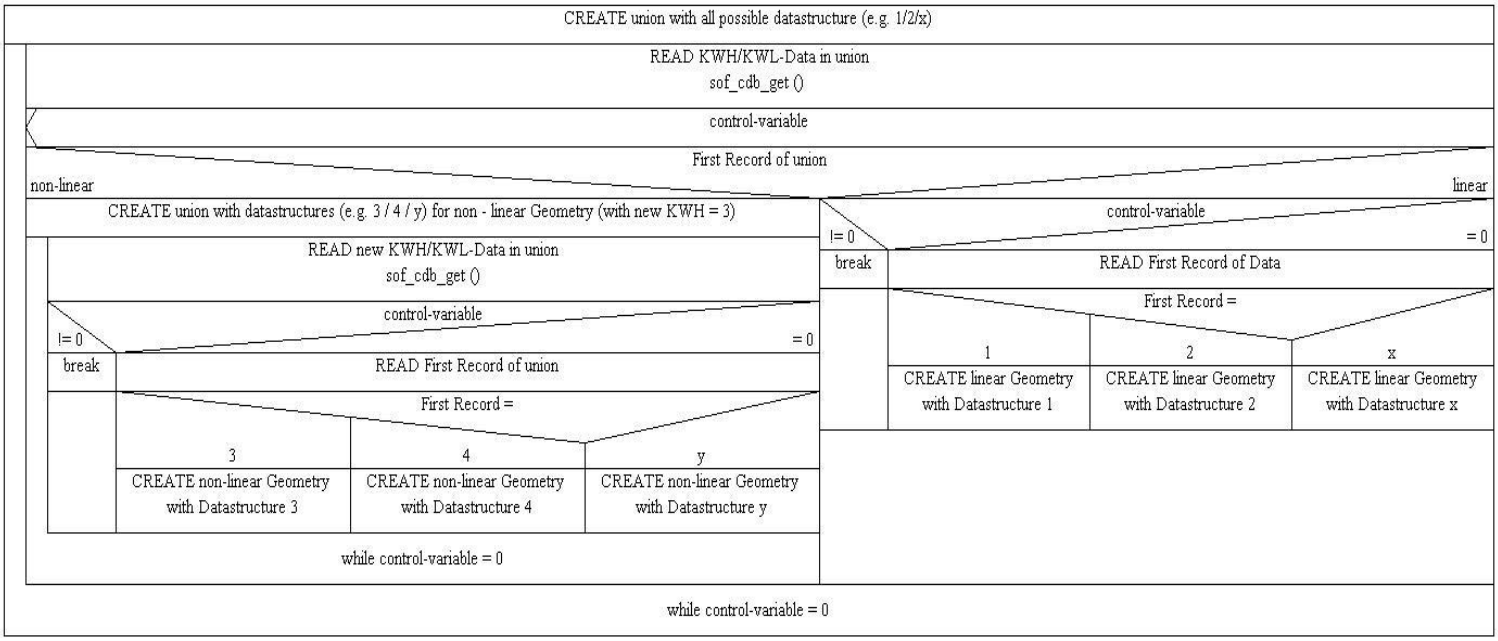


Abbildung 11: Struktogramm der Geometrie-Auslese-Schleife [Eigene Anfertigung]

Wie schon im Kapitel 4.4.1.1 erwähnt kommt bei dem Auslesen der Geometrie im Besonderen das Prinzip des Auslesens in eine Union zum Tragen. Deswegen wird zunächst wiederum eine Union mit mehreren möglichen Datenstrukturen erstellt. Je nachdem, um welchen Geometrie-Datensatz es sich handelt, müssen die Union und die zugehörigen

Datenstrukturen angepasst werden. Entsprechende Informationen sind wiederum der cdbtypegeo.h oder der cdbase.chm zu entnehmen. [3] In dem in Abbildung 11 dargestellten Struktogramm sind beispielhaft drei Datenstrukturen (1/2/x) angegeben.
Anschließend wird wie gehabt die Datenlänge der größten Datenstruktur ausgelesen, um diese in der Folge bei der Auslesefunktion `sof_cdb_get()` zu verwenden.

Danach folgt ein wichtiger Schritt, welcher die Netz- von den Geometriedaten deutlich voneinander unterscheidet. Bei den Netzdaten wird die Datenstruktur, die für die gewünschten Variablen zu wählen ist, mittels des Kwl-Wertes identifiziert. Diese Funktion übernimmt bei Geometrie-Datensätzen der erste Record des Datensatzes, während der Kwl-Wert bei Geometrien die ID darstellt. Das hat den Vorteil, dass auf diese Weise Datensätze mit gekrümmten Geometrien, wie z.B. Kreisbögen, leicht gefunden werden können.
Nimmt nun der der erste Record nach dem Auslesen in die Union eine besondere Eigenschaft an, wie z.B. einen negativen Wert, dann ist dies der Hinweis auf eine gekrümmte Geometrie.
Deswegen muss in der Folge eine Verzweigung eingefügt werden, mittels welcher zwischen einem speziellen ersten Record und einem herkömmlichen Wert unterschieden wird. Als Beispiel ist in dem Struktogramm die Unterscheidung zwischen linearen und nicht-linearen Geometrien gemacht worden. Eine detaillierte Auflistung der verschiedenen Eigenschaften, die der erste Record annehmen kann, ist wiederum der cdbase.chm zu entnehmen. [3]
Beim Auslesen der Geometrien wird des Weiteren die Funktion `sof_cdb_kenq_ex()` verwendet, die im Grunde der vorher verwendeten `sof_cdb_get()` entspricht, jedoch zusätzlich besetzte Kwl-Datenblöcke identifiziert, indem sie automatisch den Kwl-Wert auf den nächsten besetzten Datenblock der CDBase setzt und so die Arbeitszeit der Schleife minimiert.
Bei Kwl-Datensätzen mit linearen Geometrien werden wie gehabt die beladenen Datenstrukturen ausgelesen und in Geometrien des Topologiekerns von TUM.GeoFrame gespeichert. Hier ist darauf zu achten, dass die Datenstrukturen – im Gegensatz zu den Netz-Daten - nun unterschieden werden müssen. Mittels des ersten Records in der Geometrie-Datenstruktur, der bei allen hierfür reserviert ist, kann die anzuwendende Datenstruktur identifiziert werden. Ist der erste Eintrag beispielsweise m_ID = 1, muss der Datensatz mit der zugehörigen Datenstruktur 1 ausgelesen werden. Die entsprechende Zuordnung ist ebenfalls in der cdbase.chm aufgeführt. [3]

Handelt es sich jedoch um eine spezielle Geometrie, muss eine weitere Ausleseschleife integriert werden. Die Datensätze für gekrümmte Geometrien sind nicht unter den bis jetzt ausgelesenen Geometrie-Datensätzen zu finden, sondern sie müssen über einen weiteren Kwh-Schlüssel angesprochen und ausgelesen werden. Wie in der Auflistung der Kwh-Schlüssel im Code 1 in Kapitel 2.2.1 zu sehen ist, befinden sich diese Daten im Datenblock Kwh = 3.
Das Prinzip des Auslesens bleibt allerdings dasselbe, und so wird wiederum eine Do-While-Schleife integriert, die die gleiche Grundstruktur wie die herkömmliche Ausleseschleife besitzt. In dieser Schleife können nun die Daten der gekrümmten Geometrien in entsprechende Datenstrukturen eingelesen werden und müssen schließlich in ein Attribut für die Geometrie des Topologiekerns in TUM.GeoFrame gewandelt werden. Die Unterscheidung zwischen den benötigten Datenstrukturen, mit welchen der Satz ausgelesen werden muss, erfolgt wiederum über den ersten Record des Datensatzes mit dem neuen Kwh-Wert.

Beispiele für die Übersetzung von geometrischen Daten:

Viele geometrische Daten sind in der CDBase mit Parametern gespeichert, die nicht ohne weiteres in Geometrien des Topologiekerns für TUM.GeoFrame geladen werden können. Hier ist eine geometrische Übersetzung notwendig, die im Folgenden an Hand von Kreisbögen und NURBS beispielhaft aufgezeigt werden soll.

Einige Kreisbögen werden in der CDBase nur unter Angabe von Kreismittelpunkt, einer Kreistangente und einer Kreisachse gespeichert. TUM.GeoFrame besitzt für diese Parameter keine passende Funktion, um die Daten direkt in den Topologiekern einzulesen und dann entsprechend als Geometrie zu laden. In diesem Falle ist eine Übersetzung der Daten notwendig. Ein entsprechender Überblick zu der geometrischen Problemstellung ist in Abbildung 12 dargestellt.

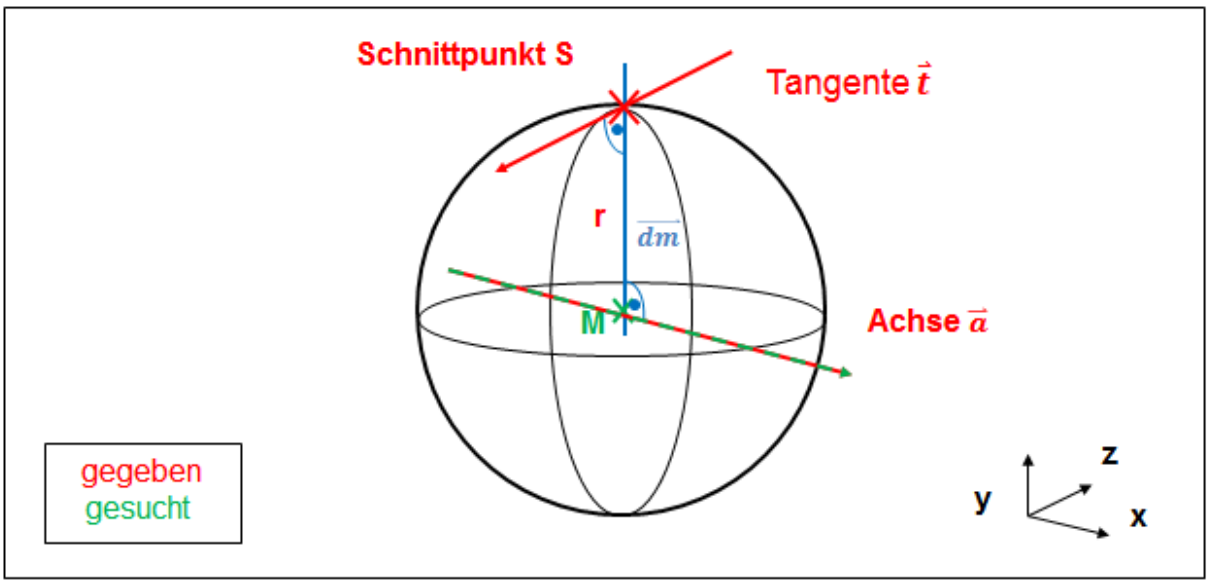


Abbildung 12 : Kreisbogen-Übersetzung [Eigene Darstellung]

Aus den bekannten geometrischen Parametern können nun alle weiteren Variablen berechnet werden:
Mittels des Kreuzproduktes aus der gegebenen Tangente T und der gegebenen Achse des Kreises, die senkrecht zueinander stehen, kann ein Vektor bestimmt werden, der von dem Schnittpunkt der Tangente mit dem Kreisbogen durch den Mittelpunkt verläuft. Multipliziert man diesen Vektor vom Schnittpunkt aus mit dem gegebenen Radius in Richtung des Mittelpunktes, erhält man die Koordinaten des Mittelpunktes M des Kreises.
Im Code sieht der Vorgang folgendermaßen aus:

```
// Berechnung Mittelpunkt des Kreises
      // Deklariere neuen Vektor
      double dm[3];
      // Kreuzprodukt von Tangente und Achse
      for (int i=0; i < 3; i++)
      dm[i] = t [(i+1)%3] * axis[(i+2)%3] - t [(i+2)%3] * axis[(i+1)%3];
      // Deklariere Mittelpunkt
      double M[3];
      // Mittelpunkt-Koordinaten aus Multiplikation von Radius mit dem
      Richtungsvektor dxyz1[3]
      for (int i=0; i < 3; i++)
      m[i] = S[i] - radius * dm[i];
```

Code 5 : Kreisbogen-Übersetzung – Berechnung des Mittelpunktes [Eigene Anfertigung]

Bei dem Laden der NURBS ergibt sich das Problem, dass die CDBase und TUM.GeoFrame die Vektoren der Knotenpunkte der NURBS unterschiedlich definieren.

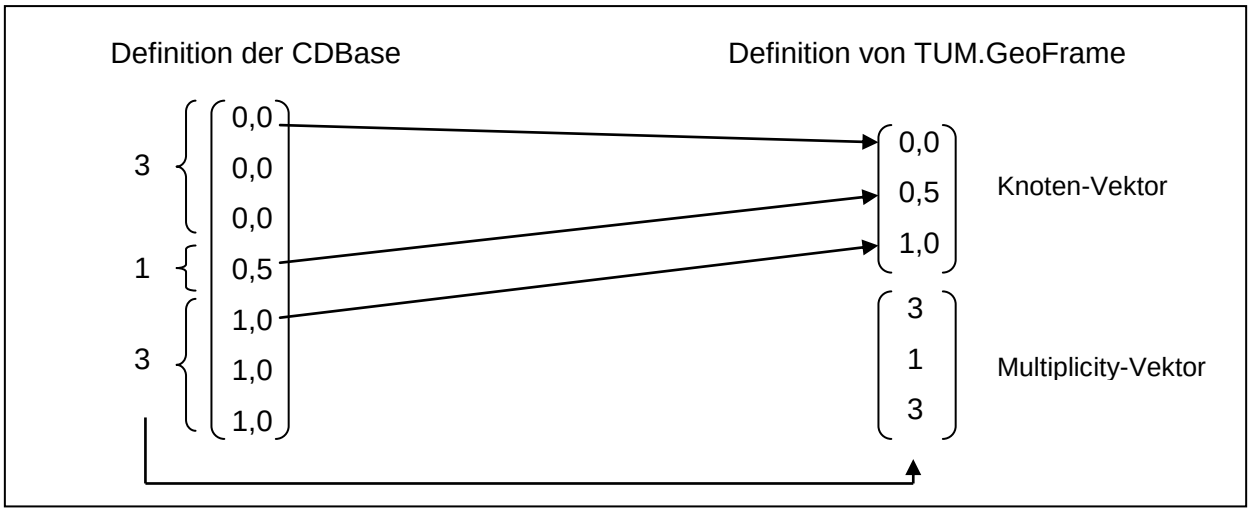


Abbildung 13: Übersetzung von NURBS [Eigene Darstellung]

Wie in Abbildung 13 dargestellt nutzt die CDBase einen einzelnen Vektor, indem dieser an Hand von Koordinaten und der Anzahl ihrer folgenden Wiederholungen im Vektor beide Daten bereitstellt. Für die Verwendung in TUM.GeoFrame sind jedoch zwei Vektoren notwendig, die diese Daten jeweils getrennt angeben. Also muss dieser Vektor zwischen beiden Programmen übersetzt werden.

Zusätzlich fordert TUM.GeoFrame – im Gegensatz zu der CDBase - als Definition, dass der erste und der letzte Eintrag des Multiplicity-Vektors, die den Grad des jeweiligen NURBS darstellen, um 1 erhöht werden. So ergibt sich schließlich als Vektor (4, 1, 4)T.

Im Programm-Code sieht dieses folgendermaßen aus:

```
// Import knot and multiplicity vector
int n_knots = 1;
// knot vector
int* indices = new int[array_size];
// multiplicity vector
int* mults = new int[array_size];
indices[0] = 0;
mults[0] = 1;
for (int i=1; i < array_size; i++)
{
      // Check if next knot has the same value
      if (abs(knots[i-1] - knots[i]) < EPS)
      {
            //
            mults[i] = mults[i-1] + 1;
            indices[n_knots-1] = i;
      }
      else
```

```
                {
                        n_knots++;
                        mults[i] = 1;
                        indices[n_knots-1] = i;
                }
        }
        // first an end record of multiplicity vector + 1
        Mults->SetValue(1, degree + 1);
        Mults->SetValue(n_knots, degree + 1);
```

Code 6 : Kreisbogen-Übersetzung – Berechnung des Mittelpunktes [Eigene Anfertigung]

4.4.2. Export einer CDBase

Bei dem Einlesen von Daten in eine CDBase ergibt sich dasselbe Problem wie bei dem Auslesen. Auch hier muss zwischen den Netz- und Geometriedaten unterschieden werden, da sich die Geometriedaten deutlich komplexer verhalten. Im Rahmen der vorliegenden Arbeit wird nur das Exportieren der Netz-Daten behandelt.

4.4.2.1. Einlesen des Netzes

Neben der Import-Funktion soll ebenfalls das Exportieren von *.cdb-Dateien in der Arbeitsumgebung von TUM.GeoFrame ermöglicht werden. Hierfür muss die Auslese-Schleife im Prinzip umgedreht werden.

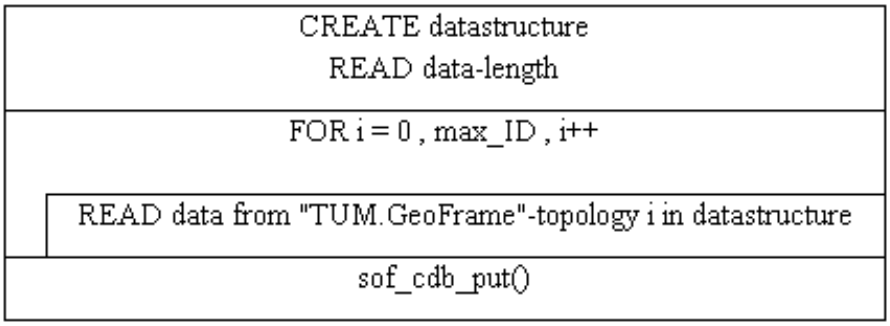


Abbildung 14: Struktogramm Einlese-Schleife der Netz-Daten [Eigene Anfertigung]

Zunächst wird – wie in Abbildung 14 dargestellt - eine Datenstruktur zu der jeweiligen Geometrie, von der die Variablen und Parameter vorhanden sind, erstellt. In diese Datenstruktur müssen nun die in TUM.GeoFrame geladenen Daten des Netzes gespeichert werden. Hierzu wird eine for-Schleife verwendet, die sämtliche Netz-Geometrien, die in TUM.GeoFrame geladen wurden, entlangläuft.

Anschließend wird die erstellte Datenstruktur mit den vorhandenen Parametern der jeweiligen Netz-Geometrie gefüllt. Da sich in der Datenbank mehr Variablen befinden können als in der jeweiligen Topologie-Klasse enthalten sind, muss im Besonderen darauf geachtet werden, dass restliche Variablen mit Standardwerten besetzt werden, da diese sonst die folgende Weiterverwendung stark beeinträchtigen können. Abschließend wird die vollständig beladene Datenstruktur mittels der Funktion `sof_cdb_put()` in einer neuerstellten CDBase-Datenbasis gespeichert.

4.5. Testlauf des Parsers

Als Testlauf für die neue Import- und Exportfunktion wurden testweise mehrere *.cdb Dateien, die sich in ihrer Geometrie und Fülle deutlich voneinander unterscheiden, in TUM.GeoFrame eingelesen. Die folgenden Darstellungen sollen einen Überblick über die Übersetzungsmöglichkeiten von Geometrien zwischen beiden Programmen geben. Die nachfolgenden Abbildungen dokumentieren CDBbase-Dateien, die zunächst im SOFiSTiK Animator dargestellt werden und anschließend in TUM.GeoFrame geladen werden. In Abbildung 19 ist überdies eine exportierte CDBase wiederum im SOFiSTiK Animator dargestellt.

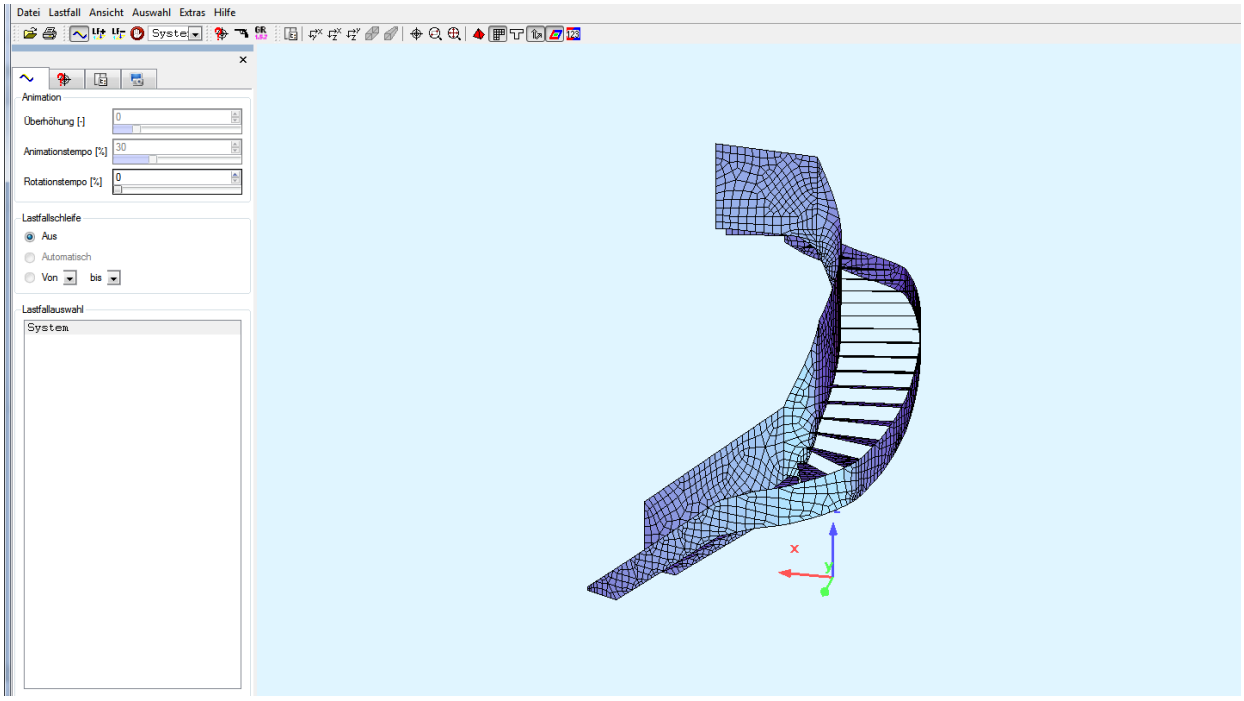


Abbildung 15: Geometrie einer Treppe mit dem SOFiSTiK Animator [SOFiSTiK Animator]

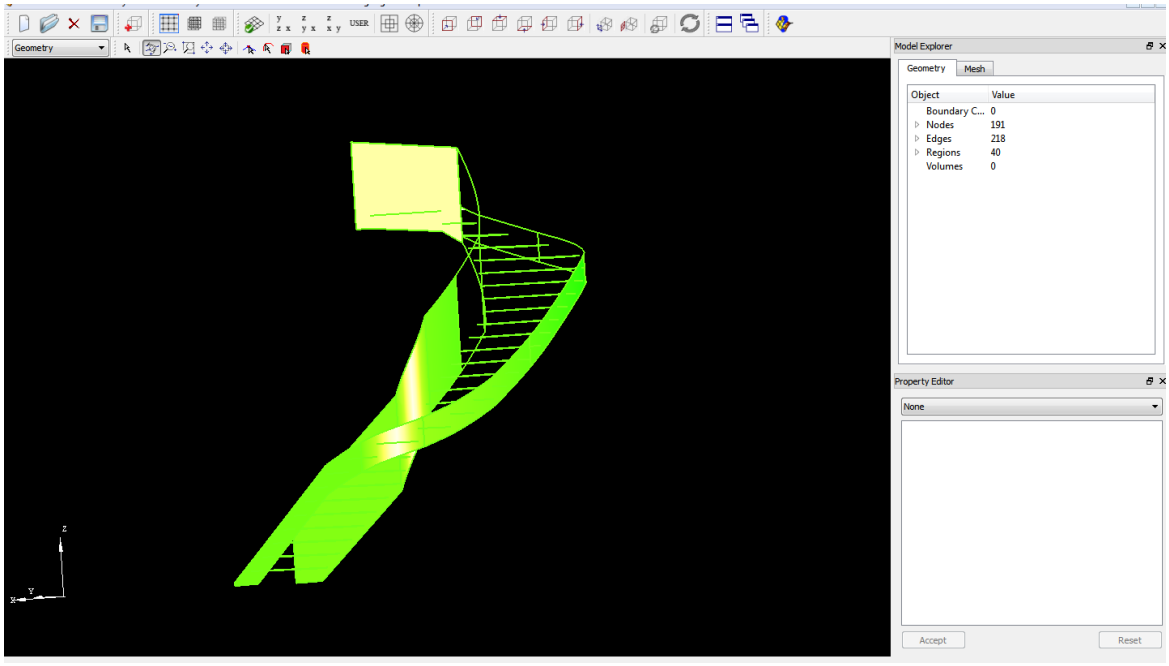


Abbildung 16: Geometrie einer Treppe in TUM.GeoFrame [Eigene Darstellung]

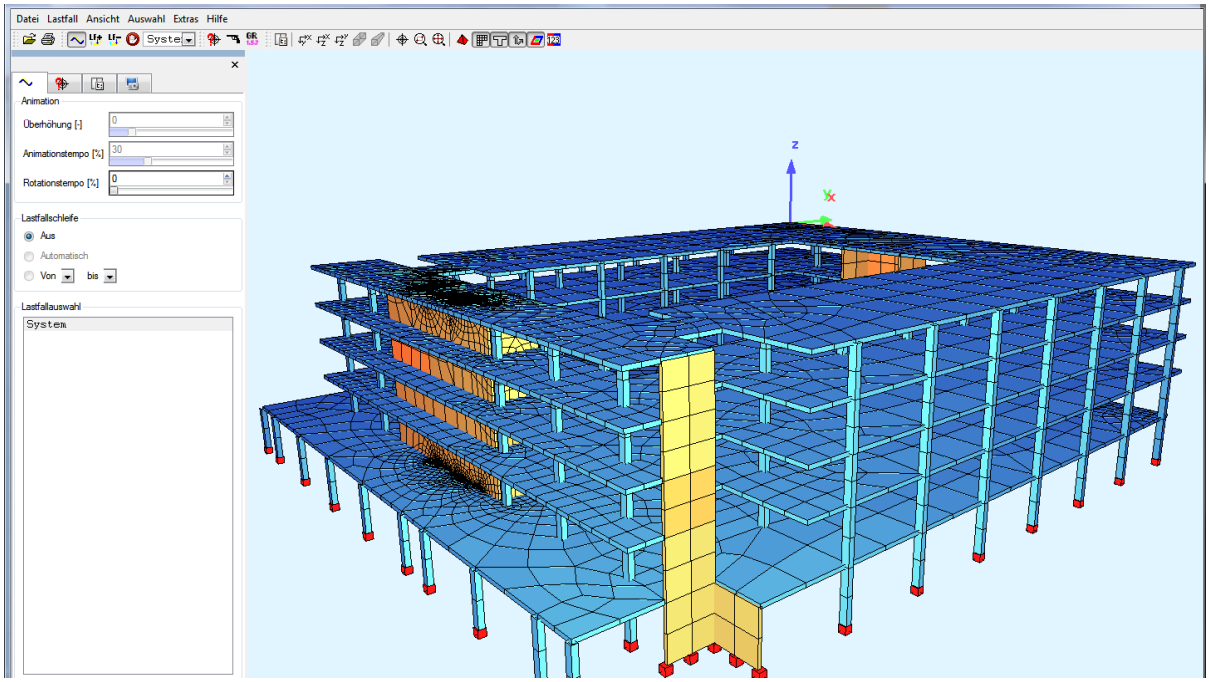


Abbildung 17: Parkhaus im SOFiSTiK Animator [SOFiSTiK Animator]

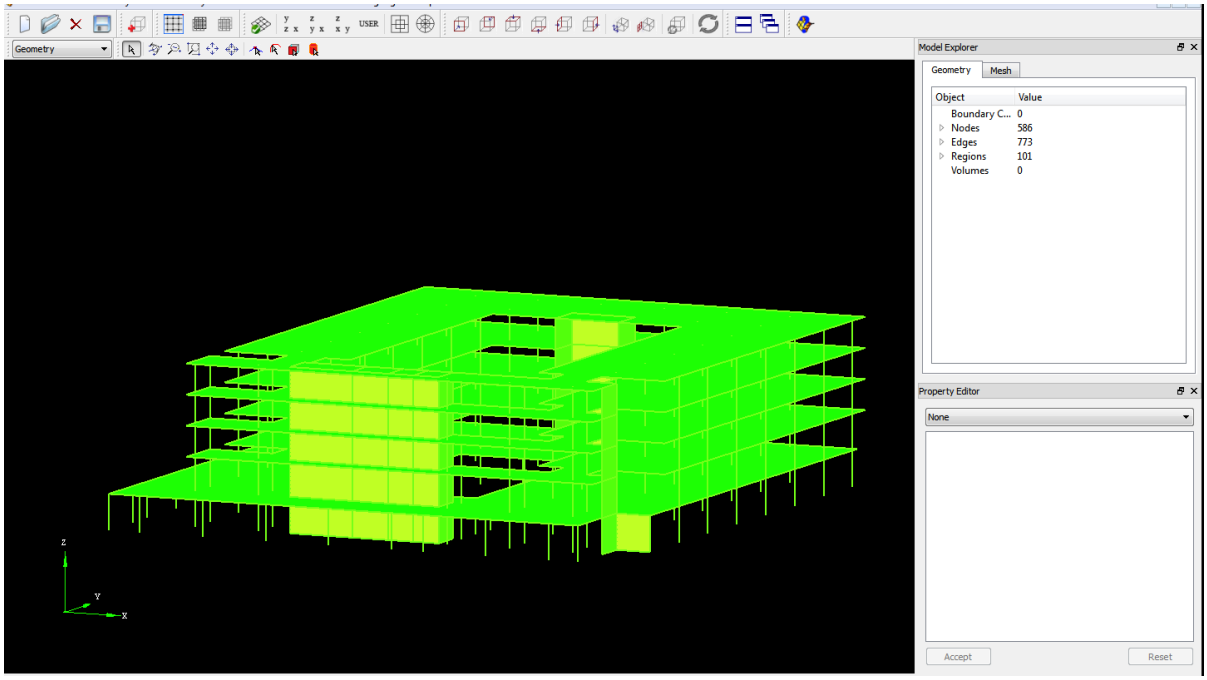


Abbildung 18: Darstellung des Parkhauses in TUM.GeoFrame [Eigene Darstellung]

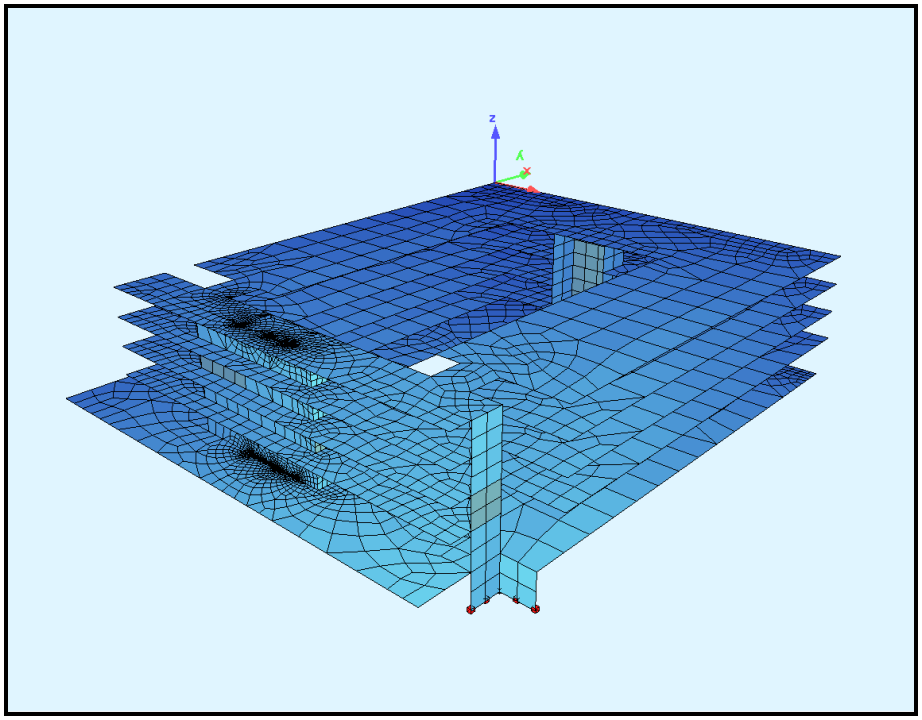


Abbildung 19: Export des Parkhauses [SOFiSTiK Animator]

5. Zusammenfassung und Ausblick

In der vorliegenden Arbeit werden neben einer grundlegenden Erklärung von Struktur und Aufbau der CDBase der SOFiSTiK AG funktionsfähige Import- und Export-Funktionen für das Framework TUM.GeoFrame zur Verfügung gestellt, die prinzipiell auch leicht in andere Programme integriert werden können. Mit den in der vorliegenden Arbeit erstellten Parser-Funktionen ist der Grundstein für eine weitgehende Kompatibilität zwischen TUM.GeoFrame und der CDBase der SOFiSTiK AG gelegt worden. Sie stellen ein grundlegendes Muster dar, an welchem folgende Funktionen orientiert werden können, um Export und Import zu vervollständigen und so die Funktionsvielfalt zwischen beiden Programmen zu vergrößern. Auf diese Weise wird es möglich, geometrische Daten einer CDBase in TUM.GeoFrame einzulesen und diese auch darzustellen. So können nun sämtliche Netz-Daten, Geometriekanten, -flächen, -volumina sowie spezielle Geometrien, wie Kreisbögen und NURBS, dargestellt werden. Bislang sind die Übertragungsmöglichkeiten noch auf diese ausgewählten geometrischen Daten begrenzt. Die Datenbasis CDBase enthält eine Fülle von Daten, die ebenfalls in TUM.GeoFrame übertragen werden könnten. Um diese Daten jedoch in TUM.GeoFrame zu laden, sind weitere Übersetzungsfunktionen notwendig, damit abweichende Geometrien oder aber Definitionen angepasst werden können.
Der nächste Schritt in der Entwicklung einer erweiterten Kompatibilität von der CDBase und TUM.GeoFrame wäre die Vervollständigung sämtlicher geometrischer Daten und der anschließende Austausch der konstruktiven Daten.

6. Literaturverzeichnis

[1] Lehrstuhl "Computation in Engineering" Technische Universität München. *Prof. Dr. Casimir Katz*. Abgerufen am 22. 1 2012 von http://www.cie.bv.tum.de/de/ueber-uns/mitarbeiter/katz

[2] Qt / Nokia Corporation. (2012). *Qt 4.7.1.* Abgerufen am 29. Januar 2012 von Qt / Nokia Corporation: qt.nokia.com/products

[3] SOFiSTiK AG. "cdbase.chm" (Version 16.15-25). Oberschleißheim.

[4] SOFiSTiK AG. (2010). Basisfunktionalitäten - Version 2010. Oberschleißheim.

[5] SOFiSTiK AG. *SOFiSTiK FEA Version 2012 SOFiCAD 2012 18.2.* Abgerufen am 29. 01 2012 von www.sofistik.de/studenten

[6] SOFiSTiK AG. *SOFiSTiK Homepage*. Abgerufen am 18. Januar 2012 von http://www.sofistik.de/sofistik/

[7] Sorger, C., Frischmann, F., Kollmannsberger, S., & Rank, E. (22. Januar 2012). TUM.GeoFrame: Automated high-order hexahedral mesh. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*.

[8] Visualization Toolkit. (2012). *VTK 5.8.0.* Abgerufen am 29. Januar 2012 von www.vtk.org/VTK/resources/software.html

7. Weitere Verzeichnisse

7.1. Abbildungsverzeichnis

7.2. Tabellenverzeichnis

7.3. Code-Verzeichnis

8. Anhang

Im Anhang ist der komplette Programmcode der Parser-Funktionen enthalten:

```
#ifndef  SOFiSTiK_data_HEADER
#define SOFiSTiK_data_HEADER
/************************************************************************
 ****                                                                ****
 ****    Header:     SOFiSTiK_data.h                                 ****
 ****                                                                ****
 ************************************************************************
 ****    Author:     Cornelius Preidel                               ****
 ************************************************************************/

/************************************************************************
 **** SOFiSTiK-Datenstrukturen                                       ****
 ************************************************************************/
     // ---- Allgemeine *.cdb Daten ----
     typedef struct FILEDATA
           {
            int  iprob;                          //     |Type of System  | Art des Systems
            int  iachs;                          //     |Orientation of gravity
            int  nknot;                          //     |Number of nodes     |Anzahl Knoten
            int  mknot;                          //     |Highest node number |Hoechste Knotennr
            int  igdiv;                          //     |Group divisor       |Gruppendivisor
            int  igres;                          //     |degrees of freedom and other options
            float xmin,ymin,zmin,xmax,ymax,zmax;     //     |bounding box  |umhüllende Box
            float xref,yref,zref,tglob[3][3];        //     |global CDB-System transformation matrix
           } struc10;
     // ---- Netz-Knoten 20/00 ----
     typedef struct CNET_NODE
           {
            int  m_NR;                   //          |node-number
            int  m_INR;                  //      |internal node-number
            int  m_KFIX;                 //          |Freedom-Degrees
            int  m_NCOD;                 //          |additional Bit-Code
            float  m_XYZ[3];             //          |X-Y-Z-ordinates
            } CNET_NODE;
     // ---- Netz-Quads 200/00 ----
     typedef struct CNET_QUAD
           {
            int  m_NR;                   //'        elementnumber
            int  m_NODE[4];              //'        nodenumbers
            int  m_MAT;                  //'        materialnumber
            int  m_MRF;                  //'        material Reinf.
            int  m_NRA;                  //         type of element
            float      m_DET[3];         //'        Parameter of Jacobi Determinant
            float      m_THICK[5];       //' (1010) element thickness
            float      m_CB;             //'        bedding factor
            float      m_CQ;             //'        tangential bedding factor
            float      m_T[9];           //         transformation matrix
            } CNET_QUAD;

     // ---- Geometrie-Knoten ----
     typedef struct CGEO_NODE
           {
           // ---- @Rec: 030/NR:0 ----
            int  m_ID;                   //          |identifier 0
            int  m_IBC;                  //          |Boundary Conditions
            float      m_XYZ[3];         //          |Coordinates of that point
            float      m_T[9];           //          |Transform-Matrix
            float      m_BX;             //          |Dimension in local x
            float      m_BY;             //          |Dimension in local y
            float      m_THICK;          //          |Plate thickness at that point
            float      m_CURV[2];        //          |Curve [x,y]
            char m_TEXT[32];             //          |Name
            } CGEO_NODE;
     // ---- Geometrie-Kanten ----
     typedef struct CGEO_EDGE
           {
                // ---- @Rec: 031/NR:?? ----
           int   m_ID;                   //          identifier 0
           int   m_PT[2];                //          Number start & end point
           int   m_NOG;                  //          Groupnumber
           int   m_MAT;                  //          Materialnumber for bedding
           int   m_IBC;                  //          Boundary condition to be suppressed or
           float       m_W;              //    (1001) width of support
           float       m_CA;             //    (1096) axial     bedding
           float       m_CQ;             //    (1096) lateral   bedding
           float       m_CM;             //    (1099) torsional bedding
           float       m_CX;             //    (1096) global X  bedding
           float       m_CY;             //    (1096) global Y  bedding
           float       m_CZ;             //    (1096) global Z  bedding
           float       m_CXX;            //    (1099) global X  tors.bed.
           float       m_CYY;            //    (1099) global Y  tors.bed.
           float       m_CZZ;            //    (1099) global Z  tors.bed.
           float       m_HMESH;          //    (1001) mesh density of edge (first record only)
           float       m_DRX;            //    local X direction
           float       m_DRY;            //    local Y direction
           float       m_DRZ;            //    local Z direction
           float       m_I_14;           //
           float       m_I_15;           //
           char  m_TEXT[32];             //    title of Line
            } CGEO_EDGE;
```

```
      // ---- Geometrie-Flächen ----
      typedef struct CGEO_REGION
            {
            // ---- @Rec:  032/NR:1,2,5,6 ----
             int  m_ID;             //          |identification
             int  m_NG;             //          |edgeno     (31/NG) | Kantennr   (31/NG)
             int  m_NA;             //          |startpoint (30/NA) | Startpunkt (30/NA)
             int  m_NB;             //          |endpoint   (30/NB) | Endpunkt   (30/NB)
             int  m_NM;             //          |midpoint   (30/NM) | Mittelpunkt(30/NM)
             int  m_IFC;            //          |Interface hinge bit pattern

             } CGEO_REGION;
      // ---- Geometrie-Volumen ----
      typedef struct CGEO_VOLUMEN
            {

             } CGEO_VOLUMEN;

#endif
```

```
/*************************************************************************
 **** class Topo_Shape : Private member funcion                       ****
 *************************************************************************/
bool Topo_Shape::import_CDB_Geometry(char* fileName)
{
// ----     Initialize cdb (fileName, Index) ----
      int         icd;
      icd = sof_cdb_init(fileName, 99);
// ----     Initialize file-information of the cdb ----
      struct      FILEDATA filedata;
      int         ret_file;
      int         l_file;
       l_file = sizeof(filedata);
      ret_file = sof_cdb_get(icd, 10, 0, &filedata, &l_file, 0);
// ----     Import nodes ----
      int   ie_node, il_node;
      int kwh_node = 30;
      int kwl_node = 0;
      int node_buffer = 0;
      // Loop over node buffers (kwh 30)
      do
      {
            // Import node buffer
            sof_cdb_kenq_ex(icd, &kwh_node, &kwl_node, +1);
            union DATA
            {
                  typeCDB_GPT gpt;
                  tagCDB_GLN  gln;
            };
            // Initialize node data
            int node_id = 0;
            double x, y, z;
            // Loop over buffer data
            do
            {
                  // Import buffer data
                  DATA buf;
                  il_node = sizeof(buf);
                  ie_node = sof_cdb_get(icd, kwh_node, kwl_node, (void*)&buf, &il_node, node_buffer);
                  // End of buffer reached
                  if(ie_node > 1)
                  {
                        // Create node
                        if (node_id != 0)
                        {
                              Topo_Node* pNode = new Topo_Node(node_id, x, y, z);
                              my_Node_Map->add_Entity(pNode);
                        }
                        break;
                  }
            // ---- Import node data ----
                  // Coordinates
                  if (buf.gpt.m_id == 0)
                  {
                        node_id = kwl_node;
                        x = buf.gpt.m_xyz[0];
                        y = buf.gpt.m_xyz[1];
                        z = buf.gpt.m_xyz[2];
                  }
                  // Update node buffer
                  node_buffer++;
            // End: Loop over buffer data
            } while (ie_node <= 1);
      // End: Loop over node buffers (kwh 30)
      } while (kwh_node == 30);
// ----     Import edges ----
      int   ie_edge, il_edge;
      int kwh_edge = 31;
      int kwl_edge = 0;
```

```
int edge_buffer = 0;
// Loop over edge buffers (kwh 31)
do
{
      // Import edge buffer
      sof_cdb_kenq_ex(icd, &kwh_edge, &kwl_edge, +1);
      union DATA
      {
            tagCDB_GLN        gln;        // ID = 0
            tagCDB_GLN_REF    gln_ref;    // ID < 0 // Hinweis auf Kreisbogen oder NURB
            tagCDB_GLN_GEO    gln_geo;    // ID = 1
      };
      // Initialize edge data
      int edge_id = 0;
      Topo_Node* pNodeA = 0;
      Topo_Node* pNodeB = 0;
      Topo_Edge_Attribute* pAttribute = 0;
      // Loop over buffer data
      do
      {
            // Import buffer data
            DATA buf;
            il_edge = sizeof(buf);
            ie_edge = sof_cdb_get(icd, kwh_edge, kwl_edge, (void*)&buf, &il_edge, edge_buffer);
            // End of buffer reached
            if ((ie_edge > 1) || (kwh_edge != 31))
            {
                  // Create edge
                  if (edge_id != 0)
                  {
                        Topo_Edge* pEdge = new Topo_Edge(edge_id, pNodeA, pNodeB, pAttribute);
                        my_Edge_Map->add_Entity(pEdge);
                  }
                  break;
            }
      // ---- Import edge data ----
            // Nodes by Id
            if (buf.gln.m_id == 0)
            {
                  edge_id = kwl_edge;
                  if (!(pNodeA = (Topo_Node*)my_Node_Map->get_ID_Entity(buf.gln.m_pt[0])))
                        return false;
                  if (!(pNodeB = (Topo_Node*)my_Node_Map->get_ID_Entity(buf.gln.m_pt[1])))
                        return false;
            }
            // Nodes by coordinates
            else if (buf.gln.m_id == 1)
            {
                  if (buf.gln_geo.m_gpg == 0)
                  {
                        edge_id = kwl_edge;
                              if (!(pNodeA = (Topo_Node*)my_Node_Map->get_Node_by_gp_Pnt(gp_Pnt(
                                    buf.gln_geo.m_xyz1[0], buf.gln_geo.m_xyz1[1], buf.gln_geo.m_xyz1[2]))))
                              return false;
                                  if (!(pNodeB = (Topo_Node*)my_Node_Map->get_Node_by_gp_Pnt(gp_Pnt(
                                    buf.gln_geo.m_xyz2[0],buf.gln_geo.m_xyz2[1],buf.gln_geo.m_xyz2[2]))))
                              return false;
                  }
            }
            // Geometric attribute for arcs and NURBS
            else if (buf.gln.m_id < 0)
            {
                  // Initialize attribute data
                  int   ie_att, il_att;
                  int kwh_att = 3;
                  int kwl_att = -buf.gln.m_id;
                  int att_buffer = 0;
                  // Loop over attribute buffers (kwh 3)
                  do
                  {
                        // Import attribute buffer
                        union DATA_ATTRIBUTE
                        {
                              tagCDB_AXIS_GEO   axis_geo;   // iln = 1 circle single precision
                              tagCDB_AXIS_NKN   axis_nkn;   // iln = 90  nurbs knots
                              tagCDB_AXIS_CPT   axis_cpt;   // iln = 91  nurbs points
                              tagCDB_GAXD_GEO   gaxd_geo;   // iln = 181 circle double precision
                              tagCDB_GAXD_NKN   gaxd_nkn;   // iln = 190 KNOTS
                              tagCDB_GAXD_CPT   gaxd_cpt;   // iln = 191 CONTROL Points
                        };
                        // Initialize attribute data
                        int att_id = 0;
                        double radius = 0.0;          // .... related to circle
                        double* xyz1 = 0;             // .... related to circle
                        double* t1 = 0;               // .... related to circle
                        double* axis = 0;             // .... related to circle
                        bool is_rational = false;     // .... related to Nurbs/BSpline curves
                        int array_size = 0;           // .... related to Nurbs/BSpline curves
                        int degree = 0;               // .... related to Nurbs/BSpline curves
                        double knots[256];            // .... related to Nurbs/BSpline curves
                        std::vector<double*> points;  // .... related to Nurbs/BSpline curves
                        // Loop over attribute buffer data
                        do
                        {
                              // Import buffer data
                              DATA_ATTRIBUTE buf_att;
```

```
il_att = sizeof(buf_att);
ie_att = sof_cdb_get(icd, kwh_att, kwl_att, (void*)&buf_att, &il_att, att_buffer);
// End of buffer reached
if ((ie_att > 1) || (kwh_att != 3))
{
      // Create attribute
      if (att_id != 0)
      {
      // ---- Circle Attribute ----
            if ((radius > 0.0) && (xyz1) && (t1) && (axis))
            {
                  // compute center point
                  double dxyz1[3];
                  for (int i=0; i < 3; i++)
                        dxyz1[i] = t1[(i+1)%3] * axis[(i+2)%3] - t1[(i+2)%3] *
                              axis[(i+1)%3];
                  double m[3];
                  for (int i=0; i < 3; i++)
                        m[i] = xyz1[i] - radius * dxyz1[i];
                  // Create attribute
                  pAttribute = new Topo_Edge_Attribute_Circle(att_id, radius,
                  gp_Ax2(gp_Pnt(m[0], m[1], m[2]), gp_Dir(axis[0], axis[1], axis[2])));
                  my_Att_Map->add_Entity(pAttribute);
            }
      // ---- BSpline Attribute ----
            else if ((degree > 0) && (array_size > 0))
            {
            // Check for sufficient number of poles
                  int n_poles = array_size - degree + 1;
                  if (n_poles == int(points.size()))
                  {
                        // Import poles and weights
                        TColgp_Array1OfPnt* Poles = new TColgp_Array1OfPnt(1, n_poles);
                        TColStd_Array1OfReal* Weights = new TColStd_Array1OfReal(1,
                              n_poles);
                        for (int i=0; i < n_poles; i++)
                        {
                              double* point = points.at(i);
                              Poles->SetValue(i+1, gp_Pnt(point[0], point[1], point[2]));
                              Weights->SetValue(i+1, point[3]);
                        }

                        // Import knot and multiplicity vector
                        int n_knots = 1;
                        int* indices = new int[array_size];
                        int* mults = new int[array_size];
                        indices[0] = 0;
                        mults[0] = 1;
                        for (int i=1; i < array_size; i++)
                        {
                              if (abs(knots[i-1] - knots[i]) < EPS)
                              {
                                    mults[i] = mults[i-1] + 1;
                                    indices[n_knots-1] = i;
                              }
                              else
                              {
                                    n_knots++;
                                    mults[i] = 1;
                                    indices[n_knots-1] = i;
                              }
                        }
                        // Import knots and multiplicities
                        TColStd_Array1OfReal* Knots = new TColStd_Array1OfReal(1,
                         n_knots);
                        TColStd_Array1OfInteger* Mults = new TColStd_Array1OfInteger(1,
                         n_knots);
                        for (int i=0; i < n_knots; i++)
                        {
                              Knots->SetValue(i+1, knots[indices[i]]);
                              Mults->SetValue(i+1, mults[indices[i]]);
                        }
                        Mults->SetValue(1, degree + 1);
                        Mults->SetValue(n_knots, degree + 1);
                        // Create attribute
                        pAttribute = new Topo_Edge_Attribute_BSplineCurve(att_id,
                         is_rational, degree, n_poles, n_knots, Poles, Knots, Mults,
                         Weights);
                        my_Att_Map->add_Entity(pAttribute);
                        // Delete arrays
                        delete indices;
                        delete mults;
                  }
            }
      }
      // Delete arrays
      if (xyz1)
            delete xyz1;
      if (t1)
            delete t1;
      if (axis)
            delete axis;
      for (int i=0; i < int(points.size()); i++)
      {
            double* point = points.at(i);
            delete point;
```

```
                }
                points.clear();
                // Break loop
                break;
          }
    // ---- Import attribute data ----
          // Circle single precision
          if (buf_att.axis_geo.m_iln == 1)
          {
                if (abs(buf_att.axis_geo.m_gpg) > EPS)
                {
                      att_id = my_Att_Map->max_id() + 1;
                      radius = buf_att.axis_geo.m_gpg;
                      if (!xyz1)
                            xyz1 = new double[3];
                      if (!t1)
                            t1 = new double[3];
                      for (int i=0; i<3; i++)
                      {
                            xyz1[i] = buf_att.axis_geo.m_xyz1[i];
                            t1[i] = buf_att.axis_geo.m_dxyz1[i];
                      }
                }
          }
          // BSpline non rational
          else if (buf_att.axis_geo.m_iln == 5)
          {
                att_id = my_Att_Map->max_id() + 1;
                is_rational = false;
          }
          // BSpline rational
          else if (buf_att.axis_geo.m_iln == 6)
          {
                att_id = my_Att_Map->max_id() + 1;
                is_rational = true;
          }
          // Circle Axis
          else if (buf_att.axis_geo.m_iln == 9)
          {
                if (!axis)
                      axis = new double[3];
                for (int i=0; i<3; i++)
                      axis[i] = buf_att.axis_geo.m_dxyz1[i];
          }
          // Bezier degree
          else if (buf_att.axis_geo.m_iln == 90)
          {
                array_size = (il_att - 2*sizeof(int))/sizeof(float);
                degree = buf_att.axis_nkn.m_deg;
                for (int i=0; i < array_size; i++)
                      knots[i] = buf_att.axis_nkn.m_s[i];
          }
          // Bezier point
          else if (buf_att.axis_geo.m_iln == 91)
          {
                double* point = new double[4];
                point[0] = buf_att.axis_cpt.m_xyz[0];
                point[1] = buf_att.axis_cpt.m_xyz[1];
                point[2] = buf_att.axis_cpt.m_xyz[2];
                point[3] = buf_att.axis_cpt.m_w;
                points.push_back(point);
          }
          // Circle double precision
          else if (buf_att.axis_geo.m_iln == 181)
          {
                if (abs(buf_att.gaxd_geo.m_gpg) > EPS)
                {
                      att_id = my_Att_Map->max_id() + 1;
                      radius = buf_att.gaxd_geo.m_gpg;
                      if (!xyz1)
                            xyz1 = new double[3];
                      if (!t1)
                            t1 = new double[3];
                      for (int i=0; i<3; i++)
                      {
                            xyz1[i] = buf_att.gaxd_geo.m_xyz1[i];
                            t1[i] = buf_att.gaxd_geo.m_dxyz1[i];
                      }
                }
          }
          // NURBS - Knots
          else if (buf_att.axis_geo.m_iln == 190)
          {
                array_size = (il_att - 2*sizeof(int))/sizeof(double);
                degree = buf_att.gaxd_nkn.m_deg;
                for (int i=0; i < array_size; i++)
                      knots[i] = buf_att.gaxd_nkn.m_s[i];
          }
          // NURBS - Control Points
          else if (buf_att.axis_geo.m_iln == 191)
          {
                double* point = new double[4];
                point[0] = buf_att.gaxd_cpt.m_xyz[0];
                point[1] = buf_att.gaxd_cpt.m_xyz[1];
                point[2] = buf_att.gaxd_cpt.m_xyz[2];
                point[3] = buf_att.gaxd_cpt.m_w;
```

```
                                    points.push_back(point);
                                }
                                // Update attribute buffer
                                att_buffer++;
                            // End: Loop over attribute buffer data
                            } while (ie_att <= 1);
                            break;
                      // End: Loop over attribute buffers (kwh 3)
                      } while(kwh_att == 3);
                }
                // Update edge buffer
                edge_buffer++;
          // End: Loop over buffer data
          } while (ie_edge <= 1);
     // End: Loop over edge buffers (kwh 31)
     } while(kwh_edge == 31);
// ----     Import regions ----
     int   ie_reg, il_reg;
     int kwh_reg = 32;
     int kwl_reg = 0;
     int reg_buffer = 0;
     // Loop over region buffers (kwh 32)
     do
     {
          // Import region buffer
          sof_cdb_kenq_ex(icd, &kwh_reg, &kwl_reg, +1);
          union DATA
          {
               tagCDB_GAR        gar;        // id = 0
               tagCDB_GAR_REF    gar_ref;    // id < 0
               tagCDB_GAR_BOUN   gar_boun;   // id = 1 äußere Kante
               tagCDB_GAR_HOLE gar_hole;     // id = 2 Innere Kante
               tagCDB_GAR_CON3 gar_con3;     // id = 3 Zwangskante
               tagCDB_GAR_MESH   gar_mesh;   // id = 9
               tagCDB_GAR_GEO    gar_geo;    // id = 10
          };
          // Break criterion
          if (kwh_reg != 32)
               break;
          // Initialize region data
          Topo_Region* pRegion = new Topo_Region(kwl_reg, 0);
          Topo_Face_Attribute* pAttribute = 0;
          Topo_Wire* pWire = 0;
          int last_node = 0;
          // Loop over buffer data
          do
          {
               // Import buffer data
               DATA buf;
               il_reg = sizeof(buf) ;
               ie_reg = sof_cdb_get(icd, kwh_reg, kwl_reg, (void*)&buf, &il_reg, reg_buffer);
               // End of buffer reached
               if ((ie_reg > 1) || (kwh_reg != 32))
               {
                    // Create region
                    pRegion->set_Attribute(pAttribute);
                    pRegion->get_Boundary()->create_Geo_Wire();
                    for (int i=0; i < pRegion->get_Wire_Vector()->n_Entities(); i++)
                         ((Topo_Wire*)pRegion->get_Wire_Vector()->get_Entity(i))->create_Geo_Wire();
                    pRegion->create_Geo_Face();
                    my_Reg_Map->add_Entity(pRegion);
                    break;
               }
          // ---- Import region data ----
               // Outer boundary
               if (buf.gar.m_id == 1)
               {
                    // Add edge
                    Topo_Edge* pEdge = (Topo_Edge*)my_Edge_Map->get_ID_Entity(buf.gar_boun.m_ng);
                    pRegion->get_Boundary()->add_Edge(pEdge);
               }
               // Inner boundary
               else if (buf.gar.m_id == 2)
               {
                    // Create new hole
                    if (last_node != buf.gar_hole.m_na)
                    {
                         pWire = new Topo_Wire(pRegion->get_Wire_Vector()->max_id() + 1, pRegion, INNER_BOUNDARY_WIRE);
                         pRegion->get_Wire_Vector()->add_Entity(pWire);
                    }
                    // Add edge
                    Topo_Edge* pEdge = (Topo_Edge*)my_Edge_Map->get_ID_Entity(buf.gar_hole.m_ng);
                    pWire->add_Edge(pEdge);
                    last_node = buf.gar_hole.m_nb;
               }
               // Internal structures (CONS)
               else if (buf.gar.m_id == 3)
               {
                    // If internal line
                    if (buf.gar_con3.m_ng > 0)
                    {
                         // Create new line
                         if (last_node != buf.gar_con3.m_na)
                         {
                              pWire = new Topo_Wire(pRegion->get_Wire_Vector()->max_id() + 1, pRegion, INTERNAL_WIRE);
                              pRegion->get_Wire_Vector()->add_Entity(pWire);
```

```
                }
                // Add edge
                Topo_Edge* pEdge = (Topo_Edge*)my_Edge_Map->get_ID_Entity(buf.gar_con3.m_ng);
                pWire->add_Edge(pEdge);
                last_node = buf.gar_con3.m_nb;
        }
        // If internal point
        else if (buf.gar_con3.m_na > 0)
        {
                // Add node
                Topo_Node* pNode = (Topo_Node*)my_Node_Map->get_ID_Entity(buf.gar_con3.m_na);
                pRegion->get_Node_Vector()->add_Entity(pNode);
        }
}
// Geometric attribute for NURBS
else if (buf.gar.m_id < 0)
{
        // Initialize attribute data
        int   ie_att, il_att;
        int kwh_att = 5;
        int kwl_att = -buf.gar.m_id;
        int att_buffer = 0;
        // Loop over attribute buffers (kwh 3)
        do
        {
                // Import attribute buffer
                union DATA_ATTRIBUTE
                {
                        tagCDB_AREA cdb_area;   // iln = 0 general buffer
                        tagCDB_AREA_PTS   cdb_area_pts;// iln = 11 area point of surface
                        tagCDB_GARD_PTS   cdb_gard_pts;// iln = 290 area point of surface
                        tagCDB_GARD_NKU   cdb_gard_nku;// iln = 291 area point of surface
                        tagCDB_GARD_NKV   cdb_gard_nkv;// iln = 292 area point of surface
                };
                // Initialize attribute data
                int att_id = 0;
                double radius = 0.0;    // .... related to cylinder
                double* xyz1 = 0;       // .... related to cylinder
                double* t1 = 0;         // .... related to cylinder
                double* axis = 0;       // .... related to cylinder
                bool is_rational = false;     // .... related to Nurbs/BSpline surface
                int array_size_u = 0;   // .... related to Nurbs/BSpline surface
                int array_size_v = 0;   // .... related to Nurbs/BSpline surface
                int u_degree = 0;       // .... related to Nurbs/BSpline surface
                int v_degree = 0;       // .... related to Nurbs/BSpline surface
                double u_knots[256];    // .... related to Nurbs/BSpline surface
                double v_knots[256];    // .... related to Nurbs/BSpline surface
                std::vector<double*> points;  // .... related to Nurbs/BSpline surface
                // Loop over attribute buffer data
                do
                {
                        // Import buffer data
                        DATA_ATTRIBUTE buf_att;
                        il_att = sizeof(buf_att);
                        ie_att = sof_cdb_get(icd, kwh_att, kwl_att, (void*)&buf_att, &il_att, att_buffer);
                        // End of buffer reached
                        if ((ie_att > 1) || (kwh_att != 5))
                        {
                                // Create attribute
                                if (att_id != 0)
                                {
                                // ---- Circle Attribute ----
                                        if ((radius > 0.0) && (xyz1) && (t1) && (axis))
                                        {
                                                // compute center point
                                                double dxyz1[3];
                                                for (int i=0; i < 3; i++)
                                                        dxyz1[i] = t1[(i+1)%3] * axis[(i+2)%3] - t1[(i+2)%3] *
                                                         axis[(i+1)%3];
                                                double m[3];
                                                for (int i=0; i < 3; i++)
                                                        m[i] = xyz1[i] - radius * dxyz1[i];
                                                // Create attribute
                                                pAttribute = new Topo_Face_Attribute_Cylinder(att_id, radius,
                                                 gp_Ax2(gp_Pnt(m[0], m[1], m[2]), gp_Dir(axis[0], axis[1], axis[2])));
                                                my_Att_Map->add_Entity(pAttribute);
                                        }
                                // ---- BSpline Attribute ----
                                        else if ((u_degree > 0) && (array_size_u > 0) && (v_degree > 0) &&
                                                        (array_size_v > 0))
                                        {
                                                // Check for sufficient number of poles
                                                int n_u_poles = array_size_u - u_degree + 1;
                                                int n_v_poles = array_size_v - v_degree + 1;
                                                if (n_u_poles * n_v_poles == int(points.size()))
                                                {
                                                        // Import poles and weights
                                                        TColgp_Array2OfPnt* Poles = new TColgp_Array2OfPnt(1, n_u_poles,
                                                         1, n_v_poles);
                                                        TColStd_Array2OfReal* Weights = new TColStd_Array2OfReal(1,
                                                         n_u_poles, 1, n_v_poles);
                                                        for (int i=0; i < int(points.size()); i++)
                                                        {
                                                                double* point = points.at(i);
                                                                Poles->SetValue(int(point[4]), int(point[5]),
                                                                 gp_Pnt(point[0], point[1], point[2]));
```

```
                                    Weights->SetValue(int(point[4]), int(point[5]), point[3]);
                              }
                              // Import knot and multiplicity vector - u
                              int n_u_knots = 1;
                              int* u_indices = new int[array_size_u];
                              int* u_mults = new int[array_size_u];
                              u_indices[0] = 0;
                              u_mults[0] = 1;
                              for (int i=1; i < array_size_u; i++)
                              {
                                    if (abs(u_knots[i-1] - u_knots[i]) < EPS)
                                    {
                                          u_mults[i] = u_mults[i-1] + 1;
                                          u_indices[n_u_knots-1] = i;
                                    }
                                    else
                                    {
                                          n_u_knots++;
                                          u_mults[i] = 1;
                                          u_indices[n_u_knots-1] = i;
                                    }
                              }
                              // Import knots and multiplicities - u
                              TColStd_Array1OfReal* U_Knots = new TColStd_Array1OfReal(1,
                                     n_u_knots);
                              TColStd_Array1OfInteger* U_Mults = new TColStd_Array1OfInteger(1,
                                     n_u_knots);
                              for (int i=0; i < n_u_knots; i++)
                              {
                                    U_Knots->SetValue(i+1, u_knots[u_indices[i]]);
                                    U_Mults->SetValue(i+1, u_mults[u_indices[i]]);
                              }
                              U_Mults->SetValue(1, u_degree + 1);
                              U_Mults->SetValue(n_u_knots, u_degree + 1);
                              // Import knot and multiplicity vector - v
                              int n_v_knots = 1;
                              int* v_indices = new int[array_size_v];
                              int* v_mults = new int[array_size_v];
                              v_indices[0] = 0;
                              v_mults[0] = 1;
                              for (int i=1; i < array_size_v; i++)
                              {
                                    if (abs(v_knots[i-1] - v_knots[i]) < EPS)
                                    {
                                          v_mults[i] = v_mults[i-1] + 1;
                                          v_indices[n_v_knots-1] = i;
                                    }
                                    else
                                    {
                                          n_v_knots++;
                                          v_mults[i] = 1;
                                          v_indices[n_v_knots-1] = i;
                                    }
                              }
                              // Import knots and multiplicities - v
                              TColStd_Array1OfReal* V_Knots = new TColStd_Array1OfReal(1,
                                     n_v_knots);
                              TColStd_Array1OfInteger* V_Mults = new TColStd_Array1OfInteger(1,
                                     n_v_knots);
                              for (int i=0; i < n_v_knots; i++)
                              {
                                    V_Knots->SetValue(i+1, v_knots[v_indices[i]]);
                                    V_Mults->SetValue(i+1, v_mults[v_indices[i]]);
                              }
                              V_Mults->SetValue(1, v_degree + 1);
                              V_Mults->SetValue(n_v_knots, v_degree + 1);
                              // Create attribute
                              pAttribute = new Topo_Face_Attribute_BSplineSurface(att_id,
                                     is_rational, u_degree, v_degree, n_u_poles, n_v_poles,
                                     n_u_knots, n_v_knots, Poles, U_Knots, V_Knots, U_Mults,
                                     V_Mults, Weights);
                              my_Att_Map->add_Entity(pAttribute);
                              // Delete arrays
                              delete u_indices;
                              delete u_mults;
                              delete v_indices;
                              delete v_mults;
                        }
                  }
            }
            // Delete arrays
            if (xyz1)
                  delete xyz1;
            if (t1)
                  delete t1;
            if (axis)
                  delete axis;
            for (int i=0; i < int(points.size()); i++)
            {
                  double* point = points.at(i);
                  delete point;
            }
            points.clear();
            // Break loop
            break;
      }
```

```
                        // ---- Import attribute data ----
                              // ???
                              if (buf_att.cdb_area.m_id0 == 11)
                              {
                                    printf("11!\n");
                              }
                              // NURBS - Knots
                              else if (buf_att.cdb_area.m_id0 == 290)
                              {
                                    double* point = new double[6];
                                    point[0] = buf_att.cdb_gard_pts.m_x;
                                    point[1] = buf_att.cdb_gard_pts.m_y;
                                    point[2] = buf_att.cdb_gard_pts.m_z;
                                    point[3] = buf_att.cdb_gard_pts.m_w;
                                    point[4] = buf_att.cdb_gard_pts.m_m;
                                    point[5] = buf_att.cdb_gard_pts.m_n;
                                    points.push_back(point);
                              }
                              // NURBS - Control Points
                              else if (buf_att.cdb_area.m_id0 == 291)
                              {
                                    att_id = my_Att_Map->max_id() + 1;
                                    array_size_u = (il_att - 2*sizeof(int))/sizeof(double);
                                    u_degree = buf_att.cdb_gard_nku.m_deg;
                                    for (int i=0; i < array_size_u; i++)
                                          u_knots[i] = buf_att.cdb_gard_nku.m_s[i];
                              }
                              // NURBS - Control Points
                              else if (buf_att.cdb_area.m_id0 == 292)
                              {
                                    att_id = my_Att_Map->max_id() + 1;
                                    array_size_v = (il_att - 2*sizeof(int))/sizeof(double);
                                    v_degree = buf_att.cdb_gard_nkv.m_deg;
                                    for (int i=0; i < array_size_v; i++)
                                          v_knots[i] = buf_att.cdb_gard_nkv.m_s[i];
                              }
                              // Update attribute buffer
                              att_buffer++;
                        // End: Loop over attribute buffer data
                        } while (ie_att <= 1);
                        break;
                  // End: Loop over attribute buffers (kwh 3)
                  } while(kwh_att == 3);
            }
            // Update region buffer
            reg_buffer++;
      // End: Loop over buffer data
      } while (ie_reg <= 1);
// End: Loop over region buffers (kwh 32)
} while(kwh_reg == 32);
// ---- Return ----
      return true;
}
```

```
/*****************************************************************************
 **** class Topo_Shape : Private member funcion                            ****
 *****************************************************************************/
bool Topo_Shape::import_CDB_Mesh(char* fileName)
{
// ----     Initialize cdb (fileName, Index) ----
      int         icd;
      icd = sof_cdb_init(fileName, 99);
// ----     Initialize file-information of the cdb ----
      struct      FILEDATA filedata;
      int         ret_file;
    int           l_file;
    l_file = sizeof(filedata);
    ret_file = sof_cdb_get(icd, 10, 0, &filedata, &l_file, 0);
// ----     Import nodes ----
      int   ie_node, il_node;
      int kwh_node = 20;
      int kwl_node = 0;
      int node_buffer = 0;
      // Loop over node buffers (kwh 20)
      do
      {
            // Import buffer data
            struct CNET_NODE buf;
            il_node = sizeof(buf);
            ie_node = sof_cdb_get(icd, kwh_node, kwl_node, &buf, &il_node, node_buffer);
            // End of buffer reached
            if(ie_node > 1)
                  break ;
            // Create boundary condition
            Topo_BoundaryCondition* pBC = 0;
            int bc_id = buf.m_KFIX;
            if (bc_id > 0)
            {
                  if (!(pBC = (Topo_BoundaryCondition*)my_BC_Map->get_ID_Entity(bc_id)))
```

```
                {
                      pBC = new Topo_BoundaryCondition(bc_id);
                      my_BC_Map->add_Entity(pBC);
                }
          }
          // Create node
          Topo_Node* pNode = new Topo_Node(buf.m_NR, buf.m_XYZ[0], buf.m_XYZ[1], buf.m_XYZ[2]);
          my_Node_Map->add_Entity(pNode);
          if (pBC)
                pNode->add_BoundaryCondition(pBC);
          // Update node buffer
          node_buffer++;
     // End: Loop over node buffers (kwh 20)
     } while (ie_node <= 1);
// ----     Import elements ----
     int   ie_elem, il_elem;
     int kwh_elem = 200;
     int kwl_elem = 0;
     int elem_buffer = 0;
     // Loop over element buffers (kwh 200)
     do
     {
          // Import buffer data
          struct CNET_QUAD buf;
          il_elem = sizeof(buf);
          ie_elem = sof_cdb_get(icd, kwh_elem, kwl_elem, &buf, &il_elem, elem_buffer);
          // End of buffer reached
          if(ie_elem > 1)
                break ;
          // Import triangle
          if ((buf.m_NODE[3] == 0) || (buf.m_NODE[2] == buf.m_NODE[3]))
          {
                // Import Nodes
                Topo_Node* pNodes[3];
                for (int i=0; i < 3; i++)
                {
                      if (!(pNodes[i] = (Topo_Node*)my_Node_Map->get_ID_Entity(buf.m_NODE[i])))
                            return false;

                }
                // Import Edges
                Topo_Edge* pEdges[3];
                for (int i=0; i < 4; i++)
                      pEdges[i] = 0;
                // Import Region
                Topo_Region* pRegion = 0;
                // Create Triangle
                Topo_Triangle* pTri = new Topo_Triangle(buf.m_NR, buf.m_MAT, pNodes, pEdges, pRegion, 0);
                my_Tri_Map->add_Entity(pTri);
          }
          // Import quadrilateral
          else
          {
                // Import Nodes
                Topo_Node* pNodes[4];
                for (int i=0; i < 4; i++)
                {
                      if (!(pNodes[i] = (Topo_Node*)my_Node_Map->get_ID_Entity(buf.m_NODE[i])))
                            return false;

                }
                // Import Edges
                Topo_Edge* pEdges[4];
                for (int i=0; i < 4; i++)
                      pEdges[i] = 0;
                // Import Region
                Topo_Region* pRegion = 0;
                // Create quadrilateral
                Topo_Quadrilateral* pQua = new Topo_Quadrilateral(buf.m_NR, buf.m_MAT, pNodes, pEdges, pRegion, 0);
                my_Qua_Map->add_Entity(pQua);
          }
          // Update element buffer
          elem_buffer++;
     // End: Loop over element buffers (kwh 200)
     } while (ie_elem <= 1);
// ---- Return ----
     return true;
}
```

```
/*************************************************************************
 **** class Topo_Shape : Private member funcion                       ****
 *************************************************************************/
bool Topo_Shape::export_CDB_Mesh(char* fileName)
{
     int icd,ie;
// ----     Initialisieren einer neuen *.cdb mittels sof_cdb_init(FileName, Index) ----
     icd = sof_cdb_init(fileName, 94);
// ----     Einlesen der Datei-Informationen der jeweiligen *.cdb----
     struct      FILEDATA filedata;
     struct      CNET_NODE netnodedata;
     struct      CNET_QUAD netquaddata;
```

```
    int           l_file;
    l_file = sizeof(filedata);
// ---- Initialize Iterator ----
    std::map<int, void*>::const_iterator pIterator;
// ----     Einlesen der Datei-Informationen der jeweiligen *.cdb----
    double x_min = MFLOAT, y_min = MFLOAT, z_min = MFLOAT, x_max = -MFLOAT, y_max = -MFLOAT, z_max = -MFLOAT;
    // ---- Write Nodes ----
    for (pIterator = my_Node_Map->get_First(); pIterator != my_Node_Map->get_Last(); pIterator++)
    {
          Topo_Node* pNode = (Topo_Node*)pIterator->second;
          if (pNode->x() > x_max)
                x_max = pNode->x();
          if (pNode->y() > y_max)
                y_max = pNode->x();
          if (pNode->z() > z_max)
                z_max = pNode->z();
          if (pNode->x() < x_min)
                x_min = pNode->x();
          if (pNode->y() < y_min)
                y_min = pNode->y();
          if (pNode->z() < z_min)
                z_min = pNode->z();
    }
    // ----     Inhalt der Datenstruktur von FILEDATA     ----
    filedata.iprob = 0;                          //|Type of System | Art des Systems  siehe "cdbase.chm"
    filedata.iachs = 0;                          //|Orientation of gravity
    filedata.nknot = my_Node_Map->n_Entities();  //|Number of nodes            |Anzahl Knoten
    filedata.mknot = my_Node_Map->max_id();      //|Highest node number        |Hoechste Knotennr
    filedata.igdiv = 0;                          //|Group divisor              |Gruppendivisor
    filedata.igres = 0;                          //|degrees of freedom and other options
    filedata.xmin  = x_min;                      //|bounding box               |umhüllende Box
    filedata.ymin  = y_min;                      //
    filedata.zmin  = z_min;                      //
    filedata.xmax  = x_max;                      //
    filedata.ymax  = y_max;                      //
    filedata.zmax  = z_max;                      //
    filedata.xref  = 0;                          //
    filedata.yref  = 0;                          //
    filedata.zref  = 0;                          //
    filedata.tglob[3][3] = 0;                    //|global CDB-System transformation matrix
    // ----     Inhalt der Datenstruktur von NetNode      ----
    // ---- Netz-Knoten-Auslese-Schleife ----
    int il;
    int pos_netnode = 0;
    il = sizeof(netnodedata);
    // ---- Write Nodes ----
    for (pIterator = my_Node_Map->get_First(); pIterator != my_Node_Map->get_Last(); pIterator++)
    {
          Topo_Node* pNode = (Topo_Node*)pIterator->second;
                netnodedata.m_XYZ[0] = float(pNode->x());
                netnodedata.m_XYZ[1] = float(pNode->y());
                netnodedata.m_XYZ[2] = float(pNode->z());
                netnodedata.m_NR = pNode->id();
                // ---- Besetzen der restlichen Variablen ----
                netnodedata.m_INR  = pos_netnode;
                netnodedata.m_NCOD = 127;
                if (pNode->get_BoundaryCondition())
                      netnodedata.m_KFIX = pNode->get_BoundaryCondition()->id();
                else
                      netnodedata.m_KFIX = 0;
                // ---- Besetzen der CDBase ----
                ie = sof_cdb_put(icd,20,00,&netnodedata,&il,pos_netnode);
                pos_netnode = pos_netnode + 1;
    }

    // ---- Netz-Quad-Auslese-Schleife ----
    int i;
    int pos_netquad = 0;
    il = sizeof(netquaddata);
    // ---- Write Nodes ----
    for (pIterator = my_Qua_Map->get_First(); pIterator != my_Qua_Map->get_Last(); pIterator++)
    {
          Topo_Quadrilateral* pQua = (Topo_Quadrilateral*)pIterator->second;
                netquaddata.m_NR = pQua->id();
                for(i=0; i<4; i++)
                      netquaddata.m_NODE[i] = pQua->get_Node(i)->id();
                // ---- Besetzen der restlichen Variablen ----
                netquaddata.m_MAT =     0;
                netquaddata.m_MRF =     0;
                netquaddata.m_NRA = 0;
                netquaddata.m_DET[3] = 0;
                netquaddata.m_THICK[5] = 0;
                netquaddata.m_CB = 0;
                netquaddata.m_CQ = 0;
                netquaddata.m_T[9] = 0;
                // ----     Setzen der Datenstruktur in die *.cdb     ----
                ie = sof_cdb_put(icd,200,00,&netquaddata,&il,pos_netnode);
                pos_netquad = pos_netquad + 1;
    }
    // ----     Setzen der Datenstruktur in die *.cdb     ----
   ie = sof_cdb_put(icd,10,0,&filedata,&l_file,0);
    // ----     Schließen der *.cdb     ----
    sof_cdb_close(0);
// ---- Retrun ----
    return true;
}
```