h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fb i
FACHBEREICH INFORMATIK

Computergeometrie und CAD-Systeme

Sommersemester 2009

Prof. Dr. Stefan Wiesmann

Inhalt

1. Aufgaben und Einsatzfelder des Computer Aided Design

Entwurf, Konstruktion: Das Vorausdenken eines technischen Gebildes,
das Vorausdenken der Gestalt,
der Funktion,
des Materials,
des Kräfteverlaufs und der mechanischen Belastung etc.

Ziel: Erstellen einer technischen Dokumentation für die Produktion, d.h. Zeichnungen, Berechnungen, Stücklisten etc.

Die Levels des Rechnereinsatzes:

Low: Traditionell werden für die technische Dokumentation Zeichnungen erzeugt. Dafür kann es zwei Gründe geben: das zu entwerfende Objekt kann tatsächlich vollständig mit einer 2D-Darstellung beschrieben werden (z.B. rotationssymetrische Objekte) und/oder bei der Produktion (z.B. auf der Baustelle) müssen Zeichnungen verwendet werden. Ausgangsinformation ist eine Skizze (auf Papier oder im Kopf), Ergebnis sind "maschinell" erzeugte Zeichnungen (auf Papier oder dem Bildschirm).

Middle: Das 3D-Gestaltmodell wird in den Rechner eingegeben mit dem Ziel, eine Serie unterschiedlichster grafischer Auswertungen (Zeichnungen oder Bilder) davon ableiten zu können. Die Ermittlung der Gestalt und deren nachgewiesene Dimensionierung erfolgt zuvor traditionell.

High: Das Konstruktionsziel wird direkt in den Rechner gegeben.
Maschinenbau: Funktionalität wird ikonenhaft dargestellt und gefordert (z.B. Getriebe, Gurtbandförderer, Ansaugrohr bei Verbrennungsmotoren).
Bauwesen, Architektur: Der Prinzipentwurf wird ad hoc skizziert (z.B. Wandlinien und Öffnungskästen).
An diesem grafischen Modell, das in keiner Weise die Vollständigkeit der Ergebniszeichnung haben wird, werden konstruktive Berechnungen, wie Statik, Dimensionierung, etc. durchgeführt und damit die endgültige Gestalt abgeleitet.

Integration des CAD in die CAx-Produkte:

Immer mehr produktive Tätigkeiten werden automatisiert, d.h. rechnergesteuert. Dafür muß in den meisten Fällen ein Modell des Produktes bereitgestellt werden. D.h., das CAD-Modell (Ergebnis) muß in irgendeine Form transformiert und überführt werden.

Beispiele: numerisch gesteuerte 3D-Fräsmaschinen für Karosseriepressen,
Gehäusegeometrie für numerisch gesteuerte Blechschneide- und Biegeautomaten,
aber auch: Gebäude- und Rauminformationen für das Facility Management.

Hauptaufgaben bei der Entwicklung von CAD-Systemen:

- **Modellierung des Produktes als Gesamtheit und Definition der rechnerinternen Darstellung,**
- **Funktionalität zur Modelleingabe (Modellierung), insbesondere zur Gestalteingabe (3D) und zur (grafischen) Kommunikation mit dem Modell,**
- **Funktionalität zur Visualisierung des Modells, von der technisch/genormten Darstellung in Form technischer Zeichnungen (z.B. Bemaßung, Schraffuren,...), der projektiven Darstellung als Strichzeichnung (z.B. Perspektive) bis hin zur photorealistischen Darstellung und Animation,**
- **Funktionalität zur Auswertung des Gestaltmodelles, wie Flächen- und Volumenberechnung, Stücklisten, Massen, Durchdringung und Kollision, Unterstützung von Bausteinlokalität (Montageprinzip),**
- **Funktionalität und Wissensverarbeitung zur Berücksichtigung von konstruktiven Gesetzmäßigkeiten und Regeln bei der Modelleingabe (Expertenwissen und Expertensysteme),**
- **Funktionalität zur fachgebietsspezifischen Berechnung, Dimensionierung, Kostenermittlung usw. (Applikationssoftware).**

2. Struktur eines CAD-Systems

2.1. Konfiguration eines CAD-Arbeitsplatzes

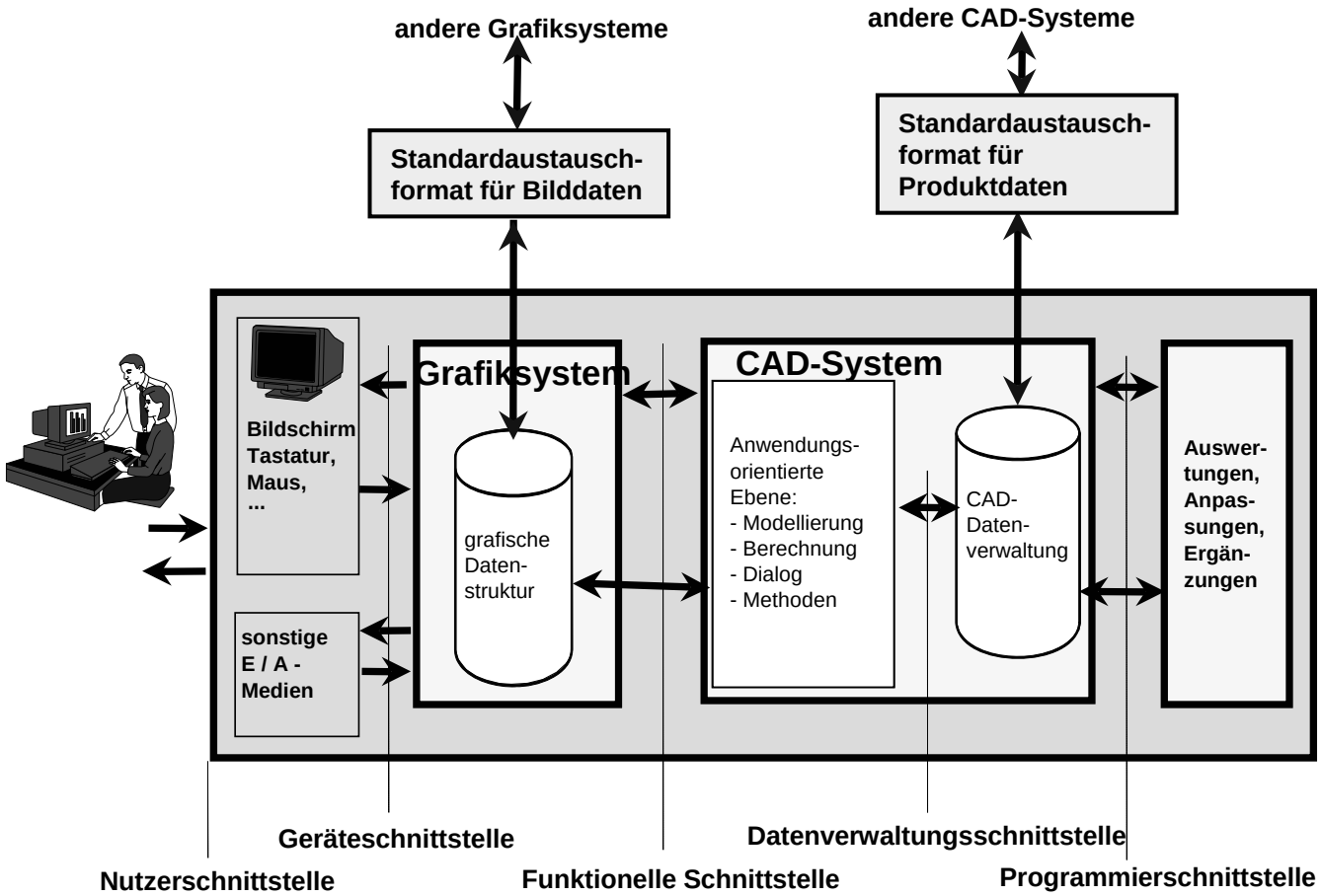
CAD-Arbeitsplätze werden charakterisiert durch:

- Verfügbarkeit am Arbeitsplatz mit lokaler Bedeutung,
- hohe Anforderungen an Reaktionszeiten während der Sitzung,
- hohe Anforderungen an die Grafikqualität und -quantität (Bildgröße),
- Arbeit im Team (parallel oder nacheinander) muß möglich sein,
- zentrale Haltung und Wartung riesiger Kataloge,
- Plattformunabhängigkeit und Kommunikation mit parallelen CAD-Systemen,
- Ablaufsteuerung durch Verwaltung von Szenarien.

Schlußfolgerungen für die Hardware- und Systemkonfiguration:

- PC oder Workstation am Arbeitsplatz (Preislimit)mit beträchtlicher lokaler Intelligenz,
- große, hochauflösende Farbgrafikbildschirme mit eigenem leistungsfähigen Prozessor,
- grafisches Hardcopy (zentral) über Drucker/Plotter(immer mehr Tintenstrahl-),
- Vernetzung und Zugang zu zentralen Datenbanken und Projekten auf dem Server,
- CAD-Basissystem, das Plattform- und Geräteabhängigkeit abfängt (AutoCAD, MicroStation, Pro/ENGINEER, Catia etc.),
- möglichst einheitliches Betriebssystem, Programmiersprache und Datenverwaltungssystem (historisch: UNIX, FORTRAN, Rel. DB; Trend: WINDOWS, C++, Java, auch OO DB),
- sicheres Projekt-, Dokument- und Versionsmanagement mit Zugriffsrechtverwaltung

2.2. Innere Struktur einer CAD-Software



Grundsätzlich muß unterschieden werden zwischen

- den grafisch-geometrischen Aufgaben zur Gestaltmodellierung und -visualisierung und
- den Berechnungen und Konstruktionsmethoden des "Fach"-CAD.

Beide Prozesse müssen ihre Informationen speichern, d.h. sie haben - zumindest anfangs - separate Datenstrukturen und Datenbanken.

Die Kommunikation mit dem User, die im CAD-Bereich vorrangig einen grafikorientierte Inhalt hat, wird vom **Grafiksystem** übernommen. Gleichzeitig verwaltet das Grafiksystem alle 2D-(Zeichnungen) und 3D-(Gestalt)Modelle.

Das innere **CAD-System** realisiert alle Berechnungen und den dafür notwendigen Dialog.

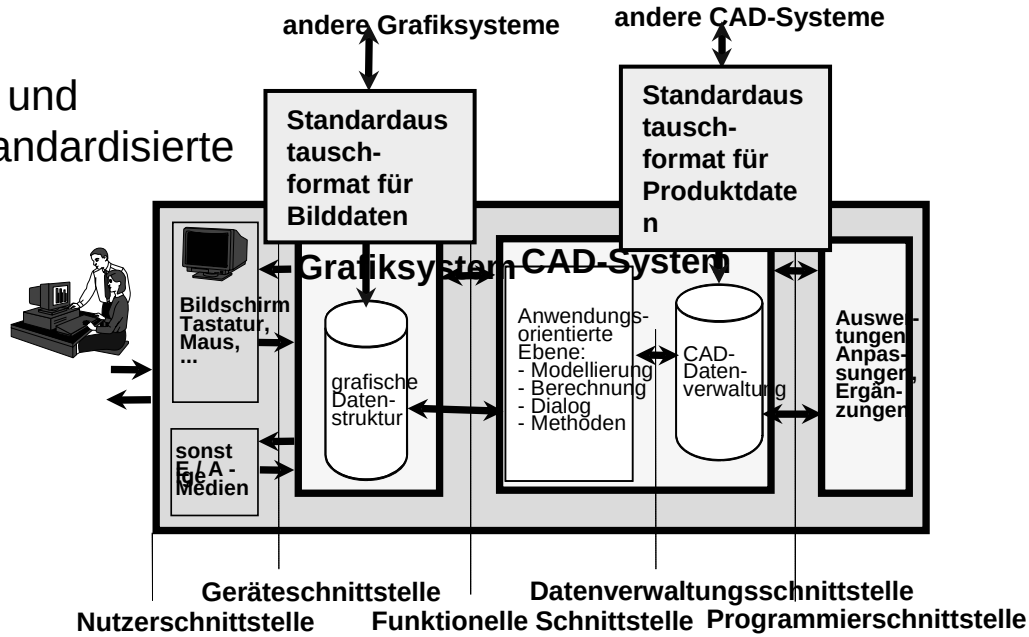
Als Schnittstelle für Systemerweiterungen und Anpassungen dient die **Programmierschnittstelle**, auch CAD-Programmiersprache genannt.

Die **Datenverwaltungsschnittstelle** regelt den Zugang zum - meist externen - Datenmodell. (SQL),

Das Grafiksystem bietet eine **funktionale Schnittstelle** zur Ansteuerung der Funktionalität des Grafiksystems und zum Auslesen bzw. Schreiben von Modellinformationen in die grafische Datenstruktur.

Wie bekannt wird der Zugang zu den Ein- und Ausgabegeräten über eine weitgehend standardisierte **Geräteschnittstelle** gelöst. (CGI, ...)

Die äußere Sicht des Gesamtsystems bezeichnet man als **Nutzerschnittsstelle** (oder auch Benutzungsoberfläche, User Interface, GUI ...).



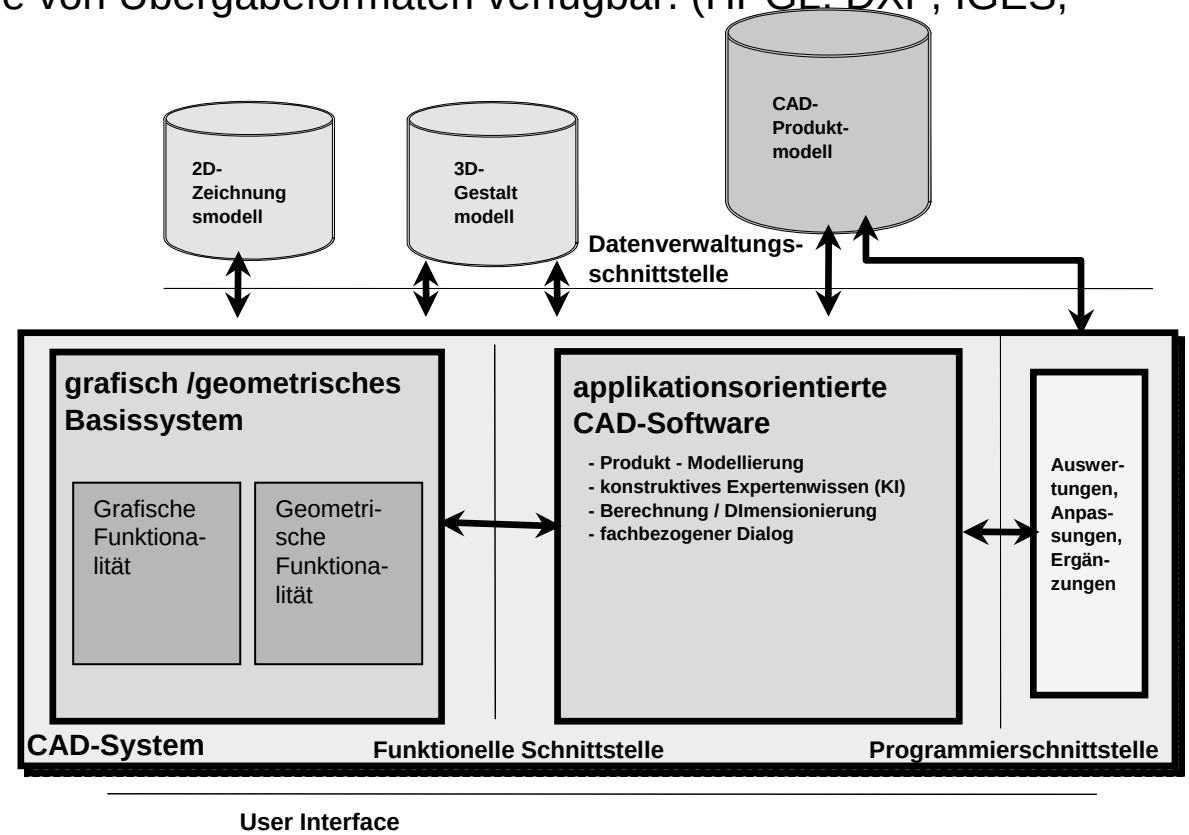
Für den **Informationsaustausch** mit anderen Systemen müssen prinzipiell zwei Niveaus unterschieden werden:

a) der Austausch von grafisch/geometrischen Informationen (Zeichnungen, Bilder, unattributierte Gestalt) mit anderen Grafiksystemen und

b) der Austausch von ganzheitlichen Modell- (Produkt-) Daten mit anderen CAD-Systemen.

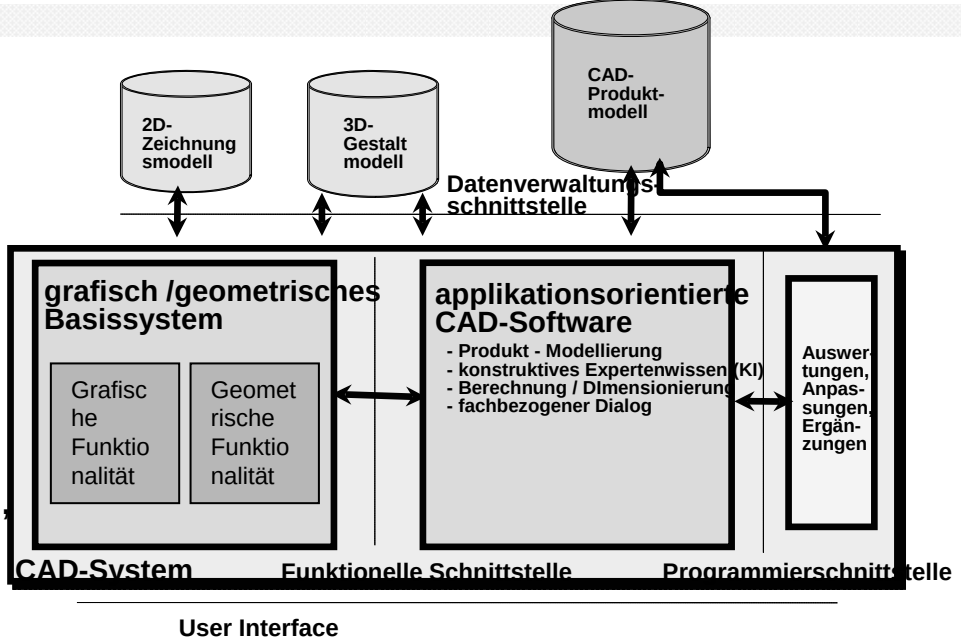
Für beide Wege sind eine Reihe von Übergabeformaten verfügbar. (HPGL, DXF, IGES, GKSM, STEP, ...)

Bei einer feineren Strukturierung der Komplexe muß weiterhin zwischen der grafischen und der geometrischen Funktionalität unterschieden werden:



Aufgaben der Grafik:

Ausgabe von grafischen Primitiven,
Verwaltung der Identifizierbarkeit und der interaktive Modifikation,
Steuerung und Ausführung der grafischen Attribute (Farbe, Linienart etc.)
Gestaltung des "Zeichenblattes", wie Rahmen, Schriftfeld etc.
Erzeugen rein grafischer Bildinhalte, wie Schraffuren, Bemaßung etc.



Aufgaben der Geometrie:

Verwaltung der (3D)-Gestaltinformation,
Funktionalität zur Modellierung der Gestalt einschließlich geometrischer Konstruktionsalgorithmen,
Auswertungsfunktionen der Gestaltinformationen, wie Volumenberechnung etc.,
alle Projektionsverfahren zur 3D-2D-Transformation einschließlich der Verdeckungsalgorithmen.

Es ist naheliegend, daß beide Funktionskomplexe eigene Daten erzeugen und verwalten müssen. So kann man streng genommen drei externe Datenmodelle unterscheiden, die Zeichnungsdaten, die Gestaltdaten und die Produktdaten. Zielstellung der Entwickler von CAD-Basissoftware ist es, diese Dreiteilung wieder zu beseitigen und ein einheitliches Datenformat und -medium für alle drei Informationsmengen bereitzustellen.

2.3. Rechnerinterne Modelle in CAD-Systemen

2.3.1. Das Zeichnungsmodell - Aufgaben und Funktionen

Eine Zeichnung ist eine von beliebig vielen Sichten auf das Modell. Bei 3D-Gestaltmodellen ist sie das Ergebnis einer Projektion.

An Zeichnungen werden aus der Sicht der technischen Dokumentation bestimmte Anforderungen gestellt:

- **Maßhaltigkeit, möglichst maßstäblich,**
- **Zoom- und Pan-Funktionen,**
- **Innerhalb eines Zeichnungslayouts (standardisiertes Format <-> Druckerausgabeformat, Rahmen, Schriftfeld, Faltmarken etc.),**
- **meist einfarbig, d.h. Unterscheidung von Linien durch Strichbreite und Linienart,**
- **Schraffuren (anstelle von gefärbten Flächen),**
- **Bemaßungsgrafik und Beschriftung.**

Das Zeichnungskoordinatensystem ist meist identisch mit dem Gerätekoordinatensystem des Druckers. Im Gegensatz dazu sind die (auf die Zeichenfläche projizierten) Modellkoordinaten reine Weltkoordinaten.

---> Definition von einem oder mehreren Ansichtsfenstern auf der Zeichnungsfläche. (Window-Viewport-Transformation)

Der Inhalt des Zeichnungsmodells besteht aus den grafischen (2D-)Primitiven. Gleichzeitig muß - solange wie machbar - die Relation zum erzeugenden Modell gewartet werde. Eine Schichtung in Layern oder Folien erlaubt das schnelle Ein- bzw. Ausschalten (für die Darstellung) von grafischen Informationen.

2.3.2. Das Gestaltmodell und geometrische Funktionalität

Will man ein Objekt realitätsnah abbilden, ist ein dreidimensionales Modell, auch Gestaltmodell genannt (Shape), in den meisten Fällen notwendig.

Gründe dafür sind:

- Die grafischen Dokumente des Objektes, also die Menge der zu erzeugenden Zeichnungen, sind in Abhängigkeit von der Nutzung zumeist nur spezielle Sichten auf das Objekt, die immer auf der selben Gestaltinformation basieren, jedoch in ihrer grafischen Darstellung sich qualitativ voneinander unterscheiden. Anstelle dieser schwer überschaubaren Zeichnungsinformationen wird dagegen nur **ein** Gestaltmodell aufzubauen und zu verwalten sein.

- Modifikationen an der Gestalt werden anhand von grafischen Modifikationen in einer relevanten Zeichnung vorgenommen. Dennoch müssen alle weiteren Zeichnungen damit inhaltlich konsistent aktualisiert werden. Die gemeinsame Wurzel ist das Gestaltmodell, das als einziges immer nachgeführt werden kann.

- oder generell: Die Realität, die es abzubilden gilt, ist - bis auf wenige Ausnahmen - dreidimensional, also muß das rechnerinterne Abbild auch dreidimensional sein, um keine Informationsverluste gegenüber dem realen Objekt hinzunehmen.

- Die Methoden der rechnerinternen Gestaltabbildung sind allgemein verfügbar. Zu nennen wären u.a. folgende:

  -**BRep (boundary representation) = "Rand"-Beschreibung,**
  -**CSG (Construction Solid Geometry) = Volumen-(Mengen-)-Beschreibung,**
  -**Octree = Clusterung in Gestaltvoxel und Notation der Objektzugehörigkeit,**
  -**Freiformflächen als Coons'sche Flächen oder Splines.**

Die Funktionalität für Gestaltmodelle gliedert sich in
- Modellierungsoperationen,
- Visualisierungsoperationen und
- Berechnungen an der Gestaltinformation.

Visualisierung der Gestalt:

- Projektion (3D-2D-Transformationen),
- 3D-Clipping und Schnittgenerierung,
- Generieren der grafischen Informationen (der grafischen Primitive),
- HiddenLine- und Hidden-Surface-Berechnungen,
- Erzeugen gerenderter Darstellungen (fotorealistischen Darstellungen)
- aber auch: Generierung von Maschen und Netzen für approximierte Abbildungen.

Berechnungen an der Gestaltinformation:

- Volumen- und Flächenberechnungen (z.B. Gauß'sches Flächenintegral),
- Abstands- bzw. Kollisionsermittlung,
- Schwerpunktsberechnung,
- u.a.

2.3.3. Das Produktmodell

Unter dem Begriff "Produktmodell" wird die Gesamtheit aller Informationen über ein technisches Objekt, die für die CA-Arbeit relevant sind, verstanden. Neben den Gestaltinformationen zählen dazu insbesondere alle weiteren Attribute, wie Material, Masse, Oberflächenbeschaffenheit usw. und die funktionalen bzw. konstruktiven **Relationen** zwischen den Teilen (Entities) des Objektes.

IPIM - Integrated Product Information Model

Die Verbindung zwischen dem CAD-Modell und den Produktdaten können auf zwei unterschiedlichen Wegen erfolgen:

a)Attribute werden als gekapselte, vom CAD primär nicht interpretierbare Informationssets den Elementen des Gestaltmodells zugeordnet und datentechnisch dort verwaltet. (Beispiel: Das AutoCAD-Konstrukt ATTRIBUT erlaubt die Zuordnung von Attributlisten zu grafisch/geometrischen Blöcken. Auf diese Weise kann beispielsweise das Material eines Bauteils seiner Gestaltinformation zugeordnet und gespeichert werden.)

b)Die Nichtgestaltinformationen werden in einer (meist relationalen) Datenbank, die parallel zur CAD-Datenbank definiert und eingerichtet wird, abgelegt. Die Relationen zwischen Gestaltinformation und Attributen werden über bidirektionale Connect-ID's verwaltet. Sowohl in der Datenbank werden Pointer auf die korrespondierende Gestaltinformation des CAD-Modells abgelegt, als auch im CAD-Modell wird der ID der zugeordneten Datenbankinformation geführt. Die CAD-Daten können auch gekapselt in der Datenbank archivert werden.

3. Die Schnittstellen (APIs) offener CAD-Basissysteme am Beispiel von AutoCAD

3.1. Die Gates eines CAD-Systems

Wünschenswert (und bei AutoCAD vorhanden) sind folgende Eingriffs- und Erweiterungsmöglichkeiten in das CAD-System:

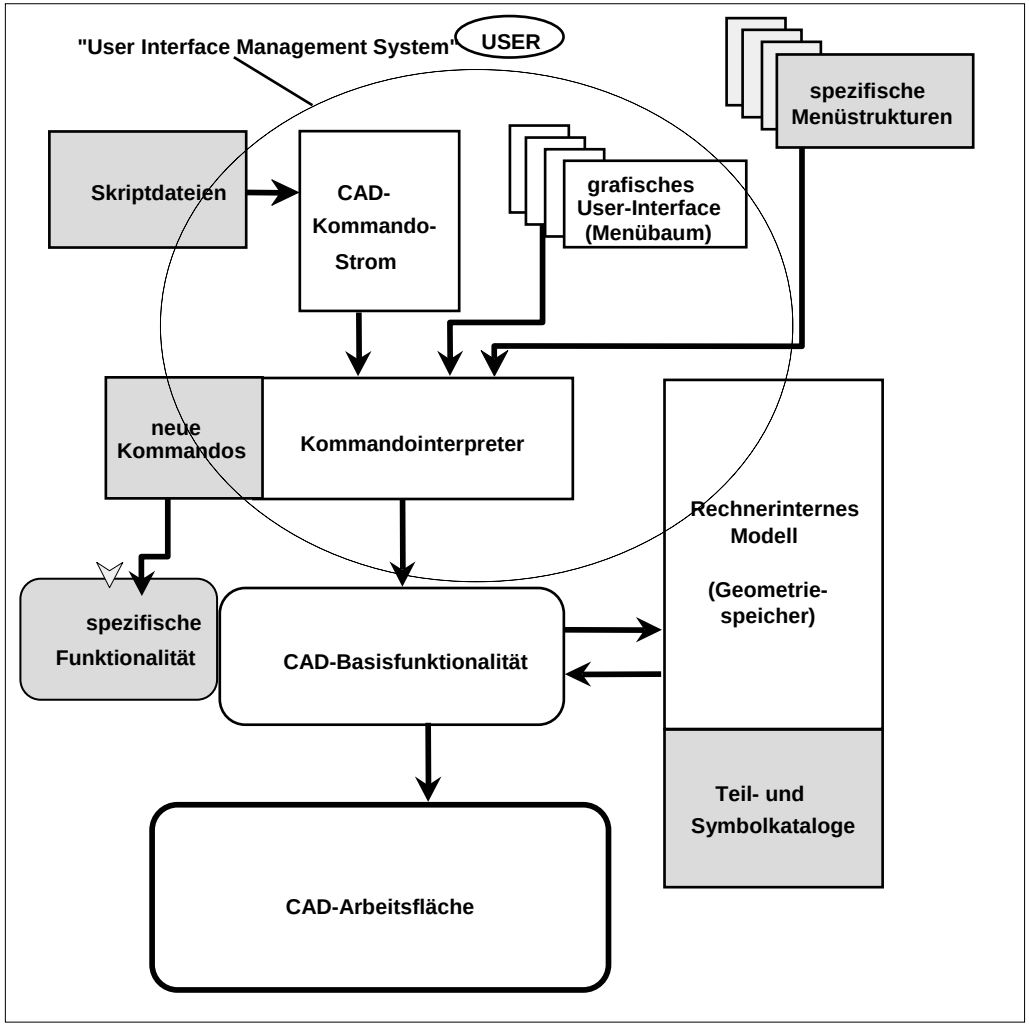
- Skripte / Batch-Dateien (Programmierung in der Sprache des CAD-Systems) und Journals (Aufzeichnung von Benutzeraktionen, sogenannte Makro-Rekorder),
- Einbinden neuer Kommandos (sowohl im Kommandointerpreter als auch im Funktionsteil), evtl. Ändern der Menüs und Dialogboxen,
- Ändern / Erweitern der Kataloge (z.B. Hinzufügen eines bisher nicht bekannten Objektes zur Objektbibliothek),

Die folgende Abbildung zeigt (in den grau unterlegten Kästen) diese Erweiterungsmöglich-keiten ("Gates") bei AutoCAD.

- Skript-Dateien können hier als Strom mehrerer AutoCAD-Kommandos in den Kommando-interpreter "eingespeist" werden.
- Die Menüstrukturen können geändert oder mit anderen Befehlen ausgestattet werden, die an den Kommandointerpreter weitergereicht werden sollen.
- Weiterhin kann der Kommandointerpreter um neue Kommandos erweitert werden, die dann neue spezifische Funktionen realisieren.

•Der Geometriespeicher kann um neue Objekte erweitert werden (beispielsweise Erweitern einer Symbolbibliothek um ein neues Symbol).

Zur Erklärung: Das UIMS (User Interface Management System) setzt alle Eingaben des Users (z.B. einen Mausklick oder einen Tastendruck) in Kommandos um.



3.2. AutoCAD-Skript / AutoCAD-Journal

Voraussetzung für die Ausführung von Skripten (Makros) ist, daß der CAD-Kommandostrom so umgeleitet werden kann, daß die Eingaben statt vom User aus einer Skript-Datei kommen.

Zusätzliche Kommandos für Skript-Dateien (mit der Datei-Endung .SCR) steuern den zeitlichen Ablauf des Kommandostroms:

SCRIPT-Kommando	Funktion
`SCRIPT <Skript-Datei>`	Starten eines Skriptes aus der aktuellen Situation
`RSCRIPT`	Erneutes Starten des Skriptes vom Anfang
`PAUSE <Millisekunden>`	Verzögerung der weiteren Bearbeitung
`R bzw. S c`	Unterbrechung eines Skriptes nach der jetzigen Aktion
`RESUME`	Fortsetzen des Skriptes nach einem Abbruch

Quelltext `DEMO.SCR`	Funktion
`REGEN`	Zeichnung regenerieren
`TEXT .5,3. 5 0`	Folgenden Text am Punkt (0.5,3.0) einfügen
`Kommen Sie zu einer Demo`	
`PAUSE 1000`	Eine Sekunde Pause
`ERASE L`	Löschen des Textes (L steht für Last, letzer Eintrag)
`RSCRIPT`	Skript wieder von vorne ausführen

Journal-Dateien können für die Produktion von Tutorials (Lernprogramme) genutzt werden, da hierbei alle Aktionen des Nutzers quasi simuliert werden.

3.3. Bedarfsgerechte Menüs

AutoCAD ermöglicht dem User, die Menüs an seine Anforderungen anzupassen oder sogar völlig neue Menüstrukturen zu implementieren.

Veränderbar sind:

- der Hauptmenübalken (AutoCAD-Bezeichnung: MenuMacro) für alle Kommandos aus dem Pulldown-Menü,
- die Kontextmenüs (AutoCAD-Bezeichnung: WSPop), für die Auswahl von Funktionen als PopUp in der Zeichenfläche,
- die Werkzeugdialoge (AutoCAD-Bezeichnung: WSToolbar)

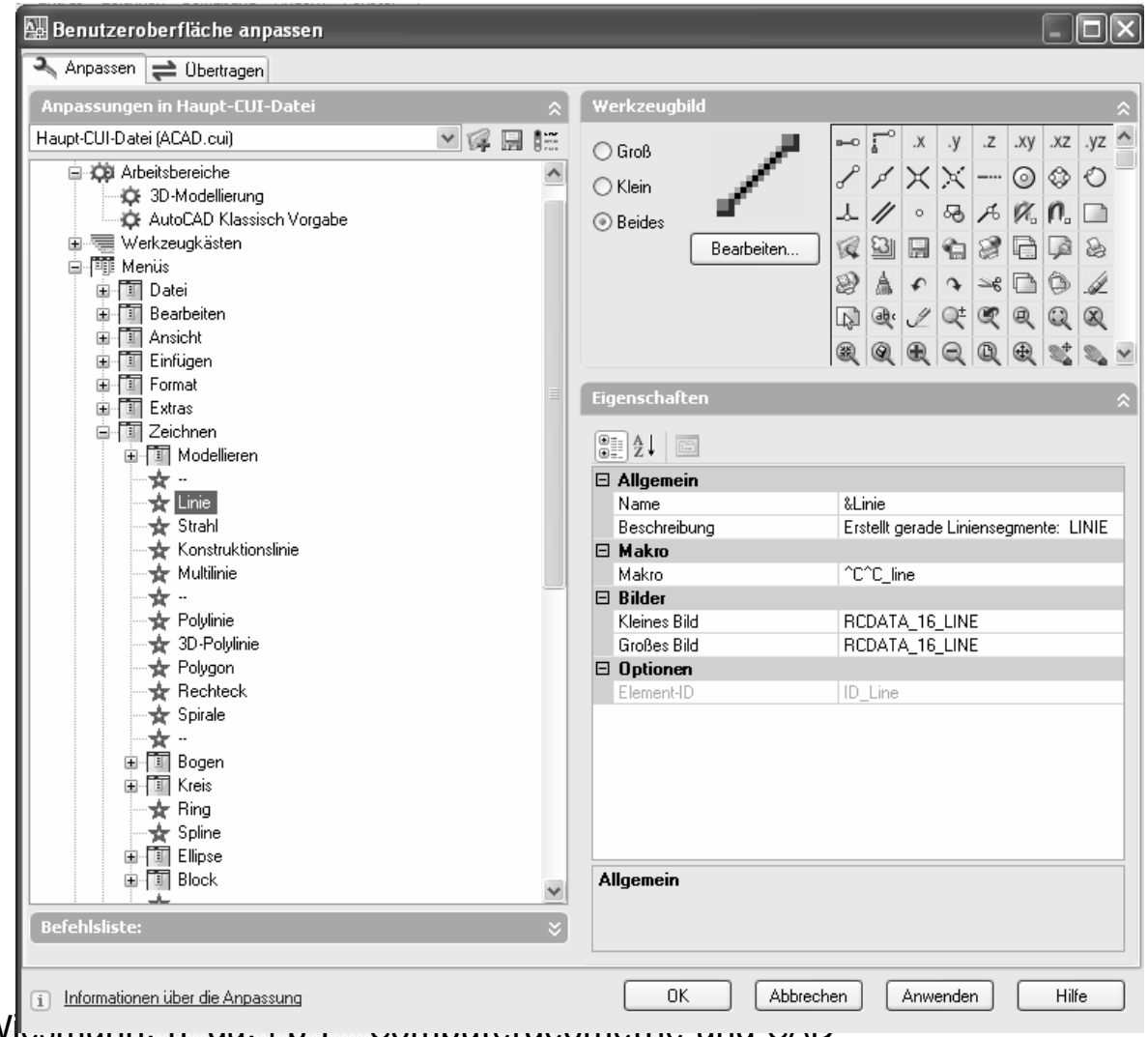
usw.

In der Menüdatei (Datei-Endung .CUI) des Systems werden alle Gestaltungs- und Reaktionsinformationen der Menüs definiert. Der Applikationsentwickler hat die Möglichkeit, das vorgegebene Standardmenü (Datei acad.CUI) zu kopieren und dann nach seinen Gesichtspunkten zu verändern. Das Laden des nunmehr "privaten" Menüs erfolgt mittels des AutoCAD-Kommandos **`menü`** bzw. mit Hilfe des Anpassungsdialoges.

Im nachfolgenden Auszug werden drei "Zeichnen"-Menüs in ihrer XML-basierten Notation definiert:

```
<MenuMacro UID="ID_Line">
  <Macro type="Draw">
    <Revision MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <ModifiedRev MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <Name xlate="true" UID="ACAD_238">Linie</Name>
    <Command>^C^C_line </Command>
    <HelpString xlate="true" UID="ACAD_1523">Erstellt gerade Liniensegmente: LINIE
    </HelpString>
    <SmallImage Name="RCDATA_16_LINE"/>
    <LargeImage Name="RCDATA_16_LINE"/>
  </Macro>
</MenuMacro>
<MenuMacro UID="ID_Ray">
  <Macro type="Draw">
    <Revision MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <ModifiedRev MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <Name xlate="true" UID="ACAD_239">Strahl</Name>
    <Command>^C^C_ray </Command>
    <HelpString xlate="true" UID="ACAD_1607">Erstellt eine einseitig unendliche
                                              Linie:STRAHL
    </HelpString>
    <SmallImage Name="RCDATA_16_RAY"/>
    <LargeImage Name="RCDATA_16_RAY"/>
  </Macro>
</MenuMacro>
<MenuMacro UID="ID_Pline">
  <Macro type="Draw">
    <Revision MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <ModifiedRev MajorVersion="16" MinorVersion="2" UserVersion="0"/>
    <Name xlate="true" UID="ACAD_242">Polylinie</Name>
    <Command>^C^C_pline </Command>
    <HelpString xlate="true" UID="ACAD_1587">Erstellt 2D-Polylinien:  PLINIE
    </HelpString>
    <SmallImage Name="RCDATA_16_PLINE"/>
    <LargeImage Name="RCDATA_16_PLINE"/>
  </Macro>
</MenuMacro>
```

Diese XML-Datei darf nicht manuell geändert werden.
Um Anpassungen an der Bedienoberfläche vorzunehmen, wird mit dem Kommando CUI ein entsprechender Dialog geöffnet, in dem nun alle gewünschten Änderungen vorgenommen werden können bzw. neue Befehle eingefügt werden.



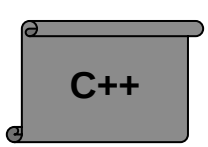
3.4. Entwicklung von CAD-Erweiterungen (Applikationen)

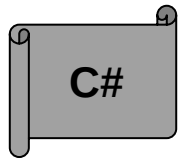
ObjectARX SDK ist eine Sammlung von Libraries und Include-Files für die Entwicklung von s.g. AutoCAD-Applications. Im Zusammenspiel mit MS VisualStudio bildet es die Entwicklungsumgebung für funktionale Erweiterung des CAD-Basissystems. In dieser Form wird es von allen fachspezifischen Entwicklern für die zahlreichen Erweiterungen der Grundfunktionalität genutzt.

Zu beachten ist, dass für das im Labor installierte AutoCAD 2007 die Version ObjectARX 2007 und MS VisualStudio 5 (C++ V.8) notwendig ist, um die geforderte Kompatibilität zu gewährleisten.
Mit der selben Entwicklungsumgebung kann man auch C#--ARX-Entwicklungen schreiben.)

ObjectARX können Sie vom Download-Verzeichnis meiner persönlichen Homepage herunterladen, VisualStudio können Sie über unsere Laboringenieure kostenfrei als akademische Linzenz bekommen. AutoCAD ist kostenpflichtig und lizensiert, aber auf der Seite "Quellen" ist die URL für das Downloaden einer studentischen (kostenfreien) Version angegeben.

Auf den folgenden Seiten wird sowohl die C++-Entwicklung als auch alternierend die C#-Programmierung beschrieben. Zur Kennzeichnung der entsprechenden Seiten werden die Ikons



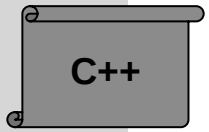


verwendet.

Kennzeichnung der ObjectARX-Klassennamen
AcRx: Kategorien für das Einbinden einer Anwendung und für die Laufzeit-Registrierung und -identifikation.
AcEd: Kategorien für das Registrieren nativer AutoCAD Befehle und für die AutoCAD Ereignisverarbeitung.
AcDb: AutoCAD Database-Klassen.
AcGi: Graphikklassen für das Rendering von AutoCAD Entities.
AcGe: Utility-Klassen für die lineare Algebra und geometrische Objekte.

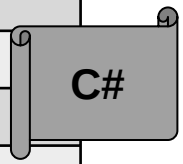
Abhängig von den genutzten Features werden gegebenenfalls noch folgende korrepondierende Libraries notwendig:

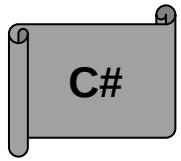
AcRx: acad.lib, rxapi.lib, acdb16.lib
AcEd: acad.lib, rxapi.lib, acedapi.lib, acdb16.lib
AcDb: acad.lib, rxapi.lib, acdb16.lib
AcGi: acad.lib, rxapi.lib, acdb16.lib
AcGe: acad.lib, rxapi.lib, acge16.lib, acdb16.lib



ObjektARX bietet neben C++ Bibliotheken auch Wrapper-Klassen für Managed .NET - Anwendungen. Damit können AutoCAD Erweiterungen mit anderen Sprache wie C# oder VB.NET programmiert werden.
Im Gegensatz zu C++ haben die Managed-Klassen keinen Prefix sondern befinden sich in einem bestimmten Namespace. Anhand des Prefixes kann man auf den jeweils benötigten Namespace schließen:

C++ Klassenprefix	Managed-Namespace der Wrapper-Klasse
AcRx	Autodesk.AutoCAD.Runtime
AcEd	Autodesk.AutoCAD.ApplicationServices
AcDb	Autodesk.AutoCAD.DatabaseServices
AcGi	Autodesk.AutoCAD.GraphicsInterface
AcGe	Autodesk.AutoCAD.Geometry





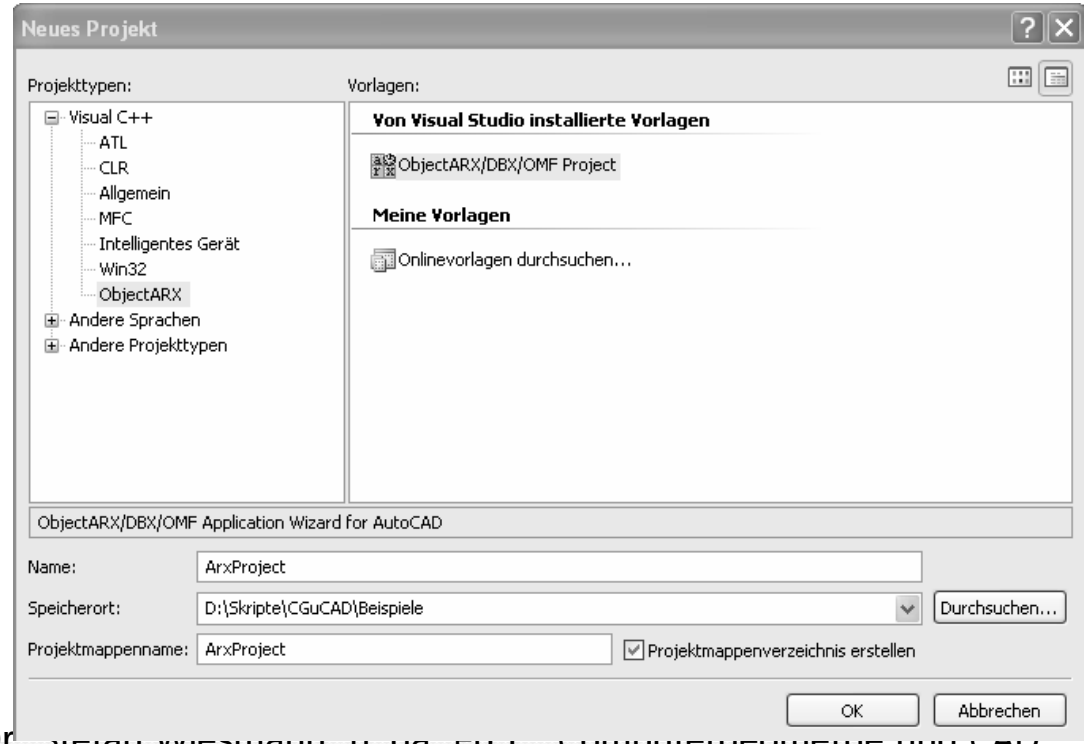
Um alle Wrapper-Klassen nutzen zu können müssen lediglich acdbmgd.dll und acmgd.dll eingebunden werden. Eine vollständige Liste der C++/.NET Prefix-Mappings ist in der ObjectARX Managed Class Reference zu finden.
Beim Anlegen eines neuen Projektes empfiehlt es sich das Einbinden der am häufigsten verwendeten Namespaces. Dadurch erspart man sich einiges an Schreibarbeit:

```
using Autodesk.AutoCAD.Runtime;
using Autodesk.AutoCAD.DatabaseServices;
using Autodesk.AutoCAD.ApplicationServices;
using Autodesk.AutoCAD.EditorInput;
using Autodesk.AutoCAD.Geometry;
```

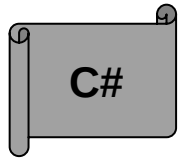
C++

Der einfachste Weg, neue VS-Projekte für eine ARX-Entwicklung einzurichten, ist die Nutzung des ObjektARX-Wizards.
Er befindet sich innerhalb des ObjectARX-Unterverzeichnisses **\ utils** in der gepackten Datei **\ ObjARXWiz**.
Nach dem Auspacken startet man das Installationpaket **ArxWizards.msi** (Achtung, es muß zu diesem Zeitpunkt natürlich eine vollständige VS Version installiert worden sein!) .
Zu Beginn des Installationsvorgangs wird man nach einem Entwickler-Kennzeichen (RDS) gefragt. Hier sollte man z.B. **CGCAD** eingeben.

Wenn alles ordentlich gelaufen ist, kann man nach Öffnen von VS ein neues Symbol in dem Projekt-Toolbar finden.

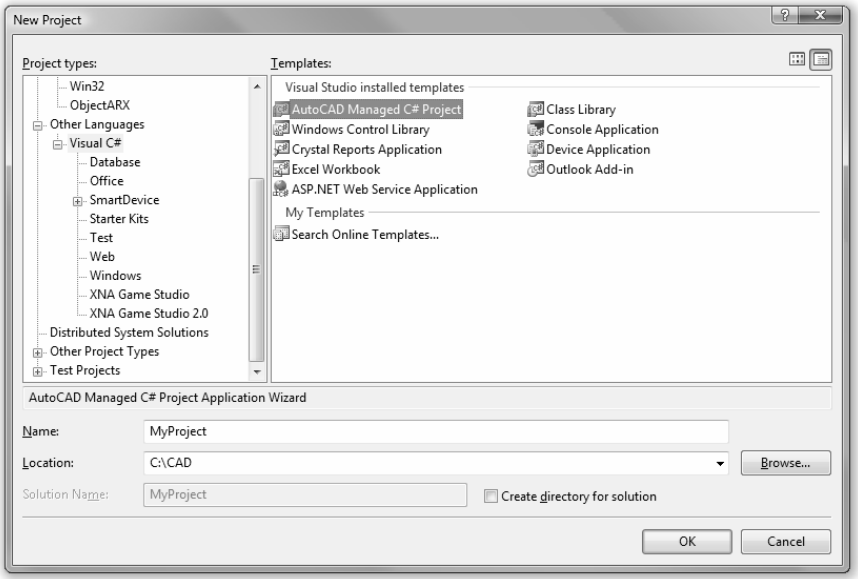


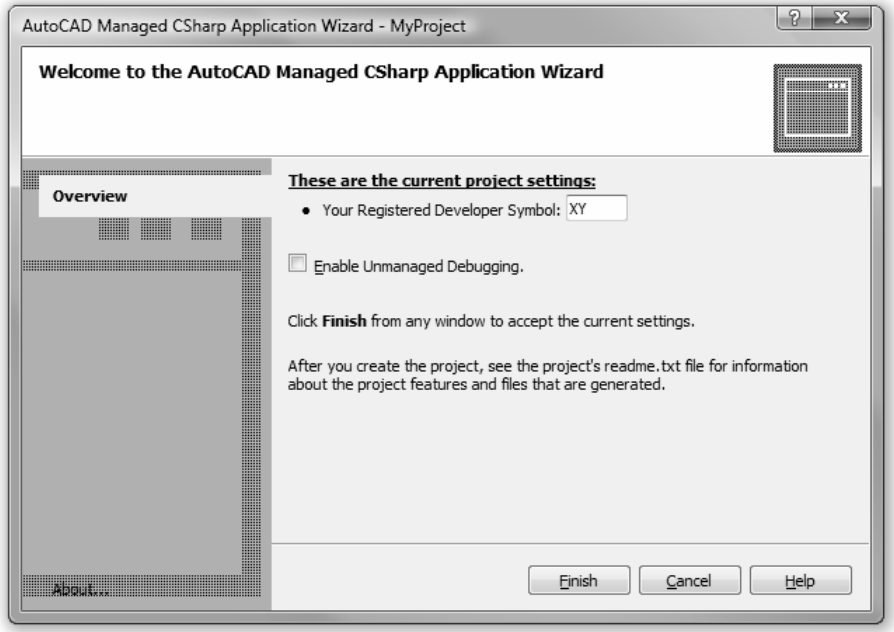
Um ein neues C#-Projekt zu erstellen, bietet sich das ObjectARX Wizard an. Es bindet die benötigten DLLs in das Projekt ein und gibt ein einfaches Grundgerüst vor, in dem leicht eigener Code einfügen kann. Außerdem ist der Debugger bereits richtig konfiguriert. Das Projekt lässt sich so direkt aus Visual Studio starten und auch debuggen. Um das Wizard zu starten wählen wir in Visual Studio 2005:



Datei ➔ Neu ➔ Projekt
Projekttyp: Andere Sprachen ➔ Visual C#
Vorlage: AutoCAD Managed C# Project

Anschließend muss nur noch ein Entwicklerkürzel gewählt werden. Die Option „Enable Unmanaged Debugging“ wird nicht benötigt. Nach dem Durchlauf des Wizards sind keine weiteren Anpassungen mehr nötig, der generierte Code kann direkt übersetzt und gestartet werden.





3.4.1. AutoCADs Datenbasis

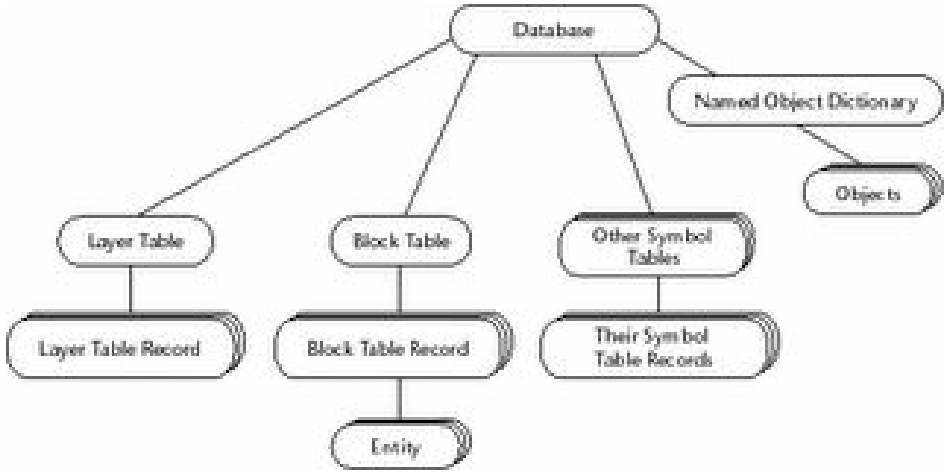
Jede AutoCAD Zeichnung stellt eine strukturierte Datenbasis dar, die verschiedene Typen von Objekten speichert. Wenn man eine neue Zeichnung öffnet, kreiert AutoCAD im Hintergrund eine wohl organisierte und leistungsfähige Datenbank. Diese Datenbank hat zunächst ein Minimum an Daten, die es erlauben, Zeichnungen mit den Basiselementen abzubilden.

Diese Basisdaten werden im Allgemeinen durch Objekte wie Layer, Linetypes, Textarten, etc. dargestellt. So werden z.B. im (default-)Layer 0 eine Standard-Textart, durchgezogene Linien und andere Eigenschaften definiert und gespeichert.

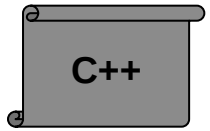
Diese Datenbasis verwaltet alle Objekte, aus denen eine Zeichnung bestehen muß. Diese Objekte werden in geeignete Container gespeichert, die erlauben, die Objekte gleicher Art zu handhaben. Auf diese Weise haben wir passende Methoden und Verfahren zum Speichern der Objekte, die zu dem entsprechenden Objekt passen.

Jedes Objekt, das in Datenbasis gespeichert wird, bekommt einen Bezeichner, die ObjectId. Dieser Bezeichner ist innerhalb der gleichen AutoCAD Session einzigartig und er ist während der gesamten Lebensdauer jedes Objektes gültig. Die ObjectId wird automatisch durch die Datenbasis erzeugt.

Innerhalb von ObjectARX haben wir zunächst drei Arten von Objekten:
Entities: Objekte mit graphischer Darstellung (Linien, Bögen, Texte,…);
Container: Spezielle Objekte zum Speichern und Handhaben von Tabellen (Layertabelle, Linetype-Tabelle,…);
Objekte: Objekte ohne irgendeine graphische Darstellung (Gruppen, Pläne,…).

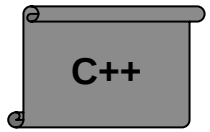


Kreieren von Objekten:
Um ein Objekt durch ObjectARX zu kreieren, haben wir eine Art Rezept, das abhängig ist, von welchem Typ dieses Objekt ist und wo wir es speichern möchten (meistens speichern wir Objekte innerhalb seiner spezifischen Container).
Im Allgemeinen gilt folgende Reihenfolge:



- **Deklariere einen Pointer zu dem später zu kreierenden Objekttyp und benenne ihn;**
- **Mit diesem Pointer rufe passende Methoden dieses Objektes auf, um seine Eigenschaften zu ändern;**
- **Hole einen Pointer zu der Datenbasis, in die Objekte kreiert werden sollen (meistens die aktuelle Datenbasis);**
- **Öffne den passenden Container, in dem sie gespeichert werden sollten;**
- **Rufe die spezifische Containermethode auf, um das Objekt zu speichern;**
- **Speichere den Rückkehrwert als ObjectId, die automatisch durch den Container erzeugt wird;**
- **Beende den Prozeß durch Schließen aller geöffneten Objekte einschließlich Container und kreierter Objekte.**

Es ist zu empfehlen. sich schnell einige handliche Hilfsklassen zu schreiben, die diese Vorgänge beinhalten, denn all diese Schritte sind doch sehr ähnlich. So könnte man also z.B. Datenbankhilfsfunktionen wie AddLayer, AddLine, AddCircle, AddTextStyle, etc. anlegen.



Es ist sehr wichtig nicht zu vergessen, geöffnete Gegenstände zu schließen, weil im negativen Fall AutoCAD abstürzen könnte.

Beispielcode zum Kreieren einer Linie (AcDbLine):
Dieser Code zeigt, wie man eine einfache Linie zwischen zwei Punkten erzeugt. Der Programmtext ist einfach und ohne jede Fehlerbehandlung. Dieser Code muß in eine ObjectARX Applikationsstruktur eingebettet werden, um zu arbeiten.

```
//zuerst deklarieren wir ein Paar Punkte
AcGePoint3d startPt(1.0,1.0,0.0);
AcGePoint3d endPt(100.0,100.0,0.0);

// jetzt instantiieren wir einen AcDbLine Zeiger
// der Konstruktor erlaubt mir, zwei Punkte mitzugeben
AcDbLine *pLine = new AcDbLine(startPt,endPt);

//jetzt deklarieren wir einen Zeiger auf den passenden Container, die BlockTable
AcDbBlockTable *pBlockTable = NULL;

// nun holen wir uns die aktuelle Datenbank und den Zeiger auf ihre BlockTable
AcDbDatabase* pDB = acdbHostApplicationServices()->workingDatabase();
pDB->getSymbolTable(pBlockTable,AcDb::kForRead);
```

C++

```
// innerhalb der BlockTable öffnen wir den ModelSpace
AcDbBlockTableRecord* pBlockTableRecord = NULL;
pBlockTable->getAt(ACDB_MODEL_SPACE,pBlockTableRecord,AcDb::kForWrite);

// danach schließen wir die BlockTable
pBlockTable->close();

//den ModelSpace-Zeiger verwenden wir, um unsere Linie zu addieren
AcDbObjectId lineId = AcDbObjectId::kNull;
pBlockTableRecord->appendAcDbEntity(lineId,pLine);

//um den Prozess zu beenden, schließen wir den ModelSpace und das Linien-Objekt
pBlockTableRecord->close();
pLine->close();
```

C#

Um Objekte aus der Datenbank zuzugreifen, seht wie bei C++ eine Methode **GetObject()** bereit. Über diese bekommt man Zugriff auf Objekte der Datenbank, wie z. B. die Block- oder die LayerTable. Eine separate Funktion für den Zugriff auf diese Objekte gibt es im Gegensatz zu C++ nicht.

Fehlerbehandlung

ObjectARX nutzt Rückgabewerte um Fehler anzuzeigen. Die .NET Wrapper-Klassen dagegen werden Fehler vorzugsweise durch das Werfen von Exceptions angezeigt. Für alle Fehlercodes gibt es eine entsprechende Exception. Es ist daher ratsam, seine Anweisungen in einen try-catch-Block zu verpacken und auch selbst Exceptions zur Fehlerbehandlung einzusetzen.

Beispielcode zum Kreieren einer Linie

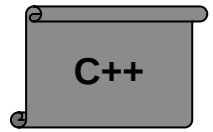
C#

```
Database db = Application.DocumentManager.MdiActiveDocument.Database;
Autodesk.AutoCAD.DatabaseServices.TransactionManager tm = db.TransactionManager;
Transaction transaction = tm.StartTransaction();
try
{
    // construct a new line
    Point3d startPt = new Point3d(1.0, 1.0, 0.0);
    Point3d endPt = new Point3d(100.0, 100.0, 0.0);
    Line line = new Line(startPt, endPt);

    // fetch blocktable from database
    BlockTable blockTable = (BlockTable)tm.GetObject(db.BlockTableId,OpenMode.ForRead,false);
    // open ModelSpace inside blocktable
    BlockTableRecord blockTableRecord = (BlockTableRecord)tm.GetObject(
                        blockTable[BlockTableRecord.ModelSpace], OpenMode.ForWrite, false);
    // use ModelSpace object to add line
    blockTableRecord.AppendEntity(line);
    // add line to database
    tm.AddNewlyCreatedDBObject(line, true);

    // commit our changes
    transaction.Commit();
}
catch (Autodesk.AutoCAD.Runtime.Exception)
{
    transaction.Dispose();
}
```

Im folgenden Abschnitt erzeugen wir eine ObjectARX Anwendungsstruktur mit dem o.g. Code und compilieren sie.

3.4.2. Struktur von Applications

Mit ObjectARX erzeugen wir praktisch eine DLL, die wahlweise MFC-Elemente beinhaltet, die während der Laufzeit von AutoCAD durch die gleichfalls hinzugefügten neuen Kommandoworte aktiviert wird.



ARX- (und DBX-)Module müssen eine Entry Point Function implementieren. Diese Funktion ist dafür verantwortlich, Nachrichten zwischen AutoCAD und der Applikation auszutauschen. Sie ersetzt quasi die main()-Funktion eines C++-Programms.
Diese Funktion implementiert folgende Funktionsdeklaration:

```
extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode Msg, void* pkt);
```

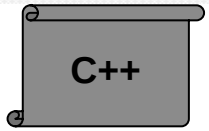
C++

Eine einfache Implementierung dieser Funktion ist:

```
extern "C" AcRx::AppRetCode acrxEntryPoint(AcRx::AppMsgCode Msg,void* pkt)
{
   switch(Msg)
   {
      case AcRx::kInitAppMsg:
         break;
      case AcRx::kUnloadAppMsg:
         break;
      default:
         break;
   }
   return AcRx::kRetOK;
}
```

Diese Funktion wird automatisch vom Wizard generiert.
Der erste Parameter (**`Msg`**) ist die Nachricht, die von AutoCAD an unsere Anwendung gesandt wird, um mitzuteilen, was geschehen ist. Das kann z.B. eine "neue Zeichnung"-Message, eine "init Application"-Message und viele andere sein. Diese Nachrichten sind wichtig, um die Applikation zu befähigen, auf jeden gewünschten Fall zu reagieren.
Der zweite Parameter (**`pkt`**) ist ein Datenpaket, das in einigen Situationen nützlich sein kann, auf die hier nicht weiter eingegangen werden muß.
Diese Funktion muß einen Wert an AutoCAD mit **`AppRetCode`** zurückliefern, z.B. **`kRetOK`** (üblicher Wert) oder **`kRetError`**, das AutoCAD zwingt, die Applikation zu entladen.

Noch einmal zur Erinnerung: diese Funktion ist sehr wichtig ist, und hier startet unsere Anwendung.



Registrieren von Befehlen

Vermutlich implementiert unsere Anwendung einige neue Befehle. Diese Befehlsworte können in AutoCAD registriert werden, wenn die acrxEntryPoint()-Funktion die kInitAppMsg-Anzeige empfangen hat, also AutoCAD uns mitteilt, dass die Applikation geladen wurde. Wenn wir diese Nachricht empfangen haben, können wir mit geeigneten Methoden jeden gewünschten Befehl registrieren.

Einzutragene Befehle müssen einen **`Group Name`**, einen **`Global Name`**, einen **`Local Name`**, einige Flags, einen **`void function pointer`** und optional einige andere Parameter haben.

Im Allgemeinen feuert der eingetragene Befehl die spezifizierte Funktion. Diese Funktion muß eine **`void`** Funktion ohne irgendwelche Parameter sein.

Wenn der Benutzer innerhalb von AutoCAD diesen Befehl aufruft, sucht AutoCAD das Befehlswort in seinem Befehlspuffer, findet diesen Befehl und feuert diese Funktion.

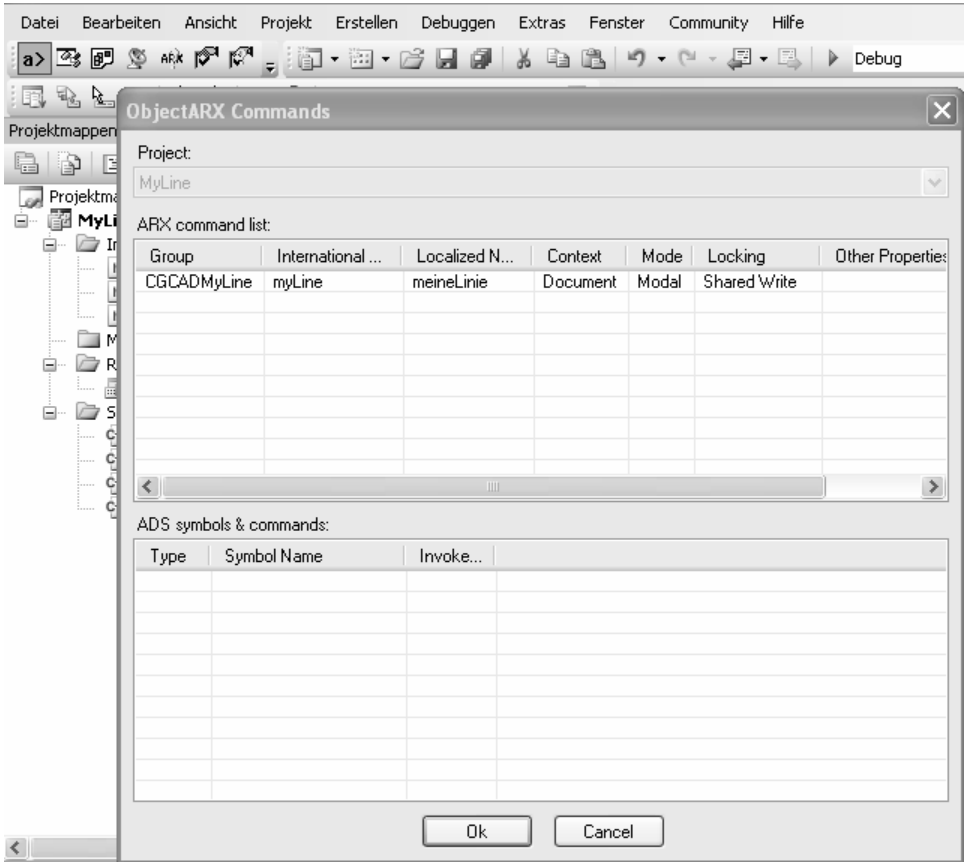
Es ist sehr wichtig, daß die eigenen neuen Befehle mit einem Präfix von 3 oder 4 Buchstaben beginnen, um Kollisionen mit anderen third-party applications zu vermeiden.

Die Hauptparameter sind:

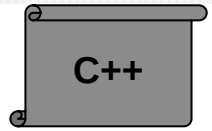
- **Group Name :** Erlaubt, die neuen Befehle einfachewr zu gruppieren, zu bilden, sie zu leeren und zu handhaben;
- **Global Name :** Der global (internationale) Name des Befehls;
- **Lokaler Name:** Der Befehlsname in der nationalen Übersetzung (z.B. deutsch);
- **Flags:** Sie können eine Kombination verschiedener Typen sein. Die zwei wichtigsten Flags sind **`ACRX_CMD_TRANSPARENT`** und **`ACRX_CMD_MODAL.`** Sie beeinflussen das Verhalten der Funktion, wie z.B. transparent (wie ZOOM) oder modal (wie Hauptbefehle);
- **void function pointer** : Hier wird der Name der leeren Funktion, die mit dem Befehl verbunden werden soll. Diese Funktion wird von AutoCAD gefeuert, wenn der Befehl aufgerufen wird.

C++

Auch bei diesem Schritt hilft uns der im folgenden Abschnitt beschriebene Wizard.
In der oberen Symboleiste (direkt unter "Datei") befindet sich ein neuer Button a> , nach dessen Anklicken sich ein Dialogfenster öffnet, das uns ermöglicht, die gewünschten neuen Kommandonamen zu definieren:

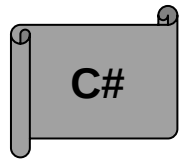


Der Kommando-Gruppenname wird aus dem Entwicklercode (s.vorn) und dem Projektnamen generiert und vorgegeben. Wir geben dann den internationalen Namen des Befehls, sowie den nationalen Namen (hier: myLine bzw. meineLinie) ein. Als Mode sollten wir Modal auswählen.
Mit dem OK-Button wird der Programm-code entsprechend erweitert.

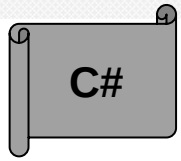


Ein einmal registrierter Befehl kann dann auch "unregistert" werden. Das wird dann geschehen, wenn AutoCAD verlassen wird oder die gesamte Befehlsgruppe entladen wird.
Wenn die **`kUnloadAppMsg`** an die Anwendung gesandt wird, könnte man weitere Lösch-operationen vornehmen.

Bemerkung: Das in **`/Beispiele`** bereitgestellte Projekt **`ArxProject`** zeigt, wie wir hier z.B. das Unlock definieren können oder die Multi-Dokument-Fähigkeit auswählen könnten.



Auch mit der .NET-Erweiterung von ObjektARX werden DLLs erzeugt. Diese können in AutoCAD nachgeladen und gestartet werden. Allerdings gibt es keinen festen Entry Point. Statt dessen werden lediglich die neuen Befehle registriert und bestimmten Methoden zugeordnet. Dabei wird kein extra Wizard benötigt, lediglich eine Zeile Code ist für die Zuordnung nötig.



Registrierung von Befehlen

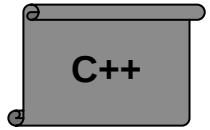
Der Code zur Zuordnung eines AutoCAD-Befehls zu einer C#-Methode sieht wie folgt aus:

```
[CommandMethod("MYAUTOCADCOMMAND")]
static public void MyMethod() {
   // …
}
```

Dabei ist zu beachten, dass die zugeordneten Methoden immer static deklariert sein müssen und keine Parameter oder einen Rückgabewert haben dürfen. Die Registrierung findet automatisch statt, wir müssen uns nicht darum kümmern.
Wie in C++ können auch zusätzliche Parameter übergeben werden wie ein lokalisierter Name oder Flags:

```
[CommandMethod("groupName", "globalName", "localizedName",
CommandFlags.Modal)]
static public void MyMethod() {
   // …
}
```

Ein „unregistieren“ wie bei C++ ist übrigens nicht möglich, genauso wie das „entladen“ des DLLs. Dies ist eine Limitierung des .NET Frameworks, das kein dynamisches Entladen von DLLs erlaubt.

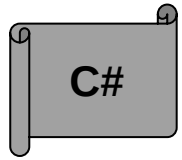


Starten der Anwendung

Wenn das Programm erfolgreich übersetzt wurde, können wir es in AutoCAD laden und starten. (Zu bemerken ist, dass ein Programmtest tatsächlich nur mit AutoCAD erfolgen kann.)
Die einfachste Methode zum Laden ist das **APPLOAD** –Kommando, das einen Dialog öffnet, mit dessen Hilfe die arx-Datei ausgewählt und geladen bzw. entladen werden kann.

Sobald die Anwendung geladen ist, kann durch Auslösen (Eintippen) des entsprechenden neuen Befehlswortes das Programm gestartet werden.
Das im Unterverzeichnis **/Beispiele** bereitgestellte VC-Projekt **MyLine** zeigt die Einbettung der im Abschnitt 3.4.1. begonnenen SimpleLine-Anwendung.

Starten der Anwendung



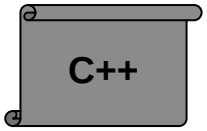
Nach dem Übersetzen kann das DLL in AutoCAD geladen werden. Dazu muss in der AutoCAD Kommandozeile das Kommando **NETLOAD** eingegeben werden. Es erscheint ein Auswahldialog, in dem das gewünschte DLL bestimmt werden muss. Ist das Programm geladen, stehen alle selbst definierten Befehle sofort zur Verfügung und können in der AutoCAD-Kommandozeile eingegeben werden.
Wie bereits erwähnt, existiert kein **UNLOAD** Befehl für das Entladen des .NET DLLs. Dies erschwert die Entwicklung etwas, da AutoCAD nach jeder erneuten Übersetzung des DLLs neu gestartet werden muss. Eine Lösung für das Problem ist nicht bekannt und stellt einen der gravierenden Nachteile der .NET-Benutzung dar.

Minimalanwendung:

```
using System;
using Autodesk.AutoCAD.Runtime;
using Autodesk.AutoCAD.EditorInput;
using Autodesk.AutoCAD.ApplicationServices;

namespace MyProject
{
    public class MyCommands
    {
        // define AutoCAD Command "helloWorld"
        [CommandMethod("helloWorld")]
        static public void test()
        {
            Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
            ed.WriteMessage("Hallo World");
        }
    }
}
```

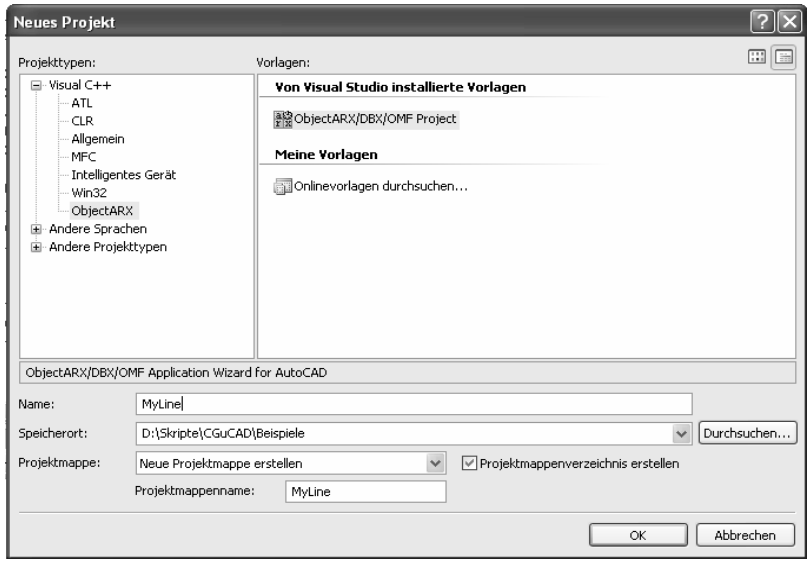
3.4.3. Einrichten einer Application mit dem Wizard

Nachdem der in dem ObjectARX SDK mitgelieferte Wizard wie vorn beschrieben installiert wurde, wollen wir ihn nun nutzen, um den Code der SimpleLine selbst als ARX-Projekt zu implementieren.

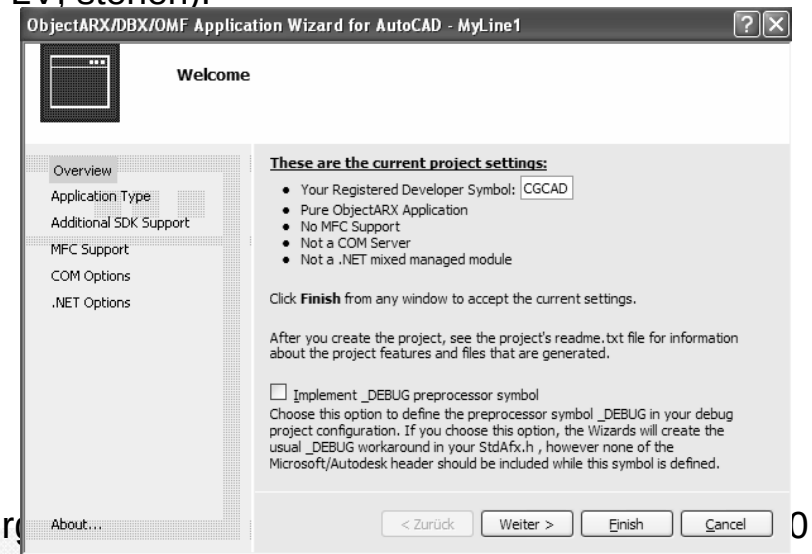


Nach dem Starten von MS VisualStudio 2005 wählen wir das Kommando **`Datei → Neu → Projekt.`**

In der nun sich öffnenden Dialogbox **`Neues Projekt`** wählt man den Eintrag **`Visual C++ → ObjectARX`** im Projekttypenbaum und dann das Icon **`ObjectARX/DBX/OMF Project`**. Anschließend gibt man den Namen des Projektes und den Speicherort ein. Dann OK drücken!



Im nun folgenden Dialog Overview wird zunächst der **RDS** (**Registered Developer Symbol**) –Name eingegeben, der verhindern soll, dass eine andere Entwicklung gleiche Kommando- und Funktionsnamen vergibt (hier lasse ich die Vorgabe **`CGCAD`**, also den Namen der LV, stehen).

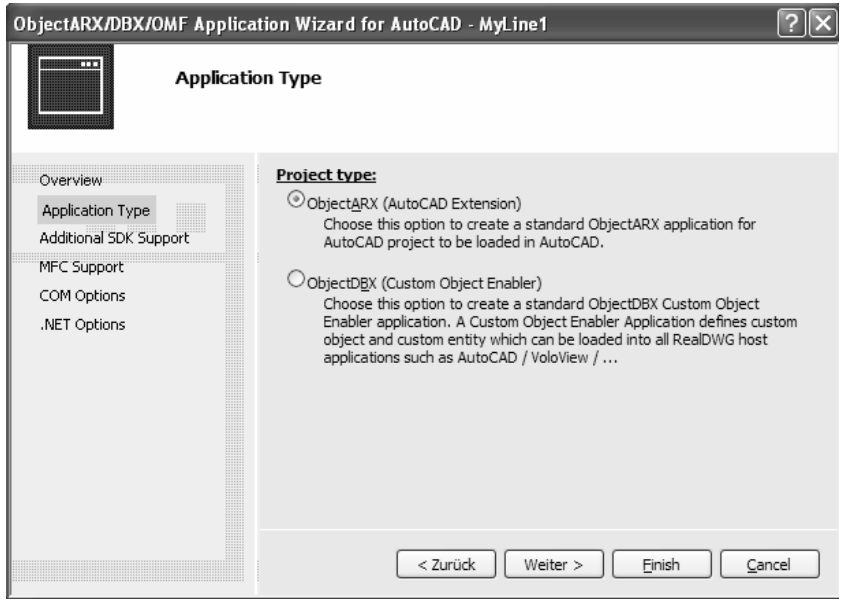


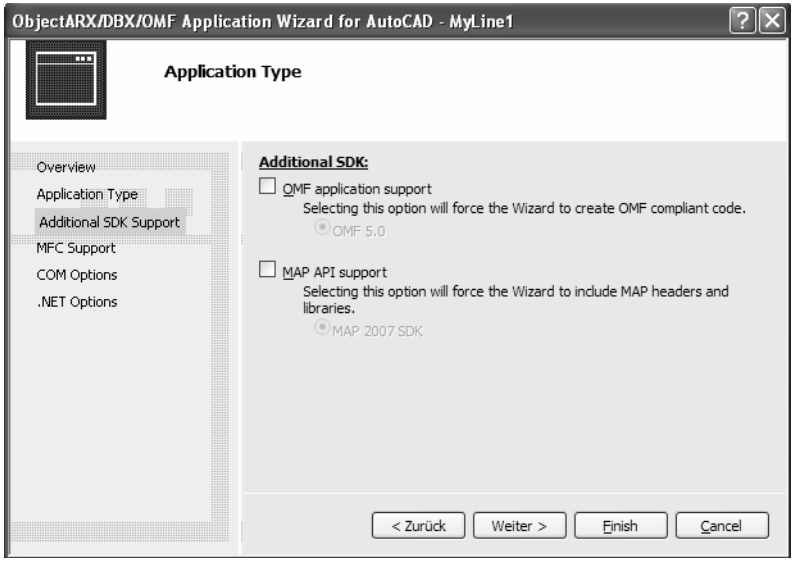
Das _DEBUG Processorsymbol kann wahlweise eingeschaltet werden, zumindest dann, wenn später ein Debug-Lauf unter AutoCAD gewünscht wird.

C++

Im nächsten Dialog wird der Application Typ ausgewählt. In unseren Aufgaben wird das immer ein ObjectARX-Projekt sein.

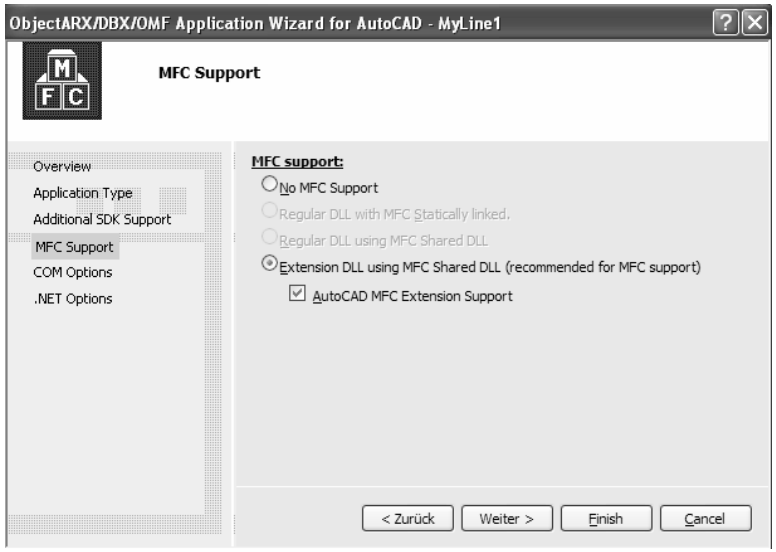
Im nächsten Schritt könnte man erweiterte Entwicklungsunterstützung beantragen, wenn wir z.B. eine Erweiterung des Autodesk Architectural Desktop (OMF) vorhaben oder Autodesk Map erweitern wollten.



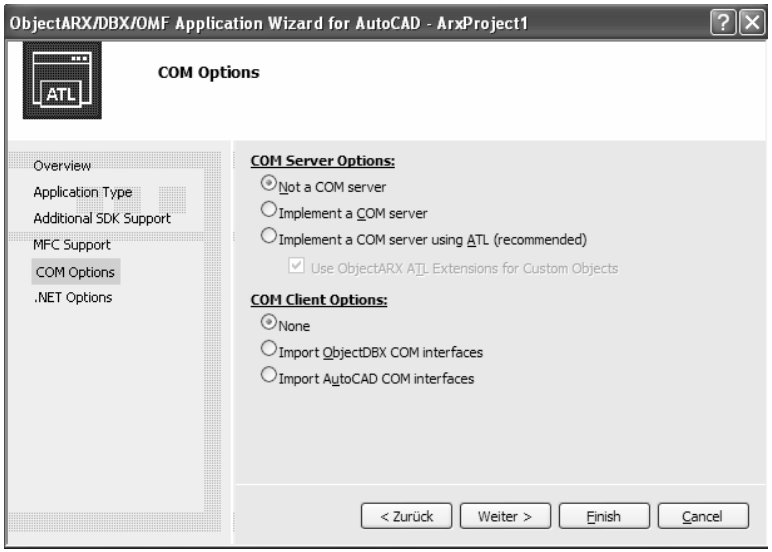


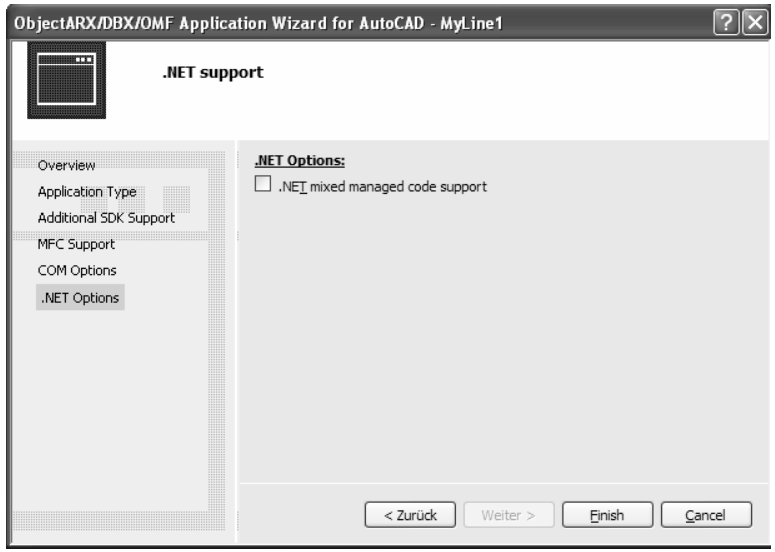
Im nun folgenden Dialog können wir auswählen, ob unsere Anwendung die Fähigkeiten der MFC unter AutoCAD nutzen will. Das sollten wir immer dann einschalten, wenn beabsichtigt ist, grafische Oberflächenelemente, wie ComboBox, Dialogs usw., innerhalb unseres Programms zu integrieren.

C++

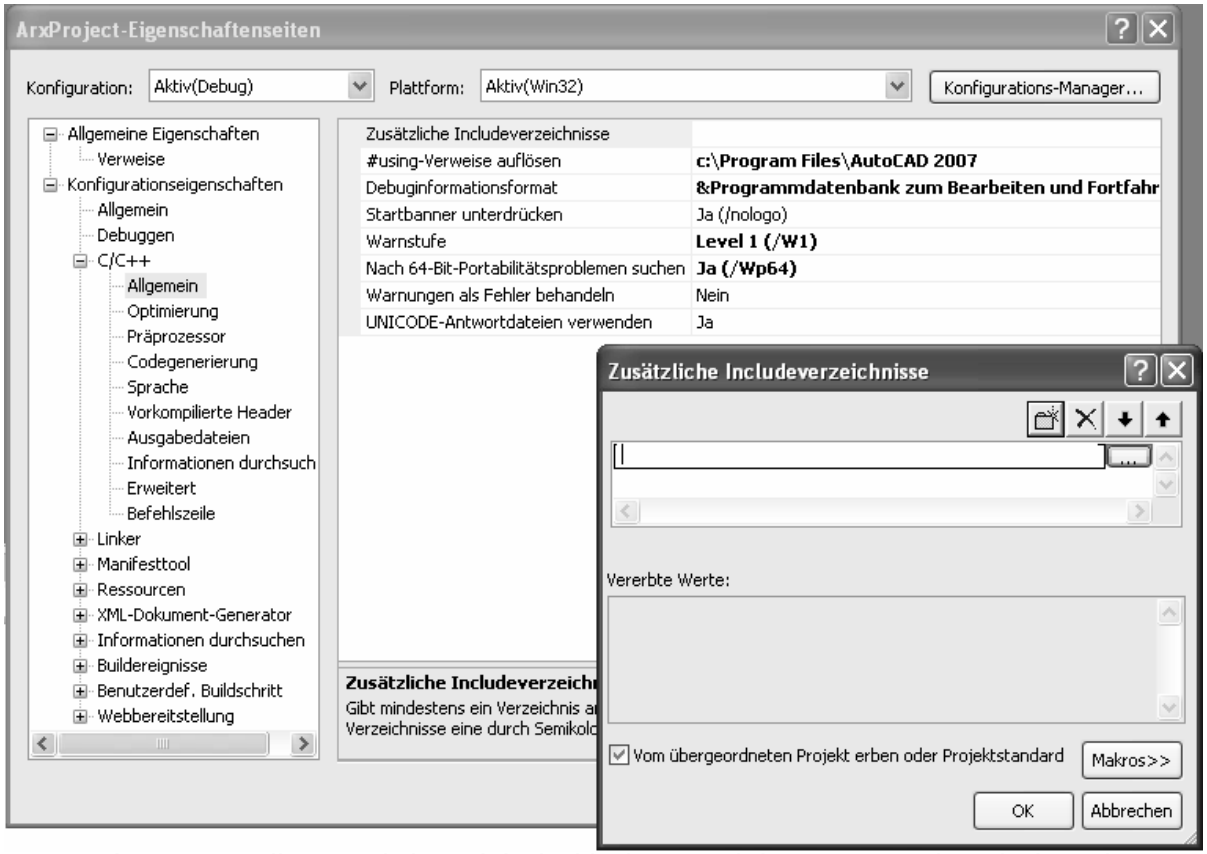


Die nächsten beiden Boxes fragen uns nach der beabsichtigten COM-Fähigkeit unserer Application und nach dem .NET –Support.
Beides benötigen wir für unsere Anwendungen nicht, also **`Finish`**!





Nicht vergessen dürfen wir die Verknüpfung der Arx-Include-Files und –Libraries mit dem VS-Projekt.
Dazu wählen wir aus dem Menü **`Projekt → Eigenschaften`**.
Im folgenden Dialog öffnen wir im Baum **`Konfigurationseigenschaften → C/C++`** und wählen **`Allgemein`** aus.

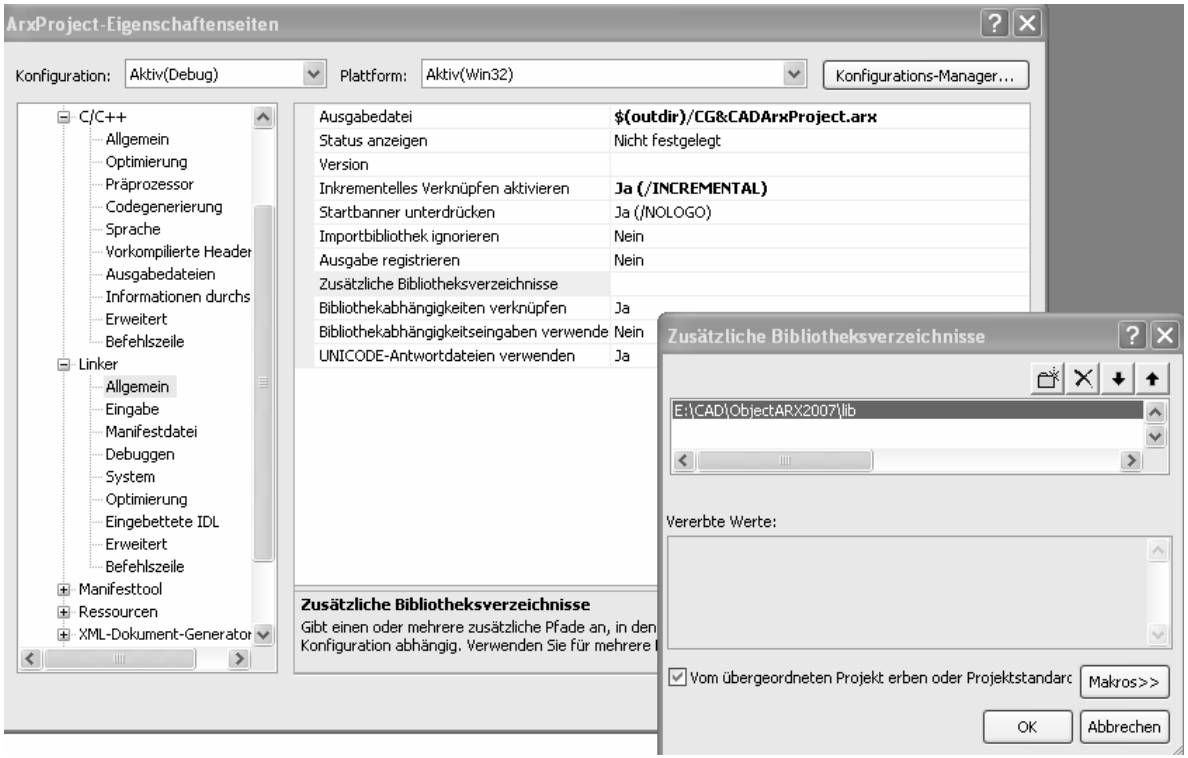


Jetzt klicken wir auf die drei Punkte am Ende der ersten Zeile (Zusätzliche Include-Verzeichnisse). Im nun sich öffnenden Fenster müssen wir mit dem Symbol "Neue Zeile" eine freie Zeile markieren und dann wieder die dortigen drei Punkte anklicken, um das Navigationsfenster zu öffnen.
Hier suchen wir das Object ARX2007-Verzeichnis /inc und öffnen es.

Bemerkung: Um diese Schritte nicht bei jedem Projekt erneut zu bearbeiten, können diese Angaben auch permanent über **`Extras → Optionen`**, die Knoten **`Projekte`** und **`Projektmappen→VC++-Verzeichnisse`** und die Auswahl **`Includedateien`** bzw. **`Bibliotheksdateien → Neue Zeile`** eingegeben werden.

C++

Analog verfahren wir mit den Bibliotheken:
`Linker → Allgemein` auswählen,
die drei Punkte in der Zeile **`Zusätzliche Bibliotheksverzeichnisse`** anklicken,
neue Zeile mit dem Symbol einfügen, drei Punkte der neuen Zeile anklicken und den ObjectARX2007-Pfad **`/lib`** öffnen.



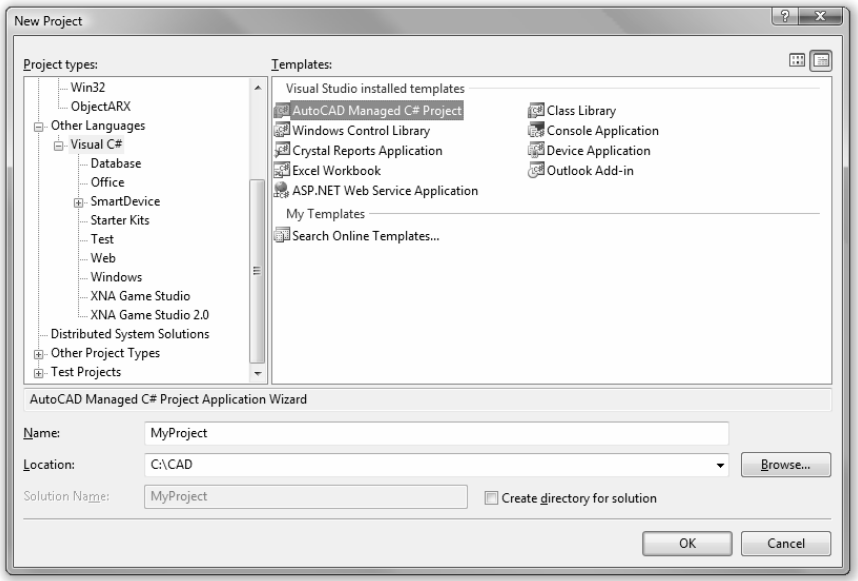
Nun sind wir soweit vorbereitet, um den neuen Code, also die neue Funktionalität, in das Skelett der Methode **`acrxEntryPoint.cpp`** hineinzuprogrammieren.

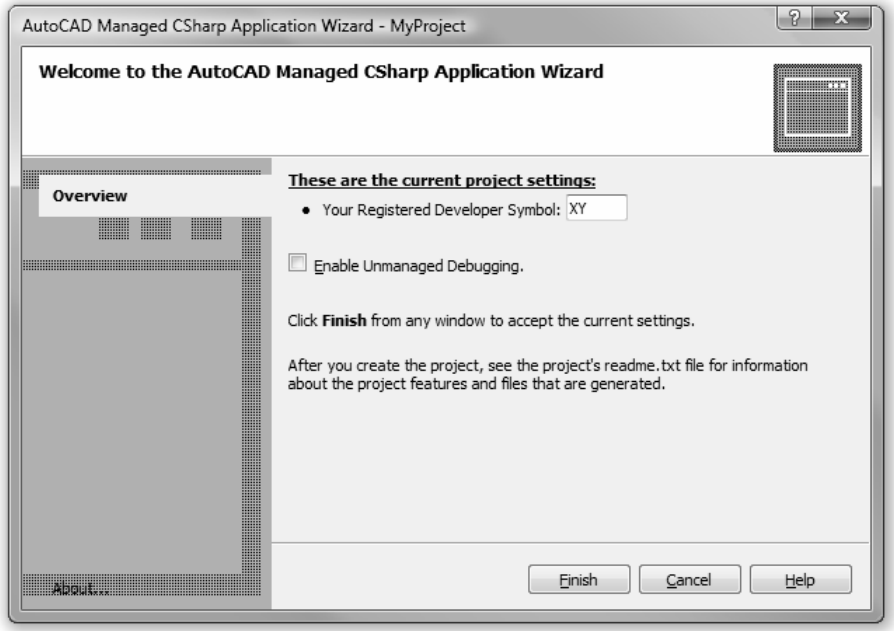
C#

Um ein neues Projekt zu erstellen, bietet sich das ObjectARX Wizard an. Es bindet die benötigten DLLs in das Projekt ein und gibt ein einfaches Grundgerüst vor, in dem leicht eigener Code eingefügt werden kann. Außerdem ist der Debugger bereits richtig konfiguriert. Das Projekt lässt sich so direkt aus Visual Studio starten und auch debuggen. Um das Wizard zu starten wählen wir in Visual Studio 2005:

`Datei ➔ Neu ➔ Projekt`
Projekttyp: **`Andere Sprachen ➔ Visual C#`**
Vorlage: **`AutoCAD Managed C# Project`**

Anschließend muss nur noch ein Entwicklerkürzel gewählt werden. Die Option „Enable Unmanaged Debugging“ wird nicht benötigt. Nach dem Durchlauf des Wizards sind keine weiteren Anpassungen mehr nötig, der generierte Code kann direkt übersetzt und gestartet werden.



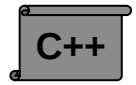


3.4.4. Objekt Lifecycle

Wie vorn schon angemerkt, ist die AutoCAD Datenbasis präzis und effektiv organisiert und erlaubt eine einfache und direkte Manipulation ihrer Objekte. Grundsätzlich gibt es zwei Basistypen: Container und Objekte.

Container sind spezielle Objekte, die einen einfachen Mechanismus zum Speichern und Editieren von persistenten Objekten oder Collections liefern. Jeder Objekttyp hat einen passenden Container, in den wir ausschließlich Objekte dieses Typs speichern sollten.
Es gibt verschiedene Container in der AutoCAD-Datenbasis, wobei die LayerTable, die LinetypeTable und die BlockTable die wichtigsten sind. Jede Container-Klasse hat Standard-Zugriffsmethoden und meist auch Methoden zum Iterieren durch seine Elemente(Objekte).
Jedes von einer Application erzeugte Objekt, das in die AutoCAD-Datenbasis bespeichert werden soll, muß mit den dafür vorgesehenen Methoden des Container gehandhabt werden.

Objekte (einschließlich Entities) sind meist Basistypen und repräsentieren jedes (grafische) Element innerhalb von AutoCAD. Sie sind implementiert durch spezifische Klassen mit ihren Standard- und spezifischen Methoden. Eine Vererbung in eigene Subklassen ist zwecks Anpassung an spezielle Anforderungen möglich.
Jedes in der Datenbasis existierende Objekt hat einen exklusiven Identifikationsschlüssel, den ObjectId. Diese Identifikation ist der "Name" des Objektes innerhalb der Datenbasis und wird benötigt, um das Objekt zu referieren, zu öffnen und zu manipulieren.



Achtung, entgegen anderen zwingenden Programmierregeln in C++, innerhalb von ObjectARX darf ein Pointer zu einem (persistenten) Objekt in der Datenbasis niemals gelöscht werden. Alle weiteren Aufgaben der dynamischen Speicherverwaltung übernimmt AutoCAD.

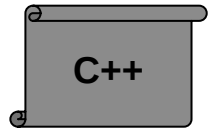
Wie kann nun ein Objekt manipuliert werden?
Folgende drei Regeln müssen dabei unbedingt beachtet werden:

- Existierende Objekte der Datenbasis sollten niemals gelöscht werden, es sei denn, sie sind wirklich zu löschen!
- Wenn ein Objekt zwar allokiert ist, doch noch nicht zur Datenbasis addiert wurde, dann muß der Pointer gelöscht werden.
- Wenn man den Pointer eines existierenden Objektes benötigt um es zu manipulieren, verwendet man seinen ObjectId und nutzt geeignete Containermethoden, um den Pointer dieses Objektes zu bekommen. Nach der Bearbeitung muß das Objekt unbedingt durch Aufruf seiner close()-Methode geschlossen werden und so die Transaktion beendet und AutoCAD darüber informiert werden. (Falls man das vergißt, quittiert das AutoCAD mit einem Absturz!)

Object Ownership und Relationship

Objekte referieren jedes andere Objekte durch die entsprechende ObjectId. Das kann eine Ownership relation oder eine sonstige Referenz sein.
Das wollen wir anhand eines zu definierenden Layers (Ebene, Folie) kennen lernen. Der Container für die Layer-Tabelle enthält Einträge (Objekte) für Layer. Jede Layer-ObjectId wird innerhalb der Entities, die auf diesem Layer liegen, referenziert. Darum kann man z.B. ein Layer nicht aus der Zeichnung löschen, solange es noch irgendein Entity auf diesem Layer gibt.

So gibt es in AutoCAD verschiedene andere Zusammenhänge von Baisisobjekten, die die Ownerships und die Referenzierung analog abbilden.



Erzeugen eines neuen Layers

Um die zuvor erzeugte "MeineLinie" sofort auf einem eigenen Layer zu "heben", erweitern wir das begonnene Beispiel um folgenden Code:

```
AcDbLayerTable* pLayerTbl = NULL;
// Get the current Database
AcDbDatabase* pDB = acdbHostApplicationServices()->workingDatabase();

// Get the LayerTable for write because we will create a new entry
pDB->getSymbolTable(pLayerTbl,AcDb::kForWrite);

// Check if the layer is already there
if (!pLayerTbl->has(_T("MYLAYER"))) // _T for char Unicode convert
{
   // Instantiate a new object and set its properties
   AcDbLayerTableRecord *pLayerTblRcd = new AcDbLayerTableRecord;
   pLayerTblRcd->setName(_T("MYLAYER"));
   pLayerTblRcd->setIsFrozen(0); // layer set to THAWED
   pLayerTblRcd->setIsOff(0);    // layer set to ON
   pLayerTblRcd->setIsLocked(0); // layer un-locked
   AcCmColor color;
   color.setColorIndex(10);      // set layer color to red
   pLayerTblRcd->setColor(color);
```

```
   // Now, add the new layer to its container
   pLayerTbl->add(pLayerTblRcd);

   // Close the new layer (DON'T DELETE IT)
   pLayerTblRcd->close();

   // Close the container
   pLayerTbl->close();
}
else
{
   // If our layer is already there, just close the container and continue
   pLayerTbl->close();
   acutPrintf(_T("\nMYLAYER already exists")); //output in the status line
}
```

C++

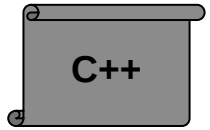
C#

Da unter .NET lediglich mit Wrapperklassen gearbeitet wird und im Hintergrund nach wie vor Unmanaged Klassen arbeiten, gelten die selben Grundsätze wie beim Einsatz von C++. Aus diesem Grund sollte man sich auch nicht darauf verlassen, dass die Garbage Collection von .NET nicht mehr benötigte Ressourcen wieder frei gibt. Statt dessen sollte immer die Funktion Dispose() aufgerufen werden wenn ein Objekt gelöscht werden soll. Diese hat die selbe Funktion wie der Destruktor in C++.

3.4.5. Object Management

Wie schon vorn beschrieben, hat jedes Objekt seinen eigenen Identifikationsschlüssel, die **`ObjectId`**. Dieser Wert ist die Basis für die Beschaffung des Objekt-Pointers und danach für die Ausführung von Lese- und Schreiboperationen.

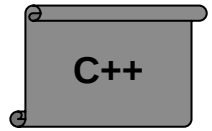
Die übliche Zugriffsmethode ist die entsprechende **`Open(…)-`**Methode mit der Spezikation (Mode) zum Lesen, zum Schreiben etc. und am Ende die korrespondierende **`Close()-`** Methode.
Mehr effizient und sicher ist die Manipulation von Objekten durch Transaktionen.
Beide Techniken werden hier kurz vorgestellt.



Die OPEN / CLOSE Methoden

Obwohl das die meist genutzte Technik ist, birgt sie die größte Fehlerwahrscheinlichkeit, weil z.B. beim Vergessen des **`Close()`** und anderen Unzulänglichkeiten es zu unerwarteten Abstürzen von AutoCAD führen kann.
Die globale Standardfunktion zum Öffnen eines Objektes ist **`acdbOpenObject().`**
Dieses Open wird jedes Objekt, das von der Klasse AcDbObject abgeleitet wurde, öffnen und einen C++-Pointer liefern, mit dessen Hilfe auf Attribute und Methoden der Klasse zu. gegriffen werden kann. Eine mögliche Signatur der Methode lautet wie folgt:

```
inline Acad::ErrorStatus acdbOpenObject(AcDbObject *& pObj,
                                        AcDbObjectId id,
                                        AcDb::OpenMode mode,
                                        bool openErased);
```



mit

pObj	**Output pointer to the opened object**
Id	**Input the object ID of the object to open**
mode	**Input mode to open object**
openErased	**Input Boolean indicating whether it's OK to open an erased object**

Diese Methode liefert einen Pointer auf ein AcDbObject, der nicht leer ist, wenn die übergebene Id in der AutoCAD-Datenbasis existiert. Beim Öffnen wird der Zugriffsstatus. also nur zum Lesen (READ), zum Schreiben (WRITE) oder zum NOTIFY, angegeben. Der letzte Parameter gibt an, dass das Objekt auch geöffnet werden kann, wenn es bereits im AutoCAD-Lösch-Status ist (Ein gelöschtes Objekt bleibt während einer AutoCAD-Sitzung wegen der komfortablen Undo-Fähigkeiten noch in der Datenbasis gespeichert bis das Zeichnungsfile gespeichert wird.).

Der Zugriffsstatus READ verhindert, dass Methoden des Objektes zum Modifizieren seines Status aufgerufen werden. Sollte man nun einfacherweise immer mit WRITE öffnen?
Auf keinen Fall!
Bei Schreibberechtigung fährt AutoCAD spezielle Pre- und Post-Prozeduren, die die Performance beträchlich senken.

C++

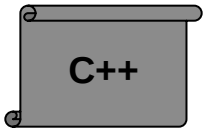
Obwohl man ein und dasselbe Objekt bis zu 256 mal parallel zum READ öffnen kann und bis zu zwei Mal zum WRITE bevor das entsprechende close() erfolgen muß, sollte man immer versuchen, ein geöffnettes Objekt sobald als möglich zu schließen.

Die beste Performance kann man erzielen, wenn grundsätzlich ein Objekt erst einmal zum Lesen geöffnet wird, dann analysiert wird, ob man es auch modifizieren will, und wenn ja, es zum Schreiben geupgradet wird (mit **`upgradeOpen()`** vom Status READ zum Status WRITE, und umgekehrt mit **`downgradeOpen()`** zurück zum Status READ.
Das folgende Beispiel zeigt eine einfache Anwendung des Open:

```
void changeColor(AcDbObjectId id)
{
   AcDbEntity* pEnt = NULL;
   if (acdbOpenObject(pEnt, id, AcDb::kForRead) == Acad::eOk)
   {
      if (pEnt->colorIndex() != 3)
      {
         pEnt->upgradeOpen();
         pEnt->setColorIndex(3);
      }
      else
      {
          acutPrintf(_T("\nEntity already has color=3"));
      }
      pEnt->close();
   }
}
```

Die Anwendung von Transaction -Methoden

Transactions erlauben den besseren und effizienteren Zugriff auf Objekte. Sie können verschachtelt benutzt werden (sogar paralleles Öffnen zum Schreiben und zum Lesen ist möglich) und sind besser für komplexere Manipulationen geeignet.
Man muß zu Beginn eine Transaktion öffnen und das Objekt besorgen (quasi eine Kopie des derzeitigen Datenbasis-Objektes), kann dann die gewünschten Modifikationen an dieser Kopie vornehmen und schließlich wird es mit **`end()`** zurückgespeichert bzw. mit **`abort()`**die Veränderung verworfen.
Insbesondere für die Manipulation von Objekten durch Nutzereingaben in Dialogboxes ist dieses Konzept geeignet, beim Öffnen der Dialogbox wird die Transaktion gestartet, die eingegebenen Werte werden in die Kopie übertragen und bei "OK" bzw. "CANCEL" wird die entsprechende Transaktionsdmethode aufgerufen.



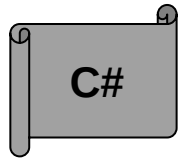
Man darf ein so geöffnetes Objekt **nicht** mit **`close()`** schließen!
Allein die **`end()-`** bzw. **`abort()-`**Methoden schließen die betroffenen Objekte automatisch.
Aus diesem Grund sollten die open()-/close().Technik auch niemals mit der Transaktionstechnik gemischt werden!
Nach Öffnen der Transaktion wird mit getObject() der Pointer zu diesem Objekt geholt (analog zu dem acdbOpenObject() in der vorherigen Technik).

Das folgende Beispiel implementiert die schon vorn gesehen Funktionalität mit der Transaktions-Methode:

C++

```
void changeColor(AcDbObjectId id)
{
   AcDbEntity* pEnt = NULL;
   acdbTransactionManager->startTransaction();
   if (acdbTransactionManager->getObject((AcDbObject*&)pEnt, id, AcDb::kForRead)
       == Acad::eOk)
   {
      if (pEnt->colorIndex() != 3)
     {
        pEnt->upgradeOpen();
        pEnt->setColorIndex(3);
     }
     else
     {
        acutPrintf(_T("\nEntity already has color=3"));
     }
   }
   acdbTransactionManager->endTransaction();
}
```

Achtung! All diese einfachen Beispiele sind natürlich nicht fehlersicher. So könnte z.B. das geholte Objekt kein Entity sein und so die nachfolgende Manipulation fehlschlagen.
Um so etwas abzufedern, könnte man die **`cast()-`**Methoden der Entities nutzen!



Transaktionen

Unter .NET gilt das Konzept der **OPEN/CLOSE-**Methoden als veraltet. Hier wird ausschließlich auf Transaktionen gesetzt. Sie funktionieren auf die selbe Art und Weise wie unter C++.

Transaktionen werden über den Transaktionsmanager mit der Methode **StartTransaction()** gestartet. Anschließend können Objekte aus der Datenbank gelesen und geschrieben werden. Das Abschließen der Transaktion ist mit **Commit()** möglich. Ist ein Fehler aufgetreten oder sollen alle Änderungen zurückgenommen werden kann die Transaktion mit **Dispose()** abgebrochen werden.

Um auf Objekte aus der Datenbank zuzugreifen, steht wie bei C++ eine Methode **GetObject()** bereit. Über diese bekommt man Zugriff auf Objekte der Datenbank, wie z. B. die Block- oder die LayerTable. Eine separate Funktion für den Zugriff auf diese Objekte gibt es im Gegensatz zu C++ nicht.

C#

Beispielcode zum Erzeugen eines neuen Layers

```
Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
Database db = Application.DocumentManager.MdiActiveDocument.Database;
Autodesk.AutoCAD.DatabaseServices.TransactionManager tm = db.TransactionManager;
Transaction transaction = tm.StartTransaction();
try {
    // get the LayerTable
    LayerTable layerTable = (LayerTable)tm.GetObject(db.LayerTableId,OpenMode.ForRead,false);

    // check if layer is already there
    if (!layerTable.Has("MYLAYER")) {
        LayerTableRecord layerTableRecord = new LayerTableRecord();
        layerTableRecord.Name = "MYLAYER";
        layerTableRecord.IsFrozen = false; // layer set to THAWED
        layerTableRecord.IsOff = false;    // layer set to ON
        layerTableRecord.IsLocked = false; // layer un-locked
        layerTableRecord.Color = Color.FromRgb(255, 0, 0); // set layer color to red

        // Now, add the new layer to its container
        layerTable.UpgradeOpen();
        layerTable.Add(layerTableRecord);
        tm.AddNewlyCreatedDBObject(layerTableRecord, true);
    }
    else {
        editor.WriteMessage("\nMYLAYER already exists");
    }

    transaction.Commit();
}
catch (Autodesk.AutoCAD.Runtime.Exception) {
    transaction.Dispose();
}
```

C#

Beispielcode zum Ändern der Farbe eines Entities

```
static public void changeColor(ObjectId id)
{
    Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    Autodesk.AutoCAD.DatabaseServices.TransactionManager tm = db.TransactionManager;
    Transaction transaction = tm.StartTransaction();
    try
    {
        Entity entity = (Entity)tm.GetObject(id, OpenMode.ForRead, false);
        if (entity.ColorIndex != 3)
        {
            entity.UpgradeOpen();
            entity.ColorIndex = 3;
        }
        else
        {
            editor.WriteMessage("\nEntity already has color=3");
        }
        transaction.Commit();
    }
    catch (Autodesk.AutoCAD.Runtime.Exception) {
    {
        transaction.Dispose();
    }
}
```

3.4.6. Entities

Entities sind Objekte und haben eine grafische Repräsentation. In Abhängigkeit von ihren Eigenschaften und Funktionalität unterscheidet man einfache und komplexe Entities. Entities werden in einem **BlockTableRecord-** Container gespeichert. Jeder dieser Container beinhaltet seine Entities bis sie gelöscht wurden oder die Datenbasis zerstört wird. Wie jedes andere Datenbasis-Objekt wird ein Eintity durch seine unique **`ObjectId`** identifiziert. Einige spezielle Entities enthalten andere Objekte, um so die Implementierung und Handhabung zu vereinfachen. Ein Beispiel dafür ist die **`AcDb3dPolyline`**, die eine Sammlung von **`AcDb3dPolylineVertex`** –Objekte enthält..

Entity Properties

AutoCAD-Entities haben verschiedene Eigenschaften, wobei einige von ihnen für alle Entities gelten. Diese gemeinsamen Attribute werden in der Basisklasse **`AcDbEntity`** gespeichert. Diese Klasse erbt nun wieder von der Klasse **`AcDbObject`** und implementiert eine Reihe von gemeinsamer Funktionalität.
Wenn wir z.B. einen Kreis kreieren (**`AcDcCircle`**), wird er folgende geerbte Eigenschaften von **`AcDbEntity`** besitzen:

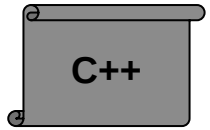
`Color  Linetype  Linetype scale  Visibility  Layer`
`Line weight  Plot style name`

Diese Eigenschaften haben spezielle Zugriffsmethoden (aus der **`AcDbEntity`** –Basisklasse), die es erlauben, ihre Werte zu lesen oder zu modizieren.
Wenn wir die Farbe eines AcDbCircle-Entities ändern wollen, öffnen wir dieses Entity zum Schreiben, nutzen die geignete Methode, um die Farbe zu ändern und schließen es wieder.

Über die vielen Möglichkeiten, Entities zu konstruieren bzw. mit ihnen zu arbeiten, informiert ausführlich die ObjectArx-Hilfe(**`ObjectARX2007\docs\arxdoc.chm`**).
Die folgende Liste zeigt lediglich die im Praktikum benötigten Objekttypen und soll so das Auffinden in der Hilfe erleichtern:

`AcDbPolyline`	Polylinie, also eine Folge von Knoten, die linear oder durch einen Kreisbogen verbunden sind
`AcDb3dPolyline`	3D-Polyline als Folge von Knoten im Raum, die linear verbunden sind
`AcDb3dPolylineVertex`	Knoten in einer 3D-Polyline
`AcDbArc`	Kreisbogen
`AcDbCircle`	Vollkreis
`AcDbFace`	3D-Flächenelement, definiert durch drei Eckpunkte
`AcDbLine`	Strecke (Gerade)
`AcGePoint3d`	Punkt
`AcDbText`	grafischer Text
`AcDbObjectIterator`	Iterator durch eine Tabelle/Liste

Beispiel: Auslesen einer Polyline (das Selektieren der gewünschten Polyline wird in Abschnitt 3.4.7. erläutert):



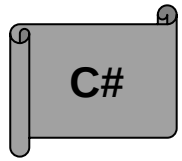

```
void iterate(AcDbObjectId id)
{
   AcDbEntity * pEntity = NULL;
   AcDbPolyline *pPline;
   AcDb3dPolyline *pPline3;

   if (acdbOpenObject(pEntity,id,AcDb::kForRead) != Acad::eOk)
   {
      acutPrintf(_T("\nError opening the entity"));
      return;
   }
   // try to cast object to a AcDb(2d/3d)Polyline object
   int polylineType = 0;
   if (pEntity->isKindOf(AcDbPolyline::desc()))
   {
      pPline = (AcDbPolyline*)pEntity;
      polylineType = 1;
   }
   else if (pEntity->isKindOf(AcDb3dPolyline::desc()))
   {
      pPline3 = (AcDb3dPolyline*)pEntity;
      polylineType = 3;
   }
```

C++

```
else
{
   acutPrintf(_T("\nNo type of AcDbPolyline object selected!"));
   pEntity->close();
   return;
}
int vertexNumber = 0;
double bulge = .0;
AcDbObjectId vertexObjId = AcDbObjectId::kNull;
AcDbObjectIterator *pVertIter = NULL;
AcDb3dPolylineVertex *pVertex3;
AcGePoint3d location;

switch (polylineType)
{
   case 1: vertexNumber = pPline->numVerts();
      for (int i=0; i<vertexNumber; i++)
      {
         if ((pPline->getPointAt(i,location) == Acad::eOk) &&
             (pPline->getBulgeAt(i, bulge) == Acad::eOk))
            acutPrintf(_T("\n1:Vertex #%d's location is:%0.3f, %0.3f,%0.3f"
                       "with bulge %0.3f"),i,location[X],location[Y],
                       location[Z],bulge);
      }
      pPline->close();
      break;
```

C++

```
      case 3: pVertIter= pPline3->vertexIterator();
         for (vertexNumber=0; !pVertIter->done();
                                          vertexNumber++,pVertIter->step())
         {
            vertexObjId = pVertIter->objectId();
            acdbOpenObject(pVertex3, vertexObjId, AcDb::kForRead);
            location = pVertex3->position();
            pVertex3->close();
            acutPrintf(_T("\n3:Vertex #%d's location is: %0.3f, %0.3f, %0.3f"),
                      vertexNumber,location[X],location[Y],location[Z]);
         }
         delete pVertIter;
   }
}
```



Properties statt Getter und Setter

Um Eigenschaften eines Objektes zu modifizieren nutzt C++ get…() und set…() Methoden. Die Wrapper-Klassen dagegen nutzen die Abstraktion von .NET und stellen entsprechende Properties zur Verfügung. Über diese kann dann auf die Eigenschaften sowohl lesend als auch schreibend (falls das Objekt vorher zum Schreiben geöffnet wurde) zugegriffen werden.
Typische Eigenschaften eines Entities sind z. B. Color, Layer, Linetype, LineWeight, LinetypeScale.
Die verschiedenen Objekttypen sind in .NET sehr ähnlich zu denen von C++ benannt. Die folgende Liste enthält alle für das Praktikum relevanten Klassen. Eine vollständige Auflistung der ObjectARX➔Managed-Zuordnung ist in der ObjectARX Dokumentation (**acad_mgd.chm**) zu finden.

ObjectARX Objekttyp	.NET Wrapperklasse
AcDbPolyline	Autodesk.AutoCAD.DatabaseServices.Polyline
AcDb3dPolyline	Autodesk.AutoCAD.DatabaseServices.Polyline3d
AcDb3dPolylineVertex	Autodesk.AutoCAD.DatabaseServices.PolylineVertex3d
AcDbArc	Autodesk.AutoCAD.DatabaseServices.Arc
AcDbCircle	Autodesk.AutoCAD.DatabaseServices.Circle
AcDbFace	Autodesk.AutoCAD.DatabaseServices.Face
AcDbLine	Autodesk.AutoCAD.DatabaseServices.Line
AcGePoint3d	Autodesk.AutoCAD.Geometry.Point3d

C#

Beispiel: Auslesen einer Polyline

```
static public void readPolyline(ObjectId id)
{
    Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
    Database db = Application.DocumentManager.MdiActiveDocument.Database;
    Autodesk.AutoCAD.DatabaseServices.TransactionManager tm = db.TransactionManager;
    Transaction transaction = tm.StartTransaction();
    try
    {
        Entity entity = (Entity)tm.GetObject(id, OpenMode.ForRead);
        Polyline polyline = null;
        Polyline3d polyline3d = null;
        int polylineType = 0;
        if (entity is Polyline)
        {
            polyline = (Polyline)entity;
            polylineType = 1;
        }
        else if (entity is Polyline3d)
        {
            polyline3d = (Polyline3d)entity;
            polylineType = 3;
        }
        else
        {
            editor.WriteMessage("\nNo type of Polyline object selected!");
            transaction.Dispose();
            return;
        }
```

C#

```
        switch (polylineType)
        {

           case 1:
                int vertexNumber = polyline.NumberOfVertices;
                for (int i = 0; i < vertexNumber; i++)
                {
                    Point3d location = polyline.GetPoint3dAt(i);
                    double bulge = polyline.GetBulgeAt(i);
                    editor.WriteMessage("\n1:Vertex #"+i+"'s location is:"+
                    location.X+", "+location.Y+", "+location.Z+", with bulge "+bulge);
                }
                break;
            case 3:
                foreach (ObjectId vertexId in polyline3d)
                {
                    PolylineVertex3d vertex3 =
                        (PolylineVertex3d)tm.GetObject(vertexId, OpenMode.ForRead);
                    editor.WriteMessage("\n3:Vertex location is:“ + vertex3.Position.X +
                        ", " + vertex3.Position.Y + ", " + vertex3.Position.Z + ",");
                }
                break;
        }

        transaction.Commit();
    }
    catch (Autodesk.AutoCAD.Runtime.Exception)
    {
        transaction.Dispose();
    }
}
```

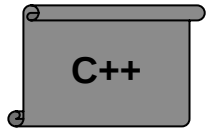
3.4.7. Selection Sets

Jetzt soll ein Weg beschrieben werden, der es erlaubt, Informationen aus der existierenden Zeichnung herauszulesen, also eine Interaktion zwischen dem Nutzer und unserem Programm aufzubauen.

Einführung

Der häufigste Weg, Informationen aus der Zeichnung zu bekommen, ist, ein oder mehrere Zeichnungselemente per Maus zu selektieren und diese dann auszulesen.
Ein anderer Weg ist, alle Zeichnungselemente, die einem Filter genügen, automatisch auszuwählen und dann auszulesen.

Ein SelectionSet ist eine Gruppe von Entities, die durch den Nutzer oder durch unsere Applikation ausgewählt wurden. AutoCAD stellt uns jeweils einen Namen, den **`ads_name`**, zur Verfügung, dessen ObjectId man mittels der Methode **`acdbGetObjectId()`** ermitteln kann.




```
Acad::ErrorStatus acdbGetObjectId (AcDbObjectId& objId,
                                   const ads_name objName);
```

Historisch nutzen einige der SelectionSet-Methoden den **`ads_name`** noch direkt, so dass dort eine Konvertierung in die Id entfallen kann.
Im Objekt **`ads_name`** können auch mehrere selektierte Objekte gespeichert sein, wenn der Nutzer oder unser Programm es so vorgesehen hat.

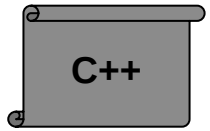
C++

Eine einfache Variante Elemente der Zeichnungsfläche zu selektieren ist die acedEntSel()-Methode. Neben dem Namen des ausgewählten Elements liefert sie auch den Punkt, also die Punktkoordinaten, des Pickpunktes.

```
extern "C" int acedEntSel(const ACHAR * str, ads_name entres,
                          ads_point ptres);
```

str	Optional ksnn ein String angegeben werden, der den Nutzer zu dieser Aktion auffordert. Wenn NULL angegeben wird, wird der Default-Prompt geschrieben
entres	Name des selektierten Entitys
ptres	Punkt, der zum selektieren gepickt wurde

Die Selection mehrerer Elemente wird mit der Methode **`acadSSGet()`** angefordert, wobei hier kein Prompting-Text als Parameter vorgesehen ist (muß also mit **`acutPrintf(_T("\n…"))`** zuvor ausgegeben werden).
Die Signatur der Methode ist folgende:




```
int acedSSGet (const char *str, const void *pt1, const void *pt2,
               const struct resbuf *entmask, ads_name ss);
```

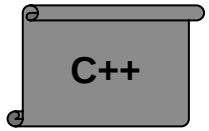
Als Parameter wird die Selektion-Option, zwei Punkte und eine Maske übergeben, zurück kommt als letzter Parameter ein Selektionset. (Nach Auswertung des Selektionsets muß man unbedingt den dynamischen Speicher mit **`acedSSFree()`** wieder freigeben.)
Als Selektion-Option wird AutoCAD die Art der Selektion mitgeteilt:

Selection Code	Description
NULL	Single-point selection (if pt1 is specified) or user selection (if pt1 is also NULL)
#	Nongeometric (all, last, previous)
:$	Prompts supplied
.	User pick
:?	Other callbacks
A	All
B	Box
C	Crossing
CP	Crossing Polygon
:D	Duplicates OK
:E	Everything in aperture
F	Fence (Zaun)

G	Groups
I	Implied
:K	Keyword callbacks
L	Last
M	Multiple
P	Previous
:S	Force single object selection only
W	Window
WP	Window Polygon
X	Extended search (search whole database)

Mit Hilfe dieser Wahlmöglichkeiten können wir nun verschiedene Modifikationen des Selektierens ansteuern. Das folgende Beispiel zeigt die häufigsten Anwendungen.

```
ads_point pt1, pt2;
ads_name ssname;
pt1[X] = pt1[Y] = pt1[Z] = 0.0;
pt2[X] = pt2[Y] = 5.0; pt2[Z] = 0.0;

// Get the current PICKFIRST or ask user for a selection
acedSSGet(NULL, NULL, NULL, NULL, ssname);

// Get the current PICKFIRST set
acedSSGet(_T("I"), NULL, NULL, NULL, ssname);

// Repeat the previous selection set
acedSSGet(_T("P"), NULL, NULL, NULL, ssname);
```

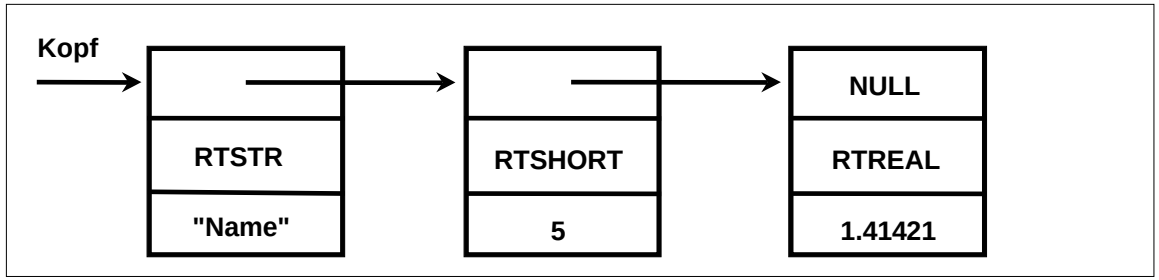


C++

```
// Selects the last created entity
acedSSGet(_T("L"), NULL, NULL, NULL, ssname);

// Selects entity passing through point (5,5)
acedSSGet(NULL, pt2, NULL, NULL, ssname);

// Selects entities inside the window from point (0,0) to (5,5)
acedSSGet(_T("W"), pt1, pt2, NULL, ssname);
```

Verwendung des Selektion-Filters

Filter kommen dann zum Einsatz, wenn nicht der Nutzer nicht aktiv werden soll oder wir Entities einer ganz bestimmten Art selektieren wollen. Dabei kann der Filter einfach (nur eine Entity-Art) oder auch zusammengesetzt sein. Ein Filter wird in einer spezielle Datenstruktur (eine verkettete Liste, deren Glieder Elemente unterschiedlichen Datentypes – mit union – beinhalten können), den Resultbuffer **`resbuf`** definiert.



C++

Um einen Filter zu nutzen, muß diese Datenstruktur aufgebaut und dann als Parameter an die **acedSSGet()** –Methode übergeben werden.
Das folgende Beispiel zeigt die Anwendung des Selektierens mit Filter:

```
struct resbuf eb1, eb2;
char sbuf1[10], sbuf2[10];
ads_name ssname1, ssname2;

// Entity name filter
eb1.restype = 0;
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL;
// Retrieve all circles
acedSSGet(_T("X"), NULL, NULL, &eb1, ssname1);

// Layer name filter
eb2.restype = 8;
strcpy(sbuf2, "0");
eb2.resval.rstring = sbuf2;
eb2.rbnext = NULL;
// Retrieve all entities on layer 0
acedSSGet(_T("X"), NULL, NULL, &eb2, ssname2);
```

C#

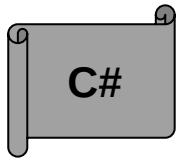
Unter .NET wird nicht mehr mit den historischen **ads_name** Strings gearbeitet. Statt dessen werden Objekte immer über ihre ObjectId angesprochen. Um ein Objekt auszuwählen und die Id eines selektieren Objektes zu erhalten, wird die Methode **GetEntity()** des Editors aufgerufen. Durch diesen Aufruf wird der Benutzer aufgefordert ein Objekt aus der Zeichnung zu markieren.

```
Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;
PromptEntityOptions options = new PromptEntityOptions("please select an entity");
PromptEntityResult result = editor.GetEntity(options);
ObjectId entityId = result.ObjectId;
Point3d pickPoint = result.PickedPoint;
```

Über **PromptEntityOptions** können auch Einschränkungen definiert werden um z. B. nur bestimmte Typen von Entities zuzulassen.

Um mehrere Objekte gleichzeitig auszuwählen steht in .NET anstelle von **acedSSGet()** die Methode **GetSelection()** des Editors zur Verfügung. Ohne weitere Parameter entspricht deren Aufruf der eines Aufrufes von **acedSSGet(NULL, NULL, NULL, NULL, ss)**. Als optionale Parameter können zusätzlich **PromptSelectionOptions** und **SelectionFilter** angegeben werden.

```
PromptSelectionOptions options = new PromptSelectionOptions();
options.AllowDuplicates = false;
PromptSelectionResult result = editor.GetSelection(options);
SelectionSet ss = result.Value;
```



Verwendung des Selektion-Filters in C#

Über Filter lässt sich die Auswahl von Objekten vollständig automatisieren und spezialisieren. So kann über sie z. B. festgelegt werden, welche Typen von Objekten wir ausgewählt haben möchten. Im Gegensatz zu C++ kommen hier keine ResultBuffer zum Einsatz. Statt dessen werden Filter mit Hilfe von **TypedValue** Paaren aufgebaut. Für jedes Paar wird ein Typ-Code und ein Wert benötigt. Um beispielsweise nach Objekttypen filtern zu können muss ein TypedValue Paar mit dem Code „Start“ und den gewünschten Objepttypen als Wert (z. B. „CIRCLE“) erzeugt werden. So können beliebig viele Kriterien für die Objektauswahl kombiniert werden. Will man nicht auf Benutzereingabe warten sondern einfach nur alle Objekte auf die der Filter passt auswählen, bietet sich außerdem die Editor-Methode **SelectAll()** an.

```
Editor editor = Application.DocumentManager.MdiActiveDocument.Editor;

// Konstruktion eines Filters der nur Kreise mit dem Radius 10.0 auswählt
TypedValue[] values = {
    new TypedValue((int)DxfCode.Start, "CIRCLE"), //DxfCode.Start steht für den Objekttyp
    new TypedValue((int)DxfCode.Real, 10.0),      // Radius
};
SelectionFilter filter = new SelectionFilter(values);
SelectionSet ss = editor.SelectAll(filter).Value;
```

Eine kleine Liste interessanter Typ-Codes:

DxfCode	Bedeutung
Start	Objekttyp (z. B. „CIRCLE“)
Real	Zahlenwert (z. B. für den Radius)
Operator	Logische Operatoren (z. B. AND, OR, ==)
LayerName	Id des Layers auf dem sich ein Objekt befindet

C++

Modifizieren von Entities aus einem Selection Set

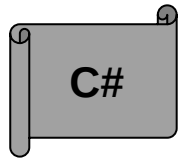
Um Entities aus einem Selection set zu modifizieren, müssen wir in einer Schleife durch alle enthaltenen Elemente iterieren, den Namen des Elementes herauslesen, von diesen **`ads_name`** eine **`ObjectId`** besorgen, das Element dann zum Schreiben öffnen, es modifizieren und dann wieder schließen. (Dasselbe könnten wir auch mit einer Transaktion, was bei komplexeren Modifikation deutlich besser ist.)

Im folgenden Beispielcode werden alle Kreis-Entities innerhalb einer Zeichnung herausgelesen und deren Farbe auf rot gesetzt:

```
// Construct the filter
struct resbuf eb1;
char sbuf1[10];
eb1.restype = 0; // Entity name
strcpy(sbuf1, "CIRCLE");
eb1.resval.rstring = sbuf1;
eb1.rbnext = NULL;

// Select All Circles
ads_name ss;
if (acedSSGet(_T("X"), NULL, NULL, &eb1, ss) != RTNORM)
{
   acutRelRb(&eb1);
   return;
}
```

C++

```
// Free the resbuf
acutRelRb(&eb1);

// Get the length (how many entities were selected)
long length = 0;
if ((acedSSLength( ss, &length ) != RTNORM) || (length == 0))
{
   acedSSFree( ss );
   return;
}

ads_name ent;
AcDbObjectId id = AcDbObjectId::kNull;

// Walk through the selection set and open each entity
for (long i = 0; i < length; i++)
{
   if (acedSSName(ss,i,ent) != RTNORM) continue;
   if (acdbGetObjectId(id,ent) != Acad::eOk) continue;
   AcDbEntity* pEnt = NULL;
   if (acdbOpenAcDbEntity(pEnt,id,AcDb::kForWrite) != Acad::eOk)
     continue;
   // Change color
   pEnt->setColorIndex(1);
   pEnt->close();
}

// Free selection
acedSSFree( ss );
```



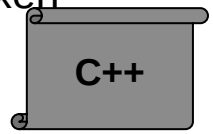
Modifizieren von Entities aus einem Selection Set mit C#

Um Entities aus einem SelectionSet zu modifizieren muss durch die Elemente einzeln iteriert werden. Für jedes Objekt enthält das SelectionSet die Id, anhand der das Objekt aus der Datenbank geladen und modifiziert werden kann. Die Iteration durch die Objekte gestaltet sich mit .NET recht unkompliziert, da die Wrapper-Klassen die in .NET Anwendungen üblichen Iterationsinterfaces implementiert haben.
Das folgende Beispiel liest aus einer Zeichnung alle Kreise heraus und färbt sie rot.

```
// construct the filter
TypedValue[] values = {new TypedValue((int)DxfCode.Start, "CIRCLE"),};
SelectionFilter filter = new SelectionFilter(values);

// select all circles
PromptSelectionResult result = editor.SelectAll(filter);
if (result.Status != PromptStatus.OK)
{
    transaction.Dispose();
    return;
}
SelectionSet selectionSet = result.Value;

// walk through the selection set and access entity in database
foreach (ObjectId id in selectionSet.GetObjectIds())
{
    Entity entity = (Entity)tm.GetObject(id, OpenMode.ForWrite, false);
    // Change color
    entity.ColorIndex = 1;
}
transaction.Commit();
```

3.4.8. Interaktion mit AutoCAD

Es gibt einen sehr einfachen Weg mit AutoCAD direkt zu agieren indem einfach die (sonst) vorzunehmende Eingabeaktionsfolge aus dem Programm an den Kommandointerpreter "gesendet" wird.
Die dafür zur Verfügung stehenden globalen Methoden sind sehr einfach und direkt. Sie sollten darum mit Vorsicht verwendet werden, insbesondere in Zusammenhang mit komplexen Operationen oder im Umfeld von programmiertem Event handling



Zwei Methoden stehen uns dafür zur Verfügung, **`acedCmd()`** und **`acedCommand()`**.
Die erste ruft das Kommando auf, in dem der übergebene Resultbuffer interpretiert wird.
Die zweite Methode enthält als (dynamische) Parameterliste der Werte mit den Kommando-schnipseln.

```
int acedCmd(const struct resbuf * rbp);
int acedCommand(int rtype, ... unnamed);
```

Um einen Resultbuffer zu bauen, verwendt man die Methode **`acutBuildList(),`** die eine verkettete Liste aufbaut. Dann muß eine Folge aus Typ-/Wertparametern übergeben werden, die jeweils einen Schnipsel des späteren Kommando darstellen. Als abschließenden Parameter muß eine 0 eingegeben werden (um die dynamische Parameterliste zu beenden).
In einem separaten Schritt kann nach Absenden des Kommandos quasi das "Return" durch den Aufruf von **`acedCommand(RTNONE)`** nachgesandt werden.
Nicht vergessen darf man das Freigeben des dynamischen Speichers für den Resultbuffer mit **`acutRelRb()`**.
Das folgende Beispiel zeigt die verschiedene Wege der Kommandogenerierung:

C++

acedCmd():

a) Verschieben des letztgenerierten Entities zum Punkt (0,0,0):

```
ads_point pt;
pt[0] = pt[1] = pt[2] = 0.0;
struct resbuf *myCom;
myCom = acutBuildList(RTSTR,_T("_MOVE"),RTSTR,_T("_LAST"),RTSTR,_T(""),
                      RTPOINT,pt,RTSTR,PAUSE,0);
acedCmd(myCom);
acedCommand(RTNONE);
acutRelRb(myCom);
```

b) Aufruf des "redraw"-Kommandos :

```
struct resbuf *cmdlist;
cmdlist = acutBuildList(RTSTR, _T("_REDRAW"), 0);
acedCmd(cmdlist);
acedCommand(RTNONE);
acutRelRb(cmdlist);
```

acedCommand():

a) Aufruf des ZOOM.Kommandos und warten auf Nutzer-Eingabe:

```
acedCommand(RTSTR, _T("_zoom"), RTSTR, PAUSE, RTSTR, PAUSE, 0);
```

C++

b) Kreieren eines Kreis- und Linien-Entities:

```
acedCommand(RTSTR, _T("_circle"), RTSTR, _T("10,10"),RTSTR, PAUSE,
            RTSTR, _T("_line"), RTSTR, _T("10,10"), RTSTR, _T("20,20"),
            RTSTR, _T(""), 0);
```

System Variable

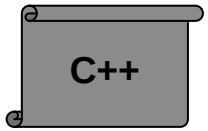
Eine fast unüberschaubare Menge von Systemvariablen konfiguriert und steuert die Arbeit mit AutoCAD. Auch aus einer Applikation lassen sich Systemvariable auslesen bzw. modifizieren. ObjectARX bietet dafür zwei Methoden **`acedGetVar()`** und **`acedSetVar(),`** um wieder mit einem Resultbuffer als Parameter auf die jeweiligen Werrte zuzugreifen.

```
int acedGetVar(const char * sym,struct resbuf * result);

int acedSetVar(const char * sym,const struct resbuf * val);
```

Der erste Parameter ist der Name der Systemvariablen, der zweite der Resultbuffer mit dem gelesenen bzw. zu schreibenden Wert.
Im folgenden Beispiel wird der Radius des **`FILLET`**-Kommandos (Ausrunden) mit einem neuen Wert belegt:

```
struct resbuf rb, rb1;
acedGetVar(_T("FILLETRAD"), &rb);
rb1.restype = RTREAL;
rb1.resval.rreal = 1.0;
acedSetVar(_T("FILLETRAD"), &rb1);
```

Es ist (wegen des union-Datentypes von resval) wichtig, dass der jeweils verlangte korrekte Datentyp in den Resultbuffer eingetragen wird, in dem obigen Beispiel also RTREAL.
Das folgende Beispiel setzt den Objekt-Schnappmodus vorübergehend auf 0 (Aus), sehr zu empfehlen bei programmgesteuerten Konstruktionen.




```
struct resbuf OldOsnap, NewOsnap;
// Get current OSNAP
acedGetVar(_T("OSMODE"), &OldOsnap);

// Chage OSNAP settings
NewOsnap.restype = RTSHORT;
NewOsnap.resval.rint = 0;
acedSetVar(_T("OSMODE"), &NewOsnap);

// Do something...

// Get back the old OSNAP
acedSetVar(_T("OSMODE"), &OldOsnap);
```

User Input Functionen

Eine weitere Gruppe vom globalen Methoden dient der direkten Kommunikation mit dem Nutzer über die AutoCAD-Kommandozeile.
Die folgende Tabelle gibt dazu eine Übersicht:

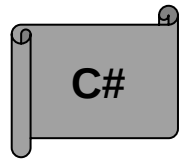
C++

acedGetInt	Fordert einen Integerwert an
acedGetReal	Fordert einen Realwert an
acedGetDist	Fordert einen Abstand an
acedGetAngle	Fordert einen Winkel an (gemessen an 0 Grad relativ zu der in **ANGBASE** definierten Basis)
acedGetOrient	Fordert einen Winkel an (gemessen an mathematisch 0 Grad)
acedGetPoint	Fordert einen Punkt an
acedGetCorner	Fordert die Ecke eines Rechtecks an
acedGetKword	Fordert ein keyword an
acedGetString	Fordert einen string an

Jede dieser Methoden liefert einen Returncode, der folgende Werte haben könnte:

RTNORM	Alles OK, richtiger Wert wurde eingegeben
RTERROR	Fehler, keine korrekte Eingabe
RTCAN	User hat mit **ESC** die Eingabe abgebrochen
RTNONE	User hat nur **ENTER** eingegeben
RTREJ	AutoCAD verwirft die Anforderung als unkorrekt
RTKWORD	User hat ein Keyword oder einen beliebigen Text eingegeben

Um AutoCAD einen Befehl über den Kommandointerpreter zu schicken, stehen unter C++ gleich zwei Funktionen zur Verfügung: **acedCmd()** und **acedCommand().** Unter .NET gibt es dafür nur eine einzige Anweisung names **SendStringToExecute()**. Als Parameter nimmt sie einen einfachen String entgegen, d.h. es ist keine Konstruktion eines Resultbuffers nötig. Der String muss wie folgt formatiert werden:



- Jedes Kommando wird mit einem \n abgeschlossen
- Am Anfang empfiehlt sich ein Escape-Zeichen (ASCII-Code 27), um eventuelle noch unabgeschlossene Commandos zu beenden

Hier einige Beispiele zur Kommandogenerierung:
Verschieben des letztgenerierten Entities um (10, 10, 0):

```
Application.DocumentManager.MdiActiveDocument.SendStringToExecute(
    "_MOVE\n_LAST\n\n10,10,0\n\n", true, false, true);
```

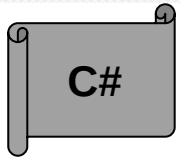
Aufruf des ZOOM-Kommandos und warten auf Nutzer-Eingabe:

```
Application.DocumentManager.MdiActiveDocument.SendStringToExecute(
    "_ZOOM\n", true, false, true);
```

Kreieren eines Kreis- und Linien-Entities:

```
char esc = (char)27;
Application.DocumentManager.MdiActiveDocument.SendStringToExecute(
    esc + "_CIRCLE\n10,10,0\n5\n_LINE\n0,0,0\n10,10,0\n\n", true, false, true);
```

Hinweis: der **SendStringToExecute()** Befehl ist nicht blockierend (asynchron), d.h. wenn der Befehl abgesendet wurde, heißt das noch nicht, dass er auch schon ausgeführt wurde. Die Ausführung kann von AutoCAD verzögert werden.



System Variablen mit C#

Auf Systemvariablen kann in .NET mit den Befehlen **Application.GetSystemVariable()** und **Application.SetSystemVariable()** zugegriffen werden. Ein Beispiel zum Auslesen und ändern kann beispielsweise wie folgt aussehen:

```
short oldMode = (short)Application.GetSystemVariable("OSMODE");

// disable object snap
Application.SetSystemVariable("OSMODE", 0);

// do something here...

// revert back to old snap mode
Application.SetSystemVariable("OSMODE", oldMode);
```

User Input Funktionen

Neben der bereits erwähnten Methode Editor.GetEntity() gibt es noch eine Reihe weiterer Methoden um mit dem Benutzer zu kommunizieren:

C++ Befehl	Managed Wrapper	C++ Befehl	Managed Wrapper
acedGetInt	Editor.GetInteger	acedGetPoint	Editor.GetPoint
acedGetReal	Editor.GetDouble	acedGetCorner	Editor.GetCorner
acedGetDist	Editor.GetDistance	acedGetKword	Editor.GetKeywords
acedGetAngle	Editor.GetAngle	acedGetString	Editor.GetString

4. Homogene Koordinaten und analytische Geometrie

4.1. Homogene Koordinaten

Homogene Koordinaten stellen eine Erweiterung der "normalen" Koordinaten um eine weitere Komponente dar.

Homogene Koordinaten wurden um die Mitte des 20. Jahrhunderts im Rahmen der projektiven Geometrie eingeführt und finden heute hauptsächlich in den Bereichen Computer Graphik und CAD ihre Anwendung.

Ein Punkt des n-dimensionalen euklidischen Raumes ist durch einen Vektor $[x_0, x_1, \ldots, x_{n-1}]^t$ gegeben. Homogenen Koordinaten (Verhältniskoordinaten) enthalten eine zusätzliche Komponente (die homogene Komponente w), die als Faktor zu jeder anderen Koordinatenkomponente zu betrachten ist.

$$\begin{bmatrix} x_0 \\ \ldots \\ x_{n-1} \\ w \end{bmatrix} == \begin{bmatrix} x_0/w \\ \ldots \\ x_{n-1}/w \\ 1 \end{bmatrix} \qquad (w \in \Re,\ w \neq 0)$$

Ein zweidimensionaler Vektor $\underline{P} = [x_p, y_p]^t$ kennzeichnet einen Punkt in der Zeichnungsebene (x-y-Ebene).

Durch Anhängen der w-Komponente erhält man einen Vektor in homogener Schreibweise.

$$\begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} == \begin{bmatrix} w*x_P \\ w*y_P \\ w \end{bmatrix} \quad (w \in \Re,\ w \neq 0)$$

entspricht dem Punkt $\underline{P} = [x_p, y_p]^t$

Die Zeichenebene kann man sich als w=1-Ebene im x-y-w-Raum vorstellen.

Punkt- und Richtungsvektoren

Für zwei Punktvektoren $\underline{P}$ und $\underline{Q}$ ist eine additive Verknüpfung $\underline{P}+\underline{Q}$ bzw. $\underline{P}-\underline{Q}$ nur dann definiert, wenn die w-Komponente von $\underline{P}$ und $\underline{Q}$ gleich sind.

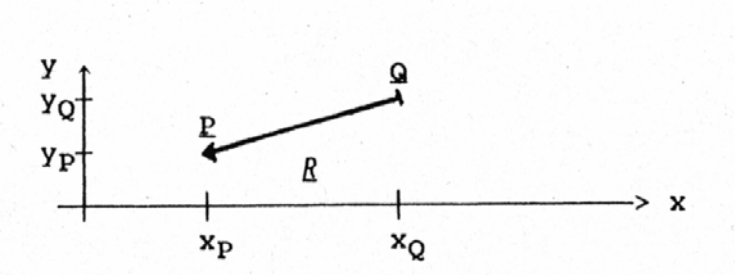
$$\underline{P} + \underline{Q} := \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} + \begin{bmatrix} x_Q \\ y_Q \\ 1 \end{bmatrix} = \begin{bmatrix} x_P + x_Q \\ y_P + y_Q \\ 2 \end{bmatrix} == \begin{bmatrix} (x_P+x_Q)/2 \\ (y_P+y_Q)/2 \\ 1 \end{bmatrix}$$

Mittelpunkt(Schwer-punkt) von $\underline{P}$ und $\underline{Q}$

$$\underline{P} - \underline{Q} := \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} - \begin{bmatrix} x_Q \\ y_Q \\ 1 \end{bmatrix} = \begin{bmatrix} x_P - x_Q \\ y_P - y_Q \\ 0 \end{bmatrix}$$

Bei der Subtraktion ergibt sich ein homogener Vektor, dessen w-Komponente den Wert Null hat. Solche Vektoren werden als Richtungsvektoren bezeichnet.

$\underline{R} := \underline{P} - \underline{Q}$



Die Länge eines Richtungsvektors wird nach Pythagoras aus der Wurzel der Summe der Quadrate ihrer Komponenten ermittelt.

Ein Richtungsvektor der Länge 1 wird als normierter Richtungsvektor bezeichnet.

Zwei Richtungsvektoren sind parallel, wenn sie linear anhängig sind, d.h. wenn es eine reelle Zahl r gibt, für die gilt:

$$\underline{S} = r * \underline{R} \quad (r \in \Re)$$

d.h.

$$x_R * y_S - y_R * x_S = 0$$

4.2. Punkte, Geraden, Strecken

Wie vorher dargestellt, verwenden wir für die Darstellung von Punkten stets den homogenen Punktvektor.

$$\underline{P} = \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} == \begin{bmatrix} w*x_P \\ w*y_P \\ w \end{bmatrix} \qquad (w \in \Re,\ w \neq 0)$$

Es wird vorausgesetzt, daß die w-Komponente eines Punktvektors =1 ist. Falls dies nicht erfüllt ist, muß der Vektor homogenisiert werden.

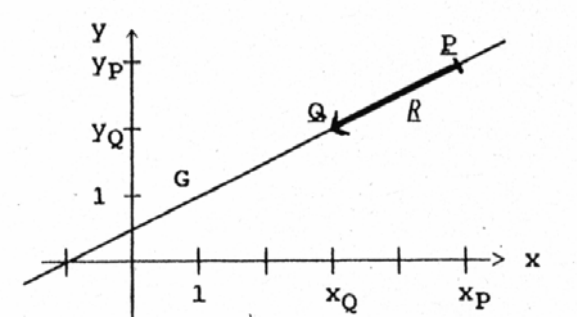
Geraden – Parametrische Geradenform

Eine Gerade G wird in der parametrischen Form durch einen Punkt- und einen Richtungsvektor definiert. Da ein Richtungsvektor durch zwei Punkte z.B. P und Q gebildet wird, trifft auch zu: eine Gerade wird durch zwei verschiedene Punkte bestimmt.

G: $\underline{X} = \underline{P} + r * \underline{R} = \underline{P} + r * (\underline{Q} - \underline{P}) \qquad (r \in \Re)$

oder komponentenweise:

$$G: \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} + r * \begin{bmatrix} x_R \\ y_R \\ 0 \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \\ 1 \end{bmatrix} + r * \begin{bmatrix} x_Q - x_P \\ y_Q - Y_P \\ 0 \end{bmatrix} \qquad (r \in \Re)$$



Diese Geradenform wird auch als generische Form bezeichnet, da für reelle Parameterwerte $\mathbf{r} \in [-\infty, \infty]$ alle Punkte X der Geraden generiert werden.

Für r=0 wird Punkt P und für r=1 Punkt Q angenommen.

Das führt auch direkt zur Definition von Strahl und Strecke:

Strahl

$\underline{X} = \underline{P} + r*\underline{R}$ für $\mathbf{r} \in \mathfrak{R}$ und $r \geqq 0$

Strecke

$\underline{X} = \underline{P} + r*\underline{R}$ für $\mathbf{r} \in \mathfrak{R}$ und $r \in [0,1]$

Geraden - Implizite Geradenform (Testform der Geraden)

Eine Gerade G ist in impliziter Form gegeben durch:

G: $a*x + b*y + c = 0$ $\quad a,b,c \in \mathfrak{R}$

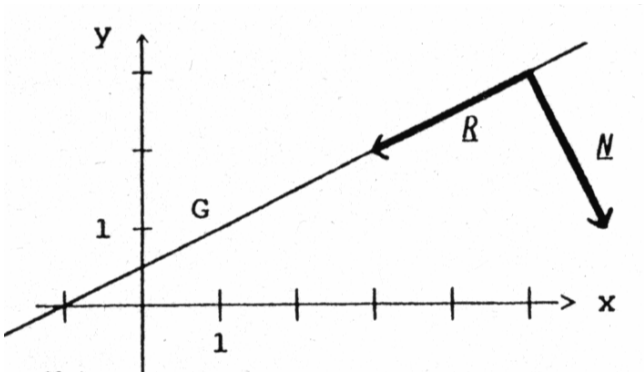
Diese Gleichung kann auch als Punktprodukt zweier homogener Vektoren geschrieben werden:

$$G: a*x + b*y + c = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underline{N} \cdot \underline{X} = 0$$

Der Vektor $\underline{N} := [a,b,c]^t$ wird als Normale bzw. Normalenvektor der Geraden G bezeichnet.

Die implizite Form wird deshalb als Testform bezeichnet, da sie verwendet wird, um z.B. zu testen, ob ein Punkt auf einer Geraden G liegt.

$$\underline{N} \cdot \underline{P} \begin{cases} = 0 \text{ genau dann, wenn } \underline{P} \text{ auf G liegt} \\ \neq 0 \text{ genau dann, wenn } \underline{P} \text{ nicht auf G liegt} \end{cases}$$



Normalen-Richtungsvektor und Richtungsvektor

Der Normalen-Richtungsvektor N der Geraden
$G: \underline{N} \cdot \underline{X} = 0$ mit dem Normalenvektor

$\underline{N} = [a,b,c]^t$ wird definiert als

$$\underline{N} := \begin{bmatrix} a \\ b \\ 0 \end{bmatrix}$$

Der Normalen-Richtungsvektor $\underline{N}$ steht senkrecht auf der Geraden.

Den Richtungsvektor $\underline{R}$ der Geraden in parametrischer Form erhält man durch das Kreuzprodukt zwischen dem Normalen-Richtungsvektor $\underline{N}$ und dem Ursprung $\underline{U}$:

$$\underline{R} = \underline{N} \times \underline{U} = \begin{bmatrix} a \\ b \\ 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} b \\ -a \\ 0 \end{bmatrix}$$

Damit hat eine Gerade auch in impliziter Form eine Orientierung: der Normalen-Richtungsvektor zeigt bezüglich der Richtung der Geraden "nach links", d.h. in die linke Halbebene. Die Multiplikation der impliziten Geradenform mit dem Faktor –1 liefert die Gerade mit der entgegengesetzten Orientierung.

Normierter Normalen-Richtungsvektor

Aus dem Normalen-Richtungsvektor $\underline{N}$ erhält man durch Multiplikation mit dem Kehrwert der Vektorlänge den normierten Normalen-Richtungsvektor $\underline{\underline{N}}$ mit der Länge 1:

$$\underline{\underline{N}} := \frac{\underline{N}}{|\underline{N}|} = \frac{1}{(a^2 + b^2)^{1/2}} * \begin{bmatrix} a \\ b \\ 0 \end{bmatrix} \quad \text{mit } |\underline{\underline{N}}| = 1$$

Schreibweisen im Skriptteil Geometrie

E, F, ...	Ebenen
$\underline{G}$, $\underline{H}$, ...	Geraden
$\underline{P}$, $\underline{Q}$, $\underline{X}$, ...	Punkte
$\underline{R}$, $\underline{S}$, ...	Richtungsvektoren
$\underline{\underline{P}}$, $\underline{\underline{R}}$, ...	normierte Richtungsvektoren bzw. Punktvektoren, deren Richtungsvektor normiert ist
$\underline{P}^j$, $\underline{R}^j$	"verkürzte" Vektoren, bei denen aus den ursprünglichen Vektoren P bzw. R die j-te Komponente gestrichen ist
A, B, ...	Matrizen
$[., ...]^t$	transponierter Zeilenvektor (platzsparend für Spaltenvektor)
$\in$	Element von
$\mathfrak{R}$	Menge der reellen Zahlen
(...)	Menge
[.., ..]	abgeschlossenes Intervall
$(...)^{1/2}$	Wurzel
|.|	Betrag (Länge) eines Vektors bzw. Determinate einer Matrix
*	normales Produkt
·	Punktprodukt (Skalarprodukt)
x	Kreuzprodukt zweier dreikomponentiger Vektoren (Vektorprodukt)
X (..,..,..)	erweitertes Kreuzprodukt von drei vierkomponentigen Vektoren
:=	Definition der linken Seite durch die rechte Seite

Zur Information: Punkt-(Skalar-) und Kreuzprodukt (Vektorprodukt)

Das **Punktprodukt** zweier Vektoren liefert eine reelle Zahl.

$$\underline{A} \cdot \underline{B} := \underline{A}^t * \underline{B} = a_0*b_0 + a_1*b_1 + \ldots a_{n-1}*b_{n-1}$$

weiter gilt:

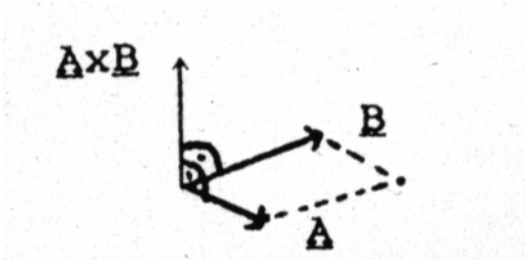
$$\underline{A} \cdot \underline{B} := |\underline{A}| * |\underline{B}| * \cos(\alpha)$$

(α ist der eingeschlossene, ungerichtete Winkel)

Das **Kreuzprodukt** zweier Vektoren A x B ist ein Vektor.
Die komponentenweise Definition des Kreuzproduktes zweier Vektoren A x B mit drei Komponenten ergibt:

$$\underline{A} \times \underline{B} := \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} \times \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} a_1b_2 - a_2b_1 \\ a_2b_0 - a_0b_2 \\ a_0b_1 - a_1b_0 \end{bmatrix}$$

Der Vektor A x B steht senkrecht auf A und auf B .



Eigenschaften:

$\underline{A} \times \underline{B} = -\underline{B} \times \underline{A}$ (Anti-Kommutativität)

$\underline{A} \times \underline{B} = 0$ (mit $\underline{A}, \underline{B} \neq 0$) gilt genau dann, wenn A und B linear abhängig sind.

4.3. Konstruktionen mit Geraden (Schnittpunkt, Parallel-Geraden, Abstand, Lot)

Der Normalenvektor einer Geraden errechnet sich aus dem Kreuzprodukt der Punktvektoren ihrer Definitionspunkte der parametrischen Geradenform:

$$\underline{N_1} := \underline{P_1} \times \underline{Q_1} = \begin{bmatrix} x_{P1} \\ y_{P1} \\ 1 \end{bmatrix} \times \begin{bmatrix} x_{Q1} \\ y_{Q1} \\ 1 \end{bmatrix}$$

analog berechnet sich $\underline{N_2}$

Der Schnittpunkt zweier Geraden (nicht zweier Strecken!) läßt sich aus dem Kreuzprodukt ihrer Normalvektoren bestimmen:

$$\underline{S} := \underline{N_1} \times \underline{N_2} = \begin{bmatrix} x_S \\ y_S \\ w \end{bmatrix}$$

Die homogene Komponente w ist hierbei nicht =1.

Ist sie =0, so sind die Richtungsvektoren parallel.

$$S_H = \begin{bmatrix} x_S / w \\ y_S / w \\ 1 \end{bmatrix}$$

Bestimmung des Abstandes eines Punktes von einer Geraden

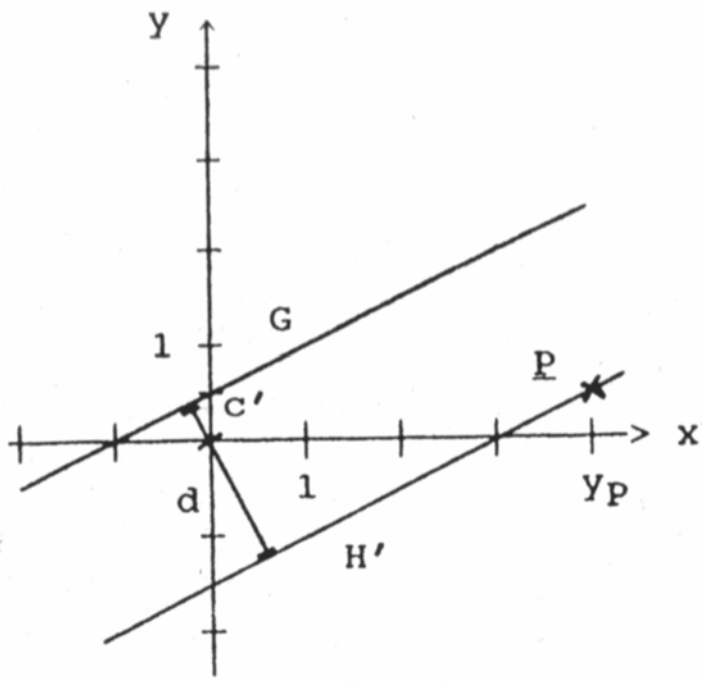
Der (vorzeichenbehaftete) Abstand **d** eines Punktes (mit w=1) zu einer in HNF gegebenen Geraden **G: N · X = a'*x + b'*y + c' = 0** ergibt sich aus:

$$d = \underline{N} \cdot \underline{P}$$

Wenn die Gerade **G: N · X** nicht in der HNF gegeben ist, so ergibt sich der betragsmäßige Abstand **|d|** eines Punktes **P** von der Geraden aus:

$$d = |\underline{N} \cdot \underline{P}| \;/\; |\underline{N}|$$

Hinweis:
Um den Abstand **d** zwischen der HNF-Geraden **G** und dem Punkt **P** zu ermitteln, bestimmt man die Parallele **H** zu **G**, die durch **P** verläuft.
Damit läßt sich der Abstand von **H** zum Ursprung leicht ermitteln.
Dazu addiert werden muß dann noch der Abstand von **G** zum Ursprung.



Bestimmung des Lotfußpunktes zu einem Punkt und einer Geraden

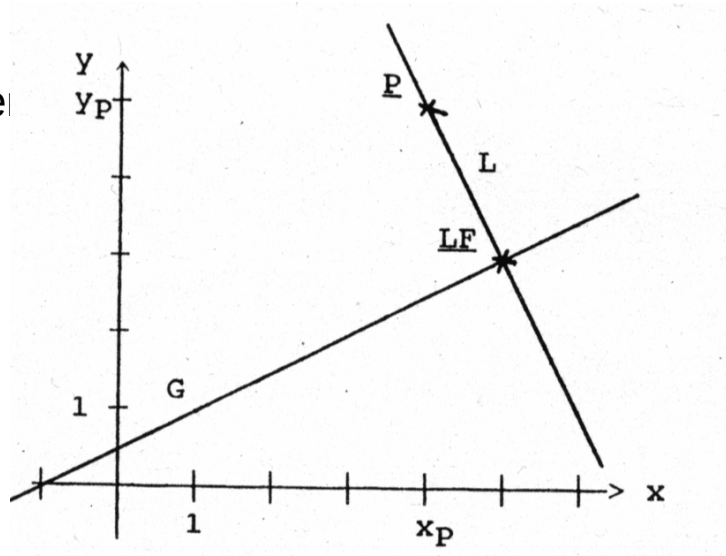
Sei **G: $\underline{N} \cdot \underline{X} = 0$** eine implizit gegebene Gerade und P ein Punkt, der nicht auf G liegt. Um seinen Lotfußpunkt LF zu bestimmen, in dem die Lotgerade L durch P die Gerade G schneidet, stellt man zunächst die parametrische Gleichung der Lotgeraden L auf:

L: $\underline{X} = \underline{P} + r*\underline{N}$ $\quad (r \in \Re)$

G und L schneiden sich für den Parameterwe

$$r_{LF} = - \frac{\underline{N} \cdot \underline{P}}{\underline{N} \cdot \underline{N}}$$

Das Einsetzen dieses Wertes in die parametrische Form der Lotgeraden liefert den Lotfußpunkt **LF**.



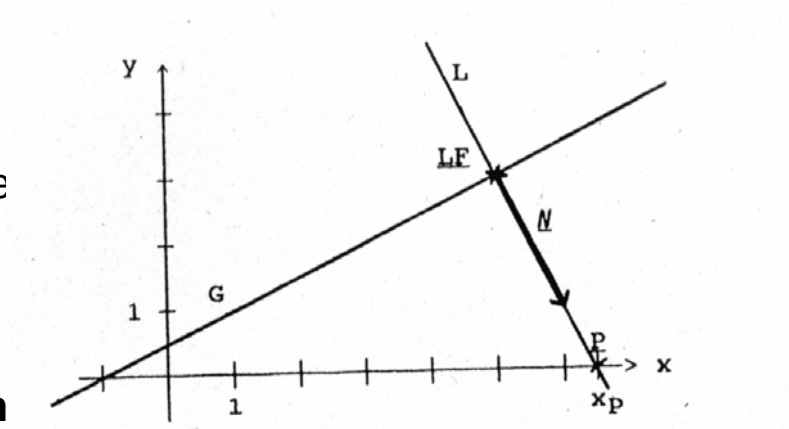
Bestimmung der relativen Lage eines Punktes bezüglich einer Geraden

Ist die Gerade G implizit gegeben, so läßt sich folgende Fallunterscheidung treffen:

G: $\underline{N} \cdot \underline{X} = 0$

$$(\underline{N} \cdot \underline{P}) \begin{cases} >0 & \text{falls } \underline{P} \text{ links der Geraden liegt} \\ =0 & \text{falls } \underline{P} \text{ auf der Geraden liegt} \\ <0 & \text{falls } \underline{P} \text{ rechts der Geraden liegt} \end{cases}$$

$\underline{N}$ läßt sich bei einer parametrisch gegebenen Gerade leicht aus $\underline{N} = \underline{A} \times \underline{E}$ bestimmen, wobei $\underline{A}$ und $\underline{E}$ die Punktvektoren der Endpunkte der Strecke sind.



Bestimmung des Winkels zwischen zwei Geraden

Seien G und H zwei sich schneidende Geraden in parametrischer bzw. impliziter Form:

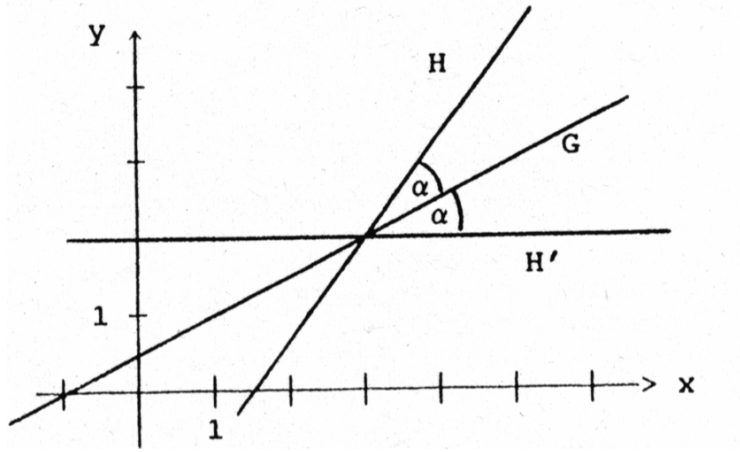
$$G: \underline{X} = \underline{P_1} + r_1 * \underline{R_1} \quad (r_1 \in \Re) \quad \text{bzw. } \underline{N_1} \cdot \underline{X} = 0$$

$$H: \underline{X} = \underline{P_1} + r_2 * \underline{R_2} \quad (r_2 \in \Re) \quad \text{bzw. } \underline{N_2} \cdot \underline{X} = 0$$

Der ungerichtete Winkel α zwischen den beiden Geraden G und H ergibt sich aus:

$$\alpha = \cos^{-1}\left(\frac{\underline{R_1} \cdot \underline{R_2}}{|\underline{R_1}| * |\underline{R_2}|} \right)$$

$$= \cos^{-1}\left(\frac{\underline{N_1} \cdot \underline{N_2}}{|\underline{N_1}| * |\underline{N_2}|} \right)$$



5. Analytische Geometrie im Raum mit Homogenen Koordinaten

5.1. Ebene und Punkte

Zunächst führen wir homogene Koordinaten für den Raum ein:

$$\underline{P} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} == \begin{bmatrix} w*x_p \\ w*y_p \\ w*z_p \\ w \end{bmatrix} \qquad w \neq 0$$

Homogene Richtungsvektoren werden wieder als Differenz zweier homogenisierter Punktvektoren definiert:

$$\underline{R} := \underline{Q} - \underline{P} = \begin{bmatrix} x_Q \\ y_Q \\ z_Q \\ 1 \end{bmatrix} - \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_Q - x_p \\ y_Q - y_p \\ z_Q - z_P \\ 1 - 1 \end{bmatrix} = \begin{bmatrix} x_R \\ Y_R \\ z_R \\ 0 \end{bmatrix}$$

Erweitertes Kreuzprodukt

Seien A, B und C drei beliebige Vektoren mit je vier Komponenten, so wird das erweiterte Kreuzprodukt als ein Vektor wie folgt definiert:

$$X(\underline{A},\underline{B},\underline{C}) := \begin{bmatrix} (-1)^0 * DET[\underline{A}^0\ B^0\ \underline{C}^0] \\ (-1)^1 * DET[\underline{A}^1\ \underline{B}^1\ \underline{C}^1] \\ (-1)^2 * DET[\underline{A}^2\ \underline{B}^2\ \underline{C}^2] \\ (-1)^3 * DET[\underline{A}^3\ B^3\ \underline{C}^3] \end{bmatrix}$$

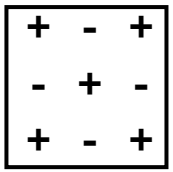
Dabei ist $\underline{A}^i$ **(i=0,1,2,3)** der auf drei Komponenten verkürzte Vektor, der aus $\underline{A}$ ensteht, wenn man die i-te Komponente streicht (entsprechendes gilt für $\underline{B}^i$ bzw. $\underline{C}^i$).

$[\underline{A}^i\ \underline{B}^i\ \underline{C}^i]$ steht für die 3*3-Matrix, die aus den drei Spaltenvektoren $\underline{A}^i$, $\underline{B}^i$ und $\underline{C}^i$ gebildet wird.

Die Determinante einer 3*3-Matrix **A** ist wie folgt definiert:

$$DET(A) = \begin{vmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{vmatrix} = a_{00}*a_{11}*a_{22} + a_{01}*a_{12}*a_{20} + a_{02}*a_{10}*a_{21} - a_{02}*a_{11}*a_{20} - a_{00}*a_{12}*a_{21} - a_{01}*a_{10}*a_{22}$$

Begründung: Nach Laplace wird eine 3*3-Determinate so berechnet, daß eine beliebige Spalte oder Zeile ausgewählt wird und die dortigen Elemente mit der 2*2-Restdeterminaten (durch Streichen der Zeile und Spalte des betrachteten Elementes) multipliziert werden (Hauptdiagonale-Nebendiagonale). Das jeweilige Vorzeichen der Produkte ergeben sich aus der Position des Elementes nach der Schachbrett-Vorzeichenmatrix.



Eigenschaften des erweiterten Kreuzproduktes

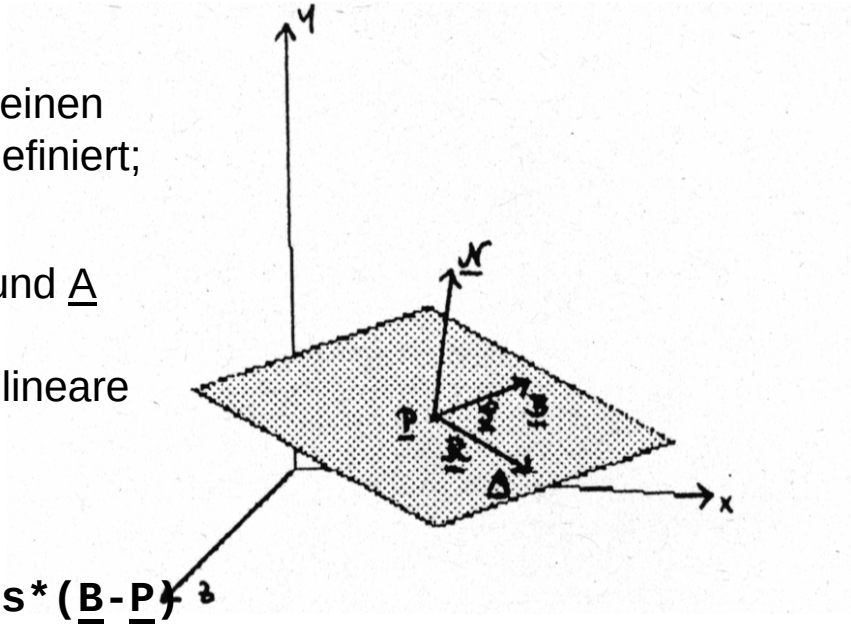
$X(\underline{A}, \underline{B}, \underline{C}) \cdot \underline{A} = 0$ wegen Parallelität, ebenso auch mit B und C

$X(\underline{A}, \underline{B}, \underline{C}) = 0$ genau dann, wenn A, B und C linear anhängig sind

Ebenen im Raum

Eine Ebene E wird in **parametrischer Form** durch einen Punkt- und zwei nichtparallele Richtungsvektoren definiert; die Richtungsvektoren "spannen die Ebene E auf".

Da ein Richtungsvektor durch zwei Punkte (hier P und A bzw. P und B) gebildet wird, ist diese Aussage gleichbedeutend zu: eine Ebene wird durch drei kollineare Punkte bestimmt.



$$E: \underline{X} = \underline{P} + r*\underline{R} + s*\underline{S} = \underline{P} + r*(\underline{A}-\underline{P}) + s*(\underline{B}-\underline{P})$$

$$(r, s \in \Re)$$

$$E: \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} + r* \begin{bmatrix} x_R \\ y_R \\ z_R \\ 0 \end{bmatrix} + s* \begin{bmatrix} x_S \\ y_S \\ z_S \\ 0 \end{bmatrix}$$

$$E: \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} + r* \begin{bmatrix} x_A - x_P \\ y_A - y_P \\ z_A - z_P \\ 0 \end{bmatrix} + s* \begin{bmatrix} x_B - x_P \\ y_B - y_P \\ z_B - z_P \\ 0 \end{bmatrix}$$

Diese Form der Ebenegleichung wird auch generische Form bezeichnet, da für reelle Parameterwerte r, s $\in [-\infty, \infty]$ alle Punkte X der Ebene generiert werden.
(Für r=0 und s=0 wird der Punkt P angenommen, für r=1 und s=0 der Punkt A, für r=0 und s=1 der Punkt B)

Eine Ebene E ist in **impliziter Form** gegeben durch:

E: a*x + b*y + c*z + d = 0 a,b,c,d $\in \Re$

Unter Verwendung von Vektoren kann die Gleichung auch so geschrieben werden:

$$E: \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \underline{N} \cdot \underline{X} = 0$$

Der Vektor $\mathbf{N = [a,b,c,d]^t}$ wird als Normale bzw. Normalenvektor der Ebene E bezeichnet. Eine Ebene E ist also die Menge aller Punkte $\underline{X}$ des Raumes, für die gilt:
$\underline{N} \cdot \underline{X} = 0$

Die implizite Form wird auch Testform genannt, da sie häufig verwendet wird, um z.B. zu testen, ob ein Punkt P in der Ebene E liegt.

$$\underline{N} \cdot \underline{P} \begin{cases} = 0 & \text{genau dann, wenn } \underline{P} \text{ in E liegt} \\ \neq 0 & \text{genau dann, wenn } \underline{P} \text{ nicht in E liegt} \end{cases}$$

(Bei $\underline{N} \cdot \underline{P} \neq 0$ ist der erhaltene Wert proportional zum Abstand des Punktes $\underline{P}$ zur Ebene E. Das Vorzeichen von $\underline{N} \cdot \underline{P}$ entscheidet, auf welcher Seite der Ebene E der Punkt P liegt.)

Der Normalen-Richtungsvektor $\underline{N}$ der Ebene **E:** $\underline{N} \cdot \underline{X} = a*x + b*y + c*z * d = 0$ hat die Form:

$$\underline{N} = \begin{bmatrix} a \\ b \\ c \\ 0 \end{bmatrix}$$

Den normierten Normalen-Richtungsvektor einer Ebene erhält man durch Division mit seiner Länge

$$\underline{\underline{N}} = \frac{\underline{N}}{|\underline{N}|}$$

Der Normalen-Richtungsvektor steht senkrecht auf der Ebene. Dadurch erhält die Ebene eine Orientierung, indem sie den Raum in einen Bereich (Halbraum) "oberhalb" und einen Bereich "unterhalb" der Ebene aufteilt. Der Normalen-Richtungsvektor zeigt in den oberen Halbraum.

Ebene in Hesse'scher Normalform (HNF)

Unter Verwendung der signum-Funktion (Vorzeichen der Zahl multipliziert mit 1) erhält man durch Multiplikation der impliziten Form der Ebene

$$E: a*x + b*y + c*z + d = \underline{N} \cdot \underline{X} = 0$$

mit der reellen Zahl $sgn(c)/|\underline{N}| = sgn(c)/(a^2 + b^2 + c^2)^{1/2}$ die Hesse'sche Normalform:

$$E_{HNF}: \frac{sgn(c)}{(a^2 + b^2 + c^2)^{1/2}} * (a*x + b*y + c*z + d) = \frac{\underline{N}}{|\underline{N}|} \cdot \underline{X} = 0$$

Ebenen in Hesse'scher Normalform werden im Folgenden auch wie folgt bezeichnet:

$$E_{HNF}: a' *x + b' *y + c' *z + d' = \underline{N} \cdot \underline{X} = 0$$

Dabei ist das Vorzeichen von c' positiv (sonst das von b' bzw. das von a').

Die HNF liefert mit **dist = -d'** direkt den vorzeichenbehafteten Abstand der Ebene zum Ursprung und kann gut zur Bestimmung des Abstandes eines Punktes von einer Ebene verwendet werden.

Umwandlung der parametrischen in die implizite Form der Ebene

Die parametrische Form der Ebene lautet:

$$E: \underline{X} = \underline{P} + r*\underline{R} + s*\underline{S} = \begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix} + r* \begin{bmatrix} x_R \\ y_R \\ z_R \\ 0 \end{bmatrix} + s* \begin{bmatrix} x_S \\ y_S \\ z_S \\ 0 \end{bmatrix} \qquad r,s \in \Re$$

Die implizite Form der Ebene ergibt sich zu:

E: $\underline{N} \cdot \underline{X} = 0$ mit

$$\underline{N} = X(\underline{P},\underline{R},\underline{S}) = \left(\begin{bmatrix} x_P \\ y_P \\ z_P \\ 1 \end{bmatrix}, \begin{bmatrix} x_R \\ y_R \\ z_R \\ 0 \end{bmatrix}, \begin{bmatrix} x_S \\ y_S \\ z_S \\ 0 \end{bmatrix} \right)$$

$$= \begin{bmatrix} + (y_R z_S - z_R y_S) \\ - (x_R z_S - z_R x_S) \\ + (x_R y_S - y_R x_S) \\ - (x_P(y_R z_S - z_R y_S) - y_P(x_R z_S - z_R x_S) + z_P(x_R y_S - y_R x_S)) \end{bmatrix}$$

Dabei wurde die 3*3-Determinaten zur Berechnung der untersten Komponente (d) nach der ersten Spalte entwickelt.

Abstand eines Punktes von einer Ebene

Der (vorzeichenbehaftete) Abstand eines Punktes Q zu einer in HNF gegebenen Ebene E: **a' *x + b' *y + c' *z + d' = N · X = 0** berechnet sich

dist = N · Q

Wenn die Ebene **E: N · X = 0** nicht die HNF hat, so kann der betragsmäßige Abstand **|dist|** eines Punktes Q von der Ebene E angegeben werden mit

|dist| = N · Q / |N|

Häufig soll nur festgestellt werden, ob zwei Punkte P und Q auf derselben Seite einer implizit gegebenen Ebene **E: N · X = 0** liegen. Dies ist genau dann der Fall, wenn **N · P** und **N · Q** dasselbe Vorzeichen haben.

Das läßt sich zu der folgenden Regel zusammenfassen:

(N · P) * (N · Q)

- **> 0** Punkte liegen auf derselben Seite
- **= 0** mindestens ein Punkt liegt in E
- **< 0** Punkte liegen auf verschiedenen Seiten

Eine Raumgerade und eine Ebene schneiden sich in einem Punkt

Der Schnittpunkt S zwischen einer parametrisch gegebenen Geraden
$G: \underline{X} = \underline{P} + r*\underline{R_G}$ $(r \in \Re)$

und einer implizit gegebene Ebene
$E: \underline{N_E} \cdot \underline{X} = 0$, die nicht G enthält, ergibt sich, wenn man den Parameterwert

$$r_S = - \frac{\underline{N_E} \cdot \underline{P}}{\underline{N_E} \cdot \underline{R_G}}$$

in die parametrische Form von G einsetzt.

Bemerkung:
Falls die Gerade G in der Ebene E liegt oder parallel zu dieser verläuft, so hat der Nenner den Wert 0 und r_S ist nicht definiert.

Drei Punkte bestimmen eine Ebene

Die Ebene E durch drei verschieden und nicht kollinear angeordnete Punkte A, B, C hat den Normalenvektor $\mathbf{N = X(A, B, C).}$

Die Ebene hat also die Form:

$$\mathbf{E: N \cdot X = X(A,B,C) \cdot X = 0}$$

Zum Beweis können die Punkte A, B, C für X in die Gleichung eingesetzt werden. Alle drei Punkte liegen liegen in der Ebene.

Die Punkte sind teilweise kollinear, wenn X (A, B, C) = 0

Lotfußpunkt in einer Ebene

Sei **E:** $\underline{N} \cdot \underline{X}$ eine implizit gegebene Ebene und $\underline{Q}$ ein Punkt, der nicht in der Ebene liegt.

Um den Lotfußpunkt $\underline{LF}$ bezüglich des Punktes $\underline{Q}$ in der Ebene **E** zu bestimmen, konstruiert man sich die Lotgerade **L**, die durch $\underline{Q}$ verläuft und die Ebene **E** senkrecht schneidet.

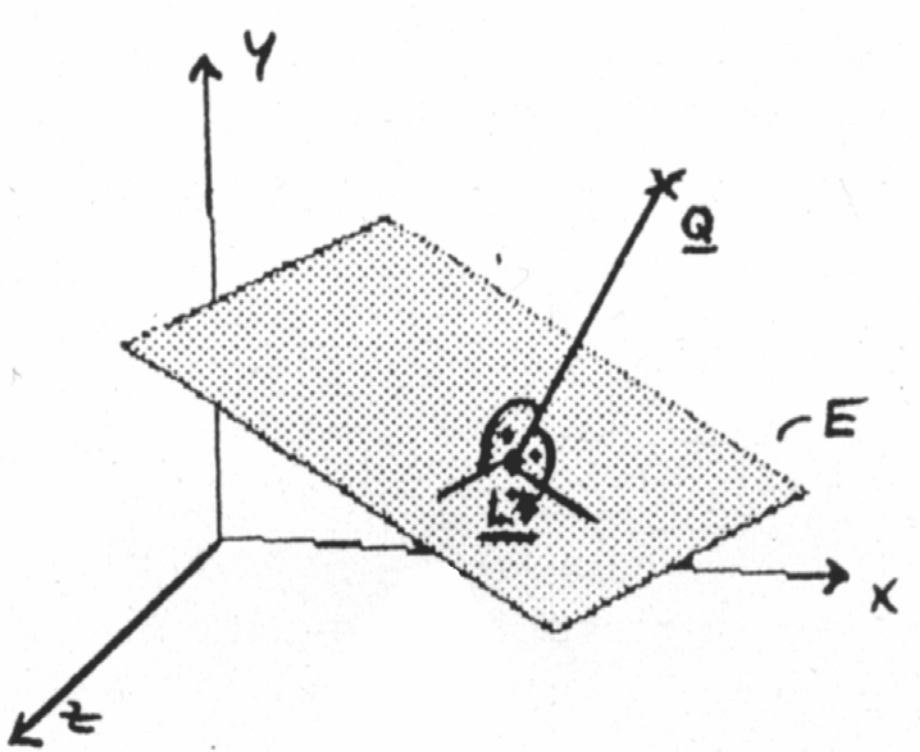
Da man den Normalen-Richtungsvektor N der Ebene E als Richtungsvektor der Lotgeraden L verwenden kann, ergibt sich diese in parametrischer Form aus:

$$L: \underline{X} = \underline{Q} + r*\underline{N} \quad (r \in \Re)$$

Der Schnittpunkt LF des Lotes L mit der der Ebene E ergibt sich (nach Seite 65), wenn man den Parameterwert

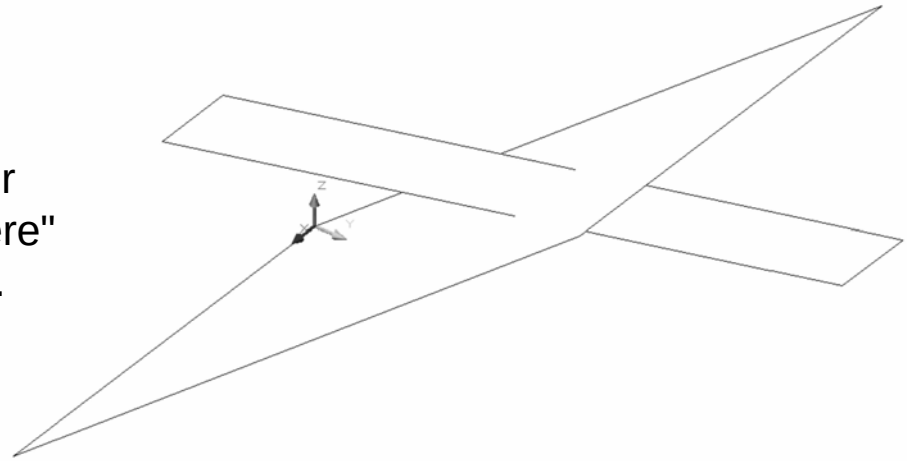
$$r_{LF} = - \frac{\underline{N} \cdot \underline{Q}}{\underline{N} \cdot \underline{N}}$$

in die o.g. parametrische Form der Lotgeraden einsetzt.



Verschneiden von zwei Ebenen

Eine häufig benötigte Konstruktion ist das Verschneiden einer (inneren) Ebene an einer anderen (äußeren) Ebene, so daß der "hintere" Teil der inneren Ebene weggeschnitten wird.



Hierzu kann man wie folgt vorgehen:

a) Bestimmen der jeweils vier Eckpunkte der Begrenzungsfläche der Ebene (Facette)

b) Bestimmen von zwei Vektoren in der äußeren Ebene (z.B. P_1 -> P_2 und P_2 -> P_3)

c) Bestimmung des Normalenvektors der äußeren Ebene, seiner Länge und der Größe d der impliziten Form.

d) Normalisieren des Normalenvektors

e) Berechnung des Abstandes (eigentlich nur der Seite) der Eckpunkte der Begrenzungsfläche der inneren Ebene zur äußeren Ebene

f) Dabei die Vektoren bestimmen, die jeweils zwei Eckpunkte auf unterschiedlichen Seiten der Ebene verbinden

g) Schnittpunkte dieser Raumgeraden mit der äußeren Ebene bestimmen

h) Entsprechende Eckpunkte der inneren Ebene durch die Schnittpunkte substituieren

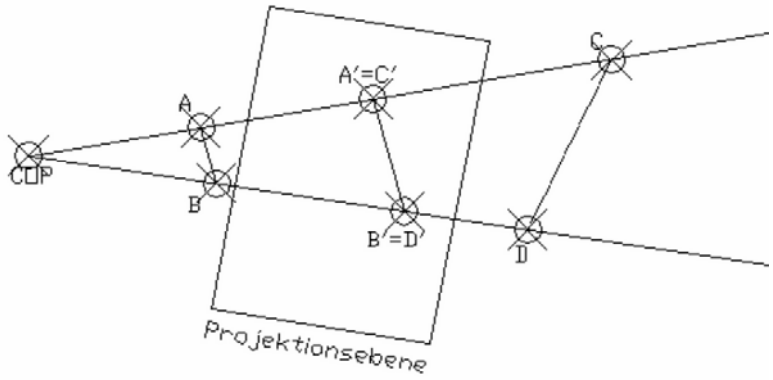
6. Projektionen und deren Implementierung

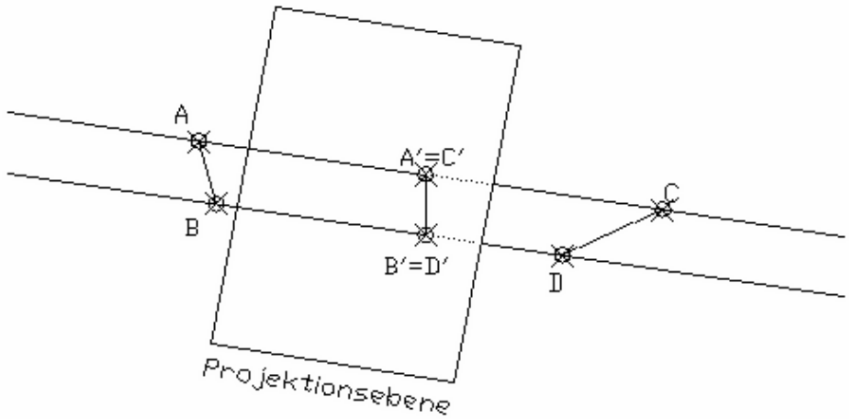
Projektionen sind spezielle räumliche Transformationen, die von einem n-dimensionalen Raum in einen Raum niedriger Dimension transformieren.

Speziell die Projektion dreidimensionaler Objekte auf eine zweidimensionale Darstellungsfläche ist eine der häufigsten Anwendungen dieses Algorithmus'.

Bei einer ebenen Projektion ist die Darstellungsfläche eine Ebene.

Die Projektion wird definiert durch die Projektionsebene und das Projektionszentrum (COP). Ist das Projektionszentrum, von dem die Projektionsstrahlen ausgehen, ein endlicher Punkt, so erhält man die perspektivische Projektion, liegt das Projektionszentrum im Unendlichen wird die Projektion als Parallele Projektion bezeichnet.

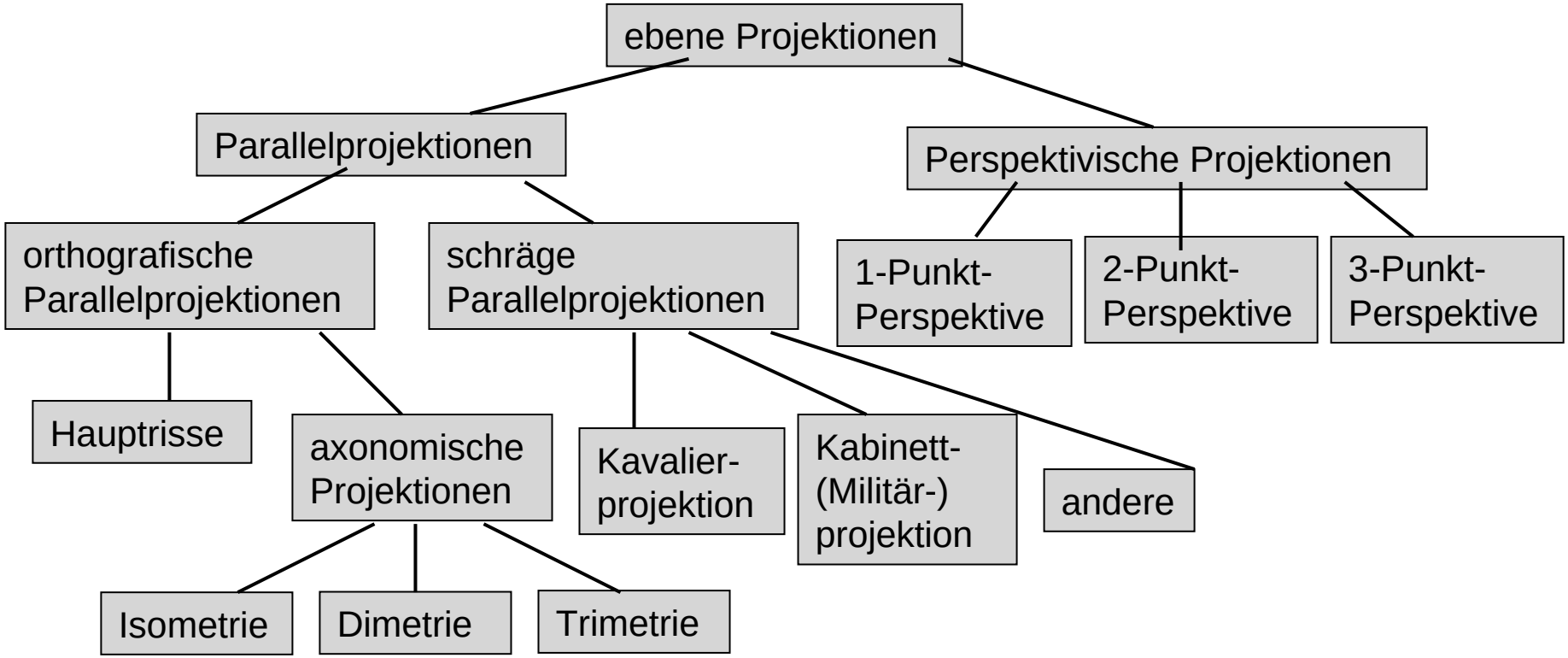




Bei beiden Projektionsarten kann ein abzubildendes Objekt bezüglich des Projektionszentrums vor oder hinter der Projektionsebene liegen.

6.1. Parallele Projektionen

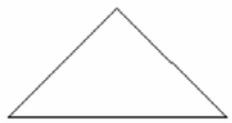
Bei der Parallelprojektion unterscheidet man zwischen orthografischer und schräger Parallelprojektion.
Bei der orthografischen Parallelprojektion stehen die Projektionsstrahlen senkrecht auf der Projektionsebene; bei der schrägen Parallelprojektion treffen die Projektionsstrahlen unter einem Winkel ϕ mit $0° < \phi < 90°$ die Projektionsebene.



6.1.1. Orthografische Parallelprojektionen

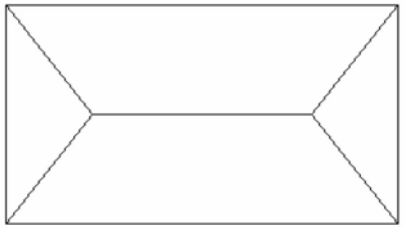
Im Konstruktionsbereich werden häufig orthografische Parallelprojektionen verwendet, bei denen die Projektionsstrahlen parallel zu einer der Koordinatenachsen verlaufen. Die dabei entstehenden Abbildungen werden als **Hauptrisse** bezeichnet.

von rechts



von links

von vorn



von oben

Betrachtet man die Projektion auf eine Ebene $\mathbf{E:\ z-z_0 = 0}$, also auf eine Ebene, die parallel zur x-y-Koordinatenebene verläuft, kann man folgende Transformationsmatrix definieren:

$$\mathbf{T_{op}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ein Punkt $\underline{\mathbf{P}} = [x_p, y_p, z_p, 1]^t$ wird durch die Projektion $\mathbf{T_{op}}$ auf den Punkt $\underline{\mathbf{P'}} = \mathbf{T_{op}} * \underline{\mathbf{P}} = [x_p, y_p, z_0, 1]^t$ abgebildet. Für $z_0=0$ ist die Projektionsebene die x-y-Ebene und $\underline{\mathbf{P'}}$ ensteht aus $\underline{\mathbf{P}}$ durch "Weglassen" der z-Komponente.

Die Projektionen auf die anderen Koordinatenbenen ergeben sich entsprechend.

Wenn man die Projektionsrichtung ändert und dabei gleichzeitig die Projektionsebene "mitdreht", so daß die Projektionstrahlen weiterhin senkrecht auf die Projektionsebene treffen, erhalten wir axonometrische Projektionen.

Die Projektionsebene E sei in impliziter Form gegeben durch $\mathbf{E:}\ \underline{\mathbf{N}} \cdot \underline{\mathbf{X}} = \mathbf{0}$ mit $\underline{\mathbf{N}} = [a,b,c,d]^t$ .Bei einer Projektion wird ein Punkt $\underline{\mathbf{P}}$ des Raumes auf den Punkt $\underline{\mathbf{P'}}$ der Projektionsebene $\mathbf{E}$ projiziert, indem der Projektionsstrahl durch den Punkt $\underline{\mathbf{P}}$ die Projektionsebene schneidet. Da der Normalen-Richtungsvektor $\underline{\boldsymbol{N}}$ der Projektionsebene $\mathbf{E}$ als Richtungsvektor der Projektionsstrahlen verwendet werden kann, hat die Projektionsgerade $\mathbf{G_p}$ die parametrische Form:

$$\mathbf{G_p:}\ \underline{\mathbf{X}} = \underline{\mathbf{P}} + r * \underline{\boldsymbol{N}}\ (r \in \Re)$$

Für die Berechnung des projizierten Punktes $\underline{\mathbf{P'}}$, d.h. des Schnittpunktes $\underline{\mathbf{P'}}$ von $\mathbf{G_P}$ mit $\mathbf{E}$ wird zunächst der Normalen-Richtungsvektor normiert:

$$\underline{\underline{\boldsymbol{N}}} = \underline{\boldsymbol{N}} \,/\, |\underline{\boldsymbol{N}}| = [a', b', c', 0]^t.$$

Durch Einsetzen des Parameterwertes

$$r_{P'} = -\frac{\underline{\underline{\mathbf{N}}} \cdot \underline{\mathbf{P}}}{\underline{\underline{\boldsymbol{N}}} \cdot \underline{\underline{\boldsymbol{N}}}} = -\,\underline{\underline{\boldsymbol{N}}} \cdot \underline{\mathbf{P}} = -\,(a' * x_P + b' * y_P + c' * z_P)$$

in die Gleichung der Projektionsgeraden $\mathbf{G_P}$ ergibt sich der Schnittpunkt (Lotfußpunkt der Lotgeraden) wie folgt:

$$\underline{P}' = \underline{P} + r_{P'} * \underline{\underline{N}} = \begin{bmatrix} (1-a'^2)* x_P - (a'*b*)*y_P - (a'*c')*z_P \\ -(a'*b')*x_P + (1-b'^2)*y_P - (b'*c')*z_P \\ -(a'*c')*x_P - (b'*c')*y_P + (1-c'^2)*z_P \\ 1 \end{bmatrix}$$

Das läßt sich in ein Produkt der Matrix T_{OP} und dem Punkt $\underline{P}$ umformen:

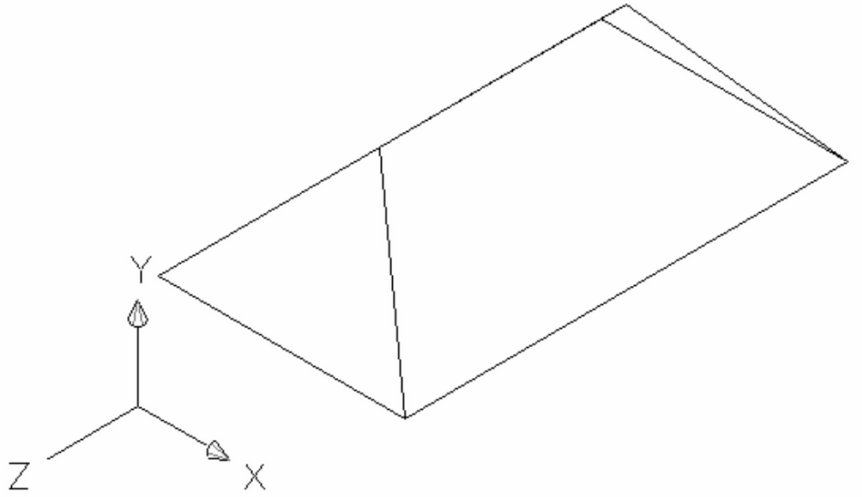
$$\underline{P}' = T_{op} * \underline{P}$$

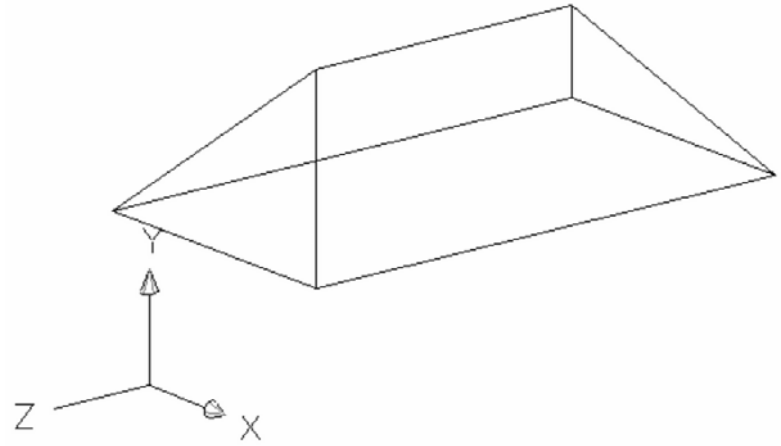
Dabei beschreibt die Matrix T_{OP} die orthogonale Parallelprojektion auf die Projektionsebene **E: $\underline{N} \cdot \underline{X} = 0$:**

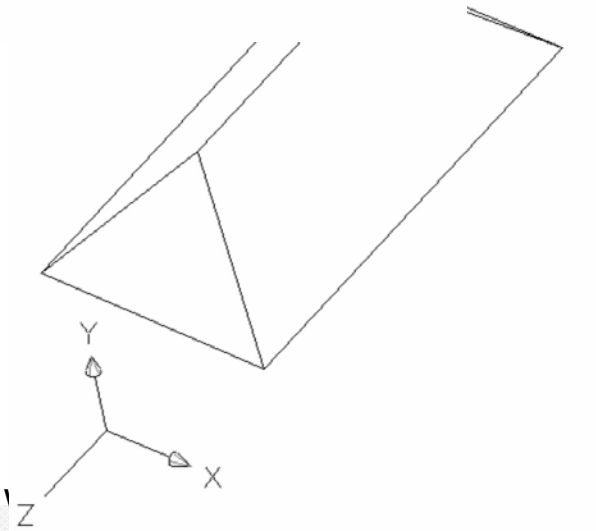
$$T_{OP} = \begin{bmatrix} (1-a'^2) & -a'*b' & -a'*c' & 0 \\ -a'*b' & (1-b'^2) & -b'*c' & 0 \\ -a'*c' & -b'*c' & (1-c'^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Wenn die Projektionsrichtung zu allen Koordinatenachsen den selben Winkel besitzt, so heißt die Projektion **isometrisch**, da sich bei der Darstellung für alle drei Achsenrichtungen die gleiche Skalierung ergibt.

Wenn die Projektionsebene zu genau zwei Koordinatenachsen den selben Winkel hat, so ist die Skalierung in zwei Koordinatenrichtungen gleich. Dies wird als **dimetrische**







Alle übrigen Winkelsituationen sind **trimetrisch**e Projektionen.

6.1.2. Schräge Parallelprojektionen

Schräge Parallelprojektionen sind solche Parallelprojektionen, bei denen die Projektionsstrahlen die Projektionsebene unter einem Winkel ϕ schneiden, der deutlich >0 ° und deutlich <90 ° ist.

In der Regel verwendet man nur solche schrägen Parallelprojektionen, bei denen die Projektionsebene parallel zu einer Koordinatenebene (z.B. der x-y-Ebene) verläuft. Dies bewirkt, daß Strecken, die parallel zu eben dieser Ebene verlaufen, in Originallänge abgebildet werden, während Strecken, die parallel zu der dritten Achse (z.B. z-Achse) verlaufen, in einem bestimmten Verhältnis skaliert werden.

Für die Herleitung der Berechnungsvorschrift soll angenommen werden, daß die x-y-Ebene als Projektionsebene verwendet wird.

Sei $\underline{\boldsymbol{R}} = [x_R, y_R, z_R, 0]^t$ der Richtungsvektor der Projektionsstrahlen einer allgemeinen schrägen Parallelprojektion auf die x-y-Ebene. (Für diese Ebene gilt $\underline{\mathbf{N}} = \underline{\boldsymbol{N}} = \underline{\underline{N}} = [0,0,1,0]^t$.)

Ein beliebiger Punkt $\underline{\mathbf{P}} = [x_P, y_P, z_P, 1]^t$ des Raumes wird bei der schrägen Parallelprojektion auf den Punkt $\underline{\mathbf{P'}}$ der x-y-Ebene abgebildet, in dem die parametrisch beschriebene Projektionsgerade $G_P: \underline{X} = \underline{P} + r*\underline{R}$ $(r \in \Re)$ die x-y-Ebene schneidet.

Der Parameterwert $r_{P'}$ für den Schnittpunkt $\underline{\mathbf{P'}} = [x_{P'}, y_{P'}, z_{P'}, 1]^t$ ergibt sich aus:

$$r_{P'} = -\frac{\underline{N} \cdot \underline{P}}{\underline{N} \cdot \underline{R}} = -\frac{\begin{bmatrix}0\\0\\1\\0\end{bmatrix} \cdot \begin{bmatrix}x_P\\y_P\\z_P\\1\end{bmatrix}}{\begin{bmatrix}0\\0\\1\\0\end{bmatrix} \cdot \begin{bmatrix}x_R\\y_R\\z_R\\1\end{bmatrix}} = -\frac{z_P}{z_R}$$

Durch Einsetzen von $r_{P'}$ in die Gleichung der Projektionsgeraden G_P erhält man den gesuchten Schnittpunkt $\underline{P'}$:

$$\underline{P'} = \underline{P} + r_{P'} * \underline{R} = \underline{P} + (-z_P/z_R) * \underline{R} = \begin{bmatrix} x_P - (x_R/z_R)*z_P \\ y_P - (y_R/z_R)*z_P \\ 0 \\ 1 \end{bmatrix}$$

Dies läßt sich als Produkt einer Matrix T_{SP} und einen Punktvektor $\underline{P}$ schreiben:

$$\underline{P'} = T_{SP} * \underline{P}$$

Dabei hat die Matrix T_{SP} die Gestalt:

$$T_{SP} = \begin{bmatrix} 1 & 0 & -x_R/z_R & 0 \\ 0 & 1 & -y_R/z_R & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

mit dem Projektions-Richtungsvektor

$$\underline{\boldsymbol{R}} = [x_R, y_R, z_R, 0]^t$$

Im Folgenden soll untersucht werden,in welchem Verhältnis eine Strecke, die parallel zur z-Achse verläuft, durch eine gegebene schräge Parallelprojektion skaliert wird. Diese Skalierung hängt im wesentlichen vom Winkel ϕ zwischen dem Projektionsstrahl und der Projektionsebene ab.

Dazu wird der Punkt $\underline{\mathbf{P}} = [0,0,1,1]^t$ der z-Achse auf die x-y-Ebene projiziert. Der projizierte Punkt $\underline{\mathbf{P'}} = [x_{P'}, y_{P'}, 0,1]^t$ ergibt sich dabei zu $\underline{\mathbf{P'}} = [-x_R/z_R, -y_R/z_R, 0, 1]^t$ (da hierbei die Projektionsrichtung entgegen der z-Achse verläuft, es also einen negativen Wert z_R gibt, wird $x_{P'}$ bzw. $y_{P'}$ wieder positiv.)

Die Strecke der Länge 1 vom Ursprung U zu Punkt $\underline{\mathbf{P}} = [0,0,1,1]^t$ wird abgebildet auf die Strecke von U zu Punkt P', deren Länge d in Abhängigkeit von ϕ angegeben werden kann:

$$d = 1/\tan(\phi) \quad (0° < \phi < 90°)$$

Die Größe d ist damit gleichzeitig der Skalierungsfaktor, mit dem alle parallel zur z-Achse verlaufenden Strecken skaliert werden.

Eine schräge Parallelprojektion auf die x-y-Ebene kann auch mit Hilfe des Winkels α zwischen der positiven x-Achse und der Geraden durch U und P'. d.h. der Projektion der positiven z-Achse auf die x-y-Ebene, angegeben werden.

Mit

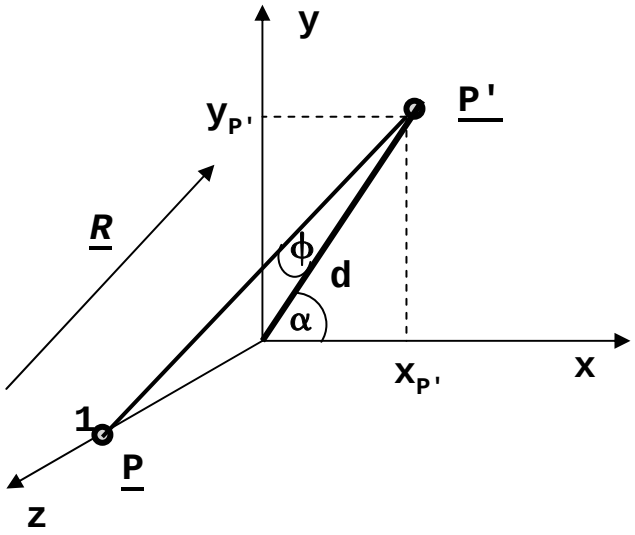
$$\sin(\alpha) = \frac{y_{P'}}{d} = \frac{-y_R}{z_R * d} \text{ und daraus } \frac{-y_R}{z_R} = d*\sin(\alpha) \text{ und analog } \frac{-x_R}{z_R} = d*\cos(\alpha)$$

kann man umformen zu:

$$T_{SP} = \begin{bmatrix} 1 & 0 & d*\cos(\alpha) & 0 \\ 0 & 1 & d*\sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Projektionsrichtung *R* kann bei gegebener Matrix T_{SP} ermittelt werden, indem man die Differenz zwischen einem projizierten Punkt P' und seinem Urbild P bildet. Für P kann man z.B. den Punkt $[0,0,1,1]^t$ auf der positiven z-Achse wählen. Mit P' = $[d*\cos(\alpha), d*\sin(\alpha),0,1]^t$ ergibt sich die Projektionsrichtung *R* wie folgt:

$$\mathbf{R} = [d*\cos(\alpha), d*\sin(\alpha), -1, 0]^t$$

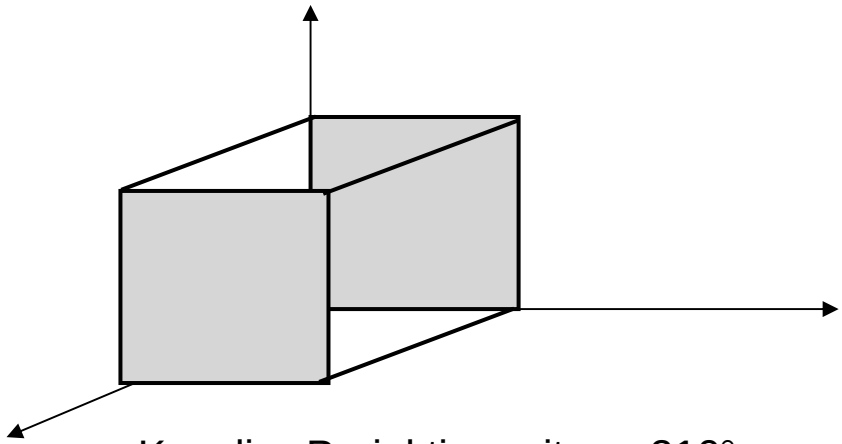


Kavalier-Projektionen sind solche schrägen Parallelprojektionen auf die x-y-Ebene, bei denen in z-Richtung keine Skalierung stattfindet, d.h. d=1.

Daraus folgt, daß die Projektionsstrahlen die Projektionsebene unter einem Winkel ϕ von 45 ° schneiden.

Die Kavalier-Projektionen können in Abhängigkeit von Winkel α zwischen der positiven x-Achse und der Projektion der positiven z-Achse angegeben werden:

$$T_{KV} = \begin{bmatrix} 1 & 0 & \cos(\alpha) & 0 \\ 0 & 1 & \sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



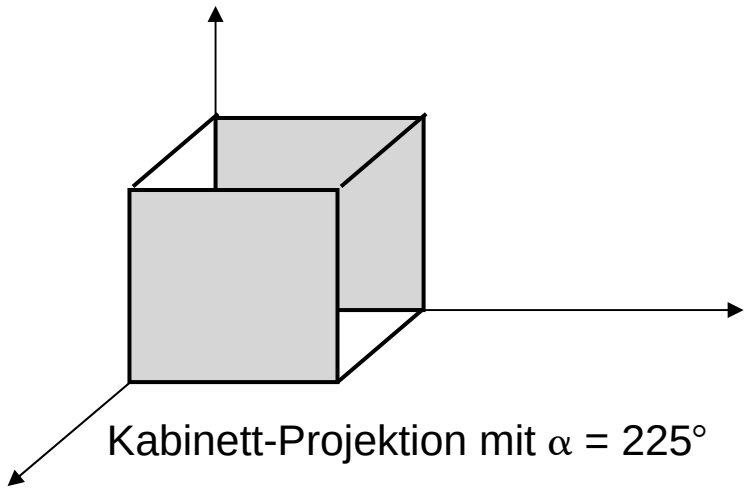
Kavalier-Projektion mit α = 210°

Bei **Kabinett-Projektionen** wird die Projektionsrichtung so gewählt, daß Strecken, die parallel zur z-Achse verlaufen, bei ihrer Projektion auf die x-y-Ebene auf die halbe Länge verkürzt werden, d.h. d=1/2.

Der Winkel ϕ, unter dem die Projektionsstrahlen die Projektionsebene schneiden, ergibt sich so zu $\phi = \arctan(2) \sim 63{,}5\,°$.

Die Kabinett-Projektion T_{KB} ergibt sich für beliebig wählbare Winkel α zwischen der positiven x-Achse und der Projektion der positiven z-Achse zu:

$$T_{KB} = \begin{bmatrix} 1 & 0 & \cos(\alpha)/2 & 0 \\ 0 & 1 & \sin(\alpha)/2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Kabinett-Projektion mit $\alpha = 225°$

6.2. Perspektivische Projektionen

Die Perspektive ist eine Zentralprojektion, die Sehstrahlen vom Augpunkt (Center of Projection = COP) zu jedem Punkt des darzustellenden Objektes definiert. Diese Strahlen haben jeweils einen Schnittpunkt mit der Projektionsebene, unabhängig davon, ob diese – aus der Sicht des Augpunktes – sich vor oder hinter den Objektpunkten befindet.

Aus dem COP und einem Punkt P wird der Sehstrahl bestimmt, dessen Schnittpunkt mit der Projektionsebene P' gilt es zu bestimmen.

Für die weitere Betrachtung soll die Projektionsebene die x-y-Ebene sein, der Augpunkt soll sich auf der positiven z-Achse befinden.

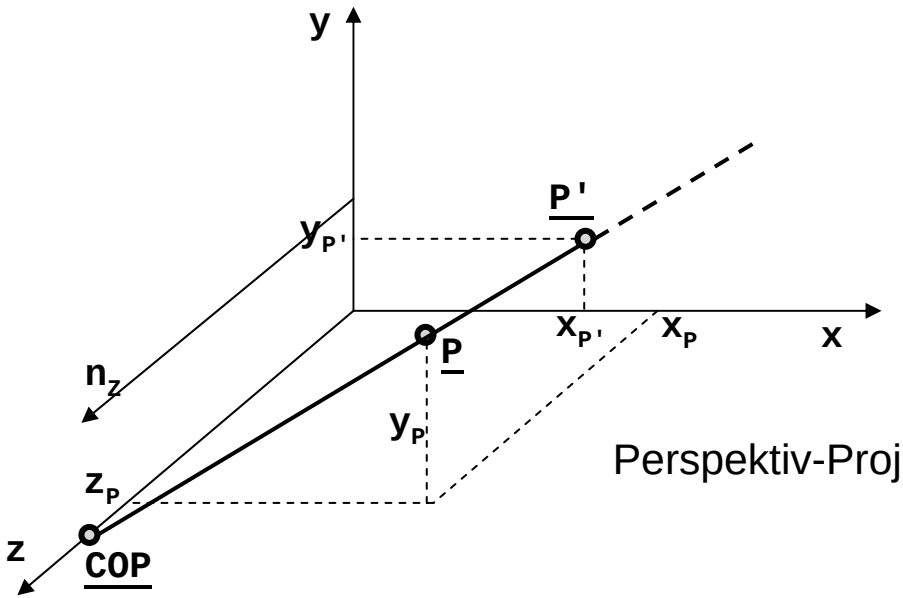
Die Matrix M_Z bestimmt sich dann zu:

$$M_Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ n_X & n_Y & n_Z & 1 \end{bmatrix}$$

wobei n_X, n_Y, n_Z die Komponenten des Normalen-Richtungsvektors der Projektionsebene sind.

Befindet sich der Augpunkt auf der positiven z-Achse, so sind $n_X=0$ und $n_Y=0$, während n_Z der Abstand des Augpunktes von der Projektionsebene ist.

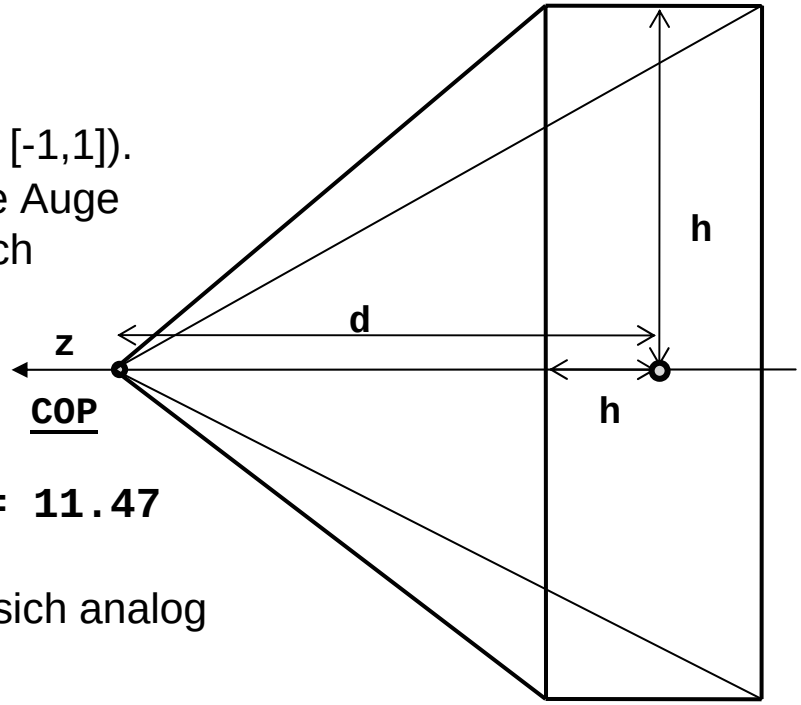
Berechnet man nun den projizierten Punkt P' = M_Z * P , so ist zu beachten, daß im Allgemeinen die vierte Komponente w ungleich 1 ist und so P' noch homogenisiert werden muß.



Perspektiv-Projektion

Bei der so beschriebenen Berechnung der Projektionen von Punkten **P** zu deren Projektionen auf der Projektionsebene **P'** werden aufgrund der Allgemeingültigkeit des Algorithmus ALLE Punkte projeziert, also auch solche, die sich im Rücken des Projektionszentrums ($z_P > z_{COP}$) befinden, als auch alle weit außerhalb des üblichen Öffnungswinkels einer (virtuellen) Kamera bzw. des Auges befindlichen Punkte behandelt. Das würde zu unrealistischen Bildern führen und die Performance des Verfahrens unnötig verschlechtern.
Abhilfe schafft das Clipping der Punkte an einer Sehpyramide. Dabei werden alle Punkte ignoriert, die sich außerhalb der Pyramide bzw. nach einer definierten "far plane", der Basisfläche der Pyramide, befinden.

Meist wird im Defaultfall der Wert für **h=1** gesetzt (Projektionsebene **x** $\in$ [-1,1], **y** $\in$ [-1,1]). Wenn man bedenkt, daß das menschliche Auge im Normalfall 170º-Bilder erfaßt, würde sich hierfür der Abstand des COP wie folgt berechnen:



cos (85°) = h/d
d = h/ cos(85°) = 1/0.0871557 = 11.47

Bei einem Öffnungswinkel von 60° ergibt sich analog
d = 1.15

Die Menge der gültigen, d.h. in der Sehpyramide befindlichen Punkte **P** läßt sich leicht durch folgende Wertebereiche ermitteln:

P ist zu betrachten, wenn

$$x_P \in [-(1-z_P/d)*h,(1-z_P/d)*h] \;\&\&$$
$$y_P \in [-(1-z_P/d)*h,(1-z_P/d)*h] \;\&\&$$
$$z_P \in [z_{Far}, z_{COP}]$$

7. Spezielle Verfahren der rechnergestützten Konstruktion

7.1. Planare Triangulation

7.1.1. Aufgabenstellung

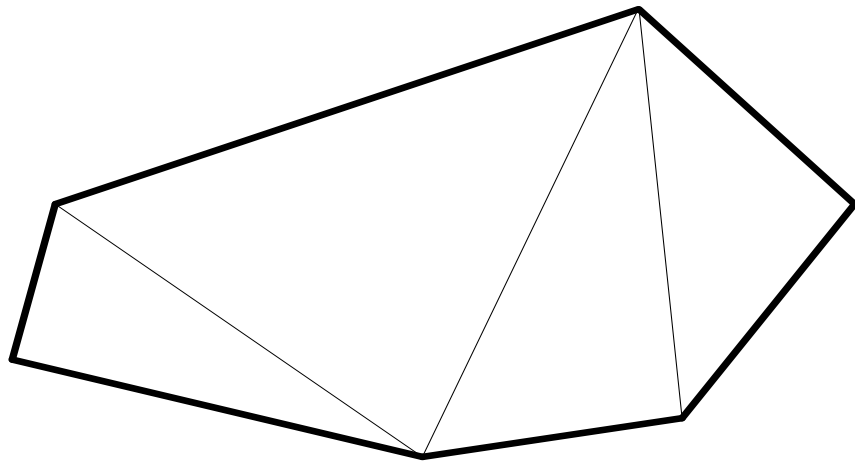
Situation:

Sowohl für die Darstellung, aber auch für numerische Berechnungen ist es erforderlich, eine beliebige planare, geradlinig begrenzte Fläche in Dreiecksflächen zu unterteilen. Diesen Vorgang nennt man Triangulation.

Ein in GDV bereits behandeltes Triangulationsverfahren ist das Delaunay-Verfahren auf der Grundlage von Voronoi-Diagrammen.

Zwei qualitative Forderungen bestehen an eine algorithmische Lösung:
a) Die Gesamtlänge aller Triangulationskanten soll ein Minimum bilden.
b) Die Größe der inneren Winkel der Dreiecke soll möglichst "normal" sein, d.h. sie sollten möglichst die gleiche Größe annehmen.

Da die Triangulation immer ein im Hintergrund ablaufendes Verfahren ist, für das der User möglichst keine Zeit einräumen möchte, ist die Berechnungszeit (als Funktion von der Anzahl der Flächenknoten) eine entscheidende Eigenschaft.



7.1.2. Die "Greedy"-Triangulation von Preparata und Shamos

Einschränkungen:

Behandelt werden nur konvexe Polygonflächen ohne Löcher. Eine Erweiterung des Verfahrens ist aber leicht durchführbar..

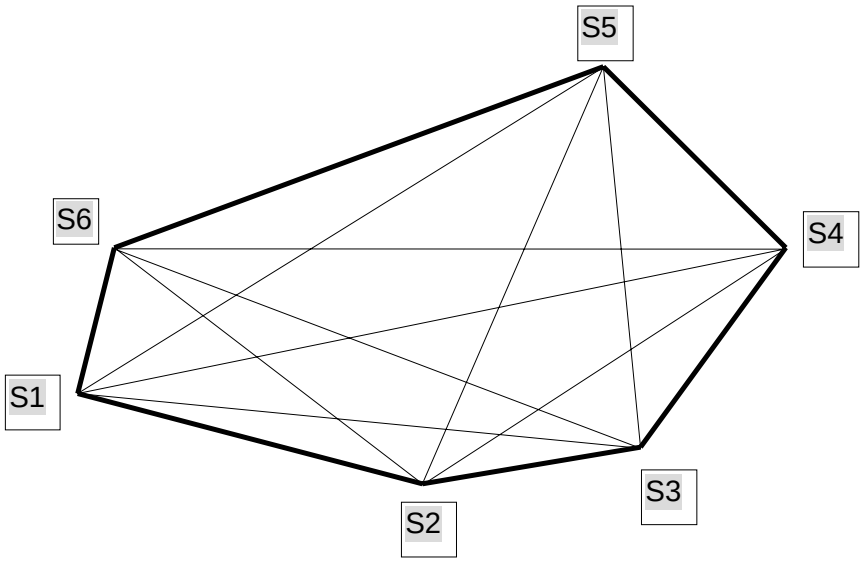
A) Struktur der Ausgangsdaten und erste elementare Operationen

Gegeben ist die Polygonfläche durch einen geordneten Satz S ihrer N Knotenpunkte. Die Ordnung repräsentiert den Umlaufsinn der Begrenzungskanten und damit auch eine Innen/Außen-Information.

Im ersten Schritt werden alle Begrenzungskanten, also alle N Kanten zwischen den Knotenpunkten S_1-S_2, S_2-S_3, ... S_{n-1}-S_n, S_n-S_1, in die Datenstruktur für die Triangulationskanten T übernommen.

B) Berechnung aller übrigen $\binom{N}{2}$ - N Kanten zwischen allen Knoten des Polygons

Es werden alle möglichen Verbindungen zwischen den N Knoten (abzüglich der bereits gebildeten Begrenzungskanten) zu Kanten generiert. Diese Kanten werden in einem Kantenpool P gespeichert und dort der Länge nach aufsteigend sortiert.



Beispiel:

N = 6
NP = (6 über 2) - 6 = 9

P (sortiert):
S3-S5
S2-S6
S2-S4
S2-S5
S3-S6
S1-S3
S1-S5
S4-S6
S1-S4

C) Auswahl der Triangulierungskanten

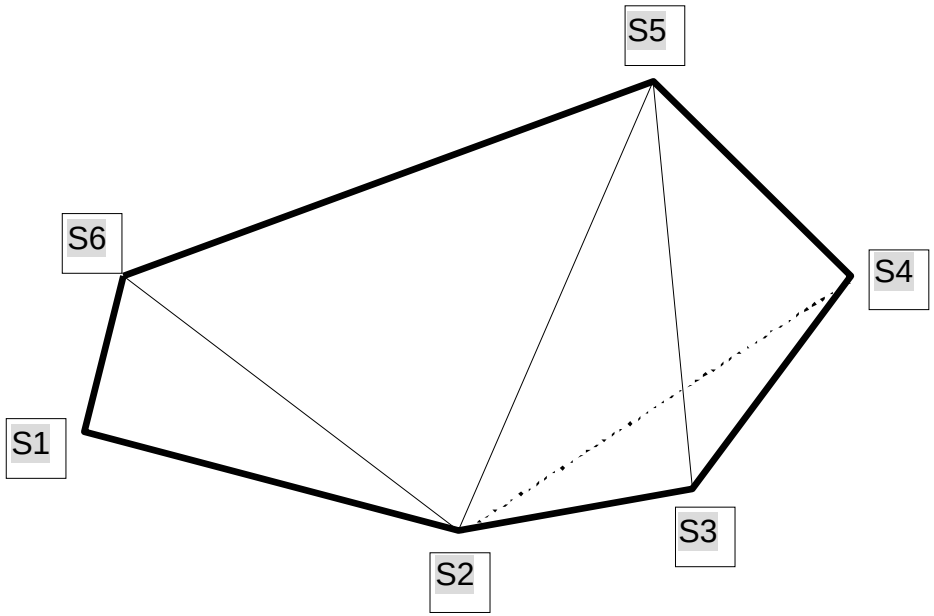
Die Vorsortierung der potentiell möglichen Kanten im Kantenpool P berücksichtigt - unter einem sehr lokalem Aspekt - die Forderung nach dem Minimum der Summe der Kantenlängen. Wir wählen also die jeweils kürzeste (d.h. erste) Kante des Kantenpools P aus und übertragen sie in die Liste der Triangulationskanten T.

Dabei muß aber noch eine wichtige Bedingung erfüllt sein:

Die neu einzutragende Kante darf keine bisher schon vorhandene Kante schneiden.

Dieser Vorgang wird abgebrochen, wenn entweder die Triangulierung vollständig ist oder der Kantenpool leer ist.
Die Vollständigkeit errechnet man wie folgt:

N Knoten des Polygons erfordern (N-2) Dreiecke,
(N-2) Dreiecke benötigen 3*(N-2) = 3*N-6 Kanten,
N Kanten werden durch die Randkontur gebildet, verbleiben 2*N-6 Kanten,
jede innere Kante gehört zu zwei Dreiecken, also benötigt man
(2*N-6)/2=N-3 innere Kanten in der Triangulierungsliste T.



Beispiel:

N = 6
NP = 9

NT = N-3 = 3 innere Kanten

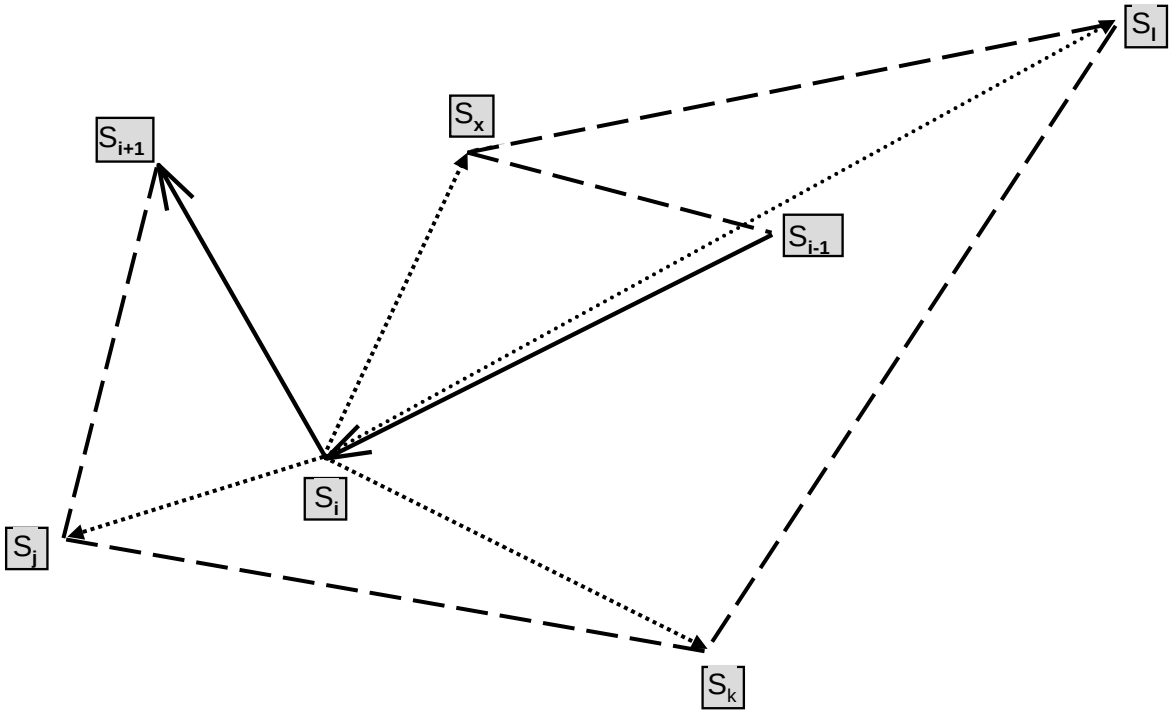
P (sortiert):
S3-S5 ja
S2-S6 ja
S2-S4 nein
S2-S5 ja und fertig !

S3-S6
S1-S3
S1-S5
S4-S6
S1-S4

D) Erweiterung des Algorithmus auf konkave Polygonflächen

Würde man bei konkaven Polygonrändern wie oben verfahren, werden im Kantenpool Verbindungslinien eingetragen, die vollständig oder teilweise außerhalb der Fläche liegen. Das muß bereits bei der Generierung verhindert werden.

Betrachtet man die Situation an einem Knoten $\mathbf{S_i}$, für den eine neue (möglichst innere) Kante zu irgendeinem anderen Knoten $\mathbf{S_x}$ generiert werden soll, so stellt man fest, daß bereits zwei Randkanten an diesem Knoten hängen, eine "ankommende" $\mathbf{S_{i-1}}$-$\mathbf{S_i}$ und eine "weggehende" $\mathbf{S_i}$-$\mathbf{S_{i+1}}$.



Sind die Kanten so orientiert, daß sich links der Kante immer die Fläche befindet (rechte Hand - Regel), lassen sich untaugliche Knotenverbindungen durch folgende Berechnungsschritte ermitteln:

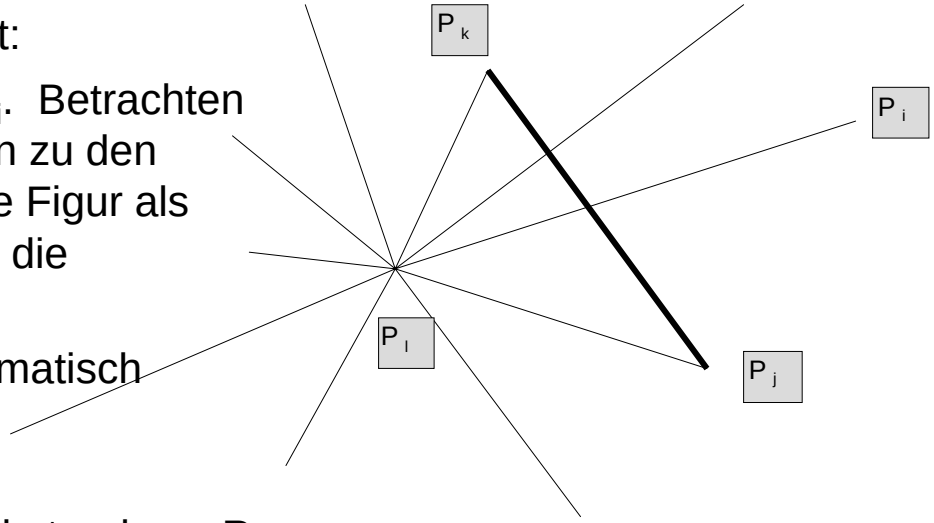
- Ermittlung der Winkel (gegenüber der positiven X-Achse) im Knoten $\mathbf{S_i}$, die die ankommende Kante $\mathbf{S_{i-1} \rightarrow S_i}$ (Anfangswinkel) und die abgehende Kante $\mathbf{S_i \rightarrow S_{i+1}}$ (Endwinkel) bilden,
- Ist der Endwinkel kleiner als der Anfangwinkel, wird zu ihm 360° = 2π addiert.
- Anfangs- und Endwinkel, zwischen denen sich der Außerhalb-Bereich befindet, werden bei jedem Knoten gespeichert.
- Wird nun eine innere Kante $\mathbf{S_i \rightarrow S_x}$ auf Zulässigkeit getestet, wird auch für sie der Winkel am Knoten (gegenüber der positiven X-Achse) berechnet. Liegt dieser Winkel φ oder der Wert (φ+360°) zwischen Anfangs- und Endwinkel, ist diese Kante nicht zulässig.
- Alle verbleibenden Knotenvektoren $\mathbf{S_i \rightarrow S_x}$ müssen nun auf einen möglichen Schnittpunkt mit allen bereits gesetzten inneren Kanten überprüft werden. Existiert ein solcher Schnittpunkt, ist die Verbindungskante für das weitere Vorgehen zu verwerfen.

Analog wird mit den Begrenzungskanten von Löchern vorgegangen.

E) Effiziente Durchführung des Schnitt-Testes

Im Prinzip besteht der Test aus der Überprüfung, ob die betrachtete Verbindungskante $\mathbf{S_i \rightarrow S_x}$ einen Schnittpunkt mit einer oder mehreren der bis zu diesem Zeitpunkt ermittelten NT Kanten der Triangulierungsliste T hat. Konventionelle Berechnungsverfahren (z.B. Schnittpunkt zweier Strecken mit dem Kreuzprodukt) sind aber für diese Aufgabe oft viel zu zeitaufwendig.

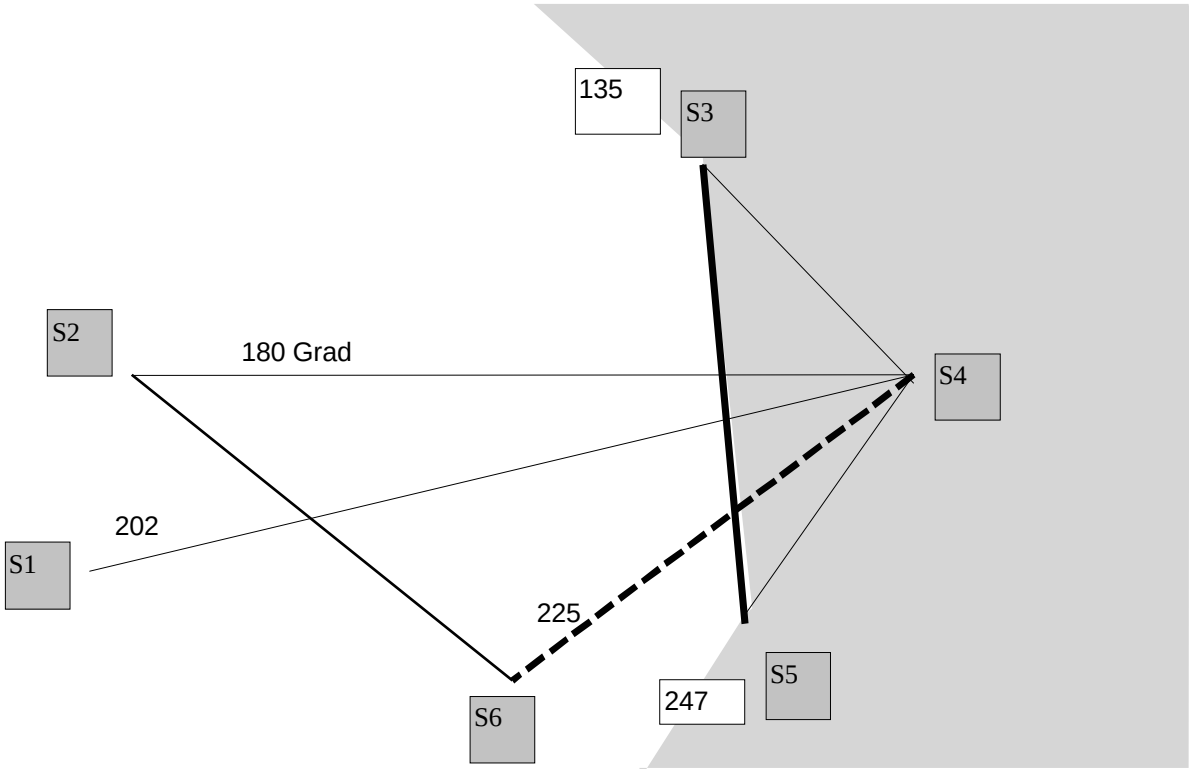
Ziel ist es, den Entscheidungstask vom Aufwand her mit dem Auswahltask auszubalancieren. Das wird durch folgende Vorgehensweise erreicht:



- Die aktuelle zu testende Kante ist $\mathbf{P_l - P_i}$. Betrachten wir alle am Knoten $\mathbf{P_l}$ hängenden Kanten zu den anderen Knoten von S, können wir diese Figur als "Speichenstern" von $\mathbf{p_l}$ bezeichnen und die folgenden Eigenschaften definieren:
- Die Speichen lassen sich z.B. im mathematisch positiven Sinn sortieren.
- Sie teilen den 2π-Winkel um $\mathbf{P_l}$ in (N-1) Winkelintervalle bzw. - wenn man den Abstand von $\mathbf{P_l}$ hinzu nimmt - in abgeschlossene Sektoren..
- Wird eine Kante $\mathbf{P_j - P_k}$ in die Triangulierungsliste T übernommen, spannt sie einen Satz zusammenhängender Sektoren in jedem Speichenstern $\mathbf{p_{ind}}$, ind $\neq$ j, k auf.
- Wenn sich, wie oben definiert, keine der Kanten in der Triangulierungsliste T schneiden soll, heißt das, daß die zu testende Kante $\mathbf{P_l - P_i}$ im Speichenstern ($\mathbf{p_l}$) keine der im Winkelsektor von $\mathbf{P_i}$ gesetzten Kante schneiden darf.

Im vorherigen Beispielbild fällt der Punkt $\mathbf{P_i}$ der zu testenden Kante $\mathbf{P_l - P_i}$ in den Sektor $\mathbf{P_jP_lP_k}$. Um zu entscheiden, ob die Kante $\mathbf{P_l}$-$\mathbf{P_i}$ der Triangulierungsliste zugefügt werden darf, müssen wir kontrollieren, ob sie einen Schnittpunkt mit der diesen Sektor schneidenden Triangulierungskante $\mathbf{P_j - P_k}$ hat und genauer, welche der beiden (Knoten $\mathbf{P_i}$ oder Kante $\mathbf{P_j}$-$\mathbf{P_k}$) näher zu $\mathbf{P_l}$ liegt.

Auf diese Weise wurde die Entscheidung auf einen speziellen Fall von planarer Punktlokalisation reduziert, bei der wir in jedem Sektor der Sterne ($\mathbf{p_l}$), für alle l, eine Triangulierungskante, die sogenannte spanning edge, unterbringen können.



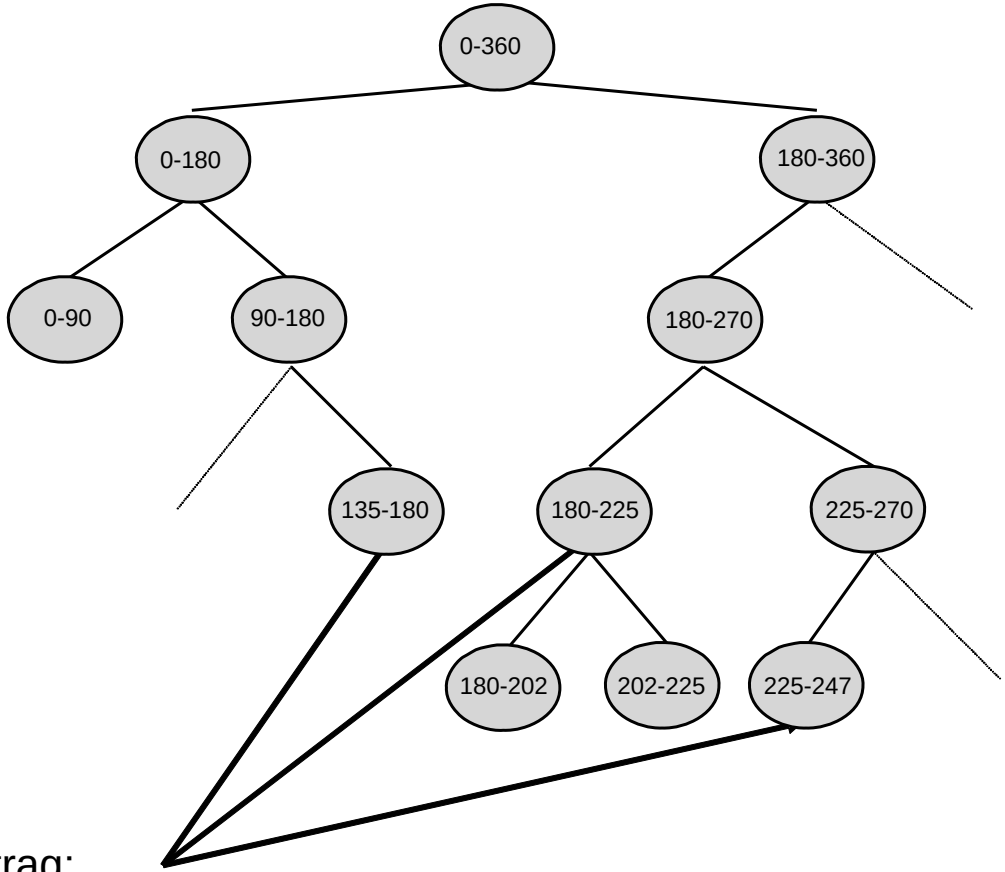
Die gefärbten Winkelsegmente des Speichensterns von S4 könnten - theoretisch - noch Endpunkte von Kanten aufnehmen.

F) Speicherung der Speichensterne in binären Segmentbäumen

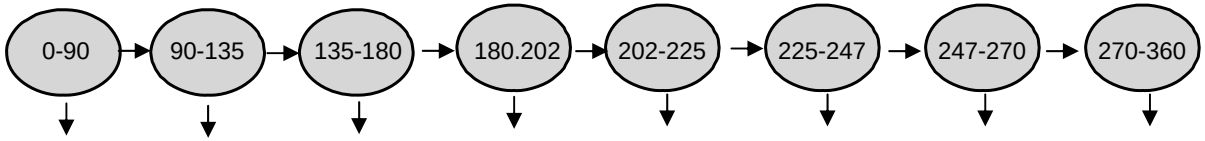
Die für die Speicherung und insbesondere für das regelmäßige Updaten der Speichensterne bei jedem Einfügen einer neuen akzeptierten Kante erforderliche Datenstruktur muß berücksichtigen, daß immer eine Folge verbundener Sektoren in jedem Speichenstern definiert werden muß. Die geeignete Struktur ist dann ein Segmentbaum (segment tree).

Die Sektoren eines Sterns ($\mathbf{p_l}$) sind die (N-1) Blätter eines Segmentbaumes $\mathbf{B_l}$. Jeder Knoten k von B_l ist mit einer Kante e(k) assoziiert, die die nächstliegende Kante zu $\mathbf{p_l}$ von allen dem Knoten k durch Einfügen zugeordnete Kanten ist. (e(k) kann auch eine leere Kante, also noch nicht gesetzt sein.)

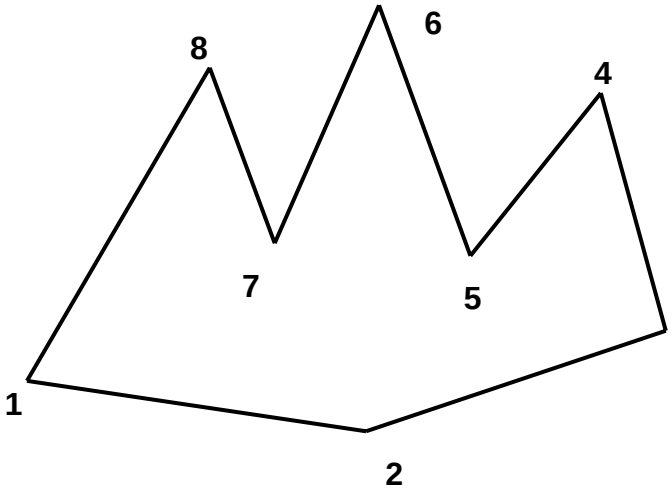
Um die Kante $\mathbf{P_l}$-$\mathbf{P_i}$ zu testen, untersuchen wir im Baum $\mathbf{B_l}$ den Pfad, der durch $\mathbf{P_i}$ identifiziert wird. Bei jedem Knoten k in diesem Pfad testen wir $\mathbf{P_l}$-$\mathbf{P_j}$ bezüglich der Kante e(k) und akzeptieren die Kante $\mathbf{P_l}$-$\mathbf{P_j}$ nur, wenn sie keine der angetroffenen Kanten schneidet. Diese Suche ist beendet, wenn die aktuell betrachtete Kante abgewiesen wird. Im anderen Fall, also wenn die Kante $\mathbf{P_l}$-$\mathbf{P_j}$ akzeptiert wurde, muß jeder Stern ($\mathbf{p_{ind}}$), ind $\neq$ l, i, noch aktualisiert werden. Das wird einfach durch Einfügen von $\mathbf{P_l}$-$\mathbf{P_j}$ in die Sterne ($\mathbf{p_{ind}}$) und - wenn erforderlich - durch Updaten jedes Knotens k im Einfügepfad vorgenommen.

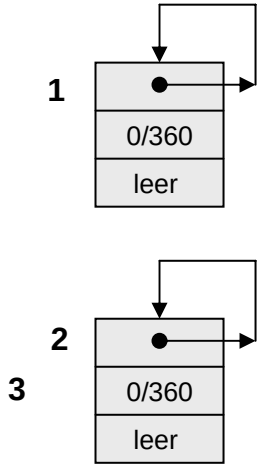


Eintrag:
Kante $\mathbf{S_3 \rightarrow S_5}$

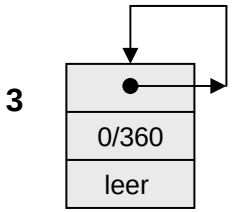


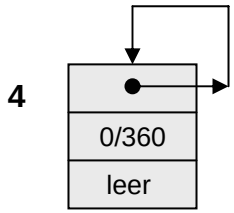
7.1.3. Beispiel für die Arbeit mit dem Speichenstern

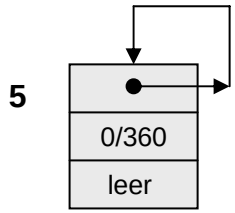


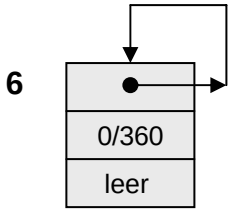


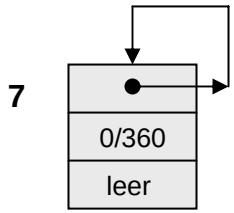
1. Generieren des Wurzelgliedes eines jeden Speichensterns (je Knoten eine verkettete Liste) und Eintragen des Gesamtwinkels als leer.

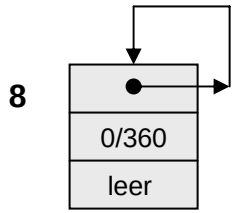






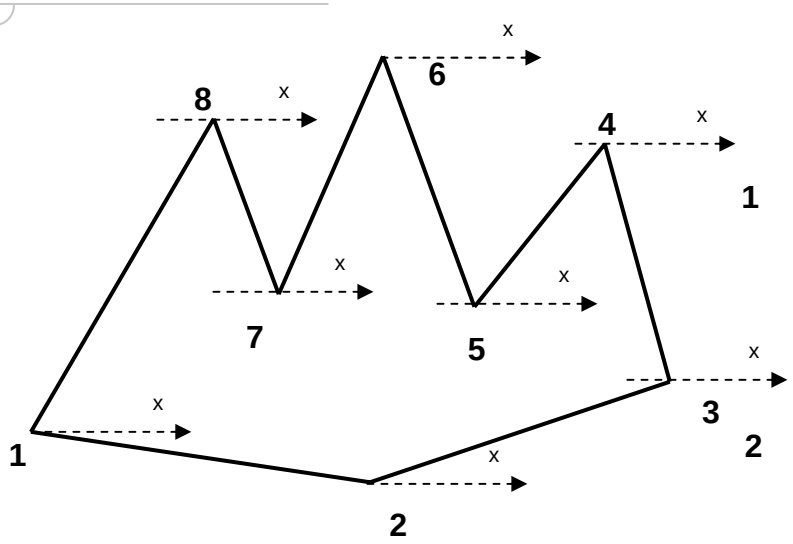


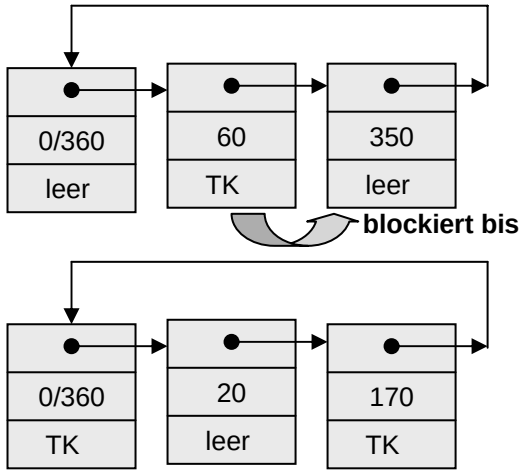


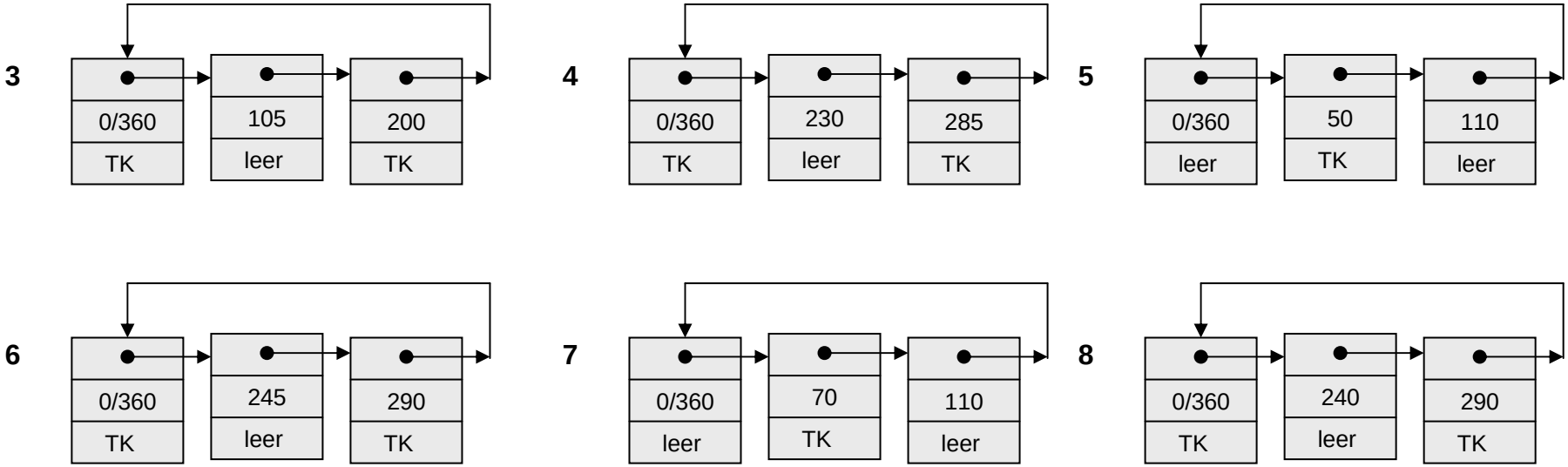


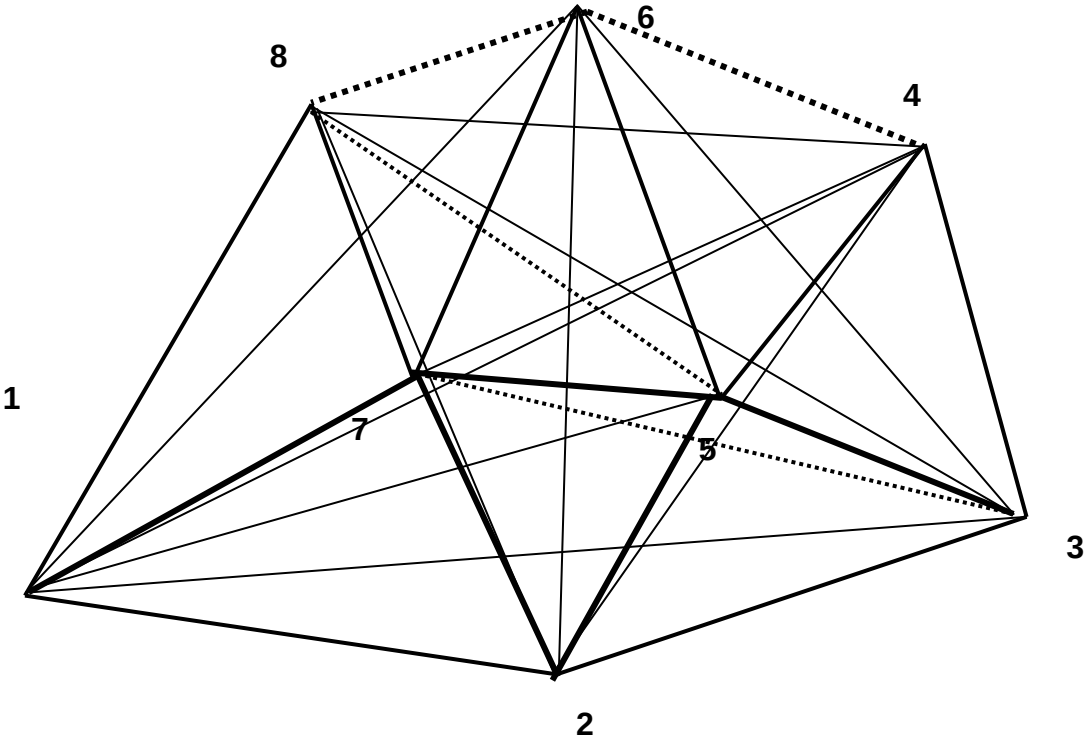
2. Eintragen der Konturkanten :

Bei konkaven Hüllpolygonen wird durch die Konturkanten in jedem Stern der "Außerhalbwinkel" blockiert (Eintrag durch Code T_K).









3. Generieren aller möglichen (inneren) Verbindungsstrecken (8 über 2 - 8 = 20) zwischen den Eckpunkten und sortieren im Kantenpool P.

1) 6-8
2) 5-7
3) 3-5
4) 2-5
5) 2-7
6) 4-6
7) 1-7
8) 5-8 Spezielles Problem !
9) 3-7 nur beachtet, um Schnitt mit gesetzten Kanten zu zeigen

...

Benötigt werden 8 – 3 = 5 gültige innere Strecken, begonnen wird mit der kürzesten.

1) 6-8 hat im Knoten 6 den Winkel 200°.

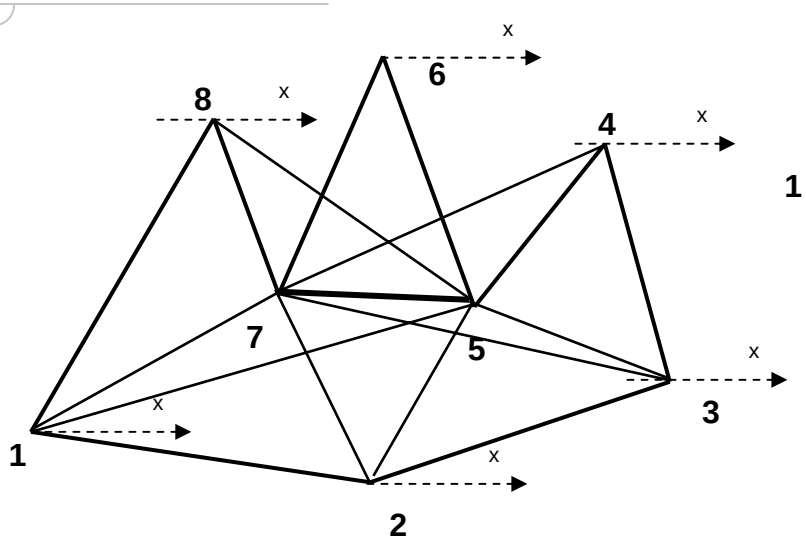
Dieser Bereich ist im Speichenstern 6 blockiert ! → Verwerfen.

2) 5-7 hat im Knoten 5 den Winkel 175°, im Speichenstern 5 ist dieser Bereich leer, also erlaubt.

Nun werden in allen Sternen außer 5 und 7 (also 1, 2, 3, 4 und 8) der jeweilige Sektor, den 5-7 dort aufspannt, blockiert.

4. Eintragen der gesetzten inneren Kanten :

Der jeweils durch eine gesetzte Kante blockierte Winkel wird in allen "übrigen" Sternen durch Eintrag des Index der blockierenden Kante i_P im Kantenpool P markiert.



1

0/360	15	30	60	350
leer	2	leer	TK	leer

2

0/360	20	60	115	170
TK	leer	2	leer	TK

3

0/360	105	155	165	200
TK	leer	2	leer	TK

4

0/360	205	230	285
TK	2	leer	TK

5

0/360	50	110
leer	TK	leer

6

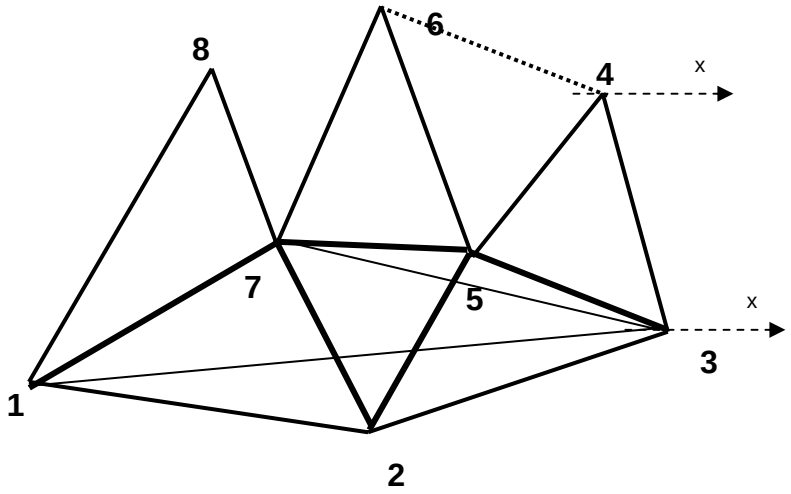
0/360	245	290
TK	2	TK

7

0/360	70	110
leer	TK	leer

8

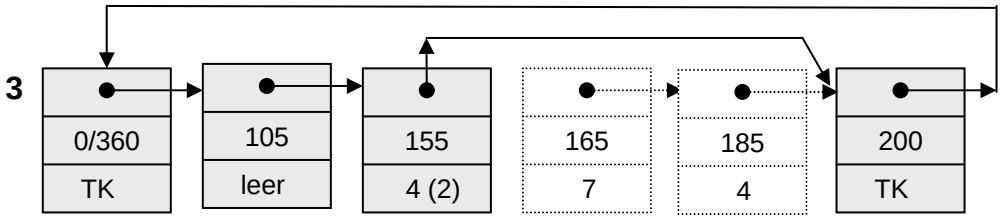
0/360	240	290	325
TK	leer	TK	2

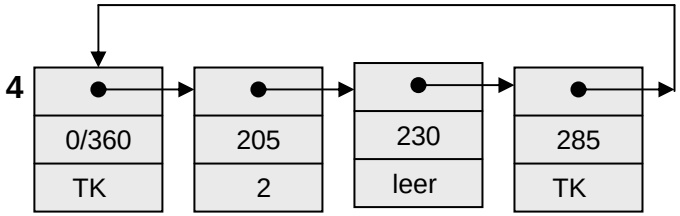


Analog wird auch mit den weiteren zu setzenden inneren Kanten verfahren, also mit
3) 3-5 → Eintrag in den Sternen 1, 2, 4, 6, 7 und 8
4) 2-5 → Eintrag in den Sternen 1, 3, 4, 6, 7 und 8
5) 2-7 → Eintrag in den Sternen 1, 3, 4, 5, 6 und 8

Die Kante 6) 4-6 hat im Stern 4 einen Winkel von 160°, ist also durch die Konturkante blockiert. Verwerfen!

7) 1-7 → Eintrag in den Sternen 2, 3, 4, 5, 6 und 8
Zur vereinfachten Darstellung wird das nur für den Stern 3 weiter ausgeführt.

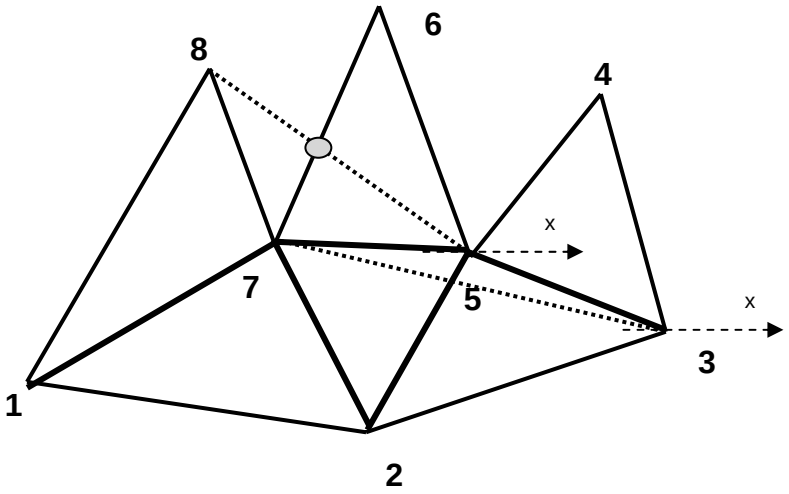


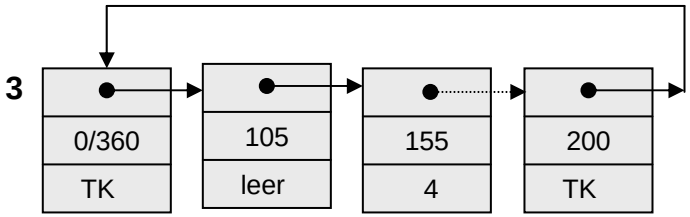


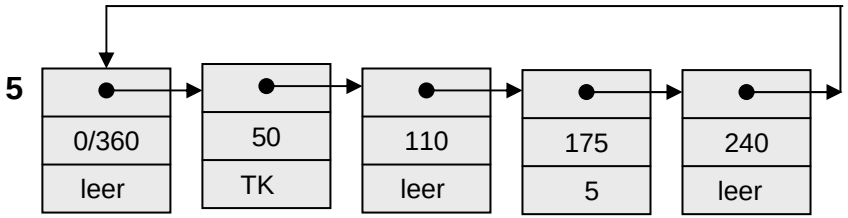
Bemerkung:

Soll durch eine gesetzte Kante ein Winkel blockiert werden, der ganz oder teilweise bereits durch eine andere Kante blockiert wird (in unserem Fall 2-5 mit Überlappung zu 5-7 oder später 1-7 mit Überlappung zu 2-5) wird die Kante notiert, die näher am Knoten liegt.

Test: z.B. liegt Knoten 7 auf der selben Seite bezüglich 2-5 wie Knoten 3 (Zentrum)?
NEIN, also 2) 5-7 überschreiben!







Nehmen wir nun an, die Kante 9) 3-7 müßte noch überprüft werden (muß sie nicht, da bereits genug Kanten gefunden wurden). Diese Kante hat im Stern 3 eine Winkel von 165°, ist also durch die Kante 4) 2-5 blockiert. Verwerfen!

5. „Extrem konkave" Kontur
Schwieriger wird es mit einer möglichen Kante 8) 5-8 .
Zwar wird der Winkel von 50° – 110° durch die Außenkontur, der Winkel 175° – 240° durch die Kante 5) 2-7 blockiert,
der Winkel für 5-8 = 140° ist aber im Stern 5 noch frei.
Dennoch darf die Kante nicht generiert werden, da sie die Kontur 6-7 schneidet und damit außerhalb der Polygon-fläche liegt.
Hier hilft nur eine (prinzipielle) Überprüfung auf mögliche Schnittpunkte mit n-2 Konturkanten (und nur mit diesen, also nicht mit den gesetzten inneren Kanten).
Ein anderer Weg wäre, die Speichensterne bereits beim Generieren der Konturkanten für diese Winkel zu besetzten.

Für den Speichenstern 5 soll das hier exemplarisch dargestellt werden.
Auch hier gilt bei überlappenden Winkelbereichen mehrerer sperrender Konturkanten, das diejenige in die Datenstruktur eingetragen wird, die näher am Knoten liegt.
Das läßt sich im konkurrierenden Fall durch Rechts-Links-Test bezüglich der bereits gesetzten Kante ermitteln.

Konturkanten

K1) 1-2

K2) 2-3

K3) 3-4

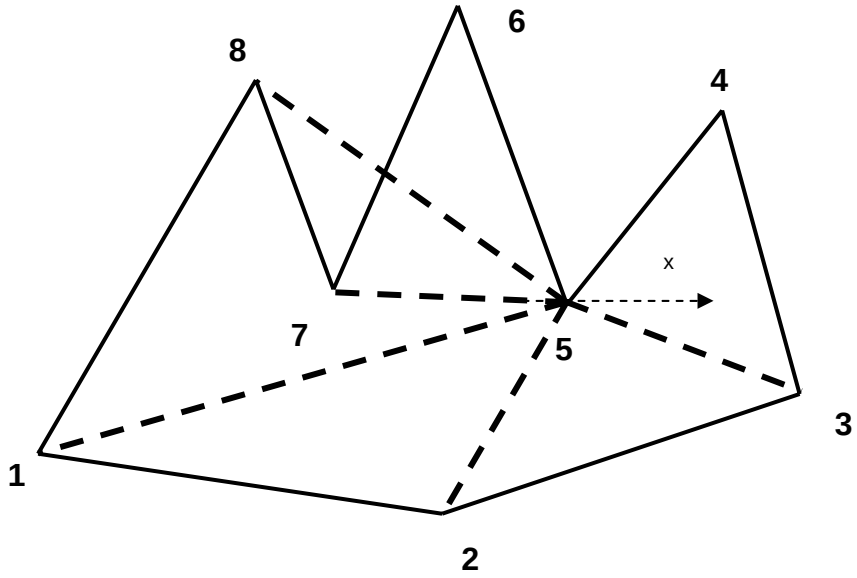
K4) 4-5

K5) 5-6

K6) 6-7

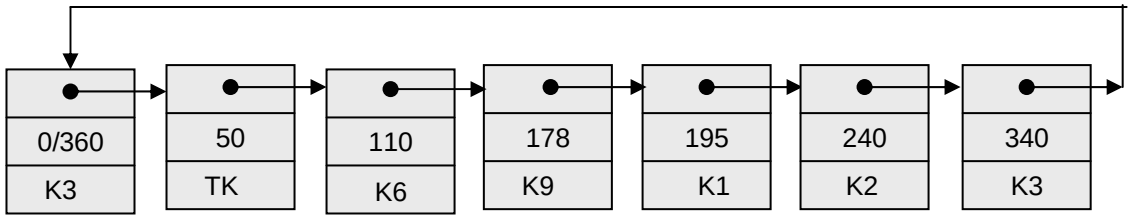
K7) 7-8

K9) 8-1



Die Konturkante K7 hat hier keinen Einfluß, da die Kante K6 ihren gesamten Winkel bedeckt und näher am Knoten 5 liegt.
Die Kante K9 belegt nur einen Teilwinkel (178-195°**),** da K6 bereits den Winkel 110-178° blockiert und näher am Knoten 5 liegt.

5



7.2. 3D-Rekonstruktion aus Dreitafelprojektionen

7.2.1. Aufgabenstellung

Situation:

•Im Denken und in der Ausbildung des (Maschinenbau-) Ingenieurs ist die Beschreibung einer Gestalt in der Dreitafelprojektion (Grundriß, Aufriß, Seitenriß) traditionell die verständlichste Form.

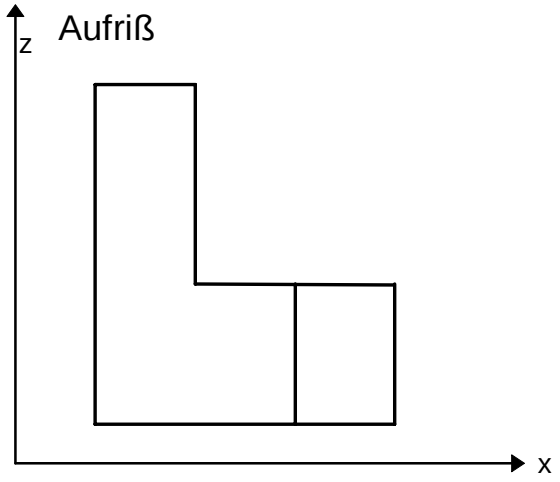
•Die grafische Präsentation der Risse ist oft unscharf (Handskizze, separate Zeichnungen, digitalisierte Fotos etc.). Es handelt sich also nicht um Rechnerprojektionen.

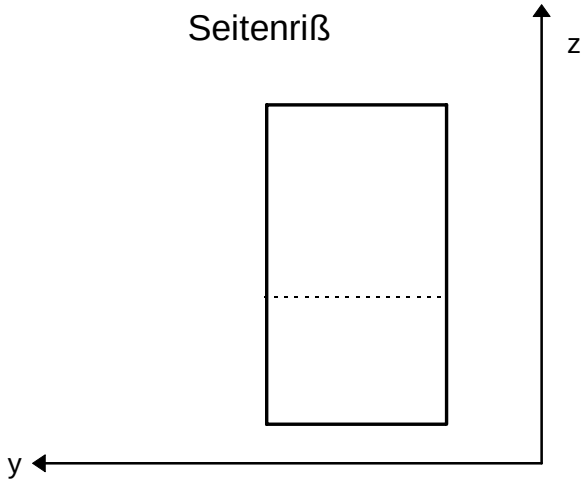
•Die Maße des Objektes lassen sich nicht unmittelbar aus der Zeichnung ableiten; erst die Bemaßungsinformation selbst determiniert die Abmessungen. Der Benutzer ist also gezwungen, die Bemaßungsinformation selbst hinzuzufügen.

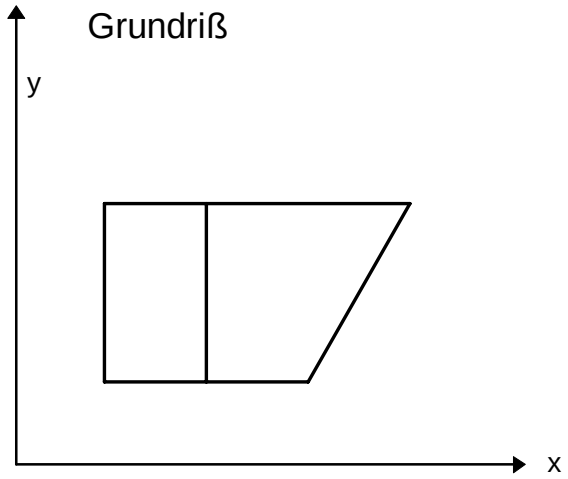
•Die Parametrisierung der Gestalt ist oft unerläßlich.

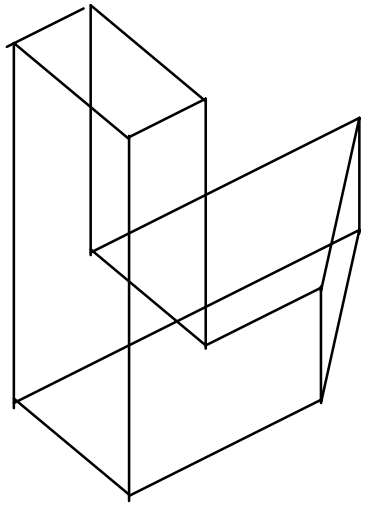
•Ziel ist immer ein universell verarbeitbares CAD-Gestaltmodell, also ein rechnerinternes 3D-Modell.

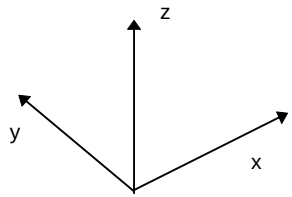
Beispiel:
Gegeben seien drei senkrechte Parallelprojektionen eines dreidimensionalen Objektes.
Die folgenden Abbildungen zeigen den Grundriß (Projektion auf x-y Ebene), den Seitenriß (Projektion auf y-z Ebene) und den Aufriß (Projektion auf x-z Ebene).
Vom Rechner wird aus den drei Rissen das folgende dreidimensionale Objekt rekonstruiert:


Aufriß
z
x


Seitenriß
z
y


Grundriß
y
x



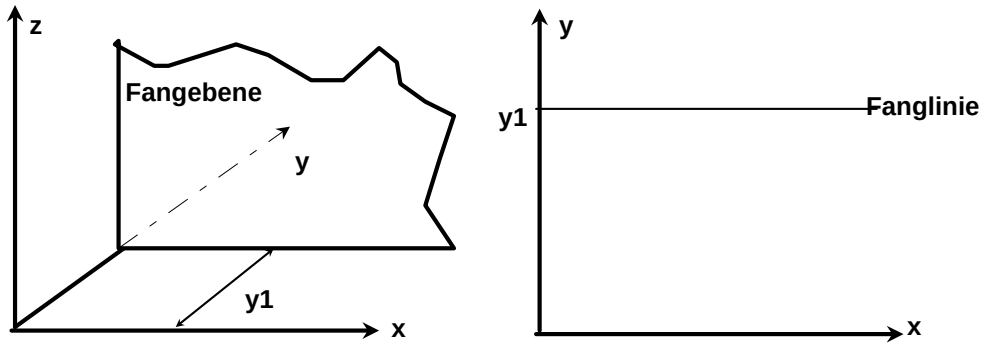

z
y
x

7.2.2. Das Verfahren (nach K. Wewer und N. Lübecke)

Einschränkungen: Behandelt werden nur ebenflächig begrenzte Objekte.
Erzeugt wird eine BRep-Darstellung.

A) Behandlung der unscharfen Vorlagen

Korrigiert wird sowohl die Zeichnungs- als auch die Digitalisier-Ungenauigkeit. Das ist eine unerläßliche Voraussetzung für die exakte räumliche Zuordnung der Eckpunkte und Maßketten. Nach der Korrektur liegen die Eckpunkte auf diskreten Rasterkoordinaten. Die maximal zulässige Eingabegenauigkeit, **Fangweite** genannt, wird vom User vorgegeben. Gefangen wird dreidimensional, d.h. in den drei Rissen jeweils paarweise für zwei Achsen. Haben z.B. drei Eckpunkte eines Körpers die y-Koordinate y1, so bilden diese drei Punkte eine Ebene senkrecht zur y-Achse mit dem Abstand y1 vom Nullpunkt.



Definition der Fangebene

Korrektur eines eindigitalisierten Koordinatenwertes y_k (x_k, z_k analog):

Die Menge aller bisher festgelegten Fanglinien (in y-Richtung) ist

$Y_F[i]$ mit i=1, ... n

Falls $|y_k - Y_F[i]| <= \text{Fangweite}$ für i=1,...n

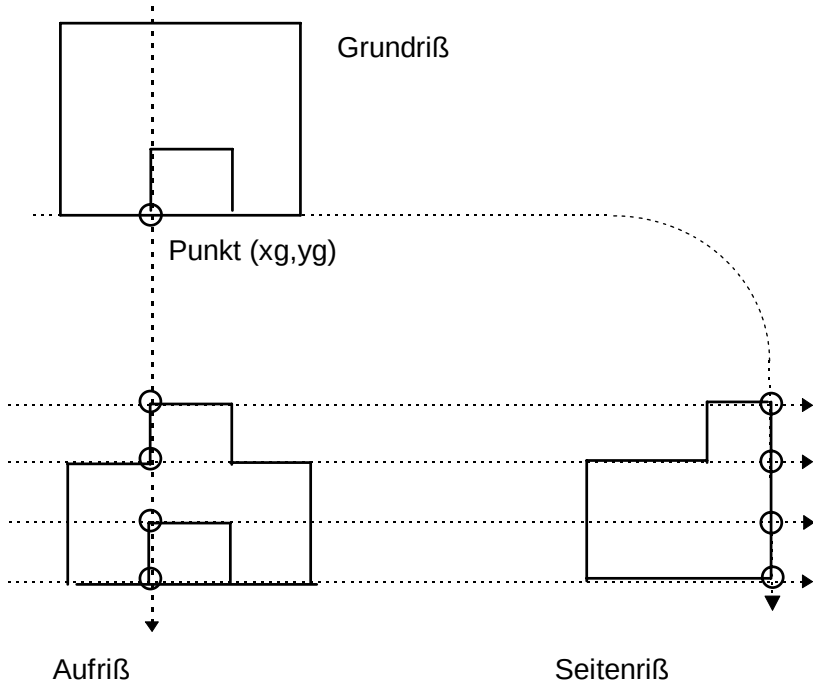
dann $y_k = Y_F[j]$ mit j = aktueller Index von i

sonst $Y_F[n+1] = y_k$;
n= n+1;

Ist der Abstand des Koordinatenwertes y_k zu einer Fanglinie aus der bisherigen Menge aller Fanglinien kleiner als die Fangweite, wird y_k auf die nächstgelegene Fanglinie gesetzt, d.h. y_k wird auf die Fanglinie gerastert. Andernfalls wird eine neue Fanglinie durch y_k gelegt.

B) Berechnung der räumliche Koordinaten

Es existiert ein räumlicher Punkt, wenn seine Projektion in allen drei Rissen als Eckpunkt vorhanden ist.



Für jedes Koordinatenpaar (x_g,y_g) des Grundrisses wird im Aufriß ein zugehöriges Paar

(x_a,z_a) mit $x_a = x_g$

gesucht. Falls das vorhanden ist, wird geprüft, ob ein

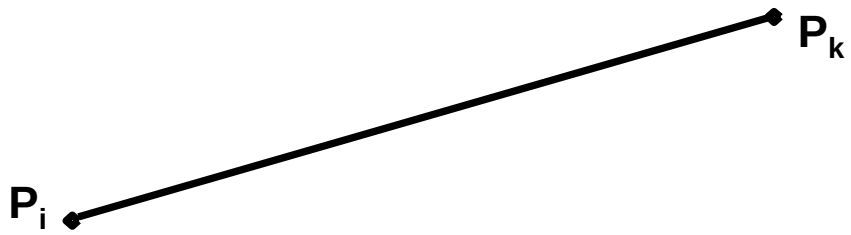
(y_s,z_s) mit $y_s = y_g$ und $z_s = z_a$

existiert.
Im positiven Fall wird ein Raumpunkt (x_g,y_a,z_s) generiert.

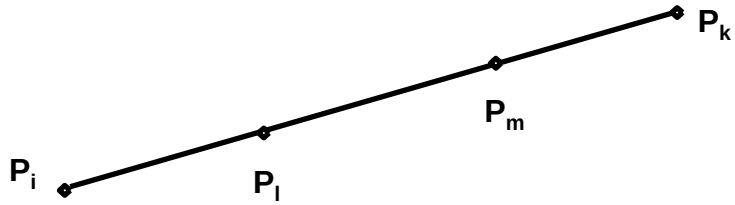
C) Ermittlung und Generierung der Kanten

Zwei Raumpunkte P_i und P_k sind verbunden, wenn sie in allen drei Rissen "quasi" verbunden sind.

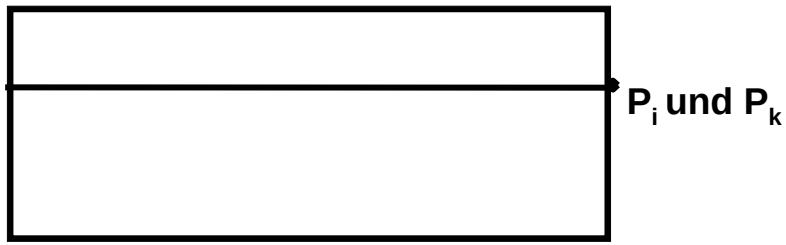
Das ist der Fall falls
•in der Projektion ist eine direkte Verbindung vorhanden ist,



•in der Projektion eine indirekte Verbindung vorhanden ist, d.h. auf der Verbindungslinie zwischen P_i und P_k liegen noch ein oder mehrere andere Punkte,

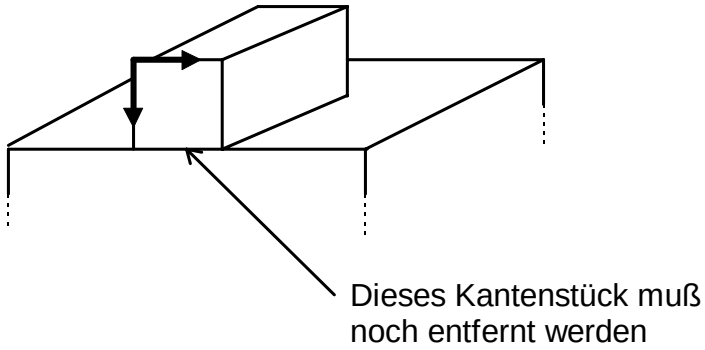


•die beiden Punkte P_i und P_k in der Projektion zusammenfallen.



Um später eine Bemessung (Parametrisierung) auswerten zu können, wird anschließend überprüft, ob jeder Maßpunkt eine Zuordnung zu einem Körper-Eckpunkt besitzt und ob jeder Körper-Eckpunkt je Koordinatenrichtung mindestens einen zugeordneten Maßpunkt besitzt. (Test auf Vollständigkeit der Bemaßung).

Bemerkung: Nicht jede ermittelte Kante muß tatsächlich eine Kante des Objektes sein.



D) Bestimmung der körperbildenden Ebenen

Aus der Menge der nun generierten Kanten werden jeweils zwei nichtparallele Kanten K_i und K_j mit einem gemeinsamen Eckpunkt gesucht.

Zwei Kanten sind dann parallel, wenn

$$\sin(\phi) = \frac{K_i \times K_j}{K_i * K_j} = 0$$

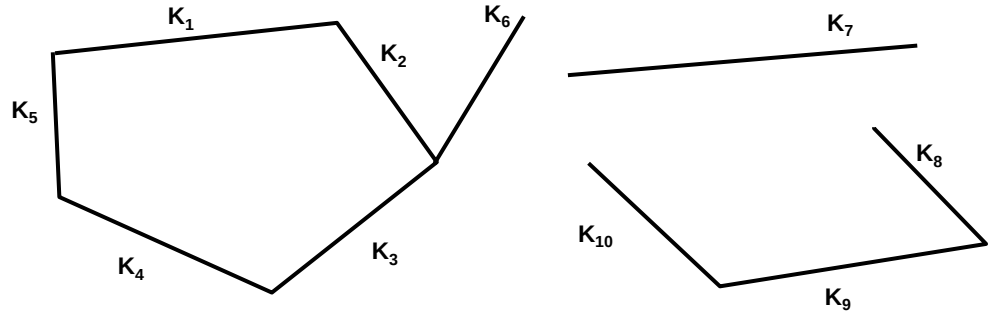
Gibt es zwei derartige Kanten (Vektoren) wird für die zwischen diesen aufgespannte Ebene die Hesse'sche Normalform (Ebenengleichung) berechnet (über Kreuzprodukt der beiden Vektoren).

$$(A*x + B*y + C*z + D) * \mu = 0$$

$$\text{mit } \mu = (+-)\ 1\ /(\ A^2 + B^2 + C^2)$$

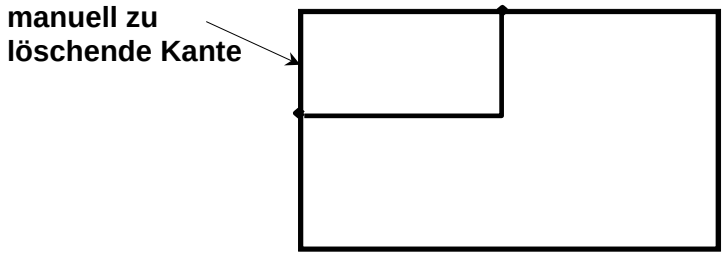
Für diese gefundene Ebene werden durch Einsetzen von Anfangs- und Endpunkte weiterer Kanten die Menge aller in dieser Ebene liegenden Kanten ermittelt. Betrachtet man nun die Menge aller Kanten in einer betrachteten Ebene, können folgende Fälle unterschieden werden:

- -geschlossene Polygonzüge
- -isolierte Kanten
- -"offene" Kanten
- -isolierte Kantenzüge



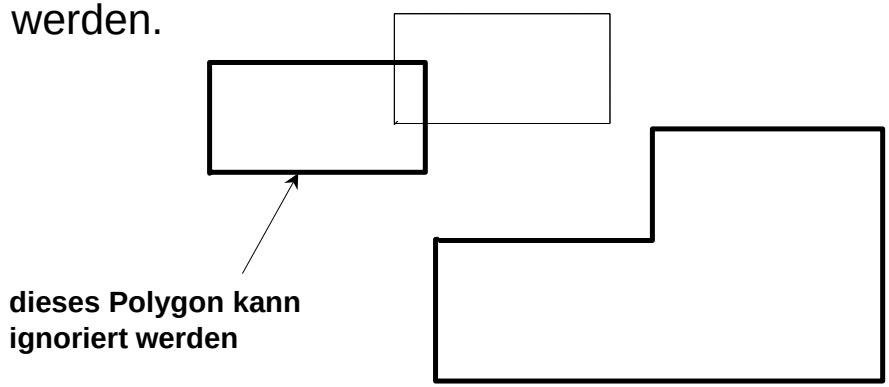
Isolierte Kanten bzw. Kantenzüge und offene Kanten sind für diese Ebene ohne Belang. Die jetzt verbleibenden (geschlossenen) Polygonzüge können aber noch immer "Pseudokanten"(-flächen) enthalten, deren Knoten dadurch gekennzeichnet sind, daß sie eine ungerade Anzahl von abgehenden Kanten besitzen. Diese Kanten müssen dann z.B. vom Benutzer per Hand gelöscht werden.

Meist muß hierbei interaktiv nur eine Kante gelöscht werden, da die weiteren dann aufgrund der obigen Regeln ohnehin eliminiert werden.



E) Löschen von "Pseudokanten"

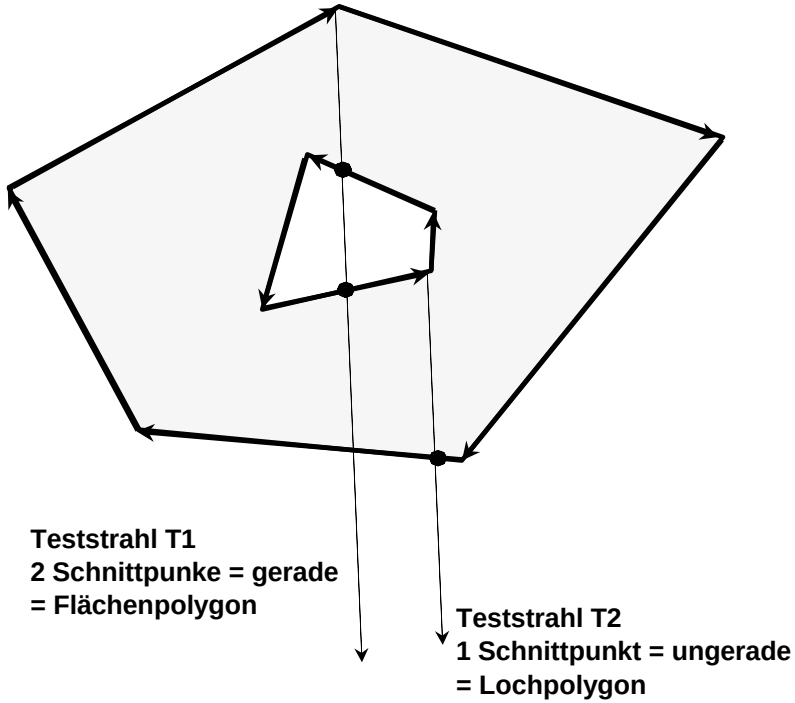
Bei genauerer Analyse dieser Situation läßt sich folgender automatisierte Weg einschlagen.
•Teilen der Kantenfigur an einem 3-Kanten-Knoten in zwei geschlossene Polygone,
•Suchen von kantengleichen Polygonen auf dazu parallele Ebenen,
•gibt es ein Vergleichspolygon auf einer dieser Ebenen, das in allen Kanten übereinstimmt, kann das betrachtete Teilpolygon ignoriert werden.

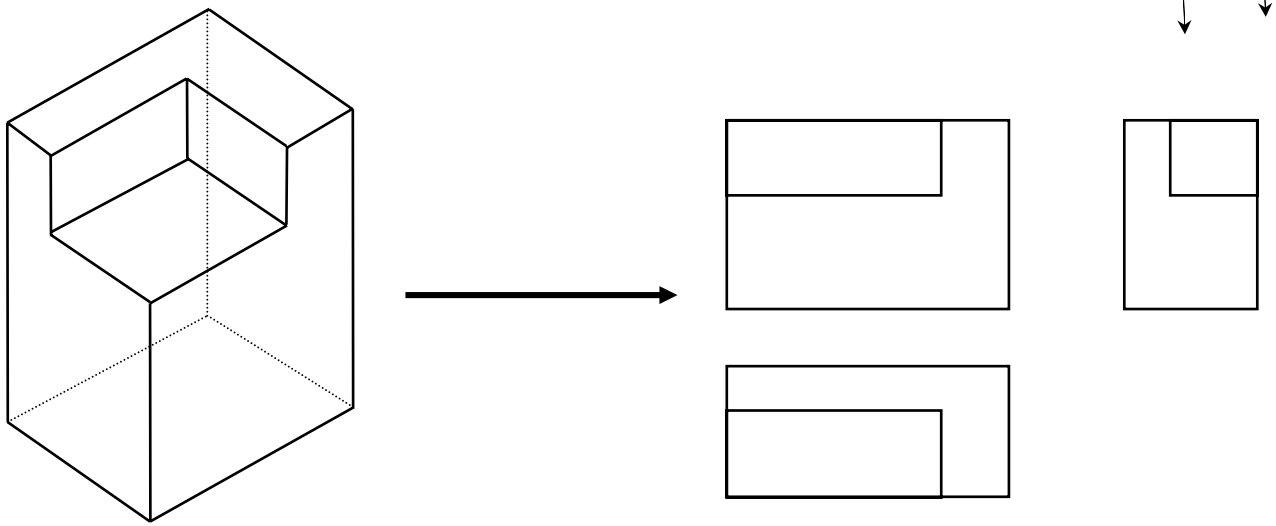


F) Ermitteln von Löchern und Orientierung der Kanten

Enthalten die verbleibenden Flächenpolygone ein oder mehrere eingeschlossene Polygone, handelt es sich um Löcher in den Flächen. Dieser Unterschied muß durch den Umlaufsinn der Kanten widergegeben werden.
Nach dem Gauß'schen Integralsatz sind rechtsdrehende Polygone äußere Flächenkanten, linksdrehende die Konturen von Löchern.
Zum Einsatz kommt hier die Gerade-Ungeraderegel, die die Schnittpunkte eines Strahles von einem beliebigen Knoten des betrachteten Polygons mit den anderen Polygonen dieser Ebene zählt und auswertet.

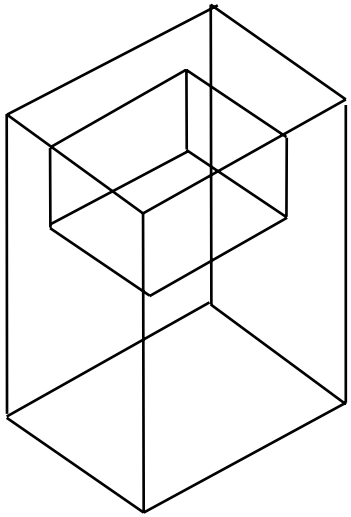
Die Gesamtheit der so ermittelten Flächen aller Ebenen bilden die Hülle des rekonstruierten Objektes (BRep-Modell) .



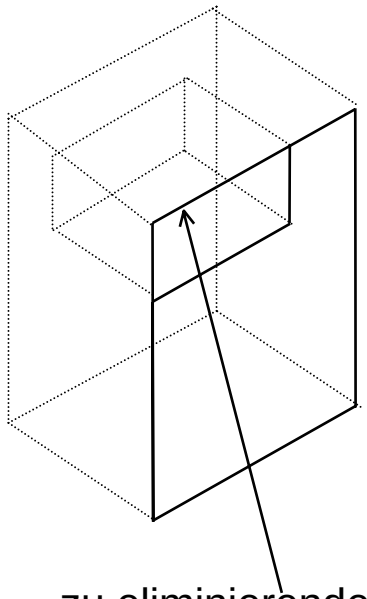


Körper

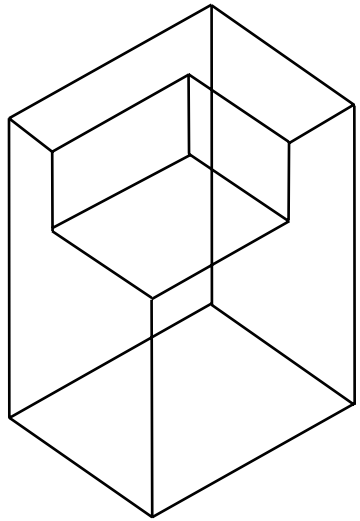
Drei-Tafel-Projektion



erstes Drahtmodell

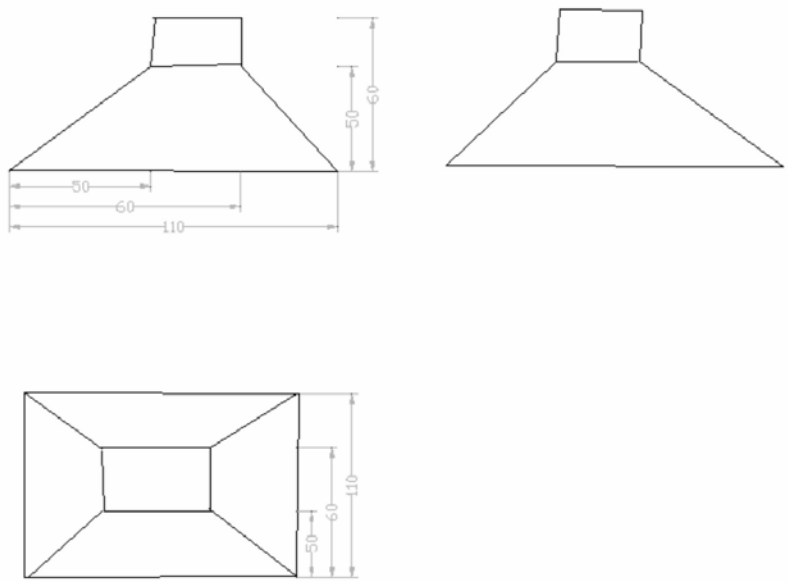


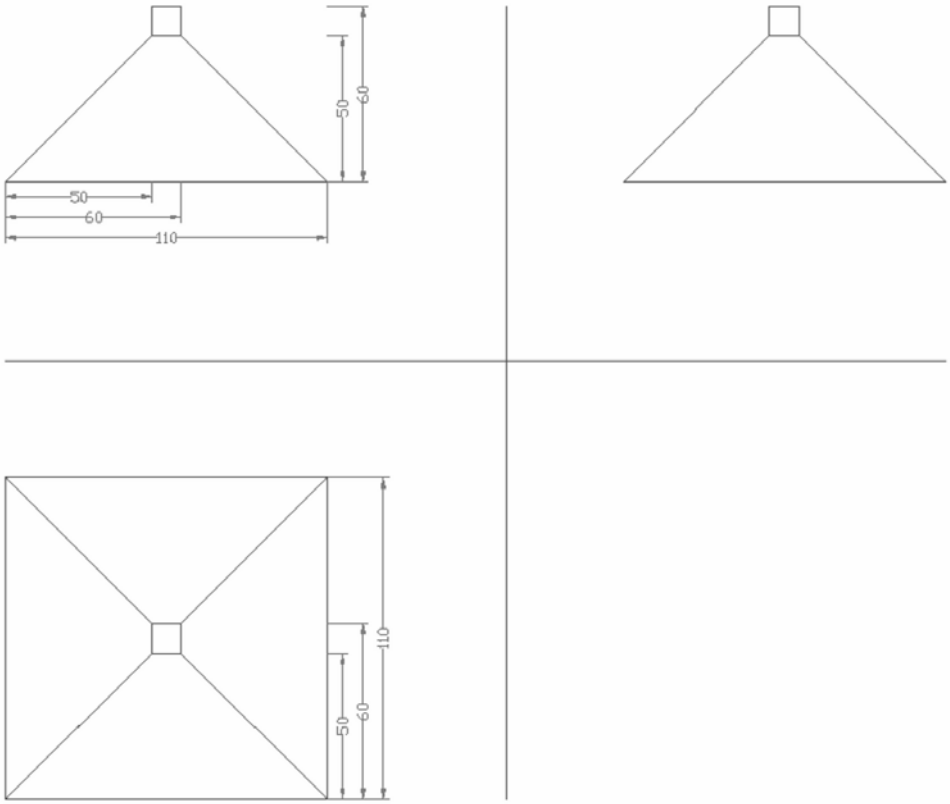
zu eliminierende Kante

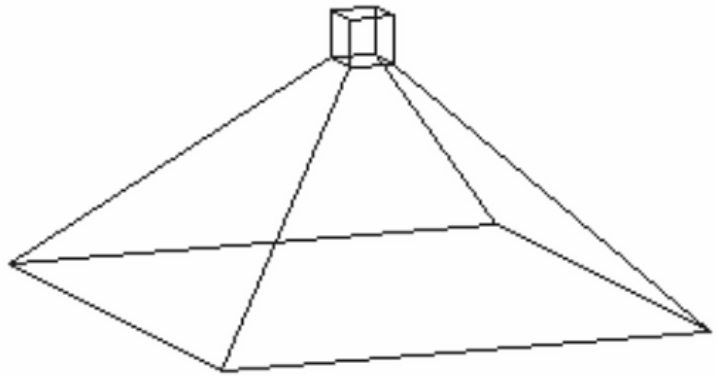


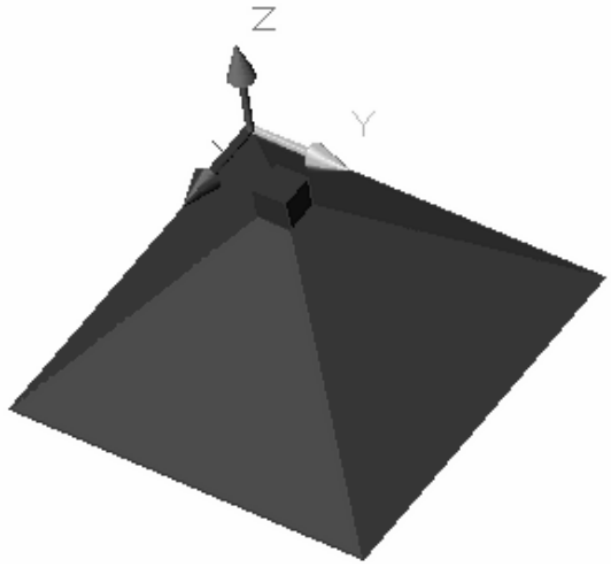
endgültiges Drahtmodell

Beispiel:




50
60
110
50
60
50
60
110




Z
Y

8. Parametrik und Constraint Management

8.1. Technischer Hintergrund parametrisierter Objekte

Im CAD-Umfeld wird häufig auf bereits vormodellierte Norm- bzw. Wiederholteile zurückgegriffen. Das entspricht einmal den wirklich praktizierten Vorgängen bei der Entwicklung eines neuen Gerätes, bei der aus Kostengründen möglichst viele Wiederholteile verwendet werden sollen. Zum anderen erspart dieser Rückbezug die mühevolle Neueingabe des Gestaltmodelles. Allein in der DIN werden für den Bereich Maschinenbau über 22 000 solcher Norm- und Wiederholteile beschrieben.

Die CAD-Sitzung besteht in einem solchen Fall also vorrangig aus rechnergestützten Montageprozessen. Dabei werden noch genau drei Informationsgruppen benötigt. Die sogenannten POD-Parameter (Plazierung, Orientierung und Dimensionierung) sind folgende:

- Einfügepunkt des Wiederholteils x,y,z (Lage des Referenzkoordinatensystems),
- Einfügerichtung bezüglich der drei Achsen x,y und z (Rotationswinkel um diese Achsen)
- und die aktuellen Abmessungen des Wiederholteils.

Während die beiden ersten Datengruppen durch typischen CAD-Input bestimmt werden (Cursorposition etc.), setzt die Eingabe von Abmessungsparametern die Existenz eines parametrisierten Modells voraus. Wie wichtig ein solches Modell ist, wird durch die Aussage belegt, daß oft über 100 Abmessungsvarianten eines Normteiles definiert sind. Es wäre unmöglich, hierfür separate CAD-Instanzen zu verwalten.

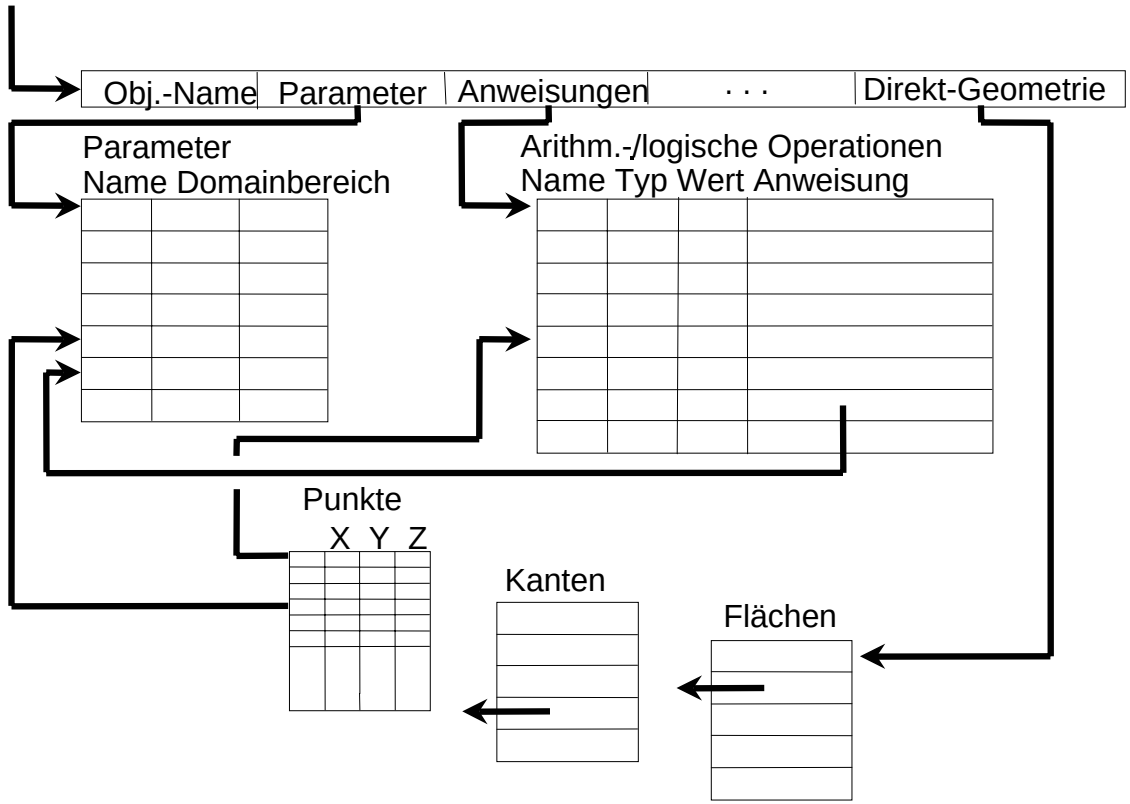
Ein zweiter Ansatzunkt ist die Tatsache, daß die Gestalt eines Objektes nach außen oft nur durch wenige Parameter modifizierbar ist. Alle weiteren inneren Maße entstehen algorithmisch aus den bestimmenden Hauptparametern. Eine Sechskantschraube wird beispielsweise allein durch Durchmesser und Länge bestimmt. Für die Konstruktion des Sechskantkopfes werden aber weitaus mehr Daten benötigt, die alle in einem direktem algorithmischen Zusammenhang zu den Hauptparametern stehen,

Schließlich muß ein dritter Gesichtspunkt betrachtet werden. Während der Erstkonstruktion eines Objektes, das einmal parametrisiert benutzt werden soll, werden geometrische Konstruktionen verwendet, deren Wirksamkeit auch bei beliebiger Wertbelegung der Parameter garantiert werden soll. Wenn ein Punkt als Tangentialpunkt einer Geraden an einen Kreis entstanden ist, muß er diese Eigenschaft auch bei verändertem Kreisradius oder Mittelpunkt behalten. Diese Art der Parametrisierung bezeichnet man als **Constraint Management**. Diese konstruktiven Relationen sind entweder als "Protokoll" der Erstkonstruktion oder - bei intelligenterer Herangehensweise - durch nachträgliche Analyse der Topologie zu gewinnen. Sie werden mit dem Modell oder auch anstelle des Modells archiviert.

Frage: Welchen Zusammenhang sehen Sie zwischen dem Constraint Management und der CSG-Modellierung ? Wie wird ein CSG-Objekt gespeichert ?

8.2. Rechnerinterne Darstellung von Parametrics und Constraints

Die Abbildung zeigt einen möglichen Ansatz zur Erweiterung eines BRep-Modells um den parametrisierten Anteil:



Für eine sinnvolle praktische Nutzung parametrisierter Wiederholteile ist die Frage der allgemeinen Nutzbarkeit bereits (CAD-) modellierter Objekte zu klären. Da jedes CAD-System eine spezifische Speicher- und Beschreibungsstruktur hat, muß dafür erst eine neutrale Modellierebene für Parametrics definiert werden, die drei Grundaufgaben erfüllen muß:

- Die Mächtigkeit der Beschreibungssprache muß so gewählt werden, daß alle notwendigen Konstruktions- und Modellierungsaufgaben im 2D und 3D, grafik-, flächen- oder volumen-orientiert, notiert werden können.
- Die Beschreibungssprache muß leicht zu compilieren, besser noch zu interpretieren sein, um den Bau der Konverter zu spezifischen CAD-Systemen zu erleichtern. --> Hier würde sich eine Spracherweiterung auf der Basis einer bestehenden modularen Programmiersprache (wie z.B. C, C++ usw.) eignen.
- Das Vokabular der Beschreibungssprache muß die Möglichkeiten der Konstruktion parametrisierter Geometrie ermöglichen.

8.3. Externe Beschreibung und Katalogisierung von Variantenkonstruktionen

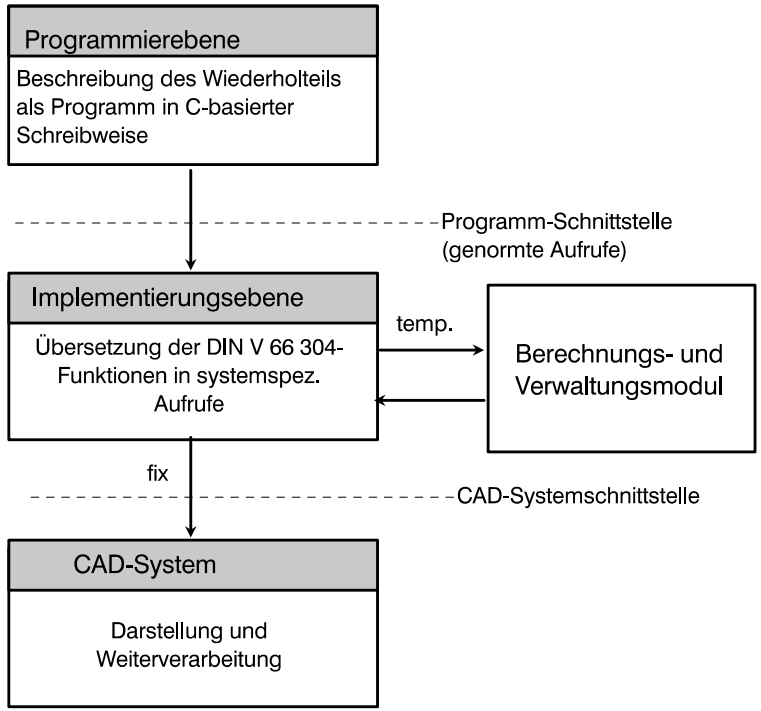
8.3.1. Struktur des Geometrieinterpreters

Wiederholteile werden auf der Programmierebene CAD-System-neutral beschrieben. Dabei steht den Entwicklern die gesamte Funktionalität der Basissprache (also z.B. C/C++) zur Verfügung. Damit lassen sich beispielsweise die Berechnungen von sekundären Abmessungen in (algorithmischer) Abhängigkeiten von den primären Parametern sehr einfach realisieren. Gleichzeitig können durch die verfügbare Unterprogrammtechnik die Modularisierung der Beschreibung oder die Erzeugung von Gestaltvarianten in Abhängigkeit von Parametern einfach und klar umgesetzt werden.

Die geometrischen Konstruktionen werden durch Aufrufe einer genormten Programmschnittstelle abgebildet.

Auf der Implementierungsebene wird dann das Programm ausgeführt und damit alle Konstruktionsaufgaben mit den konkreten aktuellen Parameterwerten gelöst.

Über eine CAD-Schnittstelle, die vom Leistungsvermögen des konkreten CAD-Systems bestimmt wird, werden dann die Gestaltinformationen in das rechnerinterne Modell des CAD-Systems übertragen.



Bemerkung:

Die Syntax der VDAPS(DIN 66304 V) wurde in der Diplomarbeit von Stefan Schwarz (2003) in eine C++-basierte Form transferiert, die im Folgenden Grundlage für die Erläuterungen ist.

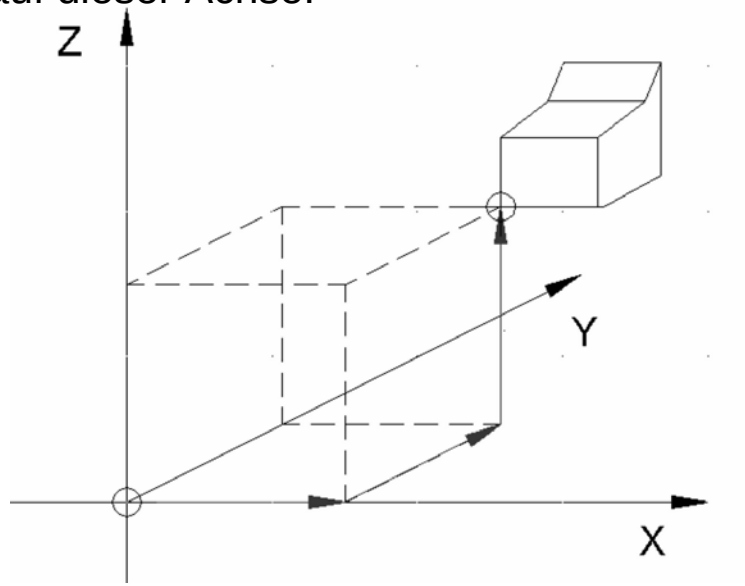
Koordinatensystem

Alle Geometrie-Definitionen beziehen sich auf ein dreidimensionales kartesisches Rechtssystem. Im Weiteren werden folgende Bezeichnungen für die Achsen verwendet:

x für die erste Achse und X für einen Vektor auf dieser Achse,

y für die zweite Achse und Y für einen Vektor auf dieser Achse,

z für die dritte Achse und Z für einen Vektor auf dieser Achse.



Vektor (Vector)

Alle in diesem System verwendeten Vektoren sind gebundene Vektoren und werden durch einen Anfangs- und einen Endpunkt definiert.

- **Standardinitialisierung:**

Ein Konstruktoraufruf ohne Parameter definiert einen Vektor von Punkt p_a = (0,0,0) zum Punkt p_e = (0,0,1).

Vector
pBgn : Point pEnd : Point
...

Beispiel:

```
Vector a;
Vector b = Vector();
```

- **Erzeugen eines Vektors aus zwei Punkten**

`Vector (Point p1, Point p2)`

- **`p1`** Anfangspunkt des Vektors
- **`p2`** Endpunkt des Vektors

Beispiel:

```
Point p1(0,0,0), p2(1,1,1);
Vector a(p1,p2);
Vector b = Vector(p2,p1);
```

- **Erzeugen eines Vektors aus einem Punkt**

`Vector (Point p2)`

-**`p2`** Endpunkt des Vektors (Ortsvektor zum Punkt p2)

Beispiel:

```
Point p2(1,1,1);
Vector a(p2);
```

- **Erzeugen eines Vektors aus einem Vektor**

`Vector (Vector v2)`

-**`v2`** Original-Vektor

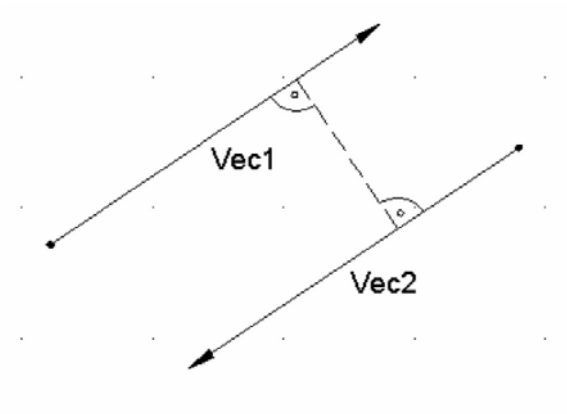
Beispiel:

```
Point p1(0,0,0), p2(1,1,1);
Vector a(p1, p2);
Vector b(a);
```

Durch Nutzen des überladenen "=" -Operators ist Kopieren auch so möglich:

```
b = a;
```

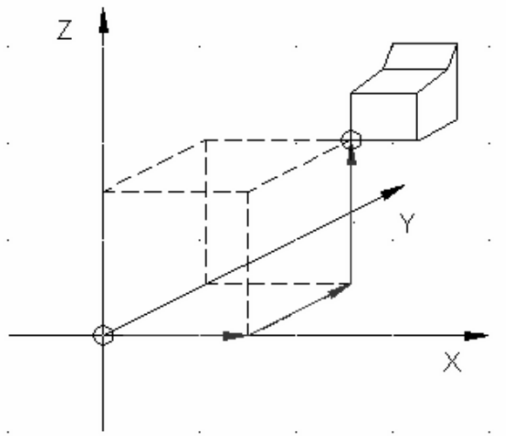
- **Ändern des Richtungssinns eines Vektors (invert)**
`invert()`



Beispiel:
```
Vector v(Point(1,0,1));
v.invert;
```

Lokales Koordinatensystem (Lcs)

Ein lokales Koordinatensystem wird durch drei Vektoren des Weltkoordinatensystems definiert.



-**Standardinitialisierung:**

Lcs()

Ein Konstruktoraufruf ohne Parameter definiert ein Lcs durch die drei Einheitsvektoren auf der x-, y- und z-Achse.

Beispiel:
```
Lcs lcs();
```

- **Lcs aus drei Punkten (Lcs)**

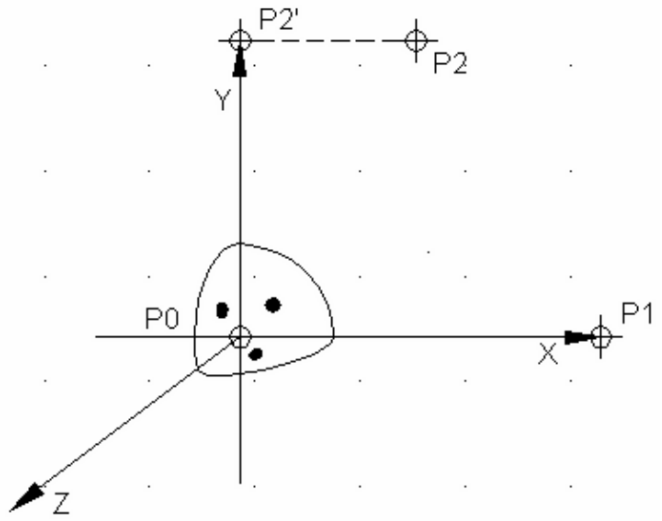
Erzeugen eines Lcs durch den Ursprung und zwei Punkten auf den ersten beiden Achsen. Die Achsen müssen rechtwinklig aufeinander stehen. Ansonsten wird das Lot vom dritten Punkt auf die Orthogonale zur Geraden erster Punkt -> zweiter Punkt gefällt und der dritte Punkt so modifiziert. Die drei Punkte dürfen nicht auf einer Geraden liegen.

`Lcs(Point origin, Point px, Point py)`

`-origin` Ursprung des lokalen Koordinatensystems

`-px` Punkt auf der x-Achse

`-py` Punkt auf der y-Achse

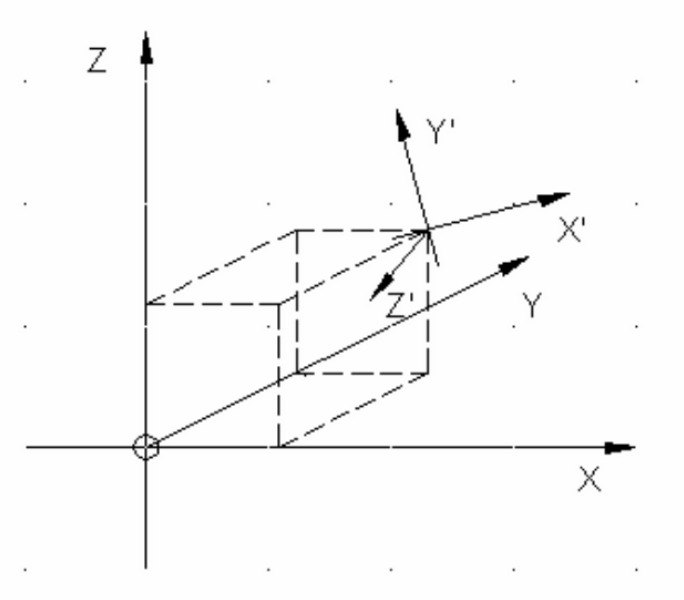


Beispiel:

```
Point p0(6,7,3), p1(8,9,4), p2(4,7,6);
Lcs lcs1(p0,p1,p2);
```

- **Setzen des aktuellen Lcs (setCurrentLcs)**

Das aktuelle Lcs ist das gültige Koordinatensystem für alle nachfolgenden Objekt-Erzeugungsoperationen.



`setCurrentLcs(Lcs lcs)`

- **`lcs`** das zu setzende Lcs

Beispiel:

```
Point p0(6,7,3), p1(8,9,4), p2(4,7,6);
Lcs lcs1(p0,p1,p2);
setCurrentLcs(lcs1);
```

Referenzpunkt (refPoint)

Der Referenzpunkt bezeichnet den Bezugspunkt im CAD-System. An diesem Punkt erfolgt die Plazierung des (lokalen) Weltkkoordinatensytems.

-**Setzen des Referenzpunktes (setRefPoint)**

`setRefPoint (Point p)`

- **`p`** Plazierungspunkt

Beispiel:

```
setRefPoint(Point(2,4,6));
```

Ausgabe in das CAD-System (draw)

Für alle Konstruktionselemente gibt es eine Methode zur Ausgabe in das CAD-System. Nach der Ausgabe kann mit den Elementen weiter gearbeitet werden.

```
draw ()
```

Beispiel:

```
Point p1;
p1.draw();
```

Rechnen mit Elementen

Für alle Elemente sind die Operatoren "+" und "-" überladen und verschieben das Element um den Ortvektor des angegebenen Punktes.

Beispiel:

```
Point p1(1,2,3);
Line l;
l = l+p1;
```

Matrixtransformationen (transform)

Transformation eines Elementes mit einer 4*4 Matrix..

`transform (Tmatrix tm, Element e)`

- **`tm`** Transformationsmatrix
- **`e`** zu transformierendes Element

Beispiel:

```
real val[4][4] = {{11,12,13,14},{21,22,23,24},{31,32,33,34},
                  {41,42,43,44}};
TMatrix tm (val);
Line l;
transform (tm, l);
```

Rechnen mit Matrizen

Für alle Elemente ist der Operator " * " überladen und transformiert das Element entsprechend der angegebenen Matrix.

Beispiel:

```
TMatrix tm (val);
Line l, l1;
l1 = tm *l;
```

Kopieren von Elementen

Für alle Elemente ist der Zuweisungsoperator " = " überladen und kopiert das Element einschließlich seines Inhaltes. Bei Gruppen und Polygonen werden alle enthaltenen Elemente kopiert.

Beispiel:

```
Point p1(1,2,3);
Point p2 = p1;
```

Spiegeln von Elementen (mirror)

- Spiegelung an einer Geraden

Spiegel eines Elementes an einer Geraden, d
angegeben wird.

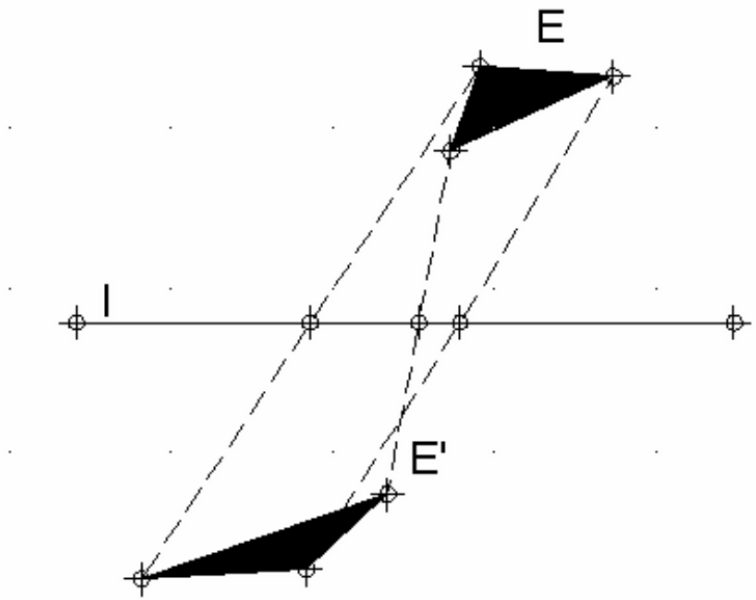
```
mirror (Line l, Element e)
```

`-l` Spiegelungsachse

`-e` zu spiegelndes Element

Beispiel:

```
Point p1(0,0,0), p2(0,1,0), p3(1,0
Line l(p1,p2);
mirror (l, p3);
```

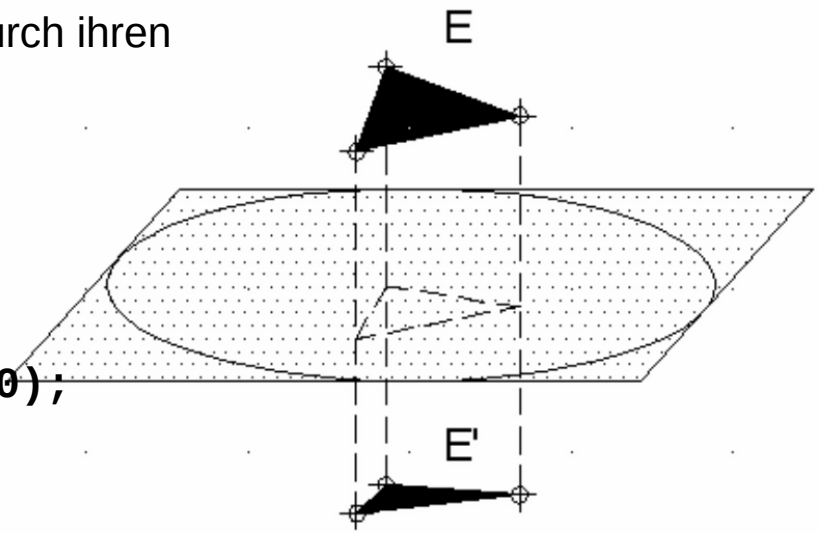


- Spiegelung an einer durch eine Vektor definierten Ebene

Spiegel eines Elementes an einer Ebene, die durch ihren Normalvektor definiert wird.

`mirror (Vector v, Element e)`

- **`-v`** Normalvektor der Spiegelebene
- **`-e`** zu spiegelndes Element



Beispiel:

```
Point p1(0,0,0), p2(0,1,0), p3(1,0,0);
Vector v(p1, p2);
mirror (v, p3);
```

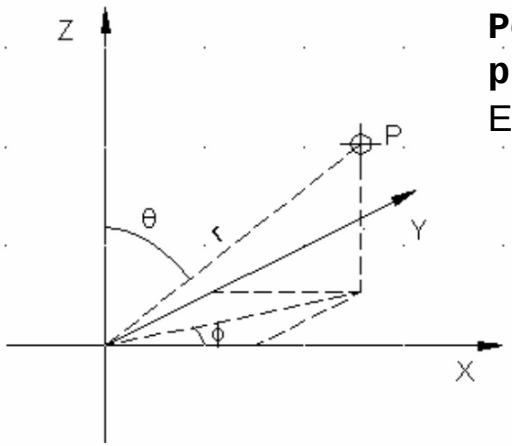
- Spiegelung an einer durch Elemente definierten Ebene

Spiegel eines Elementes an einer Ebene, die durch drei Punkte oder die Elementebenen von Kreis bzw. Kreisbogen definiert wird.

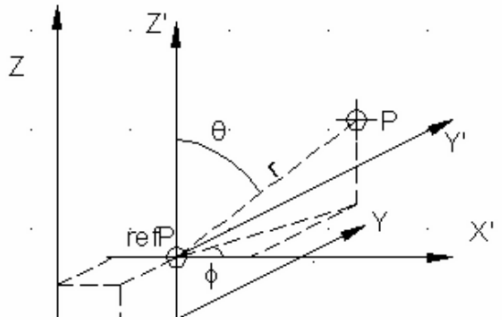
`mirror (Point p1, Point p2, Point p3, Element e)`

`mirror (Circle c, Element e)`

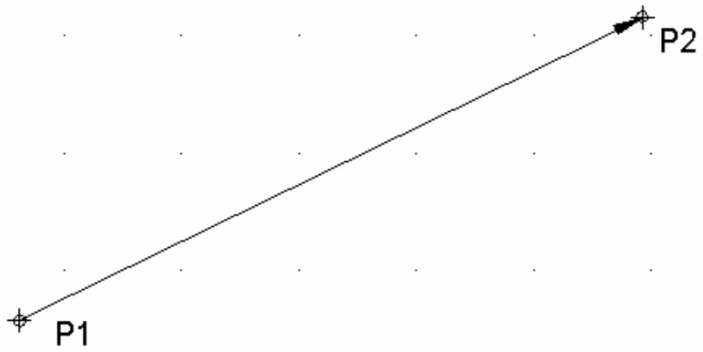
- **`-p1,p2,p3`** Punkte in der Spiegelebene
- **`-c`** Kreis in der Spiegelebene
- **`-e`** zu spiegelndes Element



`Point::Point(bool arcType, cadDouble radius,cadDouble phi, cadDouble psi)`
Erzeugt einen Punkt mit Polarkordinaten, arcType definiert Winkeleinheit

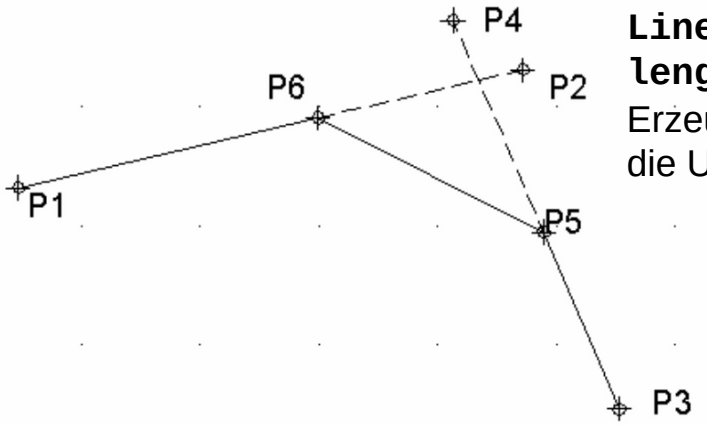


`Point::Point(Point &refPoint, bool arcType, cadDouble radius,cadDouble phi, cadDouble psi)`
Erzeugt einen Punkt mit Polarkordinaten von einem Referenzpunkt, arcType definiert Winkeleinheit



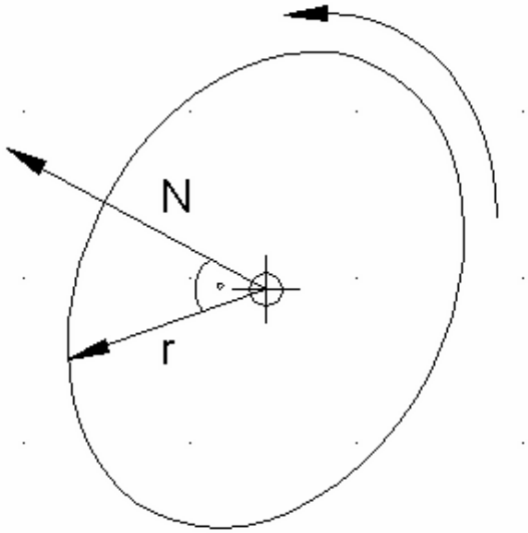
`Line::Line(Point &p1, Point &p2)`
Erzeugt eine Strecke von P_1 nach P_2

`Line::Line(Point &p, cadDouble length, Vector &direction)`
Erzeugt eine Strecke von P mit Länge und Richtung



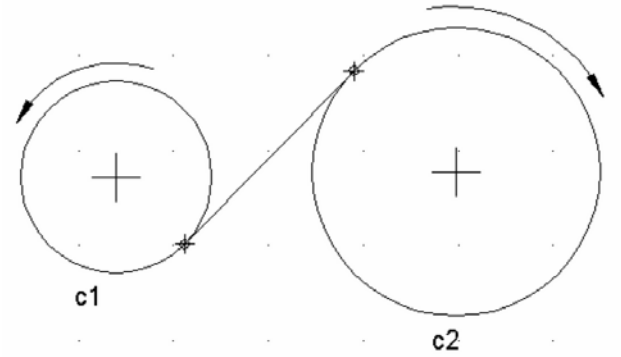
`Line &Line::chamfer(Line &l1, Line &l2, cadDouble length1, cadDouble length2, bool fillet)`

Erzeugt eine Phase zwischen den Linien l_1 und l_2, ist fillet = true werden die Ursprungsstrecken gekürzt

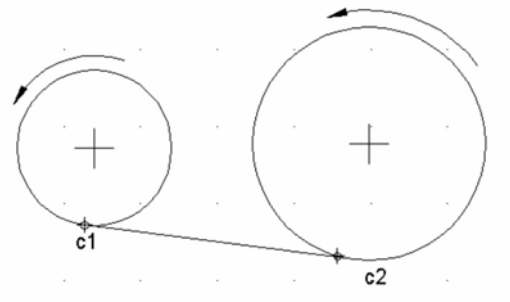


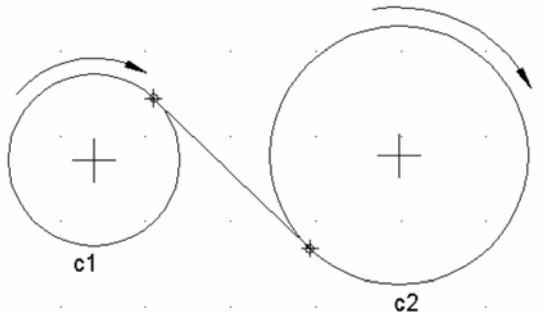
`Circle::Circle(cadDouble radius,Vector &vector)`

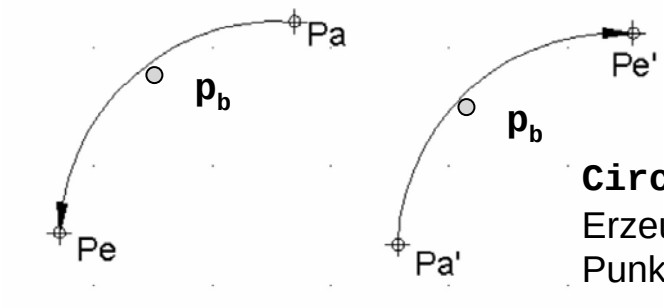
Erzeugt einen Vollkreis mit dem Radius und dem Normalenvektor des Kreises



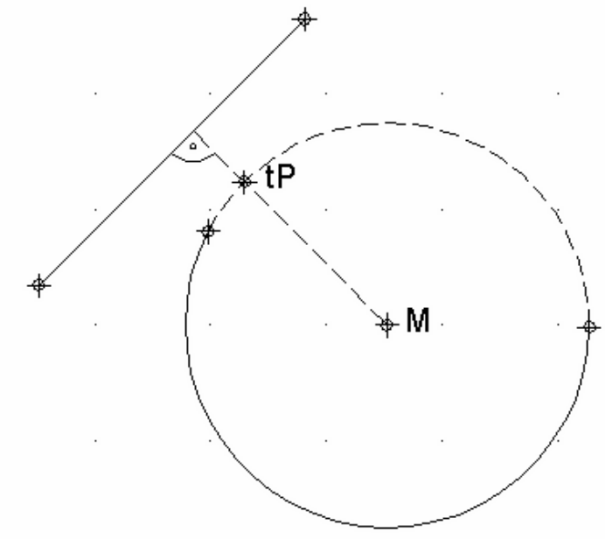
`bool Circle::tangent(Circle &c1, Line &retL)`
Erzeugt die Tangentenstrecke zwischen zwei Kreisen



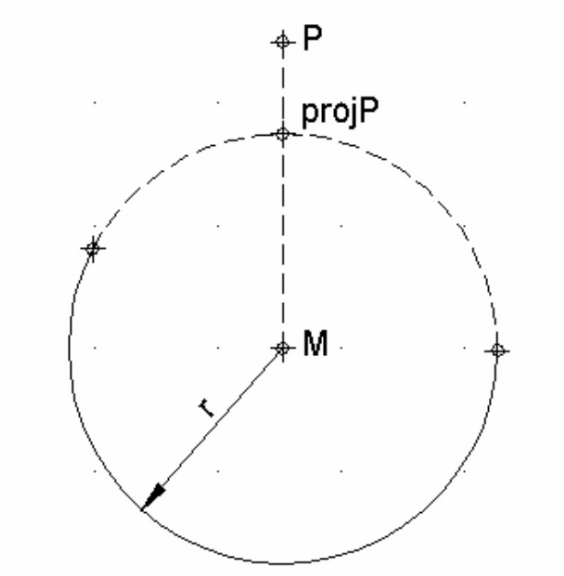




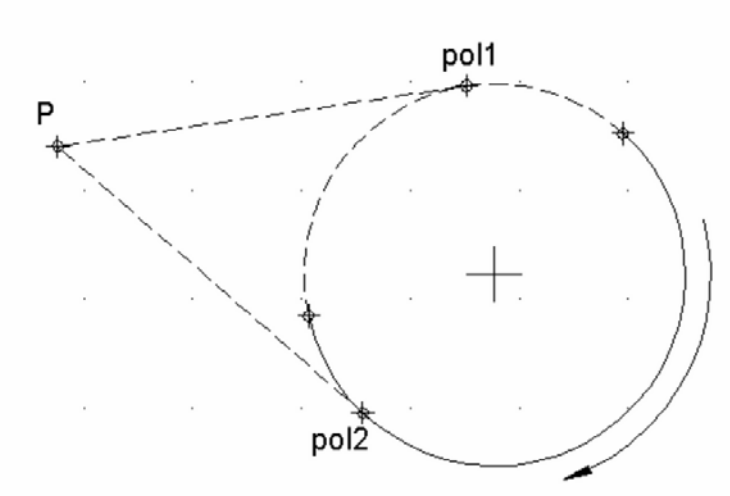
`CircleArc::CircleArc(Point &pa, Point &pe, Point &pb)`
Erzeugt einen Kreisbogen zwischen P_a und P_e, P_b ist ein beliebiger Punkt auf dem Bogen



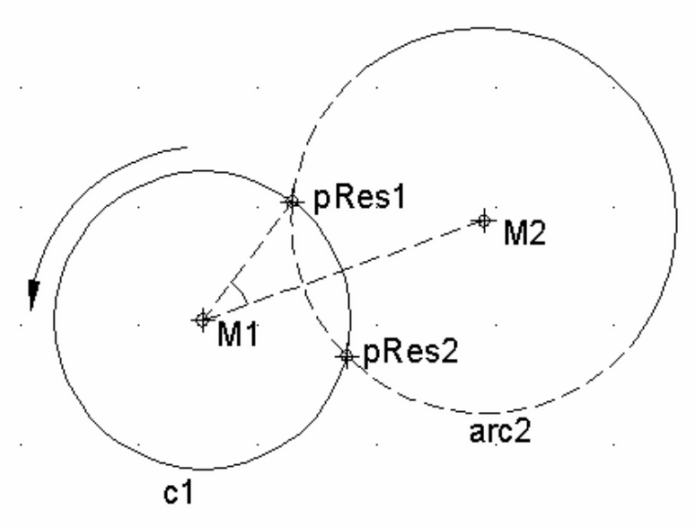
`bool Circle::tangent(Line &l1,Point &tpRet)`
Erzeugt den Tangentenpunkt der Linie l_1 an den Kreis



`bool Circle::projection(Point &p,Point &projPRet)`
Erzeugt den Schnittpunkt der Geraden P -> M mit dem Kreis

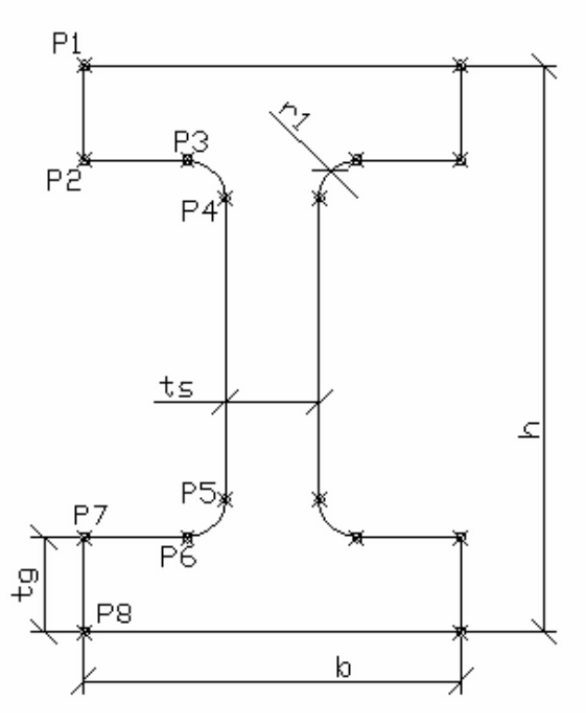


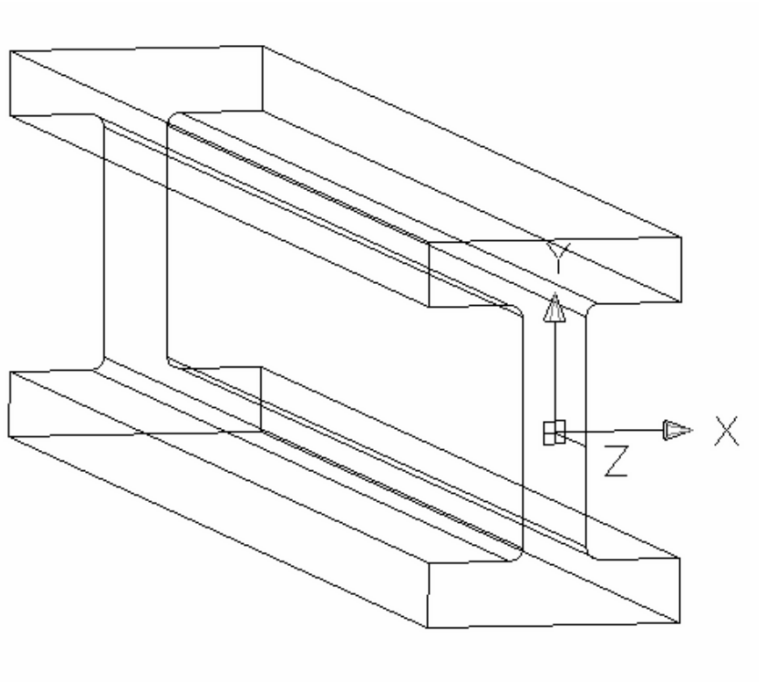
`bool Circle::pol(Point &p,Point &polRet)`
Erzeugt den Polarpunkt von einem Punkt P an den Kreis



`bool Circle::intersect(Circle &c1, Point &p1Ret,Point &p2Ret)`
Berechnet die Schnittpunke zweier Kreise, p_1 ist der Schnittpunkt mit dem kleineren Winkel

8.4. Beispiel einer Variantenkonstruktion – Doppel-T-Träger





```
//using namespace paramObj;
name ("T_Profil", "T-Profil nach DIN 2025 Teil 2");
Parameter ( "Breite" , "double" , "40" , "Breite des T-Traegers");
Parameter ( "Hoehe" , "double" , "60" , "Hoehe des T-Traegers" );
Parameter ( "Stegbreite" , "double" , "10" , "Stegbreite des T-Traegers");
Parameter ( "Gurtbreite" , "double" , "10" , "Gurtbreite des T-Traegers");
Parameter ( "Ausrundung" , "double" , "2" , "Ausrundung des T-Traegers");
Parameter ( "Laenge" , "double" , "300" , "Laenge des T-Traegers");
Group TeilSeite(cadDouble dz=0)
{
   cadDouble b,h,ts,tg,r1;
   getParam ( "Breite" , b);
   getParam ( "Hoehe" , h);
   getParam ( "Stegbreite" , ts);
   getParam ( "Gurtbreite" , tg);
   getParam ( "Ausrundung" , r1);
   cadDouble dx1,dx2,dx3,dy1,dy2,dy3;
   dx1=b/2;
   dx2=ts/2;
   dx3=dx2+r1;
   dy1=h/2;
   dy2=dy1-tg;
   dy3=dy2-r1;
   Point p1((-1)*dx1,dy1,dz),
         p2((-1)*dx1,dy2,dz),
         p3((-1)*dx3,dy2,dz),
         p4((-1)*dx2,dy3,dz),
         p5((-1)*dx2,(-1)*dy3,dz),
         p6((-1)*dx3,(-1)*dy2,dz),
         p7((-1)*dx1,(-1)*dy2,dz),
         p8((-1)*dx1,(-1)*dy1,dz);
   Line l1(p1,p2);
   Line l2(p2,p3);
   Line l3(p4,p5);
```

```
   Arc a1;
   a1.fillet(r1,l2,l3,a1);
   Line l4(p6,p7);
   Arc a2;
   a1.fillet(r1,l3,l4,a2);
   Line l5(p7,p8);
   Group grp;
   grp.addElement(l1);
   grp.addElement(l2);
   grp.addElement(l3);
   grp.addElement(a1);
   grp.addElement(l4);
   grp.addElement(a2);
   grp.addElement(l5);
   return grp;
};

void mainf()
{
   cadDouble b,h,ts,tg,r1,lae;
   getParam ( "Hoehe" , h);
   getParam ( "Laenge" , lae);
   getParam ( "Breite" , b);
   getParam ( "Hoehe" , h);
   getParam ( "Stegbreite" , ts);
   getParam ( "Gurtbreite" , tg);
   getParam ( "Ausrundung" , r1);
   Group links=TeilSeite(0);
   Line l(Point (0,0,0),Point(0,h,0));
   Group rechts=links;
   mirror(l,rechts);
   Group vorne;
   vorne.addElement(rechts);
   vorne.addElement(links);
   Point p1((-1)*b/2,h/2,0);
```

```
   Point p2(b/2,h/2,0);
   Point p3((-1)*b/2,(-1)*h/2,0);
   Point p4(b/2,(-1)*h/2,0);
   Line l1(p1,p2);
   Line l2(p3,p4);
   vorne.addElement(l1);
   vorne.addElement(l2);
   Group hinten(vorne);
   Point p(0,0,(-1)*lae);
   move (p,hinten);
   Group alle;
   alle.addElement(vorne);
   alle.addElement(hinten);
   Group seite;

   Point lBgn((-1)*b/2,h/2,0);
   Point lEnd=lBgn;
   lEnd[2]-=lae;
   l=Line(lBgn,lEnd);
   seite.addElement(l);

   lBgn[1]-=tg;
   lEnd=lBgn;
   lEnd[2]-=lae;
   l=Line(lBgn,lEnd);
   seite.addElement(l);

   lBgn[0]+=b/2-r1-ts/2;
   lEnd=lBgn;
   lEnd[2]-=lae;
   l=Line(lBgn,lEnd);
   seite.addElement(l);
```

```
    lBgn[0]+=r1;
    lBgn[1]-=r1;
    lEnd=lBgn;
    lEnd[2]-=lae;
    l=Line(lBgn,lEnd);
    seite.addElement(l);
    Group seiten;
    seiten.addElement(seite);

    mirror (Point (0,0,0),Point(1,0,0),Point(0,0,1),seite);
    seiten.addElement(seite);
    alle.addElement(seiten);

    cadDouble mval[4][4]={{-1,0,0,0},
                          {0,1,0,0},
                          {0,0,1,0},
                          {0,0,0,1}};
    TMatrix tm(mval);
    transform (tm,seiten);
    alle.addElement(seiten);
    alle.draw();
}
```

9. Objektorientierte Modellierung von Geometrie und Topologie

Ausgangssituation:

Die Aufgabe besteht in der Konstruktion eines "Speichers" (sowohl temporär = C oder C++ - Strukturen, als auch permanent = Definition einer relationalen bzw. objektorientierten Datenbank) für Gestaltinformationen.

Der semantische Inhalt des Speichers soll allgemein definiert werden und damit leicht änderbar und erweiterbar sein.

Die mathematisch/geometrischen Basiskenntnisse sind zu berücksichtigen.

Methodik:

Die Definition des Modells soll automatische übersetzbar sein.

Benötigte Werkzeuge:

eine Beschreibungsform/ -sprache, die allgemein gültig und zeitlich stabil ist und für die Compiler existieren,

eine Geometrie/Topologie - Beschreibung, die den inhaltlichen Anforderungen an den Speicher entsprechen.

Diese Forderungen erfüllt die Sprache **EXPRESS** bzw. deren grafisches Äquivalent EXPRESS – G.

Die Gestaltmodellierung findet man in den Resourcenparts von STEP (ISO 10303) (Standard for Exchance of Product Data).

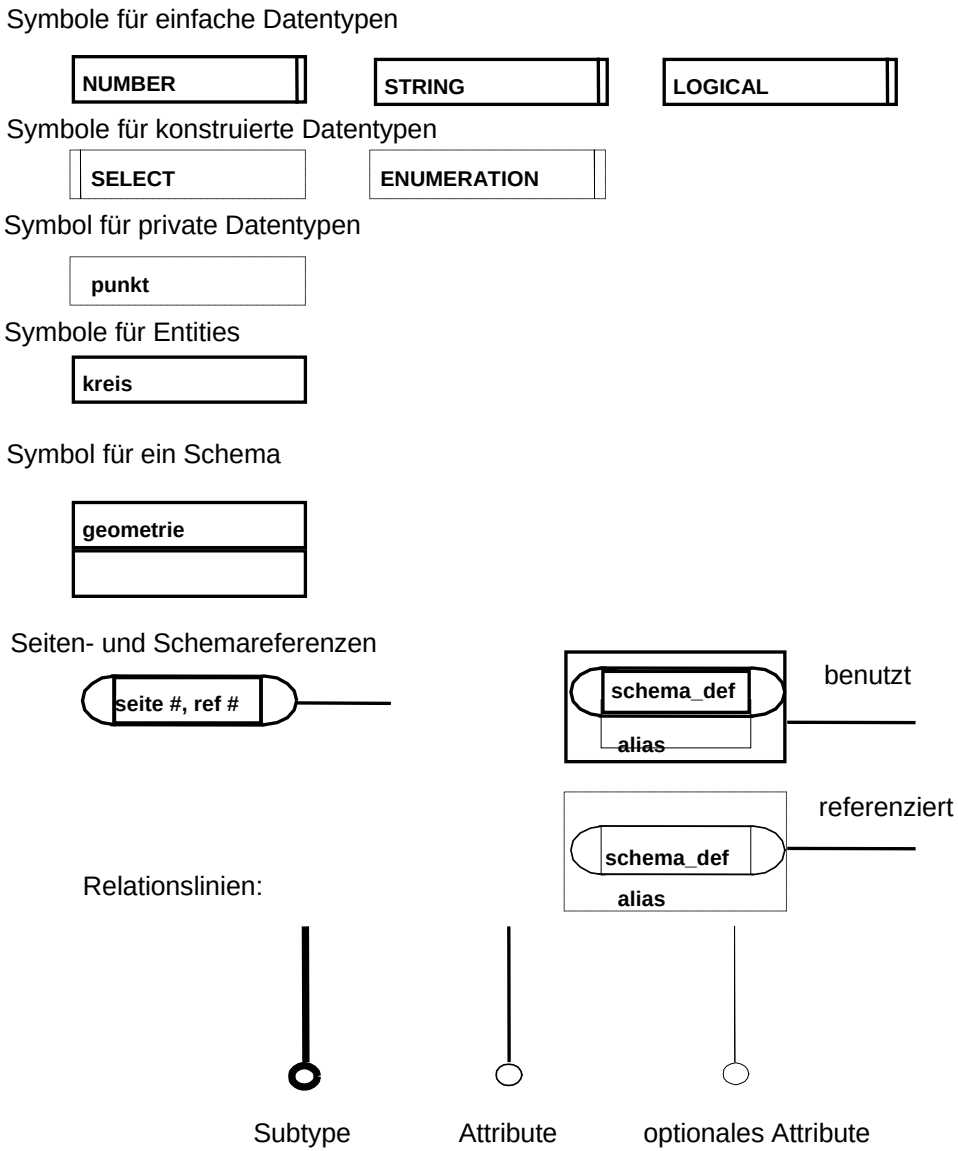
Zur Information : Der Standard STEP

STEP steht für "Standard for the Exchange of Product Model Data". Im Rahmen der rechnerintegrierten Produktion spielt der Austausch dieser produktdefinierten Daten eine zunehmende Rolle. Diese Daten werden heute überwiegend in nicht kompatiblen Formaten und Strukturen von CAD/CAM-Systemen erzeugt und verarbeitet. Die Integration dieser Daten ist Voraussetzung für Simultaneous Engineering und eine vollständige Datendurch-gängigkeit in den Unternehmen. Die zur Realisierung eines effizienten Datenaustausches notwendiger Vereinbarungen (Schnittstellen) zwischen verschiedenen Systemen müssen diesem Anspruch gerecht werden. Der Produkdatenaustausch ist sowohl firmenintern als auch -extern von Bedeutung. Interner Produktdatenaustausch zielt auf die informations-technische Verbindung der verschiedenen Unternehmensbereiche, wie Konstruktion, Fertigung, Montage und Qualitätssicherung. Der externe Produktdaten-austausch dient der Übertragung von Produktinformation zwischen Hersteller und Zulieferer.

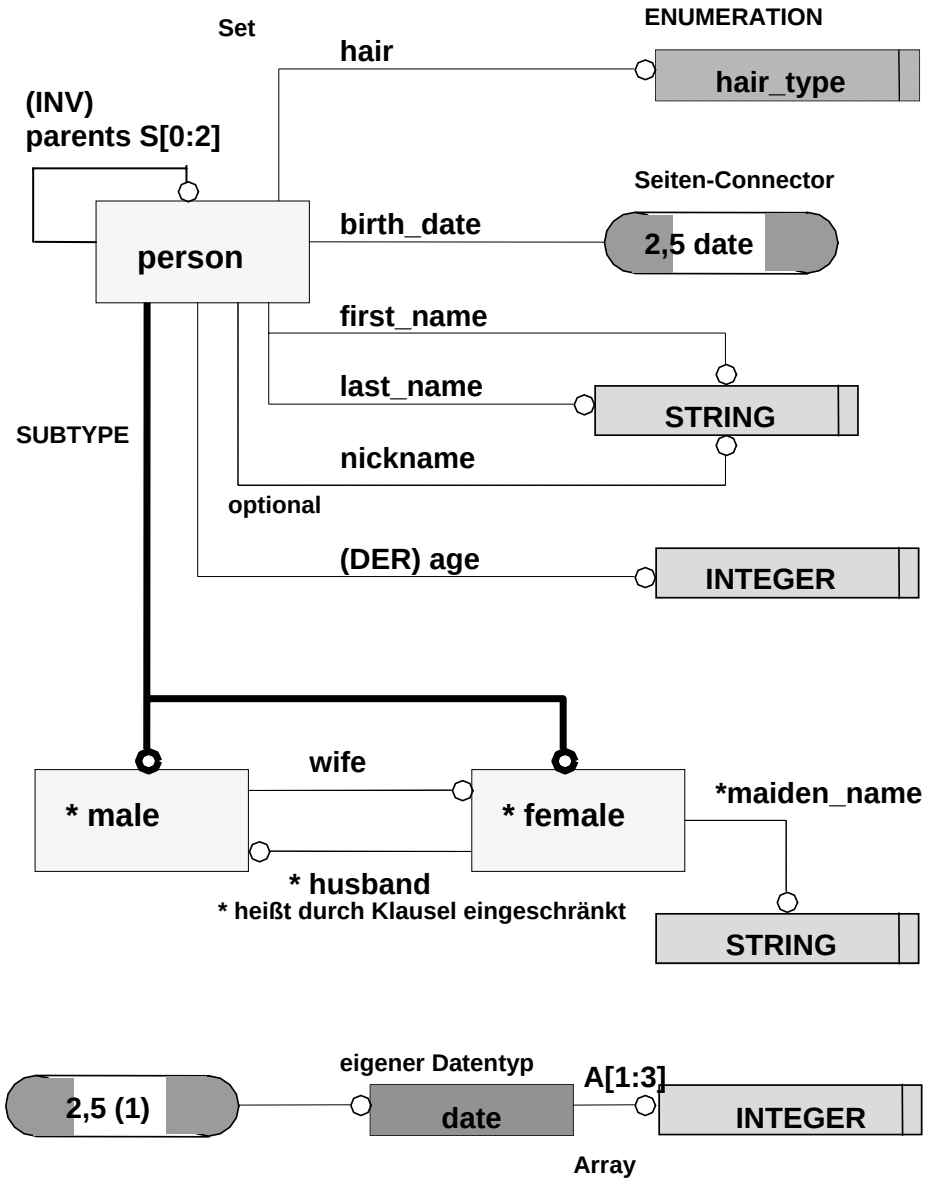
Die bisher industriell angewandten Schnittstellen wie z.B. IGES, VDAPS oder SET werden den Anspruch einer Schnittstelle zum Produkdatenaustausch nicht gerecht. Sie eignen sich zur Übertragung von Produktinformation in Teilbereichen der rechnerintegrierten Produktion in Form von technischen Zeichnungen oder einfachen Geometriemodellen. Darüber hinausgehende Information, wie z.B. Toleranzangaben, Materialeigenschaften, Stücklisten oder gar Arbeitsplanungsinformationen können nicht übertragen werden.

Zur konsistenten, widerspruchsfreien und semantisch eindeutigen Beschreibung des Produktmodells von STEP wurde die formale Beschreibungssprache EXPRESS und deren graphische Repräsentation EXPRESS-G definiert. EXPRESS ist keine Programmiersprache sondern eine Spezifikationssprache mit objektorientierten Eigenschaften zur formalen, eindeutigen Beschreibung des STEP Produktmodells in Form von sogenannten Informationsmodellen.

9.1. Grafische Modelldefinition - der grafische Subset EXPRESS-G



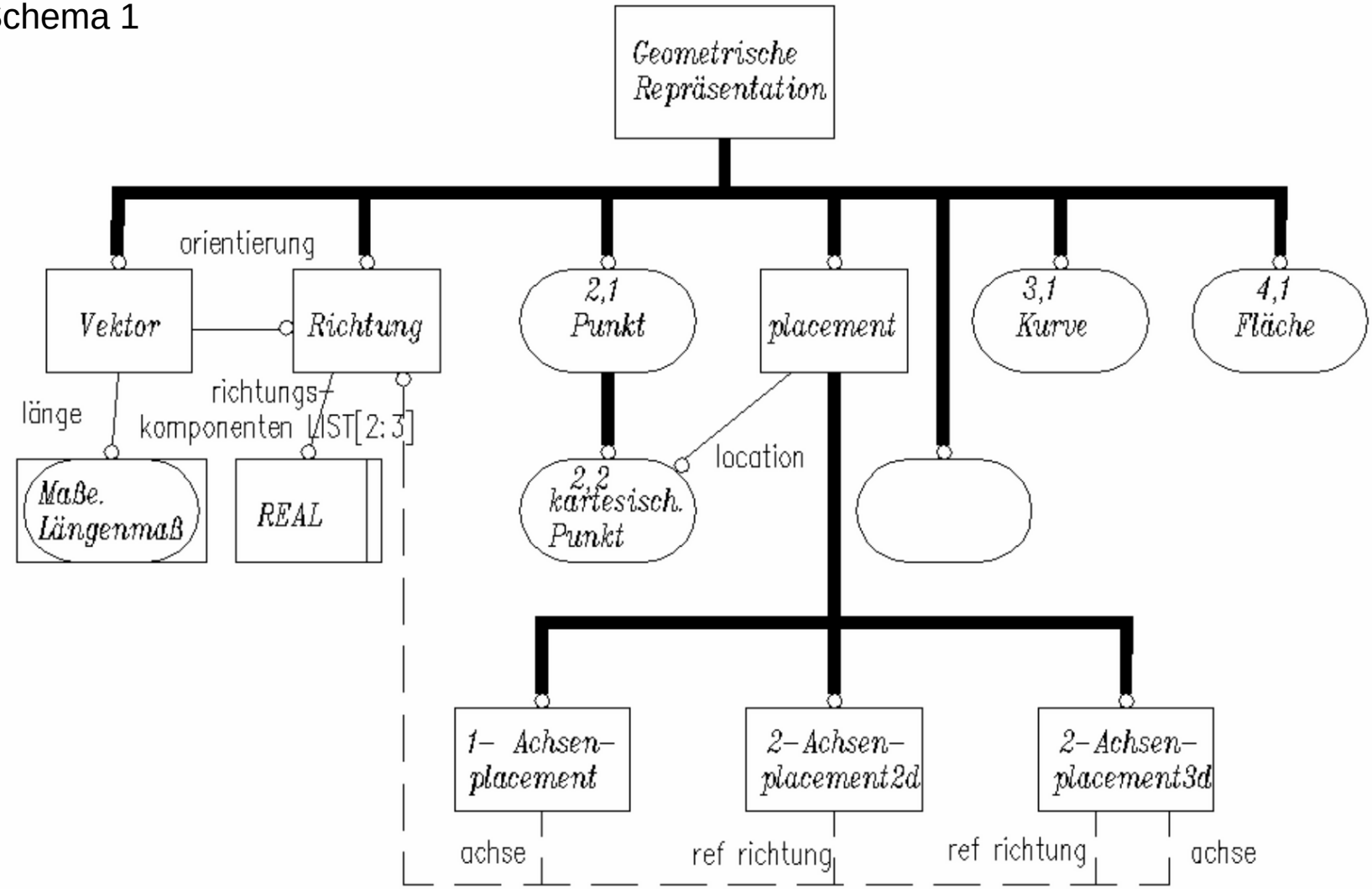
Symbolik in EXPRESS


Set
ENUMERATION
hair
hair_type
(INV)
parents S[0:2]
Seiten-Connector
person
birth_date
2,5 date
first_name
last_name
STRING
SUBTYPE
nickname
optional
(DER) age
INTEGER
wife
* male
* female
*maiden_name
* husband
* heißt durch Klausel eingeschränkt
STRING
eigener Datentyp
A[1:3]
2,5 (1)
date
INTEGER
Array

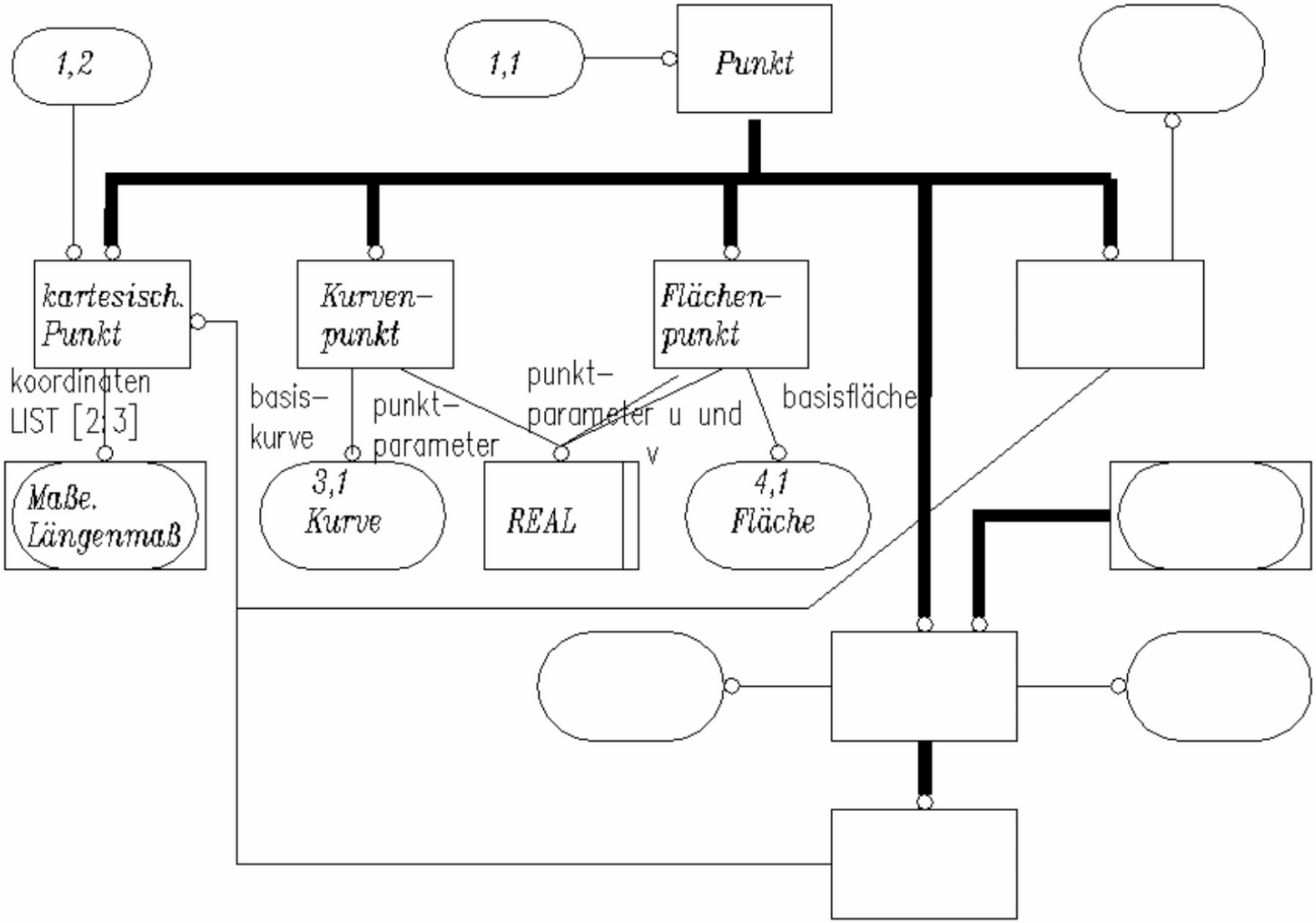
9.2. EXPRESS-basierte Geometrie und Topologie

9.2.1. Geometrie - Schemata

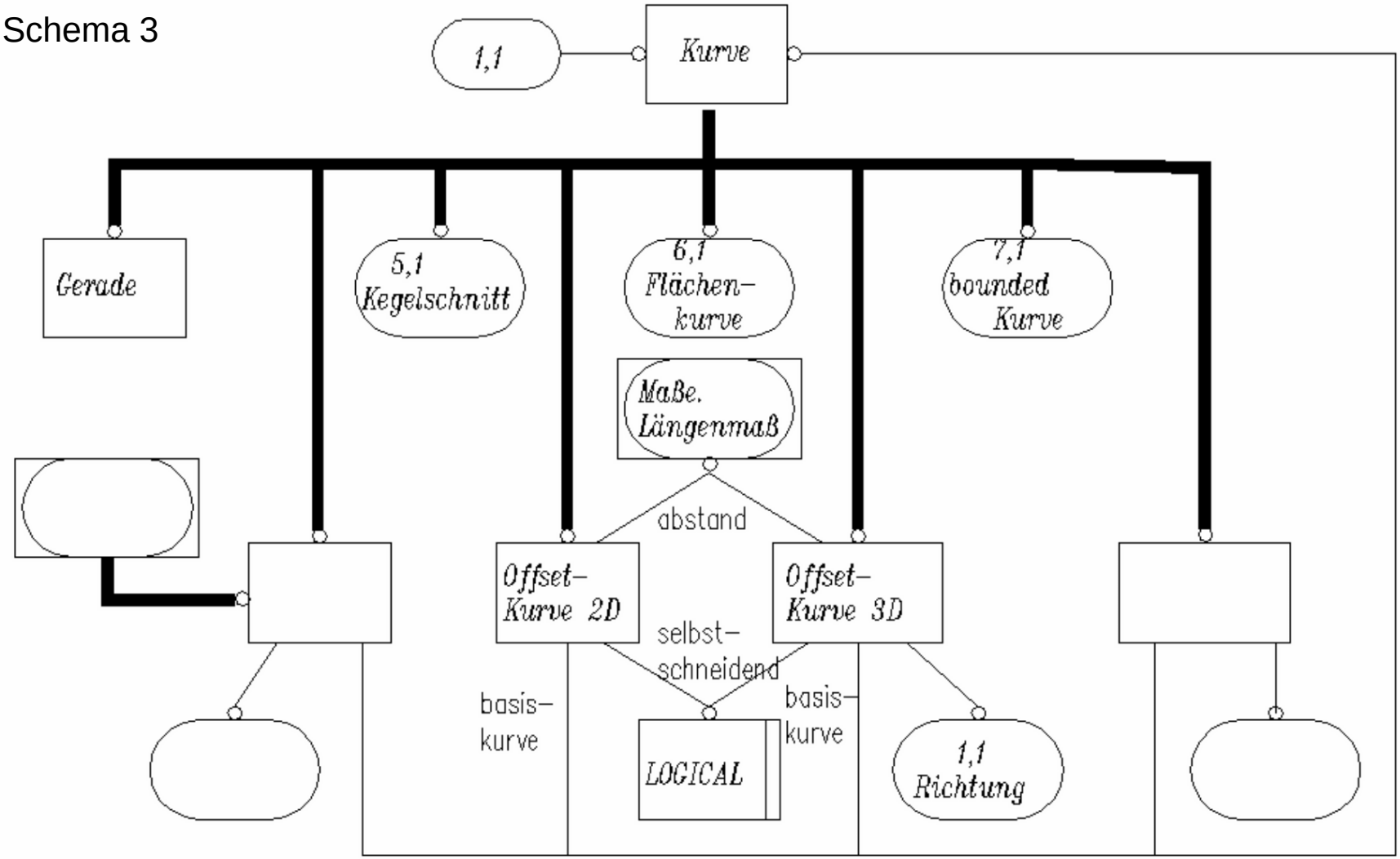
Schema 1



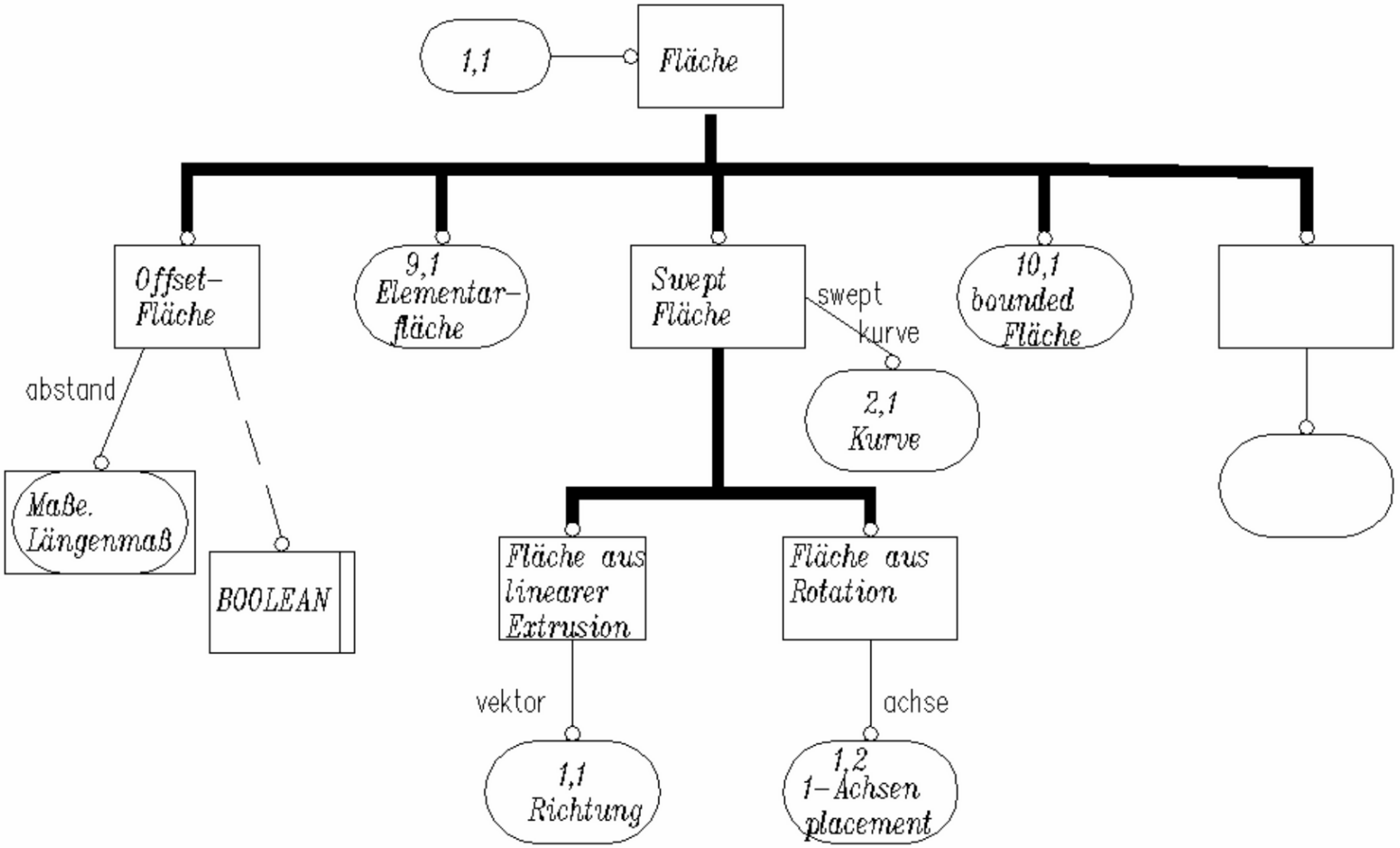
Schema 2


1,2
1,1
Punkt
kartesisch. Punkt
Kurven– punkt
Flächen– punkt
koordinaten LIST [2:3]
basis– kurve
punkt– parameter
punkt– parameter u und v
basisfläche
Maße. Längenmaß
3,1 Kurve
REAL
4,1 Fläche

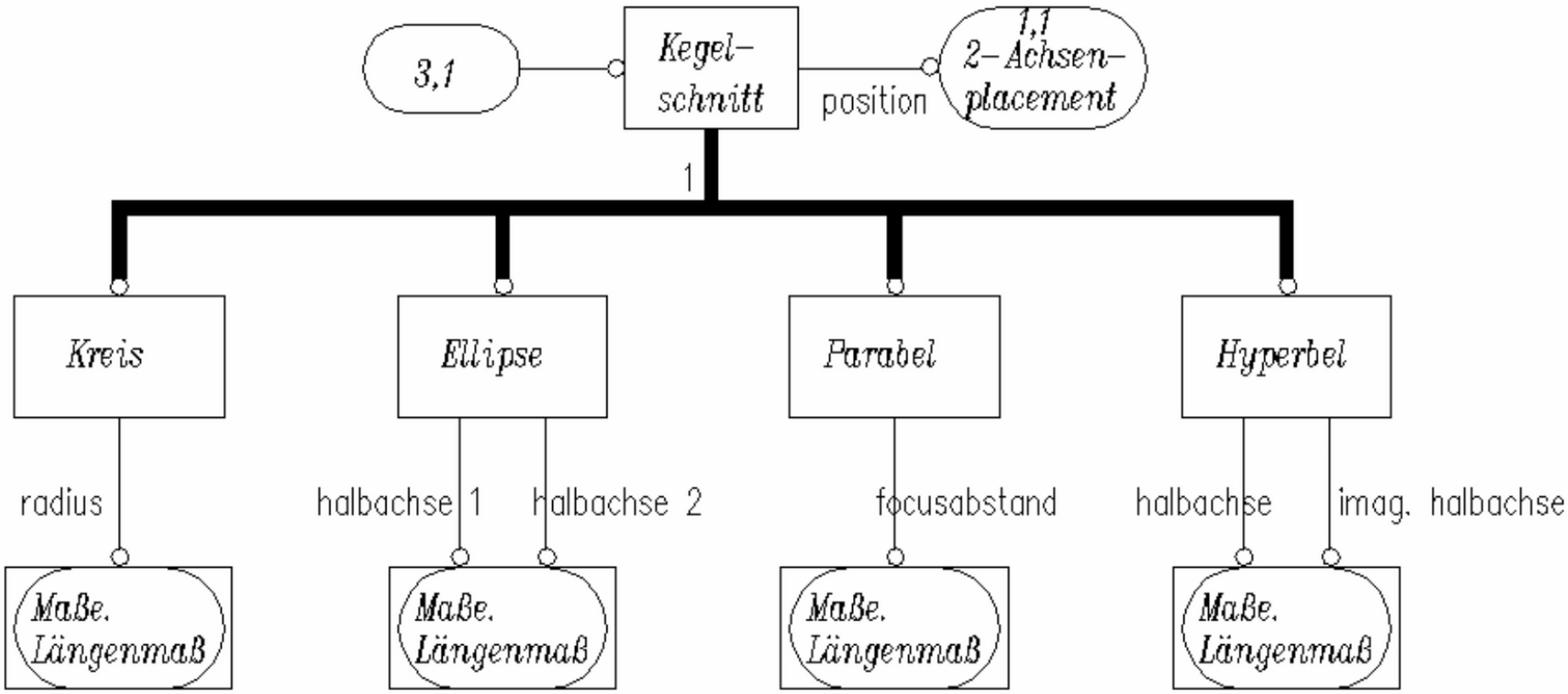
Schema 3


1,1
Kurve
Gerade
5,1
Kegelschnitt
6,1
Flächen-
kurve
7,1
bounded
Kurve
Maße.
Längenmaß
abstand
Offset-
Kurve 2D
Offset-
Kurve 3D
selbst-
schneidend
basis-
kurve
basis-
kurve
LOGICAL
1,1
Richtung

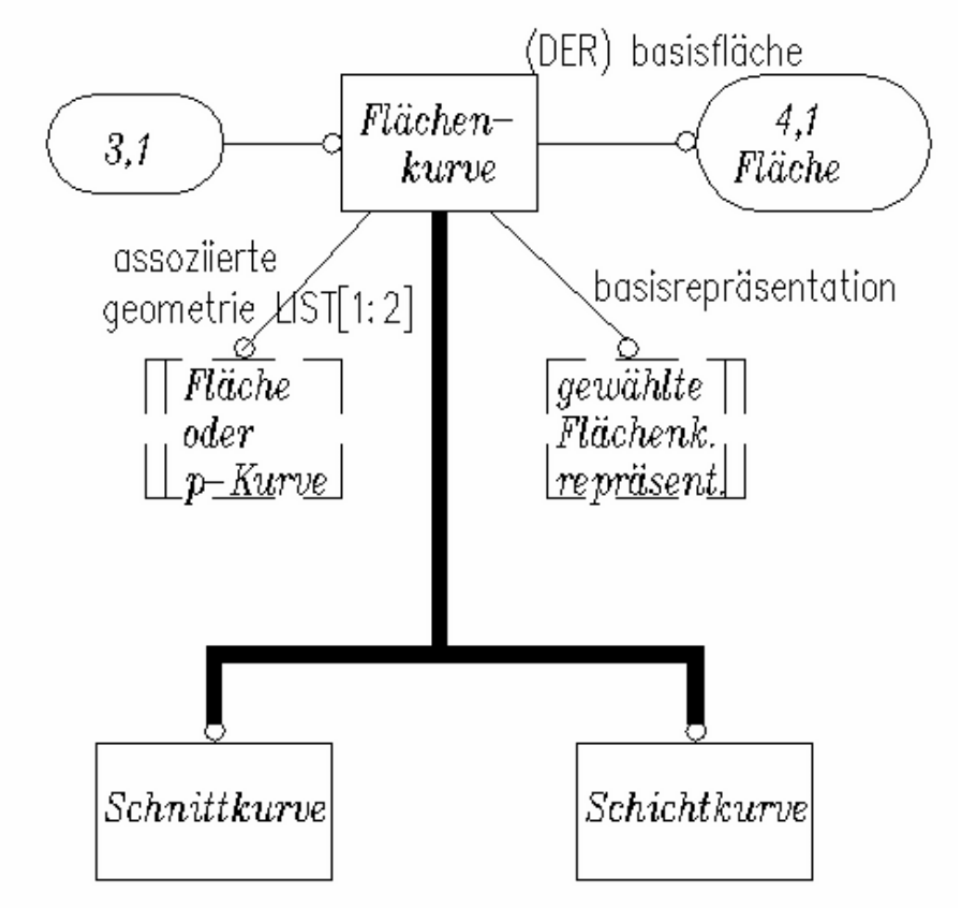
Schema 4


1,1
Fläche
Offset–
Fläche
9,1
Elementar–
fläche
Swept
Fläche
swept
kurve
10,1
bounded
Fläche
abstand
2,1
Kurve
Maße.
Längenmaß
BOOLEAN
Fläche aus
linearer
Extrusion
Fläche aus
Rotation
vektor
achse
1,1
Richtung
1,2
1–Achsen
placement

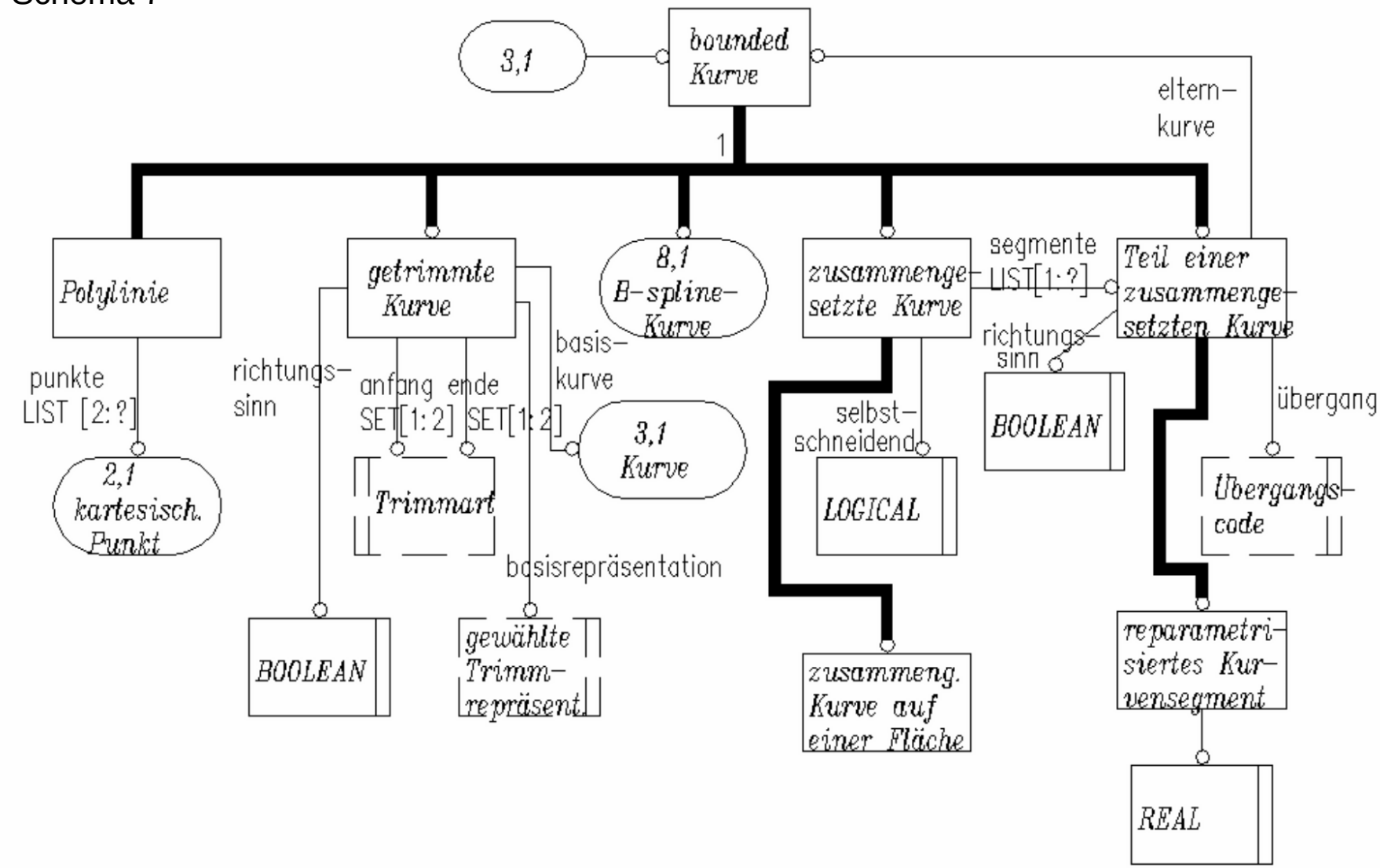
Schema 5


3,1
Kegel-
schnitt
position
1,1
2-Achsen-
placement
1
Kreis
Ellipse
Parabel
Hyperbel
radius
halbachse 1
halbachse 2
focusabstand
halbachse
imag. halbachse
Maße.
Längenmaß
Maße.
Längenmaß
Maße.
Längenmaß
Maße.
Längenmaß

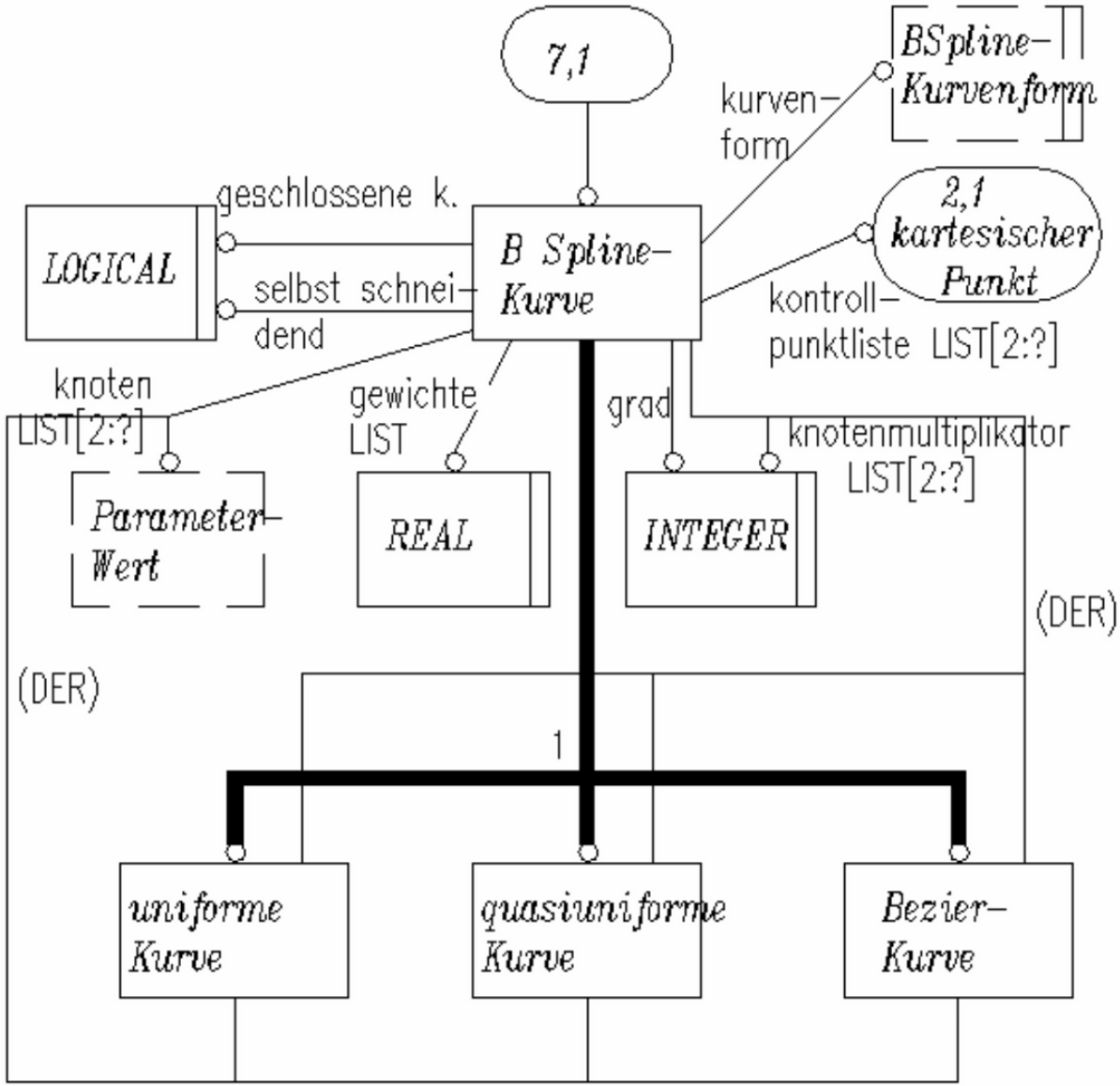
Schema 6


(DER) basisfläche
3,1
Flächen-
kurve
4,1
Fläche
assoziierte
geometrie LIST[1:2]
basisrepräsentation
Fläche
oder
p-Kurve
gewählte
Flächenk.
repräsent.
Schnittkurve
Schichtkurve

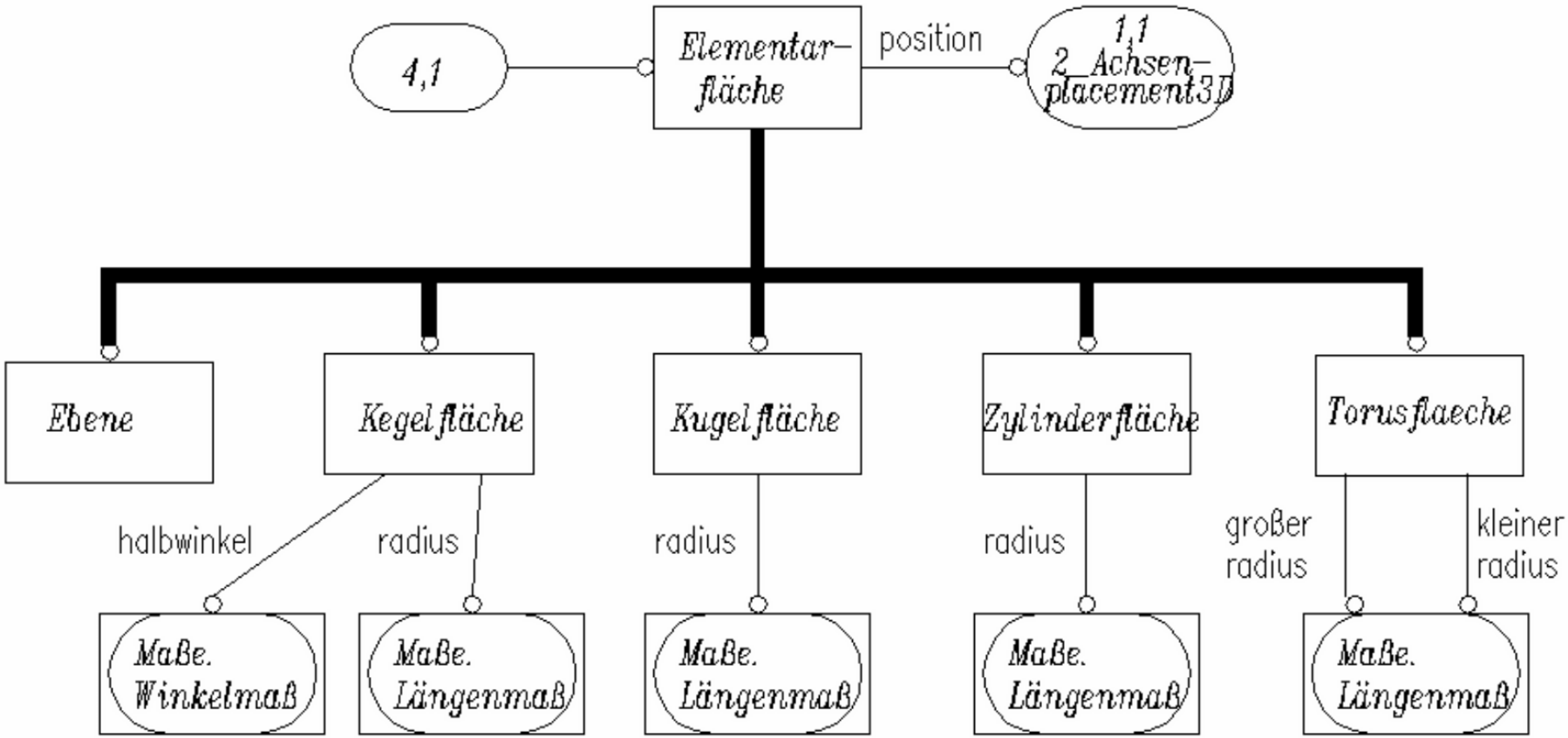
Schema 7


3,1
bounded Kurve
eltern-kurve
1
Polylinie
getrimmte Kurve
8,1 B-spline-Kurve
zusammenge-setzte Kurve
segmente LIST[1:?]
Teil einer zusammenge-setzten Kurve
punkte LIST [2:?]
richtungs-sinn
anfang SET[1:2]
ende SET[1:2]
basis-kurve
richtungs-sinn
selbst-schneidend
übergang
2,1 kartesisch. Punkt
Trimmart
3,1 Kurve
BOOLEAN
LOGICAL
Übergangs-code
basisrepräsentation
BOOLEAN
gewählte Trimm-repräsent.
zusammeng. Kurve auf einer Fläche
reparametri-siertes Kur-vensegment
REAL

Schema 8

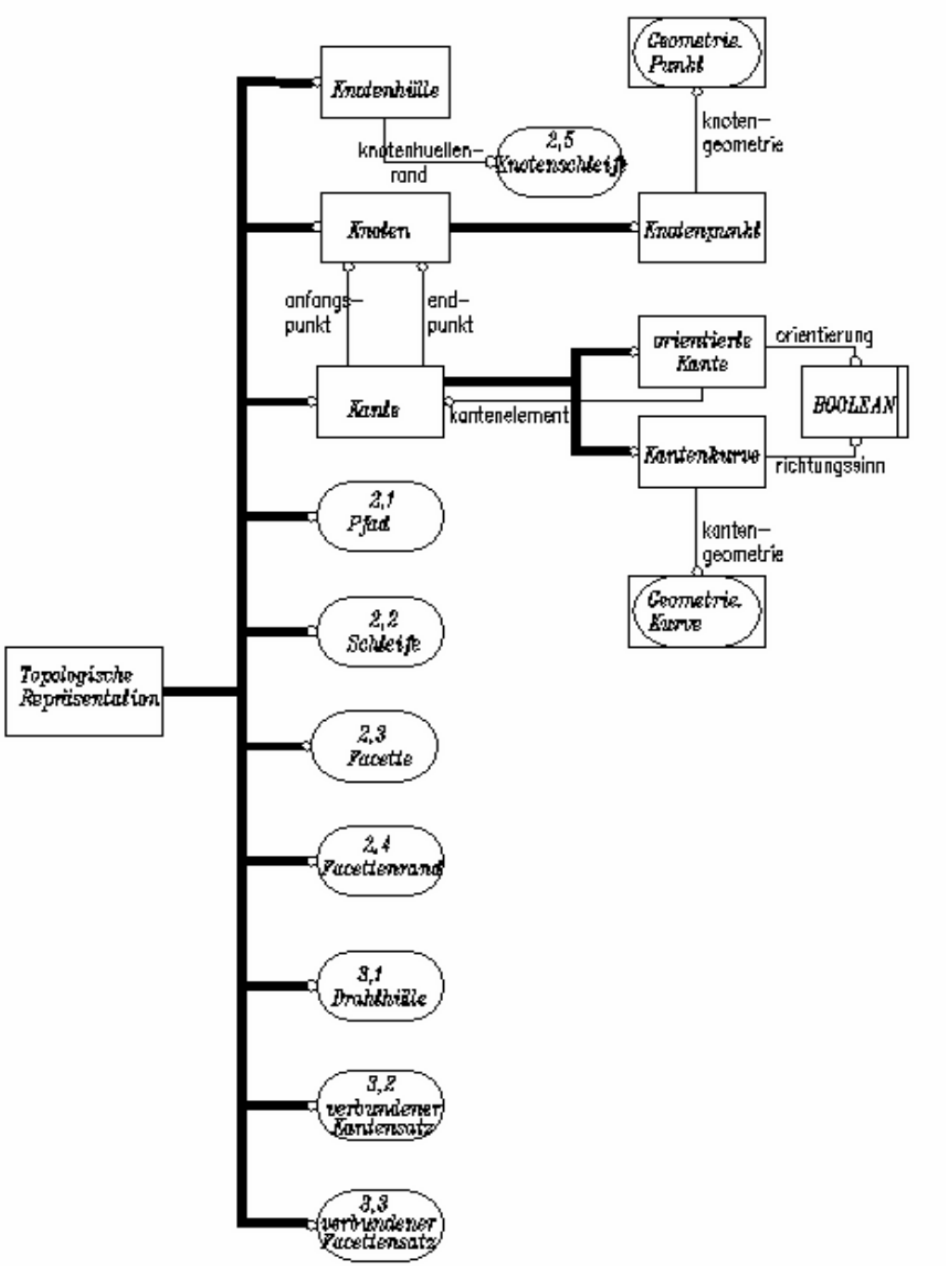

7,1
BSpline–
Kurvenform
kurven–
form
geschlossene k.
LOGICAL
B Spline–
Kurve
2,1
kartesischer
Punkt
selbst schnei–
dend
kontroll–
punktliste LIST[2:?]
knoten
LIST[2:?]
gewichte
LIST
grad
knotenmultiplikator
LIST[2:?]
Parameter–
Wert
REAL
INTEGER
(DER)
(DER)
1
uniforme
Kurve
quasiuniforme
Kurve
Bezier–
Kurve

Schema 9

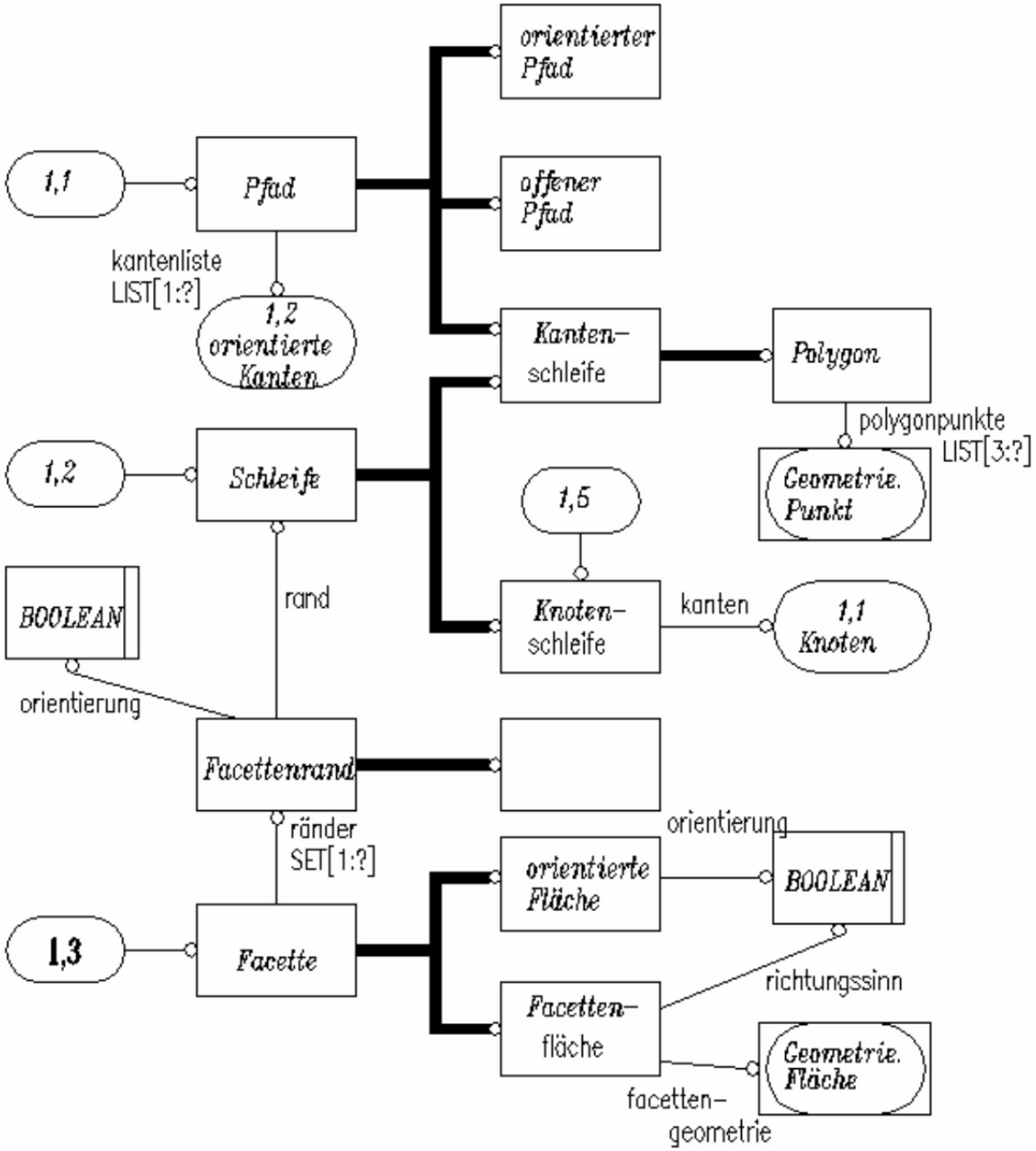

4,1
Elementar–
fläche
position
1,1
2_Achsen–
placement3D
Ebene
Kegelfläche
Kugelfläche
Zylinderfläche
Torusflaeche
halbwinkel
radius
radius
radius
großer
radius
kleiner
radius
Maße.
Winkelmaß
Maße.
Längenmaß
Maße.
Längenmaß
Maße.
Längenmaß
Maße.
Längenmaß

9.2.2. Topologie - Schemata

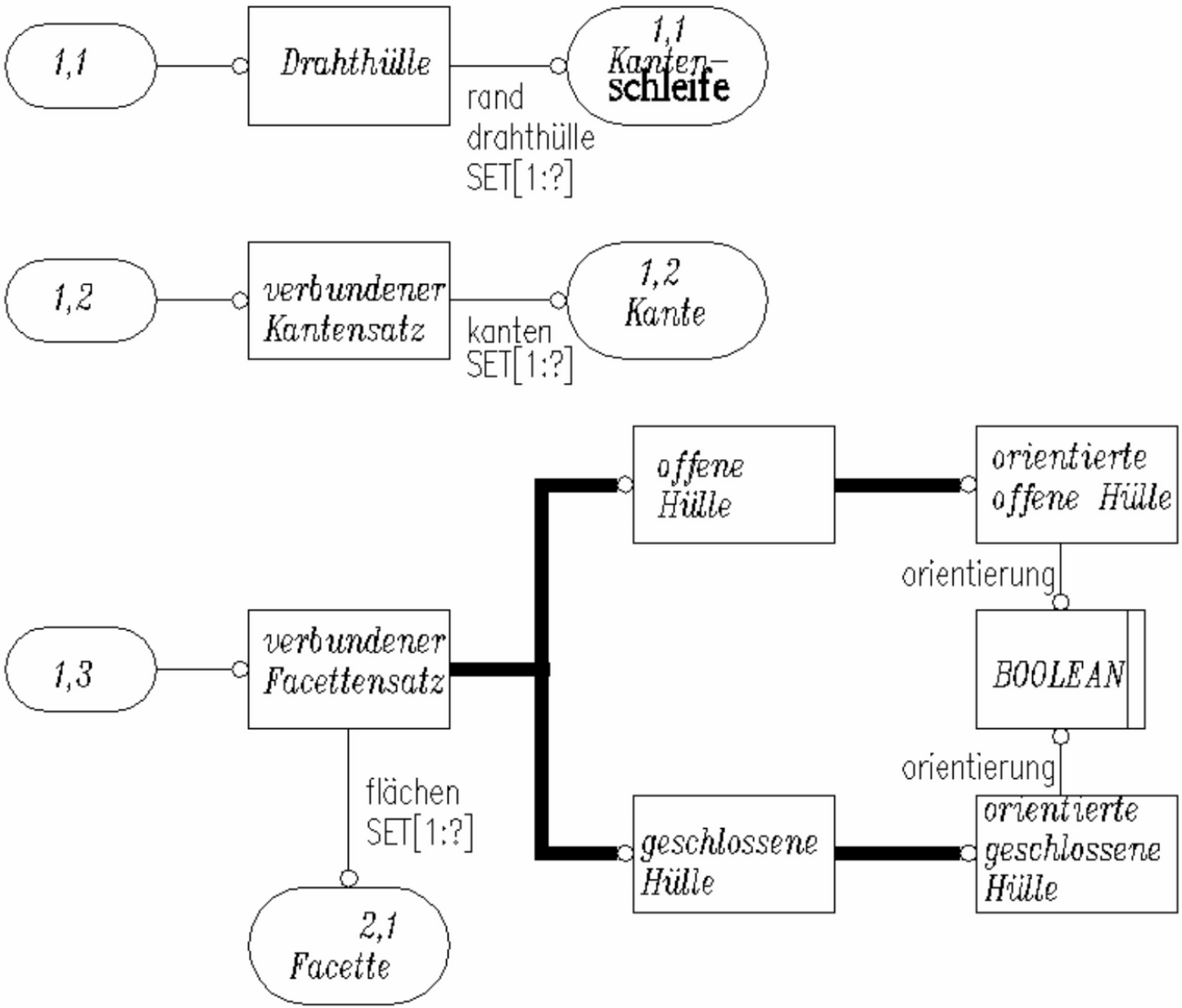
Schema 1



Schema 2

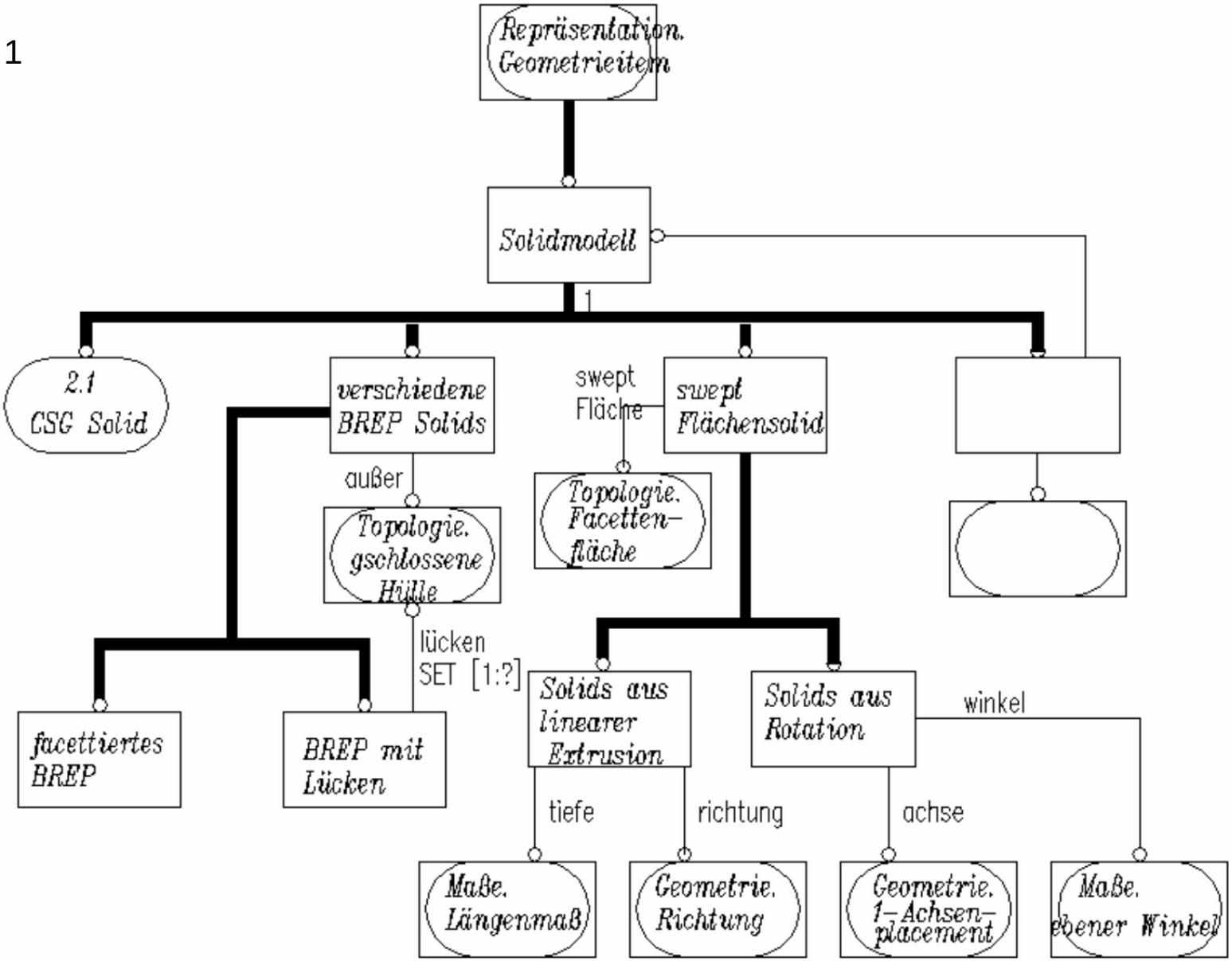

orientierter Pfad
1,1
Pfad
offener Pfad
kantenliste
LIST[1:?]
1,2
orientierte Kanten
Kanten-
schleife
Polygon
polygonpunkte
LIST[3:?]
1,2
Schleife
1,5
Geometrie. Punkt
BOOLEAN
rand
Knoten-
schleife
kanten
1,1
Knoten
orientierung
Facettenrand
ränder
SET[1:?]
orientierung
orientierte Fläche
BOOLEAN
1,3
Facette
Facetten-
fläche
richtungssinn
Geometrie. Fläche
facetten-
geometrie

Schema 3

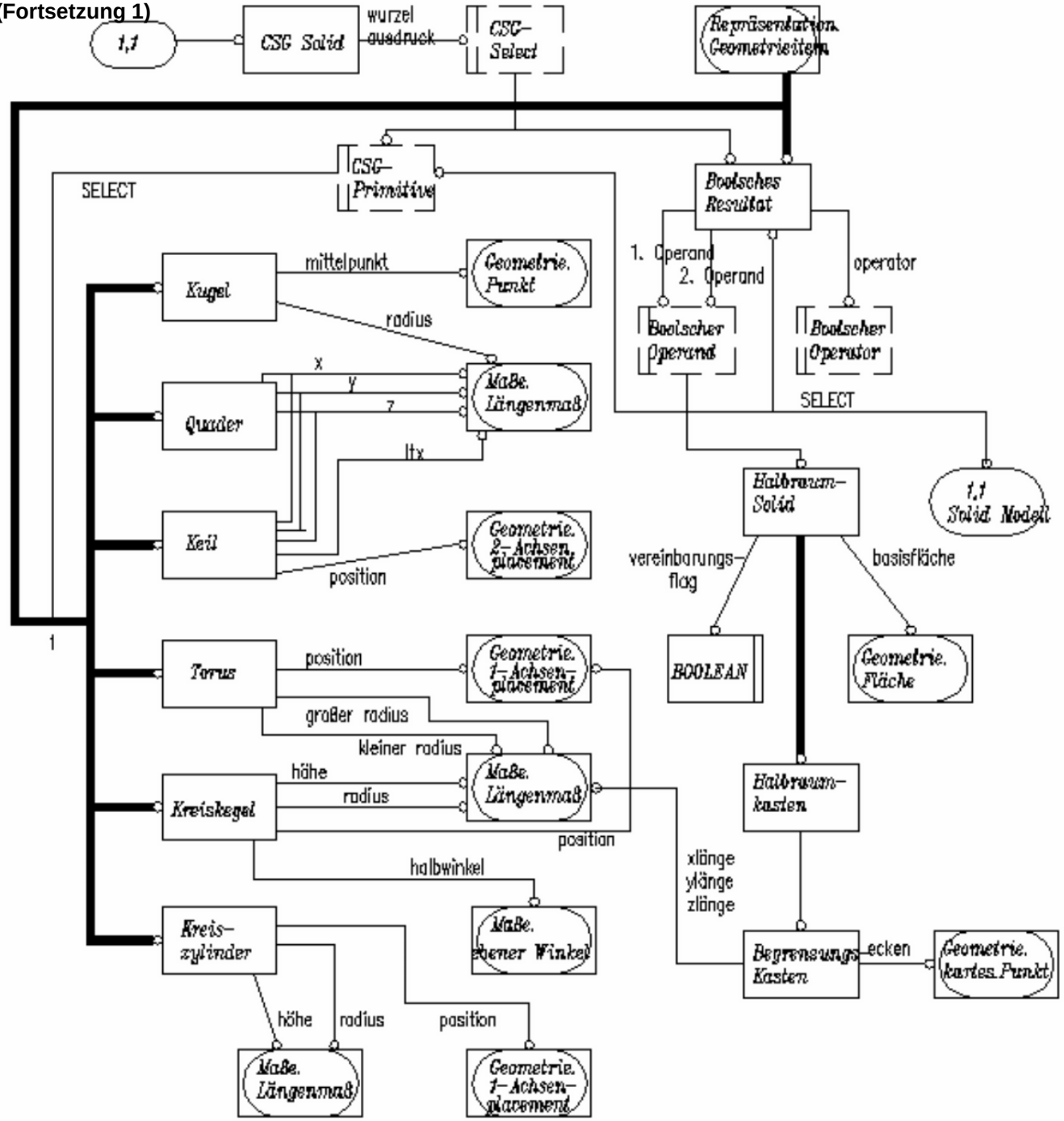

1,1
Drahthülle
1,1
Kanten-
schleife
rand
drahthülle
SET[1:?]
1,2
verbundener
Kantensatz
1,2
Kante
kanten
SET[1:?]
offene
Hülle
orientierte
offene Hülle
orientierung
1,3
verbundener
Facettensatz
BOOLEAN
orientierung
flächen
SET[1:?]
geschlossene
Hülle
orientierte
geschlossene
Hülle
2,1
Facette

9.2.3. Solid - Schemata

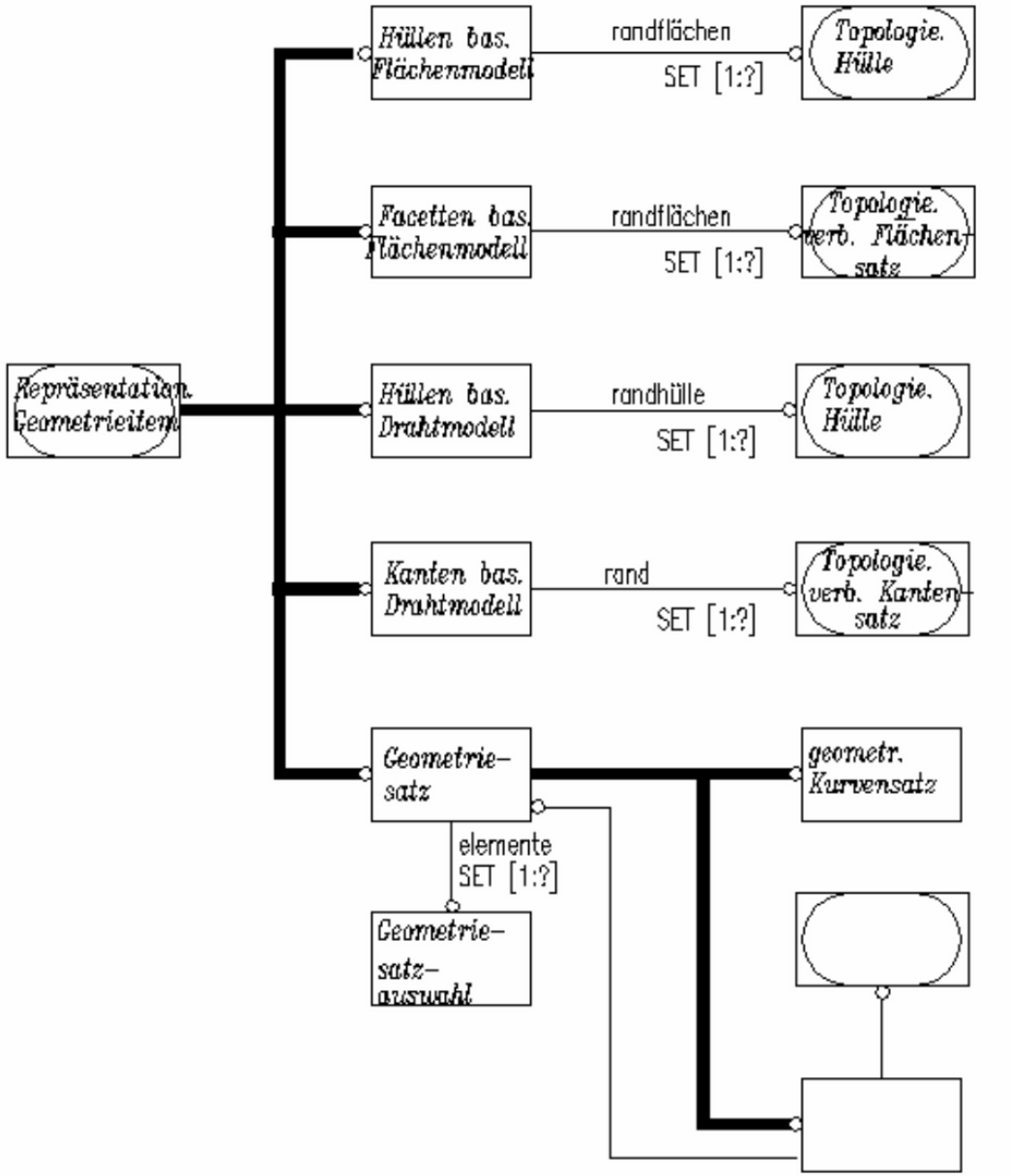
Schema 1



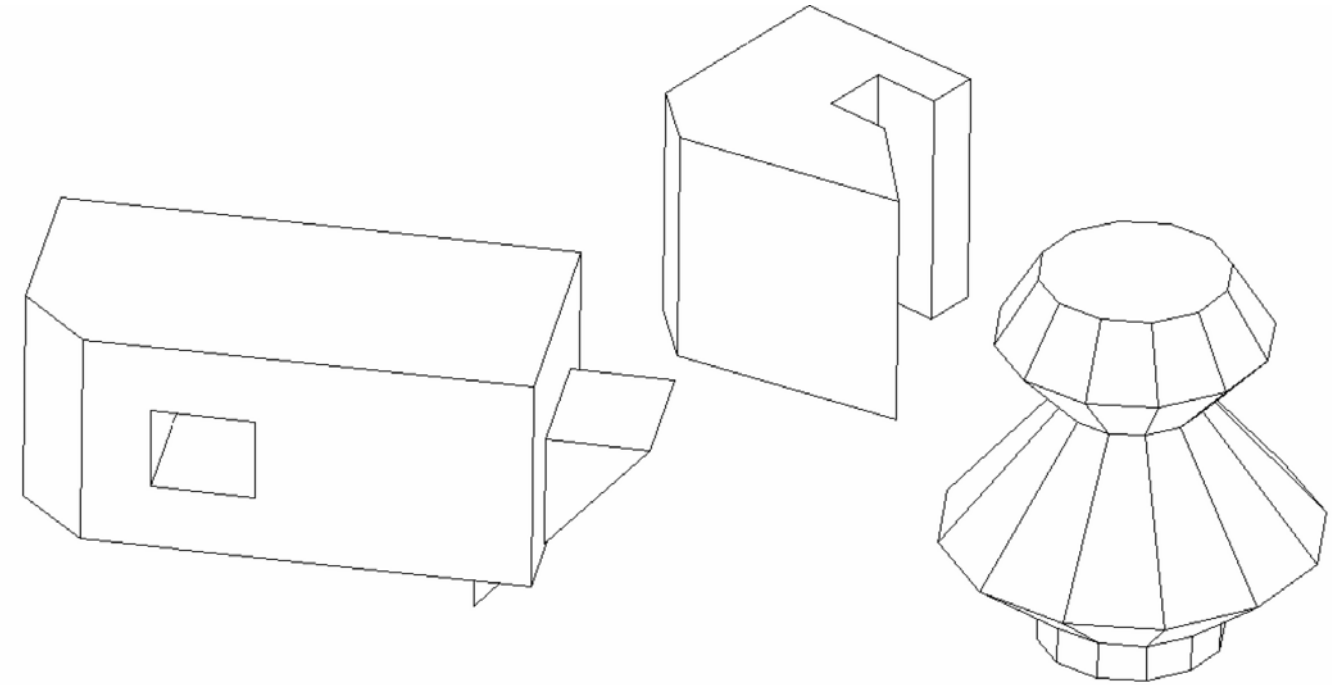
Schema 2


1,1
CSG Solid
wurzel
ausdruck
CSG-Select
Repräsentation Geometrieitem
SELECT
CSG-Primitive
Boolsches Resultat
1. Operand
2. Operand
operator
Kugel
mittelpunkt
Geometrie. Punkt
radius
Boolscher Operand
Boolscher Operator
x
y
z
Quader
Maße. Längenmaß
SELECT
ltx
Halbraum-Solid
1,1 Solid Modell
Keil
Geometrie. 2-Achsen-placement
position
vereinbarungs-flag
basisfläche
1
Torus
position
Geometrie. 1-Achsen-placement
BOOLEAN
Geometrie. Fläche
großer radius
kleiner radius
höhe
Kreiskegel
radius
Maße. Längenmaß
Halbraum-kasten
position
halbwinkel
xlänge
ylänge
zlänge
Kreis-zylinder
Maße. ebener Winkel
Begrenzungs-Kasten
ecken
Geometrie. kartes. Punkt
höhe
radius
position
Maße. Längenmaß
Geometrie. 1-Achsen-placement

Schema 3

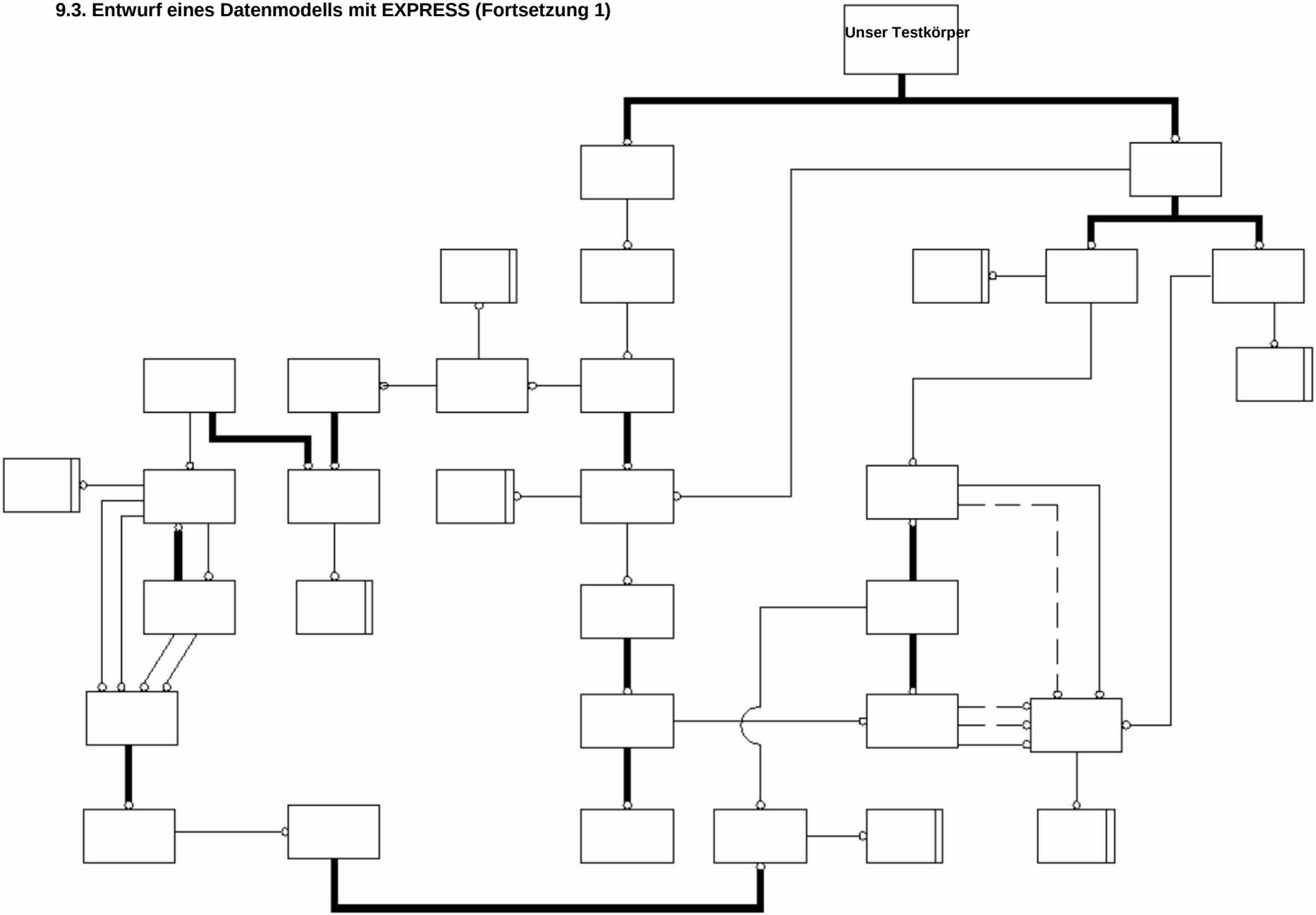

Repräsentation. Geometrieitem
Hüllen bas. Flächenmodell
randflächen
SET [1:?]
Topologie. Hülle
Facetten bas. Flächenmodell
randflächen
SET [1:?]
Topologie. verb. Flächen satz
Hüllen bas. Drahtmodell
randhülle
SET [1:?]
Topologie. Hülle
Kanten bas. Drahtmodell
rand
SET [1:?]
Topologie. verb. Kanten satz
Geometrie- satz
elemente
SET [1:?]
Geometrie- satz- auswahl
geometr. Kurvensatz

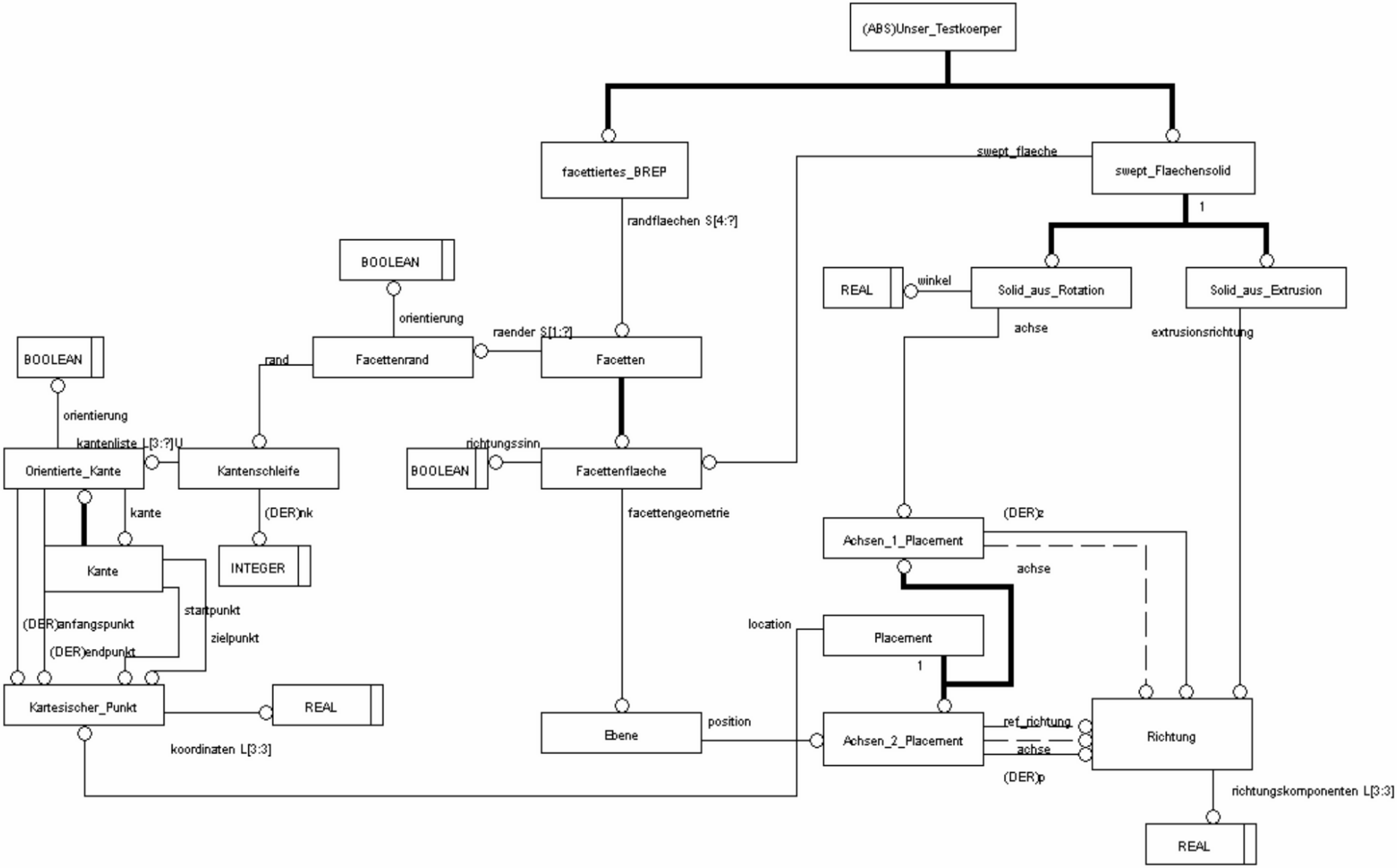
9.3. Entwurf eines Datenmodells mit EXPRESS



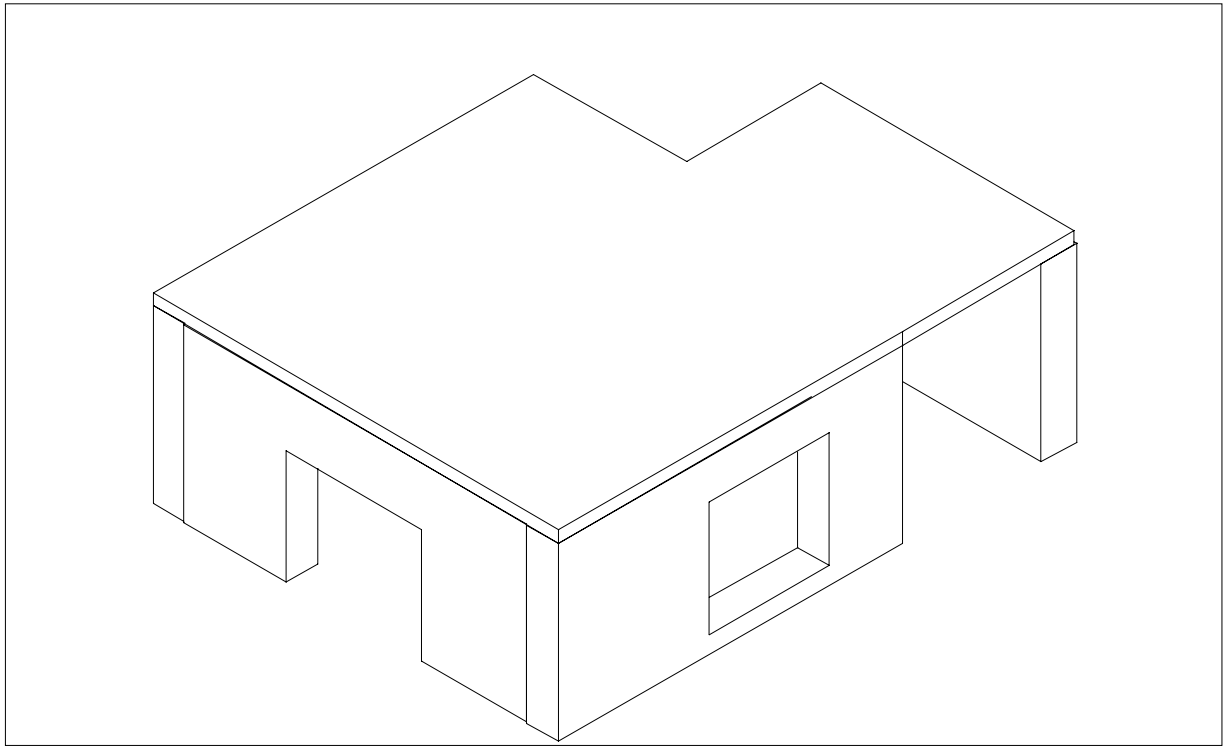
Entwickeln Sie ein EXPRESS-Datenmodell, daß Objekte speichern kann, die der obigen Abbildung entsprechen.

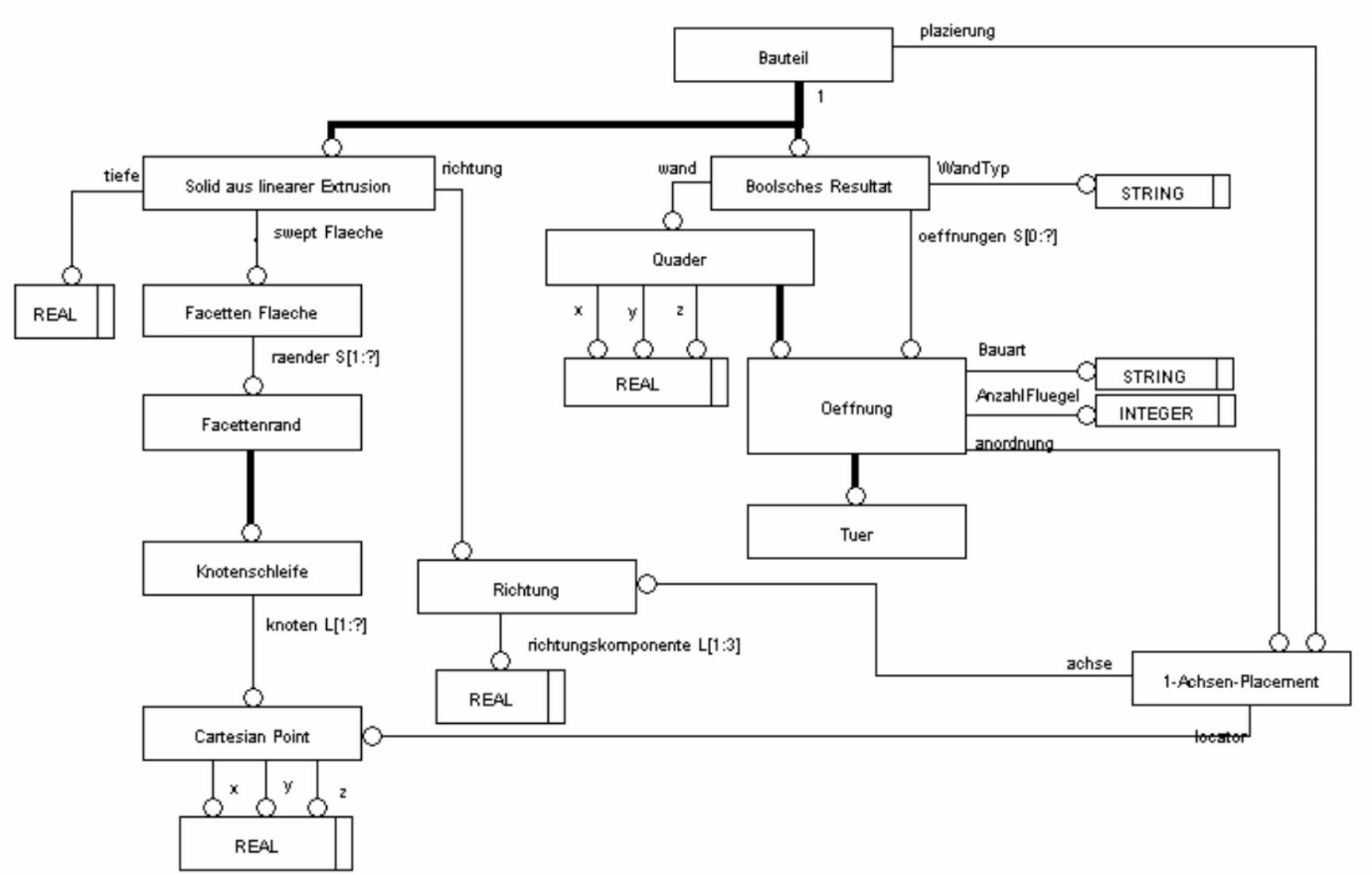
Beachten Sie, daß die drei Körper unterschiedliche Modellierungstechniken repräsentieren!


Unser Testkörper


(ABS)Unser_Testkoerper
facettiertes_BREP
randflaechen S[4:?]
swept_flaeche
swept_Flaechensolid
1
BOOLEAN
orientierung
REAL
winkel
Solid_aus_Rotation
Solid_aus_Extrusion
achse
extrusionsrichtung
raender S[1:?]
BOOLEAN
rand
Facettenrand
Facetten
orientierung
kantenliste L[3:?]U
Orientierte_Kante
Kantenschleife
BOOLEAN
richtungssinn
Facettenflaeche
kante
(DER)nk
facettengeometrie
(DER)z
Achsen_1_Placement
Kante
INTEGER
achse
startpunkt
(DER)anfangspunkt
zielpunkt
(DER)endpunkt
location
Placement
1
Kartesischer_Punkt
REAL
koordinaten L[3:3]
Ebene
position
Achsen_2_Placement
ref_richtung
achse
(DER)p
Richtung
richtungskomponenten L[3:3]
REAL

Ein zweiter Versuch: Ein Modell für eine einfache Bau-Applikation ist gesucht!




Bauteil
plazierung
1
tiefe
Solid aus linearer Extrusion
richtung
wand
Boolsches Resultat
WandTyp
STRING
swept Flaeche
Quader
oeffnungen S[0:?]
REAL
Facetten Flaeche
x
y
z
REAL
Bauart
STRING
raender S[1:?]
Oeffnung
AnzahlFluegel
INTEGER
Facettenrand
anordnung
Tuer
Knotenschleife
Richtung
knoten L[1:?]
richtungskomponente L[1:3]
achse
1-Achsen-Placement
REAL
Cartesian Point
locator
x
y
z
REAL

9.4. Die Sprache EXPRESS - Beispiel

Alle notwendigen Informationen zur Syntax der Sprache finden Sie in dem Word-Dokument "EXPRESS_Syntax.doc"

```
(****************************** CONTENTS   ***********************************
****************************** ENTITIES   ***********************************
achsen_1_placement
achsen_2_placement
ebene
facetten
facettenflaeche
facettenrand
facettiertes_brep
kante
kantenschleife
kartesischer_punkt
orientierte_kante
placement
richtung
solid_aus_extrusion
solid_aus_rotation
swept_flaechensolid
unser_testkoerper
****************************** END CONTENTS *********************************)
SCHEMA modell_vorlesung;

 ENTITY unser_testkoerper
  ABSTRACT SUPERTYPE OF (ONEOF(swept_flaechensolid,facettiertes_brep));
 END_ENTITY;
```

```
ENTITY facettiertes_brep
  SUBTYPE OF (unser_testkoerper);
  randflaechen : SET[4:?] OF facetten;
 END_ENTITY;
 ENTITY swept_flaechensolid
  SUPERTYPE OF (ONEOF(solid_aus_rotation,solid_aus_extrusion))
  SUBTYPE OF (unser_testkoerper);
  swept_flaeche : facettenflaeche;
 END_ENTITY;

 ENTITY solid_aus_rotation
  SUBTYPE OF (swept_flaechensolid);
  winkel : REAL;
  achse : achsen_1_placement;
 END_ENTITY;

 ENTITY solid_aus_extrusion
  SUBTYPE OF (swept_flaechensolid);
  tiefe : REAL;
  extrusionsrichtung : richtung;
 END_ENTITY;

 ENTITY achsen_1_placement
  SUBTYPE OF (placement);
  achse : OPTIONAL richtung;
  DERIVE
   z : richtung := richtung(0.0,0.0,1.0);
 END_ENTITY;
```

```
ENTITY placement
  SUPERTYPE OF (ONEOF(achsen_1_placement,achsen_2_placement));
  location : kartesischer_punkt;
 END_ENTITY;

 ENTITY achsen_2_placement
  SUBTYPE OF (placement);
  achse : OPTIONAL richtung;
  ref_richtung : OPTIONAL richtung;
  DERIVE
   p : richtung := richtung(0.0,0.0,1.0);
 END_ENTITY;

 ENTITY richtung;
  richtungskomponenten : LIST[3:3] OF REAL;
 END_ENTITY;

 ENTITY facetten;
  raender : SET[1:?] OF facettenrand;
 END_ENTITY;

 ENTITY facettenflaeche
  SUBTYPE OF (facetten);
  facettengeometrie : ebene;
  richtungssinn : BOOLEAN;
 END_ENTITY;

 ENTITY ebene;
  position : achsen_2_placement;
 END_ENTITY;
```

```
ENTITY facettenrand;
  orientierung : BOOLEAN;
  rand : kantenschleife;
 END_ENTITY;

 ENTITY kantenschleife;
  kantenliste : LIST[3:?] OF UNIQUE orientierte_kante;
  DERIVE
   nk : INTEGER := 2;
 END_ENTITY;

 ENTITY orientierte_kante
  SUBTYPE OF (kante);
  orientierung : BOOLEAN;
  kante : kante;
  DERIVE
   anfangspunkt : kartesischer_punkt := 0;
   endpunkt : kartesischer_punkt := 0;
 END_ENTITY;

 ENTITY kante;
  startpunkt : kartesischer_punkt;
  zielpunkt : kartesischer_punkt;
 END_ENTITY;

 ENTITY kartesischer_punkt;
  koordinaten : LIST[3:3] OF REAL;
 END_ENTITY;

END_SCHEMA;
```

Quellen

- W.-D. Groch "Skript Homogene Koordinaten" FH Darmstadt, Fb I
- N. Bronstein et al. "Taschenbuch der Mathematik"
- J. D. Foley et al. "Grundlagen der Computergraphik"
- W. M. Newman, R. F. Sproull "Grundzüge der interaktiven Computergraphik"
- F. P. Preparata, M. I. Shamos "Computational Geometry – an Introduction"
- http://arxdummies.blogspot.com/2006/03/autocad-2007.html
- …\ObjectARX2007\arxlabs\ObjectARXLabs.chm
- http://download.autodesk.com/media/adn/DevTV_Introduction_to_AutoCAD._NET_Programming/

Download-URL für studentische (kostenlose) Autodesk Civil 3D 2009 –Software

`http://students.autodesk.com/?nd=or&id=2742&ic=gaoE3Sg49T`

oder

`http://students.autodesk.com`

Download-URL für ObjectARX (oder von meiner Homepage)

`http://usa.autodesk.com/adsk/servlet/index?id=773204&siteID=123112`