DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Studienganges Informationstechnik

Bericht zum Modul Praxis II

1. Praxisphase: 24.12.2013 - 30.03.2014
2. Praxisphase: 21.06.2014 - 30.09.2014

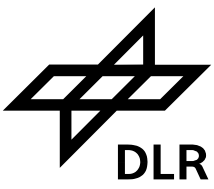
Modellierung von Klappen und Kontrollflächen in der Geometriebibliothek TiGL

von

Mark Geiger

- Martrikelnr.: 3793077 -

- Kurs: TINF12ITIN -



Deutsches Zentrum für Luft- und Raumfahrt e.V.
in der Helmholtz Gemeinschaft

Standort: Köln

Einrichtung für Simulations- und Softwaretechnik
Abteilung Verteilte Systeme und Komponentensoftware

Betreuer: Dr. Martin Siggel

Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir persönlich ohne Hilfe Dritter verfasst wurde und dass ich keine weiteren als die angegeben Hilfsmittel und Quellen benutzt habe. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen sind entsprechend gekennzeichnet. Sämtliche Quellen sind nachgewiesen und im Literaturverzeichnis ausgeführt.

Diese Arbeit ist weder ganz noch in Teilen einer anderen Prüfungsbehörde vorgelegt worden.

Köln, den 7. September 2014

Kurzfassung

Die am Deutschen Zentrum für Luft- und Raumfahrt entwickelte Geometriebibliothek TiGL und die dazugehörige Software TiGLViewer ermöglicht es den Wissenschaftlern am DLR 3D Geometrien aus CPACS Datensätzen zu erzeugen und zu visualisieren. Aktuell können von TiGL nur simple Geometrien wie z.B. Flugzeugrümpfe und die Flügel erzeugt werden.

In dieser Arbeit wird dargestellt wie TiGL und der TiGLViewer dahingehend erweitert wurden, sodass die Flügelgeometrien genauer berechnet und visualisiert werden können. Dazu gehört besonders die Modellierung und Berechnung von Kontrollflächen, Landeklappen und Spoilern. Durch die Anpassung und Erweiterung der Geometrieberechnung ist es möglich genauere Simulationen zu berechnen um z.B. die Radarsignatur des Luftfahrzeugs mit aus und eingeklappten Landeklappen zu vergleichen und zu untersuchen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
ESA	**E**uropean **S**pace **A**gency
TiGL	**Ti**GL **G**eometry **L**ibrary
TIVA	**T**echnology **I**ntergration for the **V**irtual **A**ircraft
CPACS	**C**ommon **P**arametric **A**ircraft **C**onfiguration **S**chema
XML	**E**xtensible **M**arkup **L**anguage
CASCADE	**C**omputer **A**ided **S**oftware for **C**omputer **A**ided **D**esign and **E**ngineering
CMAKE	**C**ross-platform **Make**
TIXI	**TI**VA **X**ML **I**nterface
GUI	**G**raphical **U**ser **I**nterface
LGPL	**L**esser **G**eneral **P**ublic **L**icense
GTest	**G**oogle C++ **T**esting Framework
LGS	**L**ineares **G**leichungs **S**ystem

Abbildungsverzeichnis

Quellcodeverzeichnis

Formelverzeichnis

Vorwort

Die vorliegende Arbeit ist an meinem Arbeitsplatz beim **D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V. (DLR) in der Einrichtung Simulations- Softwaretechnik Abteilung Verteilte Systeme und Komponentensoftware entstanden. Das DLR ist das nationale Forschungszentrum der Bundesrepublik Deutschland für die Forschungsgebiete Luftfahrt, Raumfahrt, Sicherheit, Energie und Verkehr. Das DLR unterhält ca. 7700 Mitarbeiter an 16 Standorten (Berlin, Köln, Göttingen, Stuttgart, uvm.) und 4 Büros in Paris, Brüssel, Washington D.C. und Tokio. Das DLR ist Mitglied in nationalen sowie internationalen Einrichtung, wie zum Beispiel der **E**uropean **S**pace **A**gency (ESA). In der ESA vertritt das DLR die deutschen Raumfahrtsinteressen und ist zuständig für die Planung und Umsetzung des deutschen Raumfahrtbudgets.[1] [2]

Die Abteilung Verteilte Systeme und Komponentensoftware vertritt innerhalb des deutschen Zentrums für Luft- und Raumfahrt das Ziel, die Forschung der Wissenschaftler mit Individualsoftware zu unterstützen. Zudem entwickelt, untersucht und identifiziert die Abteilung neue Softwaretechnologien für das DLR. Die Strategie ist es neue Software, in hoher Qualität, so individuell wie nötig, termingerecht und effizient zu entwickeln. Intern ist die Abteilung in drei Gruppen unterteilt: Verteilte Softwaresysteme, Software Engineering und High Performance Computing. Während dem zweiten Studienjahr, war ich in der Gruppe High Performance Computing tätig. Unter der Leitung von Dr. Martin Siggel, entstand in dieser Gruppe die vorliegende Arbeit.[3] [4]

[1] [DLR] vgl. Das DLR im Überblick: Absatz 1,4,5
[2] [BMBF] vgl. Projektträger im DLR
[3] [SC] vgl. Simulations- und Softwaretechnik: Verteilte Systeme und Komponentensoftware
[4] [SC] vgl. Themen

Modellierung von Klappen und Kontrollflächen in der Geometriebibliothek TiGL

1 Einleitung

Die Modellierung neuer Flugzeugmodelle ist aufwändig und komplex, eine Entwicklung solcher Modelle ohne spezielle Unterstützungstools ist undekbar. Eine dieser Unterstützungstools ist die **Ti**GL **G**eometry **L**ibrary (TiGL). TiGL und die darauf basierende Software TiGLViewer bilden die zentralen Punkte dieser Arbeit. TiGL ist eine Geometriebibliothek mit dessen hilfe dreidimensionale Flugzeuggeometrien aus parametrischen Datensätzen abgeleitet werden können. Die Daten stehen in **E**xtensible **M**arkup **L**anguage (XML) Dateien die nach der Datendefinition **C**ommon **P**arametric **A**ircraft **C**onfiguration **S**chema (CPACS) erstellt wurden zur Verfügung. TiGL und der TiGLViewer können diese Datensätze laden und darauf verschiedene Berechnungen durchführen. Ein Flugzeugmodell, welches als CPACS Datei vorliegt, kann als 3D Modell mit TiGL berechnet und im TiGLViewer angezeigt werden. Die von TiGL erstellten 3D Modelle können dann in verschiedene Dateitypen exportiert werden, welche wiederrum als Input für verschiedenste Simulationssoftwares dienen.[5]

1.1 Aufgabenstellung

Bislang werden von TiGL nur einfache Flügel und Flugzeugrümpfe modelliert. Die in dieser Arbeit niedergeschriebene Aufgabe ist es Flügelklappen und Kontrollflächen zu modellieren und diese in TiGL einzubinden. Dabei sollen verschiedene Detailgrade und Klappentypen beachtet werden, dazu gehören z.B. Landeklappen, Spoiler und Trimmklappen. Es soll dabei möglich sein die Klappen und Kontrollflächen im ein-

[5] [TIGL] vgl. TiGL Visualization

und ausgefahrenen Zustand zu betrachten. Die neu erstellten Geometrien sollen letztendlich noch im TiGLViewer visualisiert werden.

1.2 Motivation

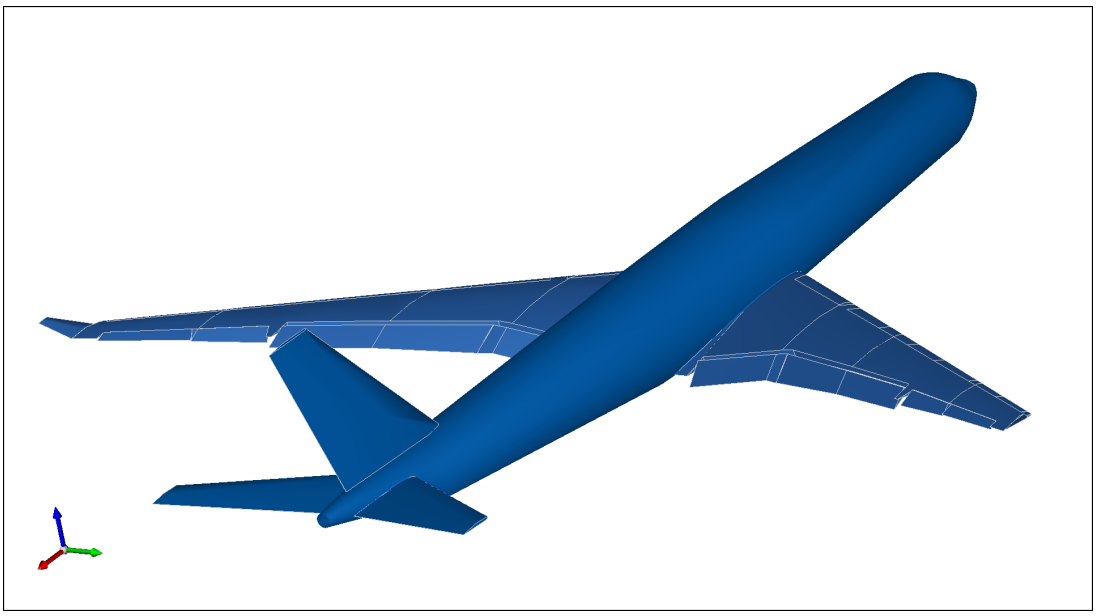


Abbildung 1: Flugzeuggeometrie mit Landeklappen[6]

Durch den erhöhten Detailgrad und die Möglichkeit Landeklappen im aus- und eingefahren Zustand darzustellen, wäre es möglich genauere Simulationen zu berechnen. Dadurch könnten ganz neue Fragen an die erstellten 3D-Modelle gestellt werden: Welche Auswirkungen haben die Klappen auf die Radarsignatur des Flugzeugs? Wie ändert sich das Flugverhalten des Flugzeugmodells durch die verschiedenen Stellwinkel der Landeklappen? Die in der Aufgabenstellung beschriebenen zusätzlichen Funktionen würden also die Vielseitigkeit und Genauigkeit von TiGL deutlich verbessern und zudem das Anwendugsspektrum erweitern. In diesem Bericht wird

[6]Quelle: Screenshot des TiGLViewers

Schritt für Schritt geklärt, wie es möglich ist aus einer simplen Flügelgeometrie eine komplexere Geometrie, wie sie in Abbildung 1 zu sehen ist, entsteht.

1.3 Herangehensweise

Das Vorgehen zum Lösen der Aufgabe lässt sich grob in drei fundamentale Abschnitte unterteilen, dem Aneignen von Wissen, der Problemlösung und der Auswertung.

Aneignung von Wissen: Um die in TiGL verwendeten Techniken, Programmiersprachen und Tools zu verstehen war es zunächst nötig, sich in die Materie einzuarbeiten. Dazu waren verschiedene Bücher und Berichte hilfreich (siehe sonstige Quellen im Literaturverzeichnis).

Problemlösung: Planung der Herangehensweise, Erstellen einer Anforderungsanalyse, Klärung von Designfragen und zuletzt dem Erstellen des Quellcodes.

Auswertung: Funktion und Richtigkeit der im Code angewandten Änderungen müssen überprüft werden. Die Auswertung besteht aus der Evaluation der vollführten Änderungen der Software und dem Anfertigen dieser Arbeit.

1.4 Struktur der vorliegenden Arbeit

Nach der bereits geschilderten Aufgabenstellung und der Herangehensweise folgen nun ein paar Grundlegende Informationen zum Verständnis der eigentlichen Problemlösung. Dazu gehört etwas allgemeineres Wissen im Umgang mit den benutzen Tools, verwendeten Sprachen, Sotwarebibliotheken und TiGL selbst. Danach wird

die Problemlösung, welche den Hauptteil dieser Arbeit ausmacht genau beschrieben. Dazu gehört auch die Anforderungsanalyse. Anschließend wird dargelegt wie die erforderliche Datenstruktur angelegt wird und wie mit Hilfe von TiXI, einer Bibliothek zum parsen von XML-Dateien, die CPACS Daten ausgelesen und zwischengespeichert werden können. Aus den ausgelesenen Daten werden dann die Flügelklappen modelliert und generiert. Im Anschluss daran wird die Transformation der Klappen behandelt und zuletzt folgt noch die Qualitätssicherung, ein Fazit und einen Ausblick in die Zukunft.

2 Grundlegende Informationen zum Verständnis

2.1 TiGL / TiGLViewer

TiGL ist eine in C++ geschrieben Geometriebibliothek. Diese ermöglicht es 3D Geometrien aus CPACS Datensätzen zu erzeugen. Auf diesen Geometrien können dann verschiedene Berechnungen angewandt werden. Die 3D Geometrien werden mit Hilfe von Open**CASCADE** (**C**omputer **A**ided **S**oftware for **C**omputer **A**ided **D**esign and **E**ngineering) generiert, einer open source Software Entwicklungsplattform.[7] Der TiGLViewer ist eine Software, welche die TiGL Bibliothek implementiert und somit das Visualisieren von Flugzeugmodellen ermöglicht (siehe Abbildung 2). Die Entwicklung an TiGL wurde 2005 im Rahmen eines Projektes namens **T**echnology **I**ntergration for the **V**irtual **A**ircraft (TIVA) begonnen und seit dem ständig fortgesetzt.[8]

2.2 C++

C++ ist eine objektorientiere weiterentwicklung von C, früher noch unter dem Namen C with Classes (C mit Klassen) bekannt, hat sich C++ heute zu einer der beliebtesten Programmiersprachen weltweit entwickelt. C++ gilt als eine höhere Programmiersprache und der Code muss kompiliert werden.[9] Im Vergleich zu anderen Sprachen wie Java oder Python ist C++ für seine gute Performance bekannt.[10] Daher wird die Sprache in vielen verschiedenen Bereichen eingesetzt: Embedded

[7] [CASCADE] vgl. OpenCASCADE History Absatz 3 und 10
[8] [TIVA] vgl. Abstract
[9] [CPP] vgl. S. 1197 Absatz 2: Introduction
[10] [CPER] siehe S.8 Figure 8: Run-time measurements

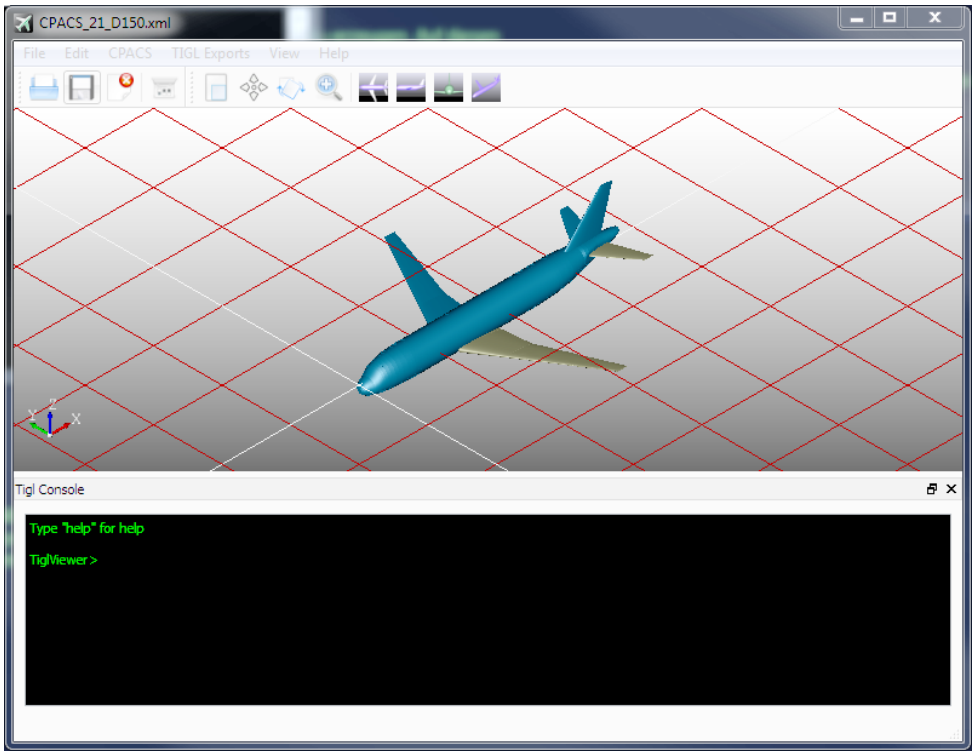


Abbildung 2: Screenshot des TiGLViewer's

Programming, High Performance Computing, Server und Client Anwendungen und auch im Entertainmentbereich.[11]

2.3 CMake

Als Build-System wurde **C**ross-platform **Make** (CMAKE) verwendet, ein open source Build-Tool das seit dem Jahr 2000 besteht und seither ständig weiterentwickelt wurde. Mit CMake ist es leicht möglich auf verschiedenen Plattformen zu entwickeln und zu testen. CMake generiert für jede Plattform ein funktionsfähiges Build-File, dabei passt sich CMake an das zur Verfügung stehende Betriebssystem

[11] [CPPH] vgl Absat 1 - 3

und die Umgebung an. So ist es z.B. möglich ein Projekt genauso leicht auf einem Supercomputer wie auf einem handelsüblichen Windows PC zu bauen. Ziel von CMake ist es das Plattformunabhängige bauen von Software zu ermöglichen und zu unterstützen. Dafür bedarf es lediglich einem einzigen Input, welcher in der CMake Sprache geschrieben wurde. Der Input ist dabei für alle Systeme derselbe, es sind also keine Änderungen an der Datei nötig um den source code auf einem anderen System zu bauen.[12]

2.4 CPACS

Das **C**ommon **P**arametric **A**ircraft **C**onfiguration **S**chema (CPACS) wurde ebenfalls 2005, genau wie TiGL selbst, im Rahmen von TIVA entwickelt. Ziel war es ein System zu schaffen mit dessen Hilfe verschiedene wissenschaftliche Tools aus verschiedenen Abteilungen gekoppelt werden konnte. Genau diese Schnittstelle zwischen den Tools bilden CPACS und TiGL. CPACS ist ein auf XML basierender parametrischer Datensatz, der das zentrale Interface zwischen den einzelnen Tools bildet. TiGL wiederum kann die CPACS Datensätze laden und als drei dimensionales CAD Modell exportieren. Durch das CPACS Format können die verschiedenen Programme nun einheitlich Daten austauschen. Mit CPACS können nicht nur Flugzeuge beschrieben werden, sondern eigentlich nahezu alles das mit Flugzeugen und Luftverkehr zu tun hat. So können z.B. auch Missionen und Flughäfen dargestellt werden. CPACS ist strikt hierarchisch aufgebaut und verfolgt ein zentrales Model, dadurch wird die Anzahl der benötigten Interfaces auf ein Minimum reduziert (Schematisch dargestellt in Abbildung 3).[13] [14]

[12] [CMAKE] vgl. S. 378 Absatz 3 (1.1): The CMake Build Tool und Absatz 4 (1.2): CMake Features
[13] [CPACS] vgl. CPACS A Common Language for Aircraft Design: Simulation and Modeling
[14] [TIVA] vgl. Abstract

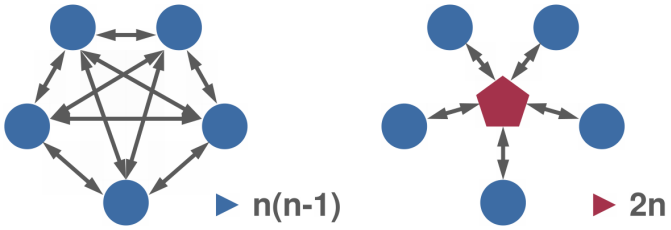


Abbildung 3: CPACS: Central Model Approach[15]

Der Quellcode 1 zeigt einen Ausschnitt einer CPACS Datei, der gezeigte Ausschnitt beschreibt einen *trailingEdgeDevice* (eine Klappe am hinteren Teil des Flügels). Aufgrund der Darstellbarkeit wurden hier einige Parameter ausgelassen.

Quellcode 1: Auschnitt einer CPACS Datei: CPACS D150_VAMP

```
1  <trailingEdgeDevice uID="D150_VAMP_W1_CompSeg1_outerFlap" xsi:type="↩
       trailingEdgeDeviceType">
2      <name>D150_VAMP_W1_CompSeg1_outerFlap</name>
3      <description>Outer flap of the D150</description>
4      <parentUID>D150_VAMP_W1_CompSeg1</parentUID>
5      <outerShape xsi:type="controlSurfaceOuterShapeTrailingEdgeType">
6          <innerBorder xsi:type="controlSurfaceBorderTrailingEdgeType">
7              <etaLE>0.325678045564</etaLE>
8              <etaTE>0.325678045564</etaTE>
9              <xsiLE>0.7195</xsiLE>
10         </innerBorder>
11         <outerBorder xsi:type="controlSurfaceBorderTrailingEdgeType">
12             [...]
13         </outerBorder>
14     </outerShape>
15     <path xsi:type="controlSurfacePathType">
16         <innerHingePoint xsi:type="controlSurfaceHingePointType">
17             <hingeXsi>0.7195</hingeXsi>
18             <hingeRelHeight>0.5</hingeRelHeight>
19         </innerHingePoint>
20         <outerHingePoint xsi:type="controlSurfaceHingePointType">
```

[15] [CPACS] http://software.dlr.de/p/cpacs/screenshot/Central%20Model%20Approach.png

```
21                  <hingeXsi>0.705129411765</hingeXsi>
22                  <hingeRelHeight>0.5</hingeRelHeight>
23              </outerHingePoint>
24              <steps xsi:type="controlSurfaceStepsType">
25                  <step>
26                      [...]
27                  </step>
28              </steps>
29          </path>
30      </trailingEdgeDevice>
```

Besonders deutlich bei diesem Quellcode Ausschnitt wird der XML Charakter von CPACS, jeder Elementtyp hat einen eindeutigen Namen und beinhaltet weitere Elemente und / oder Attribute. Die in den CPACS Dateien dargestellten Elemente tragen dabei die Informationen welche zum Beispiel dazu dienen ein Flugzeug zu beschreiben.

In Abbildung 4 ist eine grobe Übersicht über de CPACS Struktur zu sehen, gut erkennbar ist hier die hierarchische Struktur von CPACS. Die für diese Arbeit relevanten Daten, finden sich alle in dem Element *wings* unter *aircraftModelType* wieder, eine genauere Beschreibung der Datenstruktur folgt im Kapitel 3 *Problemlösung*.

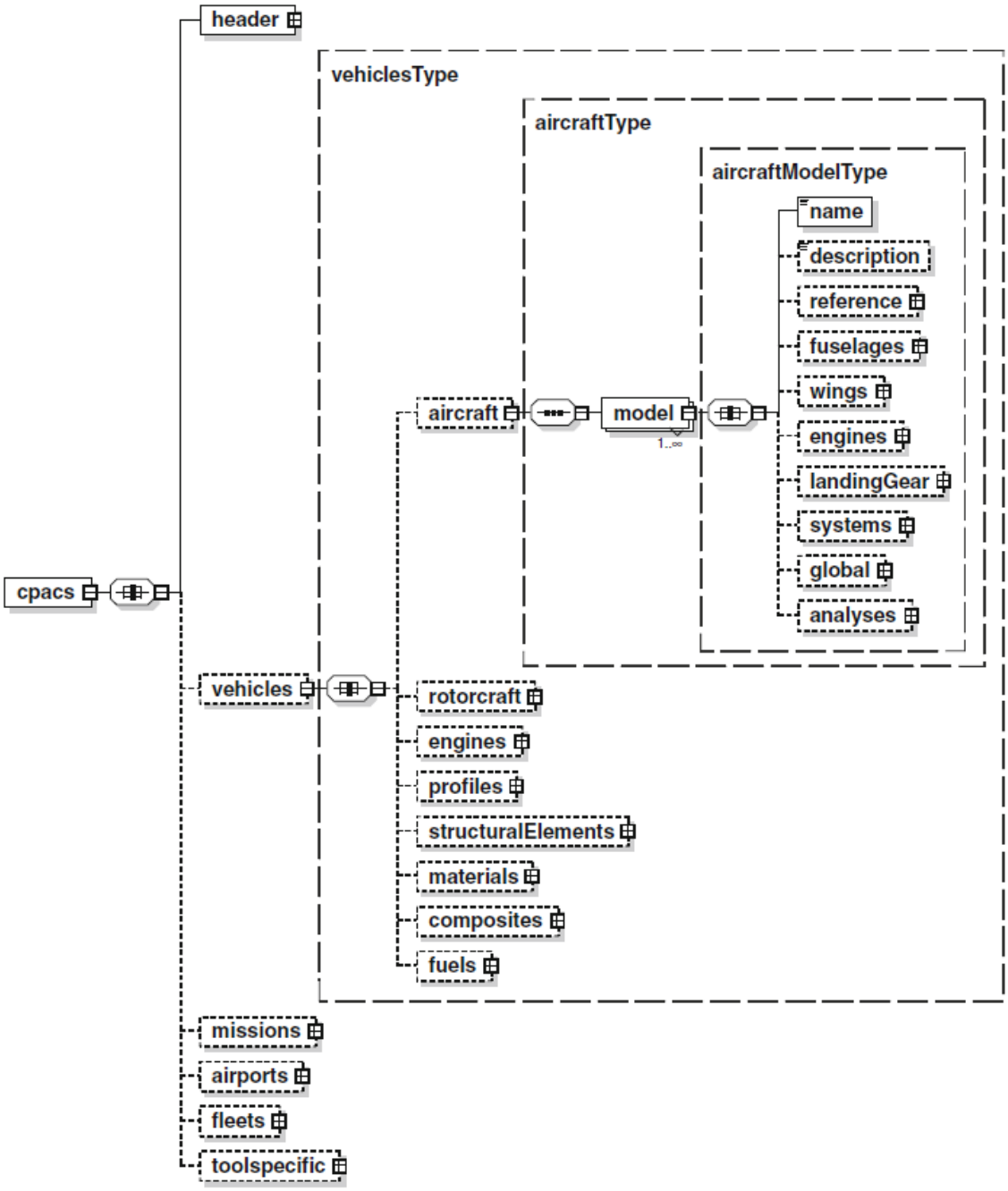


Abbildung 4: CPACS Struktur[16]

[16][TIVA] S.60 Fig. 1 Root structure of the CPACS data format

2.5 OpenCASCADE

Wie bereits erwähnt ist OpenCASCADE eine Abkürzung für Open **C**omputer **A**ided **S**oftware for **C**omputer **A**ided **D**esign and **E**ngineering. OpenCASCADE ist eine open source Plattform zur Software Entwicklung, welche es ermöglicht 3D-Modelle zu erstellen und zu visualisieren. Zuerst ins Leben gerufen wurde OpenCASCADE (damals noch EUCLID CAD) von einer französischen Firma namens Matra Datavision.[17] Seitdem hat sich das System stetig weiterentwickelt und wird heute von vielen sehr angesehenen Firmen aktiv verwendet, wie z.B. BMW, DaimlerChrysler, EADS CCR, Thales und dem DLR.[18]

2.6 TiXI

TiXI ist eine vom DLR entwickelte Bibliothek zum Parsen und Bearbeiten von XML-Dateien. In TiGL wird TiXI unter anderem dafür verwendet um die in den CPACS-Dateien enthaltene Informationen auszulesen und diese abstrakt abzuspeichern. Die verwendete Syntax ist sehr weit abstrahiert, so gibt es z.B. einfachste Befehle um einen ganz bestimmten Wert aus einer sehr großen XML-Datei auszulesen. TiXI ist daher ein sehr nützliches Tool, wenn es um die Arbeit mit XML-Dateien geht.[19]

[17] [CASCADE] vgl. OpenCASCADE History Absatz 1 - 3
[18] [CASCU] vgl. OpenCASCADE Customers
[19] [TiXI] vgl. TiXI Wiki: TiXI

2.7 Qt Development Frameworks

Das **G**raphical **U**ser **I**nterface (GUI) des TiGLViewers wird mit Hilfe des QT Development Frameworks erstellt. QT ist eine plattformunabhängige C++ Bibliothek um graphische Benutzeroberflächen für Programme zu erstellen. QT ist inzwischen sehr weit verbreitet und wird von vielen namensträchtigen Projekten und Firmen aktiv eingesetzt. Programme wie der VLC - Media Player, Skype oder Firmen wie ABB, Thales und die Deutsche Flugsicherung profitieren vom QT Development Framework.[20] [21] [22]

Neben den grafischen Aspketen von QT, gibt es auch noch umfangreiche Bibliotheken für verschiedenste Probleme. So kann QT z.B. auch für weitere Aufgaben, wie Datenbankzugriffe, genutzt werden.[23] Mit QT können leicht die bekanntesten Design-Patterns, wie z.b. Dem Model-View-Controller Pattern umgesetzt werden.[24]

QT wurde zunächst von der Firma Trolltech entwickelt. Im Jahr 2008 wurde das Projekt von Nokia aufgekauft. Nokia wollte die plattformunabhängige Entwicklung des Toolkits beschleunigen und für mobile Endgeräte benutzen. Durch die Übernahme des Projekts von Nokia wurde das Projekt unter der GNU's **L**esser **G**eneral **P**ublic **L**icense (LGPL) neu lizensiert.[25] Inzwischen wurde QT an die finnische Firma Digia abgegeben.[26]

[20] [QTWT] vgl. S. 2 - 7
[21] [QTIU] vgl. Qt in Use und Qt Showcases
[22] [QTAF] vgl Folie 5, Who uses QT
[23] [GACO] vgl. Überblick über das Qt-Framework Absatz 2
[24] [QTB] vgl. Beginning Qt Development, S. 59 Absatz 1 und 2
[25] [QTN] vgl. die Absätze 1-3
[26] [QTA] vgl. Quick History
[27] Quelle: http://www.tizenexperts.com/wp-content/uploads/2013/05/Qt-Tizen-TizenExperts.jpg

Abbildung 5: The QT-Project[27]

2.8 Google C++ Testing Framework

Als Basis um Softwaretests zu entwickeln und durchzuführen wird innerhalb von TiGL das *Google C++ Testing Framework*, kurz *Google Test* oder auch *gtest* verwendet. Google Tests sind weitestgehend Plattformunabhängig. So sind google Tests problemlos lauffähig unter Linux, Mac OS X, Windows, Cygwin und noch vielen weiteren Betriebssystemen.[28] Durch eine sehr hohe Abstraktionsschicht ist es mit Hilfe von gTest möglich sehr einfach, schnell und zielgerichtet Tests zu entwickeln.
Einige Grundaspekte haben maßgeblich zur Entwicklung von gTest beigetragen:[29]

- Unabhängigkeit und Wiederholbarkeit
- Tests sollten den getesteten Code widerspiegeln
- Zusätzliche Informationen bei fehlgeschlagenen Tests
- Leicht zu benutzen und schnell

[28] [MOFO] vgl. GTest introduction
[29] [GOO] vgl. Introduction: Why Google C++ Testing Framework?

Mehrere einzelne Tests können zu komplexeren Testfällen oder Testszenarien zusammengefasst werden. Dadurch ist es möglich sehr nahe am eigentlichen Code zu testen und dessen Struktur widerzuspiegeln. Das Google C++ Testing Framework hat eine sehr große Community und wurde / wird sehr ausführlich dokumentiert.[30]

Quellcode 2: Beispiel GoogleTest

```
1 #include "gtest/gtest.h"
2
3 TEST(TrailingEdgeDevice, getNormalOfTrailingEdgeDevice) {
4     [...]
5     gp_Vec normalTED = gp_Vec(props.Normal().XYZ());
6     gp_Vec normalWCS = trailingEdge.getNormalOfTrailingEdgeDevice();
7
8     ASSERT_NEAR(normalTED * normalWCS,0,1e-7)
9 }
```

Quellcode 2 zeigt einen simplen Test der mittels Google Test realisiert wurde. Im gezeigten Beispiel wird geprüft ob zwei Vektoren orthogonal zueinander stehen. Dafür wird geprüft ob das Skalarprodukt der beiden Vektoren Null (mit einer gewissen Toleranz) ergibt . Weitere Codebeispiele und Erklärungen zum Google C++ Testing Framework folgen in Kapitel 4.

[30] [IBM] vgl. Why use the Google C++ Testing Framework?

3 Problemlösung

Im nun folgenden Kapitel werden die Problemlösung und die eigentlichen Umsetzung der geschilderten Aufgabe klar dargestellt und erläutert. Dazu gehören Aufgaben wie dem Einlesen von CPACS Daten, dem Modellieren und Transformieren von Landeklappen und Kontrollflächen und dem Visualisieren im TiGLViewer.

3.1 Anforderungsdefinition

Um das Problem strukturiert und möglichst effizient zu lösen, war zunächst eine genau Anforderungsdefinition nötig. Hier wurde festgelegt, was genau die Anforderungen an die zu ergänzenden Features sein sollten und wie diese zu bewerten sind. Auf eine Beschreibung der Anforderungen durch Use Cases (nach Alistair Cockburn)[31] wurde gezielt verzichtet, da die Aufgabenstellung sehr genau formuliert ist und kaum Raum für Fehlinterpretationen von Anforderungen zulässt.

3.1.1 Ermittlung / Analyse

Die Anforderungen spiegeln sich in der gestellten Aufgabe wieder, folgende Anforderungen können aus der Aufgabenstellung abgeleitet werden:

- **Datenstruktur:** Die nötige Datenstruktur, um die zusätzlich benötigten CPACS Daten in ein logisches Datenmodel zu überführen, muss entwickelt und implementiert werden.

[31] Use Cases beschreiben Anwendugsfälle der Mensch-Maschine-Interaktion detailliert. Die Funktionalität des Gesamtsystems soll so korrekt und verständlich dargestellt werden. Use Cases sind inzwischen eine weit verbreitete Methode um Anwendugsfälle aus Sicht des Kunden zu beschreiben um so Fehlinterpretationen zu vermeiden.

- **Einlesen von Daten:** CPACS Daten müssen ausgelesen werden können und in die erzeugten Datenmodelle überführt werden.
- **Erzeugen der Geometrien:** Die drei dimensionalen Klappengeometrien müssen aus den Daten erzeugt werden.
- **Verfahrwege:** Die Klappen sollen verschiedene, wie in den CPACS Daten beschriebene, Transformationen durchführen können.
- **Visualisierung:** Die neu erzeugten Flügelgeometrien sollen im TiGLViewer visualisiert werden.
- **Export:** Optional - Die 3D Modelle sollen als verschiedene Dateiformate exportiert werden können.

3.1.2 Strukturierung und Abstimmung

Die Folgende Abarbeitungsreihenfolge für die Aufgaben und Anforderungen ergibt sich Aufgrund von Abhängigkeiten unter den einzelnen Anforderungen und deren Wichtigkeit für den Kunden. Einzelne Anforderungen wurden teilweise zu Paketen zusammengefasst, da eine simultane Entwicklung dieser Anforderungen den Arbeitsfluss begünstigt.

- **1. Datenstruktur** und **Einlesen der Daten**
- **2. Erzeugen der Geometrien** und **Visualisierung**
- **3. Verfahrwege**
- **4. Export**

Die Datenstruktur und das Lesen der Daten muss offensichtlich funktionsfähig sein, damit es überhaupt möglich ist die Daten aus den CPACS Datein zu extrahieren. Diese Daten werden dann unter anderem für die Geomtriegenerierung gebraucht. Deswegen wurden diese Anforderungen an erster Stelle bearbeitet. Das Erzeugen der Geometrien und deren Visualisierung wurde vor der Implementation der Verfahrwege bearbeitet. Dadurch wird der Fortschritt bei den Entwicklungen schneller sichtbar. Desweiteren wird durch die Möglichkeit die Geometrien zu visualisieren das Debuggen und Implementieren der Verfahrwege extrem vereinfacht. Da der Export eine optionale Anforderung ist, wird dieser erst nach Abarbeitung aller anderer Anforderungen realisiert.

3.1.3 Anforderungsbewertung

Die Anforderungen sind unterschiedlich gewichtet, oberster Priorität hat das Erzeugen der Geometrien, wie oben bereits erwähnt, ist hierzu allerdings eine vernünftige Datenstruktur erforderlich. Die Implementierung der Verfahrwege hat ebenfalls eine sehr hohe Priorität, da ohne diese ein Betrachten von verschiedenen Stellwinkeln der Klappen bei Simulationen nicht möglich ist. All diese Anforderungen sind untereinander widerspruchsfrei realisierbar und von der Aufgabenstellung gefordert.

Die zusätzlichen Features beeinflussen die bereits vorhandenen Funktionen von TiGL und dem TiGLViewer nicht, daher können die durchgeführten Änderung nur zu einer Steigerung des Funktionsumfangs von TiGL führen.

3.2 Erstellen der benötigten Datenstruktur

Um die eben genannten Anforderungen zu realisieren, war es nötig einige zusätzliche Klassen in die TiGL Bibliothek einzubinden. Die neu erstellten Klassen spiegeln sich zum größten Teil in den CPACS Definitionen wieder. In Abbildung 6 wird eine grobe Übersicht über den Aufbau von Kontrollflächen in CPACS dargestellt. Grün markiert sind Elemente, welche zur Lösung des Problems im Rahmen dieser Arbeit erstellt wurden. Orange eingefärbte Elemte sind hierbei atomare Daten und Werte, diese werden letztendlich benötigt um die 3D Geometrien zu modellieren und um die Verfahrwege zu berechnen. ControlSurfaces ist ein Element von componentSegment (in Grafik nicht dargesellt), die Klasse componentSegment existiert bereits und ist Ausgangspunkt der hier dargestellten Erweiterungen.

Besonders wichtig sind alle Elemente rund um *trailingEdgeDevice*, da diese den Großteil der Klassen und Elemente enthalten, die im Rahmen dieser Arbeit erstellt wurden. *Spoilers* und *leadingEdgeDevices* haben eine ähnliche bis fast identische Struktur wie ein *trailingEdgeDevice*. Ein *trailingEdgeDevice* beschreibt Landeklappen und Kontrollflächen an der Hinterseite von Flügeln und *leadingEdgeDevices* sind Klappen an der Vorderseite.

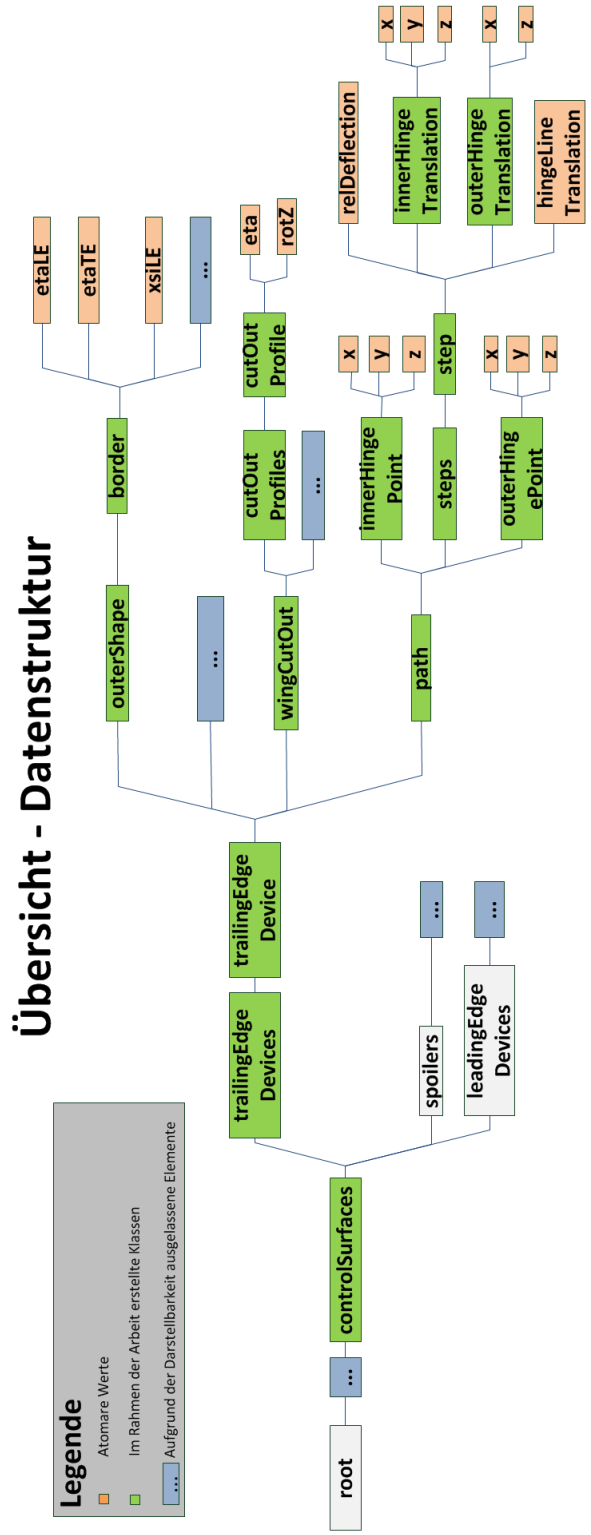


Abbildung 6: Schematische Darstellung der neu erstellten Datenstrukturen

3.3 CPACS Daten lesen

Um die notwendigen Werte aus den CPACS Daten einzulesen, wurde für jede erstellte Klasse, welche sich in dem CPACS Schema wiederfindet, eine ReadCPACS Methode geschrieben. Die Methode liest alle Unterelemente eines Elementes ein. Falls ein Unterelement nicht atomar ist wird bei diesem Element ebenfalls die ReadCPACS Methode aufgerufen. Es hat also jedes nicht atomare Element eine ReadCPACS Methode, welche vom dazugehörigen Elternelement beim Laden einer CPACS Datei, aufgerufen wird.

Quellcode 3: Beispiel einer ReadCPACS Methode

```
1  void CCPACSWingComponentSegment::ReadCPACS(TixiDocumentHandle tixiHandle, ↩
       const std::string& segmentXPath)
2  {
3      [...]
4
5      // getting Element Name
6      char* ptrName = NULL;
7      tempString    = segmentXPath + "/name";
8      elementPath   = const_cast<char*>(tempString.c_str());
9      if (tixiGetTextElement(tixiHandle, elementPath, &ptrName) == SUCCESS)
10         name          = ptrName;
11
12     // getting Element ControlSurfaces
13     tempString = segmentXPath + "/controlSurfaces";
14     elementPath = const_cast<char*>(tempString.c_str());
15     if (tixiCheckElement(tixiHandle,elementPath) == SUCCESS)
16         controlSurfaces->ReadCPACS(tixiHandle, elementPath);
17
18     [...]
19 }
```

In Quellcode 3 ist ein Ausschnitt der Methode *ReadCPACS* der Klasse *CCPACSWingComponentSegment* dargestellt. Die Methode bekommt zwei Variablen als Parameter übergeben, *tixiHandle* und *segmentXPath*. Der *tixiHandle* wird für die Anwendung der TiXI-Methoden benötigt. Im *segmentXPath* ist der Pfad zu dem aktuellen Element enthalten. Jedes Element führt die *ReadCPACS* Methode seiner Unterelemente aus und übergibt diesen dabei den auf diese Elemente angepassten *elementPath*, hier zu sehen in Zeile 16. Dort wird bei dem Unterelement *controlSurfaces* die *ReadCPACS* Methode aufgerufen. In dem gezeigten Ausschnitt werden zwei Elemente geladen, *controlSurfaces* und der Name des *CCPACSWingComponentSegments*. Ebenfalls im Quellcode sichtbar sind die Überprüfungen die mit Hilfe von TiXI im Vorfeld gemacht werden, hier in Zeile 9 und 15. So werden Fehler, wenn zum Beispiel die CPACS Datei beschädigt oder unvollständig ist vermieden.

3.4 Modellierung der Kontrollflächen

Da nun alle benötigten Daten in abstrakten Modellen wiedergegeben werden können und die Daten durch die erstellten *ReadCPACS* Methoden auch aus den CPACS Dateien geladen werden können, ist es nun möglich die Geometrien der Landeklappen und Kontrollflächen zu modellieren. Da die genaue Form von Kontrollflächen sehr komplex sein kann, werden im ersten Modellierungsschritt zunächst nur sehr simple Geometrien aus den CPACS Daten erstellt. Durch die zunächst sehr einfach Modellierung ist es auch schneller möglich die Transformation der Klappen zu testen und umzusetzen.

Zur Modellierung: Um ein einfaches Modell der Klappen zu erstellen sind nicht viele Daten nötigt. Alles was benötigt wird sind die vier Eckpunkte der Landeklappe auf der Oberseite des Flügels. Diese werden in *outerShape* beschrieben, ein Element der Klasse *CCPACSTrailingEdgeDevice*. Das *outerShape* beinhaltet eine *outerBorder* und eine *innerBorder*, dort sind jeweils zwei Punkte gespeichert, welche relativ zu einem *CCPACSWingComponentSegment* (einer logischen Zusammenfassung von mehreren Flügelsegmenten), die innere und äußere Kante der Klappen beschreiben. Die eigentliche Modellierung wird mithilfe von OpenCascade realisiert, OpenCascade bietet verschiedene Funktionen um 3D Geometrien zu erstellen und mit diesen zu interagieren.

Der gesamte Modellierungsprozess lässt sich in den folgenden Schritten zusammenfassen:

- **Relative Koordinaten in Weltkoordinaten umwandeln:** Die in den CPACS-Daten gespeicherte Koordinaten sind relativ angegeben. Für die 3D Modellierung müssen diese in Weltkoordinaten umgerechnet werden. Üblich ist hier die Verwendung von sogennanten *eta/xsi* - Koordinaten. Durch diese können Punkte auf der Flügelgeometrie angegeben werden. Der *eta* Wert ist eine Spannweitenkoordinate und der *xsi* Wert eine Flügeltiefenkoordinate. Veranschaulicht ist dies in Abbildung 7 zusehen.

- **Face generieren:** Aus den vier Eckpunkten wird ein Face (ein topologisches Element, welches eine Fläche im dreidimensionalen Raum beschreibt) erzeugt.

- **Cut-out-shape erstellen:** Das Face wird nun benutzt um ein Prisma zu erzeugen, diese wird später dazu benutzt um die Klappen aus dem Flügel aus zuschneiden.

- **Klappen-Geometrie generieren:** Die Klappen und auch die Flügelgeometrie (ohne Klappen) werden mithilfe von boolschen Operationen mit dem Flügel und dem Cut-out-shape berechnet.

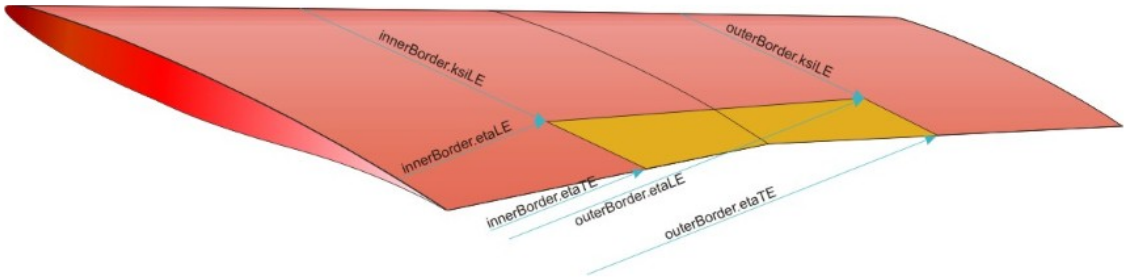


Abbildung 7: Eta / Xsi Koordinaten[32]

Um zu den gegebenen relativen Koordinaten die dazugehörigen Weltkoordinaten zu berechnen wird eine bereits vorhandene Methode *GetPoint* von *CCPACSWingComponentSegment* benutzt. Diese Funktion nimmt die gegebenen *eta/xsi* - Koordinaten und berechnet daraus die dazugehörigen Weltkoordinaten. Zurück gibt die Funktion einen Punkt im dreidimensionalen Raum. In Quellcode 4 ist dargestellt, wie die *GetPoint* Methode verwendet wird, um die vier Eckpunkte zu berechnen.

Quellcode 4: Aussschnitt der Methode getFace() von CCPACSTrailingEdgeDevice

```
1 CCPACSTrailingEdgeDeviceBorder outerBorder = getOuterShape().getOuterBorder();
2 gp_Pnt p1 = _segment->GetPoint(outerBorder.getEtaLE(),outerBorder.getXsiLE());
3 gp_Pnt p2 = _segment->GetPoint(outerBorder.getEtaTE(),1.0f);
4
```

[32] [CDOC] Remarks, Element outerShape bei TrailingEdgeDevice

```
5 CCPACSTrailingEdgeDeviceBorder innerBorder = getOuterShape().getInnerBorder();
6 gp_Pnt p3 = _segment->GetPoint(innerBorder.getEtaLE(),innerBorder.getXsiLE());
7 gp_Pnt p4 = _segment->GetPoint(innerBorder.getEtaTE(),1.0f);
```

Als Parameter benötigt die *GetPoint* Methode die relativen Koordinaten. Eine dreidimensionale Weltkoordinate berechnet sich aus einem *xsi* und einem *eta* Wert. Der *xsi* Wert gibt die relative Entfernung des Punktes zur führenden Kante des Flügels an. Der *eta* Wert gibt die relative Entfernung des Punktes zur inneren Seitenkante des Flügels an (Spannweitenkoordinate und Flügeltiefenkoordinate). Beide Werte haben eine Reichweite von 0 bis 1 und spiegeln so einen Punkt auf der Oberseite des Flügels im dreidimensionalen Raum wieder.

Da nun die vier Punkte, welche die Oberfläche der Landeklappe auf der Oberseite des Flügels definieren berechnet sind, kann daraus ein Face erzeugt werden. Ein Face ist ein topologisches Element der OpenCascade Bibliothek. Ein topologisches Objekt ist eine Komposition von mehreren primitiven geometrischen Elementen (Linien, Punkte, Kurven, Flächen, ...). Das Face definiert eine Fläche im dreidimensionalen Raum durch ein darunter liegendes Wire. In diesem Fall wird das Wire durch einige Zwischenschritte direkt aus den vier Punkten erzeugt. Das Wire kann dann direkt ein Face umgewandelt werden. Was genau ein Wire ist und wie die Umwandlung funktioniert zeigt Quellcode 5 und der darauf folgende Absatz.

Quellcode 5: Erstellen der Oberfläche von Klappen

```
1 TopoDS_Edge edge1 = BRepBuilderAPI_MakeEdge(gp_Pnt(p1.XYZ()),gp_Pnt(p2.XYZ()));
2 TopoDS_Edge edge2 = BRepBuilderAPI_MakeEdge(gp_Pnt(p2.XYZ()),gp_Pnt(p4.XYZ()));
3 TopoDS_Edge edge3 = BRepBuilderAPI_MakeEdge(gp_Pnt(p3.XYZ()),gp_Pnt(p4.XYZ()));
4 TopoDS_Edge edge4 = BRepBuilderAPI_MakeEdge(gp_Pnt(p3.XYZ()),gp_Pnt(p1.XYZ()));
5
```

```
6 TopoDS_Wire wire = BRepBuilderAPI_MakeWire(edge4,edge3,edge2,edge1);
7 TopoDS_Face face = BRepBuilderAPI_MakeFace(wire);
```

In den Zeilen 1 - 4 werden aus den einzelnen Punkten Edges erzeugt, diese geben die Kanten für das zukünftige Face an. Ein Edge beschreibt eine Kurve mit definiertem Anfang und Ende. Aus den Edges wird in den Zeilen 6 und 7 schließlich das Wire erzeugt. Ein Wire besteht somit aus mehren zusammengesetzten Kurven. Anschließend wird aus den erstellten Edges dann das Face erzeugt. Das Wire ist in diesem Fall eine Verbindung von allen Kanten des nun entstandenen flächenlosen Vierecks.

In einer später durchgeführten Optimierung wurde die Funktion dahingehen erweitert, dass das Face nicht direkt aus den vier berechneten Punkten der *GetPoint* Methode erstellt wird, sondern dass diese Punkte zunächst auf eine Ebene projiziert werden, welche parallel zum Flügel verläuft. Dadurch wird die Perfomance von später benötigten OpenCascade Funktionen merklich verbessert.

Aus dem nun erstellten Face kann durch einen Verschiebungsvektor ein Prisma erzeugt werden. Wie dies genau funktioniert zeigt der Quellcode 6.

Quellcode 6: Prismaerstellung aus TopoDS_Face

```
1 gp_Vec vec = getNormalOfControlSurfaceDevice();
2 if ( _type == SPOILER ) {
3     vec.Multiply(determineSpoilerThickness()*2);
4 } else {
5     vec.Multiply(determineCutOutPrismThickness()*2);
6 }
7
```

```
8 TopoDS_Shape prism = BRepPrimAPI_MakePrism(face, vec);
```

In Zeile 1 wird der Richtungsvektor erstellt, dieser gibt an in welche Richtung das Face sozusagen ausgedehnt wird. Die Klappe soll sich in Richtung des Normalenvektors der Grundfläche ausdehnen. Danach wird die nötige Dicke der CutOutBox berechnet, unterschieden wird hier zwischen Spoilern und allen anderen Klappentypen. Spoiler haben nämlich eine spezielle Dicke, da diese nicht den kompletten Flügel durchschneiden. In Zeile 8 wird dann aus dem Face mithilfe des Richtungsvektors ein *TopoDS_Shape* erzeugt, in diesem Fall ist das Shape ein rechteckiges Prisma, welches so dimensioniert ist, sodass es genau an den vier Eckpunkten der Klappen den Flügel durchstößt. Veranschaulicht wird dies in Abbildung 8.

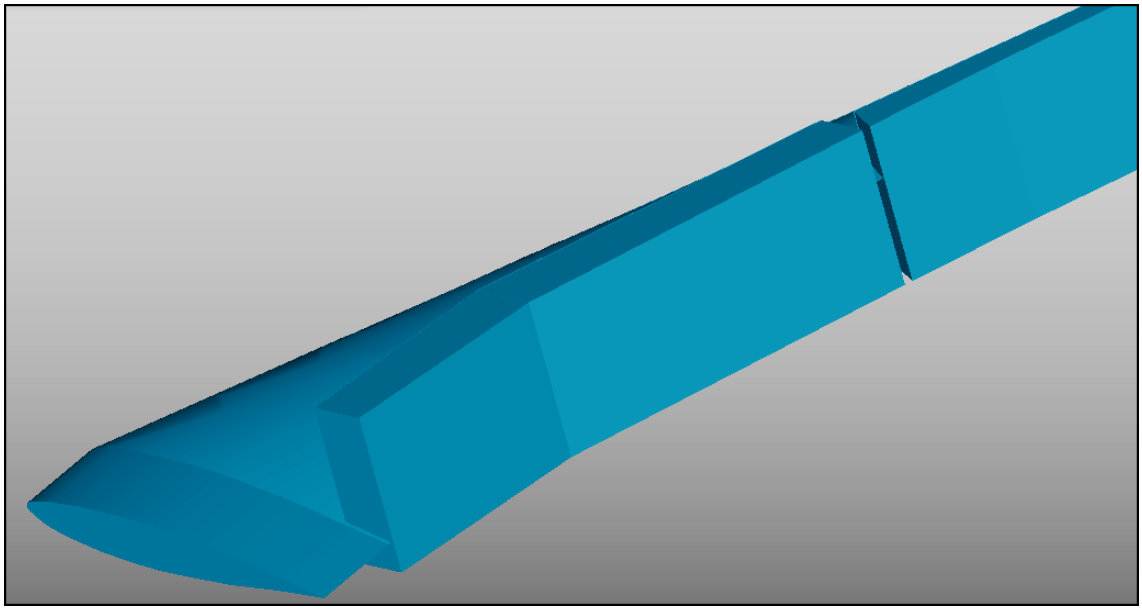


Abbildung 8: Veranschaulichung der *Cut-out-shape*

Mithilfe des erstellten Rechtecks kann nun durch die boolschen Operationen *cut* und *common* von der OpenCascade Bibliothek die benötigte Klappengeometrie sowie die übrig bleibende Flügegeometrie ohne die Klappen berechnet werden.

Quellcode 7: Schnittflächenberechnung zweier TopoDS_Shapes in OpenCASCADE

```
1  // box built out of the 4 ´edges of the trailingEdgeDevice outer shape.
2  TopoDS_Shape trailingEdgePrism = trailingEdgeDevice.getCutOutShape();
3
4  // create intermediate result for boolean ops
5  BOPCol_ListOfShape aLS;
6  aLS.Append(trailingEdgePrism);
7  aLS.Append(wingLoft);
8  BOPAlgo_PaveFiller dsFill;
9  dsFill.SetArguments(aLS);
10 dsFill.Perform();
```

In Zeile 2 des Quellcodes 7 wird das eben beschriebene Rechteckobjekt erstellt. Danach wird in Zeile 5 ein Container für Shapes erzeugt indem auch gleich die Rechteckgeometrie (*trailingEdgePrism*) und die Flügelgeometrie (*wingLoft*) abgelegt wird. In den Zeilen 8 - 10 werden die Schnittflächen der zwei angefügten Geometrien berechnet.

Quellcode 8: Berechnung der entgültigen Klappengeometrie

```
1  // create common and cut out structure of the wing and the trailingEdgeDevice
2  wingLoftCut  = BRepAlgoAPI_Cut(wingLoft, trailingEdgePrism, dsFill);
3  trailingEdgeLoftCut = BRepAlgoAPI_Common(trailingEdgePrism, wingLoft, dsFill);
```

Der Quellcode 8 zeigt nun wie die entgültige Geomtrie der Klappen und der restlichen Flügelgeometrie erzeugt wird. In Zeile 2 wird der Restflügel und in Zeile 3 die gemeinsame Fläche, also die Klappe berechnet. Die neuen Geometrien werden

mithilfe der boolschen Operationen *BRepAlgoAPI_Cut* und *BRepAlgoAPI_Common* bestimmt, diese liefern eine neue Geometrie zurück und zwar die gemeinsame Menge, bzw. die Differenz der Menge. Die gemeinsame Menge, also das gemeinsame Shape, bildet nun die Landeklappe, bzw. die Kontrollfläche.

In Abbildung 9 ist die nun erstellte Klappengeometrie sichtbar gemacht, im unteren Teil wurde das Höhenruder entfernt um die verbleibende neue Flügelgeometrie zu veranschaulichen.

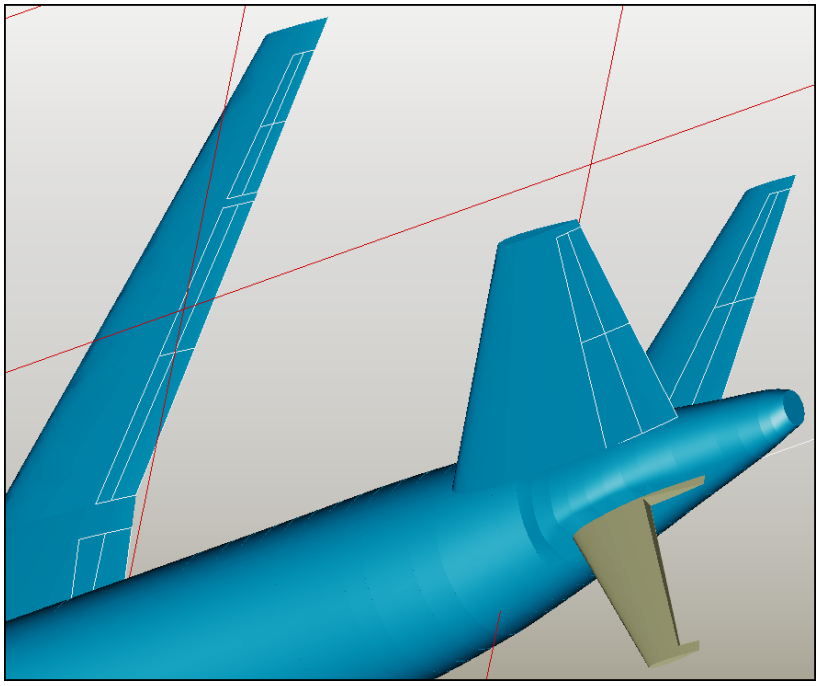


Abbildung 9: Klappengeometrie und Höhenruder ohne Klappe[33]

[33]Screenshoot des TiGLViewers

3.5 Verfahrwege und Klappenrotation

Die Transformation welche die Verfahrwege der Klappen anhand der CPACS Daten passend verschiebt, rotiert und positioniert ist wie folgt definiert:

$$T(\vec{p}_{trans}, \phi, \theta, \alpha) = T(\vec{p}_{trans}) * K^{-1} * R(\phi) * R(\theta) * R(\alpha) * K$$

Formel 1: Gesamttransformation

In diesem Absatz wird nun genauer erläutert, wie diese Transformation zustande kommt, woher die Daten kommen und warum eine solch komplexe Rechnung notwendig ist.

Um die Transformation durchzuführen werden einige Parameter benötigt, teils stammend aus den CPACS-Daten und teilweise von Benutzereingaben. Benötigt werden vier Punkte: *innerHingePoint, outerHingePoint, innerHingePointT, outerHingePointT*. Diese Punkte bilden die Rotationsachse vor und nach der Transformation. Um genau diese Achse, die sogenannte *HingeLine*, soll die Klappe rotieren. Die zwei Punkte vor der Transformation sind direkt in der CPACS Datei abgelegt, die transformierten Punkte lassen sich mithilfe eines in den CPACS Daten angegebenen Verschiebungsvektors berechnen. Nun ist die alte und die neue (nach der Verschiebung) Rotationsachse bekannt, allerdings nur in Weltkoordinaten.

3.5.1 Definition der Verfahrwege und Rotationen in CPACS

Abbildung 10, ein Auszug einer CPACS-Datei, zeigt wie der Pfad innerhalb von CPACS definiert ist.

```
<path xsi:type="controlSurfacePathType">
    <innerHingePoint xsi:type="controlSurfaceHingePointType">
        <hingeXsi>0.812922199791</hingeXsi>
        <hingeRelHeight>0.5</hingeRelHeight>
    </innerHingePoint>
    <outerHingePoint xsi:type="controlSurfaceHingePointType">
        <hingeXsi>0.7195</hingeXsi>
        <hingeRelHeight>0.5</hingeRelHeight>
    </outerHingePoint>
    <steps xsi:type="controlSurfaceStepsType">
        <step xsi:type="controlSurfaceStepType">
            <relDeflection>0</relDeflection>
            <innerHingeTranslation>
                <x>0.0</x>
                <y>0.0</y>
                <z>0.0</z>
            </innerHingeTranslation>
            <outerHingeTranslation>
                <x>0.0</x>
                <z>0.0</z>
            </outerHingeTranslation>
            <hingeLineRotation>0</hingeLineRotation>
        </step>
        <step xsi:type="controlSurfaceStepType">
            <relDeflection>1</relDeflection>
            <innerHingeTranslation>
                <x>1.0</x>
                <y>0.0</y>
                <z>0.0</z>
            </innerHingeTranslation>
            <outerHingeTranslation>
                <x>1.0</x>
                <z>0.0</z>
            </outerHingeTranslation>
            <hingeLineRotation>35.0</hingeLineRotation>
        </step>
    </steps>
</path>
```

Abbildung 10: Bewegungspfad definition in CPACS[34]

Im Unterelement *path* wird zunächst ein *innerHingePoint* und ein *outerHingePoint* definiert. Diese sind in der folgenden Grafik 11 als grüne Kreise markiert. Die grüne Linie bezeichnet die durch die zwei Punkte entstandene *HingeLine*. Der Pfad enthält außerdem noch eine Liste von *steps*. Ein *step* beschreibt jeweils einen Verschiebungsvektor für inner und outer *HingePoint*, sowie eine Rotation α. Wie sich diese Werte auf die eigentliche Transformation auswirken zeigt die Grafik 11. Die Klappen wird bei einer gegebenen *deflection* von eins um eins in X-Richtung verschoben und um 35 Grad um die HingeLine rotiert.

[34] Ausschnitt aus dem CPACS Datensatz: CPACS_21_D150.xml

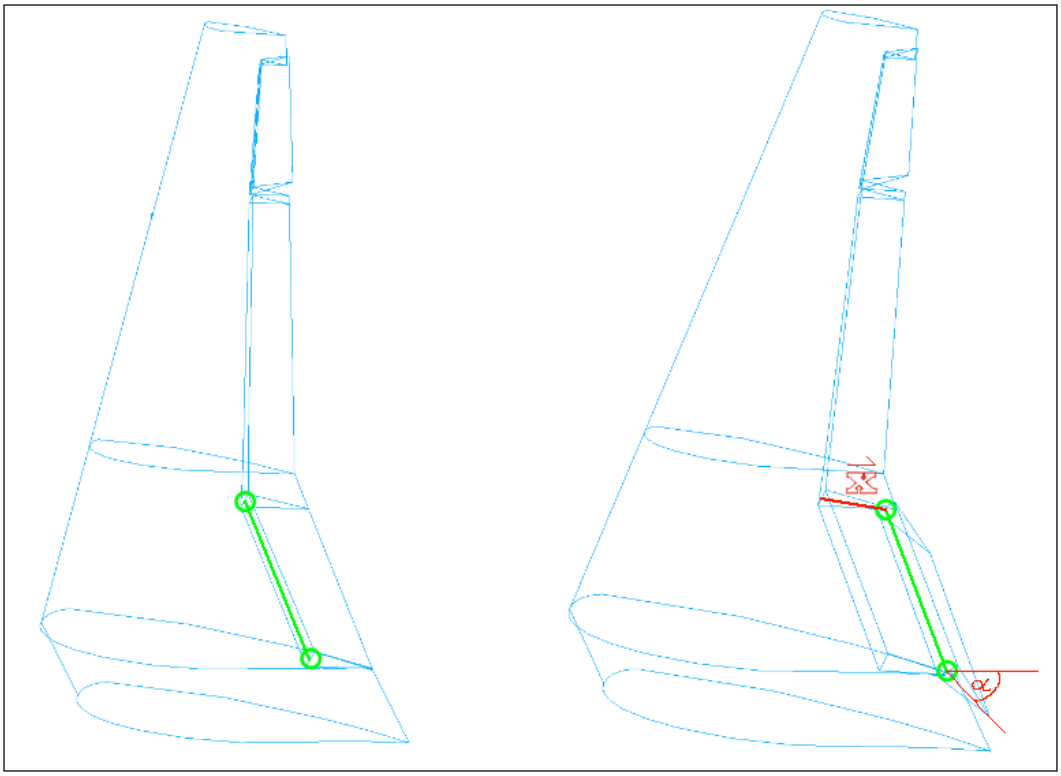


Abbildung 11: Visualisierung der Verfahrwege[35]

Jeder *step* generiert eine Art Wegpunkt, der zu einer dazugehörigen *deflection* den Verschiebungsvektor und die Rotation beschreibt.

Es sind nur Deflections von der minimal definierten Deflection bis zur maximal definierten Deflection möglich. Zu angegebenen Deflections, die zwischen zwei definierten Werten liegen, werden mit Hilfe von linearer Interpolation die Verschiebungsvektoren und die Rotation bestimmt.

[35] Aus Screenshoots des TIGL-Viewers erstellte Grafik

Siehe dazu folgendes Beispiel:

> *Wenn bezogen auf die in Abbildung 10 definierte Landeklappe eine gewünschte Deflection von 0.5 angegeben würde, dann würde sich dich Klappe um 0.5 in X-Richtung verschieben und um 17.5 Grad um die HingeLine rotieren.*

3.5.2 Homogene Koordinaten

In den folgenden Berechnungen werden alle Transformationsmatrizen in homogenen Koordinaten dargestellt. Das ist ein übliches Vorgehen in der Computergrafik und beruht darauf, dass alle möglichen Transformationen durch Matrixmultiplikationen dargestellt werden sollten. Die Multiplikation ist eine sehr gut optimierte Rechenoperation in modernen Prozessoren und kann dadurch in sehr kurzer Zeit von einem Computer berechnet werden. Wie im folgenden Kapitel zu sehen ist, können tatsächlich alle fundamentalen Transformationen (Translation, Rotation, Skalierung und noch mehr) durch eine Matrixmultiplikation dargestellt werden. Genau dafür sind allerdings homogene Koordinaten notwendig, denn die Translation könnte ansonsten nicht durch eine Matrixmultiplikation dargestellt werden.

Sei $\vec{p}$ ein dreidimensionaler Vektor, dann würde sich eine Verschiebung von $\vec{p}$ um den Verschiebungsvektor $\vec{t}$ mit herkömmlichen Koordinaten wie folgt berechnen:

$$\vec{p_t} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ p_3 + t_3 \end{pmatrix}$$

Formel 2: Darstellung einer Translation ohne homogene Koordinaten

Deutlich zu erkennen ist hier, dass die Translation nicht durch eine Matrixmultiplikation sondern durch eine Vektoraddition von zwei Vektoren dargestellt wird. Gewünscht ist allerdings eine Matrixmultiplikation, da so alle Transformationen einheitlich durch Matrixmultiplikationen darstellbar wären.

Die Idee besteht darin, die Vektoraddition der Translation irgendwie durch eine Matrixmultiplikation darzustellen. Dazu wird ein einfacher Trick angewandt, die Vektoren werden um eine zusätzliche vierte Dimension erweitert (Matrizen bekommen eine zusätzliche Spalte und Zeile). Jetzt ist es ganz leicht möglich die Translation als Matrixmultiplikation darzustellen:[36]

$$\vec{p_t} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} p_1 + t_1 \\ p_2 + t_2 \\ p_3 + t_3 \\ 1 \end{pmatrix}$$

Formel 3: Darstellung einer Translation mit homogenen Koordinaten

3.5.3 Bestimmung der Transformationsmatrix zum lokalen Koordinatensystem

Es ist leichter eine Transformation / Rotation entlang der Rotationsachse durchzuführen, wenn die Punkte in einem lokalen Koordinatensystem vorliegen, welches eine Achse direkt mit der Rotationsachse, der *HingeLine* teilt. Die restlichen zwei Achsen sollten orthogonal angeordnet sein. Die Transformation zum eben genannten lokalen Koordinatensystem findet sich in der oben genannten Gleichung unter der

[36] [WVSG] siehe Translation (Verschiebung)

Bezeichnung K wieder. K^{-1} ist dementsprechend die Rücktransformation zu dem Weltkoordinatensystem. Um K vollständig zu beschreiben werden die drei Vektoren: $\vec{e}_x,\ \vec{e}_y$ *und* $\vec{e}_z$ benötigt. Diese stellen die drei Achsen des lokalen Koordinatensystems dar. Das Koordinatensystem muss lediglich den zuvor genannten Anforderungen entsprechen, das heißt es gibt mehrere Möglichkeiten drei geeignete Achsen zu finden. Eine mögliche Lösung sieht wie folgt aus:

$$\vec{e}_x = \begin{pmatrix} p2_1 - p1_1 \\ p2_2 - p1_2 \\ p2_3 - p1_3 \end{pmatrix} \quad \vec{e}_y = \begin{pmatrix} 1 \\ -\frac{e_{x_1}}{e_{x_2}} \\ 0 \end{pmatrix} \quad \vec{e}_z = \begin{pmatrix} -\frac{(e_{x_1} * e_{x_3})}{(e_{x_1}^2 * e_{x_2}^2)} \\ \frac{(e_{x_2} * e_{x_3})}{(e_{x_1}^2 * e_{x_2}^2)} \\ 1 \end{pmatrix}$$

Formel 4: Mögliche Lösung der Achsen X,Y,Z des lokalen Koordinatensystems

Die X-Achse ($\vec{e}_x$) kann sehr einfach bestimmt werden, hier muss nur die Anforderung erfüllt sein, dass die X-Achse entlang der *HingeLine* verläuft erfüllt werden. Um $\vec{e}_x$ zu bestimmen wird demzufolge einfach der äußere Punkt der *HingeLine* ($\vec{p1}$) vom inneren Punkt ($\vec{p2}$) abgezogen.

Um $\vec{e}_y$ richtig zu bestimmen muss gelten, dass $\vec{e}_y$ orthogonal zu $\vec{e}_x$ steht. Damit das erfüllt ist muss $< \vec{e}_x,\ \vec{e}_y > = 0$ gelten. Durch auflösen des Skalarproduktes entsteht folgende Gleichung:

$$e_{x_1} * e_{y_1} + e_{x_2} * e_{y_2} + e_{x_3} * e_{y_3} = 0.$$

Formel 5: Y-Achsenberechnung des lokalen Koordinatensystem

Um diese Gleichung zu lösen gibt es zwei Freiheitsgrade, das folgt daraus dass $\vec{e}_y$ orthogonal zu $\vec{e}_x$ liegen muss. Wie sich $\vec{e}_y$ tatsächlich im Raum befindet ist aber

egal. Das heißt um die Gleichung zu lösen können zunächst zwei Variablen frei gewählt werden. Um eine möglichst einfache Lösung zu erhalten macht es Sinn einfache Werte wie 0 und 1 zu wählen. Hier wurde $e_{y_1} = 1$ *und* $e_{y_3} = 0$ gewählt. Durch Einsetzen in die Gleichung und einer kleinen Umformung ergibt sich daraus $e_{y_2} = -\frac{e_{x_1}}{e_{x_2}}$. Somit ist $\vec{e_y}$ ebenfalls eindeutig bestimmt.

Um die Z-Achse, also $\vec{e_z}$ zu berechnen muss gewährleistet sein, dass $\vec{e_z}$ orthogonal zu $\vec{e_x}$ und zu $\vec{e_y}$ ist. Daraus folgt, dass folgendes gelten muss: $< \vec{e_x},\ \vec{e_z} > = 0$ und $< \vec{e_y},\ \vec{e_z} > = 0$. Da es auch hier noch einen Freiheitsgrad gibt kann eine Variable frei gewählt werden, hier $e_{z_3} = 1$. Bringt man dies nun in die Koordinatenform dann folgt: $e_{x_1} * e_{z_1} + e_{x_2} * e_{z_2} + e_{x_3} * e_{z_3} = 0$ und $e_{y_1} * e_{z_1} + e_{y_2} * e_{z_2} + e_{y_3} * e_{z_3} = 0$. Daraus ergbit sich nun ein lineares Gleichungssystem:

$$\left(\begin{array}{ccc|c} e_{x_1} & e_{x_2} & e_{x_3} & 0 \\ 1 & -\frac{e_{x_1}}{e_{x_2}} & 0 & 0 \end{array}\right)$$

Formel 6: LGS zum berechnen der Z-Achse des lokalen Koordinatensystem

Dieses kann nun mit Hilfe des *gaußschen Eliminationsverfahren*[37] gelöst werden. Daraus ergibt sich dann $e_{z_1} = -\frac{(e_{x_1} * e_{x_3})}{(e_{x_1}^2 * e_{x_2}^2)}$ *und* $e_{z_2} = \frac{(e_{x_2} * e_{x_3})}{(e_{x_1}^2 * e_{x_2}^2)}$ *und* $e_{z_3} = 1$.

Jetzt sind alle Achsen des lokalen Koordinatensystems bekannt. Für die Anwendung müssen die Achsen noch normalisiert werden. Für einen normalisierten Vektor muss gelten, dass dessen Betrag = 1 ergibt, also $||v|| = 1$. Eine mögliche Umrechnung eines nicht normalisierten Vektors zu einem normalisierten Vektor sieht wie folgt aus:

[37] [GAU] Das Gaußsche Eliminationsverfahren
[38] [BIEL] Normalisierung von Vektoren

$$\vec{v}_{normalisiert} = \frac{1}{||v||} * \vec{v}$$

Formel 7: Normalisierung eines Vektors[38]

Aus den drei Achsen-Vektoren kann dann folgende Transformationsmatrix erstellt werden:

$$Welt_{trans} = \begin{pmatrix} e_{x_1} & e_{x_2} & e_{x_3} & 0 \\ e_{y_1} & e_{y_2} & e_{y_3} & 0 \\ e_{z_1} & e_{z_2} & e_{z_3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 8: Transformationsmatrix von Weltkoordinaten zum lokalen Koordinatensysten

Da das Koordinatensystem um den *innerHingePoint* aufgebaut wurde, muss zum Schluss die Matrix noch genau um diesen Punkt verschoben werden. Der Ursprung des Koordinatensystems liegt, dann nämlich genau beim Anfang der *HingeLine*. Dazu wird folgende Translationsmatrix, welche eine Verschiebung genau um den *innerHingePoint* berschreibt gebraucht:

$$Translationsmatrix = \begin{pmatrix} 1 & 0 & 0 & -p1_1 \\ 0 & 1 & 0 & -p1_2 \\ 0 & 0 & 1 & -p1_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 9: Translationsmatrix um lokales Koordinatensystem richtig zu positionieren

p1 ist hierbei der *innerHingePoint*. Die Multiplikation der Transformationsmatrix (Formel 8) mit der Translatonsmatrix (Formel 9) ergibt nun die gesuchte Matrix K.

$$K = \begin{pmatrix} e_{x_1} & e_{x_2} & e_{x_3} & 0 \\ e_{y_1} & e_{y_2} & e_{y_3} & 0 \\ e_{z_1} & e_{z_2} & e_{z_3} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -p1_1 \\ 0 & 1 & 0 & -p1_2 \\ 0 & 0 & 1 & -p1_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 10: Berechnung der finalen Transformationsmatrix K

Nun muss noch die inverse Matrix K^{-1} zu K berechnet werden, diese Arbeit wird durch bereits vorhandene OpenCascade Methoden enorm erleichtert.

Quellcode 9: invertieren einer Matrix in OpenCascade

```
1 transformFromLocal = transformToLocal.Inverted();
```

Quellcode 9 zeigt wie mittels openCascade eine Matrix invertiert werden kann. TransformToLocal beinhaltet die Matrix K und durch die Methode *Inverted()* wird die invertierte Matrix zurückgegeben und in transformFromLocal also in K^{-1} gespeichert.

Im Falle von OpenCascade wird die Matrix mittels einer LR-Zerlegung (auch LU Decomposition oder Dreieckszerlegung) invertiert, ein üblicher Algorithmus zum invertieren von Matrizen. Die herkömmliche Matrix wird in zwei Dreiecksmatrizen (obere und untere) zerlegt und somit als Produkt der beiden neuen Matrizen dargestellt.[39]

[39][LRZ] vgl. hierzu LR-Zerlegung TU-Berlin

Per Hand könnte man eine Matrix mittels des Gauss-Jordan-Eliminationsverharen's[40] invertieren.

Damit sind nun K und K^{-1} bekannt, als nächstes werden die Rotationsmatrizen berechnet.

3.5.4 Rotationsmatrizen

Um die Rotation richtig darzustellen werden drei Rotationsmatrizen benötigt $R(\phi),\ R(\theta)$ *und* $R(\alpha)$, eine für jede Achse. Die Rotation um die X-Achse, bzw. um die *HingeLine* ist die Einzige die durch einen in den CPACS Daten angegebenen Wert durchgeführt wird. Die Rotationen um die anderen Beiden Achsen ergeben sich automatisch durch die Verschiebung der *HingeLine* durch den Raum.

Der Winkel α, also die Rotation um die X-Achse ist bekannt, da diese bereits vorgegeben ist, wie genau dies geschieht wird im Kapitel (Visualisierung im TiGLViewer) geklärt. Die Winkel θ und ϕ, für die Rotation um die Y-Achse und die Z-Achse müssen allerdings noch berechnet werden. Die Rotationsparameter θ und ϕ ergeben sich aus folgender Betrachtung: *p1 (innerHingerPoint)* befindet sich im Zentrum des Koordinatensystems und die Position des Punktes *p2 (outerHingerPoint)* gibt daher nach Ausfahren der Klappe die gesuchten Winkel in Kugelkoordinaten wieder. Für die Berechnung wird die Länge r benötigt, diese wird folgendermaßen berechnet $r = \sqrt{x^2 + y^2 + z^2}$. Zudem gilt, $\phi = \arctan(y/x)$ und $\theta = \arccos(z/r)$. Diese Formeln folgen aus den Rechenregeln die beim Rechnen mit Kugelkoordinaten

[40] [GJV] vgl. S.13 Gauss-Jordan-Eiliminationsverfahren

angewandt werden.[41]

Um die Länge r zu bestimmen, müssen also zunächst x,y und z gefunden werden. Dazu muss zunächst der Richtungsvektor der bereits verschobenen Punkte *innerHingePoint* und *outerHingePoint* berechnet werden: $\vec{p_t} = p\vec{2}s - p\vec{1}s$. Anschließend muss zwischen jedem Wert des Vektors $\vec{p_t}$ und den zuvor bestimmten Achsen des lokalen Koordinatensystems das Skalarprodukt gebildet werden. Dadurch wir der Richtungsvektor in das lokale Koordinatensystem verschoben. In folge dessen muss nur noch Länge des neuen Vektors bestimmt werden:

$$\vec{p_k} = \begin{pmatrix} < \vec{p_{t1}}, \vec{e_x} > \\ < \vec{p_{t2}}, \vec{e_y} > \\ < \vec{p_{t3}}, \vec{e_z} > \end{pmatrix} \text{ und } r = ||\vec{p_k}||$$

Formel 11: Berechnung der Länge r in Kugelkoordinaten

Da nun x, y und z bekannt sind ($\vec{p_{k1}}, \vec{p_{k2}}, \vec{p_{k3}}$) sind können mit Hilfe der Länge r die Winkel Phi und Theta bestimmt werden:

$$\phi = atan2(\vec{p_{k2}}, \vec{p_{k1}}) \text{ und } \theta = \arccos(\vec{p_{k3}}/r)$$

Formel 12: Berechnung der Winkel ϕ *und* θ

Zur Berechnung des Winkels ϕ wird eine spezielle Form des Arcustanges verwendet, *atan2*. Dies ist nötig, da je nach Vorzeichen der Werte $\vec{p_{k1}}$ und $\vec{p_{k2}}$ ein andere Teil der Arcustangensfunktion gewählt werden muss. Der atan2 ist folgendermaßen definiert:[42]

[41] [TUFB] Berechnung von r, und den Winkeln Theta und Phi
[42] [STUTT] siehe Definition von arctan

$$atan2(y,x) = \begin{cases} arctan(y/x) & ,wenn\ x > 0 \\ sgn(y) * \pi/2 & ,wenn\ x = 0 \\ arctan(y/x) + \pi & ,wenn\ x < 0\ und\ y >= 0 \\ arctan(y/x) - \pi & ,wenn\ x < 0\ und\ y < 0 \end{cases}$$

Formel 13: Definition atan2

Zum Schluss muss der Winkel θ noch angepasst werden. Das ist nötig da das lokale Koordinatensystem „gedreht“ ist. Man hätte $\vec{e_x}$, $\vec{e_y}$ und $\vec{e_z}$ auch gezielt so bestimmen können, dass diese zusätzliche Drehung nicht notwendig ist. Der Winkel α muss noch in eine radiale Größe umgerechnet werden, der eigentliche Wert von α kann den CPACS Daten entnommen werden.

$$\alpha_{neu} = (\alpha * \pi)/180$$
$$\theta_{neu} = -\pi/2 + \theta$$

Formel 14: Anpassung der Winkel α und θ

Wenn man nun alle berechneten Winkel und Werte in die Standardformen der Rotationsmatrizen einsetzt, dann ergeben sich die einzelnen benötigten Rotationsmatrizen $R(\phi)$, $R(\theta)$ und $R(\alpha)$:

$$R(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_{neu} & -\sin\alpha_{neu} & 0 \\ 0 & \sin\alpha_{neu} & \cos\alpha_{neu} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R(\theta) = \begin{pmatrix} \cos\theta_{neu} & 0 & -\sin\theta_{neu} & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta_{neu} & 0 & \cos\theta_{neu} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 15: Die Rotationsmatrizen $R(\phi)$, $R(\theta)$ und $R(\alpha)$

3.5.5 Translationsmatrix

Bei einem Blick zurück auf die gesamte Transformation $T(\vec{p}_{trans}, \phi, \theta, \alpha) = T(\vec{p}_{trans}) * K^{-1} * R(\phi) * R(\theta) * R(\alpha) * K$ fällt auf, dass jetzt nur noch $T(\vec{p}_{trans})$ bestimmt werden muss, die Translationsmatrix. Zu beachten ist hierbei, dass die Translation erst nach der Rücktransformation in das Weltkoordinatensystem stattfindet.

Eine allgemeine Translationsmatrix genügt der Form:[43]

$$Translationsmatrix = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 16: Form einer Translationsmatrix

[43] [TUM] S. 5 homogene Koordinaten, Translationsmatrix

Die Translationsmatrix lässt sich sehr einfach bestimmen, dafür wird zunächst die Verschiebung des *innerHingePoints* benötigt. Diese berechnet sich durch das Subtrahieren des verschobenen Punktes mit dem ursprünglichen Punkt.

Falls $\vec{p1}$ und $\vec{p2}$ unterschiedlich weit voneinander entfernt sind als $\vec{p1s}$ und $\vec{p2s}$, ist die Positionierung der Klappe bezüglich der *HingeLine* nicht mehr ganz korrekt. Deswegen muss die Klappe zusätzlich noch in Richtung der lokalen X-Achse zentriert werden. Um den Verschiebungsvektor für die Translationsmatrix zu berechnen wurde folgende Gleichung erstellt:

$$\vec{p}_{trans} = \vec{p1s} - \vec{p1} + \frac{1}{||\vec{p2s} - \vec{p1s}||} * (\vec{p2s} - \vec{p1s}) * (-(\ ||\vec{p1} - \vec{p2}|| - ||\vec{p1s} - \vec{p2s}||\ )/2)$$

Formel 17: Translation des innerHingePoints und Zentrierung der Klappe

Der erste Ausdruck $\vec{p1s} - \vec{p1}$ ist für die eigentliche Verschiebung des *innerHingePoints* notwending. Alles darauf folgende bezieht sich auf die Zentrierung der Klappen. Wichtig für die Zentrierung ist der Teil $(\vec{p2s} - \vec{p1s})$, dieser gibt eine Richtung an, in welche die Klappe verschoben werden muss und zwar entlang der verschobenen *HingeLine*. Der Faktor $\frac{1}{||\vec{p2s} - \vec{p1s}||}$ dient dazu den eben genannten Richtungsvektor zu normalisieren, siehe dazu Formel 7. Zuletzt wird der normalisierte Vektor mit $-(\ ||\vec{p1} - \vec{p2}|| - ||\vec{p1s} - \vec{p2s}||\ )/2$ multipliziert, das ist genau die Hälfte der Differenz der alten *HingeLine-Länge* und der neuen *HingeLine-Länge* . Falls die neue *HingeLine* länger sein sollte wird die Klappe dadurch genau in die Mitte der *HingeLine* verschoben.

Wenn nun der Verschiebungsvektor $\vec{p}_{trans}$ für die Werte t_x, t_y und t_z (siehe Formel 16) eingesetzt wird dann entsteht die gesuchte Translationsmatrix $T(\vec{p}_{trans})$:

$$T(\vec{p}_{trans}) = \begin{pmatrix} 1 & 0 & 0 & p_{trans1} \\ 0 & 1 & 0 & p_{trans2} \\ 0 & 0 & 1 & p_{trans3} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Formel 18: Finale Translationsmatrix

3.5.6 Anwendung der gesamten Klappentransformation

Da jetzt alle einzelnen Transformationen bekannt sind, kann durch Multiplikation der einzelnen Transformationen die gesamte Transformationsmatrix $T(\vec{p}_{trans}, \phi, \theta, \alpha, sx)$ durch Einsetzen in Formel 1 ($T(\vec{p}_{trans}, \phi, \theta, \alpha) = T(\vec{p}_{trans}) * K^{-1} * R(\phi) * R(\theta) * R(\alpha) * K$) bestimmt werden.

Da nun die Gesamttransformation $T(\vec{p}_{trans}, \phi, \theta, \alpha)$ bekannt ist, kann diese an die Klappengeometrie angewandt werden. Wie das mittels OpenCascade funktioniert zeigt Quellcode 10.

Quellcode 10: Anwenden von Transformationen in OpenCascade

```
1 BRepBuilderAPI_Transform form(trailingEdgeLoftCut,trailingEdgeDevice.↩
      getTransformation(flapStatus[trailingEdgeDevice.getUID()]));
2 trailingEdgeLoftCut = form.Shape();
```

In Zeile 1 wird die Transformation bereits durchgeführt, *trailingEdgeLoftCut* ist dabei die aktuelle Klappe, also bereits das dreidimensionale Objekt. Mittels *trailin-*

gEdgeDevice.getTransformation(flapStatus[trailingEdgeDevice.getUID()]) wird die eben berechnete, gesamte Transformation zurückgegeben. in der Map *flapStatus* ist für jede Klappe der gewünschte Verfahrungsgrad angeben, also wie weit die Klappe ausgeklappt und rotiert werden soll. Dadurch wird auch indirekt der Winkel α angegeben. In Zeile 2 wird dann nur noch die neue transformierte Landeklappe zugewiesen.

Das Ergebnis der Transformation ist in Abbildung 12 am Flügel erkennbar. Der Flügel besitzt vier *TrailingEdgeDevices*, diese sind am rechten Flügel leicht transformiert zusehen.

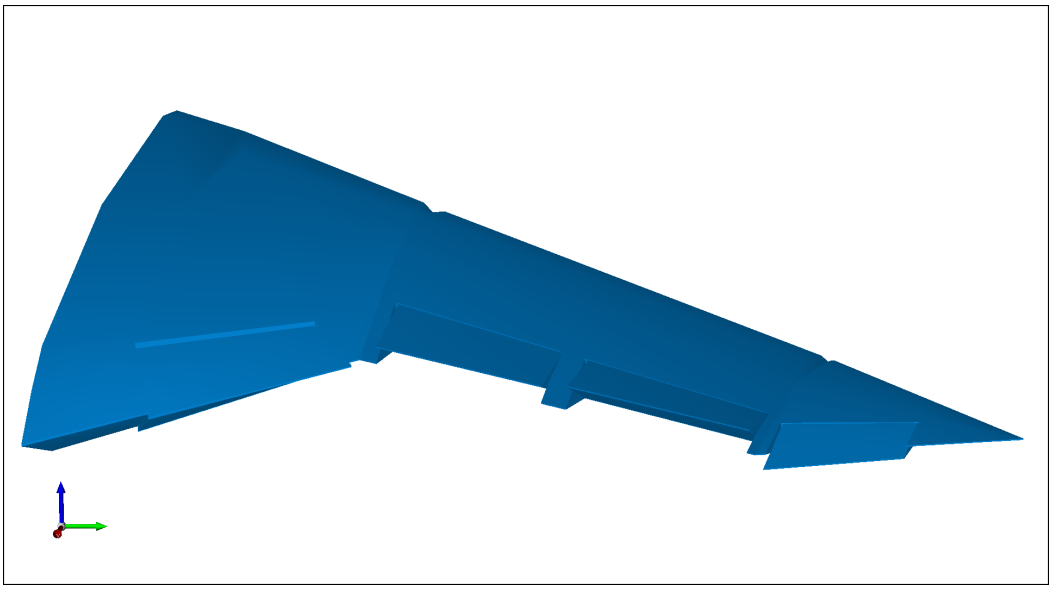


Abbildung 12: Transformierte Klappen am rechten Flügel[44]

[44]Screenshoot des TiGLViewers

3.6 Anbindung an die grafische Oberfläche

Um den Benutzer mit den neu entwickelten Funktionalitäten von TiGL zu interagieren ist eine Schnittstelle zwischen Funktion und Benutzer nötig. In TiGL selbst gibt es die Funktion *ExtendFlap(std::string flapUID, double flapDeflectionPercantage)*. Diese Funktion braucht eine *flapUID*, welche eine Klappe eindeutig identifiziert und eine *flapDeflectionPercentage*, welche angibt, wie weit die Klappe verfahren / rotiert werden soll.

Im TiGLViewer wurde ebenfalls eine Schnittstelle definiert, über den Menüreiter CPACS - Wings lässt sich die Funktion *Show Wing Flaps* ausführen.

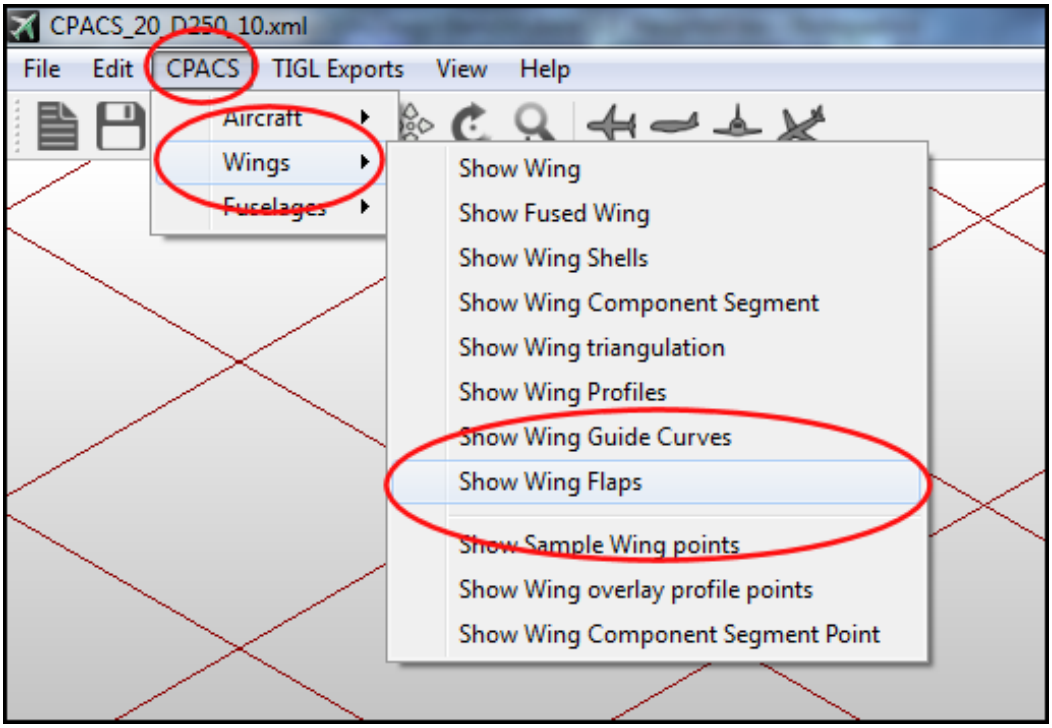


Abbildung 13: Schnittstelle zur Flapberechnung[45]

Der Button *Show Wing Flaps* konnte dank des einfach zu handhabenden grafischen Editors des QT - Creators (einer IDE speziell für die Entwicklung mit QT) einfach per Drag and Drop erstellt werden. Lediglich die Verbindung zu einer im TiGL-Viewer enthaltenen Funktion muss manuell definiert werden. Dies geschieht in der Klasse *TiglViewerWindow* und wurde für den Button *Show Wing Flaps* wie folgt realisiert:

Quellcode 11: die Funktion connect()

```
1 connect(drawWingFlapsAction, SIGNAL(triggered()), cpacsConfiguration, SLOT(↩
      drawWingFlaps()));
```

Dem Button wurde zuvor im grafischen Editor die Aktion, *drawWingFlapsAction* zugewiesen. Der Befehl in Quellcode 11 verknüpft diese Aktion nun mit einer Funktion *drawWingFlaps*. Der Parameter *SIGNAL(triggered())* gibt an, dass die Aktion ausgelöst wird, wenn der Button betätigt wurde. Die Funktion *drawWingFlaps* wird also nun immer dann ausgeführt wenn der Button gedrückt wurde.

Die Funktion *drawWingFlaps* der Klasse *TIGLViewerDocument* bildet also den Einstiegspunkt für die im nächsten Kapitel (Visualisierung im TiGLViewer) beschriebene Visualisierungsfunktion. Um die Eingabe von Parametern für Benutzer zu erleichtern wurde ein Dialog (*TIGLViewerSelectWingAndFlapStatusDialog*) entwickelt, welcher direkt nach einem Klick auf den *Show Wing Flaps* - Button geöffnet wird.

[45] Screenshoot des TiGLViewers

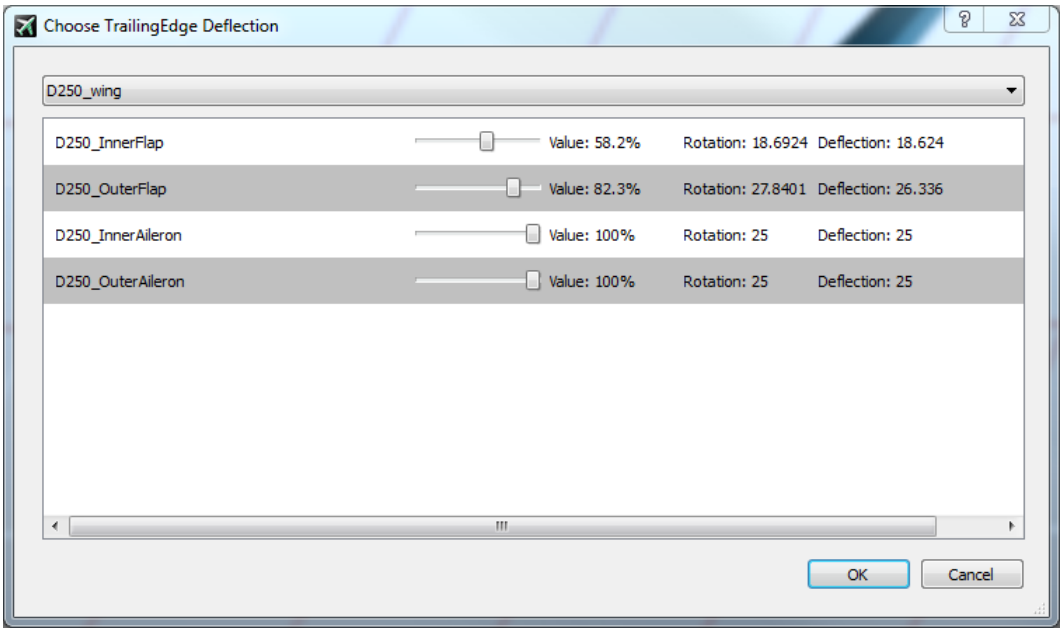


Abbildung 14: TIGLViewerSelectWingAndFlapStatusDialog[46]

In der ComboBox ganz oben kann einer von allen verfügbaren Flügeln ausgewählt werden, dazu werden dann alle in dem Flügel enthaltenen *controlSurfaceDevices* angezeigt. Für jeden *controlSurfaceDevice* kann ein Wert (0 - 100) eingestellt werden, welcher angibt wie weit eine Klappen verfahren werden soll. Die Klappen bewegen sich auf einem in den CPACS-Daten vorgegebenen Pfad.

Wenn der Benutzer im Dialog also alle Deflections definiert hat und auf OK klickt, dann wird der TiGLViewer damit beginnen den angegebenen Flügel mit allen dazugehörigen *controlSufraceDevices* zu visualisieren.

[46]Screenshoot des TiGLViewers

3.7 Visualisierung im TiGLViewer

Man kann zwischen zwei verschiedenen Arten der Visualisierung unterscheiden: die statische und die dynamische Visualisierung. Bei der statischen Visualisierung wird das Flugzeugmodell einmalig gezeichnet. Bei der dynamischen Visualisierung wird das Modell bei jeder Änderung der Eingangswerte teilweise neu gezeichnet. Die statische Visualisierung wird für das endgültige Zeichnen des Flugzeugmodells verwendet. Die dynamische wird für die Anzeige während dem Einstellen der Klappenausschläge benutzt, dadurch bekommt der Benutzer sofortiges Feedback zu den geänderten Werten.

3.7.1 Statische Visualisierung

Die Visualisierung im TiGLViewer findet in der Klasse *TIGLViewerDocument* statt. Einstiegspunkt der Visualisierung ist die Methode *drawWingFlaps()*, diese wird von der GUI ausgeführt. In Quellcode 12 ist die Funktion drawWingFlaps zu sehen. Diese erstellt zunächst eine Liste alle verfügbaren Flügel und öffnet dann den Auswahl-Dialog. Nach Schließen des Dialogs wird die Flugzeuggeometrie gebaut und anschließend angezeigt.

Quellcode 12: Die Methode drawWingFlaps im Überblick

```
1 QStringList wings;
2 tigl::CCPACSConfiguration& config = GetConfiguration();
3 for (int i = 1; i <= config.GetWingCount(); i++)
4 {
5     tigl::CCPACSWing& wing = config.GetWing(i);
6     std::string name = wing.GetUID();
7     if (name == "") {
8         name = "Unknown Wing";
```

```
9       }
10      wings << name.c_str();
11  }
12
13  TIGLViewerSelectWingAndFlapStatusDialog dialog(0,m_cpacsHandle, this);
14  int dialogValue = dialog.exec(wings);
15  if (dialogValue == 0 || dialog.getSelectedWing() == "" ) {
16      myAISContext->EraseAll();
17      return;
18  }
19  // Draw fused Shape
20  myAISContext->EraseAll();
21  tigl::CCPACSWing& wing = GetConfiguration().GetWing( dialog.getSelectedWing() );
22  TopoDS_Shape wingShape = wing.BuildFusedSegmentsWithFlaps(false, dialog.↩
        getControlSurfaceStatus());
23  displayShape(wingShape);
```

In Zeile 1 - 11 wird die besagte Liste aller Flügel erstellt, Flügel die keinen Namen haben, werden mit dem Namen *Unknown Wing* belegt. Danach wird mit der im vorherigen Kapitel beschriebene Dialog erstellt und geöffnet. Der Dialog gibt einen ausgewählten Flügel zurück, falls ein der Aufruf des Dialogs einen ungültigen Wert zurück gibt wird die Aktion abgebrochen. Danach wird in Zeile 22 genau dieser Flügel mit allen Landeklappen und Kontrollflächen gebaut und anschließend in Zeile 23 durch *displayShape()* angezeigt.

3.7.2 Dynamische Visualisierung

Mit einer Dynamischen Visualisierung ist gemeint, dass das Flugzeugmodell bereits gezeichnet wird wenn sich der Dialog öffnent und sich danach sofort, je nach Änderung bei den Einstellungen des Dialogs, an die neuen Werte anpasst. D.h. beim Verschieben eines Sliders klappt sich die dazugehörige Landeklappe in einer flüssigen Animation aus oder ein.

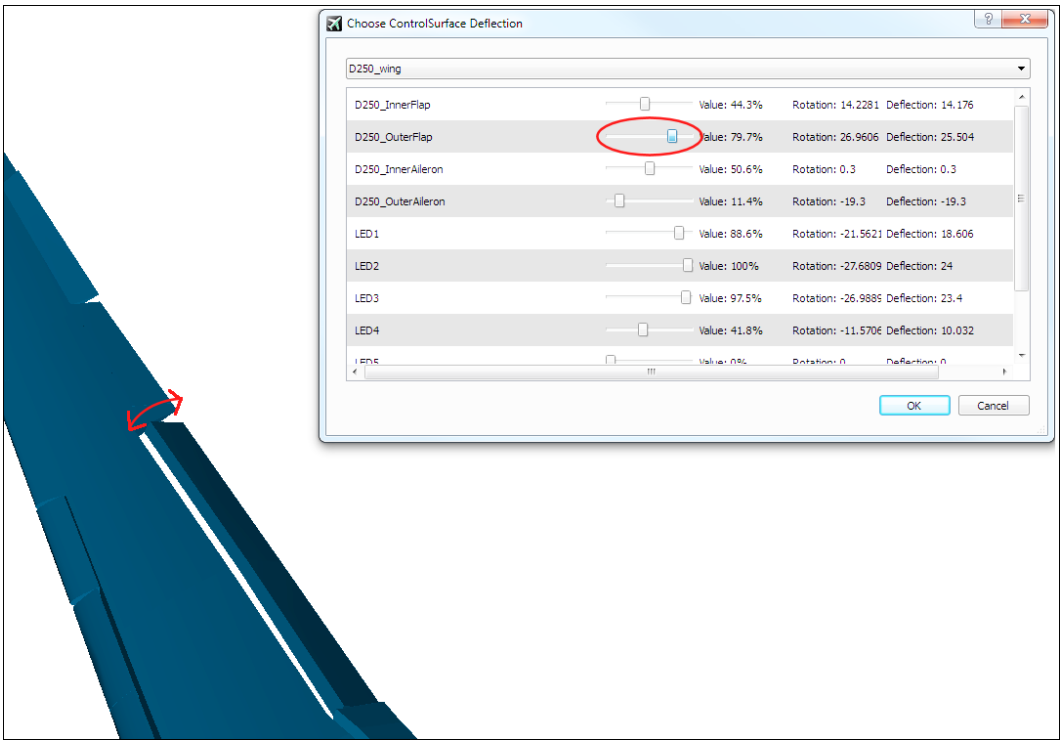


Abbildung 15: Live Aktualisierung von Klappengemetrien[47]

Zeile 13 des Quellcodes 12 zeigt, dass dem Dialog die aktuelle this-Referenz des TIGLViewerDocument's übergeben wird. Der Dialog braucht diese um im TIGLViewerDocument Funktionen zu triggern, welche nach einer Änderung der Slider im Dialog das Flugzeugmodell neu zeichnen.

Es gibt zwei Funktionen im TIGLViewerDocument welche vom Dialog getriggert werden, das ist zum einen die Funktion

(1) *drawWingFlapsForInteractiveUse(std::string selectedWing)*

und zum anderen die Funktion

[47]Screenshot des TiGLViewers

(2) *updateControlSurfacesInteractiveObjects(std::string selectedWing, std::map<std::string,double>flapStatus, std::string controlUID)*

Die erste Funktion benötigt als Parameter nur eine ID des Flügels, welcher visualisiert werden soll und wird immer dann aufgerufen, wenn der Benutzer im Dialog einen neuen Flügel auswählt. Quellcode 13 zeigt wie die Funktion *drawWingFlapsForInteractiveUse* aufgebaut ist.

Quellcode 13: Ausschnitt: Draw-Wing-Flaps for interactive use

```
1  TopoDS_Shape wingWithoutFlaps = wing.GetWingWithoutFlaps();
2  displayShape(wingWithoutFlaps);
3
4  for ( int i = 1; i <= wing.GetComponentSegmentCount(); i++ ) {
5
6      tigl::CCPACSWingComponentSegment &componentSegment = (tigl::↩
           CCPACSWingComponentSegment&) wing.GetComponentSegment(i);
7      tigl::CCPACSControlSurfaceDevices* controlSurfaceDevices = componentSegment.↩
           getControlSurfaces().getControlSurfaceDevices();
8
9      for ( int j = 1; j <= controlSurfaceDevices->getControlSurfaceDeviceCount();↩
            j++ ) {
10         tigl::CCPACSControlSurfaceDevice &controlSurfaceDevice = ↩
               controlSurfaceDevices->getControlSurfaceDeviceByID(j);
11
12         flapsForInteractiveUse[controlSurfaceDevice.getUID()] = displayShape(↩
               controlSurfaceDevice.GetLoft());
13         std::map<std::string,double> flapStatus;
14         updateControlSurfacesInteractiveObjects(selectedWing,flapStatus,↩
               controlSurfaceDevice.getUID());
15     }
16 }
```

In den den ersten zwei Zeilen wird eine Flügelgeometrie erstellt und visualisiert die keine Kontrollflächen und Landeklappen enthält. Diese wurden sozusagen aus der

Flügelgeometrie ausgeschnitten. Die Zeilen 10 - 14 werden für jeden *controlSurfaceDevice* ausgeführt, das heißt für alle Klappen unabhängig vom Klappentyp. Jede Klappe wird anschließend in Zeile 12 visualisiert. Das grafische Objekt wird dabei in einer Map (*flapsForInteractiveUse*) gespeichert. Danach wird in den Zeilen 13 und 14 jede Klappe noch zusätzlich durch die Funktion *updateControlSurfacesInteractiveObjects* zu den default Werten verfahren.

Die zweite Funktion *updateControlSurfacesInteractiveObjects* arbeitet jetzt nur noch mit einer Klappe und verfährt diese mittels den übergebenen Parametern. Diese Funktion wird immer dann ausgeführt, wenn im Dialog irgendeine Einstellung an den Slidern geändert wurde.

Quellcode 14: Dynamische Transformation von Klappen

```
1 gp_Trsf trsf = controlSurfaceDevice.getTransformation(flapStatus[↩
      controlSurfaceDevice.getUID()]);
2 myAISContext->SetLocation((Handle_AIS_InteractiveObject) flapsForInteractiveUse[↩
      controlSurfaceDevice.getUID()],trsf);
```

In der ersten Zeile des Quellcodes 14 wird die Gesamttransformation (siehe Kapitel 3.5) der jeweiligen Klappen in *trsf* zwischengespeichert. Danach wird in der zweiten Zeile die Klappe, welche aktualisiert werden muss, mit der Gesamttransformation neu transformiert und neu gezeichnet.

3.8 Der Wing-Status-Dialog

Im letzten Abschnitt der Problemlösung wird noch einmal der *TIGLViewerSelectWingAndFlapStatusDialog* näher betrachtet. Als Schnittstelle zwischen Benutzer

und den neu erstellten TiGL-Funktionen muss dieser möglichst fehlerfrei, übersichtlich und intuitiv sein.

Das Design ist schlicht gehalten, um eine gute Übersichtlichkeit zu gewähren. In der Abbildung 14 ist die GUI des Dialogs zu sehen. Das Design wurde so gestaltet, damit der Benutzer auf Anhieb erkennen kann, mit welchem Elementen er interagieren kann.

Der Benutzer kann den zur Visualisierung gewünschten Flügel auswählen und den Status jeder Landeklappe bestimmen. Falls ein Flügel ausgewählt wird, der keine Kontrollflächen hat so wird kein *ControlSurfaceDevice* angezeigt, sondern ein Text wiedergegeben der den Benutzer darauf hinweißt, dass der ausgewählte Flügel keine Kontrollflächen besitzt (siehe Abbildung 16). Zudem wird der OK-Button des Dialogs deaktiviert, so können keine ungültige Rückggabewerte vom Dialog erzeugt werden.

Die GUI wurde größtenteils im graphischen Editor des QT-Creators entworfen. Lediglich die Einträge für die einzelnen *ControlSurfaceDevices* müssen mittels Code hinzugefügt werden. Das ist nötig, da die Anzahl der Devices je nach ausgewähltem Flügel variiert.

Selbst nur ein Flügel eines umfangreiches Flugzeugmodell kann sehr viele Kontrollflächen Landeklappen besitzen. Daher gibt es, falls ein Datensatz mehrere verschiedene Klappentypen beinhalten, die Möglichkeit Filter zu definieren. Dann

[49]Screenshot des TiGLViewers

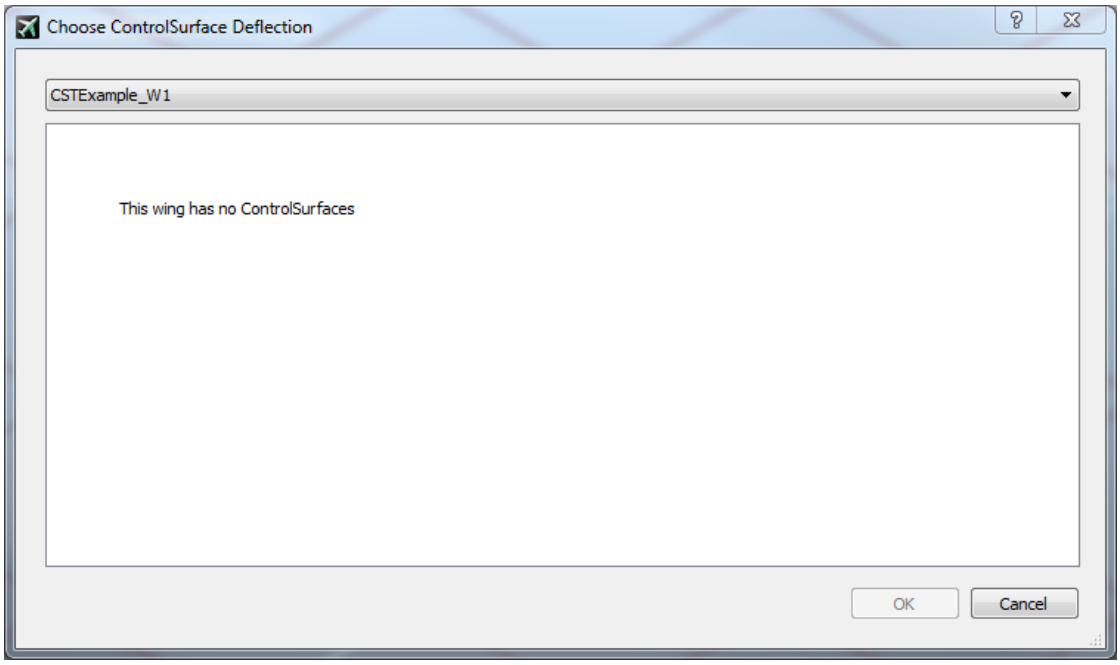


Abbildung 16: WingFlapStatusDialog mit ausgewähltem Flügel ohne Landeklappen[49]

werden nur Klappen der gewünschten Klappentypen angezeigt. In Abbildung 17 sind die Filter Checkboxen für *TrailingEdgeDevices*, *LeadingEdgeDevices* und *Spoiler* zu sehen.

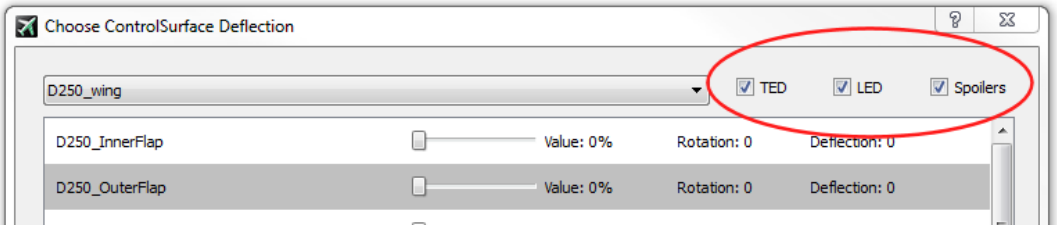


Abbildung 17: WingFlapStatusDialog Filter Optionen[50]

[50]Screenshot des TiGLViewers

4 Qualitätssicherung

In großen Softwareprojekten ist es besonders wichtig eine hohe Qualität sowohl für die eigentliche Software, als auch den Quellcode und die Dokumentation zu sicher. Durch eine hohe Qualität werden Wartbarkeit und Entwicklungsgeschwindigkeit nachhaltig verbessert und auf Dauer sichergestellt. Zudem wird durch eine umfangreiche Abdeckung mit Tests die Korrektheit der Software regelmäßig überprüft. In TiGL wurden mehrere Maßnahmen getroffen um eine hohe Qualität des Projektes sicherzustellen.

4.1 Coding Guidelines

Um den Code in TiGL einheitlich und leserlich zu gestalten wurden TiGL-weit *coding guidlines* eingeführt. Dadurch wird sichergestellt, dass der von verschiedenen Entwicklern erstellte Code den gleichen Stilregeln genügt. Dadurch wird durch die verschiedenen Bereiche von TiGL die Lesbarkeit und somit auch die Wartbarkeit optimiert. Coding Guidlines sind ein Satz von Regeln, die den Programmierstil und die gewünschte formale Struktur des Quellcodes beschreiben. In TiGL müssen die Regeln ständig eingehalten werden.

4.2 Implementierung von Testfällen

Im Rahmen dieser Arbeit wurden mehrere Tests entwickelt, welche die Funktionalität und Richtigkeit von bestimmten Methoden zur Erstellung und Transformation von Klappen testen. Dadurch wird auch bei zukünftigen Änderungen sichergestellt,

dass Fehler und Bugs schnell entdeckt und auch behoben werden können.

Um Software vernünftig testen zu können muss der Quellcode auch dafür geeignet sein, dass heißt er muss möglichst modular geschrieben sein. Funktionen sollten daher möglichst nur eine spezielle Aufgabe erfüllen, welche dann getestet werden kann.

Bei den neu erstellten Funktionen für die Klappen wird hauptsächlich die Mathematik für die Transformation und das Erstellen der Geometrien getestet.

4.2.1 Überprüfung der Transformationsmatrix zum lokalen Koordinatensystem

Die Prüfung der Transformationsmatrix zum lokalen Koordinatensystem kann in drei Bereiche unterteilt werden. Als erstes wird geprüft ob der Punkt p1, also der innerHingePoint vor der Verschiebung tatsächlich wie gewollt den Ursprung des Koordinatensystems bildet.

Danach wird überprüft, ob der Punkt p2 (der *outerHingePoint*) transformiert in seiner Y und Z Komponente gleich Null ist und dass seine X Komponente genau der Entfernung der Punkte p1 und p2 entspricht.

Der dritte Test prüft, ob ein Punkt der zum lokalen Koordinatensystem und wieder zurück transformiert wurde, auch tatsächlich wieder den gleichen Vektor ergibt.

Quellcode 15: Lokale Transformationsmatrix: Ursprungstest

```
1  tigl::CTiglControlSurfaceTransformation transformation(p1,p2,p1s,p2s,90);
2  gp_Vec v1(p1.XYZ());
3  gp_Vec v2(p2.XYZ());
4
5  gp_Trsf t = transformation.getToLocalTransformation();
6  gp_Pnt p1Trans = p1.Transformed(t);
7  /* p1Trans has be be zero, because p1 is the LocalCoord-´System origin */
8  ASSERT_EQ(p1Trans.X(),0.0);
9  ASSERT_EQ(p1Trans.Y(),0.0);
10 ASSERT_EQ(p1Trans.Z(),0.0);
```

Der Quellcode 15 zeigt wie der erste Test prüft, ob der Punkt p1 wirklich den Ursprung des lokalen Koordinatensystems bildet. Dazu wird zunächst die Gesamttransformation erstellt, danach wird in den Zeilen 5 und 6 nur die Transformation zum lokalen Koordinatensystem auf den Punkt p1 angewandt. In den folgenden Zeilen wird dann überprüft, ob der transformierte Punkt p1Trans null in jeder Komponente ist.

Quellcode 16: Lokale Transformationsmatrix: Test der X-Achse

```
1  gp_Pnt p2Trans = p2.Transformed(t);
2  ASSERT_NEAR(p2Trans.Y(),0.0,1e-7);
3  ASSERT_NEAR(p2Trans.Z(),0.0,1e-7);
4  /* only the x value of p2Trans can varry */
5  ASSERT_NEAR(p2Trans.X(), (v1-v2).Magnitude(), 1e-7 );
```

In Quellcode 16 wird geprüft, dass der Punkt p2 wenn er transformiert ist, sich nur auf der X-Achse befinden kann. Dazu wird zunächst der Punkt p2 transformiert und anschließend wird geprüft, ob die Y und Z Komponente des Vektors 0 ist. Anschließend kann noch die genaue Position auf der X-Achse überprüft werden. Der X-Wert muss genau der Entfernung der Punkte p1 und p2 entsprechen.

Quellcode 17: Lokale Transformationsmatrix: Gleichheitstest $K * K^{-1} = E$

```
1 gp_Vec v1SameCheck = v1.Transformed(t);
2 v1SameCheck.Transform(transformation.getFromLocalTransformation());
3 gp_Vec zeroVec = v1 - v1SameCheck;
4 /* v1 transformed and reTransformed should still be the same vector */
5 ASSERT_NEAR(zeroVec.X(),0,1e-7);
6 ASSERT_NEAR(zeroVec.Y(),0,1e-7);
7 ASSERT_NEAR(zeroVec.Z(),0,1e-7);
```

Der Quellcode 17 zeigt die Implementierung des dritten Tests. Dazu wird ein Punkt in Zeile 1 zunächst ins lokale Koordinatensystem transformiert und in Zeile 2 wieder durch die Rücktransformation zurück transformiert. Anschließend wird überprüft ob der hin- und rücktransformierte Vektor dem ursprünglichen Vektor gleicht.

4.2.2 Testen der Rotation

Der Quelltext 18 zeigt einen einfachen Test der die Richtigkeit der Rotation von Klappen um die Z-Achse prüft.

Quellcode 18: Testfunktion: Rotation bei Verschiebung der HingeLine

```
1  TEST(TiglControlSurfaceTransformation, rotPhiTransformation)
2  {
3      gp_Pnt p1(1,1,0);
4      gp_Pnt p2(1,2,0);
5      gp_Pnt p1s(p1.XYZ());
6      gp_Pnt p2s(2,2,0);
7      tigl::CTiglControlSurfaceTransformation transformation(p1,p2,p1s,p2s,90);
8
9      gp_Trsf t = transformation.getRotPhiTransformation();
10     gp_Quaternion quad = t.GetRotation();
11     ASSERT_NEAR(quad.GetRotationAngle(),M_PI/4,1e-7);
12 }
```

In den Zeilen 3 - 6 werden die vier HingePoints erstellt, welche die HingeLine vor und nach der Verschiebung definieren. In Zeile 7 wird dann aus den vier Punkten die nötige Gesamttransformation erstellt. Danach wird die Transformationsmatrix für die Rotation um die Z-Achse mittels *getRotPhiTransformation* zwischengespeichert. in Zeile 11 wird der eigentliche Test durchgeführt, es wird überprüft ob die berechnete Rotation um die Z-Achse gleich $\frac{\pi}{4}$ ist. Falls dies nicht der Fall sein sollte, wird der Test fehlschlagen.

In Abbildung 18 ist zusehen, warum der Winkel genau $\frac{\pi}{4}$ sein muss. Zur einfacheren Veranschaulichung wurde eine zweidimensionale Darstellung gewählt.

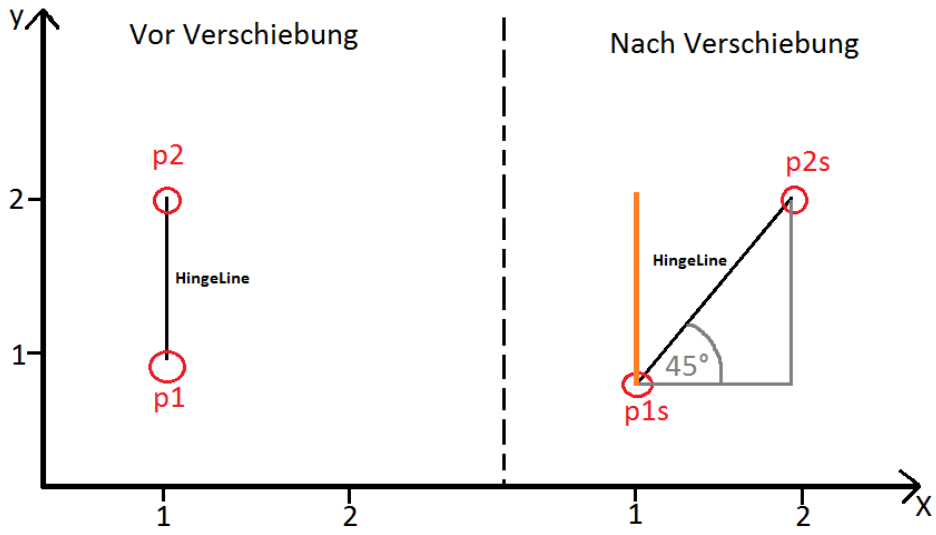


Abbildung 18: Veranschaulichung der Berechnung des Winkels ϕ[51]

[51] Ausschnitt aus dem CPACS Datensatz: CPACS_21_D150.xml

Da das lokale Koordinatensystem um den Punkt p1 aufgebaut wird, wird die reine Translation zunächst nur durch die Verschiebung des Punktes p1 bestimmt. Daraus würde dann die in orange eingezeichnete HingeLine entstehen. Um die richtige HingeLine zu erhalten, muss nun noch eine Rotation um die Z-Achse (die Z-Achse verläuft Senkrecht durch den Punkt p1) von 45° erfolgen. Der Test prüft ob die Funktion zum Bestimmen der Rotationsmatrix um die Z-Achse in genau diesem Testszenario das richtige Ergebnis von 45° ($\frac{\pi}{4}$) liefert. Der selbe Test kann für verschiedene Werte ausgeführt werden, um so andere Szenarien zu testen. Für die Berechnung der Rotationsmatritzen um die X- und Y- Achse gibt es ähnliche Tests.

4.2.3 Testen der Translation

Die Translationsmatrix lässt sich sehr einfach auf ihre Richtigkeit prüfen. Hier muss nur bedacht werden, wie die Translation definiert ist und zwar um die Verschiebung des innerHingePoints.

Quellcode 19: Testfunktion: Translation bei Verschiebung der HingeLine

```
1  float transX = 1, transY = 0, transZ = 0;
2  gp_Pnt p1(1,1,1);
3  gp_Pnt p2(1,2,1);
4  gp_Pnt p1s(1+transX,1+transY,1+transZ);
5  gp_Pnt p2s(p2.XYZ());
6  tigl::CTiglControlSurfaceTransformation transformation(p1,p2,p1s,p2s,90);
7
8  gp_Trsf t = transformation.getTranslationTransformation();
9  gp_XYZ translationPart  = t.TranslationPart();
10
```

```
11 ASSERT_NEAR(translationPart.X(),transX,1e-6);
12 ASSERT_NEAR(translationPart.Y(),transY,1e-6);
13 ASSERT_NEAR(translationPart.Z(),transZ,1e-6);
```

In der ersten Zeile wird die gewünschte zu testende Translation angegeben, hier könnte jeder beliebige Wert stehen. Durch die angegebenen Translationswerte wird aus dem Punkt p1 der Punkt p1s erzeugt. Danach wird wie gehabt die Gesamttransformation erzeugt und dann die Translationsmatrix zwischengespeichert. In den Zeilen 11 - 13 werden dann die eigentlichen Überprüfungen durchgeführt, es wird geprüft ob die in der Translationsmatrix enthaltenen Translationswerte den initial angegebenen Werten entsprechen. Dadurch wird sichergestellt, dass die erstellte Translationsmatrix keine Fehler enthält.

4.2.4 Testen der Geometrieerstellung

Um die Richtigkeit der erstellten Geometrien zu testen, wurden zwei Testfunktionen implementiert. Eine überprüft die Ausrichtung der Klappen, indem sie den Normalenvektor der Oberfläche mit dem durchschnittlichen Normalenvektor auf der Oberseite des Flügels vergleicht.

Die zweite Testfunktion prüft, dass die projizierten Punkte zum Erstellen der Grundseite der Ausschneide Box (siehe Abbildung 8) auch wirklich in einer Ebene liegen.

Testen der Ausrichtung

Um die Ausrichtung zu testen wird für jede Kontrollfläche, die tatsächliche Ausrichtung mit der zuvor gewünschten Ausrichtung, auf Gleichheit überprüft. Dies geschieht indem der Normalenvektor der Klappe mit dem im Vorfeld berechneten gewünschten Normalenvektor zur Klappenerstellung verglichen wird. Um den Normalenvektor der Klappe zu erhalten wird in Zeile 1 des Quellcodes 20 eine Funktion *getNormalOfControlSurfaceDevice()* benutzt, der Normalenvektor der Klappe bestimmt deren gewünschte Ausrichtung.

Quellcode 20: Test zur Überprüfung der Ausrichtung von Klappen

```
1  gp_Vec normalCSD = controlSurface.getNormalOfControlSurfaceDevice();
2
3  TopoDS_Face aCurrentFace = face;
4  Standard_Real umin, umax, vmin, vmax;
5  BRepTools::UVBounds(aCurrentFace,umin, umax, vmin, vmax);
6  Handle(Geom_Surface) aSurface = BRep_Tool::Surface(aCurrentFace);
7  GeomLProp_SLProps props(aSurface, umin, vmin, 1, 0.01);
8  gp_Vec normalWCS = gp_Vec(props.Normal().XYZ());
9
10 ASSERT_NEAR(std::fabs(normalCSD.X()),std::fabs(normalWCS.X()),1e-4);
11 ASSERT_NEAR(std::fabs(normalCSD.Y()),std::fabs(normalWCS.Y()),1e-4);
12 ASSERT_NEAR(std::fabs(normalCSD.Z()),std::fabs(normalWCS.Z()),1e-4);
```

In den Zeile 3-8 wird der Normalenvektor der Grundfläche des *cut-out-shapes* berechnet, um diesen zu erhalten sind einige OpenCASCADE Funktionen nötig die hier zwecks Überschaubarkeit nicht genauer erläutert werden. In den folgenden Zeilen werden die zwei Vektoren miteinander verglichen. Der Test prüft ob die tatsächliche Ausrichtung der erzeugten Klappe auch der gewünschten Ausrichtung entspricht.

Testen der Projektion

Das Testen der Projektion soll sicherstellen, dass die projizierten Punkte in einer Ebene liegen. Das ist eine nötige Voraussetzung, damit später verwendete Funktionen nicht fehlschlagen. Der Quellcode 21 zeigt einen Auschnitt der Testfunktion.

Quellcode 21: Überprüfung der Projektion

```
1 // define point1 - point4 and pp1 - pp4 here.
2 controlSurface.getProjectedPoints(point1,point2,point3,point4,pp1,pp2,pp3,pp4);
3
4 gp_Pnt sv = gp_Pnt(pp1.XYZ());
5 gp_Vec normal = (pp2 - pp1)^(pp3 - pp1);
6 gp_Pln plane(sv,normal);
7 ASSERT_EQ(plane.Contains(gp_Pnt(pp4.XYZ()),1e-5),Standard_Boolean(true));
```

In der zweiten Zeile wird die Funktion *getProjectedPoints([...])* benutzt, diese soll hier auf ihre Richtigkeit überprüft werden. Als Parameter nimmt diese Funktion die vier Punkte (point1 - point4), welche die Klappenkontur definieren und gibt die auf eine Ebene projizierten Punkte in den Platzhaltern pp1 - pp4 zurück.

Danach wird aus drei der vier Punkte eine Ebene erzeugt und anschließend geprüft ob der vierte Punkt auch innerhalb dieser Ebene liegt.

4.3 Dokumentation

Um nachfolgenden Entwicklern die Arbeit an den neu entstandenen Teilbereichen in TiGL zu erleichtern, wurde mit dieser Arbeit eine umfangreiche Dokumentation über die durchgeführten Änderungen und neuen Algorithmen angefertigt.

Eine gute Dokumentation der eingeführten Neuerungen ist wichtig, um nachhaltig das Verständnis über die Änderungen und eine gute Wartbarkeit des Codes zu gewährleisten.

Für TiGL wird zudem ein Wiki gepflegt. Neben der Dokumentation wurden auch hier spezielle Erkenntnisse festgehalten.

5 Zusammenfassung

Zu Beginn des letzten Kapitels werden, die im laufe der Arbeit erlangten Ergebnisse, gesammelt und bewertet. In Folge dessen wird ein kleiner Ausblick in die Zukunft dargestellt. Und zuletzt wird noch ein Fazit über die gesamte Arbeit gezogen.

5.1 Ergebnisse

Rückblickend auf die in Kapitel 3.1.1 (Anforderungsdefinition: Ermittlung / Analyse) ermittelten Aufgaben, wird deutlich dass alle Anforderungen bis auf die optionale Anforderung der Exportfunktion implementiert wurden.:

- Erstellen einer passenden Datenstruktur
- Einlesen der CPACS-Daten
- Erzeugen der Klappengeometrien
- Implementierung von Verfahrwegen
- Visualisierung im TiGLViewer
- Optional: Export

Es wurde eine zur Problemlösung passende Datenstruktur erstellt. Die in den CPACS-Daten enthaltene Informationen können ausgelesen und zur Erstellung der Klappengeometrie genutzt werden. Die in den CPACS-Daten beschriebenen Verfahrwege können an die Klappengeometrie angewandt und zeitgleich im TiGLViewer visualisiert werden.

5.2 Ausblick in die Zukunft

In Zukunft könnten für die Klappen noch weitere Detailgrade implementiert werden. In CPACS gibt es die Möglichkeit Klappen auf verschiedene Arten zu definieren. Momentan werden nur zwei dieser verschiedenen Möglichkeiten in TiGL genutzt. Desweiteren könnte die Exportfunktionalität von TiGL dahingehend erweitert werden, dass auch spezielle Exportfunktionen für die Klappengeometrien bereitgestellt werden.

5.3 Fazit

Aufgabe war es den Detailgrad der Erzeugung von Flügelgeometrien in TiGL zu erweitern. Durch die im Rahmen dieser Arbeit implementierten Features ist es nun möglich, anstatt den früher nur sehr simplen Flügelgeometrien, jetzt auch Flügel mit den verschiedensten Klappentypen (Landeklappen, Trimmklappen, Spoilern, ...) zu visualisieren. Dadurch wurde TiGL um eine wichtig Funktionalität erweitert. Alle im Vorfeld beschriebenen muss Ziele wurden im Rahmen dieser Arbeit bearbeitet und implementiert. Die optionale Exportfunktion wurde nicht implementiert.

Abschließend lässt sich sagen, dass die in dieser Arbeit erzielten Ergebnisse, einen positiven Mehrwert für TiGL darstellen. TiGL wurde um eine weitere umfangreiche Funktion erweitert und dies ohne vorhandene Funktionalitäten einzuschränken oder negativ zu beeinflussen.

Literaturverzeichnis

Referenzierte Quellen

[DLR] *Ofizielle Website des Deutschen Zentrums für Luft- und Raumfahrt e.V.* http://www.dlr.de/ Abruf: 05. Februar 2014.

[BMBF] *Bundesministerium für Bildung und Forschung* http://www.bmbf.de/de/1659.php Abruf: 12. März 2014.

[SC] *Webpräsenz der Einrichtung Simulations- und Softwaretechnik des DLR* http://www.dlr.de/sc/ Abruf: 7. Februar 2014.

[TIGL] *Projektseite TiGL* http://software.dlr.de/p/tigl/home/ Abruf: 6. März 2014.

[CASCADE] *Webpräsenz von OpenCASCADE Technology* http://www.opencascade.org/about/profile/history/ Abruf: 11. Februar 2014.

[TIVA] *Carsten M. Liersch, Martin Hepperle:* CEAS Aeronautical Journal December 2011, Volume 2, Issue 1-4, pp 57-68 A distributed toolbox for multidisciplinary preliminary aircraft design

[CASCU] *Webpräsenz von OpenCASCADE: List of Customers* http://www.opencascade.com/customers/ Abruf: 14. Februar 2014.

[CPP] *Levitt, S.P. :* AFRICON, 2004. 7th AFRICON Conference in Africa Volume: 2 Digital Object Identifier: 10.1109/AFRICON.2004.1406879 Publication Year: 2004 , Page(s): 1197 - 1202 Vol.2 IEEE Conference Publications.

[CMAKE] *Hoffman, B. ; Cole, D. ; Vines, J. :* DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009 Digital Object Identifier: 10.1109/HPCMP-UGC.2009.62 Publication Year: 2009 , Page(s): 378 - 382 IEEE Conference Publications.

[CPPH] *Website zu C++, The C++ Resources Network* http://www.cplusplus.com/info/history/ Abruf: 24. März 2014.

[CPER] *Robert Hundt: Loop Recognition in C++/Java/Go/Scala* https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf Abruf 20. Februar 2014.

[CPACS] *Projekteintrag zu CPACS auf Software.dlr.de* http://software.dlr.de/p/cpacs/home/ Abruf: 10. März 2014.

[TIXI] *Öffentliches Wiki von Tixi auf google code* http://code.google.com/p/tixi/ Abruf: 16. Februar 2014.

[CDOC] *CPACS Dokumentation* http://code.google.com/p/cpacs/downloads/list Abruf: 25. März 2014.

[QTWT] *Qt Framework A cross-platform widget toolkit* https://noppa.aalto.fi/noppa/kurssi/t-

111.5350/viikkoharjoitukset/T-111_5350_qt.pdf Abruf: 03. Juli 2014.

[QTIU] *Qt in Use* http://qt.digia.com/Qt-in-Use/ Abruf: 03. Juli 2014.

[QTAF] *A Brief Introduction to the Qt Application Framework* http://www.slideshare.net/zblair/brief-introtoqt Abruf: 03. Juli 2014.

[GACO] *Überblick über das Qt-Framework* http://openbook.galileocomputing.de/python/python_kapitel_24_005.htm Abruf: 03. Juli 2014.

[QTB] *Rischpater, R. and Zucker, D.* Beginning Nokia Apps Development: Doing More with Qt (2010). Apress, [online] pp.87-137. Available at: http://dx.doi.org/10.1007/978-1-4302-3178-3_5.

[QTN] *Future of Qt brighter after Digia buys licensing biz from Nokia* http://arstechnica.com/information-technology/2011/03/future-of-qt-brighter-after-digia-buys-licensing-biz-from-nokia/ Abruf: 05. Juli 2014.

[QTA] *About Qt* http://qt.digia.com/about-us/ Abruf: 18. August 2014.

[IBM] *A quick introduction to the Google C++ Testing Framework* http://www.ibm.com/developerworks/aix/library/au-googletestingframework.html Abruf: 01. Juli 2014.

[GOO] *Getting started with Google C++ Testing Framework* https://code.google.com/p/googletest/wiki/Primer Abruf: 01. Juli 2014.

[MOFO] *GTest* https://developer.mozilla.org/en-US/docs/GTest Abruf: 01. Juli 2014.

[GAU] *Wirtschafts Universität Wien: Das Gaußsche Eliminationsverfahren* http://statistik.wu-wien.ac.at/ leydold/MOK/HTML/node9.html Abruf: 24. Juni 2014.

[WVSG] *Werner-von-Siemens-Gymnasium: Homogene Koordinaten* http://www.wvs.be.schule.de/faecher/informatik/material/grafik/homogene_koordinaten.html Abruf: 09. Juli 2014.

[BIEL] *Universität Bielefeld: Lineare Algebra Grundlagen* http://www.homes.uni-bielefeld.de/afeldmann3/mathematik/linearealgebra/buch/lineare_algebra_grundlagennode112.html Abruf: 08. Juli 2014.

[LRZ] *TU Berlin: LR-Zerlegung Numerische Mathematik* http://www3.math.tu-berlin.de/Vorlesungen/WS12/NumMath1/uebung/LR-Zerlegung.pdf 10. Juli 2014.

[GJV] *Universität Trier: Kapitel 2: Matrizen* http://www.math.uni-trier.de/ schulz/prosem0708/Erschens.pdf Abruf: 08. Juli 2014.

[OLDE] *Uni Oldenburg: 3D-Transformationen* http://olli.informatik.uni-oldenburg.de/Grafiti3/grafiti/flow7/page7.html Abruf: 24. Juni 2014.

[JAUN] *Jacobs-University: Rotation und Euler Winkel* http://plum.eecs.jacobs-

university.de/download/diplom/node26.html Abruf: 25. Juni 2014.

[TUFB] *TU-Freiberg: Kugelkoordinaten Drehimpuls- und Laplace-Operator* http://tu-freiberg.de/fakult2/thph/lehre/qm-i/l-imp.pdf Abruf: 25. Juni 2014.

[STUTT] *Universität Stuttgart: Kugelkoordinaten* http://mo.mathematik.uni-stuttgart.de/inhalt/aussage/aussage441/ Abruf: 09. Juli 2014.

[TUM] *TU München: Computer Aided Medical Procedures* http://campar.in.tum.de/twiki/pub/Chair/TeachingWs08CPP/-3DTransformationenViewingLight.pdf Abruf: 26. Juni 2014.

Sonstige Quellen

[1] *Blanchette, J.; Summerfield, M.:* C++ GUI Programmierung mit Qt 4: Die offizielle Einführung. Addison Wesley Verlag, 2008 (Programmers's Choiche). - ISBN 978382732791

[2] *Timo Tischler* Bericht zum Modul Praxis III: Plattformunabhängige Visualisierung von Flugzeuggeometrien. März 2013.

Anhang

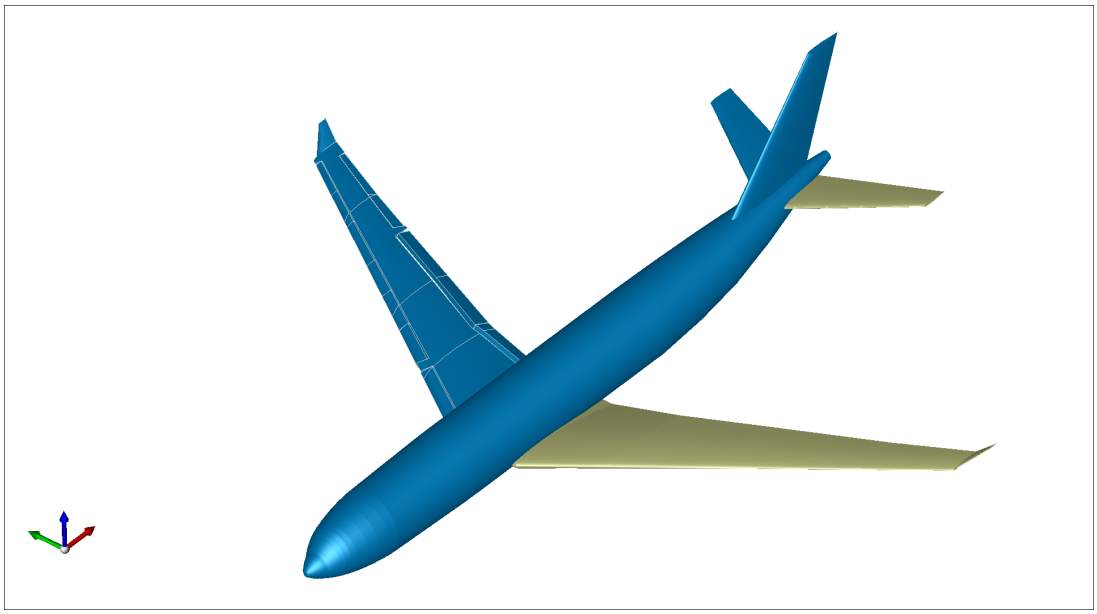


Abbildung 19: Flugzeug mit Landeklappen

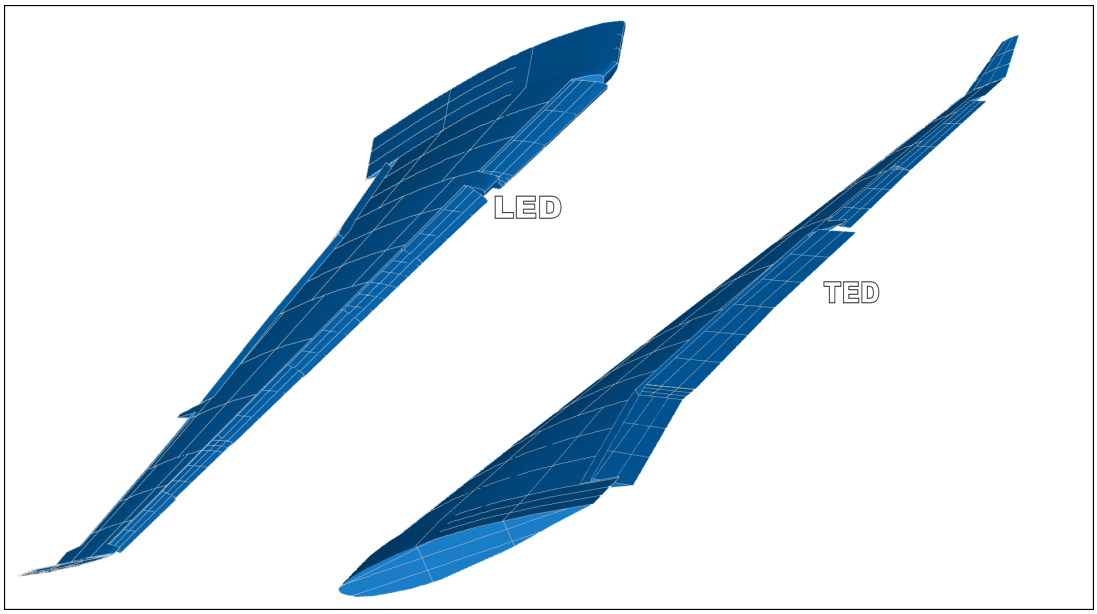


Abbildung 20: Flügel mit TrailingEdgeDevices und LeadingEdgeDevices

Collapse All

CPACSdoc

trailingEdgeDevice Element

See Also Send Feedback

Trailing edge device of the wing.

Namespace: Empty
Schema: cpacs_schema.xsd

Type

trailingEdgeDeviceType

Parents

trailingEdgeDevices trailingEdgeDevicesType

Children

Name	Occurrences	Description
Sequence		
name		Name of the trailing edge device.
description	[0, 1]	Description of the trailing edge device.
parentUID		UID of the parent of the TED. The parent can either be the uID of the componentSegment of the wing, or the uID of another TED. In the second case this TED is placed within the other TED (double slotted flap). In this way n-slotted TEDs can be created.
outerShape		Outer shape definition of the control surface.
wingCutOut	[0, 1]	Cut out of the parents structure due to a control surface.
structure	[0, 1]	Structure of the wing
path		Definition of the deflection path of the control surface.
tracks	[0, 1]	Control surface tracks (mechnaical link between control surface and parent).
actuators	[0, 1]	Definition of actuators of the control surface, that are not placed within a track.
cruiseRollers	[0, 1]	Definition of cruise rollers/mid-span stops. Those features are small rolls at the leading edge of a flap that keep the flap within the bending wing at cruise configuration.
interconnectionStruts	[0, 1]	Definition of interconnection struts. Those struts connect two neighbouring flaps and are load carrying in case of an actuator of flap track failour.
zCouplings	[0, 1]	Definiton of z-couplings. Those elements couple two neighbouring flaps in z-direction.

Attributes

Name	Type	Required	Description
externalDataDirectory	string		
externalDataNodePath	string		
externalFileName	string		
uID	string		

Remarks

A trailingEdgeDevice (TED) is defined via its outerShape relative to the componentSegment. The WingCutOut defines the area of the skin that is removed by the TED. Structure is similar to the wing structure. The mechanical links between the TED and the parrent are defined in tracks. The deflection path is described in path. Additional actuators, that are not included into a track, can be defined in actuators.

See Also

Reference

trailingEdgeDevicesType

Send comments on this topic to daniel.boehnke@dlr.de

Collapse All

CPACSdoc

path Element

See Also Send Feedback

Definition of the deflection path of the control surface.

Namespace: Empty
Schema: cpacs_schema.xsd

Type

controlSurfacePathType

Parents

leadingEdgeDevice leadingEdgeDeviceType

Children

Name	Occurrences	Description
Sequence		
innerHingePoint		controlSurfaceHingePointType
outerHingePoint		controlSurfaceHingePointType
steps		Definition of the steps of the control surface deflection path.

Attributes

Name	Type	Required	Description
externalDataDirectory	string		
externalDataNodePath	string		
externalFileName	string		

Remarks

See Also

Reference

leadingEdgeDeviceType

Send comments on this topic to daniel.boehnke@dlr.de

Collapse All

CPACSdoc

step Element

See Also Send Feedback

controlSurfaceStepType

Namespace: Empty
Schema: cpacs_schema.xsd

Type

controlSurfaceStepType

Parents

controlSurfaceStepsType steps

Children

Name	Occurrences	Description
Sequence		
relDeflection		Relative deflection. This value is an double value and must be unique with all steps. Can be seen as a kind of 'uID' of the step. The value can have any range and is not limited from 0 to 1. 0 means no deflection. The values of relDeflection define the order of the different steps (0 to the highest relDeflection for the positive deflection; 0 to the lowest (negative) value for the negative deflection (if any).
innerHingeTranslation	[0, 1]	Translation of the inner hinge line point within the hinge line coordinate system. Defaults to zero. Not allowed for spoilers!
outerHingeTranslation	[0, 1]	Translation of the outer hinge line point within the hinge line coordinate system. Defaults to the values of the inner hinge line point. Not allowed for spoilers!
hingeLineRotation	[0, 1]	Positve rotation around the hinge line, heading from the inner to the outer border. Defaults to zero.

Attributes

Name	Type	Required	Description
externalDataDirectory	string		
externalDataNodePath	string		
externalFileName	string		

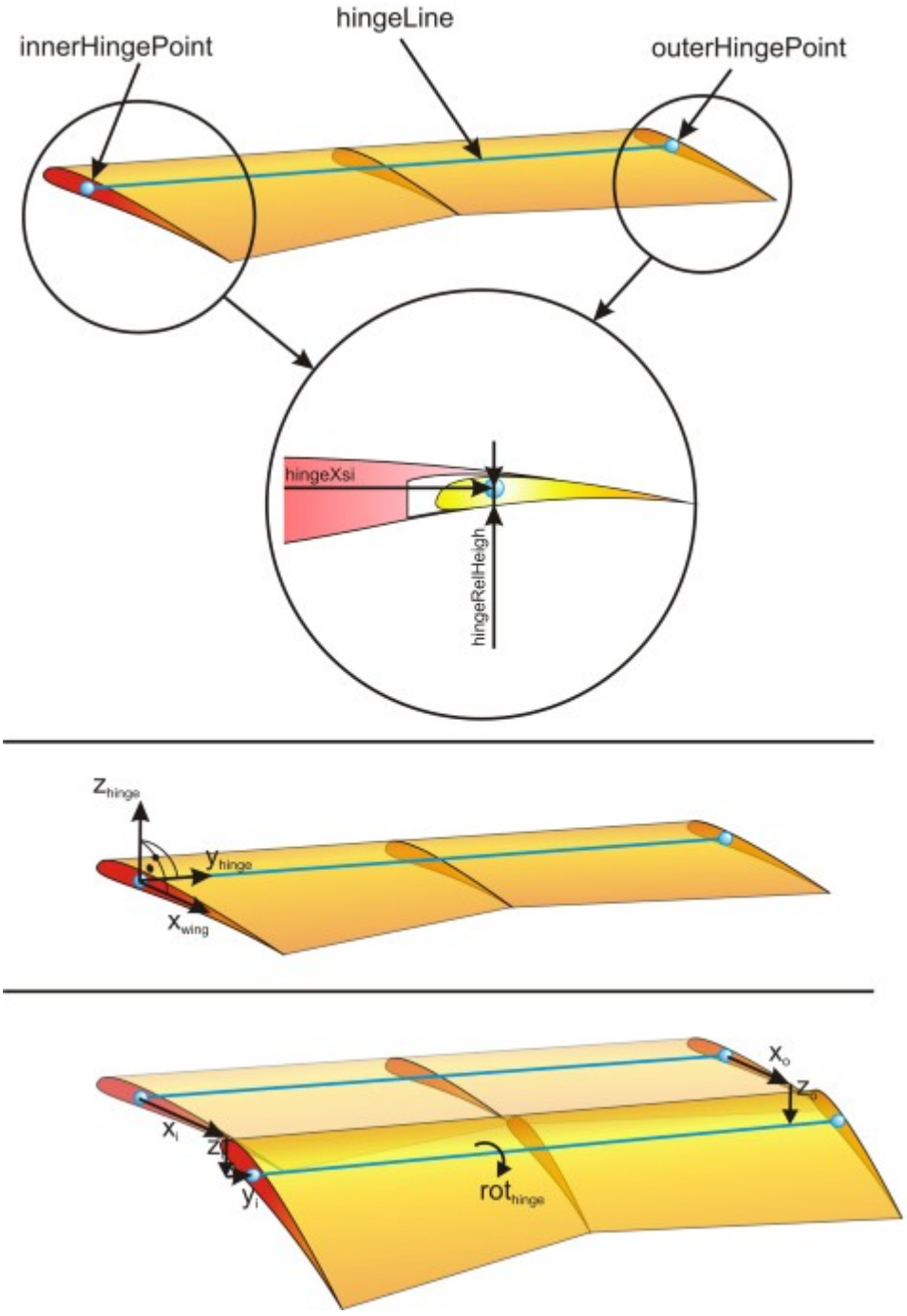
Remarks

The deflection path of the control surface is defined within the hinge line coordinat system. This is defined as follows: The x-hinge coordinate equals the wing x-axis. The y-hinge coordinate equals

the hinge line axis (see above; positive from inner to outer hinge point). The z-hinge line is perpendicular on the x-hinge and y-hinge coordinate according to the right hand rule. The rotation of the control surface is defined as rotation around the positive y-hinge line.

The deflection of the is defined in any number of steps. The deflection of the control surface is done as follows: First the x-deflection at the inner and outer border; afterwards the z-deflection of the inner and outer border; last the y-deflection of the inner border. The y-deflection is only defined at the inner border, as it is identical to the outer border. If no values for the outer border deflection are given, they default to the values of the inner border.

An example can be found below:



See Also

Reference

controlSurfaceStepsType