Entwicklung einer Software zur Generierung von dreidimensionalen Flugführungsanzeigen

Dem Fachbereich Maschinenbau
der Technischen Universität Darmstadt
zur
Erlangung des Grades eines Doktor–Ingenieurs (Dr.–Ing.)
genehmigte

Dissertation

vorgelegt von

Guillaume Smietanski
aus Paris

Berichterstatter :	Prof. Dr.–Ing. W. Kubbat
Mitberichterstatter :	Prof. Dr.–Ing. R. Anderl
Tag der Einreichung :	28.08.2002
Tag der mündlichen Prüfung :	05.03.2003

D 17

Entwicklung einer Software zur Generierung von dreidimensionalen Flugführungsanzeigen vorgelegt von Guillaume Smietanski.

Um die wachsende Zahl von Fluganzeigen zu reduzieren, wurden in der Vergangenheit die zentralen Fluginstrumente entwickelt, die eine große Zahl von Anzeigen in sich vereinen. Verschiedene Forschungen haben gezeigt, dass mit der Erweiterung um eine dreidimensionale Geländedarstellung hin zu einem *Synthetic Vision System* (SVS) das Situationsbewusstsein der Piloten verbessert wird.

Am Institut für Flugsysteme und Regelungstechnik der Technischen Universität Darmstadt wurden Mitte der neunziger Jahre SVS-Flugführungsanzeigen mit dreidimensionalen Elementen erarbeitet. Die ursprüngliche Version der Anzeigen des Fachgebiets diente als erste Versuchsplattform und bestätigte einerseits deren Potential, andererseits die Notwendigkeit weiterer Forschungsarbeit zu dreidimensionalen Flugführungsanzeigen.

Durch die Ergebnisse und einen zunehmenden Nutzerkreis wurde es erforderlich, eine zweite Generation der Flugführungsanzeigen zu entwickeln. Diese Software bildet die Grundlage, zur Entwicklung und Erprobung neuer Mensch-Maschine-Schnittstellen. Dies stellt zusätzliche Anforderungen im Vergleich zu einer Software, deren Entwicklung abgeschlossen ist. Die neue Generation, die in dieser Arbeit vorgestellt wird, ermöglicht neben anderem:

- die Flugführungsanzeigen in verschiedenen Umgebungen zu erproben,
- eine einfache Erweiterung (z.B. 2D/3D Symbolik, Kommunikationsschnittstelle etc.),
- neue Anzeigenformate,
- und die einfache Erzeugung von Varianten für Forschungszwecke.

Die Struktur und Realisierung der Hauptkomponenten dieser objektorientierten Flugführungsanzeigensoftware werden erlautert:

- die Generierung der konventionellen zweidimensionalen Darstellung,
- die dreidimensionalen Elemente und die Verwaltung einer Datenbank zur Geländedarstellung,
- die notwendigen Kommunikationsschnittstellen,
- und die Interaktion zwischen diesen Komponenten.

Die Beschreibung erfolgt mit Hilfe der *Unified Modelling Language* (UML).

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet Flugsysteme und Regelungstechnik der Technischen Universität Darmstadt.

Ich danke Herrn Professor Dr.-Ing. W. Kubbat, dem Leiter des Fachgebiets, für die Ermunterung und besondere Unterstützung, die er mir während der Erstellung dieser Arbeit zuteil werden ließ.

Für die freundliche Übernahme des Koreferates möchte ich mich bei Herrn Professor Dr.-Ing. R. Anderl bedanken.

Ich danke weiterhin den wissenschaftlichen Mitarbeiterinnen und Mitarbeitern des Fachgebiets und allen Studenten, die zum Gelingen dieser Arbeit beitragen haben.

Ich hatte das Vergnügen, mit Arnd Helmetag, Jochen Kaiser, Peter Lenhart und Udo Mayer zu arbeiten und wertvolle Freizeit zu haben. Dafür bin ich ihnen dankbar. Ich danke besonders Arnd Helmetag, der mich Professor Dr.-Ing. W. Kubbat vorgestelt hat, und der mich auch in schwierigen Momenten unterstützt hat.

Mein besonderer Dank gilt Bettina und Pierre-Yves Crepin für viele anregende und fruchtbare Diskussionen.

Nicht zuletzt möchte ich meinen Eltern und meiner Schwester danken, die an mich glaubten und immer da waren, wenn ich ihre Hilfe brauchte. Sie haben meine Entscheidungen immer unterstützt und mich gefördert.

Ich versichere an Eides statt, dass ich diese Arbeit mit Ausnahme der ausdrücklich erwähnten Hilfen selbständig durchgeführt habe.

Darmstadt, im August 2002

(Guillaume Smietanski)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Abkürzungen

2D	Zweidimension
3D	Dreidimension
ADF	Automatic Direction Finder
ATTAS	Advanced Technologies Testing Aircraft System
AWARD	All Weather ARrival and Deparature
CDTI	Cockpit Display of Traffic Information
CFIT	Controlled Flight Into Terrain
CORBA	Common Object Request Broker Architecture
CRJ	Canadair Regional Jet
CSMA	Carrier Sense Multiple Access
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
DASA	DaimlerChrysler Aerospace
DH	Decision Height
DME	Distance Measuring Equipment
DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
ECAM	Electronic Centralized Aircraft Monitoring
EADS	European Aeronautic Defense and Space Company
EFIS	Electronic Flight Instrument System
EGT	Exhaust Gas Temperature
EVS	Enhanced Vision System
FDDI	Fiber Distributed Data Interface
FMA	Flight and Mode Annunciator
FMS	Flight Management System
FSF	Fligt Safety Foundation
FSR	Flugsysteme und Regelungstechnik Institut
FTP	File Transfert Protocol
GPS	Global Positioning System
GPWS	Ground Proximity Warning System
HDPFD	Head Down Primary Flight Display
HLA	High Level Architecture
HMD	Head Mounted Display
HSD	Horizontal Situation Display
HUD	Head Up Display
ICAO	International Civil Aviation Organization
ILS	Instrument Landing System
IP	Internet Protocol
ISAS	Integrated Situation Awareness System
ISAWARE	Increasing Safety through collision Avoidance WARning intEgration

LAN	Local Area Network
LCD	Liquid Crystal Display
LOD	Level of Detail
MAN	Metropolitan Area Network
MDA	Minimum Descend Altitude
MFD	Multifunction Display
MORA	Minimum Off Route Altitude
ND	Navigation Display
NLR	Netherlands National Aerospace Laboratory
OMG	Object Management Group
OOP	Objektorientierte Programmierung
OSI	Open System Interconnection
pfDCS	Performer Knoten: *Dynamic Coordinate System*
PPP	Point to Point Protocol
RFM	Reflective Memory
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SVS	Synthetic Vision System
TCAS	Traffic Collision Avoidance System
TCL	Terrain Clearance Line
TCP	Transmission Control Protocol
TIN	Triangular Irregular Network
TUD	Technische Universität Darmstadt
UDP	User Datagram Protocol
UML	Unified Modelling Language
VAPS	Virtual Prototypes Application
VOR	Very High Frequency Omnidirectional Radio Range
VPI	Virtual Prototypes Inc.
VSD	Vertical Situation Display
WAN	Wide Area Network
WGS	World Geodaetic System

Begriffserläuterungen

Abgeleitete Klasse (OOP)
Siehe **Unterklasse**.

Abstrakte Klasse (OOP)
Eine abstrakte Klasse ist eine **Klasse**, die eine oder mehrere **pur virtuelle Methoden** hat; keine **Instanz** solcher Klasse ist möglich.

Aggregation (UML)
Eine besondere Variante der **Assoziation** ist die Aggregation. Hierbei handelt es sich ebenfalls um eine Beziehung zwischen zwei **Klassen**, jedoch mit der Besonderheit, dass die Klassen zueinander in Beziehung stehen wie ein Ganzes zu seinen Teilen [Oes99].

Aktor (UML)
Der Anwender wird als sogenannter Aktor dargestellt. Ein Aktor ist genaugenommen eine außerhalb des Systems liegende "Klasse", d.h., es könnte sich auch um ein externes, angeschlossenes System handeln [HKK01]. Ein Aktor kann eine Person, eine Klasse oder ein Teil einer Software sein.

Anwendungsfall (UML)
Ein Anwendungsfall beschreibt eine typische Interaktion zwischen dem Anwender und dem System, d.h. er stellt das externe Systemverhalten aus der Sicht des Anwenders dar.

Anwendungsfalldiagramm (UML)
Ein Anwendungsfalldiagramm (engl. *use case diagram*) zeigt den Zusammenhang zwischen **Anwendungsfällen** und den daran beteiligten Aktoren [HKK01].

Assoziation (UML)
Eine Assoziation ist eine Beziehung zwischen Instanzen von zwei oder mehreren Klassen. Durch eine Verknüpfung wird ein logischer Zusammenhang definiert ("use-a" Beziehung) [Ach97].

Attribut (OOP)
Siehe **Parameter**.

Ausnahme (OOP)

Eine Ausnahme ist, vereinfacht gesagt, das Auftreten einer abnormalen Bedingung bei der Programmausführung. Im Gegensatz dazu wird mit dem Begriff *Fehler* die Situation beschrieben, die eintritt, wenn Software ihre Spezifikation nicht erfüllt. Tritt bei der Programmausführung eine Ausnahme auf (zum Beispiel Division durch 0, Speicher erschöpft etc.) so gibt es im allgemeinen drei Möglichkeiten, darauf zu reagieren [Str97]:

- Ignorieren: Wenn eine Ausnahme auftritt, so wird sie ignoriert
- Abbruch: Beim Auftreten einer Ausnahme terminiert das Programm mit einer entsprechenden Meldung.
- Verwendung von Fehlercodes: Funktionen geben durch einen Rückgabewert (oder einen Parameter) an, ob ihre Ausführung erfolgreich war.

Basisklasse (OOP)

Eine Basisklasse ist eine **Klasse** innerhalb einer Vererbungsbeziehung, von der eine abgeleitete Klasse Eigenschaften erbt (Verallgemeinerung von abgeleiteten Klassen) [Ach97].

Dynamische Bindung (OOP)

Beim Aufruf einer **virtuelle Methode** wird erst zur Laufzeit des Programms entschieden, welche von mehreren möglichen Methoden ausgeführt wird [Ach97].

Freund (eng. friend -OOP-)

In C++ ist es möglich, dass eine Klasse bestimmten Funktionen oder auch Methoden anderer Klassen den Zugriff auf alle ihre Elemente (auch die **private**-Elemente) erlauben kann. Funktionen bzw. Klassen, denen dies mögklich sein soll, werden als sogenannte *Freunde* der Klasse angegeben [HKK01].

Funktion

Eine Funktion ist die Implementierung einer Berechnung. Im Gegesatz zu einer Methode gehört eine Funktion keiner Klasse an.

Instanz

Eine Instanz ist ein Abbild eines Objekts in einem Programm [Ach97]. Siehe **Objekt**.

Klasse (OOP)

Klassen sind rein statische Beschreibungen einer Menge möglicher Objekte - der Exemplare dieser Klasse. Eine Klasse ist ein erkennbares Element eines Programmtextes. Im Gegensatz dazu ist ein **Objekt** ein gänzlich *dynamisches* Konzept, das nicht zum Programmtext, sondern zum Speicher des Rechners gehört [Mey90].

Klassendiagramm (UML)

Ein Klassendiagramm für mehrere Klassen zeigt die statische Beziehung zwischen den einzelnen Klassen [HKK01].

Klassenparameter (OOP)

Es gibt Fälle, in denen Daten nicht *objektbezogen*, sondern *klassenbezogen* benötig werden, das heißt, dass die Daten nur einmal pro **Klasse** existieren und nicht für jedes Objekt in einer eigenen Ausfertigung. Ein solches Datenelement wird als statisch bezeichnet. Auf ein statisches Datenelement können alle Instanzen der Klasse gemeinsam zugreifen.

Klassenmethode (OOP)

Eine Klassenmethode ist die Implementierung einer *Methode* einer **Klasse**, die nur mit **Klassenparametern** arbeiten darf. Man kann eine Klassenmethode ohne Instanz aufrufen. Eine Klassenmethode betrifft nicht nur eine Instanz, sondern alle Instanzen und auch künftigen Instanzen einer **Klasse**.

Komposition (UML)

Eine besondere Form der **Aggregation** liegt vor, wenn die Einzelteile vom Aggregat (dem Ganzen) existenzabhängig sind; dieser Fall wird Komposition genannt [Oes99]. Wenn das Ganze also gelöscht werden soll, werden auch alle existenzabhängigen Einzelteile mitgelöscht.

Konstruktor (OOP)

Ein Konstruktor ist eine **Methode**, die am Anfang der Lebensdauer einer *Instanz* automatisch ausgeführt wird und den nötigen Speicherplatz anlegt. Außerdem kann ein Konstruktor beliebige Initialisierungen vornehmen[Ach97]. Der Konstruktor kann bei Bedarf Parameter übernehmen, aber keinen Rückgabewert liefern - nicht einmal *void* [JL00].

Mehrfachvererbung (OOP)

Die Mehrfachvererbung ist eine erweiterte Form der **Vererbung**, bei der eine **Klasse** von mehreren **Basisklassen** erben kann [Ach97].

Methode (OOP)

Eine **Methode** ist die Implementierung einer Funktion eines Objekts [HKK01].

Nachricht (OOP)

Siehe Methode.

Objekt (OOP)

Objekte sind Laufzeitelemente, die während der Ausführung eines Systems erzeugt werden. Zur Laufzeit gibt es nur Objekte; im Programm sehen wir nur Klassen [Mey90]. Jedes Objekt ist ein Modell eines Gegenstandes aus der Anwendung[Ach97].

Operatoren (OOP)

Operatoren sind einzelne Sonderzeichen, beziehungsweise Sequenzen von Sonderzeichen, oder reservierte Wörter mit vordefinierter Bedeutung. Operatoren bestimmen Aktionen, die auf Programmobjekten ausgeführt werden können. [Str97]

Operator *delete* (OOP)

Der Operator delete wird dazu verwendet, vom System dynamischen Speicher freizugeben.

Operator *new* (OOP)

Der Operator new wird dazu verwendet, vom System dynamischen Speicher anzufordern.

Parameter (OOP)

Daten in **Klassen** werden in der objektorientierten Sprechweise als Attribute oder Parameter bezeichnet.

Polymorphie (OOP)

Eigenschaft einer Operation, sich bei Anwendung auf unterschiedliche Objekte unterschiedlich zu verhalten [Ach97].

Private (OOP)

Nur von innerhalb der Klasse (= von eigenen **Methoden**) und von **friend**-Klassen beziehungsweise -Methoden oder -Funktionen aus kann auf die private-Elemente (Methoden oder Parameter einer Klasse) zu gegriffen werden. Von außen sind diese Elemente nicht sichtbar. Auch von abgeleiteten Klassen aus ist kein Zugriff möglich [Str97].

Protected (OOP)

Protected-Elemente (Methoden oder Parameter einer Klasse) sind für **Methoden** der eigenen **Klasse** frei zugänglich. Zusätzlich kann von abgeleiteten Klassen (**Unterklasse**) auf protected-Elemente der eigenen **Basisklassen** zugegriffen werden [Str97].

Public (OOP)

Public-Elemente (Methoden oder Parameter einer Klasse) sind frei zugänglich und können sowohl von innerhalb der **Klasse** (= in den eingenen Element-**Methoden**) als auch von außerhalb (= von anderen Methoden oder Funktionen aus) angesprochen werden [Str97].

Pur virtuelle Methode (OOP)

Eine pur virtuelle Methode ist eine **Methode**, die in der **Basisklasse** keine Implementierung hat, die Implementierung ist in der **Unterklasse** definiert.

Sequenzdiagramm (UML)

Während ein **Klassendiagramm** die Beziehungen der **Klassen** untereienander in der Art eines Bauplanes darstellt (statisch), werden Sequenzdiagramme dazu verwendet, einen bestimmten Ablauf bzw. eine bestimmte Situation darzustellen (dynamisch). Ein Sequenzdiagramm gibt ein Szenario wieder und zeigt die einzelnen Methoden der Objekte untereinander, die nötig sind, um den gewählten Ablauf zu erreichen [HKK01].

Statische Methode (OOP)

Siehe **Klassenmethode**.

Statische Parameter (OOP)

Siehe **Klassenparameter**.

Überladen (OOP)

Definition mehrerer Varianten einer Operation, die sich durch die übergebenen Paramter unterscheiden [Ach97].

Unix

Das allgemeine Mehrbenutzer-Betriebsystem UNIX mit Teilnehmer- und Dialogbetrieb wurde ab 1969 in den USA bei Bell Laboratories (AT&T) in Assembler entwickelt und später zu ca. 90% in C umgeschrrieben. Bedingt durch die zugrundeliegende Schichtenarchitektur von UNIX ist das Betriebsystem bis auf ca. 10% Hardware-unabhängig. Diese

Eigenschaft der Portabilität, verbunden mit großer Flexibilität und Leistungsfähigkeit, hat zu einer starken Verbreitung von UNIX geführt.

Unterklasse (OOP)
Eine Klasse innerhalb einer Vererbungsbeziehung, die von einer **Basisklasse** Eigenschaften erbt (Spezialisierung einer Basisklasse) [Ach97].

Vererbung (OOP)
Mechanismus der Objektorientierung, durch den gemeinsame Eigenschaften mehrerer **Klassen** zu einer **Basisklasse** zusammengefasst werden können [Ach97].

Virtuelle Basisklasse
Wenn bei Mehrfachvererbung nicht erwünscht ist, dass mehrere Basisklassenobjekte erzeigt werden, können virtuelle Basisklassen verwendet werden. Von diesen Basisklassen wird nur ein Subobjekt erzeugt, auf das über verschiedene Vererbungswege zugegriffen werden kann [Bre99].

Virtuelle Methode (OOP)
Eine virtuelle Methode wird in einer **Basisklasse** definiert und in abgeleiteten Klassen überschrieben. Als virtuelle wird die **Methode** dann bezeichnet, wenn die Methodenresolution zur Laufzeit erfolgt [Ach97].

Zeiger *this* (OOP)
Jede **Methode** einer **Klasse** verfügt über einen versteckten Parameter: den Zeiger this, der auf das eigene **Objekt** zeigt. Die Aufgabe des Zeigers this besteht darin, auf das jeweilige Objekt zu verweisen, dessen Methode aufgerufen wurde [JL00].

UML Darstellung

In dieser Arbeit werden die folgenen UML-Diagramme verwendet:

- Paket-Diagramme
- Use-Case-Diagramme
- Klassendiagramme
- Sequenzdiagramme

Die folgene Darstellung beschreibt die benutzten UML-Diagramme. Die Begriffe sind auf Seite xii erklärt.

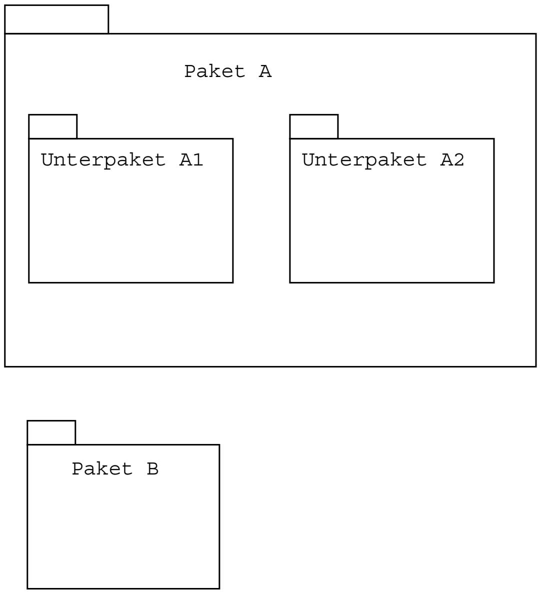


Abb. i: Paket Diagramme

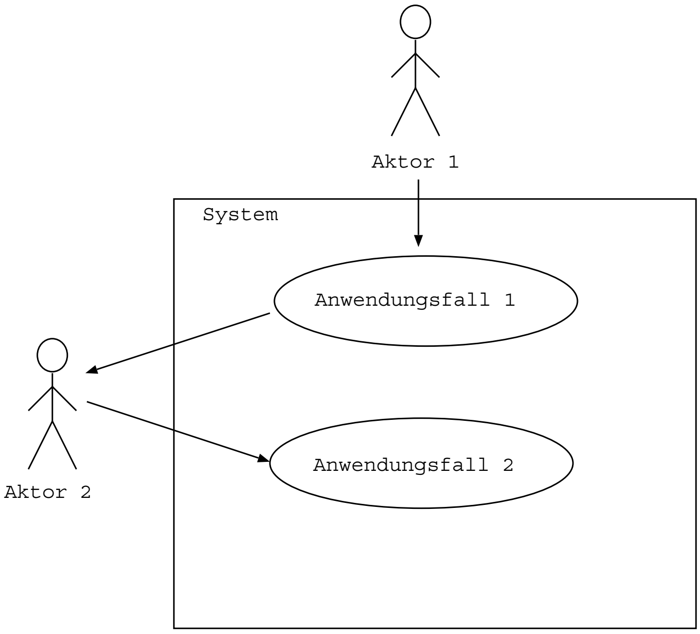
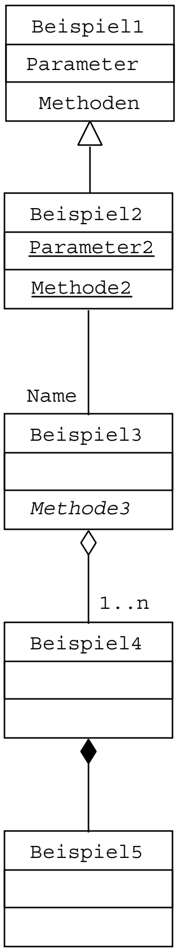


Abb. ii: Use Case Diagramme



Die Klasse *Beispiel1* mit ihren Parametern und Methoden.

Vererbung

Beispiel2 ist eine Unterklasse, oder Subklasse der Basisklasse, oder Superklasse *Beispiel1*. Parameter2 ist ein Klassenparameter und Methode2 eine Klassenmethode.

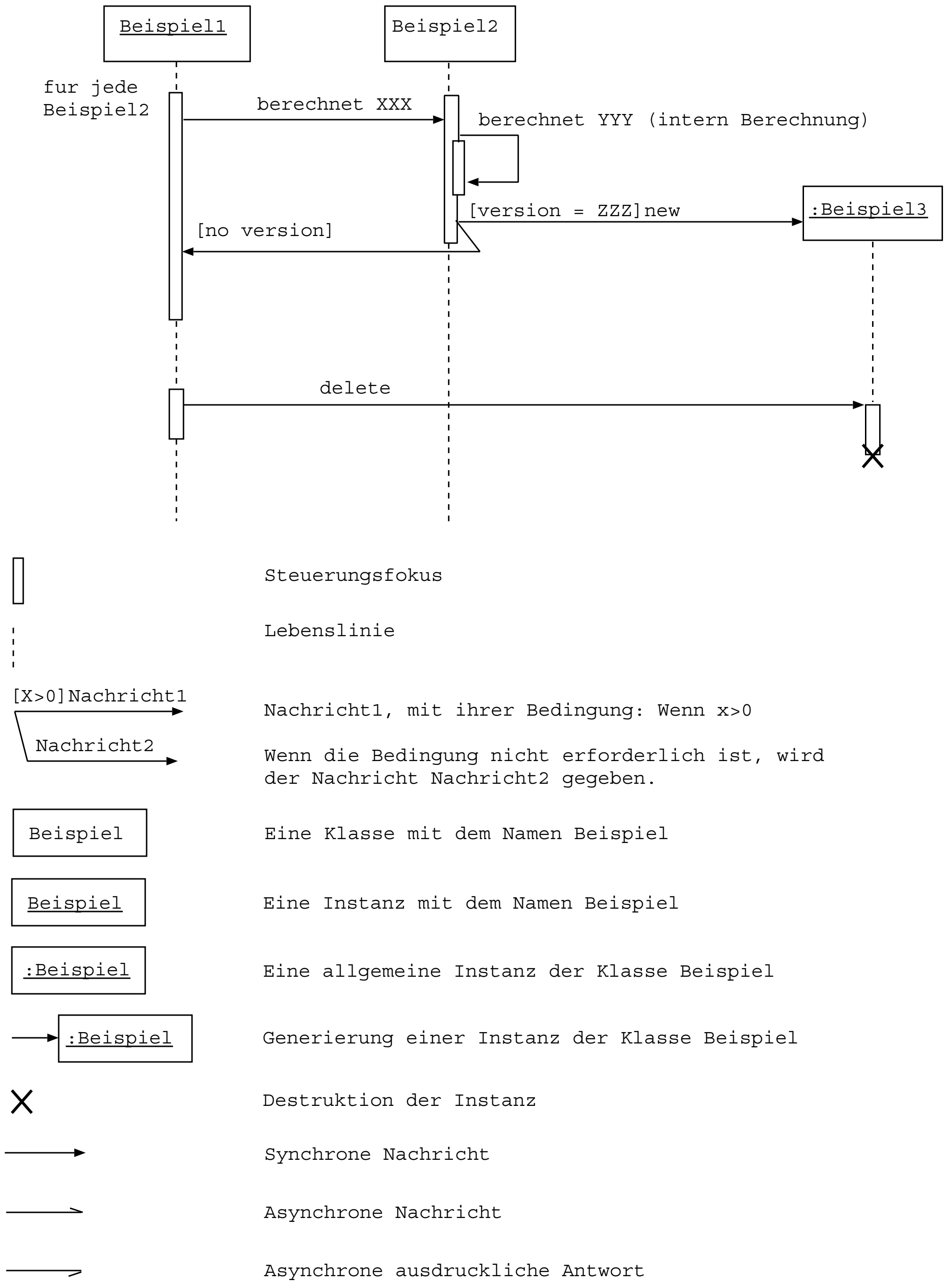
Assoziation zwischen der Klasse *Beispiel2* und *Beispiel3*. Der Name der Assoziation ist hier Name.

Methode3 ist eine pure virtuelle Methode.

Aggregation. Es gibt 1 bis n Aggregate.

Komposition. Es ist hier eine einzelne Komposition.

Abb. iii: Klassendiagramme



Steuerungsfokus

Lebenslinie

Nachricht1, mit ihrer Bedingung: Wenn x>0

Wenn die Bedingung nicht erforderlich ist, wird der Nachricht Nachricht2 gegeben.

Eine Klasse mit dem Namen Beispiel

Eine Instanz mit dem Namen Beispiel

Eine allgemeine Instanz der Klasse Beispiel

Generierung einer Instanz der Klasse Beispiel

Destruktion der Instanz

Synchrone Nachricht

Asynchrone Nachricht

Asynchrone ausdruckliche Antwort

Abb. iv: Sequenzdiagramme

1 Einleitung

Um die wachsende Zahl von Fluganzeigen zu reduzieren, wurden in der Vergangenheit die zentralen Fluginstrumente entwickelt, die eine große Zahl von Anzeigen in sich vereinen. Es handelt sich um den aus dem künstlichen Horizont entwickelten *attitude direction indicator* (ADI) und den aus der Radiokompassanzeige entwickelten *horizontal situation indicator* (HSI). Die wichtigsten Einzelfluganzeigen gruppieren Höhe, Vertikalgeschwindigkeit, Geschwindigkeit, Machzahl, Steuerkurs und Richtung zu Funkfeuern [Bro94].

Die elektromechanischen Zentralinstrumente wurden später im sogenannten *Glass Cockpit* durch elektronische Anzeigen (Bildröhren) ersetzt und erhielten die Bezeichnung *Primary Flight Display* (PFD) und *Navigation Display* (ND) (siehe Abb. 1.1).

Ein HUD[1] wird manchmal zusätzlich vor dem Piloten eingebaut (siehe [KO01]). Es ist eine durchsichtige Anzeige, die schon seit längerer Zeit in Militärflugzeugen benutzt wird. Sie zeigt die wichtigsten Informationen (etwa die gleichen Informationen, die auf dem PFD gezeigt werden) auf einer halbdurchsichtigen Scheibe vor dem Fenster des Cockpits an. So kann der Pilot ohne Kopfbewegung die Informationen erhalten und gleichzeitig die Außenwelt beobachten.

Verschiedene Forschungen haben die Überlegenheit einer dreidimensionalen Geländedarstellung in Flugführungsanzeigen gezeigt (siehe [(NL96], [The97], [RLS95] oder [HKLP00]).

Am Institut für Flugsysteme und Regelungstechnik der Technischen Universität Darmstadt wurden etwa Mitte der neunziger Jahre Flugführungsanzeigen mit dreidimensionalen Elementen, vor allem einer perspektivischen Geländedarstellung erarbeitet (siehe [HGK96], [HKP99] und Abb. 1.2). Dies erfolgte in Zusammenarbeit mit der Industrie, die eine flugtaugliche Anzeigehardware dazu entwickelte (siehe [SKP$^+$97]).

Einige Flugkampagnen und die Erprobung der ersten Generation dieser Anzeigen im Forschungssimulator des Instituts haben gezeigt, dass es möglich ist, mit solchen Anzeigen zu fliegen und dass solche Anzeigen bei den Piloten eine sehr große Akzeptanz finden, und dass das Situationsbewusstsein des Piloten deutlich verbessert werden kann [HKLP00].

1.1 Ziele der vorliegende Arbeit

Die ursprüngliche Version der Anzeigen des Fachgebiets diente als erste Versuchsplattform und bestätigte einerseits deren Potential, andererseits die Notwendigkeit weiterer

---

[1] Head Up Display

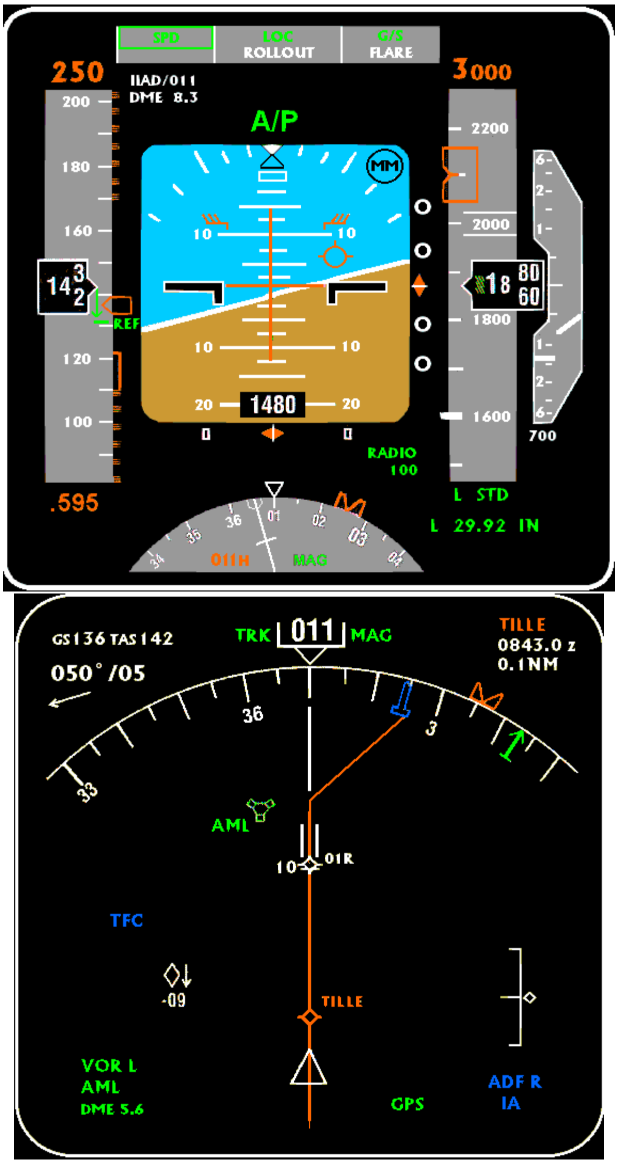


Abb. 1.1: Boeing 777 PFD und ND

Forschungsarbeit zu dreidimensionalen Flugführungsanzeigen.

Aufgrund einiger Einschränkungen der ersten Version und deren Prototypenstatus wurde entschieden, im Rahmen dieser Arbeit ein neues Softwarekonzept zu erarbeiten und zu implementieren, welches sich im wesentlichen durch eine erhöhte Flexibilität, optimierte Performance und nicht zuletzt durch eine verbesserte Pflegbarkeit und Erweiterbarkeit auszeichnen sollte.

Die Anzeigen der ersten Generation waren nur lokal begrenzt einsetzbar. Beim Start der Software wurde die vorhandene, lokale Geländeinformation komplett in den Speicher ge-

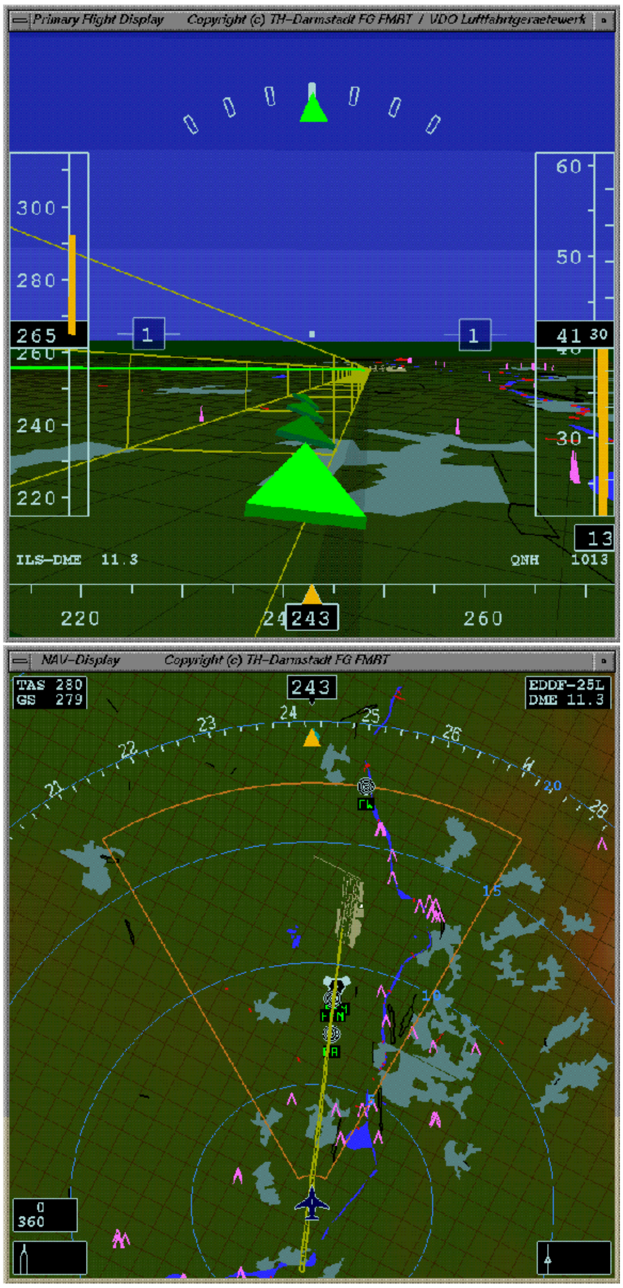


Abb. 1.2: Dreidimensionale Flugführungsanzeige

laden. Eine weltumfassende Geländedatenbank in ausreichender Präzision überschreitet aber auch heute noch die verfügbare Menge an RAM-Speicher, so dass der Ladevorgang der Geländedaten sukzessive zur Laufzeit erfolgen muss.

Eine weitere Beschränkung der Software lag in der geringen Flexibilität hinsichtlich der technischen Randbedingungen. Die erste Version der Anzeigen war ausschließlich für den Einsatz im Forschungscockpit des Fachgebiets und im Versuchsträger ATTAS entwickelt worden. Durch die positiven Evaluierungsergebnisse eröffneten sich weite Forschungsfelder, die jedoch eine Integration der Anzeigen in gänzlich anders strukturierte Umgebungen nötig machten. In der zu entwickelten Software sollten besonders flexible Schnittstellen bereitgestellt werden, die sich sowohl software- wie hardwareseitig in fremde Kommunikationsnetze einbinden lassen als auch unterschiedlichste Displayhardware unterstützen können.

Diese Software bildet das Grundelement, zur Entwicklung und Erprobung von neuen Mensch-Maschine-Schnittstellen. Abbildung 1.3 zeigt einen Zyklus der Entwicklung einer Anzeigengeneration. Dieser Prozess ist stark durch iteratives Vorgehen geprägt. Auch hieraus folgt die Forderung nach einer hohen Flexibilität und Erweiterbarkeit der Software.

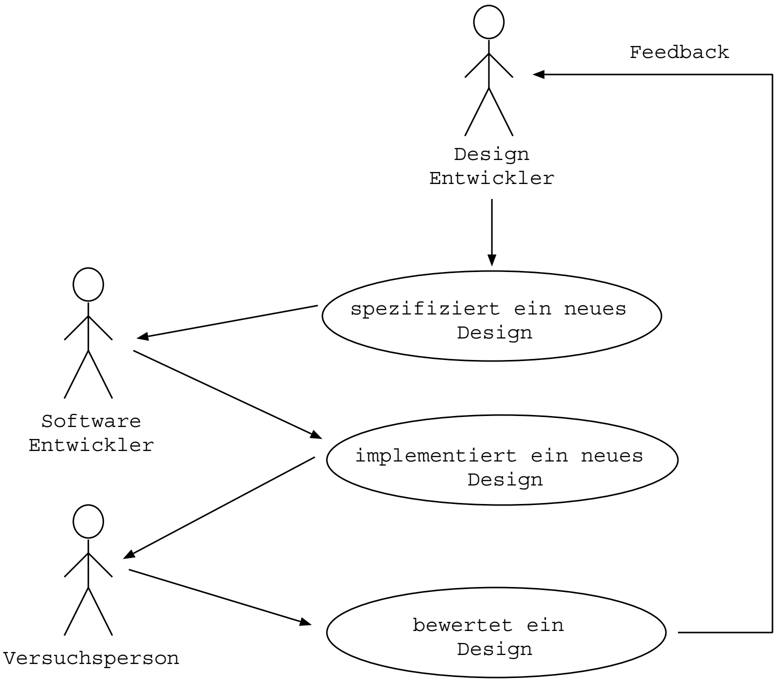


Abb. 1.3: Entwicklungszyklus

Die Flugführungsanzeige-Software muss nicht nur eine hohe Flexibilität besitzen, sondern ebenso eine zeitkritische Performance. Die Beurteilung der Software erfolgt durch die Piloten, die sehr schnell Informationen über die Lage des Flugzeugs benötigen, um entsprechend reagieren zu können.

1.2 Struktur der Arbeit

Eine dreidimensionale Flugführungsanzeige besteht aus vier Hauptkomponenten (siehe Abb. 1.4):

- den normalen Flugführungselementen, hier zweidimensionale Elemente genannt,
- den dreidimensionalen Elementen mit der Geländedarstellung,
- den Kommunikationsschnittstellen zwischen dem eigentlichen Display und anderen Anwendungen,
- und dem Programmkern, der die vorher genannten Teile verwaltet.

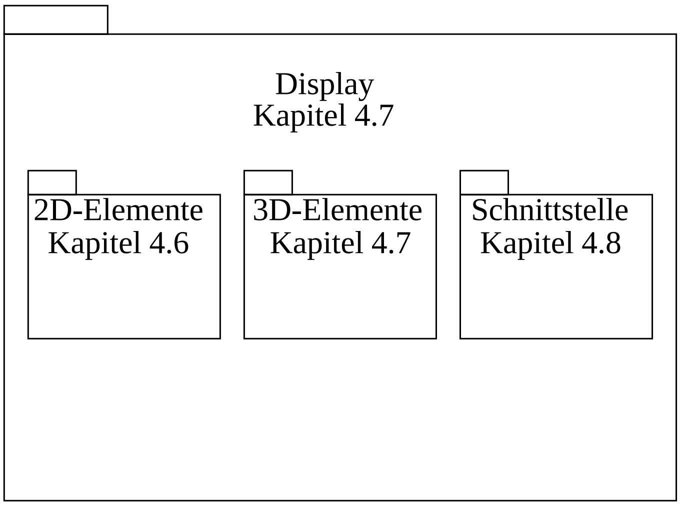


Abb. 1.4: Anwendungspaket

Im nächsten Kapitel werden die am Fachgebiet Flugsysteme und Regelungstechnick entwickelten Flugführungsanzeigen kurz beschrieben.

In Kapitel 3 wird auf die grafischen Anforderungen der Software eingegangen. Allgemeine Grundlagen der Kommunikationsschnittstelle werden dargestellt, die entsprechenden Anforderungen werden aufgelistet.

Kapitel 4 wird die System-Entwicklung erläutern. Dazu gehören die Softwareumgebung sowie einige wichtige computergrafische Funktionalitäten mit ihren Folgen für die Anwendung. Die Entwicklung der zweidimensionalen und dreidimensionalen Darstellung wird danach im Detail beschrieben. Es wird erläutert, warum für die zweidimensionale Darstellung, sowie für die Kommunikationsschnittstelle eine eigene Implementierung generiert wurde, anstatt hier fertige Produktlösungen zu verwenden.

In Kapitel 5 werden ausgehend von den spezifischen Anforderungen verschiedene Projekte vorgestellt in denen die entwickelten Flugführungsanzeigen eingesetzt wurden.

Die Arbeit schliesst mit einer Zusammenfassung und einem Ausblick in Kapitel 6.

2 Anzeige-Format

Auf Basis von Standard-Flugführungsanzeigen (HDPFD[1], HMD[2], HUD, ND und VSD[3]) entwickelte das Cavok-Team des FSR ein neues Konzept mit einer integrierten dreidimensionalen Geländedarstellung (siehe [HKP99]). Die Standard-Symbologien zeigen wichtige Informationen wie Geschwindigkeit, Höhe, Kurs usw. an. Diese Elemente werden, wie bei den bisher eingesetzten Flugführungsanzeigen, zweidimensional dargestellt.

Die dreidimensionale Darstellung zeigt unter anderem fremde Flugzeuge, *En Route* Kanäle und das überflogene Gelände. Hierzu werden zwei planare Projektionen, die perspektivische und die orthogonale Projektion, verwendet, die in Kapitel 4.4.2 genauer beschrieben werden. Die Übersichtstabelle 2.1 zeigt, welche Anzeige eine zweidimensionale und/oder dreidimensionale Darstellung aufweist und welche Projektionsmethode ggf. angewandt wird. .

	HDPFD	HUD	HMD	Visual	ND	VSD
2D	X	X	X	-	X	X
3D	X	-	X	X	X	X
Perspektivische	X	-	X	X	-	-
Orthogonal	-	-	-	-	X	X

Tab. 2.1: Übersichtstabelle der Cavok Flugführungsanzeige

Die folgenden Abschnitte (2.1 und 2.2) geben einen kurzen Überblick über den grafischen Inhalt der perspektivischen und orthogonalen Anzeige.

2.1 Perspektivische Anzeigen

Die perspektivische Anzeige ist bezüglich der Grafik hauptsätzlich in zwei Teile getrennt:

- Die zweidimensionalen Elemente. Ein Auszug wird in Abschnitt 2.1.1 gegeben.
- Die dreidimensionalen Elemente mit einer perspektivischen planaren Projektion, eine vereinfachte Beschreibung findet sich in Abschnitt 2.1.2.

---

[1] *Head Down Perspective Flight Display*

[2] *Head Mounted Display*

[3] *Vertical Situation Display*

2.1.1 2D-Formatelemente der perspektivischen Anzeigen

HDPFD, HUD und HMD benutzen im wesentlichen die gleiche zweidimensionale Funktionalität. Sie liefern Kurzzeitinformationen, das heißt, alle diese Displays sollen die aktuelle Lage des Flugzeugs beschreiben. Die Darstellung jedes Elementes kann von Anzeige zu Anzeige anders aussehen. Aber die Logik jedes Elementes bleibt bei allen Anzeigen praktisch gleich.

Auf diesen Anzeigen werden die folgenden Elemente dargestellt [AG]:

- Die Einstellungen und digitale Anzeigewerte:
    - FMA[4],
    - die Höhenmesser- und Barometereinstellung,
    - ILS Werte[5]
    - und die Machzahl
- Die Skalen:
    - Geschwindigkeitskala,
    - Höhenmesserskala,
    - Variometerskala,
    - Kursskala,
    - und ILS-Informationen
- Attitude-, und Führungsinformationen:
    - Pitch- und Rollskala mit künstlichem Horizont,
    - Slip-Indicator,
    - Flight Director,
    - Flight Path Vector
    - und Radiohöhenmesser.

Durch farbige Markierung oder zusätzliche Bildelemente werden in Skalen Sollwerte, Istwerte, Maximal- und Minimalwerte markiert [Bro94].

---

[4] Flight and Mode Annunciator

[5] Instrument Landing System

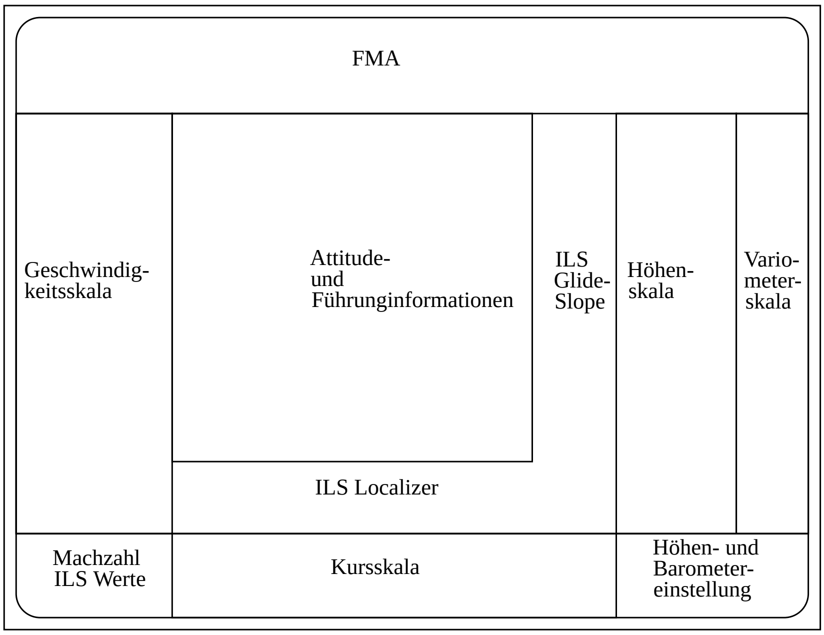


Abb. 2.1: Perspektivische 2D-Elemente

2.1.2 Darstellung des Hintergrunds bei perspektivischen Anzeigen

Auf dem *Primary Flight Display* wird das Gelände perspektivisch dargestellt. Die Blickrichtung der perspektivischen Projektion ist parallel zur Flugzeugachse, der Blickpunkt liegt im Cockpit. Wenn der Referenzpunkt für jeden Piloten auf der Achse seiner Augenposition wäre, ergäbe das zwei perspektivisch leicht verschobene Bilder (siehe Abb. 2.2).

Diese perspektivische Darstellung ändert sich grundsätzlich nur mit der Bewegung des Flugzeuges. Es gibt keine Zoomstufe und normalerweise eine konstante Blickrichtung: die Richtung der Flugzeuglängsachse. Diese Richtung ist nur bei einem *Head-Mounted-Display* nicht festgelegt: Die Sichtrichtung ändert sich dann mit der Bewegung des Flugzeugs, aber auch mit der Blickrichtung des Piloten. Ohne ein *Head-Tracking-System* würde die Sichtrichtung des Bildes im HMD immer der Längsachsen-Richtung des Flugzeuges entsprechen. Der Einsatz des Head-Tracking-Systems erlaubt dem Betrachter, Darstellungen der Blickrichtung zu erhalten, in die er den Kopf dreht. Die Referenzen der Sichtpunkte von einem HMD und von einem *Head Up Display* (HUD) sind nicht die gleichen wie beim *Primary Flight Display*. Da der Pilot gegenüber der Längsachse des Flugzeugs versetzt sitzt, ist die Blickrichtung eines HUD nicht identisch mit der Längsachse, sondern parallel dazu. Der Referenzpunkt liegt im Sichtfeld des Piloten. Bei einem HMD

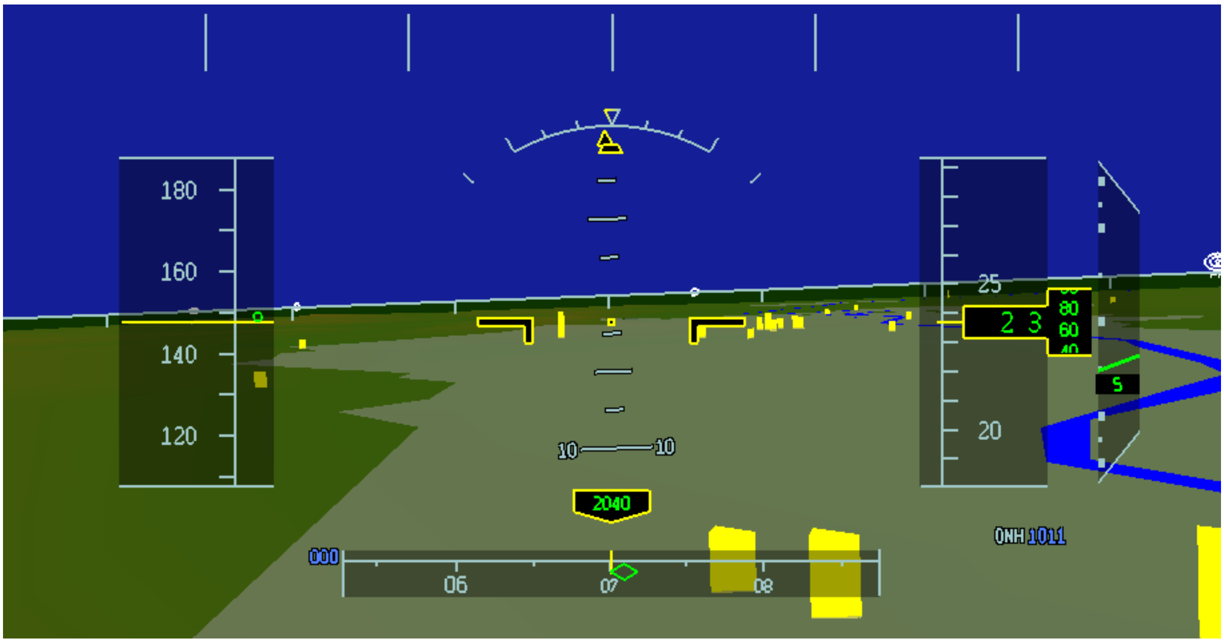


Abb. 2.2: 3D HDPFD

oder HUD muss die Darstellung möglichst exakt mit der Außenwelt übereinstimmen, da Parallaxen sehr störend sind.

Mit einer perspektivischen Projektion kann man Hindernisse entweder real nachmodellieren oder mit einem einheitlichen *Hindernissymbol* zeichnen. Auf den orthogonalen Anzeigen sind Hindernisse als Standard-Luftfahrt-Symbole abgebildet. Diese Symbole hängen nicht von der Größe des Hindernisses ab. Als Folgerung haben einige Symbole zwei Arten der Darstellung oder zwei Arten von Logik: Eine für die perspektivische Anzeige und eine andere für die orthogonale Anzeige (siehe Abb. 2.3).

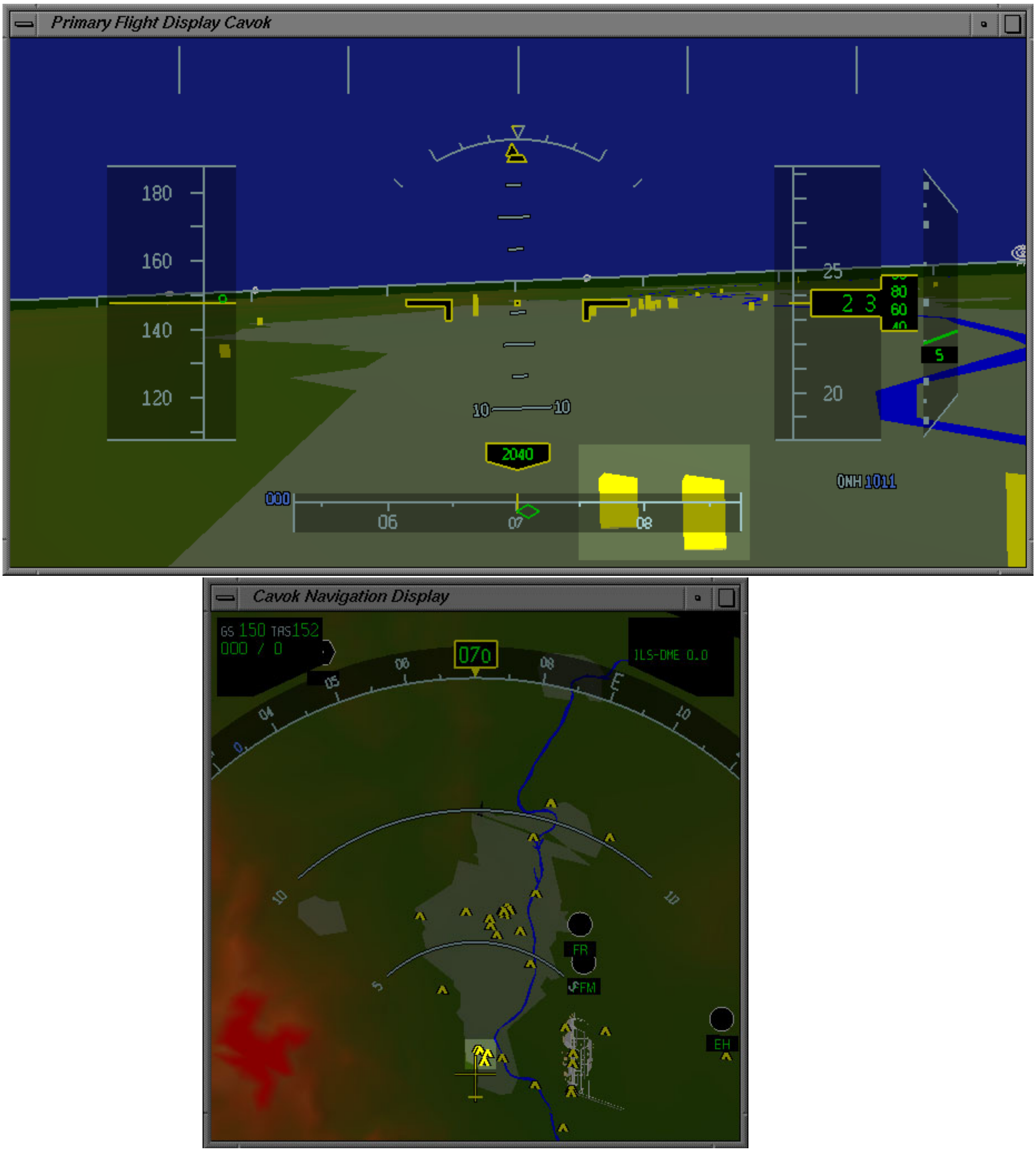


Abb. 2.3: Unterschiedliche Darstellung eines Symbols

2.2 Orthogonale Anzeigen

Wie die perspektivische Anzeige kann man die orthogonale Anzeige hauptsätzlich in zwei Teile trennen:

- Zweidimensionale Elemente, ein Auszug wird in Abschnitt 2.2.1 gegeben
- Dreidimensionalen Elemente mit einer orthogonalen planaren Projektion, eine vereinfachte Beschreibung ist in Abschnitt 2.2.2 gegeben.

2.2.1 2D-Formatelemente der othogonalen Anzeigen

Die orthogonalen Anzeigen liefern hauptsächlich mittelfristige Fluginformationen, und zwar die Position des Flugzeugs relativ zum vorgeplanten Flugweg und Wetterinformationen.

Es gibt mehrere orthogonale Anzeigen wie ND, VSD und CDTI[6]. Diese Arbeit ist auf die orthogonale Anzeige *Navigation Display* als Beispiel begrenzt. Die drei Hauptmodi des Navigation Displays sind:

- ROSE: eine 360 Grad Kompass-Rose-Darstellung. Das Symbol des eigenen Flugzeugs befindet sich in der Mitte der Anzeige.
- ARC: ein 90 Grad Teil der Kompass-Rose-Darstellung. Das Symbol des eigenen Flugzeugs befindet sich 1/8 der Anzeigehöhe über der unteren Kante des Fensters.
- PLAN: eine 360 Grad Kompass-Rose-Darstellung. Das Symbol des eigenen Flugzeugs befindet sich irgendwo auf der Anzeige und wird sogar manchmal nicht sichtbar bleiben. Diese Darstellung verwendet nicht den Kurs des Flugzeugs als Referenz für die Ausrichtung wie die zwei vorherigen Modi, sondern Nord wird hier immer Oben dargestellt.

Es werden in verschiedenen Zoomstufen dargestellt:

- Digitale Werte,
- Flugzeugssymbole,
- Kompassrose,

---

[6] Cockpit Display of Traffic Information

- Anzeigenadeln und
- Funknavigationssymbole

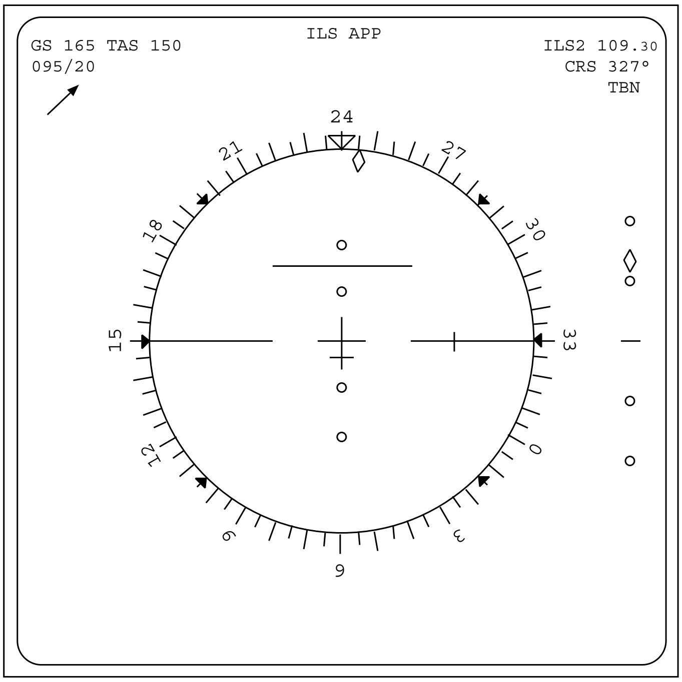


Abb. 2.4: Orthogonales 2D-Element im Rose ILS Mode

2.2.2 3D-Darstellung des Hintergrunds bei orthogonalen Anzeigen

Geländedarstellung

Im Hintergrund der Cavok-Displays ist das Gelände dargestellt. Es gibt zwei Arten von Darstellungen: eine für die orthogonale Anzeige und eine andere für die perspektivische Anzeige (siehe Abb. 2.5).

Die orthogonale Darstellung ist eine Sicht von oben (Draufsicht). Häufig wird eine nicht rotierende Karte verwendet: das Flugzeugsymbol dreht oder bewegt sich auf der Karte, und die Karte ist als North Up gezeichnet. Diese Lösung hat einige Vorteile, unter anderem ist es sehr einfach, die Informationen für diese Darstellung zu generieren. Eine Bitmap-Karte oder andere Karten, die zur Verfügung stehen, reichen aus.

Diese Lösung hat aber bezüglich der Sicherheit einen sehr großen Nachteil: Die Darstellung ist wenig intuitiv erfassbar. Ein Hindernis, das sich auf der linken Seite des Flugzeugs befindet, kann auf dem *Navigation Display* an verschiedenen Stellen erscheinen:

Abb. 2.5: 3D-Orthogonale Anzeige

Wenn die Karte nach Norden ausgerichtet ist und das Flugzeug nach Süden fliegt, würde dieses Hindernis auf der Anzeige rechts von Flugzeug angezeigt werden. Darum ist eine rotierende Karte besonders bei der Navigation des Flugzeugs vorteilhaft, da die nach Norden ausgerichtete Karte eine mentale Rotation erfordet, um den Piloten in die eigene Position zu versetzen [Kau98].

Die Beleuchtung einer Szene hilft, die Form von Objekten durch eine Kontrastierung deutlich sichtbar zu machen. Bei der Landschaftsabbildung werden die Konturen des Geländes durch die Wiedergabe einer gerichteten Lichtquelle deutlich hervorgehoben. Durch die Beleuchtung des Geländes in der Parallelprojektion gelangt man zu einer auch auf Landkarten verwendeten Schummerungsdarstellung, die einen reliefartigen Eindruck vermittelt und so die Plastizität des Geländes erhöht [Kau98].

Abhängig von der Position der Lichtquelle wird man kleine Schatten (vertikale Lichteinstrahlung) oder große Schatten (flache Lichteinstrahlung) sehen. Die Schatten und die Beleuchtung vergrößern die Plastizität einer Geländedarstellung: Mit einer flachen Lichteinstrahlung werden die charakteristischen Landschaftmerkmale hervorgehoben. Aber bei zu flacher Lichteinstrahlung entsteht das Risiko, dass ein Berg Täler verdeckt. Ein Kompromiss ist hier notwendig, oder eine dynamische Änderung, die von der Art des Geländes abhängt.

Die Lichtrichtung hat einen zusätzlichen Effekt. Es gibt Abbildungen von Reliefs, die - ob man will oder nicht - *invertiert* erscheinen, also bei denen alle Berge wie Täler und alle Täler wie Berge wirken. Die Abbildung 2.6 zeigt deutlich: die Berge auf dem ersten Bild sind als richtige Berge erkennbar. Trotz der Farbkodierung werden die gleichen Berge auf dem zweiten Bild fälschlicherweise als Täler empfunden. Zwischen den beiden Bilder ändert sich nur die Lichtrichtung: Auf dem ersten Bild steht die Lichtquelle oben links, und auf dem zweiten steht sie unten rechts.

Als Erklärung erläutert Metzger [Met75] unter anderem dass, im Freien das Licht zumeist von oben kommt. Die Menschen erwarten, dass die Lichtquelle einer Karte an der oberen linken Ecke steht. Wenn die Karte gedreht wird, muss die Beleuchtung trotzdem immer an der oberen linken Ecke bleiben.

Dies heißt also, dass die Beleuchtung des Geländes im Fall einer rotierenden Karte für jedes Bild neu berechnet werden muss. Daher ist es nicht mehr möglich, eine (Papier-) Karte einfach zu digitalisieren. Man muss statt dessen mit einem dreidimensionalen Modell des Geländes arbeiten, obwohl die endgültige Darstellung eine Parallelprojektion ist.

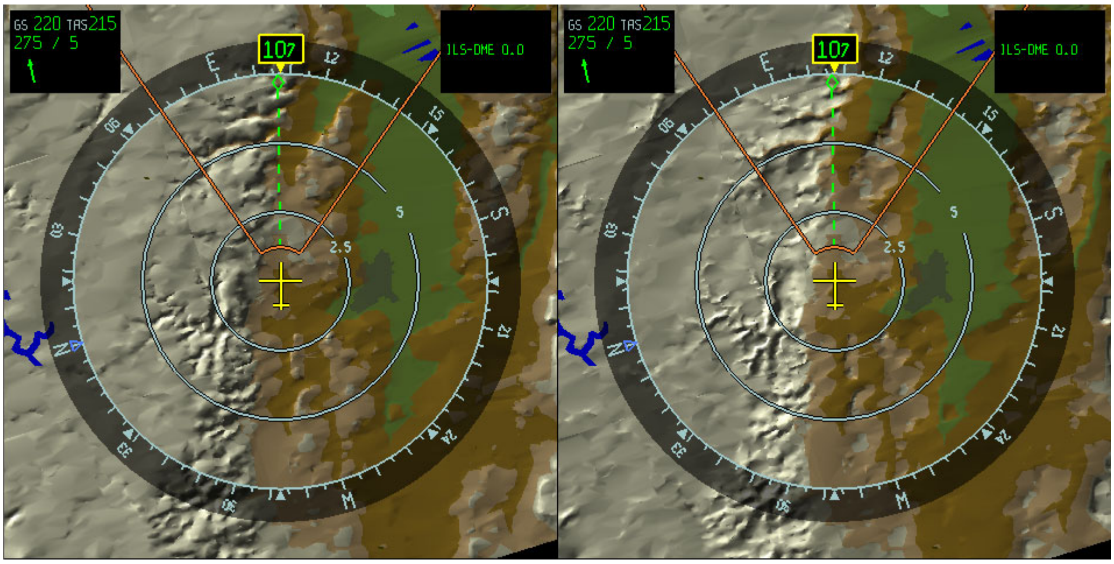


Abb. 2.6: Invertierung

Die Geländedarstellung im Hintergrund wird durch einige Informationen im Klartext ergänzt, wie zum Beispiel durch die Namen der Funkfeuer. Der Text muss auch bei einer Drehung der Karte durch eine Kursänderung immer lesbar bleiben, das heißt, dass er in der entgegengesetzten Richtung rotiert (siehe Abb. 2.7).

Der Text ist also dynamisch, er verschiebt sich absolut mit der Karte, aber dreht sich relativ zur Karte. Es kann sein, dass der Text eine wichtige Information verdeckt. Die Piloten müssen dann die Möglichkeit haben, diesen Texte auszuschalten. Wie der Text sollen auch einige andere Symbole ausschaltbar sein. Zum Beispiel sind Flüsse, Städte, Straßen oder

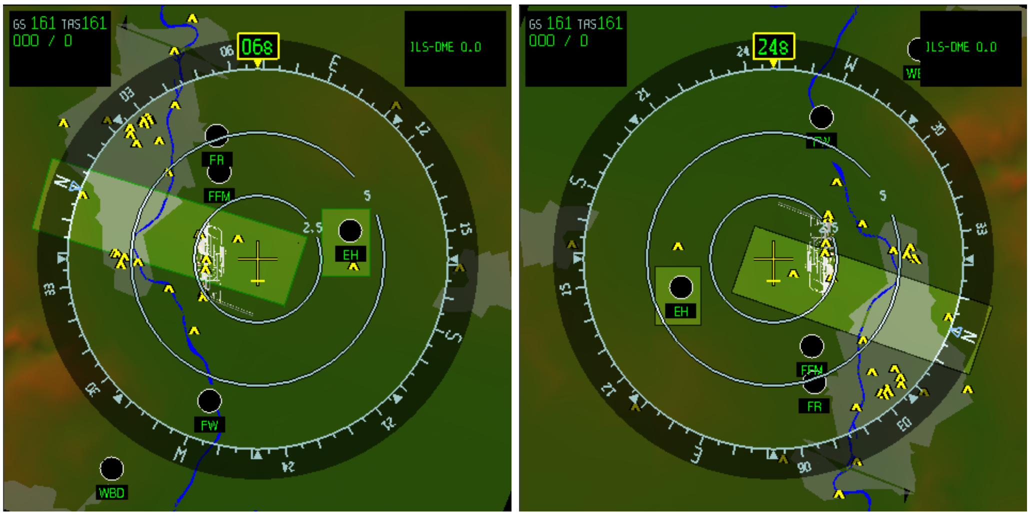


Abb. 2.7: *Navigation Display* mit Text

Eisenbahnen manchmal überflüssig. Deshalb müssen derartige Elemente einzeln und als Gruppe ausschaltbar sein.

Einfluss von ROSE und ARC Modus auf die Darstellung

In Kapitel 2.2.1 wurden die drei Hauptmodi des *Navigation Displays* genannt. ROSE-, ARC- und PLAN-Mode haben für die Geländedarstellung eine besondere Bedeutung.

Die *Heading up*-Darstellung (alle Modi außer PLAN) benutzen eine Richtungsreferenz, die mit der Längsachse des Flugzeuges identisch ist. Diese Referenz wird für die normale Navigation gewählt.

Unterschied zwischen ROSE-Modi und ARC-Modus für die Geländedarstellung

Das Symbol des eigenen Flugzeuges befindet sich nicht immer in der Mitte des *Navigations Displays* wie im Look-around Modus (dazu gehören alle ROSE-Modi). Diese ROSE-Modi sind insofern interessant, als man auch Informationen über die Situation hinter dem eigenen Flugzeug erhält.

Im Normalfall braucht der Pilot mehr Informationen über das vorausliegende Gebiet als vom Gebiet hinter dem Flugzeug. Daher gibt es einen Arc Modus oder Look-Forward-Modus. In dieser Darstellung befindet sich das Symbol für das eigene Flugzeug 1/8 der Anzeigehöhe über der unteren Kante des Fensters (siehe Abb. 2.8).

Einfluss des PLAN Modus auf die Darstellung

Mit der North-Up-Darstellung wird die Karte immer in Nordrichtung dargestellt. Wie der Name schon vermuten lässt, ist diese Darstellung für die Flugplanung gedacht.

Im Plan Modus kann sich die Referenz der Karte bewegen. Andernfalls könnte der Pilot nicht alle Wegpunkte sehen, wenn die geplante Flugstrecke auch in der maximalen Zoomstufe des *Navigation Displays* nicht komplett darstellbar ist. Aus diesem Grund befindet sich die Referenz auf einem Waypoint (anstelle des eigenen Flugzeuges) und man kann auf den nächsten Waypoint wechseln.

Einerseits kann sich die Referenz der Karte im Plan-Modus auf dem *Navigation Display* bewegen, anderseits befindet sich die Referenz der perspektivischen Anzeige immer dort, wo das Flugzeug gerade ist. Daher ist die Geländedarstellung der orthogonalen und der perspektivischen Displays im Plan-Modus des *Navigation Displays* nicht mehr konsistent. Weil der Pilot in diesem Fall mit dem ND arbeitet, hat das ND Vorrecht. Das HDPFD darf folglich kein Gelände mehr darstellen und sollte als Hinweis für den Piloten eine Warnung “PLAN MODE” anzeigen.

Die Abbildung 2.8 zeigt verschiedene Modi des *Navigation Display*.

Zoomstufe

Der Pilot benötigt auch verschiedene Zoomstufen im *Navigation Display*, die von der Flugphase und der Aufgabe abhängig sind. Wenn das Flugzeug rollt, braucht der Pilot einen sehr kleinen Maßstab, während des Reisefluges ist das Gegenteil der Fall (siehe 2.9).

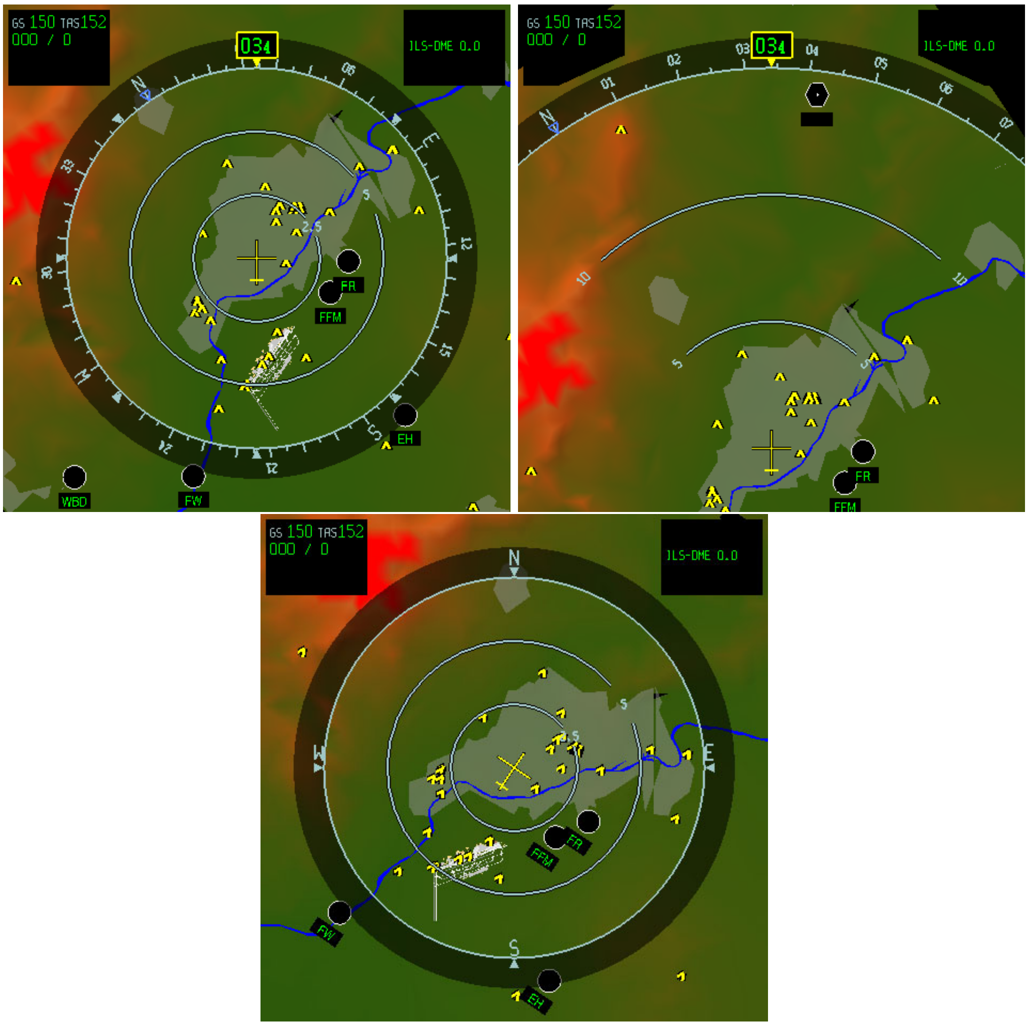


Abb. 2.8: Rose-, Arc- und Plan-Mode

Abb. 2.9: Ansicht des ND im ARC-Modus bei Unterschiedliche Zoomstufen

3 Anforderungen

In Kapitel 2 wurde die Flugführungsanzeige des FSR beschrieben. Das folgende Kapitel erläutert die Anforderungen für die Realisierung dieser Anzeige. Die Anforderungen der Entwickler und der Anwender werden zuerst erklärt, dann die Anforderungen an die funktionale Darstellung und zum Schluss die Kommunikationsanforderungen.

3.1 Anforderungen der Entwickler und der Anwender

3.1.1 Flexibilität

Die geplante Anwendung dient als Basis für die Entwicklung neuer Flugführungsanzeigen. Dabei ist zu berücksichtigen, dass sich der Entwicklungszyklus für verschiedene Anzeigeformate häufig wiederholt.

Wie Abbildung 3.1 zeigt, definiert ein Formatentwickler ein Format (oft in Kombination mit bereits vorhandenen Formaten), und wenn er dazu eine Softwareerweiterung braucht, wird er diese Änderung für den Softwareentwickler spezifizieren. Der Softwareentwickler erweitert die Anwendung, und der Formatentwickler wird sie durch Versuchspersonen bewerten lassen. Für die Bewertung selbst braucht der Formatentwickler viel Flexibilität, um den Versuchspersonen verschiedene Versionen anzuzeigen. Die im Rahmen dieser Arbeit konzipierte Anwendung wird in mehrere Pakete und Unterpakete aufgeteilt, und die Elemente eines Unterpakets sollen so weit wie möglich unabhängig von den anderen sein, trotzdem aber mit den anderen kompatibel bleiben. So wird es für einen Formatentwickler möglich, ein Element eines Unterpakets durch ein anderes auszutauschen. Wenn es möglich ist, soll dieser Austausch keine Neukompilation erfordern.

Um den Aufwand für die Softwareentwicklung zu minimieren, soll die Anwendung so einfach wie möglich erweiterbar sein. Aber sie soll auch genug Flexibilität besitzen, damit möglichst viele Versuche ohne Softwareerweiterung auskommen.

Für die Entwicklung und Erprobung soll diese Anwendung in verschiedenen Umgebungen lauffähig sein. Die Flugführungsanzeigen müssen also in verschiedenen Flugsimulatoren funktionieren, sollen aber auch bei realen Flugversuchen eingesetzt werden können. Daher muss die Kommunikationsschnittstelle der Software sehr viel Flexibilität bezüglich der Umgebung aufweisen.

Eine wichtige Besonderheit der im Rahmen dieser Arbeit erstellten Flugführungsanzeige ist die dreidimensionale Darstellung, für die eine Datenbank notwendig ist. Um die

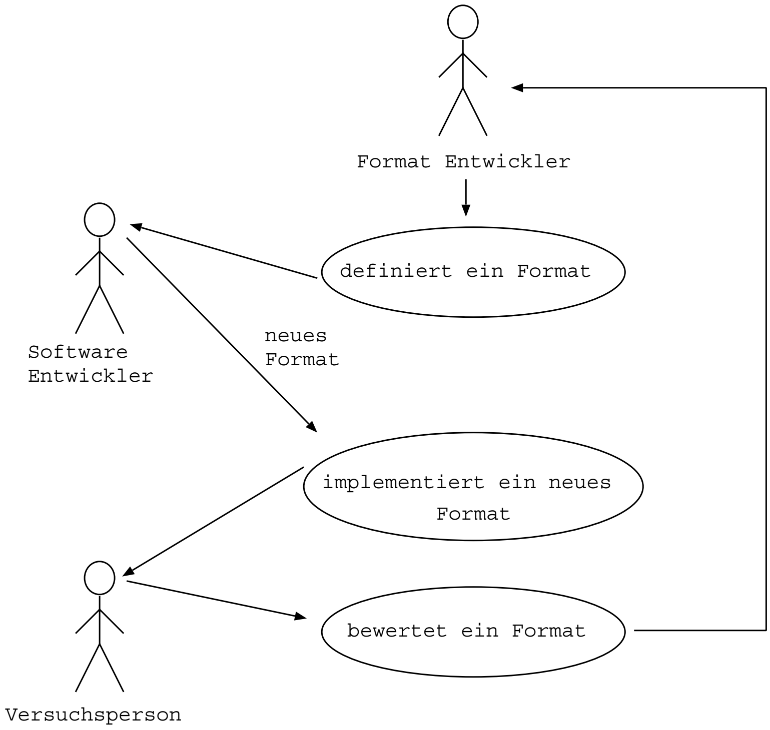


Abb. 3.1: Entwicklungsaktoren

Flexibilität auch bezüglich des 3D-Geländes zu ermöglichen, wurde ein am FSR entwickeltes neues Datenbankformat verwendet. Dieses erlaubt die flexible Modellierung und Generierung einer Datenbank, die später von der Anzeigesoftware gelesen wird. In dieser Kombination kann die Flugführungsanzeige mit Ausnahme von Nord- und Südpol weltweit verwendet werden. Eine Modifikation der Datenbank wird in Zukunft auch die Abbildung der Pole erlauben. Die Anzeigen sind nicht mehr auf einen Ort begrenzt und benutzen natürlich Welt- statt Lokalkoordinaten. Weiterhin beschreibt diese Arbeit, wie die Informationen der Datenbank zur Laufzeit geladen und dargestellt werden und wie die Software auf den Inhalt der Datenbank reagieren wird. Neue Flexibilität bezüglich der Datenbank wird gezeigt.

Mit Hilfe dieser Software können Flugführungsanzeigen für unterschiedliche Aufgaben erzeugt werden. Diese Arbeit wird diese Anzeigen nicht bewerten, sondern beschreiben, wie solche Anzeigen generiert werden können. Beispielhaft werden in dieser Arbeit verschiedene Anzeigen vorgestellt, wobei deren Besonderheiten bezüglich der Programmierung erläutert werden. Diese Arbeit wird die aktuell erwartete Flexibilität und ihre Implementierung erläutern.

3.1.2 Performance

Wie erwähnt, sollen diese Anzeigen in Flugversuchen eingesetzt werden. Das bedeutet, dass diese Anwendung nicht nur die Anforderung der Formatentwickler, sondern auch die des Endbenutzer, das heißt der Piloten, berücksichtigen muss. Aus Sicht der Piloten muss diese Anwendung eine hohe Bildfrequenz haben. Bei Untersuchungen für minimale Bildraten für die Durchführung einer Landung wurden 11Hz als gerade noch akzeptabel empfunden [HGK96]. Man muss daher versuchen, die Bildfrequenz so stark wie möglich zeitlich zu optimieren.

3.1.3 Modularer Systemaufbau

Diese Anwendung wird als Basis für die Implementierung neuer Anzeigen benutzt. Mit einem modularen Aufbau soll ein flexibler Einsatz für die Entwicklung neuer Anzeigen oder deren Erweiterung erreicht werden.

Meyer gibt in [Mey90] fünf Kriterien, die ein modulares System erfüllen muss. In dieser Arbeit sollen diese Kriterien als Grundregeln betrachtet werden. Die fünf Kriterien von Meyers sind folgende:

- Das Kriterium modulare *Zerlegbarkeit* ist erfüllt, wenn die Entwurfsmethode bei der Zerlegung eines neuen Problems in verschiedene Teilprobleme hilft, dass deren Lösungen jeweils getrennt gesucht werden können.

- Eine Entwurfsmethode erfüllt das Kriterium der modularen *Kombinierbarkeit*, wenn sie die Herstellung von solchen Softwareelementen fördert, die miteinander frei zur Herstellung neuer Systeme kombiniert werden können, möglicherweise in einer von der anfänglichen Entwicklungsumgebung sehr verschiedenen Umgebung.

- Eine Entwurfsmethode begünstigt modulare *Verständlichkeit*, wenn sie die Herstellung von Modulen unterstützt, die für sich allein durch einen menschlichen Leser verstanden werden können. Im schlimmsten Falle sollte der Leser nur wenige Nachbarmodule mit ansehen müssen.

- Eine Entwurfsmethode erfüllt das Kriterium der modularen *Stetigkeit*, wenn eine kleine Änderung in der Problemspezifikation sich als Änderung in nur einem Modul oder wenigen Modulen auswirkt, die mit Hilfe der Methode aus der Spezifikation abgeleitet werden können. Solche Änderungen sollten nicht die Architektur des Systems berühren, also die Beziehungen zwischen den Modulen.

- Eine Entwurfsmethode erfüllt das Kriterium der *Modulgeschütztheit*, wenn sie zu Architekturen führt, in denen die Auswirkungen einer zur Laufzeit in einem Modul auftretenden Ausnahmesituation auf dieses Modul beschränkt bleiben oder sich höchstens auf wenige benachbarte Module fortpflanzt.

Nach Meyer sollten dazu noch fünf Prinzipien erfüllt sein:

- Sprachliche Moduleinheiten: die Module müssen zu syntaktischen Einheiten der benutzten Sprache passen.
- Wenige Schnittstellen: Jedes Modul sollte mit möglichst wenigen anderen kommunizieren.
- Effiziente Schnittstelle (lose Kopplung): Wenn zwei Module überhaupt miteinander kommunizieren, sollten sie so wenig Informationen wie möglich austauschen.
- Explizite Schnittstellen: Wenn zwei Module A und B kommunizieren, dann muss das aus dem Text von A oder B oder beiden hervorgehen.
- Geheimnisprinzip (Information Hiding): Jede Information über einen Modul sollte modulintern sein, wenn sie nicht ausdrücklich als öffentlich erklärt wird. Man nennt dies auch Kapselung. Die Schnittstelle sollte nur einige der Moduleigenschaften enthalten. Der Rest sollte intern bleiben.

3.1.4 Technische Randbedingungen

Bildschirm

In diesem Unterkapitel sind grafische Anforderungen beschrieben. Diese Anforderungen sind selbstverständlich nicht nur für die zweidimensionalen grafischen Darstellungen, sondern auch für 3D gültig.

Für die Darstellung der Flugführungsanzeige wird natürlich ein Anzeigemedium gebraucht. Es gibt verschiedene Arten von Anzeigemedien, die ihrerseits Anforderungen an die Software stellen. Der Entwickler der Anzeigeanwendung wird nicht unbedingt die im Flugzeug verwendete Hardware zu Verfügung haben, und wird wahrscheinlich bevorzugt mit einer höheren Auflösung arbeiten.

Als konkretes Beispiel haben die EFIS- und ECAM- Anzeigen in einem Airbus A320 ein quadratisches Format: 7,25 x 7,25 Zoll [Ind89]. Der künftige A380 wird wahrscheinlich rechteckige Anzeigen (8 x 10 Zoll) benutzen [MKG99]. Die Bildschirme des Forschungscockpits [ARE+98] des FSR haben eine Bilddiagonale von 14,1 Zoll und sind ebenfalls rechteckig.

Abb. 3.2: *Multi Function Display*

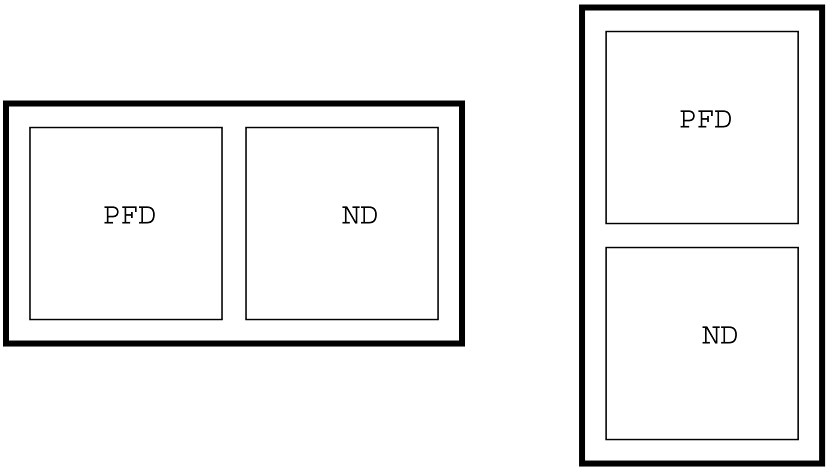
Mit einem rechteckigen Bildschirm ist es möglich, Kombinationen bestender Anzeigen oder neue Anzeigeformate zu realisieren, die mit einem quadratischen Bildschirm unmöglich wären. Das *Multi Function Display*, MFD, (siehe Abb. 3.2) im Canadair Regional Jet (CRJ) beispielsweise hat eine andere Art von Darstellung als die Airbus Displays. In [Fol98] gibt es auch ein Beispiel für Kombinationen eines HDPFD und eines HSD[1] auf dem selben gleichen Bildschirm. Die zweite Anzeige (als Ersatz des ND) besteht in dieser Studie aus einem *Multi Function Display*, das entweder Wetterradar, Verkehr oder Rollführungsdarstellung zeigt.

Diese Beispiele belegen, dass die Anwendung mit mehreren Bildschirmauflösungen und mit quadratischen oder rechteckigen Anzeigen arbeiten können muss. Aber wie die Abbildung 3.3 zeigt, gibt es zwei Möglichkeiten, eine rechteckige Anzeige zu benutzen. Die Software muss im Hochkantformat (es ist normalerweise keine Standard-Darstellungsart) die Darstellung entsprechend drehen können.

Heutige Grafikkarten verfügen auch über eine sogenannte 3D-Beschleunigung. Mit einer 3D-Beschleunigung wird ein Teil der Darstellung direkt in der Hardware der Grafikkarte berechnet. Eine 3D-Beschleunigung erleichtert also die Arbeit des Hauptprozessors und

---

[1] *Horizontal Situation Display*



Horizontalformat Hochkantformat

Abb. 3.3: Rechteckige Formate

optimiert die Berechnung von dreidimensionalen Darstellungen. Um eine solche Grafikbeschleunigung nutzen zu können, braucht eine Software Befehle, die in einer Softwarebibliothek zusammengefasst sind. Heute ist *OpenGL* [NDW93] fast ein Standard, und viele Grafikkarten können mit dieser Bibliothek arbeiten.

Für bestimmte Anzeigen sind mehrere Projektoren nötig. In der Außensicht im Simulator des FSR wird eine Fläche von 180x40 Grad dargestellt. Für eine solche Fläche werden mehrere Projektoren verwendet. In einem binokularen HMD werden zwei Projektoren, einer für jedes Auge, benutzt. Dafür müssen mehrere Projektoren von der gleichen Software verwaltet werden. Die Anwendung muss auch die verschiedenen Bilder synchronisieren.

Es gibt Rechner, die mehrere Prozessoren haben. Um die Performance zu verbessern, werden die verschiedenen Prozessoren gleichzeitig verwaltet. Das heißt, dass die Anwendung eine Simultanverarbeitung ausführen muss. Dies wird im Unterkapitel 4.7.8 weiter erläutert.

Als Zusammenfassung der Anforderung der Hardware soll die Darstellung der Flugführungsanzeige möglichst arbeiten mit:

- unterschiedlicher Orientierung des Bildschirms (die Darstellung wird um 90, 180, 270 Grad gedreht),
- unterschiedlicher Auflösung,

- unterschiedlichen Farbmöglichkeiten,
- unterschiedlichen Anzeigemedien,
- 3D-Grafikbeschleunigung,
- mehreren Projektoren,
- mehreren Prozessoren.

3.2 Anforderungen an die funktionelle Darstellung

3.2.1 Anforderungen an die zweidimensionalen Elemente

Die zweidimensionalen Elemente verfügen über eine Logik, die die Änderung der Darstellung bestimmt. Die Geschwindigkeitsskala reagiert zum Beispiel auf die Geschwindigkeit des Flugzeugs und soll ständig eine aktuelle Darstellung liefern. Außerdem hat die Geschwindigkeitsskala während des Starts und während der Landung verschiedene Darstellungen. Für den Start braucht der Pilot unter anderem die *Rotation Speed* (VR) und die *Decision Speed* ($V1$), diese Werte sind nur während des Starts notwendig (siehe Abb. 3.4).

Jedes 2D-Element soll einen Skalierungsfaktor, abhängig von der Auflösung der gesamten Anzeige, haben. Dieser Skalierungsfaktor steht für ein fertiges System normalerweise fest. Aber für die Entwicklung einer neuen Anzeige oder eines neuen Anzeigekonzeptes ist es von Vorteil, wenn dieser Skalierungsfaktor nicht konstant ist, sondern dass ein Versuchsleiter, auch *Instruktor* genannt, diesen Faktor zur Laufzeit ändern kann. Aus dem gleichen Grund sollen die Elemente auch keine feste Position auf der Anzeige haben.

Die 2D-Elemente der verschiedenen Flugzeughersteller sind nicht identisch (siehe Abb. 3.5). Das bedeutet also auch, dass es für eine Anzeige mehrere Darstellungen eines Elementes geben könnte. Für einige Symbole ist auch eine unterschiedliche Logik notwendig. Für eine Forschungsoftware, die in verschiedenen Umgebungen und Simulatoren funktionieren soll, ist es zwingend erforderlich, mehrere Versionen des gleichen Elementes benutzbar zu haben.

Während der verschiedenen Flugphasen braucht der Pilot nicht unbedingt immer alle 2D-Elemente. Beispielsweise ist es anzunehmen, dass während der Rollphase nicht die vollständigen Geschwindigkeits- und Höhe-Skalen nötig sind [MKG99], weil die Dynamik der Werte sehr klein bleibt; nur die aktuelle Geschwindigkeit bzw. die aktuelle Höhe

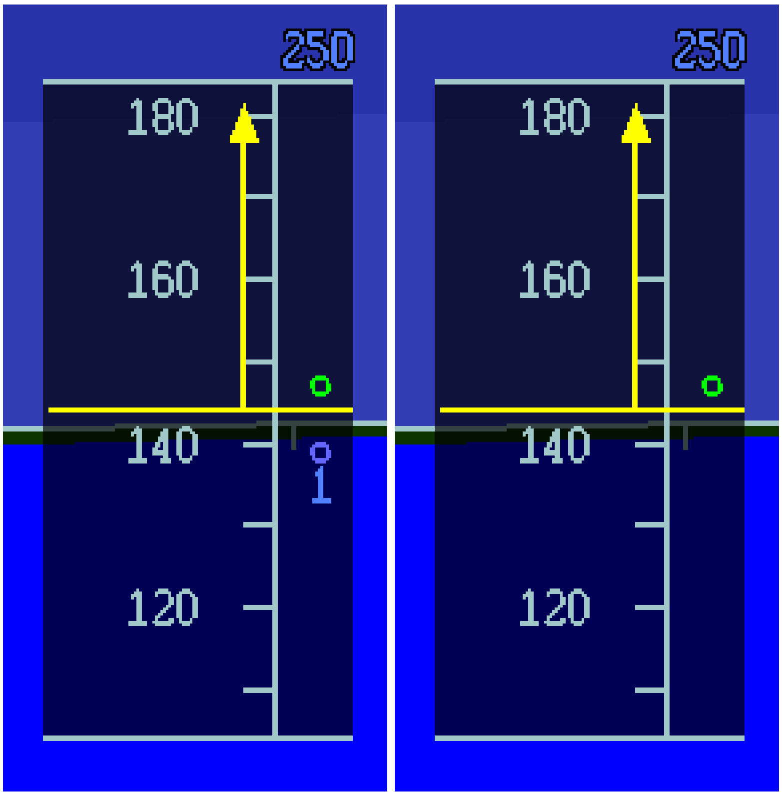


Abb. 3.4: Start-und Reisegeschwindigkeitsskala

ist nötig. Diese automatische Ausblendung unnötiger Symbole wird *Declutter-Mode* genannt. Die Symbole sollen auch manuell ausschaltbar sein. In der Testphase muss man manchmal alles bis auf ein Symbol ausschalten können, um die Logik, die Darstellung und die Performance überprüfen zu können.

Für Untersuchungen an der Mensch-Maschine-Schnittstelle sollen alle Darstellungen mit allen Anzeigen funktionieren: Eine Geschwindigkeitskala zum Beispiel muss in einem *Head Down Primary Flight Display* oder einem *Head Mounted Display* funktionieren.

Es gibt farbige und monochrome Bildschirme. Wenn die 2D-Elemente mit jeder Anzeige kompatibel sein sollen, folgt daraus, dass die Farben der Elemente nicht fest implementiert sein dürfen. Statt dessen werden die Farben der Elemente aus einer Konfigurationdatei initialisiert. Wenn der Benutzer sich andere Farben wünscht, muss er nur diese Datei ändern und braucht nicht die Software neu zu kompilieren.

Mit Hilfe einer Funktion der graphischen Bibliothek ist es möglich, die Darstellung fast aller 2D-Elemente auf einen bestimmten Raum zu begrenzen, indem nur Ausschnitte der berechneten Elemente angezeigt werden. So wird zum Beispiel die Skala in Abbildung

Abb. 3.5: Airbus PFD, Boeing PFD und CRJ MFD

3.6 oben und in der überlagerten schwarzen Fläche abgeschnitten.

Es gibt trotzdem Elemente wie den Flight-Path-Vector, die keine Limitierung haben. Sie können an jeder Stelle des Fensters dargestellt werden und werden nicht beschnitten. Für solche Symbole existiert jedoch dann eine Limitierung, wenn die externen Anwendungen, die ihre Logik beeinflussen, solche Limitierungen enthalten.

Die Tabelle 3.1 fasst die funktionalen Darstellungsanforderungen der zweidimensionalen Elemente zusammen.

3.2.2 Anforderungen an die dreidimensionalen Elemente

Es gibt bezüglich der dreidimensionalen Darstellung im wesentlichen zwei Teile: Gelände und solche Elemente, die sich relativ zum Gelände bewegen, wie z.B. Fremdflugzeuge oder der Prädiktor mit seinem Schatten (zwei gekoppelte Symbole, siehe [Pur99]).

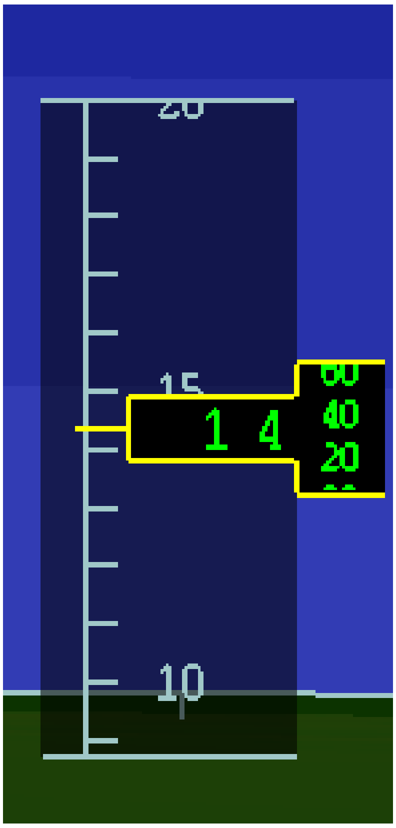


Abb. 3.6: Räumliche Limitierung einer Skala

	Vertikale Bandskalen	Horizontale Bandskalen	Rundskalen	Digital Werte	Symbole
Version	Initialisiert	Initialisiert	Initialisiert	Initialisiert	Initialisiert
Farbe	Initialisiert	Initialisiert	Initialisiert	Initialisiert	Initialisiert
Ein- ausschaltbar	Interaktiv	Interaktiv	Interaktiv	Interaktiv	Interaktiv
Declutter Mode	Automatisch/ Interaktiv	Automatisch/ Interaktiv	-	-	-
Größe	Interaktiv	Interaktiv	Interaktiv	Fest	Fest
Position in Fenster	Interaktiv	Interaktiv	Interaktiv	Interaktiv	Logik
Räumliche Limitierung	Fest	Fest	Fest	Fest	Keine/ externe Logik

Tab. 3.1: Anforderungen an die 2D-Elemente

Die Trennung in zwei Teile wurde wegen der Komplexität der Verwaltung der Geländedarstellung eingeführt. Es ist zum Beispiel nicht möglich, die Informationen für die

Darstellung der ganzen Erde auf einmal zu laden. Weiterhin werden zusätzlich zum Gelände einige Navigationsinformationen angezeigt, zu denen auch Textinformation in der orthogonalen Anzeige gehört (siehe Kapitel 2.2.2). Wie bereits erläutert wurde, soll der Text auf der Anzeige mit einer konstanten Richtung unabhängig von der Flugrichtung dargestellt werden. Um dies zu realisieren, muss man eine Logik in die Verwaltung des Geländes einfügen.

Wenn die Software gleichzeitig verschiedene Anzeigen (zum Beispiel ein PFD und ein ND) verwalten soll, wird für alle Anzeigen gemeinsam nur einmal die Geländeinformation aus der Datenbank geladen, um Speicherplatz zu sparen. Um außerdem Rechenzeit zu sparen, wird die Datenbank nur einmal verwaltet.

Die Komplexität der 3D-Symbole ist kleiner. Es ist möglich, einmalig ein Modell zu definieren, zu speichern und nur wenn es nötig ist, diese Symbole darzustellen. Die Komplexität dieser Symbole besteht in der Verwaltung der Bewegung und der Farbkodierung dieser Symbole (ein Fremdflugzeug wird anders gefärbt, wenn sich die Gefahrenstufe ändert). Die Berechnung dieser Verwaltungen sollte nicht unbedingt in dieser Flugführunganzeigesoftware durchgeführt werden. Heutzutage wird z.B. ein getrenntes Gerät die Informationen zu diesen Fremdflugzeugen (Position und Farbkodierung) berechnen. Dieses Gerät schickt zur Anzeige nur die Informationen über die Fremdflugzeuge, die die Anzeige darstellen muss. Es wäre möglich, dies auch für den Prädiktor und andere 3D-Elemente zu realisieren. Für die meisten 3D-Elemente trifft dies jedoch nicht zu und die Flugführungsanzeige muss einen grossen Teil der Informationen selbst berechnen. Externe Berechnungen würden die Zeitperformance der Flugführungsanzeige-Software verbessern, aber dies würde zu einer hohen Belastung des Netzwerks des FSR führen und damit die gesamte Zeitperformance der Flugsimulation verschlechtern.

Wie erläutert wurde, spielt die Lichtquelle eine Rolle für die Plastizität der Darstellung [Kau98]. Um Versuche zu ermöglichen, muss die Lichtquelle eine interaktiv veränderliche Position, Farbe und Intensität besitzen.

Die Tabelle 3.2 fasst die Anforderungen bezüglich der dreidimensionalen Darstellung der Flugführungsanzeige zusammen.

3.3 Anforderungen an die Kommunikationsschnittstelle

Die Flugführungsanzeigesoftware braucht Information, die über die Kommunikationsschnittstelle ausgetauscht wird. Tanenbaum beschreibt in [Tan95], wie ein offenes System mit einem anderen offenen System kommunizieren kann, indem es Standardregeln

		Perspektivische Anzeigen	Orthogonale Anzeigen
3D-Symbole		ein- ausschaltbar	ein- ausschaltbar
	Prädiktor	Flugphase	Rollphase
	Schatten des Prädiktors	Flugphase	-
	Fremdenflugzeuge	X	X
Lichtquelle			
	Position	interaktiv	interaktiv
	Farbe	interaktiv	interaktiv
	Intensität	interaktiv	interaktiv
Referenz		interaktiv	interaktiv
	Voreingestellte Blickrichtung	horizontal	senkrecht n. unten
	Voreingestellte Position	Flugzeugreferenz	Über dem Flugzeug
Gelände		ein- ausschaltbar	ein- ausschaltbar
	Bewegung	interaktiv	interaktiv
	Position	flugzeugabhängig	flugzeug- und modusabhängig
	Weltweite Anwendbarkeit	X	X
	Verschiedene DB Inhalte	X	X
	Symbol Darstellung	anzeigespezifisch	anzeigespezifisch
	Objekte mit fester Orientierung	optional	X
	Objekte mit fester Größe	optional	X
	Geländefarbkodierung	X	X

Tab. 3.2: Anforderungen an die 3D-Elemente

befolgt, die das Format, den Inhalt und die Bedeutung gesendeter und empfangener Nachrichten festlegen. Einige Regeln werden beim Senden vom Werten verwendet. Die Benutzer dieser Werte lesen die Information nach den selben Regeln. Diese Regeln beschreiben

- wie das Netzwerk organisiert ist,
- ob der Sender die Werte nur an einen bestimmten, mehrere oder alle Rechner gleichzeitig schickt,
- welches Format (Ganzzahlvariablen oder Fließkommazahl) oder welche Größe (einfache Präzision oder doppelte Präzision) die Werte haben,
- ob alle Werte voneinander unabhängig sind oder ob sie in mehreren Strukturen zusammengefasst sind.

Das Flugführungsanzeige-Programm soll in verschiedene Simulationsumgebungen integriert werden können.

Um dieser Anforderung zu genügen, muss man erstens die verschiedenen Aspekte einer Kommunikationsschnittstelle betrachten. Dann werden die Anforderungen der Software bezüglich der Kommunikation analysiert.

3.3.1 Medium

Die Analyse der Rechnerarchitektur in einem Netzwerk oder die Netzwerktopologie ist ein komplexes Thema, das mehr oder weniger die Hardwareorganisation betrifft. Dieses Kapitel wird sich auf die Softwareseite und das Protokoll begrenzen.

Netzwerk-Architektur

Um Daten zwischen Rechnern auszutauschen, braucht man ein Medium zwischen den Rechnern. Um sicherzustellen, dass die Daten richtig ankommen, braucht man zusätzlich Softwarearchitekturen.

Heutzutage existiert eine Norm, die OSI[2] und eine Realisierung, TCP/IP-UDP/IP[3], die sich an der OSI orientiert, die Referenz für die Netzwerk Architektur [Puj95].

Die OSI Architektur besteht aus sieben Schichten (siehe Abb.3.7).

Bevor die sieben Schichten weiter erläutert werden, ist noch eine Vorbemerkung nötig. Eine Datei wird selten als einzelnes, zusammenhängendes Datenpaket im Netzwerk verschickt. Es gibt mehrere Gründe dafür. Erstens, wenn eine Datei sehr groß ist, könnte niemand sonst das Netzwerk benutzen, solange diese Datei nicht ganz übertragen ist. Zweitens müsste, wenn ein Fehler auftritt, die Datei komplett neu geschickt werden. Um dies zu vermeiden, werden die Dateien in *Sockets* oder kleine Teile aufgeteilt. Für jeden Teil muss der Absender überprüfen, ob er etwas schicken darf. Wenn ein Teil einen Fehler hat, muss der Absender nur diesen *Socket* neu senden. Sind die Teile klein genug, kann der Empfänger zuweilen allein mit Hilfe der Prüfsumme einen Fehler korrigieren, er braucht dann keine neue Sendung. Diese *Sockets* werden durch die verschiedenen Schichten geändert: Jede Schicht des Absenders fügt ihre eigene Information ein. Jede Schicht eines der Zwischenempfänger und des Endempfängers wird die Information der Schicht vom *Socket* lesen, interpretieren und löschen.

---

[2] Open System Interconnection

[3] TCP: Transport Controp Protocol.
IP: Internet Protocol
UDP: User Data Protocol

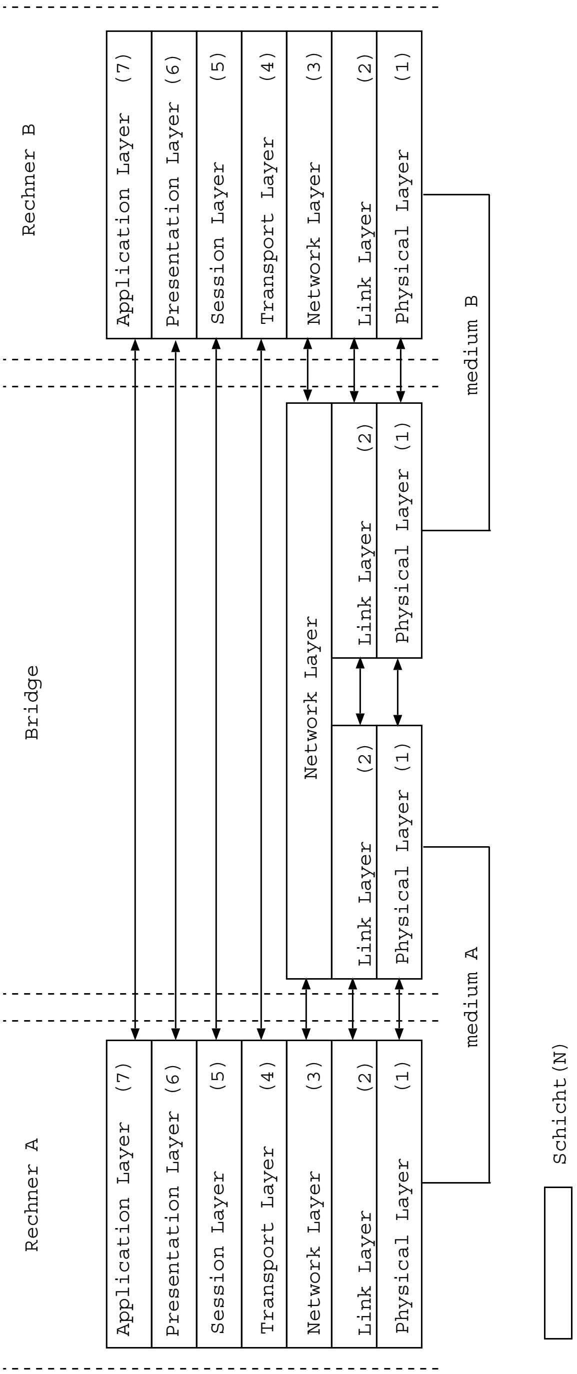


Abb. 3.7: OSI Schicht

Jede Schicht (N) muss Dienste anbieten. Um einen Dienst ausführen zu können, sind Protokolle definiert. Ein Protokoll beschreibt Regeln, um Informationen auszutauschen. Diese Informationen werden an die gleiche Schicht (N) (siehe Abb. 3.7), auf einem anderen Gerät geschickt. Ein Dienst der Schicht (N) kann auch die Information vorbereiten, um sie an die nächste Schicht (N+1) zu versenden.

Die physikalische Schicht (1) stellt die im allgemeinen elektrischen funktionellen und prozeduralen Mittel zur Verfügung, um eine physikalische Verbindung herzustellen oder zu lösen. Diese Verbindung erlaubt binären Austausch mit der Sicherungsschicht [Puj95].

Die Sicherungsschicht (2), auch *Data-Link*-Schicht genannt, beinhaltet die funktionellen und prozeduralen Mittel, um logische Verbindungen zwischen Netzwerkeinheiten zu generieren, zu erhalten und freizulassen. Diese Schicht kann Fehler der physikalischen Schicht korrigieren: Ein elektrisches Signal kann etwa 200m pro Mikrosekunde überbrücken [Puj95]. Wenn ein Benutzer nur 20 Mikrosekunden braucht, um eine Nachricht zu schicken, würde das bedeuten, dass entweder nur er allein etwas sendet, oder dass das Netzwerk kleiner als 4km (200*20) sein muss. Ansonsten gäbe es unlösbare Konflikte: Ein Rechner sendet und wird nicht unbedingt bemerken, dass ein anderer im Netzwerk gleichzeitig sendet. Die zweite Schicht hat die Aufgabe, solche Probleme zu lösen.

Die Netzwerkschicht (3) ermöglicht es, Informationen bis zum Empfänger zu übertragen. Bis zum Empfänger werden die *Sockets* durch einige Zwischenknoten geschickt. Die dritte Schicht soll dafür sorgen, dass es keinen *Stau* gibt. Sie muss abhängig von der Belastung der Zwischenknoten einen Transportweg wählen. Dieser Weg wird dynamisch berechnet und ist nicht unbedingt der gleiche für alle *Sockets* einer Nachricht. Diese Schicht berücktsichtigt ebenfalls die Adresse des Absenders und des Empfängers.

Pakete können auf dem Weg vom Sender zum Empfänger verlorengehen. Obwohl einige Anwendungen ihre eigene Fehlerbehandlung mitbringen, ziehen andere eine zuverlässige Verbindung vor. Es ist die Aufgabe der Transportschicht (4), diesen Service bereitzustellen. Der Gedanke dabei ist, dass die Kommunikationssteuerungsschicht in der Lage sein sollte, eine Nachricht an die Transportschicht zu übergeben, und dann erwarten kann, dass die Nachricht ohne Verlust übertragen wird [Tan95].

Die Sitzungssicht (5), auch *Session*-Schicht genannt, überprüft, ob die Kommunikation möglich ist. Es ist sinnlos etwas zu schicken, wenn kein Empfänger da ist. Sie öffnet und schließt eine Verbindung zwischen den Rechnern. Ihr Zweck ist die Synchronisation des Datenaustauschs zwischen den Anwenderinstanzen sowie die Resynchronisation bei einem Wiederaufsetzen der Verbindung nach Fehlern in der Datenübertragung [Kub96].

Die Präsentationschicht (6) muss sich um die Syntax kümmern. Ihr Zweck ist die Kodierung und Darstellung der auszutauschenden Information (ASCII, EBCDIC, XDR) [Kub96]. Der Empfänger muss natürlich die Befehle oder die Informationen des Absenders verstehen. Daten können in verschiedenen Systemen unterschiedlich kodiert werden.

Damit sie dennoch austauschbar sind, ist es erforderlich, sie in einem bestimmten Format darzustellen, das von beiden Systemen verstanden wird. Ein gutes Beispiel sind zwei Mikroprozessoren, die einen Wert auf verschiedene Weise interpretieren: Einer liest von links nach rechts und der andere von rechts nach links. Ohne Übersetzung können die beiden Mikroprozessoren überhaupt keine Werte austauschen. Vor der eigentlichen Datenübertragung muss die Kodierung der Daten ausgehandelt werden. Die festgelegte Datenkodierung wird Transfersyntax genannt. In der Zukunft wird vermutlich diese Schicht nicht mehr notwendig sein, weil die Rechner immer mehr standardisiert werden.

Die siebte Schicht, die Applikationschicht, gibt jeder Software die Möglichkeit, die OSI-Umwelt zu erreichen. Hier werden anwendungsunterstützende Funktionen in komfortabler Form zur Verfügung gestellt. Dies können Dateitransfer, elektronische Post, Virtual-Terminal-Services, usw. sein. Man kann hier zwischen anwendungsspezifischen und allgemein verwendbaren Diensten unterscheiden.

Mit der Abbildung 3.8 wird eine ähnliche Architektur, die TCP/IP-Architektur erläutert. Im heute weit verbreiteten Schichtenmodell der Internetprotokolle existieren vier Kommunikationsschichten, die die Aufgaben mehrerer Schichten im OSI-Modell übernehmen [Eng01].

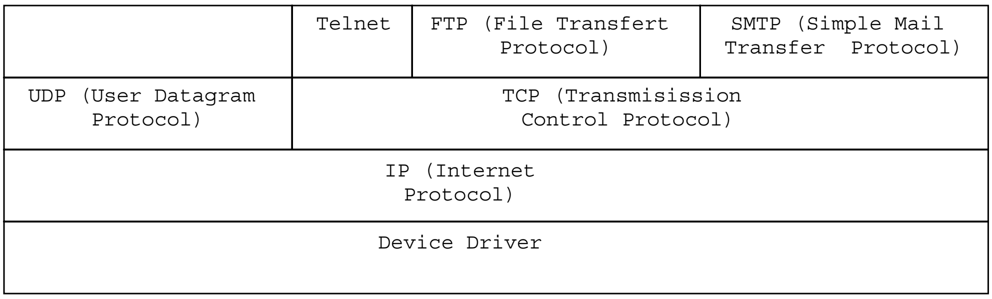


Abb. 3.8: TCP/IP und UDP/IP Architektur

Das IP (Internet Protocol) ist ein Protokoll auf Netzwerkebene. Das TCP (Transmission Control Protocol) ist ein Protokoll auf Transportebene. Über diesen zwei Protokollen sind andere Protokolle definiert:

- das FTP (File Transfer Protocol) erlaubt, Dateien auszutauschen,
- das SMTP (Simple Mail Transfer Protocol) ermöglicht die elektronische Post,
- das Telnet-Protokoll ist ein Bildschirm-Protokoll.

TCP ist ein Protokoll, das eine feste Verbindung braucht. Jeder *Socket* ist eine Sammlung einer konstanten Anzahl von *Bytes*. Eine Datei besteht aus Bytes. Jedes Byte wird in einer bestimmten Ordnung geschickt und in der gleichen Ordnung empfangen. Dieses Protokoll benutzt ein *Port-Konzept*. Für TCP hat ein *Port* nur eine Beziehung zu einer einzigen Software bzw. zu einem einzigen Softwarepaket, welches gemeinsam einen Datenfluss benutzt (siehe Abb. 3.9).

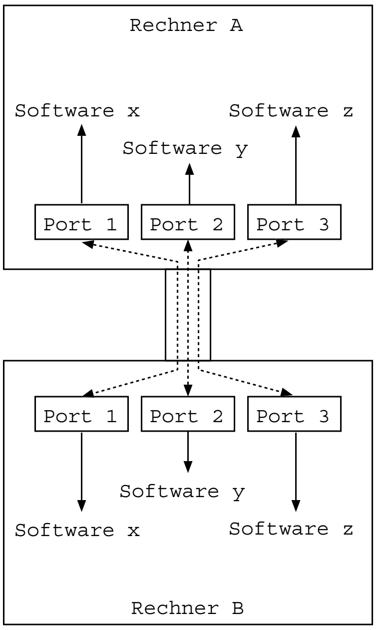


Abb. 3.9: Port

Das TCP Protokoll ist ein gesichertes Protokoll. Es wurde vom *Department Of Defense* (DOD) der Vereinigten Staaten entwickelt. Bevor zwei Rechner miteinander Werte austauschen, wird eine Verbindung fest definiert. Dabei werden zum Beispiel die Geschwindigkeit des Austausches und der Weg definiert. Der Absender und der Empfänger überprüfen regelmäßig durch eine Bestätigung des Empfängers, dass die Bytes richtig ausgetauscht worden sind. Diese Sicherheit bedeutet leider manchmal auch Unflexibilität und kann die Performance verschlechtern. Wenn z.B. ein Wert an mehrere Rechner geschickt werden soll, muss man eine Verbindung mit jedem Rechner generieren und den Wert mehrmals schicken. Für ein Flugsimulationssystem ist das zu begrenzend: zu viele Werte werden geschickt und viele Werte werden in verschiedenen Softwarepaketen, die auf verschiedenen Rechnern laufen, benötigt. Dafür wurde in den USA ein weiteres Protokoll entwickelt: das UDP -User Data Protocol- (siehe Abb. 3.8).

Das UDP erlaubt es, Werte ohne feste Verbindung auszutauschen: Die Werte werden einfach in das Netzwerk geschickt, sie sind nicht an einen bestimmten Rechner adressiert. Die Anwendungen müssen selbst entscheiden, welche Werte aus dem Datenstrom sie benötigen. Der Absender erwartet keine Rückmeldung der verschiedenen Empfänger. Daher enthält das UDP keine Möglichkeit, Fehler zu korrigieren oder den Datenfluss zu überprüfen (die *Sockets* können zum Beispiel zu schnell für den Empfänger gesendet worden sein). Dieses Protokoll benutzt ebenfalls ein *Port-Konzept*. Jede Sendung enthält eine *Source-Port*-Identifikation der Absender-Anwendung und die *Destination-Port*-Identifikation der Empfänger-Anwendung.

Datenformat

Damit die verschiedenen Programme Werte austauschen können, muss man Formate definieren. Nicht nur die Größe der Werte (einfache oder doppelte Präzision für Dezimalwerte) spielt eine Rolle sondern auch, ob Werte in Strukturen eingebunden sind.

Die Position des Flugzeugs z.B. wird durch drei Werte definiert (x,y,z), deshalb werden diese Werte in einer Struktur versammelt. Um aber den Zustand des Flugzeugs zu beschreiben, sind weitere Werte wie der Geschwindigkeitsvektor, der Beschleunigungsvektor, die Winkel usw., nötig. Alle diese Werte stehen in Bezug zueinander, deshalb werden auch sie in dieser Struktur zusammengefasst. Weiterhin sind untereinander konsistentz Daten nötig. Dies wird sichergestellt durch den Austausch von ganzen Strukturen anstelle von einzelnen Werten.

Die Bildung von Strukturen erleichtert außerdem die Administrationen der Vielzahl an benötigten Daten.

3.3.2 Anforderungen

Die Software soll die Möglichkeit haben, verschiedene Schnittstellen zu benutzen. Jedoch ist es unmöglich, alle Kommunikationsarten zu implementieren. Die vorherige Beschreibung der verschiedenen Netzwerk-Typen zeigt deutlich, dass es zu viele Möglichkeiten gibt, um alles sofort zu implementieren. Es ist vorteilhafter, einige Grundprinzipien zu realisieren.

Bei einigen Schnittstellen kann man erwarten, dass die verschiedenen Schichten des Netzwerks schon implementiert sind. Zum Beispiel ist es nicht nötig, die Schichten des UDP/IP Protokolls zu schreiben, weil es schon fertige Funktionen gibt. In solchen Fällen müssen die Funktionen dieser Schnittstelle nur diese *UDP/IP-Funktion* mit den richtigen Parametern aufrufen. Die Realisierung muss auch erlauben, dass ein Teil einer Schicht für bestimmte Schnittstellen nachprogrammiert wird.

Die Kommunikationsschnittstelle muss, wenn nötig, die Hardware initialisieren, das heißt also, dass diese Schnittstelle nicht nur mit den letzten, sondern auch mit den ersten OSI-Schichten (siehe Abb. 3.7) arbeiten können muss.

In bestimmten Fällen, wie in der Beispielanwendung ATTAS-Flugkampagne 1997 [HKLP00], ist es nicht möglich, nur die eigenen Kommunikationsschnittstellen zu verwenden. Das Konzept der Kommunikationsschnittstelle soll Brücken erleichtern, damit die Werte auf zwei verschiedene Arten von Netzwerken verteilt werden können.

Eine Software liest Werte von einem Netzwerk und schreibt die Werte in ein anderes Netzwerk. Der Rechner B der Abbildung 3.10 wird als Brücke benutzt: Er liest die Flugzeugwerte vom Rechner A und muss die Werte mit der internen Kommunikationsschnittstelle des FSR auf den Ethernet-Bus schicken.

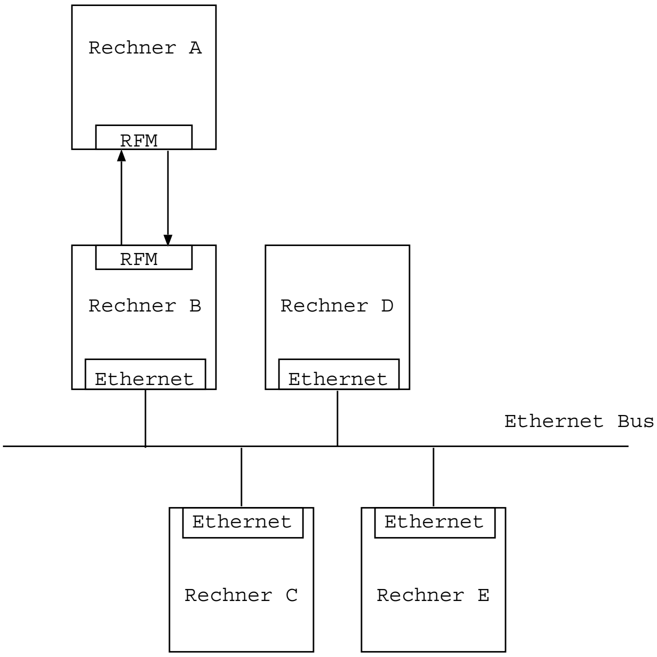


Abb. 3.10: ATTAS-Architektur

Die Flexibilität der Kommunikationsschnittstelle darf sich nicht nur auf das Protokoll und auf die Architektur des Netzwerks beschränken. Es ist auch möglich, dass beteiligte Partner eine unterschiedliche Vorstellung von den Formaten der Werte oder den Strukturen haben. Die ISAWARE-Schnittstelle [GMR00] ist hier ein gutes Beispiel: Es wird eine sehr große Struktur benutzt. Diese Struktur fasst Inhalte mehrerer Strukturen zusammen. Aber in dieser großen Struktur sind die Werte in einer anderen Reihenfolge geordnet als in der Standardkommunikation des FSR. Es gibt verschiedene Möglichkteiten, die Position und Lage des Flugzeugs zu beschreiben:

- erstens die Koordinaten, zweitens den Geschwindigkeitsvektor und drittens die Winkel (Standardanordnung am FSR)
- oder die Koordinaten nach dem Geschwindigkeitsvektor und dann die Winkel (projektspezifische Ordnung).

Es wurde gezeigt, dass eine Flexibilität der Struktur nötig ist. Das Format oder die Dimensionen der Werte könnten differieren. Für das Beispiel der Position des Flugzeugs kann man die Werte als Bogenwinkel in Sekunden, Grad oder Radiant speichern. Es zeigt, dass Format- bzw. Dimensionsübersetzer nötig sind, um Modularstetigkeit erreichen.

Die Schnittstelle ist für jedes Projekt unterschiedlich. Wie vorher beschrieben, soll diese Schnittstelle transparent sein. Die Anzeigesoftware muss mit jeder externen Schnittstelle kompatibel sein. Somit wird das Kriterium der Modular-Verständlichkeit und Stetigkeit erfüllt (siehe [Mey90]).

Es sollte vorgesehen werden, Tools für die Änderung der Schnittstelle zu benutzen.

Die Schnittstelle muss eine gewisse Fehlertoleranz aufweisen. Jeder Softwareentwickler, der eine Änderung für sein Projekt macht, wird die Schnittstelle adaptieren. Eine Änderung für ein bestimmtes Schnittstellenprotokoll könnte Seiteneffekte auf die anderen zur Folge haben. Für eine neue Schnittstelle könnte ein weiterer Parameter benötigt werden. Dazu wird die existierende Struktur modifiziert, was ebenfalls die Modifikation der Schnittstellengegenseite erfordert. Treten dabei Fehler auf, sollte der Schnittstellenteil der Software auf eine definierte Weise reagieren und einen unkontrollierten Abbruch des Programms vermeiden.

Die folgende Liste fasst die Anforderungen an die Kommunikationsschnittstelle zusammen:

- Jedes Projekt kann mit einer anderen Schnittstelle arbeiten
- Erweiterbarkeit aller OSI-Schichten
- Konfigurierbarkeit der Netzwerkhardware
- Einfache Realisierung einer Netzwerkbrücke
- Keine feste Strukturierung der Werte

- Die Kommunikationsschnittstelle soll Format- (Ganzzahl, Fließkommazahl, einfache doppelte Präzision...) und Dimensionübersetzer (z.B. Winkel in Grad, Radiant, Bogen) integrieren können.

- Gleiche Basis für jede Schnittstelle

- Die Kommunikationsschnittstelle muss eine gewisse Fehlertoleranz aufweisen.

- *Black Box* für den Rest der Software:
  - Die gleiche Methode liefert die Werte für alle Schnittstellen
  - Die Generierung bleibt versteckt
  - Es gibt höchstens eine Instanz einer solchen *Black Box*, auch wenn aus Sicht der Software mehrere nötig scheinen.

- Konsistenz: Alle generierten *Black Boxes* werden zusammen, aber genau einmal für jede Bilderstellung aktualisiert, es darf nicht möglich sein, dass man nur eine einzelne Struktur aktualisiert.

- Keine Initialisierung der Werte im Quellcode, sondern nur mit Textdateien

4 Systementwicklung

Die vorherigen Kapitel haben die Ziele dieser Arbeit und die Anforderungen, die an die Entwicklung der Anwendung gestellt werden, beschrieben. Der folgende Abschnitt wird einige Definitionen, die in der Arbeit weiter verwendet werden, erläutern. Danach wird kurz die Entwicklungsumgebung der Anwendung beschrieben. Es ist nicht möglich, alle computergrafischen Funktionalitäten zu erläutern. Stattdessen werden nur diejenigen erklärt, welche für die Software eine besonderen Rolle spielen. Außerdem werden einige in dieser Arbeit getroffenen Entscheidungen erläutert: Es gibt zum Beispiel verschiedene Farbverläufe; die bekanntesten werden kurz beschrieben, der schließlich Verwendete wird am Ende des Unterkapitels angegeben.

Nach dem Abschnitt *Computergrafische Funktionalität* 4.4 wird ein kurzer Überblick über die Software gegeben. Dann wird die Realisierung der 2D-Darstellung, der 3D-Darstellung und zum Schluss die Kommunikationsschnitttstelle erläutert.

4.1 Definitionen

Performance
Wie schon mehrfach erwähnt wurde, spielt die Zeit eine besondere Rolle, weil diese Software in einem Mensch-Maschinen-System im Flugzeug zur Anwendung kommt, und weil der Pilot sofort auf die Informationen, die auf dieser Anzeige dargestellt sind, reagieren muss, um den Zustand des Flugzeugs zu beeinflussen. Die Zeitperformance ist also wichtig, und dies wird einige Entscheidungen in der Gestaltung begründen. Später wird man von Zeitperformance oder auch abgekürzt von Performace sprechen.

Anzeige
Eine Anzeige oder ein Display wird in dieser Arbeit die Einheit einer Darstellung sein. Eine Anzeige kann also ein HDPFD, oder ein HUD, usw. sein. Normalerweise hat eine Anzeige ein Fenster. Es gibt auch Anzeigen, die mehrere Fenster haben, wobei diese Fenster dann gekoppelt sind. Eine Außensicht-Anzeige mit mehreren Projektoren (daher mehreren Fenstern) ist ein Beispiel. Auch eine stereoskopische Anzeige könnte zwei gekoppelte Fenster benutzen. Die beiden Fenster sind gekoppelt derart,

- dass eine Änderung innerhalb eines Fensters auch zu einer Änderung innerhalb des anderen Fensters führt: Wenn die Position des Referenzpunkts einer Außensicht-Anzeige sich ändert, wird diese Änderung jedes Fenster betreffen.

- oder, dass der Sinn der Darstellung sich mitändert: Zum Beispiel werden beide Fenster der stereoskopischen Darstellung geändert, wenn die Parallaxe geändert wird.

Ein *Multi Function Display* besteht eigentlich aus mehreren Anzeigen. Es könnten ein HDPFD, ein ND und ein VSD[1] auf einem Bildschirm dargestellt werden. Im Lauf dieser Arbeit werden diese als mehrere getrennte Anzeigen behandelt, die auf dem gleichen Bildschirm dargestellt werden und die manchmal von der selben Software berechnet werden. Es sind aber mehrere unabhängige Einheiten.

Es gibt Anzeigen, die nur zweidimensionale Elemente darstellen sollen: Zum Beispiel ein HUD, ein ECAM oder ein FMS. Diese Displays bestehen zur Zeit nur aus Text oder einfachen Bildern. Ein Display kann auch nur dreidimensionale Elemente zeigen, z.B. eine Außensichtanzeige für eine Simulation.

Für diese Anwendung stellen die Displays normalerweise gleichzeitig zweidimensionale und dreidimensionale Elemente dar.

2D-Elemente

Unter dem Begriff zweidimensionaler Elemente, *2D-Elemente*, versteht man grafische Symbole, die keine räumliche Tiefe haben. Außer in einer stereoskopischen Darstellung befinden sich alle diese Elemente in der gleichen Ebene. Diese Symbole sind haupsätchlich Elemente, die man auf gegenwärtigen Flugführungsanzeigen findet.

In dieser Arbeit werden als Beispiele die Darstellung der Geschwindigkeitsskala oder der Höhenskala gebraucht.

Symbol

Ein 2D-Element wird oft aus verschiedenen Symbolen gebildet. Die Geschwindigkeitsskala ist eine Skala, die zusätzliche Informationen, *Symbole*, darstellt: Die minimale Geschwindigkeit, die maximale Geschwindigkeit, die Geschwindigkeit bei der die Klappen gesetzt werden (*Flap speed*) usw. Diese Symbole haben ihre eigene Dynamik und Logik: die höchste Geschwindigkeit ist nicht die gleiche während des Startes und während des Fluges, sie hängt vom Flugzustand und der Konfiguration (z.B. Klappen ausgefahren oder nicht) ab. Die Skala selbst wird in dieser Arbeit nicht als getrenntes Symbol erwähnt, sie ändert sich nur mit der Geschwindigkeit und wird durch sonst nichts beeinflusst.

---

[1] Vertical Situation Display

3D-Elemente

3D-Elemente sind im Gegentsatz zu 2D-Elementen grafische Symbole, die eine räumliche Tiefe visualisieren.

Die Größe solcher Elemente in einer perspektivischen Darstellung muss sich mit der Entfernung zum Betrachter ändern.

Es gibt verschiedene 3D-Elemente. Die folgene Liste enthält die gegenwärtig vorhandenen:

- Das Gelände,
- Die *Intruder*: Andere Flugzeuge im Luftraum,
- Das *Follow Me Aircraft*: Ein Flugführungs-Symbol. Der Pilot soll diesem wie in einer Flugformation folgen.
- Der *3D Flight Path Vector*: Dies sind einige Flugführungs-Symbole, die in ihren Funktionen dem *Follow Me Aircraft* ähnlich sind.
- Der *Prädiktor*: Ein 3D-Element, entwickelt von Purpus (siehe [Pur99]), zeigt die vorausberechnete, zukünftige Position des Flugzeugs.
- Die *Route*: Die Flugkanäle zeigen den Sollweg des geplanten Fluges.

Entwickler werden sehr wahrscheinlich noch weitere, eigene 3D-Elemente erarbeiten.

PfChannel, pfPipe, pfPipeWindow

An dieser Stelle sollen einige Fachausdrücke, die bei der Erzeugung dreidimensionaler digitaler Bilder von Bedeutung sind, erläutert werden. Die folgende Erklärung stammt aus [Eck97]. Eine Szene wird durch einen Software-*Channel* (später auch als *pfChannel* bezeichnet) abgebildet. Die Ansicht der Szene wird mit einer Pipeline (*pfPipe*) erzeugt und in einem Bildpuffer gespeichert. Der Bildpuffer wird auf dem Bildschirm in einem Fenster (*pfPipeWindow*) oder auf einem begrenzten Teil des Bildschirms abgebildet (siehe Abb. 4.1).

Abhängig von der Position der virtuellen Kamera und ihrer Bildrichtung wird die Szene anders berechnet. *pfChannel* stellt dabei die Kamera dar.

Die Pipeline ist die Verbindung zwischen der Software und der Hardware (dem Bildschirm). Es gibt normalerweise soviele Pipelines wie Bildschirme (oder Projektoren). Eine Pipeline ist mindestens notwendig.

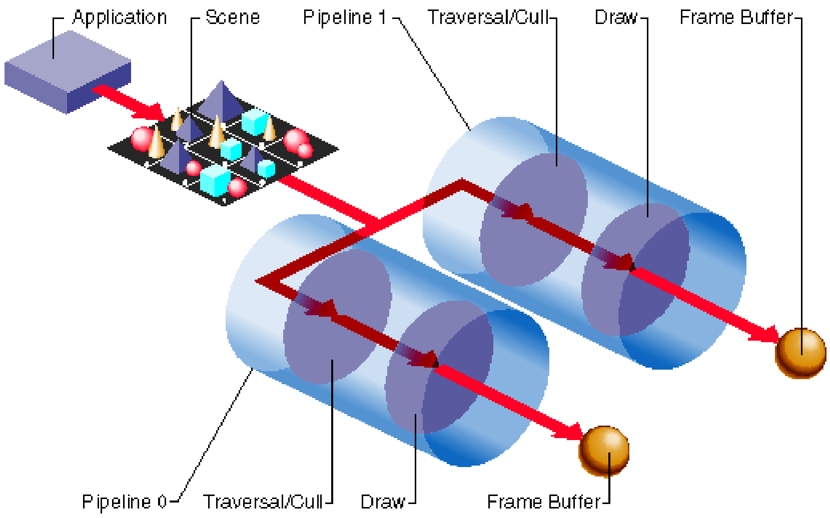


Abb. 4.1: Szene, Channel Pipeline, Fenster und Frame buffer

4.2 Hardware, grafische Bibliothek und Betriebsystem

Vorrangige Zielhardware sind Silicon Graphics Rechner. Dies hat primär historische Gründe. SGI hat eine Vorreiterrolle bei der Entwicklung von Hardware und Software für Grafikdarstellungen eingenommen. Ein Beispiel dafür ist die inzwischen weit verbreitete grafische Bibliothek *OpenGL* (Open Graphic Library). In dieser Arbeit wird *OpenGL* ebenso wie die Bibliotheken *Open Inventor* und *OpenGL Performer* von SGI verwendet.

Personal Computer, PC, werden zur Zeit immer leistungsfähiger. Es gibt für PC sehr schnelle Prozessoren mit Hochleistungs-Grafikkarten. Es ist also sinnvoll, diese Anwendung nicht nur für die SGI Rechner, sondern auch für PCs zu entwickeln. Wie später eingehend begründet wird, verbessern Mehrprozessor-Rechner die Performance der Software, besonders für die Flugführungsanzeigesoftware.

Die SGI Rechner laufen mit UNIX (IRIX) als Betriebsystem. Seit kurzer Zeit können SGI-Rechner auch mit Linux[2] arbeiten. Die genannten Bibliotheken wurden für PC unter Linux portiert. Die im Rahmen dieser Arbeit konzipierte Anwendung wurde für die beiden Betriebsysteme entwickelt.

UNIX und Linux erlauben die Verwendung von Skripten. Ein Skript ist, vereinfacht dargestellt, eine Liste von Betriebssystemkommandos in einer Datei. Die Kommandos werden vom Kommando-Interpreter analysiert, und dann werden die Programme mit den entspre-

---

[2] Linux ist ein Unix-ähnliches Betriebssystem. Es gibt dazu Versionen von Linux für PC.

chenden Parametern zur eigentlichen Ausführung der Kommandos aufgerufen [SB01]. Ein Skript sieht fast wie eine einfache Software aus, die nicht kompiliert werden muss, wobei der Umfang von Skripten eingeschränkt ist. Die Möglichkeit zur Automatisierung von Vorgängen bei der Erstellung von Software wird in dieser Arbeit genutzt.

Zusammengefasst wird in dieser Arbeit in der folgenden Umgebung operiert:

- Hardware:
  - verschiedene Silicon Graphics Rechner mit unterschiedlichen Prozessoren und Prozessorenzahl, einer oder mehreren *Grafikpipes*
  - PC.
- Betriebsystem:
  - IRIX 6.5 für SGI
  - Linux SuSE 7.3 für PC.
- Grafikbibliothek:
  - *OpenGL* als Basis der grafischen Bibliothek
  - *OpenGL Performer* als Hauptbibliothek
  - *Open Inventor* für die Datenbank .

4.3 Objektorientiertes Design und Programmierung

Objektorientierte Entwicklungen haben mehrere Vorteile. Wie in der realen Welt bestehen Objekte zumeist aus verschiedenen anderen Objekten: Ein Haus besteht aus Mauern, Dach, Fenster und Türen. Abhängig vom Betrachter kann man jedes von diesen Objekten noch als Gruppierung weiterer Objekte sehen. Eine objektorientierte Entwicklung ist auf dem gleichen Prinzip aufgebaut. Dies spricht für eine *zerlegbare Modularität* (siehe Kapitel 3.1.3). Teilweise bilden die Objekte in der Software physikalische Objekte aus der realen Umgebung ab, dies spricht für eine *verständliche Modularität*. Ein objektorientierter Entwurf respektiert normalerweise auch die anderen Kriterien von Meyers (siehe 3.1.3). Diese Arbeit wird einen objektorientierten Ansatz benutzen.

UML, *Unified Modelling Language*, ist eine Sprache und Notation, um objektorientierte Systeme zu beschreiben. So legt die UML zum Beispiel Symbole für Klassen, Objekte und deren Beziehungen untereinander fest [HKK01]. Solche Modelle sind eigentlich als

Schnittstelle einer Anwendung zwischen Software-Entwicklern und anderen Entwicklern gedacht. Diese Entwickler kennen nicht notwendigerweise die Programmierung, aber sie müssen mit der Anwendung arbeiten. Die Unified Modeling Language definiert Regeln, damit verschiedene Personen über die gleiche Anwendung einfach kommunizieren können. Man muss nur einige einfache Regeln lernen, um UML Diagramme zu verstehen. Solche Diagramme werden in dieser Arbeit benutzt, um den Aufbau der Software zu verdeutlichen.

Objektorientierte Softwareentwicklung gibt es bereits seit 30 Jahren. Am Anfang stand die Programmiersprache Smalltalk, die das Klassenkonzept von der Programmiersprache Simula-67 übernahm und weiterentwickelte. Mit Beginn der 90er Jahre hat sich C++ als dominierende objektorientierte Sprache durchgesetzt. Seit 1996 gewinnt neben C++ noch Java an Bedeutung, während Smalltalk mehr und mehr zurückgedrängt wird [HKK01]. C++ ist eine objektorientierte Erweiterung von C, einer prozeduralen Sprache, die sehr nahe an Maschinensprache oder Maschinenlogik orientiert ist. Daher ist diese Sprache für sehr viele Prozessoren benutzbar. Als Erweiterung von C gibt es für sehr viele Prozessoren einen C++ Compiler.

Diese Arbeit verwendet C++ als objektorientierte Sprache. Im Lauf der Arbeit werden die verschiedenen Besonderheiten des objektorientierten Entwurfs und der objektorientierten Programmierung (OOP) erklärt.

4.4 Computergrafische Funktionalitäten

In diesem Kapitel werden die wichtigsten Funktionalitäten der Computergrafik erläutert. Wie bei vielen computergrafischen Dokumenten ist hier das Buch von Foley [FvDFH95] die Hauptreferenz, danach der *OpenGL Programming Guide* [NDW93].

4.4.1 Geometrische Referenz

Für jede grafische Darstellung, nicht nur für die Computer Anwendung, braucht man mindestens eine geometrische Referenz. Für diese Software ist es notwendig, mehrere Referenzen zu benutzen (siehe Abb. 4.2):

- Die Anwendung verwendet und generiert ein oder mehrere Fenster. Diese Fenster kann man überall auf dem Bildschirm plazieren. Eine Bildschirmreferenz ist damit nötig. Diese Referenz ist normalerweise die linke untere Ecke des Bildschirms. Die X-Achse ist die untere Kante des Bildschirms und die Y-Achse die linke Kante des Bildschirms. Es handelt sich um ein Eulersystem. Mit dieser Referenz wird die Position des Fensters und die Größe (Skalierungsfaktor) definiert. In der Abbildung 4.2 sieht man zwei Fenster der gleichen Anwendung mit zwei verschiedenen Größen und Positionen. Der Fensterinhalt kann entweder eine feste Größe besitzen oder mit der Fenstergröße variieren.

- Innerhalb eines Fensters gibt es wieder eine Referenz. Alle Punkte im Fenster haben keine Bildschirmkoordinaten, sondern Fensterkoordinaten. Der Referenzpunkt liegt normalerweise (aber nicht unbedingt) in der linken unteren Ecke des Fensters. Die Achsen sind normalerweise parallel zur Bildschirm-Achse.
- Die Referenz der Anwendung wird auch Weltreferenz der Anwendung genannt. Für eine Flugführungsanwendung ist zum Beispiel die Mitte der Erde eine Weltreferenz. Dies kann ein System mit mehreren Dimensionen sein. Für die Flugführungsanwendung ist es entweder ein System mit zwei Dimensionen für die Skalen, oder ein System mit drei Dimensionen, zum Beispiel für die Geländedarstellungen.

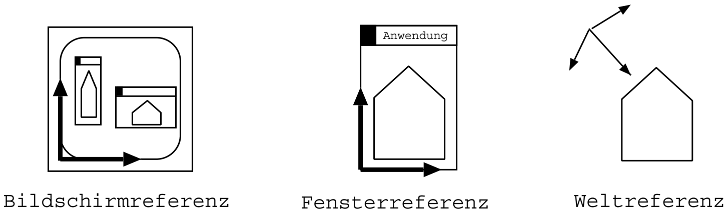


Abb. 4.2: Grafische Referenz in einer Softwarenanwendung

Für die Flugführungsanzeigen werden zwei Weltreferenzen definiert. Die erste ist für die 3D-Darstellung notwendig. Das Flugzeug bildet das *Zentrum* der Welt. Bei der perspektivischen Anzeige sind zum Beispiel die Referenzen bezüglich des Sichtpunktes wie folgt definiert (Körperfestes Achsenkreuz laut DIN 9300 [fN90]):

- X ist nach vorn gerichtet,
- Y ist nach rechts gerichtet,
- und Z ist nach unten gerichtet.

Um die Bewegung darzustellen, gibt es zwei Lösungen:

- Entweder das Flugzeug bewegt sich über ein statisches Gelände (exozentrische Darstellung),
- oder das Flugzeug ist ortsfest und das Gelände bewegt sich (egozentrische Darstellung).

Die Mehrheit aktueller PFD benutzt die zweite Lösung, eine egozentrische Darstellung, weil sie der Sichtwahrnehmung der Flugzeugbesatzung entspricht.

- Bei der Navigationsanzeige im ROSE Modus steht das Flugzeug fest in der Mitte der Anzeige, die anderen Elemente bewegen sich.
- Bei der perspektivischen Anzeige ist das Flugzeug ebenfalls ortsfest, der *künstliche Horizont*, oder die dargestellte Erde, bewegt sich: Wenn das Flugzeug seine Querneigungslage ändert bleibt das Flugzeugsymbol fest und die Querneigung wird durch die Bewegung der *Erde* gezeigt.

In der vorliegenden Flugführungsanzeigen-Software ist die Referenzposition der 3D-Darstellung fast immer genau dort, wo sich das Flugzeug befindet.

Für die 2D-Elemente sind die Achsen wie folgt definiert:

- X_{2d} ist parallel zur unteren Kante des Fensters und ist von links nach rechts ausgerichtet,
- Y_{2d} ist parallel zur linken Kante des Fensters und ist von unten nach oben ausgerichtet,
- Z_{2d} ist senkrecht zum Bildschirm und zum Betrachter ausgerichtet.

2D-Elemente brauchen ebenfalls eine dritte Dimension, z, wenngleich alle 2D-Elemente auf einer konstanten Ebene dargestellt werden. Im Abschnitt 4.6.7 wird erläutert, dass die 2D-Elemente wie auf einer Glasscheibe dargestellt sind. Diese Glasscheibe muss definiert sein, um zu berechnen, welche Elemente der 3D-Welt vor den 2D-Elementen dargestellt werden.

4.4.2 Planare Projektionen

Im vorherigen Abschnitt wurde bereits eine Projektion erwähnt: Die Weltreferenz kann mehr als zwei Dimensionen besitzen, die Darstellung erfolgt auf einem Bildschirm. Der Bildschirm kann auch im stereoskopischen Modus nur zwei Dimensionen darstellen.

Im Allgemeinen hat eine Projektion eines Koordinatensystems mit n-Dimensionen weniger als n-Dimensionen. Für eine grafische Softwarenanwendung gibt es zwei Arten von Projektionen. Sie unterscheiden sich durch die Position des Projektionszentrums:

- die perspektivische Projektion, bei der sich alle Abbildungsgeraden der Projektion im endlich entfernten Projektionszentrum schneiden,
- die Parallelprojektion, bei das Zentrum der Projektion im Unendlichen liegt, wodurch die Abbildungsgeraden der Projektion parallel sind.

Perspektivische Projektion

Eine perspektivische Projektion entspricht am ehesten der Lochkamera mit unendlicher Schärfentiefe. Wenn man als Beispiel die Photographie nimmt, ist der Film die Projektionsebene der Darstellung. Das Zentrum der Projektion darf nicht der Projektionsebene angehören. Wenn dieses Zentrum der Projektion zwischen einem Element und der Ebene steht, ist das Bild seitenverkehrt (erste Projektion auf der Abbildung 4.3). Die Größe eines Bildelementes hängt von seiner Entfernung vom Beobachter ab (zweite und dritte Projektion der Abbildung 4.3) sowie von der Entfernung des Zentrums der Projektion zur Projektionsebene.

Die Projektionsebene ist prinzipiell unendlich groß. Das ergibt für eine Anwendung mit der Darstellung auf einem Monitor endlicher Fläche keinen Sinn. Daher wird die Abbildung in der Höhe und der Breite begrenzt, wodurch ein Sichtfenster in der Projektionebene entsteht. Verbindet man das Projektionszentrum mit den Fensterrändern, entsteht die sogenannte Sichtpyramide. Die Grenze des Volums ist durch zwei Winkel definiert. Diese beiden Winkel sind gleich, wenn die Grundfläche der Pyramide ein Quadrat ist. Unter Angabe eines Seitenverhältnisses kann die Sichtpyramide auch mit nur einem Winkel definiert werden:

$$\frac{y}{x} = \frac{tan\frac{\text{vertical Field Of View}}{2}}{tan\frac{\text{horizontal Field Of View}}{2}} \tag{4.1}$$

Um die nötige Rechenleistung zu begrenzen, reduziert man die Tiefe der Pyramide zwischen zwei Ebenen: die *near clipping*-Ebene und die *far clipping*-Ebene. Die *near clipping* Ebene darf nicht zu weit entfernt sein, weil ein Objekt, das vor dieser Ebene steht, die Szene vielleicht verdeckt. Dieses Objekt wäre nicht im berechneten Bereich und würde folglich nicht dargestellt.

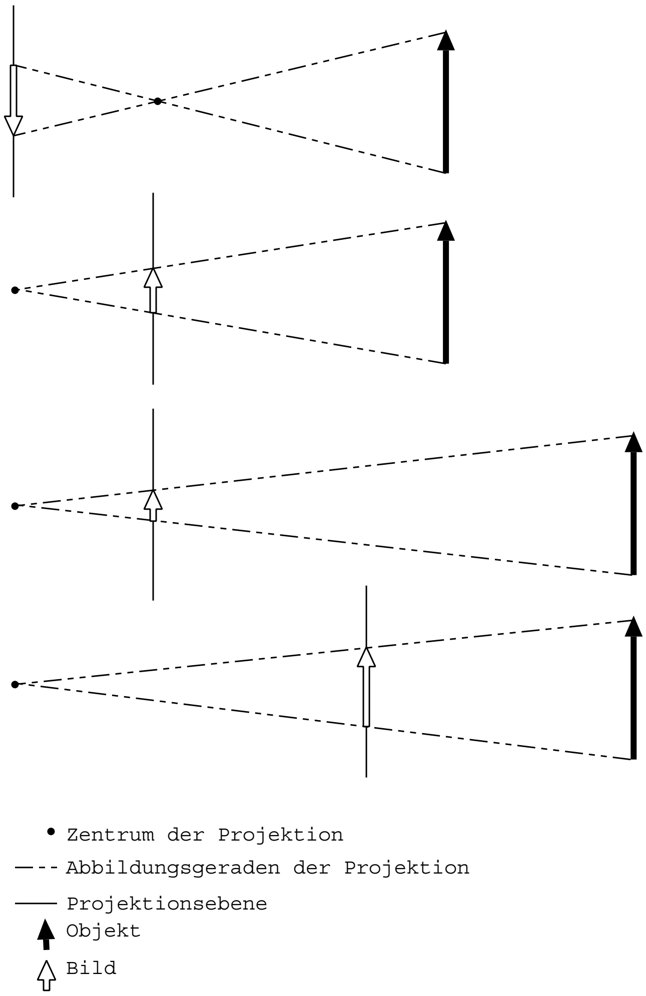


Abb. 4.3: Projektion

Zusammenfassend kann gesagt werden, dass eine Zentralprojektion in der Computergrafik durch einen Pyramidenstumpf definiert wird, wobei die Pyramidenspitze im Projektionszentrum liegt. Der Pyramidenstumpf ist vollständig beschrieben durch zwei Winkel (einen für die *vertikale,* der zweite für die *horizontale* Grenze) und zwei Ebenen die parallel zur Projektionsebene stehen: die *near* und *far clipping*-Ebene (siehe Abb. 4.4).

Grafische Orthogonalprojektion

Das Zentrum einer Parallelprojektion steht unendlich weit vor der Projektionsebene.

Bei einer parallelen Projektion gibt es ein begrenztes Volumen. Statt einer Pyramide ist die Form ein Würfel. Die Basis ist parallel zur Projektionsebene.

Parallelprojektionen werden benutzt, um die 3D-Welt der orthogonalen Anzeigen und die

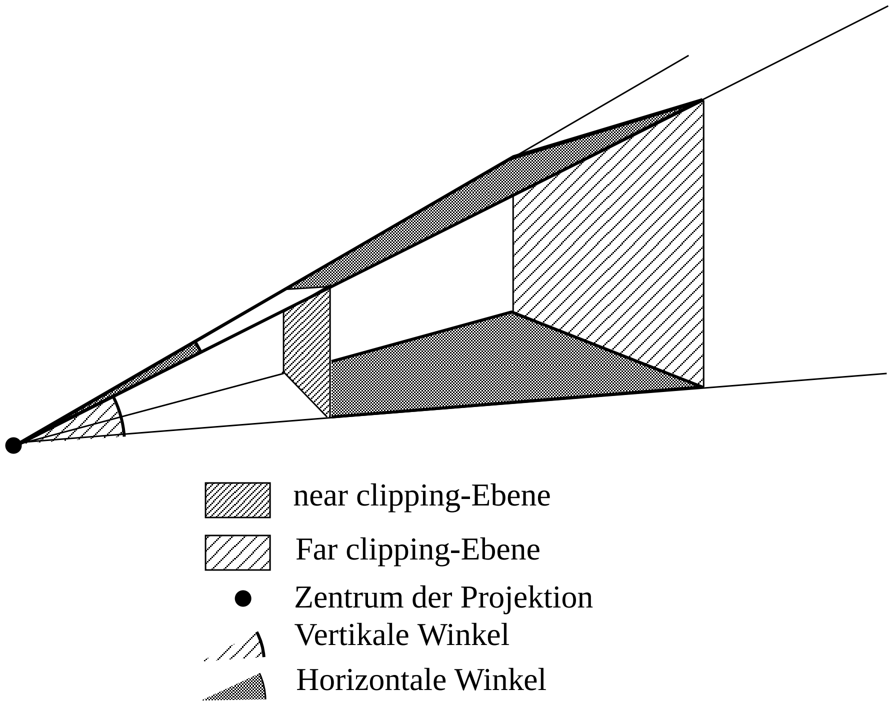


Abb. 4.4: Volumen einer perspektivischen Projektion

2D-Elemente abzubilden.

4.4.3 Homogene Koordinaten

Im vorherigen Abschnittel wurden verschiedene Projektionen erläutert. Eine Erklärung der mathematischen Formeln, die in der Software benutzt werden, wird jetzt nötig. Es wird gezeigt, warum ein allgemeines 3D-System nicht ausreicht, und was statt dessen benutzt wird.

Zu den allgemeinen geometrischen Transformationen gehören Rotation, Skalierung, Scherung und Translation. Die folgenden Transformationen werden in einem Eulersystem berechnet (x_0, y_0, z_0 sind die ursprüngliche Kordinaten eines Punktes P_0, x_1, y_1, z_1 die Koordinaten nach der geometrischen Transformation):

- eine Rotation (hier eine Rotation um die Z Achse von θ):

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} &= \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= R * P_0 \end{aligned} \tag{4.2}$$

- die Skalierung:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} &= \begin{pmatrix} scale_x & 0 & 0 \\ 0 & scale_y & 0 \\ 0 & 0 & scale_z \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= S * P_0 \end{aligned} \tag{4.3}$$

- die Scherung:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} &= \begin{pmatrix} 1 & shear & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= Sh * P_0 \end{aligned} \tag{4.4}$$

  x hängt von x und y ab: $Sh * [x, y, z]^T = [x + ay, y, z]^T$. Daher ist diese Transformation nicht orthogonal aber trotzdem affin. Diese Transformation ist für eine schräge Parallelprojektion nötig.

- die Translation:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} &= \begin{pmatrix} translation_x \\ translation_y \\ translation_z \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= T + P_0 \end{aligned} \tag{4.5}$$

Diese Matrixsysteme haben einen großen Nachteil: Die Berechnung einer Transformation ist hier entweder eine Addition für die Translation oder eine Multiplikation für alle anderen Transformationen. Außerdem sehen wir immer, dass für die Translation die *Transformations-Matrix* ein Vektor ist und, dass alle anderen mit $3 * 3$-Matrizen berechnet werden. Das heißt, dass man in der Software einen Test machen muss, um zu wissen, um welche Transformation es sich handelt. Dann wird die entsprechende Matrixoperation berechnet. Es ist nicht möglich, eine Folge Transformationen nur mit Matrizen zu realisieren, ohne zusätzliche Information und Logik zu verwenden.

Statt dreidimensionale Matrizen bzw. Vektoren zu benutzen, empfiehlt es sich, mit *homogenen Koordinaten* zu arbeiten. Ein Punkt wird so mit einem vierdimensionalen Vektor definiert:

$$P = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Dabei dürfen nicht alle Koordinaten null sein. Ein Punkt P hat unendlich viele Koordinaten, die den gleichen Ort in der dreidimensionalen Welt beschreiben: $[x, y, z, w]^T$ definiert den gleichen Punkt wie $[x * t, y * t, z * t, w * t]$. Man spricht von *homogenen Koordinaten* wenn $w = 1$: $P = (x_0, y_0, z_0, 1)^T$.

Mit den homogenen Koordinaten werden die Transformationen folgendermaßen berechnet:

- eine Rotation (Rotation um die Z Achse von θ):

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} &= \begin{pmatrix} cos(\theta) & sin(\theta) & 0 & 0 \\ -sin(\theta) & cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= R * P_0 \end{aligned} \tag{4.6}$$

- die Skalierung:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} &= \begin{pmatrix} scale_x & 0 & 0 & 0 \\ 0 & scale_y & 0 & 0 \\ 0 & 0 & scale_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= S * P_0 \end{aligned} \tag{4.7}$$

- die Scherung:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} &= \begin{pmatrix} 1 & shear & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= Sh * P_0 \end{aligned} \tag{4.8}$$

- die Translation:

$$\begin{aligned} \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & translation_x \\ 0 & 1 & 0 & translation_y \\ 0 & 0 & 1 & translation_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix} \\ \Leftrightarrow \quad P_1 &= T * P_0 \end{aligned} \tag{4.9}$$

Für die Rotation, Skalierung und Scherung ergibt sich fast keine Änderung gegenüber den $3 * 3$-Matrizen. Aber auch die Translation benutzt jetzt wie die anderen Transformationen eine Matrix, die das gleiche Format hat. Sie wird, wie die anderen Transformationen, durch Multiplikation einer Matrix mit einem Vektor berechnet. Hier ist kein Test mehr nötig, ein Datenverarbeitungsgraph ist ohne zusätzliche Information und auch ohne zusätzliche Logik möglich. Verschiedene Transformationen t_i werden dann so berechnet: $P_0 = t_1 * t_2 * ... * t_n * P_1$ wobei t_i irgendeine der vorherigen Transformationen darstellt.

Die Reihenfolge der Rechnung oder die *geometrische Transformation* ist zu beachten: $R * T$ ist nicht gleich $T * R$. Dies kann zu Problemen führen. Wenn ein Objekt schon translatiert wurde und man eine Drehung um eine Achse des Objektes durchführen möchte, muss man wie folgt rechnen: $T^{-1} * R * T$. Sonst erhält man die Drehung um eine andere Achse als die eigentlich gewünschte Achse des Objektes (siehe Abb. 4.5).

Um die Berechnung zu vereinfachen, wurden einige Klassen für diese Software entwickelt, die unter anderem die Reihenfolge der Berechnungen verwalten.

4.4.4 Grafische Grundelemente

Punkte sind für eine allgemeine geometrische Anwendung ein sehr wichtiges Grundelement, wenngleich Punkte für eine Darstellung des Geländes nicht geeignet sind.

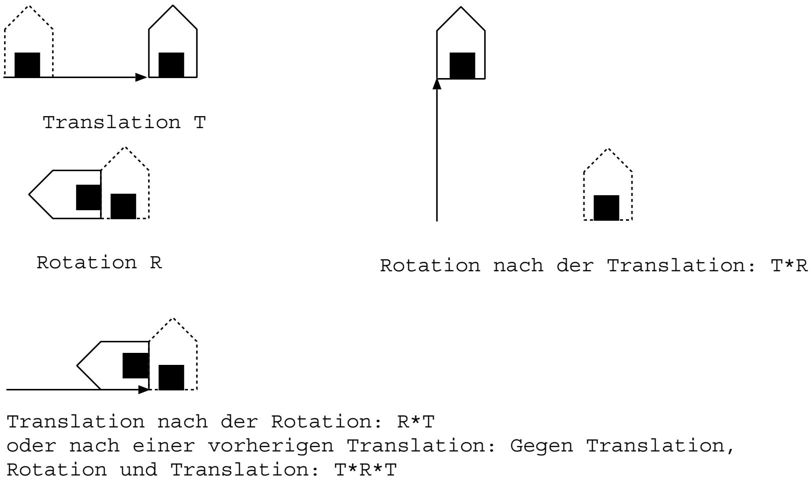


Abb. 4.5: Ordnung der Transformationen

Es ist üblich, Punkte zu verbinden und damit Linien zu bilden. Diese Darstellung wird *wireframe* genannt.

Ohne weitere Rechnung hätte diese Art der Darstellung eine sehr gute Performance und es ist keine Rechnung notwendig, um zu wissen, welche Fläche vor der anderen steht, da die Flächen nicht gefüllt und damit durchsichtig sind.

Es ist möglich, statt Linien Flächen zu benutzen. Flächen werden durch Linien begrenzt, die Strecken sind selbst durch Ecken, *Vertexes,* definiert. Eine Fläche kann teiltransparent sein und hat eine Farbe.

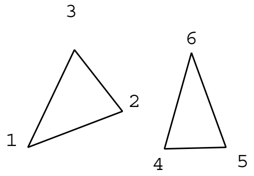
Gefüllte Polygone werden oft mit Hilfe von Dreiecken grafisch generiert. Die Flächen werden mit Ketten von Polygonen erzeugt, diese Polygone bestehen selbst aus Ketten von Dreiecken und die Dreiecke sind durch drei Punkte oder drei Koordinaten definiert, die über Strecken verbunden werden.

In den verwendeten Grafikbibliotheken ist die Ordnung der Punkte in einem Dreieck sehr wichtig: nur eine Seite eines Dreiecks ist gefärbt, die andere Seite ist durchsichtig. Um die gefärbte Seite zu definieren, ist eine Ordnung festzulegen: Die Punkte werden entweder im Uhrzeigersinn oder entgegen dem Uhrzeigersinn geordnet. In der Abbildung 4.6 sind die zwei ungeordneten Dreiecke entgegen dem Uhrzeigersinn gezeichnet.

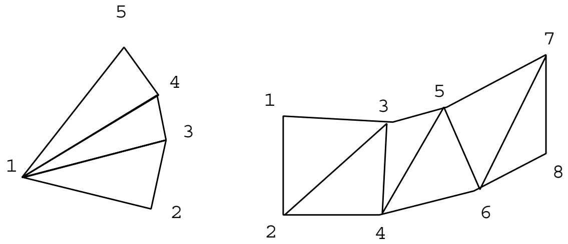
Die Reihenfolge der Dreiecke spielt eine Rolle für die Berechnungszeit. In der Abbildung 4.6 sind drei Beispiele zur Reihenfolge der Punkte und der Dreiecke gegeben. Bei den ungeordneten Dreiecken ist die Performance am schlechtesten. Die zwei anderen Beispiele zeigen Dreiecke, die in kürzerer Zeit berechnet und gezeichnet werden. Bei dem zweiten

Beispiel ist ein Punkt ein konstanter Punkt: Die Dreiecke sind 1,2,3-1,3,4-1,4,5. Diese Ordnung ist wichtig, um die Spitze zu zeichnen. Bei dem letzten Beispiel werden immer die zwei letzten Punkte eines Dreiecks für das nächste Dreieck benutzt: 1,2,3-3,2,4-3,4,5-5,4,6 usw.

Die Reihenfolge der Dreiecke ist daher eine sehr wichtige Anforderung an die Geländedatenbank.



ungeordnete Dreiecke



Rendering geordnete Dreicke

Abb. 4.6: Grafische Reihenfolge

4.4.5 Verdeckungsrechnung

In 4.4.4 wurde erläutert, dass eine Seite eines Dreieckes gefärbt ist, und dass die andere durchsichtig ist. Die gefärbte Seite soll also den Hintergrund verdecken, um die Tiefenschlüssel nachzubilden.

Das gängigste Verfahren der Verdeckungsrechnung ist das Z-Buffer-Verfahren. Als Vorteil gegenüber anderen Verfahren ist hier keine Vorsortierung der Punkte bezüglich der Tiefe notwendig. Diese Regel ist für die Flugführungsanzeige sehr wichtig, da die Tiefenstaffelung der orthogonalen Anzeige und der perspektivischen Anzeige sich unterscheiden: Die Ordnung der Ebenen beim PFD erfolgt entlang der X Achse. Eine Fläche A verdeckt eine Fläche B, wenn $x_a < x_b$. Bei einer Navigationsanzeige muss man die Flächen nach ihren Z-Koordinaten sortieren. Wenn die Verdeckungsrechnung eine Ordnung der Punkte oder der Flächen bezüglich der Tiefe benötigte, würde das bedeuten, dass man zwei getrennte Datenquellen, eine für das HDPFD und eine für das ND, bräuchte.

Zur Realisierung des Z-Buffer-Algorithmus wird ein Feld von Bildschirmpixeln erzeugt. Für einen Punkt P_{W1} der 3D-Welt (siehe Abb. 4.7) wird ein Punkt P_{P1} auf der Projektionsfläche berechnet. Die Koordinate des Punktes auf der Projektionsfläche entspricht der Koordinate des Feldes. Die Entfernung D_{W1} zwischen dem Sichtpunkt P_{P1} und P_{W1} wird in diesem Feld gespeichert. Dazu wird zusätzlich die Farbe der Punkte gespeichert. Wenn ein anderer Punkt P_{W2} auch auf P_{P1} projiziert wird, wird die Entfernung D_{W2} von P_{W2} mit D_{W1} verglichen. Im Z-Buffer-Feld werden die Entfernung und die Farbe des näheren Punktes gespeichert: Hier P_{W2} bzw D_{W2}.

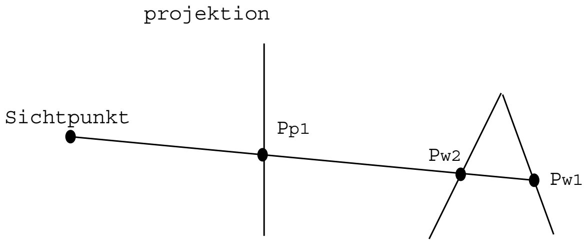


Abb. 4.7: Z-Buffer und Projektion

4.4.6 Koplanarität

Falls Koplanarflächen in der Datenbank existieren, muss das Problem der Koplanarität durch Software gelöst werden. Das Problem der Koplanarität entsteht durch die begrenzte Präzision des Rechners. Wenn zwei Flächen sehr nahe bei einander sind, werden sie wegen des Z-Buffer-Algorithmus und wegen der unzureichenden Präzision der Berechnung flackern: Einmal wird die Fläche A eine Fläche B verdecken, im nächsten Bild vielleicht umgekehrt. Dieses Flackern irritiert die Betrachter.

Die zwei Flächen A und B können auch auf der gleichen Ebene liegen. In der Flugführungsanzeige kann dieser Effekt oft passieren: Eine *Taxiline* liegt auf einer Landebahn, die Landebahn selbst befindet sich auf dem Gelände. Man muss hier eine Logik einfügen, um die Ordnung der Flächen für die Darstellung zu beschreiben.

Als erste Lösung wird jede Ebene bezüglich der vorherigen in der Tiefe verschoben. Diese Verschiebung hängt von dem Sichtpunkt und dem Ort ab, wo sich die verschiedenen Koplanarflächen befinden. Diese Lösung ergibt die beste Performance. Andererseits aber, befindet man sich in der Nähe der beiden Flächen, kann man in einigen Fällen den Offset sehen, was wiederum zu Fehlinterpretationen oder Irritationen führen kann.

Nach einem anderen Verfahren werden alle Flächen nacheinander gezeichnet. Für dieses Verfahren spielt die tiefste Fläche eine besondere Rolle: Diese Fläche ist die Referenz, um die Verdeckung mit den Nicht-Koplanarelementen zu definieren. Die erste Fläche wird als erste dargestellt, dann ersetzen die anderen Koplanarflächen die Farbe der ersten Fläche. Dieses Verfahren ist mit einigen Grafikkarten nicht viel langsamer als das vorherige. Aber die Performance kann sich auch sehr stark verschlechtern. In der Flugführungsanzeige-Anwendung ist für die Geländedarstellung die Erde die erste Fläche, die anderen Flächen sind immer über dem Gelände. Dann werden die verschiedenen Flächen sortiert, und sie werden in der gleichen Ordnung gezeichnet. Zum Beispiel kommen nach dem Boden die *Taxiways*, dann die *Taxilines*, darüber die Landebahn und letztlich die Landebahnmarkierungen.

4.4.7 Beleuchtung

Die physikalischen Eigenschaften des Lichts zu modellieren wäre zu komplex; es geht in diesem Kapitel also nur darum, die wichtigsten Effekte der Lichtstrahlung künstlich zu generieren. Einige Vereinfachungen bezüglich der physikalischen Ausleuchtung werden gemacht, um Performanceverluste zu vermeiden. Aber um nicht ganz unrealistisch zu werden, dürfen diese Vereinfachungen nicht zu umfangreich sein.

Als erste und einfachste Implementierung, wird jedes Objekt selbstleuchtend generiert. Alle Materialien sollen eine Intensität k_i haben. Eine Darstellung nur mit einem solchen Verfahren der Beleuchtung wirkt unrealistisch.

Anstelle einer eigenen Beleuchtung leuchtet eine externe Quelle die Flächen diffus aus. Wenn diese Quelle nicht gerichtet ist, kann man ein ambientes Licht definieren. Jedes Material benutzt dazu einen Faktor, das ambiente Reflexionvermögen des Materials. Diese Rechnung ist realistischer, denn jedes Material hat ein anderes Reflexionvermögen.

In Wirklichkeit ist die Lichtquelle gerichtet. Die Reflexion ändert sich mit dem Winkel zwischen der Normalen der beleuchteten Fläche und der Richtung der Lichtquelle. Es ist möglich, die Normale der Fläche zu berechnen. Für ein Gelände ist es für die Performance günstiger, die Normale jedes Polygons vorher zu berechnen und diese Werte bereits in der Datenbank abzuspeichern. Eine solche Vorausberechnung ist nur möglich, wenn keine Scherungtransformation vollzogen wird. Eine Scherungstransformation erfordert eine neue Berechnung des Normalenvektors, andernfalls würden die angegebenen Normalenvektoren nicht senkrecht zu ihren Flächen sein. Daher muss man in diesem Fall ein solche Transformation vermeiden.

Der nächste Effekt ist die gerichtete, *spiegelnde* Reflexion. Diese Reflexion ist wichtig bei stark reflektierendem Material wie Wasser, Glas oder Metall. In einer Geländedarstellung betrifft dies jedoch nur Wasserflächen. Weil die Rechnung noch komplexer wird und die Performance sich zusätzlich verschlechtert, wurde dieser Effekt nicht modelliert.

Schattierung

Auch die Schattierung ändert die Objektfarbe. Dazu gibt es einige Algorithmen in der Literatur. Zur Zeit wird der Algorithmus, der auf dem Z-Buffer-Algorithmus basiert (siehe Kapitel 4.4.5), benutzt. Das Z-Buffer-Verfahren ist bekannt und in vielen Rechnern ist es direkt implementiert. Die erste Version dieses Algorithmus wurde von Williams entwickelt (siehe [FvDFH95]) und ist als *two-pass z-Buffer Shadow Algorithm* (Zweifach-Durchlauf Z-Buffer-Schattierungsalgorithmus) benannt. Der erste Durchlauf wird aus dem Sichtpunkt der Lichtequelle berechnet. Dann wird dieser Algorithmus aus dem Sichtpunkt des Beobachters oder der Kamera wiederholt.

Der zweite Durchlauf ist für jede Lichtquelle nötig. Obwohl das Z-Buffer-Verfahren ein sehr bekanntes Verfahren ist und leistungsmäßig gute Ergebnisse ergibt, benötigt ein Z-Buffer-Durchlauf eine lange Rechenzeit. Man darf nicht vergessen, dass man theoretisch für jeden Punkt einer Szene den Algorithmus wiederholen muss. Das heißt, dass diese Rechnung sich sehr oft wiederholt. Darum wird man diese Berechnung möglichst vermeiden und allenfalls für die Hauptlichtquelle verwenden.

Jedoch hat M. Purpus [Pur99] für die perspektivische Anzeige einen Flugprädiktor entwickelt. Für dieses 3D-Symbol erklärt er die Notwendigkeit eines Schattens. Dieser Schatten muss mit einer anderen Lichtquelle oder einem anderen Standpunkt der Sonne berechnet werden. Die Lichtquelle muss im Zenit stehen, damit der Schatten genau unter diesen Prädiktor fällt. Der Schatten des Prädiktors kann deshalb nicht mit dem normalen Verfahren berechnet werden, weil dies bedeuten würde, dass auch andere Elemente auf dem Gelände von dieser *Sonne* einen Schatten hätten. Die Schatten der anderen Elemente wären mit der Lichtquelle (die in der linken oberen Ecke der orthogonalen Anzeige sitzt) inkonsistent. Der Prädiktorschatten wird als einfache Fläche unter dem Prädiktor dargestellt. Die Form der Fläche wird vorher modelliert. Zur Lösung wird die Fläche mit den Koordinaten x,y des Prädiktors und z, der aktuellen Höhe des Flugzeugs über dem Boden, verwendet. Diese Lösung genügt nicht, wenn das Gelände nicht horizontal ist. Um dies zu verbessern, wird diese Fläche als Koplanarfläche (siehe Kapitel 4.4.6) gezeichnet. Für eine kleine positive Neigung reicht diese Lösung aus. Für große Neigung wird der Schatten nicht richtig gezeichnet: Er wird nicht auf dem Gelände dargestellt, sondern in der Luft. Eine Art von Textur sollte dann als alternative Lösung betrachtet werden.

Transparenz

Der Schatten des Prädiktors wird durch eine Fläche auf dem Boden dargestellt. Aber ein Schatten ist nie nur eine schwarze Fläche, ein Schatten soll die Farben verdunkeln. Um dies zu realisieren, wird die Fläche als durchsichtige schwarze Fläche definiert. Um die mögliche Verschmelzung von 2D-Elementen mit der Geländedarstellung zu verhindern, kann ebenfalls Transparenz eingesetzt werden (siehe 4.6.11).

Normalerweise reflektieren durchsichtige Elemente teilweise auch das Licht. Die Reflexion hat eine lange und komplexe Berechnung zur Folge. Reflexionlosigkeit ist nicht ganz konform mit der Realität, wirkt sich subjektiv beurteilt aber kaum aus. Deswegen wird hier eine einfache Farbänderung der Flächen hinter dem transparenten Element ohne zusätzliche Reflexion berechnet.

4.4.8 Farbverlauf

Der Farbverlauf (*Shading*) einer Fläche wird basierend auf der Farbe der Eckpunkte und der Beleuchtung durchgeführt. Theoretisch ergibt sich der Farbverlauf mit einer Berechnung der Normalenfläche für jeden Punkt. In diesem Unterkapitel sind nur die Hauptverfahren beschrieben.

Flat Shading

Der einfachste und schnellste Verlauf ist das Konstant- oder *Flat-Shading*. Dieser Verlauf benutzt ein Beleuchtungsmodell zur Berechnung eines einzigen Intensitätswertes. Dieser Wert wird benutzt, um ein ganzes Polygon zu färben. Für jedes Polygon wird eine neue Intensität berechnet. Diese Lösung ist unter folgenden Bedingungen gültig:

- Wenn die Lichtquelle unendlich weit vom Polygon entfernt ist, damit $\vec{N} \cdot \vec{L}$ konstant bleibt (mit $\vec{N}$ als Normalvektor und $\vec{L}$ als Richtungsvektor der Lichtquelle)
- Wenn der Betrachter unendlich weit vom Polygon entfernt ist, damit $\vec{N} \cdot \vec{V}$ konstant bleibt (mit $\vec{V}$ als Richtungsvektor des Betrachters *Viewer*)
- Wenn das Polygon keine Näherung einer gekrümmten Fläche ist, sondern wenn das Polygon die korrekte Fläche modelliert.

Dieses Verfahren ist einfach und daher schnell. Andererseits ergibt es einen sehr schlechten Eindruck für nahe Darstellung: Eine Fläche wird aus verschiedenen Polygonen generiert. Jedes Polygon hat seine bestimmte Farbe, daher sieht man die Kanten der Polygone deutlich. Daraus folgt eine eckige, unruhige und inhomogene Darstellung.

Gouraud Shading

Um einige Nachteile des *Flat Shading* zu beheben, hat Gouraud einige neue Verfahren entwickelt. Das *Gouraud Shading* benutzt eine gerichtete Lichtquelle.

Gegen den Kanteneffekt wird die Farbe eines Polygons mit der Farbe der benachbarten Polygone linear interpoliert. Bei einer kontinuierlichen Änderung sieht man normalerweise die Grenze der Polygone einer Fläche nicht mehr.

Für die Interpolation sind die Normalen der Ecken, *Vertexes*, (siehe Abb. 4.8) der Polygone, und die Lichtintensität der Ecken statt der Flächen nötig. Die Punktnormalen werden wie folgt berechnet:

$$\vec{N}_v = \frac{\sum_{i=1}^{n} \vec{N}_i}{\| \sum_{i=1} n\vec{N}_i \|} \tag{4.10}$$

Die Berechnung ist aufwendig, daher werden diese Punktnormalen besser vorberechnet und in der Datenbank gespeichert.

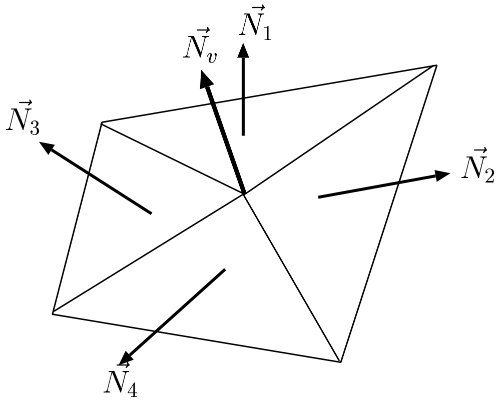


Abb. 4.8: Flächen und Punktnormale:

Für jeden Eckpunkt eines Polygons wird schließlich die Intensität des Lichts berechnet. Dann werden I_a und I_b (siehe Abb. 4.9) interpoliert: Die Farbe von a ändert sich mit der Entfernung von Punkt 1. Wenn $y_p = y_1$ ist, wird die Intensität I_a genau I_1. Je weiter a von Punkt 1 entfernt ist bzw. je näher a Punkt 2 ist, desto ähnlicher ist die Intensität des Punkts a der des Punktes 2.

Mit dem gleichen Prinzip wird die Intensität des Punktes b berechnet, wobei y_a und y_b immer gleich sind. Ein zweiter Durchlauf wird in x-Richtung gemacht, um die Intensität von P von I_a und I_b zu interpolieren.

$$\begin{aligned} I_a &= I_1 - (I_1 - I_2)\frac{y_1 - y_p}{y_1 - y_2} \\ I_b &= I_1 - (I_1 - I_3)\frac{y_1 - y_p}{y_1 - y_3} \\ I_p &= I_b - (I_b - I_a)\frac{x_b - x_p}{x_b - x - a} \end{aligned} \tag{4.11}$$

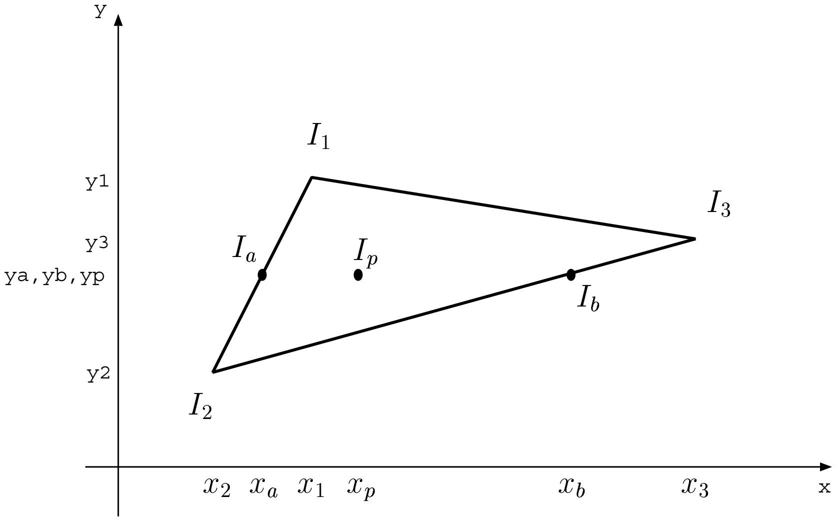


Abb. 4.9: Interpolation der Intensität

Ein Eckpunkt gehört zu mehreren Dreiecken. Die Abbildung 4.10 zeigt ein interessantes Problem: Der Punkt F ist kein Eckpunkt des linken Dreiecks ABC. Die Gerade AC wird sehr wahrscheinlich sichtbar sein, wegen zwei verschiedener Interpolationen. Die erste Farbinterpolation wird mit dem Dreieck ABC durchgeführt, die zweite kommt von den drei anderen Dreiecken AEF, FED und FDC. Hieraus ergibt sich also eine Anforderung an die Datenbank : Eine solche Konfiguration darf nie auftreten. Ein Algorithmus für die Triangulation sollte dies vermeiden. Eine Erklärung über Algorithmen für die Triangulation erfolgt in [O'R93].

Phong Shading

In der Literatur findet man auch das *Phong Shading*. Dies ist ein Farbverlauf, mit dem Reflexionen modelliert werden. Dieser Verlauf benötigt eine viel höhere Rechenzeit als das *Gouraud Shading*. In einer Geländedarstellung gibt es wenig Reflexion: Hauptsächlich reflektieren des Wassers und einige künstliche Elemente in Städten. Der Aufwand,

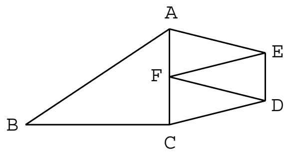


Abb. 4.10: Verteilte Eckpunkte

um für wenige Elemente eine kleine Verbesserung der Realitätsnähe in der Darstellung des Geländes zu erhalten, ist zu groß.

Der Farbverlauf des *Gouraud Shading* scheint zur Zeit der Geeignetste. Er verbessert bei tragbarem Rechenaufwand deutlich die Qualität der Darstellung gegenüber dem *Flat Shading*. Die Abbildung 4.11 zeigt die gleiche Geländedarstellung, einmal mit einem *Flat-* und einmal mit einem *Gouraud Shading*.

4.4.9 Antialiasing

Ein bekannter und unangenehmer Effekt tritt bei dem Zeichnen von Linien auf. Durch die auf einer Pixelmatrix basierenden Darstellung des Bildes erscheinen Linien oft eckig, sofern sie nicht horizontal oder vertikal verlaufen. Eine höhere Auflösung kann die Qualität der Darstellung verbessern. Um den Effekt noch stärker abzumildern, kann man eine Interpolation wie die Interpolation des *Gouraud Shading* benutzen. Die Geraden werden etwas dicker gezeichnet, wobei die Randpixel stufenweise an die Farbe des Hintergrunds angepasst werden. Diese Dämpfung ist als *Antialiasing* bekannt. Es gibt einige Grafikkarten, die die Darstellung intern glätten. In diesem Falle verbessert sich die Darstellung deutlich. Für diese Arbeit kann man davon ausgehen, dass die Anzeige mit einer solchen Grafikkarte realisiert wird.

4.4.10 Textur

Um die Realität der Darstellung zu verbessern, gibt es die Möglichkeit, auf das Gelände eine *Bitmap*, beispielsweise ein digitalisiertes Satellitenbild, zu *legen*.

Eine Textur erlaubt eine feine und realistische Darstellung (wenn eine Phototextur verwendet wird), die ohne Textur sehr viele Polygone erfordern würden. Um dieses Photo oder Bild auf eine Oberfläche zu *legen*, muss man für jedes Polygon die Verdeckung und die Beleuchtung berechnen. Hier kann man die übrigen Punkte des aktuellen Z-Buffers

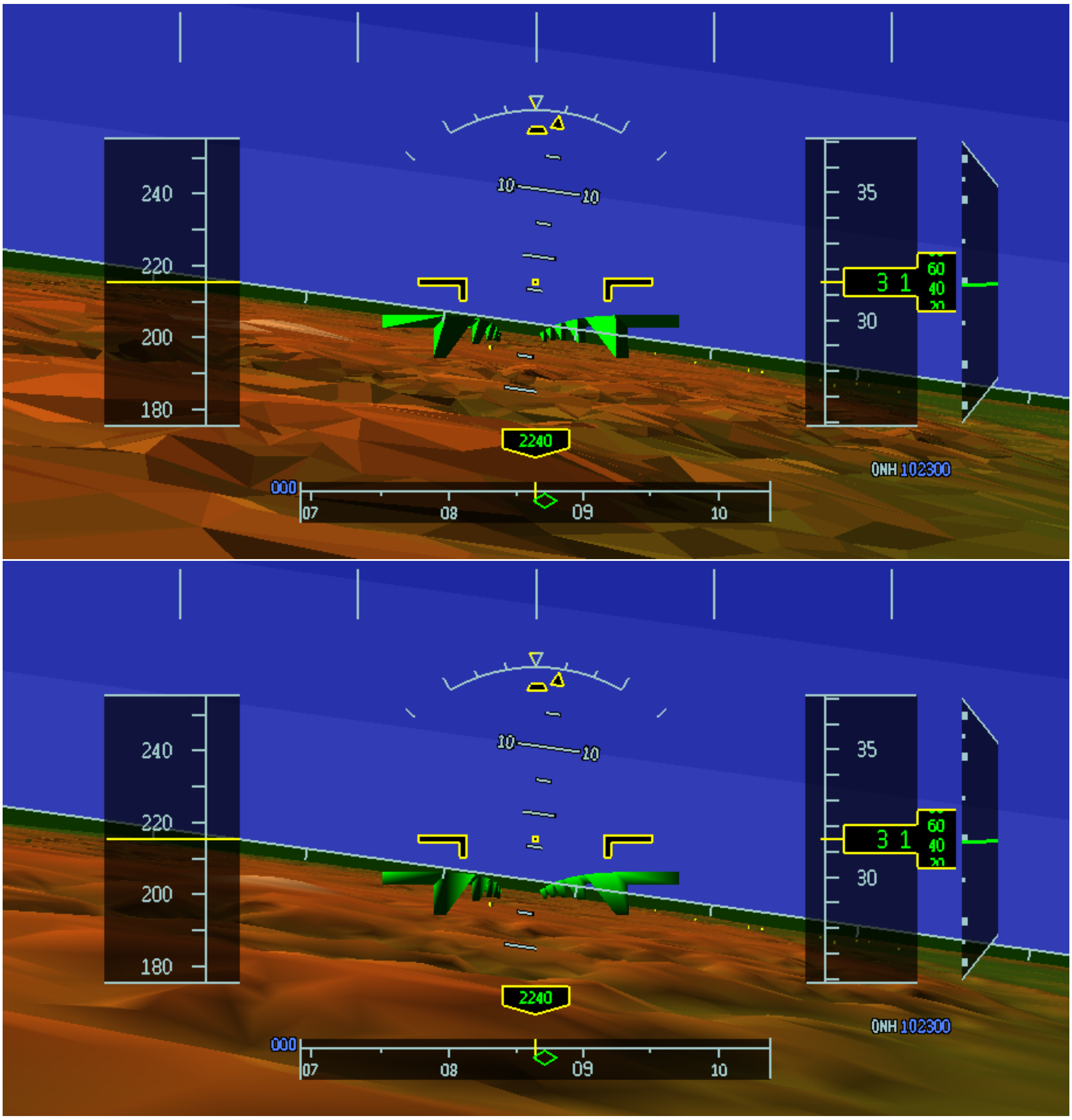


Abb. 4.11: Flat und Gouraud Shading

mit einer zusätzlichen Rechnug an die Farbe der Textur anpassen. Die Textur ersetzt den Farbverlauf (z.B. das *Gouraud Shading*).

Eine Textur kostet viel Rechenzeit, es sei denn, man verwendet Grafikkarten, die speziell auf die Verarbeitung von Texturen ausgelegt sind. Auch in diesem Falle ist die Größe der Textur entscheidend: Die Textur liegt oft in einem speziellen Speicherbereich, der begrenzt ist. Wenn zu viele Texturen benötigt werden, muss der Rechner ständig Kopien von einem Speicherbereich in einen anderen erstellen, was wieder die Performance reduziert.

Die Qualität der Darstellung hängt stark von der Qualität der Textur ab. Wenn die Textur eine hohe Auflösung hat, kann man sie problemlos aus geringer Entfernung betrachten. Wenn aber die Auflösung nicht ausreicht, wird die Darstellung pixelig oder unscharf. Abbildung 4.12 zeigt die gleiche Szene in zwei verschiedenen Flughöhen. Man sieht hier deutlich, dass die Auflösung der Textur an dieser Stelle nicht ausreicht.

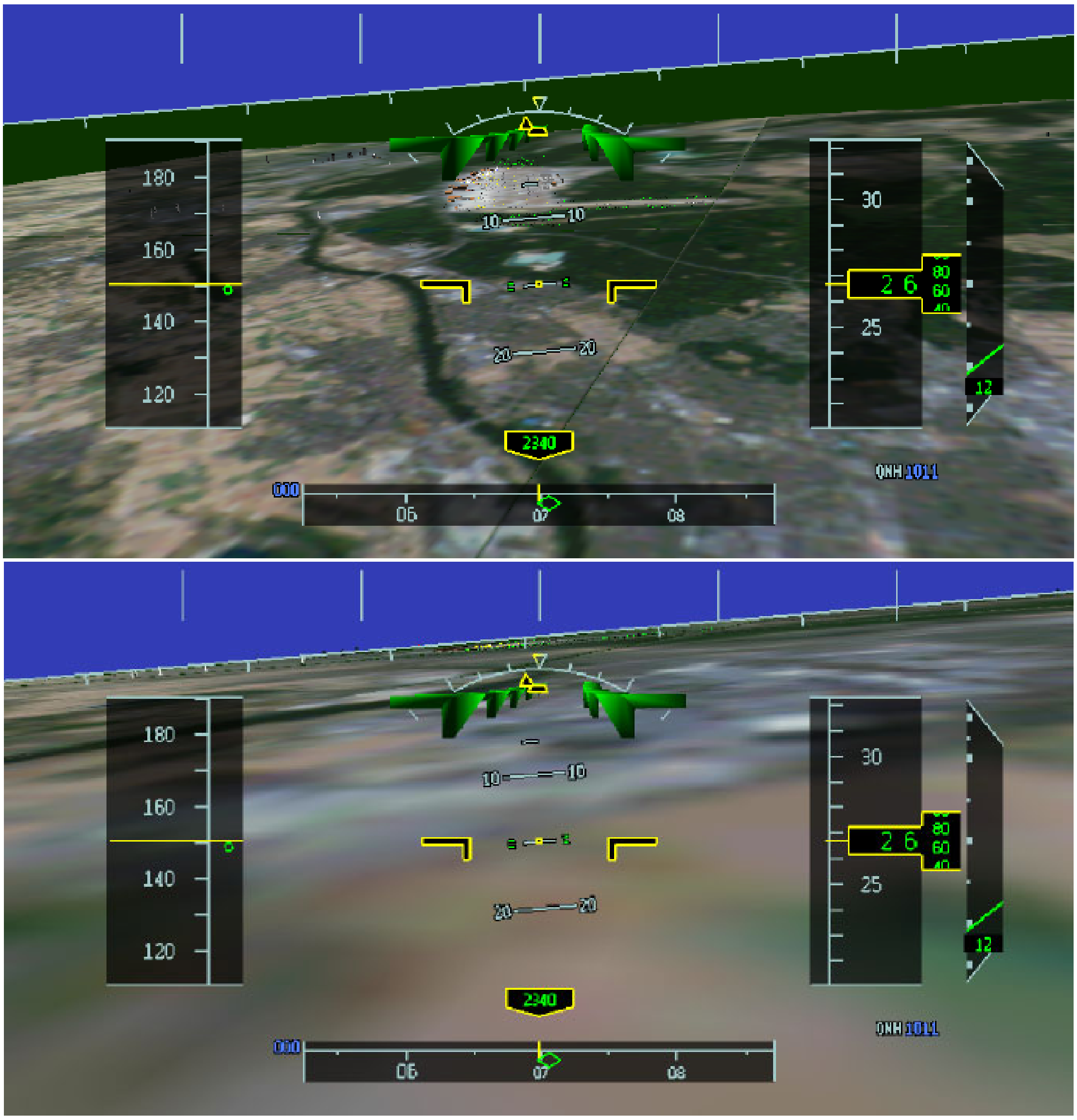


Abb. 4.12: Darstellung mit Textur

Eine Textur wird im Allgemeinen benutzt, um Kulturgelände darzustellen: Wälder werden ungefähr als grüne Flächen dargestellt, eine Wüste wird braun sein usw. Eine zusätzliche Höhenfarbkodierung ist unmöglich.

Für eine gute Qualität der Darstellung einer Flugführungs-Anzeige scheint zur Zeit die Textur keine gute Lösung zu sein. Es würde eine zu große Auflösung der Textur benötigt, woraus dann eine schlechte Performance resultieren würde. Wenngleich diese Software auch Darstellungen mit Textur erlaubt, ist dies zur Zeit nicht zu empfehlen.

4.4.11 Level Of Detail

Eine zweckmäßige Anzahl von Polygonen eines modellierten Objekts hängt von der Entfernung der Kameraposition zum Objekt ab. In großer Nähe wird eine hohe Anzahl von Polygonen benötigt, um das Objekt in ausreichender Präzision darstellen zu können. Im Gegensatz dazu ist in großer Entfernung eine hohe Anzahl von Polygonen nicht nur unnötig, sondern unter Umständen sogar ungünstig für die grafische Qualität. Übersteigt die rechnerisch mögliche Auflösung des Objektes die Auflösung des Bildschirmes, kann dies zu groben Verzerrungen führen (siehe Abb. 4.13).

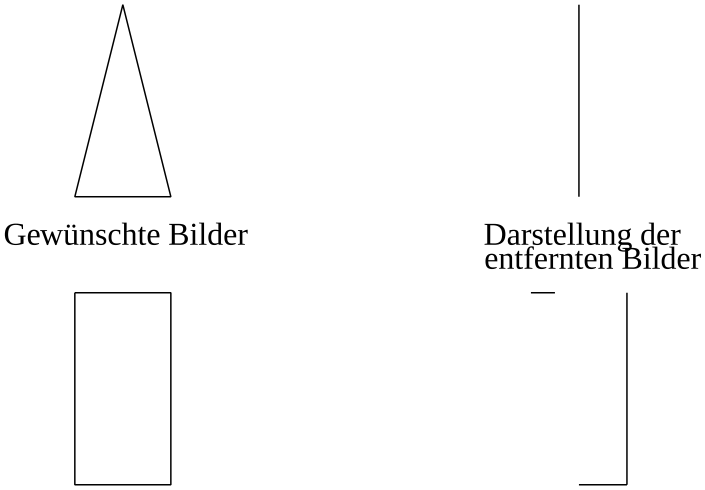


Abb. 4.13: Verzerrungen durch zu hohe Anzahl von Polygonen

Das Verfahren *Level of Detail*, LOD, berücksichtigt das beschriebene Problem, indem für ein und das gleiche Objekt mehrere Modelle mit unterschiedlicher Polygonanzahl gespeichert werden. Abhängig von der Entfernung wird ein geeignetes Modell dargestellt.

Zusätzlich kann mit LOD auch die Performance beeinflusst werden, indem bei unzureichender Rechnerleistung schrittweise die Modellqualität der Objekte reduziert werden kann.

Während das LOD-Verfahren in perspektivischen Anzeigen wie einem HDPFD sehr sinnvoll ist, ist dessen Einsatz in einer Navigationsanzeige nicht problemlos möglich. Diese Anzeigen benutzen eine parallele Projektion. Für ein ebenes Gelände ist der Punkt, der sich genau unter dem Betrachter befindet, der nächste. Die Punkte am Rand der Anzeige sind weiter entfernt vom Betrachter. Daher wird für Objekte am Rand der Anzeige ein weniger detailliertes Modell herangezogen, als für Objekte im zentralen Bereich der Anzeige. Für eine Anzeige mit einer parallelen Projektion soll die Qualität des Modells aber innerhalb des gesamten Anzeigebereichs konstant sein und nicht von der rechnerischen Entfernung abhängen, sondern von der gewählten Zoomstufe.

Der *Level Of Detail* (LOD) eines Objekt hängt also von der Art der Anzeige ab. Die eine hängt von der Entfernung ab (für die perspektivischen Anzeigen), die andere von der Zoomstufe (für die orthogonalen Anzeigen).

4.4.12 Stereoskopische Darstellung

Stereoskopische Darstellungen nutzen die Fähigkeit des Menschen zur dreidimensionalen visuellen Informationserfassung, wobei hierbei jedem Auge ein eigenes Bild präsentiert wird, das sogenannte Halbbild (siehe [Met75]). Die Halbbilder werden vom visuellen System des Menschen fusioniert, so dass ein echter räumlicher Tiefeneindruck entsteht [May01b]. Es gibt verschiedene Verfahren, um die stereoskopische Umgebung künstlich mit einer 2D-Darstellung zu reproduzieren. Auf jeden Fall ist es notwendig eine Darstellung für jedes Auge zu erzeugen, solange Holographie noch nicht einsetzbat ist. Die Zentren der Projektion befinden sich für jede dieser beiden Darstellungen an einer anderen Position, die beiden Projektionsebenen sind selten identisch.

Hammer [Ham98] beschreibt ein stereoskopisches PFD. Die in dieser Arbeit entwickelte Flugführungsanzeige soll für die verschiedenen Anzeigen (PFD, HUD, ND, HMD) stereoskopische Darstellung erlauben und mehrere Techniken zur Trennung der Halbbilder unterstützen.

Die stereoskopische Darstellung wird in verschiedenen Anzeigen unterschiedlich verwaltet. Die folgende Liste ist ein Vorschlag und die Anzeige soll solche Möglichkeiten bieten oder mindestens vorbereiten:

- Aufbau des Stereobildes:
    - Zwei getrennte Fenster für ein stereografisches Anzeigemedium oder HMD

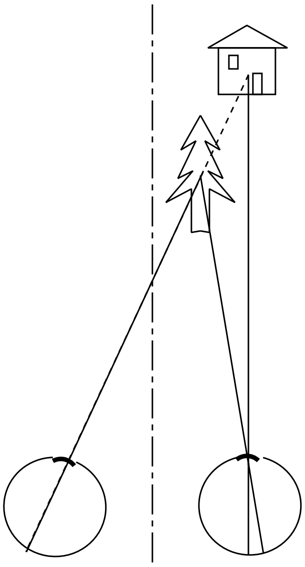


Abb. 4.14: Stereoskopische Sicht

  - Ein Fenster mit zwei Darstellungen mit zwei mal zwei Buffern (zwei für das linke Auge und zwei für das rechte Auge); die beiden Buffer sind so groß wie das Fenster. Für *Shutter-Brillen*-Darstellung mit hoher Bildschirmfrequenz. Solche Darstellungen werden zum Beispiel für das PFD benutzt.
  - Ein Fenster mit der gleichen Größe wie für eine monoskopische Darstellung, aber vertikal gestaucht. Dies gilt für Bildschirme mit *interlace* Mode.
- Stereoskopische Parameter:
  - Man muss die Möglichkeit vorsehen, die Zentren der Projektion zu verschieben, um Parallaxen zu ändern.
  - Die Richtung der Projektion muss variabel sein.

4.5 Systemüberblick

Bezüglich der Darstellung ist die Software in zwei Hauptteile, die zweidimensionale (Kapitel 4.6.1) und die dreidimensonale Darstellung (Kapitel 4.7) getrennt. *Display2D* ist der Verwalter der 2D-Elemente. Die 3D-Elemente werden von *display3D* verwaltet.

Display verwaltet die Grundfunktionalität einer Anzeige. Die betroffenen Funktionen gelten gleichzeitig für Anzeigen, die nur 2D-Elemente (wie das HUD), nur 3D-Elemente (wie das Visual), oder 2D- und 3D-Elemente (wie das PFD oder ND usw.) verwenden.

Werte, die die Flugdynamik wiedergeben, zum Beispiel die aktuelle Position des Flugzeugs und seine Geschwindigkeit, werden von anderen Anwendungen mit Hilfe einer Kommunikationsschnittstelle (Kapitel 4.8) gelesen.

Eine Anwendung spielt für die Flugführungsanzeigessoftware eine besondere Rolle: die Bedienungsanwendung. Sie erlaubt unter anderem, die Elemente zu variieren oder ein neues Format zu definieren. Sie legt zum Beispiel die Position und die Größe der 2D-Elemente fest.

Initialisierungswerte werden soweit wie möglich nicht im Quellcode fest implementiert, sondern aus Initialisierungsdateien gelesen. Eine Klasse zur Verwaltung solcher Dateien wird in Kapitel 4.6.9 beschrieben.

Die Abbildung 4.15 zeigt vereinfacht die Generierung einer Bildschirmanzeige. Sie zeigt die Verbindung der Flugführungsanzeiges-Software mit anderen Anwendungen.

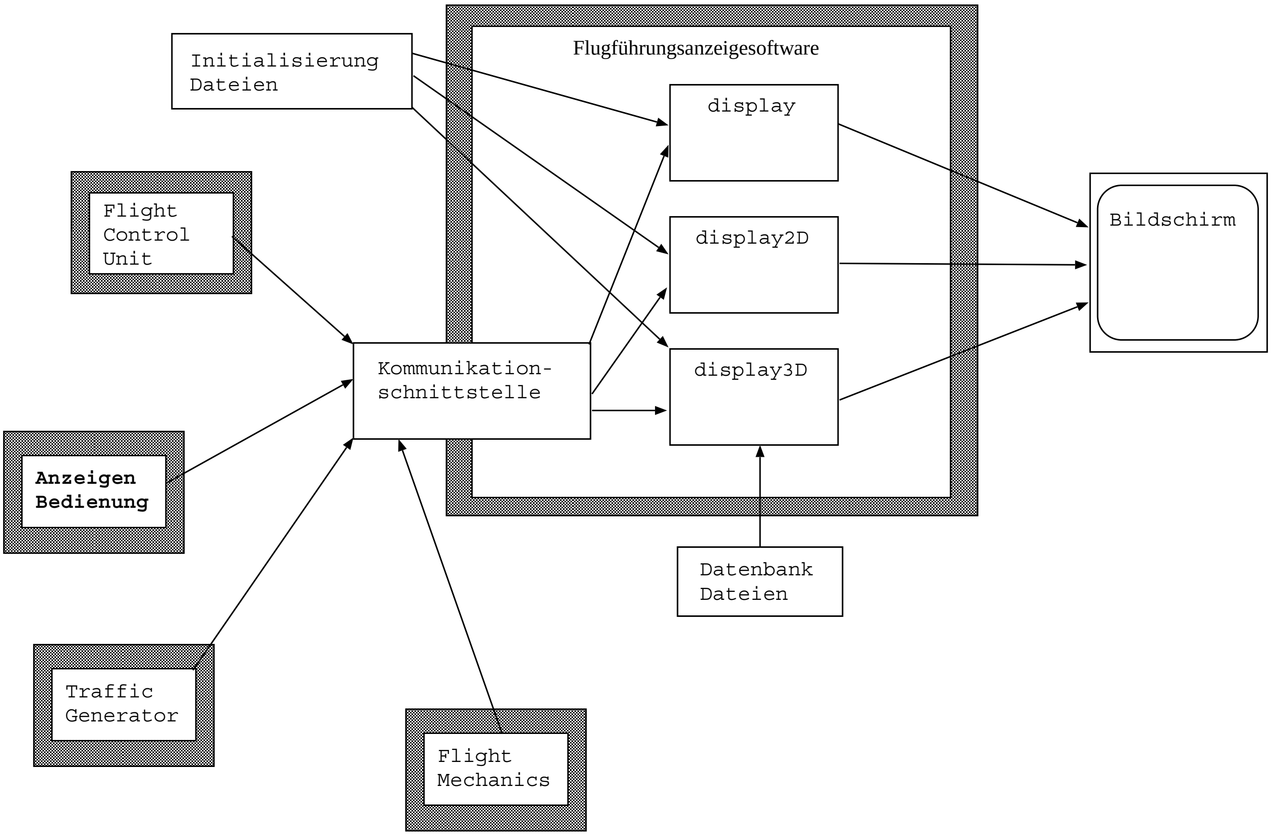


Abb. 4.15: Generierung eines Bildes

4.6 Zweidimensionale grafische Darstellung

4.6.1 Externe Werkzeuge zur Entwicklung der 2D-Elemente

Kommerzielle Anwendungen am Beispiel von VAPS

Einige Werkzeuge stehen für neue Entwürfe zur Verfügung. Eine solche Software soll die Entwicklung erleichtern. Mit ihrer Hilfe sollen sich neue Elemente schnell generieren und mit geringstem Aufwand und ohne Programmierkenntnisse animieren lassen.

Die Virtual Prototypes Application (VAPS) von Virtual Prototypes Inc. (VPI) ist das in der Luftfahrt übliche Werkzeug. VAPS ist ein menügesteuertes Tool [Ben97], das in seinem modularen Aufbau ein einfaches und schnelles Erstellen von Displayelementen und deren Animation ermöglicht.

Leider lassen sich bestimmte Anzeigeelemente nicht nachbilden [Höh97]: *Die Anzeige der Fluglage und das Zählwerkverhalten in der barometrischen Höhenskalenanzeige können zum Beispiel nicht nachgebildet werden. Auch kann man die VAPS-Objekte nicht beliebig kombinieren, da sie in einem Modul von VAPS unvorhersehbare Verhaltensweisen zeigten. Hier sei auf das Verschwinden der Symbole in den entsprechen Anzeigen im HD-PFD hingewiesen, die unvermittelt bei einem bestimmten Wert aus den Displaybereichen verschwanden.*

VAPS generiert am Ende einen Quellcode für eine Anzeigeanwendung. Wie vorher gefordert, soll diese Anzeige auch eine Darstellung im Hintergrund erlauben. Diese Anwendung muss Werte von anderen Anwendungen benutzen. Das heißt, dass man vielleicht keine Programmierkenntnisse benötigt, um die Elemente zu generieren, aber man braucht danach unbedingt Programmierkenntnisse für die Schnittstelleneinbindung und für die Darstellung des Hintergrunds.

Lösungsansatz mit Hilfe einer Datenbank

Man könnte Skalen und Elemente auch realisieren, indem man diese als Modelle in einer Datenbank ablegt. Als Beispiel bietet sich, wegen der vergleichsweise geringen Komplexität, die Höhenskala an. Sie bewegt sich häufig nur in einer Richtung. Neben den grafischen Elementen muss man in der Datenbank jedoch auch die der Bewegung zugrunde liegende Logik und die verwendeten externen Variablen ablegen. Die Vertikalbewegung der Skala lässt sich durch $moveY = baroAltitude$ beschreiben, wobei die Variable $baroAltitude$ über die Kommunikation eingelesen wird.

Die Anstellwinkelskala hingegen ist komplexer, weil sich diese Skala in zwei Richtungen bewegt. Die Bewegung ist vertikal für den Anstellewinkel, aber zusätzlich ist die Rotation um die z-Achse des Flugzeuges zu berücksichtigen.

Die Geschwindigkeitskala scheint so *einfach* wie eine Höhenskala zu sein. Aber eine Geschwindigkeitsskala verfügt über einige Informationen, die nicht ständig dargestellt werden. Die *Decision Speed* oder $V1$ wird z.B. nur während des Starts dargestellt und sogar nur solange das Hauptfahrwerk noch auf dem Boden ist. In der Skala gibt es viele andere kleine Symbole, die über eine eigene Logik verfügen. Es stellt sich daher die Frage, wie eine Datenbank aussehen müsste, um die Implementierung all dieser Symbole und ihrer Logik zu ermöglichen.

Ein solches System wäre natürlich sehr flexibel, weil zur Implementierung eines neuen Befehls nur die Datenbank und nicht die Software geändert werden müsste. Wenn man eine neue Darstellung einer Skala ausprobieren möchte, würde man nur ein neues Datenbankmodell erzeugen. Um die Generierung eines neuen Modells zu vereinfachen, müsste man allerdings ein zusätzliches Tool entwickeln.

Eine Datenbanklösung führt noch zu einem weiteren Problem: Die Software braucht Werte für die Dynamik der Elemente. Die Software berechnet diese Werte nicht selbst, sondern liest sie mit Hilfe einer Schnittstelle. Die Bedienung der Schnittstelle muss also auch in dem Modell gespeichert sein.

Insgesamt sprechen die Entwicklung eines neuen Tools, die Komplexität des Datenbankmodells und die Komplexität der Schnittstelle innerhalb der Software gegen diese Lösung.

4.6.2 Objektorientierte Implementierung

Der vorherige Abschnitt hat die Grenze einer getrennten Anwendung aufgezeigt. Als alternative Lösung ist jedes 2D-Element zu programmieren. Die beschriebenen Anforderungen zeigen, dass es für ein Element mehrere Versionen geben muss.

Eine objektorientierte Lösung hat hierbei den Vorteil, durch Benutzung von Vererbung[3], effizient verschiedene Versionen eines Elements implementieren zu können.

Auch wenn in der Regel jedes Projekt einen eigenen, vollständigen Symbolsatz besitzt (siehe Abb. 4.16), sollen dennoch auch gemischte Symbolsätze aus vorhandenen Elementen einfach erzeugt werden können. In der vorliegenden Software ist dies sogar ohne erneutes Compilieren möglich.

Die Verbindung der 2D-Elemente mit

---

[3] Mechanismus der Objektorientierung, durch den gemeinsame Eigenschaften mehrerer Klassen zu einer Basisklasse zusammengefasst werden können. Eine Klasse ist die Beschreibung von Gegenständen mit gleichen Eigenschaften [Ach97].

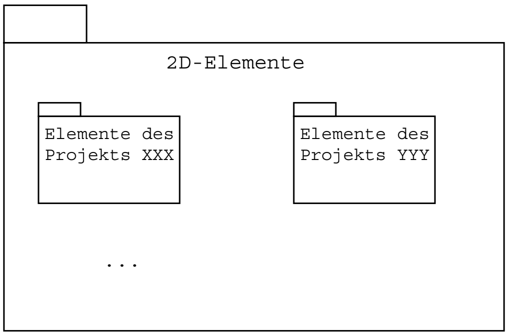


Abb. 4.16: Projektelemente

- ihrem Verwalter (*display2D* siehe Kapitel 4.6.7),
- den voreingestellten Werten, die aus einer Datei gelesen werden (mit iniFiles siehe Kapitel 4.6.9),
- dem Bediener: er definiert ein Format, er ändert die Größe des Fensters oder die Größe und die Position eines Elementes, er schaltet das Element ein- oder aus, und kann das Ganze re-initialiseren, damit es möglich ist, zum Beispiel andere Symbole zu benutzen (siehe Kapitel 4.6.8),
- dem Piloten, Autopiloten oder Flugregler, der die Lage (und andere dynamische Zustände) des Flugzeugs ändert

wird in Abbildung 4.17 gezeigt.

4.6.3 Element2D: Basisklasse der 2D-Elemente

Jedes Element hat eine Position in der Anzeige. Die Position ist während der Initialisierung des 2D-Elements fest definiert (siehe Kapitel 4.6.9), und nur ein Offset kann ein Element noch im Fenster bewegen. Dieser Offset ist in der Klasse *element2D* gespeichert. Die Werte des Offset werden der Anzeige von einer anderen Software geliefert (siehe Kapitel 4.6.8).

Ein anderer externer Einfluss betrifft die Änderung der Größe und der Position der Elemente. Weder die Größe noch die Position eines Elements sind fest implementiert, sondern hängen unter anderem von der Größe des Fensters ab: Wenn das Fenster größer wird, werden die Elemente im gleichen Verhältnis vergrößert und entsprechen neu positioniert.

Es gibt mehrere Arten von Verschiebung:

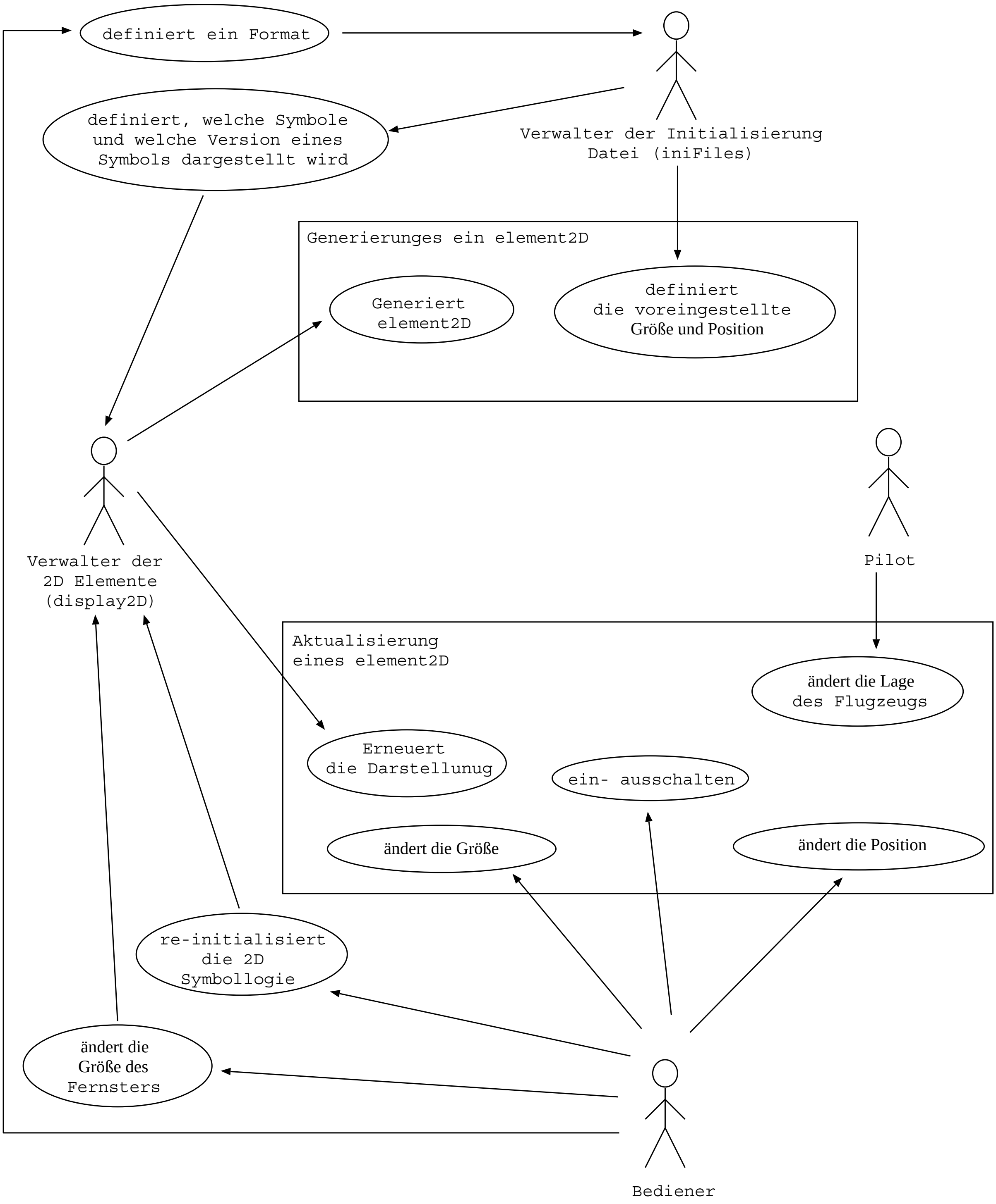


Abb. 4.17: Use Case Diagramm der 2D-Elemente

- Ein Fenster kann in verschiedene Kanäle, Software-*Channel* (siehe Seite 45), unterteilt sein. Ein solcher Kanal ist oft so groß wie das Fenster selbst, aber für einen bestimmten Fall (siehe Kapitel 4.4.12) müssen mehrere Kanäle in einem Fenster dargestellt werden, und diese Kanäle sind nicht so groß wie das Fenster selbst. Daher muss es möglich sein, eine Kanalverschiebung, $channelOrigin$ zu benutzen. Der Verwalter der 2D-Elementen (siehe Kapitel 4.6.7) liefert diese Werte.

- Die Verschiebung jedes Elements eines Kanals geschieht von einem Nullpunkt aus, der in der Mitte des Kanals liegt.
  - Der vordefinierte Offset $iniOffset$ hängt vom Projekt ab: dieser Wert wird vom Verwalter der Initialisierungdateien geliefert (siehe Kapitel 4.6.9).
  - Der Laufzeit-Offset $xFormsOffset$ hängt von einer externen Anwendung, der Software für die Bedienung des *Instructors* (siehe Kapitel 4.6.8) ab.
  - Eine gegebenenfalls vorhandene stereoskopische Parallaxe $paraAct$ hängt von einer externen Anwendung ab, die den entsprechenden Paramater liefert.

Die Größe eines Elements hängt ab von

- Dem vordefinierten Wert $iniSize$. Weil es eine Abhängigkeit von der Auflösung und des Seitenverhältnisses des Bildschirms geben kann, wird jede Größe ($size_x$, $size_y$) eines Elements gleichzeitig mit den beiden Größen ($channelSize_x$,$channelSize_y$) des Kanals skaliert.
- Die dargestellte Größe $xFormsSize$ hängt wie der Offset von der gleichen externen Anwendung ab.

Die Größe des Kanals, $channelSize$ und die Position des Kanals $channelOrigin$ gelten für jedes Element, daher werden die beiden Felder als Klassenparameter definiert. So ist es möglich, diese Werte einmal für jedes Element mit einer Klassenmethode, *preDraw*, zu berechnen.

Die Orientierung der Anzeige wird einmal für alle Symbole gespeichert. Diese Orientierung ist die gleiche für jede Anzeige und für jedes Element innerhalb einer Anzeige. Dies spricht für einen Klassenparameter. Weil dieser Wert sich zur Laufzeit nicht ändert, wird er während der Initialisierung gesetzt.

Alle Elemente, die blinken sollen, müssen gleichzeitig aufleuchten bzw. erlöschen. Dies wird ermöglicht durch einen statischen Parameter, der von der Rechneruhrzeit abhängt. Dieser Status ist dann für alle Unterklassen der gleiche und nur einmal gespeichert. Die Aktualisierung wird in einer Klassenmethode, der Methode *preDraw*, aufgerufen. Die

Methode *predraw* soll nur die allgemeinen Werte aktualisieren und ihre Änderung soll für alle Elemente einer Anzeige gleichzeitig gültig sein. Diese Methode wird einmal pro Frame für jedes Fenster vom Verwalter der 2D-Elemente aufgerufen.

Die Klasse *element2D* soll unter anderem eine *drawAll* Methode besitzen, um ein gesamtes Element darzustellen. Jede Unterklasse muss eine Definition dieser Methode besitzen, eine allgemeine Definition ist nicht möglich, deshalb ist diese Methode *pur virtual*. Die Klasse *element2D* ist aus diesem Grund eine abstrakte Klasse.

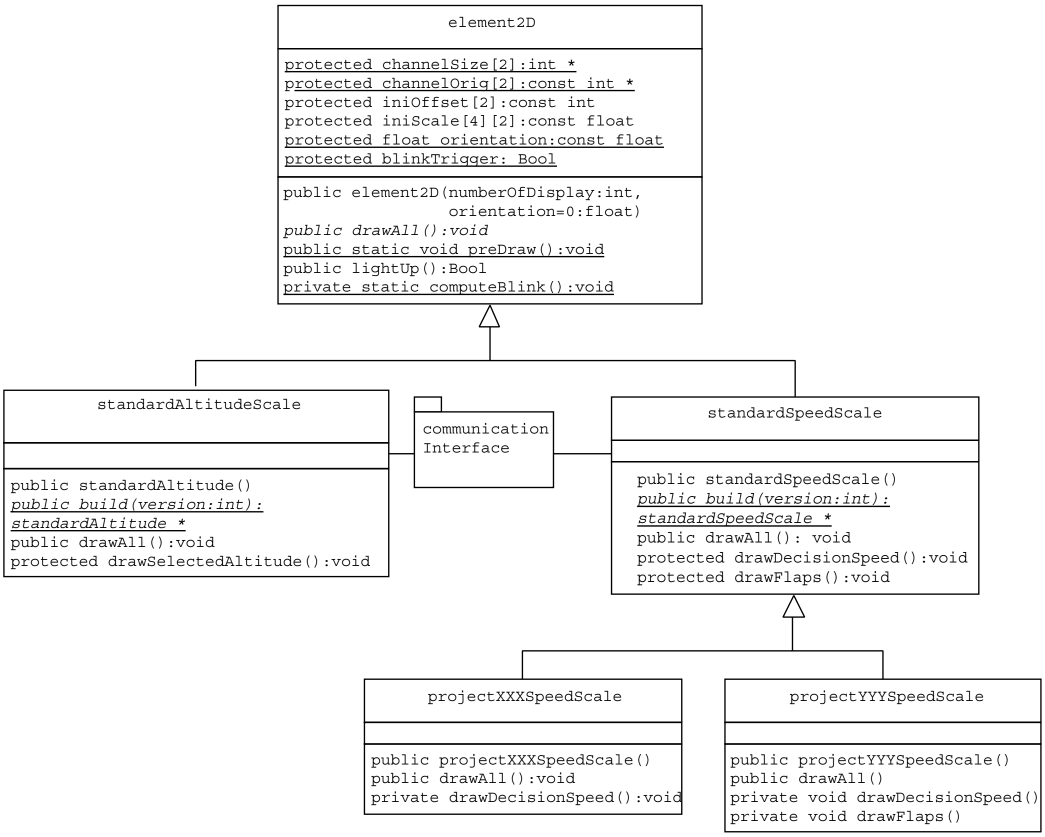


Abb. 4.18: Klassendiagramm der Klasse *element2D*

4.6.4 Standardelemente

Für jede Art von 2D-Elementen, wie Geschwindigkeitskala, Höhenskala oder Kursskala wird eine Standardklasse, z.B. *standardSpeedScale*, definiert. Diese Klasse erbt die eigenschaften von *element2D*. Sie benutzt eine Schnittstelle (siehe Kapitel 4.8), um Werte, die nicht von der Software erzeugt werden, zu lesen.

Für die Darstellung der Elemente werden einfache Grafikbefehle benutzt. Die Elemente benötigen keine weiteren Informationen und liegen in einer Ebene. Daher kann man zweidimensionale grafische Befehle und Koordinaten benutzen. Eine solche Darstellung setzt sich aus Punkten, Linien und Polygonen zusammen (siehe Kapitel 4.4.1).

Für ein ganzes Element werden mehrere kleine Symbole benutzt. Zum Beispiel sollen bei der Geschwindigkeitskala unter anderem die *Decision Speed*, *Flaps Speed*, *Selected Speed* und die *Trend Speed* dargestellt werden. Aber wie schon erläutert, werden diese Elemente nicht dauernd angezeigt. Für jedes einzelne Symbol existiert eine Methode, um das Symbol abzubilden. Eine Hauptmethode, *drawAll*, kann alle diese untergeordneten Methoden aufrufen.

drawAll trägt auch die Verantwortung dafür, anhand der Informationen der Bedienung des Instruktors (siehe Kapitel 4.6.8) das ganze Element ein- oder auszuschalten. Dies wird mit Hilfe der Schnittstelle, der Software und der Bedienungsanwendung der Flugführungsanzeige möglich. Die Methode *drawAll* wertet jedoch nicht die interne Logik eines 2D-Elementes aus, um die einzelnen Symbole ein- oder auszuschalten.

4.6.5 Projektspezifische Elemente

Wenn eine neue Darstellung oder eine neue Logik nötig ist, muss man eine neue Klasse von der Standardklasse ableiten. Dort können neue Darstellungsmethoden und neue Berechnungen eingefügt oder die Standarddefinitionen der Basisklasse ersetzt werden (siehe Abb. 4.18).

Jede Methode, die die einzelnen Symbole einer Skala neu berechnet und/oder neu zeichnet, wird die Standardversion überschreiben. Murray hat die Performance von Methoden verglichen [Mur93]. Er zeigt, dass die virtuellen Methoden für die Performance die zeitaufwendigsten sind. Trotzdem schreibt er, dass man auf die virtuelle Methode nicht vollständig verzichten kann, da es anderfalls Konzeptprobleme gibt. Weiterhin ist Meyer’s Standpunkt [Mey97], dass die C++ Konvention inkonsistent ist: Die Bindungen in C++ sind defaultmäßig statisch, und daher stört C++ die objektorientierte Softwareentwicklung. Es widerspricht der Wiederbenutzbarkeit. Auf der anderen Seite kosten die virtuellen Methoden zur Zeit in C++ mehr Rechenaufwand, um diese dynamische Bindung zu lösen.

Als Kompromiss ist statt jeder dieser Methoden nur die Hauptmethode *drawAll* virtuell. Diese Hauptmethode ist in der Unterklasse gelegentlich ganz genau so wie die der Standard-Klasse, dann werden die *lokalen* Methoden aufgerufen. Die lokalen Methoden sind die Methoden der Unterklasse, falls sie existieren. Wenn es keine Überschreibung durch eine abgeleitete Klasse gibt, werden die Methoden der Standardklasse aufgerufen.

4.6.6 Generierung von projektspezifischen Elementen

Wie schon erläutert, kann jedes Element beliebige Darstellungs- oder Logikversionen enthalten. Eine Datei soll beschreiben, welche Art von Darstellung bzw. Logik eines Elements der Benutzer möchte. Abhängig von der Version, die in der Datei steht, möchte man, dass sich zum Beispiel die Geschwindigkeitskala automatisch generiert. Wenn man Zeiger zum Beispiel auf *standardSpeedScale* benutzt, braucht man nicht zu wissen, welche Version der SpeedScale erzeugt wird. Diese *standardSpeedScale* verhält sich dann wie eine *Black-box* für den Rest der Anwendung. So ist es möglich, die verschiedenen Versionen eines Elementes schnell auszutauschen.

Um das Objekt zu generieren, muss der Verwalter der zweidimensionalen Elemente für jede Art von Element ein Kommando aufrufen. Der Verwalter kennt nur das Standardelement, so werden die zerlegbaren und stetigen Kriterien unterstützt. Das Standardelement muss mit einer Logik feststellen, welche Projektversion generiert wird. Die Standard-Version ist natürlich auch eine Version und kann sich selbst generieren. Die Abbildung 4.19 zeigt zum Beispiel wie eine spezifische Geschwindigkeitskala generiert wird. Der 2D-Verwalter wird für jedes neue Bild bis der Verwalter eine neue Version des Elementes braucht, die Methode *drawAll* aufrufen.

Für die Realisierung würde man einen neuen *operator new* schreiben. Der *operator new* soll die erwartete Version selbst berücksichtigen. Er hat eine interne Logik, um diese erwartete Version zu definieren. Diese Anforderung spricht für die Stetigkeit [Mey90]: Eine neue Version der *speedScale* erfordert eine neue Klasse und die Klasse *standardSpeedScale* muss erweitert werden. Aber die Klasse, die eine *speedScale* benutzen soll, wird nicht wieder kompiliert. Die *standardspeedScale* bleibt eine Black-box. Sie ist die Schnittstelle der Software bezüglich einer *speedScale*, auch wenn die gewünschte Instanz eine Unterklasse ist.

Ein neuer *operator new* würde wie folgt aussehen:

```
void *
standardSpeedScale::operator new(size_t sizeT){
   ...
   switch (speedScaleVersion){
      case STANDARD: return ::new standardSpeedScale();
```

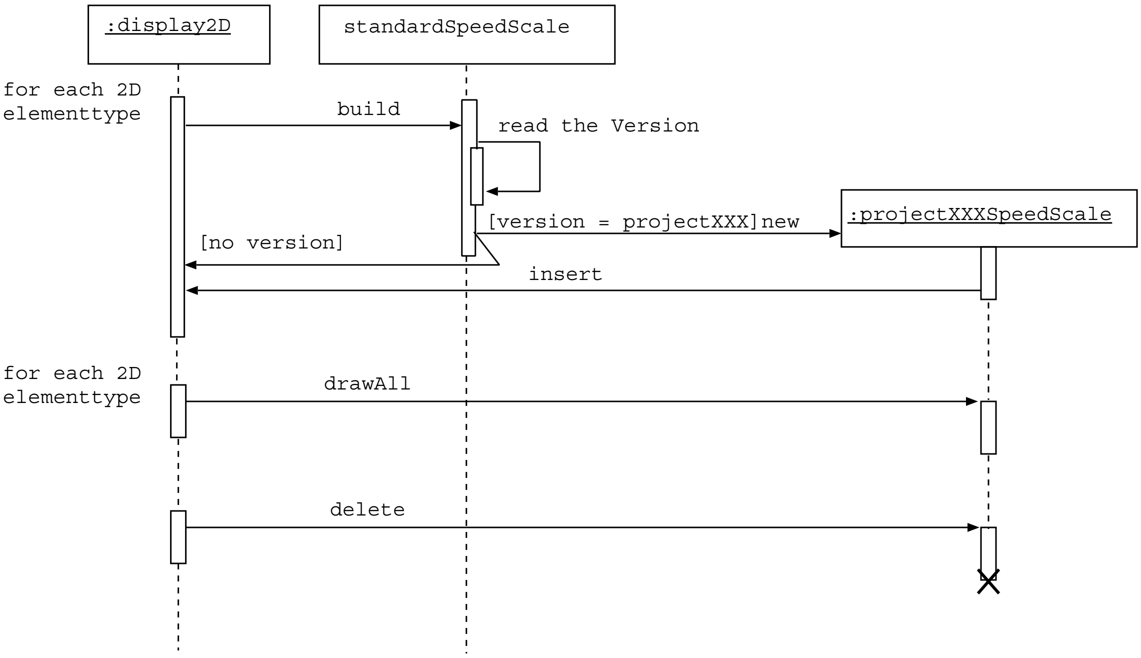


Abb. 4.19: Sequenzdiagramm: Generierung und Benutzung einer Geschwindigkeitsskala

```
        case PROJECTXXX: return ::new projectXXXSpeedScale();
        case PROJECTYYY: return ::new projectYYYSpeedScale();
    }
}
```

Die Überschreibung des *operator new* ist leider unmöglich. Wenn ein *operator new* aufgerufen wird, wird erstens der Platz für das neue Objekt im Speicher des Rechners reserviert. Dann wird der Konstruktor des neuen Objekts aufgerufen. Der Konstruktor soll unter anderem die dynamische Bindung, *dynamic Binding*, der virtuellen Methode vorbereiten. In diesem Fall, wenn man ein *projectXXXSpeedScale* erwartet, wird der new Operator von *standardSpeedScale* eine neue Instanz der Klasse *projectXXXSpeedScale* generieren. Die virtuellen Methoden für die Klasse *projectXXXSpeedScale* sind am Anfang richtig vorbereitet worden, weil der Konstruktor von *projektXXXSpeedScale* aufgerufen wurde. Aber der *operator new* von *standardSpeedScale* wird dann den Konstruktor von *standardSpeedScale* aufrufen. Dort werden die virtuellen Methoden als *standardSpeedScale* für die dynamische Bindung vorbereitet, die erste Vorbereitung wird also hier mit den falschen Werten überschrieben. Ein Aufruf einer virtuellen Methode wie *drawAll*, wird nicht die *projectXXXSpeedScale* Version, sondern wird immer die Standardversion benutzen, obwohl *drawAll* virtuell ist.

Als Lösung wird eine statische *build*-Methode statt des Operators new verwendet. Diese *build*-Methode muss eine Klassenmethode sein, weil sie eine Instanz der Klasse Standardelement oder eine derer Unterklassen generiert. Es ist also nicht möglich, eine normale Methode (oder Objekt-Methode) zu benutzen. Die Implementierung dieser *build*-Methode ist genau dieselbe, wie der *operator new* der *StandardSpeedScale*, der vorher beschrieben wurde.

4.6.7 Verwaltung der 2D-Elemente mit der Klasse Display2D

Jedes Element ist eine Instanz entweder eines Standardelements oder eine Unterklasse dieser Klasse. Jedes Standardelement erbt von der Klasse *Element2D*. Es ist also möglich, eine heterogene verkettete Liste zu generieren. Eine heterogene verkettete Liste ist eine verkettete Liste von Objekten, die sich alle von einer Basisklasse ableiten, aber jede Instanz dieser verketteten Liste kann eine Instanz irgendeiner Unterklasse[4]sein. Wenn die Basisklasse eine virtuelle Methode hat, wird jedes Objekt dieser verketteten Liste sich anders verhalten. So kann man für jede Instanz der verketteten Liste eine virtuelle Methode aufrufen, ohne Berücksichtigung des Typs der Instanz. Zum Beispiel ist es möglich, für eine verkettete Liste von 2D-Elementen eine einfache Schleife vom Anfang der verketteten Liste bis an ihr Ende zu benutzen, welche die Darstellungsmethode der Objekte aufrufen wird.

Eine Liste von 2D-Elementen wird in einer *display2D*-Klasse generiert. Der Verwalter der Initialisierungdateien (siehe Kapitel 4.6.9) wird mit Hilfe einer Formatdatei beschreiben, welches Element für eine Anzeige dargestellt wird. Der Verwalter der Initialisierungdateien wird mit Hilfe einer anderen Datei die voreingestellte Größe des Fensters definieren. Diese Werte werden weiter zur Klasse *element2D* gegeben.

Nach der Initialisierung ruft *display2D* für jeden Frame einmal *preDraw* und alle *drawAll* der Elemente auf (siehe Kapitel 4.6.3).

display2D ist auch die Schnittstelle zwischen den 2D-Elementen und dem Rest der Software. Wie später erläutert wird, existiert für die Cavok Flugführungsanzeige auch eine 3D-Darstellung.

Die 2D-Elemente sind wie auf einer Fensterscheibe, *pfChannel*, dargestellt. Zum Beispiel soll *display2D* diese Fensterscheibe immer an die Größe des Software-Fensters anpassen.

---

[4] Ein Objekt einer abgeleiteten Klasse kann wie ein Objekt seiner Basisklasse behandelt werden, wenn es über Zeiger oder Referenzen manipuliert wird [Str00]. Umgekehrt kann eine Basisklasse *nicht* mittels Zeigern auf abgeleitete Klassen manipuliert werden.

display2D muss auch den Stereoaspekt verwalten. Für eine Stereodarstellung benötigt die Anzeige zwei Fensterscheiben, eine für das linke Auge und eine zweite für das rechte.

Die Abbildung 4.20 zeigt die Standardabbildung *display2D* mit der Verbindung zu den wichtigsten Klassen.

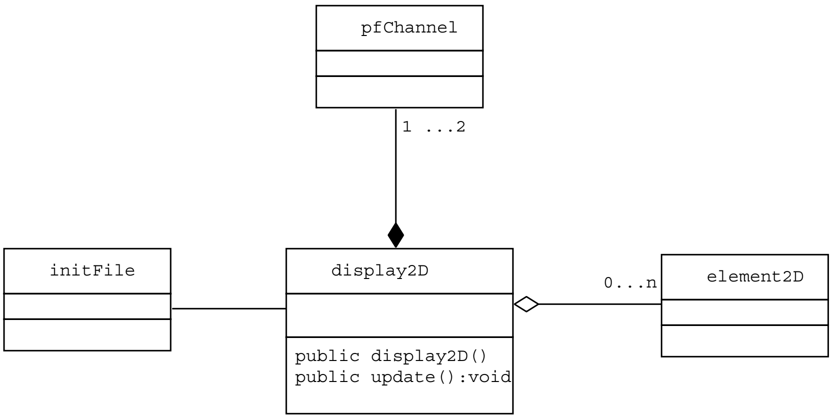


Abb. 4.20: Klassendiagramm der Klasse *display2D*

4.6.8 Bedienung der Software

Diese Software ist eine Mensch-Maschine-Schnittstelle für Piloten, und sie soll nicht nur im Flugsimulator funktionieren, sondern auch in echten Flugzeugen, wie es zum Beispiel in einer Flugversuchskampagne 1999 [Cav00] geschah. Dazu ist es notwendig, die Bedienung von der Darstellungssoftware zu trennen. Dies ermöglicht die Bedienung von unabhängigen Fenstern. Die Anzeigeanwendung soll unter anderem die Informationen mit Hilfe eines Netzwerks lesen. Die Entwicklung der Netzwerkschnittstelle ist für eine Forschungsanwendung komplex und wird in Kapitel 4.8 erläutert.

Die Frage hier ist zu wissen, ob alle Werte der Konfiguration über das Netzwerk geschickt werden müssen. Ohne näher darauf einzugehen, kann man absehen, dass zu viele unnötige Informationen das Netzwerk belasten würden. Es wirkt sich negativ auf die Gesamtleistung einer Simulation und natürlich im Endeffekt auch auf die Leistung der Flugführungsanzeigeanwendung aus. Andererseits möchte man so viel Flexibilität wie möglich. Das heißt, man möchte nur eine einzige Softwareversion, die flexibel genug ist, um zum Beispiel verschiedene Versionen eines 2D-Elementes anzuzeigen, ohne dass man die Software wieder kompilieren muss.

Die Versionen der Elemente ändern sich nicht oft. Deshalb ist es sinnvoll, dass die Versionen der Elemente in einer Datei gespeichert sind. Diese Datei kann man während der Laufzeit ändern, und braucht dann nur ein Signal zu senden. Dieses Signal bedeutet, dass die Anzeigesoftware 2D-Elemente ändern soll.

Mit Hilfe einer *initFile*-Klasse (siehe Kapitel 4.6.9) liest *display2D* eine Datei. Die Art der Skalen wird gesucht: Als Beispiel wird *speedScale* gesucht. Für jedes Element, das in dieser Datei gefunden wird, wird die Version des Elementes eingelesen. Die voreingestellte Position $iniOffset$, die voreingestellte Skalierung $iniScale$ (siehe Kapitel 4.6.3) und die Farbe der Symbole mit ihrem Hintergrund werden immer aus solchen Dateien gelesen. So ist es möglich, viele Kombinationen zu erhalten, ohne dass man die Software neu kompilieren muss.

Wie gerade erläutert, wird die Bedienung der Anzeigesoftware mit Dateien für die statischen Werte und einer getrennten Anwendung für die dynamischen Werte durchgeführt.

4.6.9 Verwaltung der Dateien zur Initialisierung mit der Klasse initFile

Eine *initFile*-Klasse wurde also entwickelt, um Initialisierungwerte aus einer Datei zu lesen.

Die Instanz einer solchen Klasse braucht den Namen der Datei. Der Pfad muss ebenfalls vorher definiert sein. Wenn die Datei nicht in diesem Pfad gefunden wird, muss die Klasse die Datei in einem anderen voreingestellten Pfad suchen. Wenn diese Datei immer noch nicht gefunden wird, muss *initFile* normalerweise eine Ausnahme mit einer Fehlermeldung erzeugen.

Die Klasse *initFile* muss Kommentare in den Dateien erlauben. Unter UNIX und Windows fängt jede Kommentarzeile mit “#” oder “;” an. Man benutzt hier die gleiche Regel.

Wenn die Datei richtig geöffnet wird und wenn sie lesbar ist, werden die Parameter gesucht. Die Klasse *iniFile* sucht Wörter. Wenn das genaue Wort nicht gefunden wird, muss wieder eine Ausnahme generiert werden. Diese Ausnahme schützt die Software gegen falsche oder unmögliche Initialisierungen. Ist das Wort gefunden, erwartet diese Klasse, dass die Initialisierungswerte hinter einem “=” stehen. Die Initialisierung wird mit dem nächsten Wert durchgeführt.

Die Initialisierung wird eigentlich mit *read*-Methoden durchgeführt. Mit Überladung sind die Methoden *read* für jeden konventionellen Typ (*char, int, float, double...*) definiert. Es muss auch eine *string*-Lesemethode geben, wobei die Klasse ein Feld von Buchstaben erwartet und maximal $n-1$ Buchstaben liest. Im letzten Buchstaben wird ein *Ende von String* eingefügt. Es schützt die Software vor einer zu großen Kette von Buchstaben.

Die *read*-Methode sucht zunächst das Stichwort und übergibt den entsprechenden Parameter. Eine weitere Methode *read* ohne Stichwort kann an diesem Punkt weiter lesen.

4.6.10 Darstellungen im Hintergrund der 2D-Elemente

Heutzutage besitzen die Navigationsanzeigen als Hintergrund der zweidimensionalen Elemente folgende zweidimensionale Darstellung: das Wetterradarecho oder ein Teil des Geländes, wenn dieses eine Gefahr darstellt. Kapitel 4.7 erläutert unter anderem eine dreidimensionale Geländedarstellung. Vorher wird noch eine kurze Beschreibung einiger zweidimensionaler Hintergründe und ein Problem bezüglich der zweidimensionalen Elemente und ihres Hintergrundes erklärt.

Wetterradardarstellung

Das Dokument ARINC 708A-3 [ari99] beschreibt die Daten, die von einem Wetterradar geschickt werden. Das Wetterradarecho wird mit verschiedenen Kreissegmenten gebildet, um die Wolken vor dem Flugzeug zwei dimensional darzustellen. Ein solches Kreissegment wird durch einen Winkelbereich (jeweils $0.0879°$ breit) und einen Abstandsbereich definiert. Es ist in 512 radiale Segmente unterteilt, denen jeweils eine Farbe zugeordnet werden kann. Die Informationen, die zur Anzeige gelangen, sind unter anderem:

- der Tilt, also der Nickwinkel des Wetterradars
- der Index des Winkelbereichs (In Abb. 4.21: α). Der Winkelbereich mit dem Index 0 steht direkt vor dem Flugzeug; die Bereiche links vom Flugzeug haben negative Indizes, die Bereiche rechts positive Indizes.
- ein Abstandsbereich oder Skalenbereich
- ein Wert, der jedem der 512 radialen Segmente eine von sieben verschiedenen Farben zuweist (siehe Abb. 4.21).

Andere Darstellungen

Außer der Wetterradardarstellung auf einem *Navigation Display* kann man andere 2D-Bilder auf einem MFD[5] darstellen. Die Darstellung mit 2D-Elementen und einem importierten Bild im Hintergrund würde zum Beispiel eine Darstellung der Approach Chart oder

---

[5] Multi Function Display

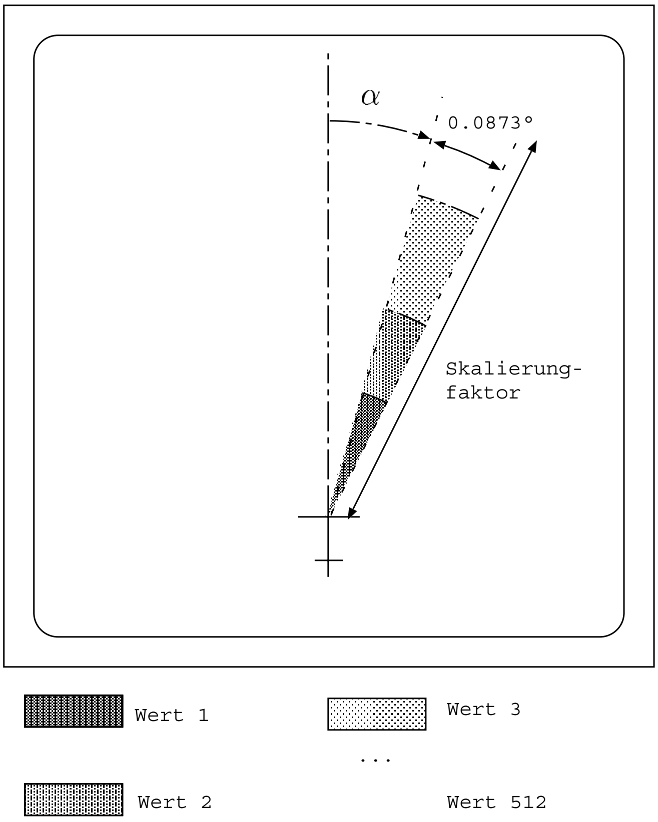


Abb. 4.21: Wetterradar Daten

der Checkliste ermöglichen. Ein anderes Beispiel ist ein Satellitenbild, das die Wetterlage über einem größeren Gebiet zeigt (siehe hier die Studie in [Fol98]).

4.6.11 Interferenz der 2D-Elemente mit dem Hintergrund

Wie schon beschrieben, werden die 2D-Elemente wie auf einer Fensterscheibe dargestellt. Hinter dieser Fensterscheibe wird etwas anderes dargestellt. Eine Interferenz (fehlender Kontrast) ist zwischen den 2D-Elementen und dem Hintergrund möglich (erste Darstellung der Abbildung 4.22). Dieses Kapitel wird zwei Lösungen erläutern, sie werden benutzt, um Interferenzen zu vermeiden.

Transparenz

Die Elemente seien auf einer virtuellen Fensterscheibe dargestellt. Hierbei ist es möglich, die Transparenz der Fensterscheibe interaktiv zu variieren. Abbildung 4.22 zeigt als Beispiel die Höhenskala einer perspektivischen Anzeige mit einem Hintergrund. Die Ziffern der Skala sind im zweiten oder dritten Bild besser lesbar. Wie der Skalierungsfaktor und die Position der Elemente soll der Alpha-Blending-Faktor dynamisch sein. So ist es möglich, die genauen Werte der Transparenz zu definieren.

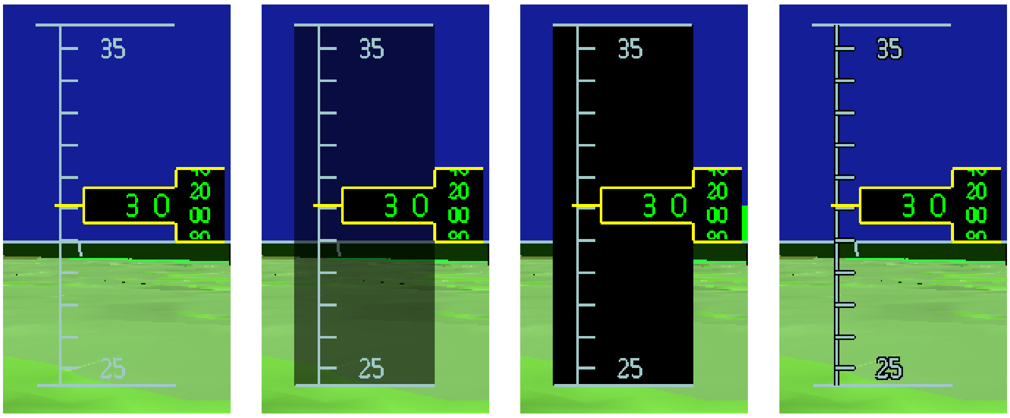


Abb. 4.22: Höheskala: verschiedene Transparenzfaktoren und Konturlinien

Eine Skala, die nicht ganz durchsichtig ist, wird im Vordergrund mit einer dahinter liegenden Alpha-Blending-Fläche dargestellt. Alles, was hinter dieser Fläche liegt, wird seine Originalfarbe ändern: Die Farbe wird dunkler werden, wenn zum Beispiel die Alpha-Blending-Farbe schwarz ist.

Eine Skala hat üblicherweise helle Farben. Wenn die Skala nicht ganz durchsichtig ist, gibt es normalerweise keinen Konflikt, und die Informationen auf der Skala sollten immer lesbar sein.

Konturlinie

Dennoch gibt es einige Skalen die ganz durchsichtig sein müssen. Zum Beispiel ist eine Alpha-Blending-Nickwinkelskala auf einem HDPFD nicht möglich. Stattdessen werden Linien mit Konturlinien, die andere Farben haben, dargestellt. Normalerweise kann man eine schwarze Konturlinie benutzen. Für den Fall, dass der Hintergrund schwarz ist, ist diese Konturlinie nicht sichtbar, aber die eigentliche Linie ist sichtbar. Wenn der Hintergrund die gleiche Farbe wie die eigentliche Linie hat, ist die eigentliche Linie nur mit Hilfe der Konturlinien sichtbar (siehe Abb. 4.22).

Wie werden diese Konturlinien erzeugt? Als erste einfache Lösung könnte man eine dicke schwarze Linie und dann darüber eine dünnere Linie mit einer anderen Farbe darstellen. Diese Lösung ist die einfachste und die schnellste. Die grafische Bibliothek, die zur Verfügung steht, zeichnet nur entweder in X- oder in Y-Richtung und nicht in beide Richtungen gleichzeitig eine Linie fetter. Ein Text wird dann nie eine saubere Konturlinie bekommen. Es kann noch schlimmer werden, wenn der Text rotiert (siehe Abb. 4.23 links).

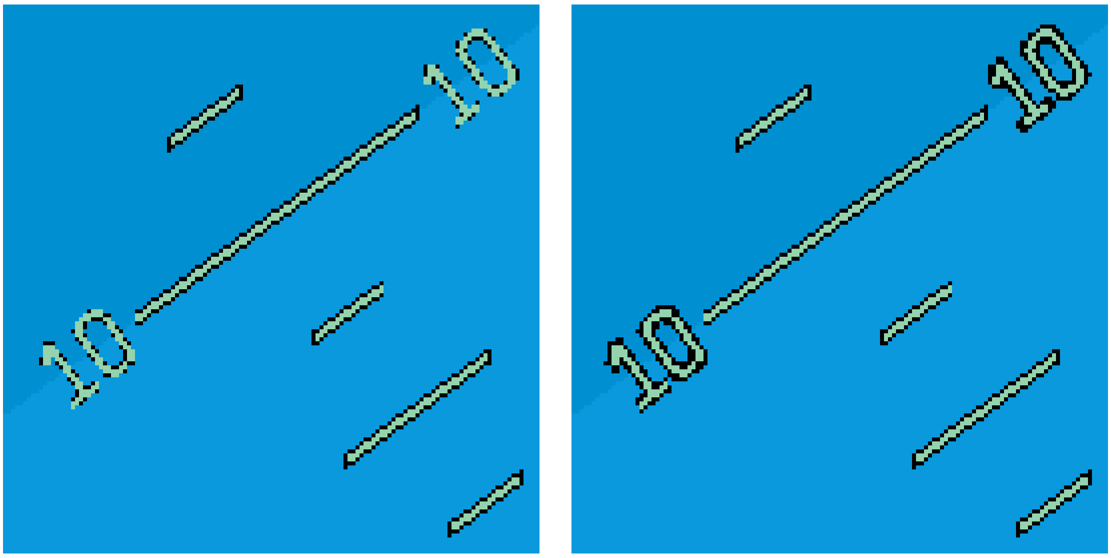


Abb. 4.23: Konturlinien

Als Lösung für einen Text muss man statt fetter Buchstaben die Buchstaben fünfmal zeichnen. Die Buchstaben werden in schwarz mit einem Pixel nach links, rechts, unten und dann oben geschoben. Als letztes werden die Buchstaben ohne Verschiebung mit ihrer eigenen Farbe gezeichnet (siehe Abb. 4.23 rechts).

4.7 Dreidimensionale grafische Darstellung

Es kann für einige Anzeigen hinter den 2D-Elementen auch eine andere Darstellung als die 2D-Wetterradar-Abbildung geben. Hinter den 2D-Elementen, werden Gelände, Fremdflugzeuge, der Prädiktor [Pur99], oder ein Flugkorridor [Mul99],[The97] dargestellt. In diesem Kapitel wird die dreidimensionale Geländedarstellung in Hinblick auf die Anforderungen der Flugführungsanzeigen erläutert. Danach wird die Realisierung der 3D-Darstellung erklärt.

4.7.1 Hintergrund

Darstellung des Hintergrunds bei orthogonalen Anzeigen

Wie in 2.2.2 beschrieben wurde, besitzt die othogonale Darstellung eine Sicht des Geländes von oben (Draufsicht). Eine rotierende Karte wird hier verwendet. Das heißt, das Flugzeugsymbol bewegt sich normalerweise nicht, sondern die Darstellung des Geländes.

Es wurde schon erläutert, dass Textinformationen mit Geländesymbolen verbunden sind. Die Textinformationen scheinen eine konstante Richtung zu haben. Eigentlich werden diese Textinformationen immer entgegengesetzt der Richtung der Geländerotation gedreht.

Weiterhin ist es möglich, die Zoomstufe des *Navigations Display* zu ändern. Die Skalierung des Geländes hat einen Einfluss auf den Text. Die Textinformationen müssen nicht nur eine konstante Richtung, sondern auch eine konstante Größe haben. Wenn die Größe des Textes sich mit der gesamten Zoomstufe ändern würde, könnte bei einer großen Zoomstufe der Text das Gelände verdecken. Wie bei einer Änderung der Richtung ändert sich die Skalierung des Textes umgekehrt proportional zur Zoomstufe. So bleibt die Größe des Textes immer konstant, auch wenn die Zoomstufe sich ändert (siehe Abb. 4.24).

Darstellung des Hintergrunds bei perspektivischen Anzeigen

Auf dem *Primary Flight Display* wird ein perspektivisches Gelände dargestellt. Der Augpunkt liegt im Cockpit normalerweise auf der Längsachse des Flugzeuges.

In den nächsten Unterkapiteln werden die verschiedenen Klassen der 3D-Darstellung und ihre Rollen erläutert. Weiterhin wird die Beziehung zwischen den 2D-Elementen und 3D-Elementen erläutert.

4.7.2 Einrichten der Grundfunktionalität mit der Klasse Display

Konfiguration der *Pipe*

Aktuelle Grafikrechner erlauben mehrere Arbeitsplätze. Das heißt, dass mehrere Bildschirme, mehrere Tastaturen und Maus an nur einen Rechner angeschlossen sind. Für die Benutzer scheint es, als ob es mehrere Rechner gäbe. Man spricht dabei von verschiedenen *Pipes*. Auf solchen Rechnern ist es aber auch möglich, alle Ausgaben über nur eine Eingabe zu steuern. Dieser Modus wird *Single Keyboard* genannt.

Dieser Modus hat eine wichtige Bedeutung. Es ist nämlich nicht möglich, mit nur einem Projektor eine sehr große Szene darzustellen. Für die Außensicht eines Flugsimulators braucht man jedoch eine große Projektionsfläche. Im Institut für Flugsysteme und Regelungstechnik wird zum Beispiel die Außensicht auf einer Fläche von 180x40 Grad projiziert. Dazu werden drei Projektoren verwendet. Diese drei Projektoren dürfen auf keinen

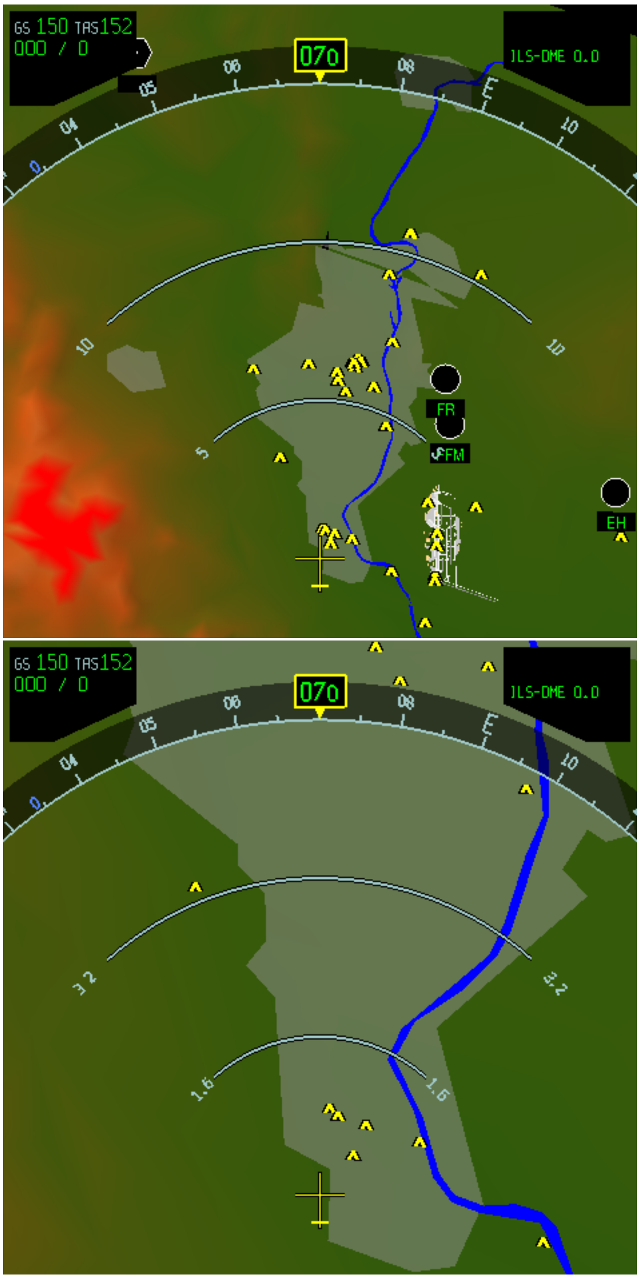


Abb. 4.24: Verschiedene Zoomstufen

Fall unabhängig arbeiten: Die Bilder müssen synchronisiert werden, damit keine Übergänge zu sehen sind. Dies ist zum Beispiel ein Grund, Rechner mit mehreren *Pipes* zu benutzen, denn dann ist es möglich, die drei Bilder zu synchronisieren. Man hat dann den Eindruck, dass es sich nur um ein einziges Bild zu handelt.

Der Vollständigkeit halber sei erwähnt, dass es möglich ist, die Flugführungsanzeige-

Anwendung zur Erzeugung einer einfachen Außensicht in einem Flugsimulator zu benutzen. Diese Möglichkeit erlaubt es, die durchsichtfähige Anzeigegeräte (wie HUD oder HMD) mit der Außensicht zu kalibrieren. Bei einer separaten Software für die Aussensicht weiss man nicht, ob die Verschiebung durch die Software oder die Hardware verursacht wird.

Aus Sicht der Software wird der *Multi-Pipe* Modus in der Klasse *display* verwaltet. Eine *Pipe* ist hier die Schnittstelle zwischen der Software und der grafischen Hardware. Die Vorbereitung einer *Mutli-Pipe*-Darstellung wird in *configPipe* (siehe Abb. 4.26) der Klasse *display* durchgeführt. Hier generiert diese Klasse ein Feld von *Pipes*. Die Zahl der Elemente ist von Anfang an festgelegt. Die Anzahl der benutzten *Pipes* hängt von der Art der Anzeige ab: Für ein HDPFD oder ND braucht man normalerweise nur eine *Pipe*, aber man braucht im Gegensatz dazu eins bis drei *Pipes* für die Außensicht einer Simulation. Folglich muss die Konfiguration der *Pipe* virtuell sein.

Der Sichtpunkt aller Projektoren für eine Außensicht ist der gleiche. Allerdings ist die Sichtrichtung unterschiedlich. Die *Pipe* 0 ist immer genau nach vorn gerichtet, die *Pipe* 1 nach links und die *Pipe* 2 nach rechts. Die Richtung der Projektoren muss mit der Richtung der Bildprojektion genau übereinstimmen. Dies wird nicht mit der Konfiguration der *Pipe* realisiert. Die Konfiguration der *Pipe* betrifft nur die Konfiguration der Schnittstelle zwischen der Hardware und der Software und wird in *configPipe* der Klasse *display* durchgeführt.

PfPipeWindow Konfiguration

Auf jeder *Pipe* wird mindestens ein Fenster pro Anzeige geöffnet. Die Methode *configPWindow* soll die Fenster einer Instanz der Klasse *PfPipeWindow* konfigurieren. Es ist nötig zu wissen, wie groß die Anzeige sein soll. Die Größe hängt auch von der Orientierung des Bildschirms ab. Bei einem Hochkantformat muss man die Werte x und y vertauschen. Die Position des Fensters hängt von der Anzeige selbst und von der Anzahl der Anzeigen ab. Das zeigt, dass die Konfigurations-Methode virtuell sein muss. Da man keine Default-Version definieren kann, ist diese Methode pur virtuell. Am Ende der Konfiguration ist das Fenster geöffnet und wird bis zum Beenden der Software geöffnet bleiben.

PfChannel Konfiguration

In einem einzelnen Fenster ist es möglich, mehrere Bereiche zu definieren. Diese Bereiche werden Software-Kanal oder *Channel* genannt. Die Breite des *Channels* entspricht fast immer der Breite des Fensters. Für eine 3D-Anzeige mit 2D-Elementen gibt es regulär einen Kanal für die 3D-Welt und einen für die 2D-Elemente. Allerdings gibt es Ausnahmen. Die Kanalanzahl hängt z.B. auch davon ab, ob die Anzeige monoskopisch oder stereoskopisch ist. Aus diesem Grund hat die Klasse *display* eine pur virtuelle Methode, *configChannel*, um die Kanäle zu konfigurieren.

Diese Methode konfiguriert die Art der Projektion, orthogonal oder perspektivisch . Der Referenzpunkt und die Orientierung der Achse werden durch die Methode *setViewMat* von der Klasse *pfChannel* definiert. Dazu wird eine $4 * 4$ Matrix definiert: Der obere linke $3 * 3$ Anteil dieser Matrix definiert eine Rotations-Matrix, die unterste Zeile der Matrix definiert die Position des Sichtpunkts. Die Rotations-Matrix soll die Welt-Koordinaten in *OpenGL Performer*-Koordinaten drehen. Die Achsen der Anwendung entsprechen der Luftfahrtnorm [fN90]:

- X ist nach vorn gerichtet,
- Y ist nach rechts gerichtet,
- und Z ist nach unten gerichtet.

Die Achsen von *OpenGL Performer* sind hingegen so definiert:

- $X_{performer}$ ist nach rechts gerichtet,
- $Y_{performer}$ ist nach vorn gerichtet,
- und $Z_{performer}$ ist nach oben gerichtet.

Eine Rotationsmatrix wandelt die Weltreferenz in die *OpenGL Performer* Referenz um.

Für den Betrachter der perspektivischen Anzeige ist die Sichtachse die X Richtung der Anwendungswelt. Die Rotationmatrix für die perspektivische Anzeige lautet also:

$$\begin{aligned}
R_{pfd} &= R_{z(-90)}.R_{y(180)} \\
\Leftrightarrow\quad R_{pfd} &= \begin{pmatrix} cos(-90) & -sin(-90) & 0 \\ sin(-90) & cos(-90) & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} cos(180) & 0 & sin(180) \\ 0 & 1 & 0 \\ -sin(180) & 0 & cos(180) \end{pmatrix} \qquad (4.12) \\
\Leftrightarrow\quad R_{pfd} &= \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}
\end{aligned}$$

Der Betrachter des ND sieht von oben auf das Flugzeug. Die Rotationmatrix sieht dann wie folgend aus:

$$\begin{aligned} R_{nd} &= R_{y(90)}.R_{z(90)} \\ \Leftrightarrow\ R_{nd} &= \begin{pmatrix} cos(90) & 0 & sin(90) \\ 0 & 1 & 0 \\ -sin(90) & 0 & cos(90) \end{pmatrix} * \begin{pmatrix} cos(90) & -sin(90) & 0 \\ sin(90) & cos(90) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ \Leftrightarrow\ R_{nd} &= \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{aligned} \tag{4.13}$$

In einem VSD sieht der Betrachter in die Richtung von Y_{world}:

$$\begin{aligned} R_{vsd} &= R_{x(180)} \\ \Leftrightarrow\ R_{vsd} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(180) & -sin(180) \\ 0 & sin(180) & cos(180) \end{pmatrix} \\ \Leftrightarrow\ R_{vsd} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \end{aligned} \tag{4.14}$$

Der Sichtpunkt liegt nicht unbedingt im Flugzeug-Referenzpunkt. Die Navigationsanzeige, in der von oben herab geschaut wird, zeigt dies deutlich. Es ist sicherzustellen, dass beispielsweise auch über dem eigenen Flugzeug fliegende Fremdflugzeuge in der Anzeige dargestellt werden. Demnach kann der Sichtpunkt dort nicht im Flugzeug-Referenzpunkt liegen, sondern muss sich oberhalb maximal möglicher Flughöhen befinden.

In der Methode *configChannel* der Klasse *display* werden die Art der Projektion, sowie die Referenzpunkte und die Orientierung definiert. Für die perspektivische Projektion wird dazu noch des *Field Of View* (siehe Kapitel 4.4.2 Seite 51) und die far und *near clipping* -Ebene definiert. Für das Field Of View ist nur ein einziger Winkel nötig, wenn die Größe des Fensters ($horizontal_g$,$vertical_g$) an den horizontalen und vertikalen Winkel ($horizontal_{FOV}$,$vertical_{FOV}$) angepasst wird (siehe Gl. 4.15). Das bedeutet, dass die Methode *configPWin* auf diesen Winkel einen Einfluss hat:

$$horizontal_g = vertical_g \frac{tan(\frac{horizontal_{FOV}}{2})}{tan(\frac{vertical_{FOV}}{2})} \tag{4.15}$$

Diese Methode ist auch eine pur virtuelle Methode. Die Werte der Konfiguration werden nur für eine bestimmte Anzeige festgelegt.

Für die stereoskopische Darstellung des HUD [Kai03] ist es wegen eines Modus der Hardware nötig, die linke und rechte Sicht untereinander zu zeichnen. Somit wird das Fenster nicht geändert, sondern nur die zwei *Channels*, die für eine stereoskopische Darstellung benutzt werden. Ihre vertikale Größe wird verkleinert. Der erste *Channel* schließt mit der Unterkante des Fensters ab und der zweite *Channel* ist genau darüber (siehe Abb. 4.25).

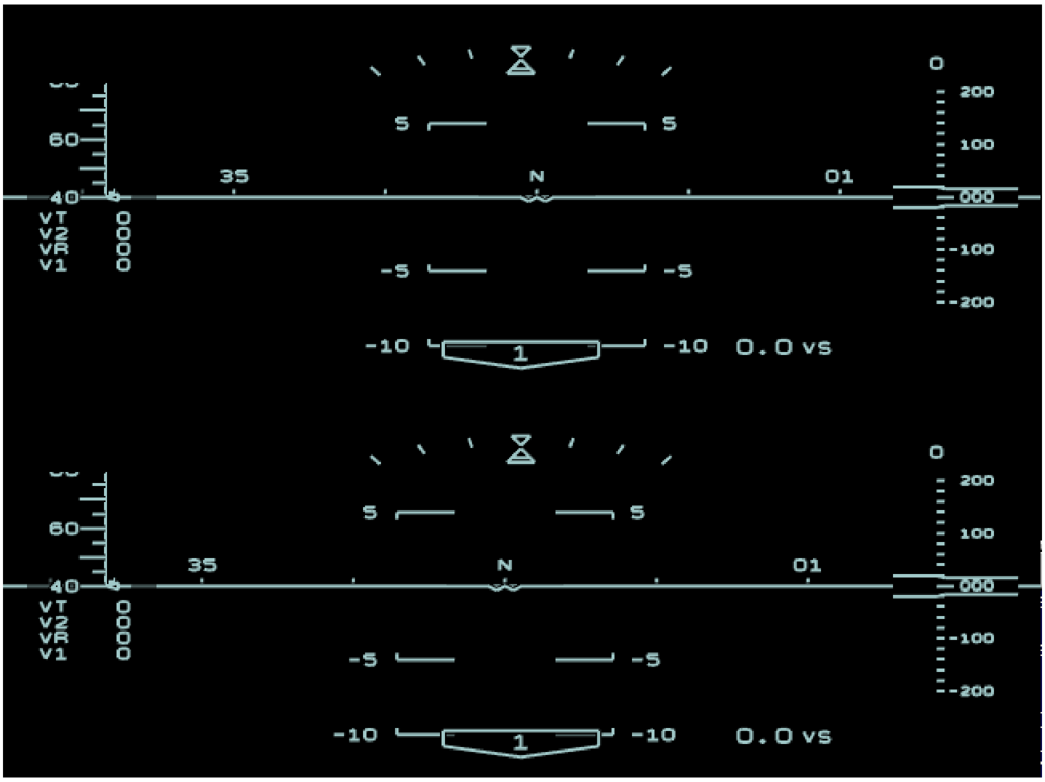


Abb. 4.25: Die Channel des stereoskopischen HUD

Die Abbildung 4.26 zeigt die Organisation der Klasse *display*.

4.7.3 Verwaltung der 2D-Elemente in der Anzeigeanwendung

Im Kapitel 4.6.7 wurde die Klasse *display2D* erläutert. Diese Klasse soll die Elemente in einer 2D-Anzeige verwalten. Deshalb soll diese Klasse von *display* erben: ein *display2D* ist eine Anzeige, die nur 2D-Elemente hat und verwaltet. Sie wird die Kanäle der Klasse *display* für die 2D-Elemente konfigurieren: Eine Parallelprojektion wird benutzt; die Referenzen sind fest definiert. Die Methode *configChannel* wird hier beschrieben. Aber weil

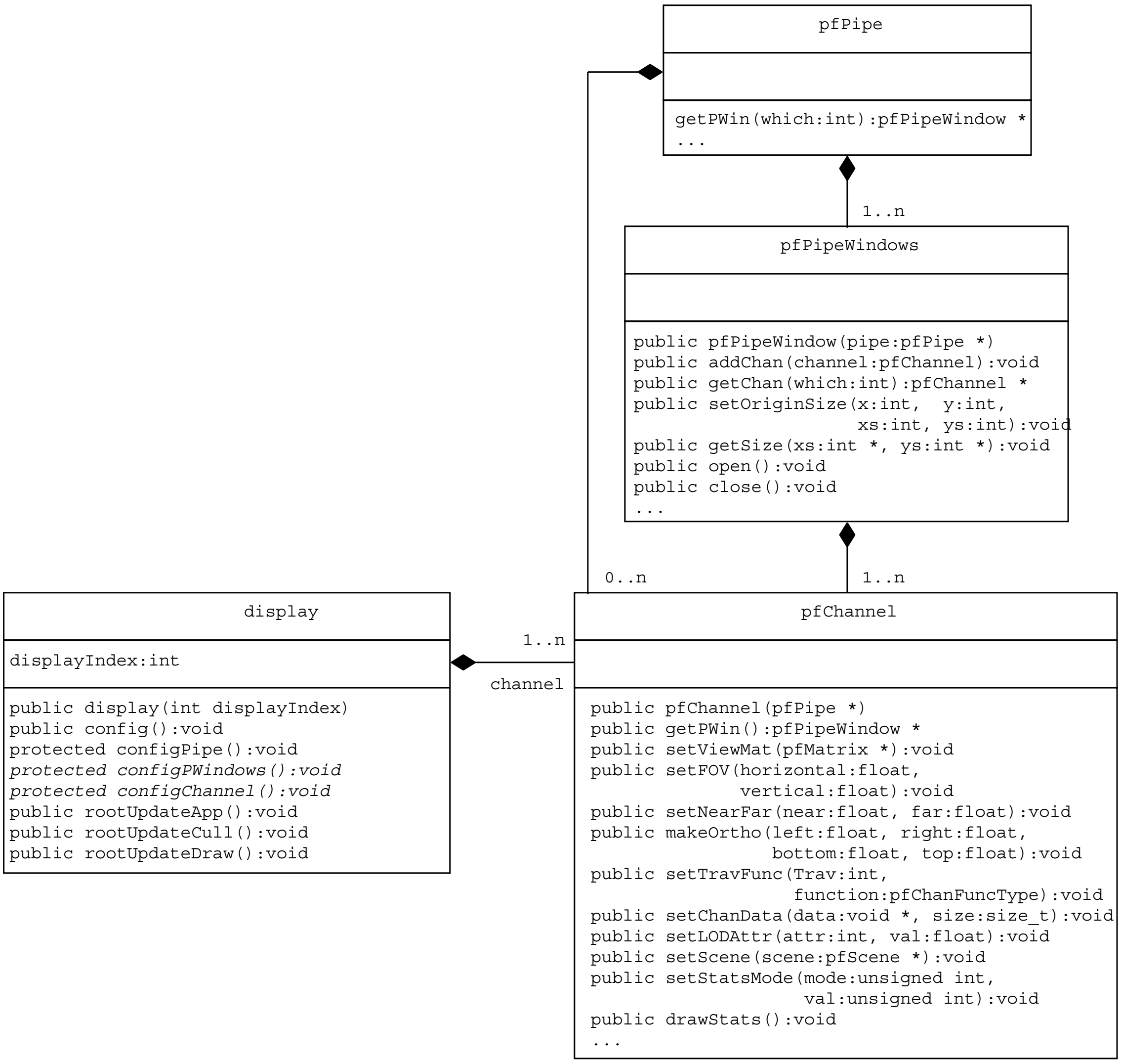


Abb. 4.26: Klassendiagramm der Klasse *display*

unter anderem noch andere Klassen von *display2D* erben, ist es notwendig, dass diese Methode virtuell bleibt.

Die Darstellung der 2D-Elemente wurde mit *OpenGL* entwickelt. *OpenGL Performer* benutzt im Hintergrund auch *OpenGL*. Es stellt sich die Frage, wie man von einem Grafikbaumsystem auf ein Prozeduralsystem wechseln kann? *OpenGL Performer* liefert hier eine Lösung. Um diese Lösung zu verstehen, muss man sich etwas tiefer gehend mit *OpenGL Performer* auseinandersetzen.

***callback*-Methode**

Um ein Bild zu generieren, durchläuft *OpenGL Performer* nach der Initialisierung drei bis vier Phasen: Die letzte Phase muss die Datenbank verwalten, aber diese Phase wird oft in der Literatur nicht erwähnt. Alle diese Phasen können als getrennte Prozesse laufen, wobei dies sich für die drei ersten Phasen nur für Rechner mit mehreren Prozessoren lohnt [Eck97]. Die drei Phasen sind:

- Die *Application*-Phase. Hier wird die Dynamik der Anwendung berechnet. Für die Flugführungsanzeige wird zum Beispiel die Information über die Position des Flugzeuges aktualisiert.
- Die *Culling*-Phase. Hier ändert *OpenGL Perfomer* den grafischen Baum. In vereinfachter Form kann man sagen, dass alles, was nicht im Sichtvolumen ist (definiert durch das Field-Of-View und die *near*- und *far-clipping* Ebenen), aus dem grafischen Baum gelöscht wird.
- Die *Drawing*-Phase . Hier werden unter anderem der Z-Buffer, die Beleuchtung und die Farbe laufend berechnet.

Für jede dieser Phasen erlaubt *OpenGL Performer*, dass der Softwareentwickler sogenannte *callback*-Funktion definiert. Sie sind keine Methoden, sondern nur Funktionen[6]. Der Typ und die Anzahl der Parameter solcher Funktionen sind von *OpenGL Performer* fest definiert. *OpenGL Performer* macht einige interne Berechnungen, ruft zu einem bestimmtem Zeitpunkt die *callback*-Funktion auf und fährt danach mit seiner Arbeit fort. *Callback*-Funktionen sind nicht Bestandteil von *OpenGL Performer*, stellen aber eine direkte Verbindung zur aktuellen Arbeit von *OpenGL Performer* her.

Für die 2D-Elemente gibt es die Möglichkeit, mit *OpenGL* zu arbeiten. Hier wird also für die Kanäle der 2D-Displays eine *callback*-Funktion in der *Drawing* Phase der 2D-Kanäle aufgerufen.

Die *callback*-Funktion für einen Kanal wird mit einem Zeiger auf den betroffenen Kanal aufgerufen. In einer Instanz der Klasse *pfChannel*, der Kanalklasse von *OpenGL Performer*, ist es möglich, einen allgemeinen Zeiger zu speichern. Hier kann der Zeiger *this*[7] von der Instanz *display* gespeichert werden. In der *callback*-Funktion wird man nur diese Benutzerinformation lesen. Mit diesem Zeiger kann man die *callback*-Methoden, hier *rootUpdateApp*, *rootUpdateCull* und *rootUpdateDraw*, aufrufen.

---

[6] Eine Funktion gehört nie einer Klasse

[7] Jede Methode einer Klasse verfügt über einen versteckten Parameter: den Zeiger *this*, der auf das eigene Objekt zeigt. Die Aufgabe des Zeigers *this* besteht darin, auf das jeweilige Objekt zu verweisen dessen Methode aufgerufen wurde [JL00]

Die *callback*-Methode der Draw-Phase, *rootUpdateDraw*, ist in *display2D* definiert. Diese Methode ruft einmal für jedes Bild die Klassenmethode *preDraw* der Klasse *element2D* (siehe Kapitel 4.6.3) und alle Methoden *drawAll* der Instanz auf. Sie aktualisiert auch die Größe der Fenster, wofür die Methode *getSize* von *pfPipeWindow* benutzt wird.

Die Abbildung 4.27 zeigt die Organisation der Klasse *display2D*.

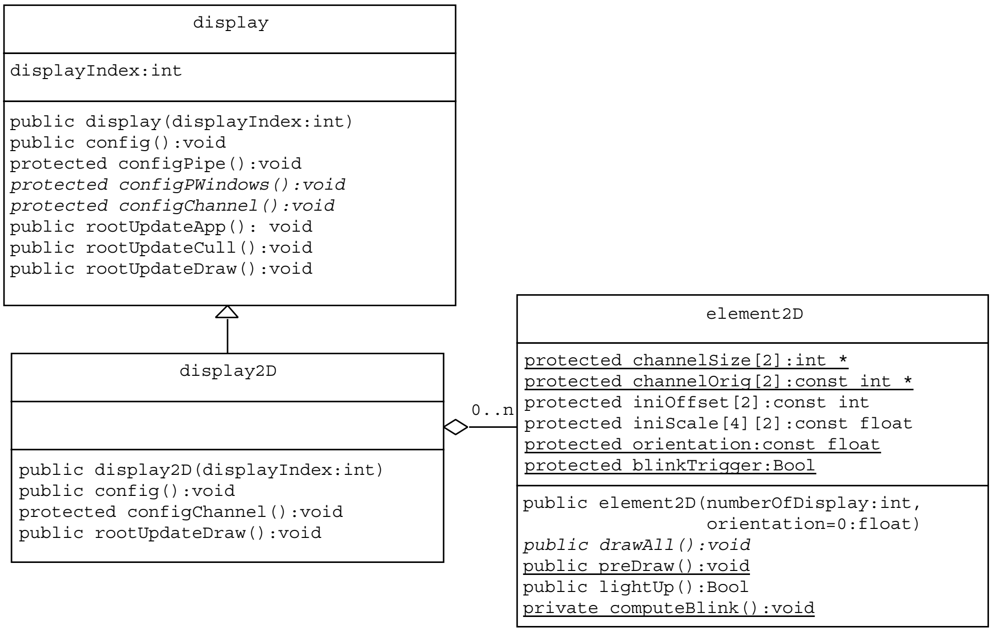


Abb. 4.27: Klassendiagramm der Klasse *display2D*

4.7.4 Verwaltung der 3D-Elemente in der Anzeigenanwendung

Die Verwaltung der 3D-Elemente ist etwas komplexer als jene der 2D-Elemente. Einige *callback*-Methoden sind hier definiert: eine in der *Application-Phase* und eine in der Darstellungsphase.

Die *callback*-Methode der *Application Phase* soll unter anderem die Position und die Lage des Flugzeuges aktualisieren. Die Dynamik der 3D-Elemente wird auch von dieser *callback*-Methode aufgerufen: Die Berechnung des Prädiktors, oder der Fremdenflugzeuge wird mit Hilfe eines Aufrufs einer Methode des 3D-Elemente-Verwalters vollzogen. Wie für die 2D-Elemente wird also eine Aktualisierung für jedes 3D-Element aufgerufen.

Für die 3D-Welt mit *OpenGL Performer* ist es notwendig, eine Szene, *pfScene*, zu definieren. Eine Szene ist der erste Knoten im grafischen Baum von *OpenGL Performer*. Ein

Kanal, *pfChannel*, besitzt normalerweise eine Szene. Die 2D-Elemente brauchen diese nicht, weil sie nicht mit *OpenGL Performer* entwickelt wurden. Die Szene wird in den *Channel* (siehe Abb. 4.26) eingefügt.

Eine Lichtquelle muss die Szene beleuchten. Die Lichtquelle, *sun* in der Abbildung 4.28, wird mit der Klasse *pfLightSource* erzeugt. Es wurde auf Seite 14 erwähnt, dass die Lichtquelle immer an einer *ortsfesten* Stelle der Anzeige bleiben muss, um den Invertierungseffekt zu vermeiden: Wenn der Kurs des Flugzeugs sich ändert, muss die Lichtquelle sich mitbewegen. In 4.4.1 wurde gezeigt, dass sich das Gelände anstatt des Flugzeugs bewegt. Daher ist die Szene bezüglich des Flugzeugs statisch, es ist also auch möglich, die Hauptlichtquelle direkt an die Szene zu binden.

Diese Software gibt nun die Möglichkeit, die Position der Sonne während der Laufzeit zu ändern. Dies ist möglich, wenn die Lichtquelle an ein *Dynamic Coordinate System*, *pfDCS*, gebunden ist.

Wie die Position sind auch die Farben der Lichtquelle und die Intensität während der Laufzeit änderbar. Dies wird innerhalb der *callback*-Methode *rootupdateApp* gemacht.

Die Geländeinformation in der orthogonalen Anzeige und der perspektivischen Anzeige ist die gleiche. Sichtrichtung und Projektionsart sind allerdings nicht identisch. Um Rechenzeit und Speicherplatz zu sparen, ist es interessant, die Geländeinformation in den beiden Arten von Display (*orthogonalDisplay* und *perspectiveDisplay*) so weit wie möglich wiederzuverwenden.

Ein Teil der Flugzeugbewegung, nämlich der Kurs und die horizontale Bewegung, betrifft beide Arten von Displays. Dies wird mit dem Knoten, *pfdNdDcs* der Klasse *pfDCS*[8], im grafischen Baum definiert. Dieser Knoten reagiert auf den Kurs und die laterale Position. Er ist auch der Wurzelknoten für die Geländeinformation. Über diesem Knoten steht der Knoten *dcs*. Für die perspektivische Anzeige reagiert er auf Nick- und Rollwinkel und auf die Höhe. Für ein *Navigation Display* berücksichtigt dieser Knoten die Zoomstufe. Für ein *Vertical Situation Display* reagiert *dcs* auf die Höhe und die Zoomstufe.

Die Klasse *display3D* verwaltet die Dynamik des Knoten *pfdNdDcs*, jedoch nicht jene des Knoten *dcs*. Die Bewegung von *pfdNdDcs* wird im *rootUpdateApp* berechnet.

Die Abbildung 4.28 zeigt die Organisation der Klasse *Display3D*. Die Klasse *Display3D* besitzt einige Instanzen der Klasse *drawObject3D*. Diese Klasse verwaltet die Modelle und die Bewegungen von 3D-Elementen wie dem Prädiktor, Fremdflugzeugen oder dynamischen Kanälen.

---

[8] *Dynamic Coordinate System*

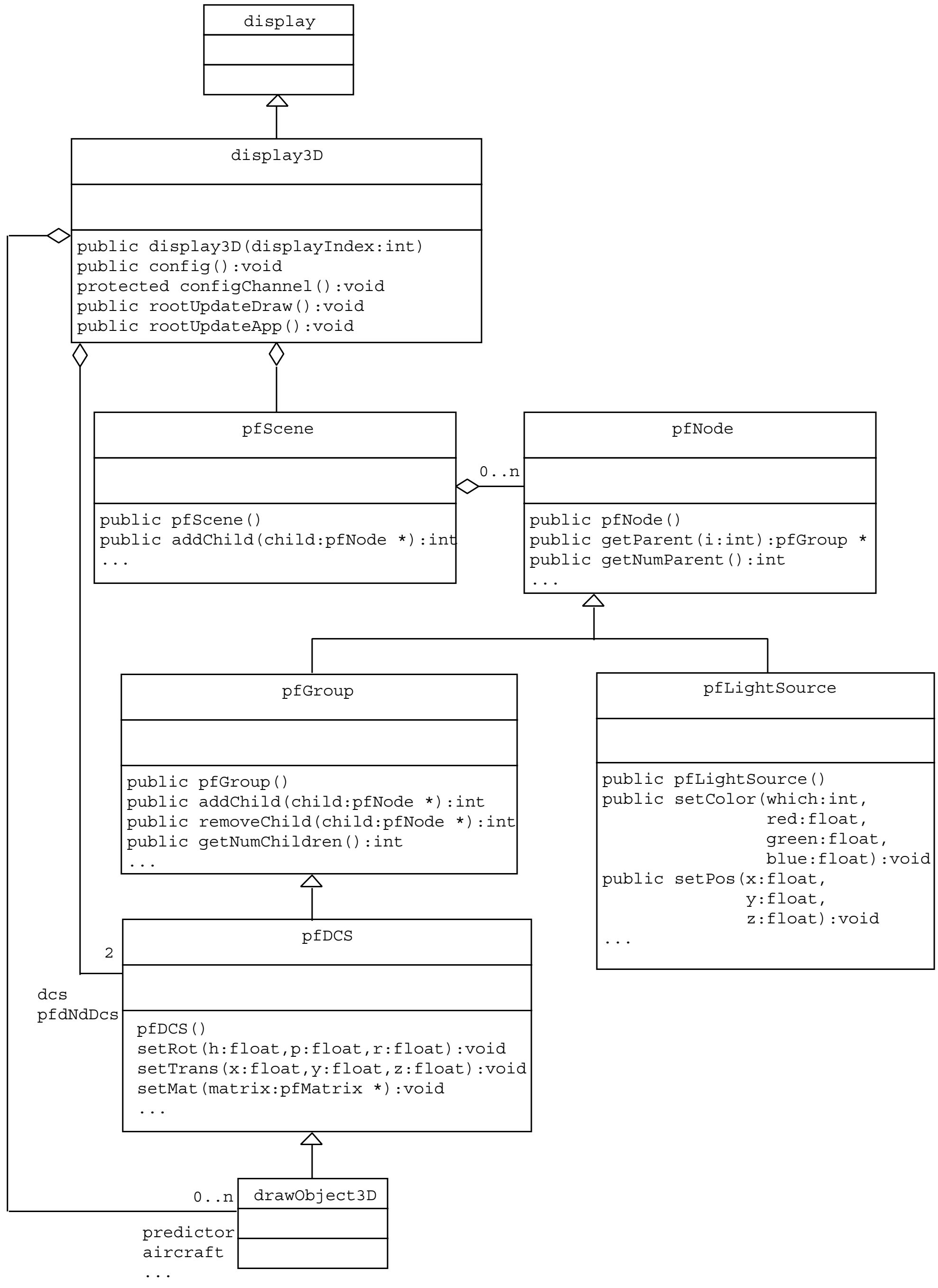


Abb. 4.28: Klassendiagramm der Klasse *display3D*

4.7.5 Verwaltung der orthogonalen Anzeigen

Die orthogonalen Anzeigen benutzen eine orthogonale Projektion für die Szene. Die Hauptaufgabe der Klasse *orthogonalDisplay* besteht in der richtigen Konfiguration und

Verwaltung solcher Projektionen. Die Richtung der Achsen des Kordinatensystems kann hier noch nicht fest definiert werden, weil die Sichtrichtung eines ND und eines VSD nicht die gleiche ist.

Die Abbildung 4.29 zeigt das Klassendiagramm der orthogonalen Anzeigen.

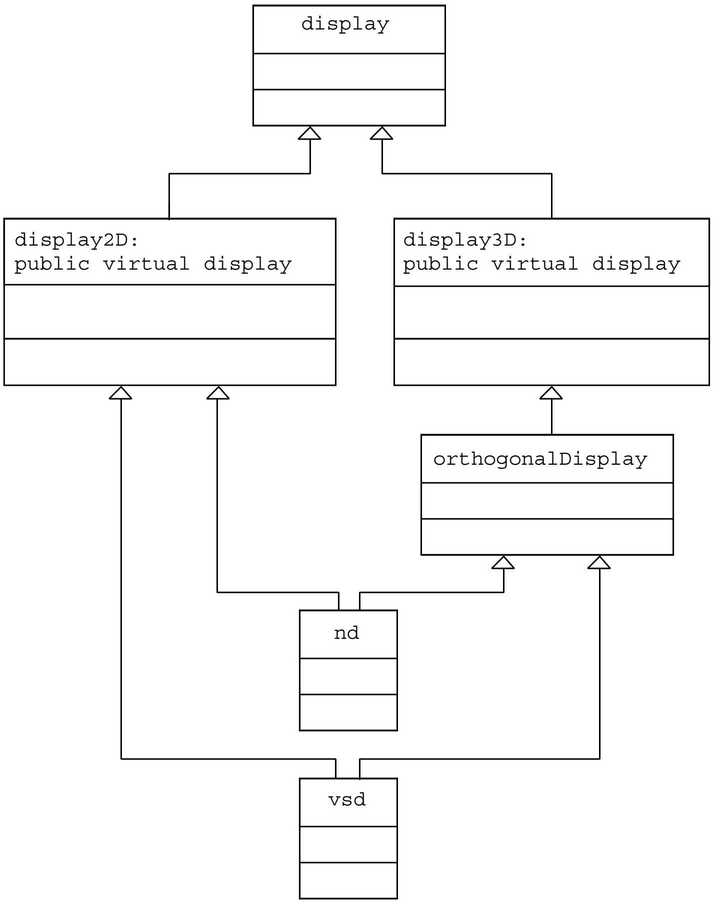


Abb. 4.29: Klassendiagramm der Klasse orthogonalDisplay

Klasse nd

Ein *Navigation Display* ist sowohl eine 2D-Anzeige (*display2D*) als auch eine 3D-Anzeige (*display3D*), wobei die Projektion eine Parallelprojektion ist (siehe Abb. 4.29).

Die Klasse *nd* erbt gleichzeitig von *display2D* und *orthogonalDisplay* beide sind Unterklasse der Klasse *display*. Diese Mehrfachvererbung generiert eine Reihe von Konflikten. Es gibt zum Beispiel zwei Sichten des Objekts *channel*, dass in *display* definiert ist. Eine Sicht entsteht durch die Vererbung von *orthogonalDisplay* und eine Sicht durch die Vererbung von *display2D*. In der Klasse *nd* muss man deshalb spezifizieren, welche Sicht vom *channel* man verwenden möchte. Für jedes Objekt und jede virtuelle Methode, die der Klasse *display* gehört, gibt es einen solchen Konflikt. Um solche Konflikte zu vermeiden, kann man mit C++ spezifizieren, welche Sicht jetzt gemeint ist:

```
display2D::channel[Channel2D+i]->setViewMat(matrix2D);
orthogonalDisplay::channel[Channel3D+i]->setViewMat(matrix3D);
```

Die erste Zeile definiert die Position und die Richtung des Sichtpunkts in der Sicht des *display2D*, und die nächste Zeile macht das gleiche, aber für die 3D-Welt und durch *orthogonalDisplay*.

Trotzdem bleibt eine Frage offen: Muss man die Instanz der Klasse *display* zweimal generieren? Muss man zum Beispiel zwei Felder von *pfChannel* benutzen und ebenso zwei Fenster? Hier erlaubt C++, die Instanz nur einmal zu definieren. Die Klassen *display2D* und *display3D* erben die Klasse *display* virtuell: Die Klasse *display* ist eine virtuelle Basisklasse der Klassen *display2D* und *display3D*.

Somit werden die internen Instanzen der Klasse *display* nur einmal definiert. Die virtuellen Methoden der Klasse *display* müssen aber auch in *nd* wieder definiert werden. In *nd* müssen die Methoden nur die zugehörigen Methoden der 2D- und 3D-Teile ihrer Basisklassen aufrufen. So sind zum Beispiel die *callback*-Methoden in *nd* wieder definiert und rufen die *callback*-Methoden der Klasse *display2D* und *orthogonalDisplay* auf. Es gibt auch einige Methoden, die nicht die Definition der Basisklasse aufrufen. Bestes Beispiel ist hier die Konfiguration des Sichtpunktes oder *configChannel*: Für die 2D-Teile wird die Methode von *nd* die Version von *display2D* aufrufen, aber die Konfiguration der 3D-Kanäle ist nirgends definiert; es ist nur möglich, in *nd* selbst diese Konfiguration zu realisieren.

Die Konfiguration und die Berechnung einer stereoskopischen Darstellung werden so weit wie möglich in der Basisklasse vollzogen.

Klasse vsd

Die gleichen Prinzipien werden für *vsd* benutzt. Aber der Sichtpunkt und die Orientierung werden unterschiedlich berechnet. Wie schon erläutert wurde, soll *vsd* nicht nur die Zoomstufe verwalten, sondern auch die Höhe des Flugzeugs. Diese wird im *nd* nicht verwaltet.

Diese Anzeige muss auch einen Geländeschnitt berechnen. Diese Rechnung wird nicht ausschließlich in der Anzeige-Software durchgeführt. Die Geländequerschnittsdaten können auch aus einer vorberechneten Datei gelesen werden.

4.7.6 Verwaltung der perpektivischen Anzeige

Im Gegensatz zur Klasse *orthogonalDisplay*, kann man in der Klasse *perspectiveDisplay* viel mehr Eigenschaften der Unterklassen vorgeben. Die Unterklassen modellieren zur Zeit nur perspektivische Anzeigen, die immer den gleichen Sichtpunkt und die gleiche Orientierung haben. Das HMD ist hier eine Ausnahme: Wegen des Head-Trackings ist die Sichtrichtung abhängig von der Kopfposition des Piloten. Aber dies ist lediglich eine zusätzliche Bewegung, die Grundbewegung bleibt die gleiche. Für das HUD ist die Referenz nicht exakt die gleiche, wie die des HDPFD oder des Visuals, aber auch sie verändert sich zur Laufzeit nicht.

Der Knoten *dcs* (siehe Kapitel 4.7.4) muss also auf die Längsneigung, den Rollwinkel und auf die Höhe reagieren. Der *pfdNdDcs* reagiert auf den Kurs und die horizontalen Koordinaten. Diese Verwaltung befindet sich in der Klasse *perspectiveDisplay*.

Die Abbildung 4.30 zeigt das Klassendiagramm der perspektivischen Anzeigen.

Klasse visual

Die Klasse *visual* modelliert eine einfache Außensicht. Für diese Anzeige bedarf es keiner 2D-Elemente. Hier ist die Vererbung also viel einfacher als beim *orhtogonalDisplay* oder den anderen perspektivischen Anzeigen.

Im Gegensatz zu anderen perspektivischen Anzeigen wird der Himmel mit einem einfachen blauen Hintergrund gezeichnet.

Klasse hdPfd

Ein Head Down Perspective Flight Display ist eine einfache perspektivische Darstellung mit 2D-Elementen. Diese Klasse erbt von *perspectiveDisplay* und *display2D*.

Der Himmel wird durch einen Halbzylinder definiert. Dieser Halbzylinder folgt dem Kurs und der Querneigung, hingegen nicht dem Nickwinkel. Man kann für diesen Himmel verschiedene Modelle verwenden, der Himmel wird beim Starten der Software ähnlich wie ein Geländeteil geladen. Bei einem dieser Modelle, dem sogenannten *pitch sky*, ist der Halbzylinder in 10-Grad-Segmente mit jeweils anderer Farbe unterteilt (siehe Abb. 4.31). Dadurch, dass der Halbzylinder der Nickbewegung nicht folgt, liefert dieses Modell den Piloten eine zusätzliche Information über den Nickwinkel des Flugzeuges.

Dieser Himmel muss der vertikalen Bewegung des Flugzeugs folgen. Man könnte statt dessen auch einen Zylinder mit einem sehr großen Radius definieren. Allerdings besteht dann die Gefahr, dass die Flughöhe größer wird als dieser Radius und das Flugzeug durch

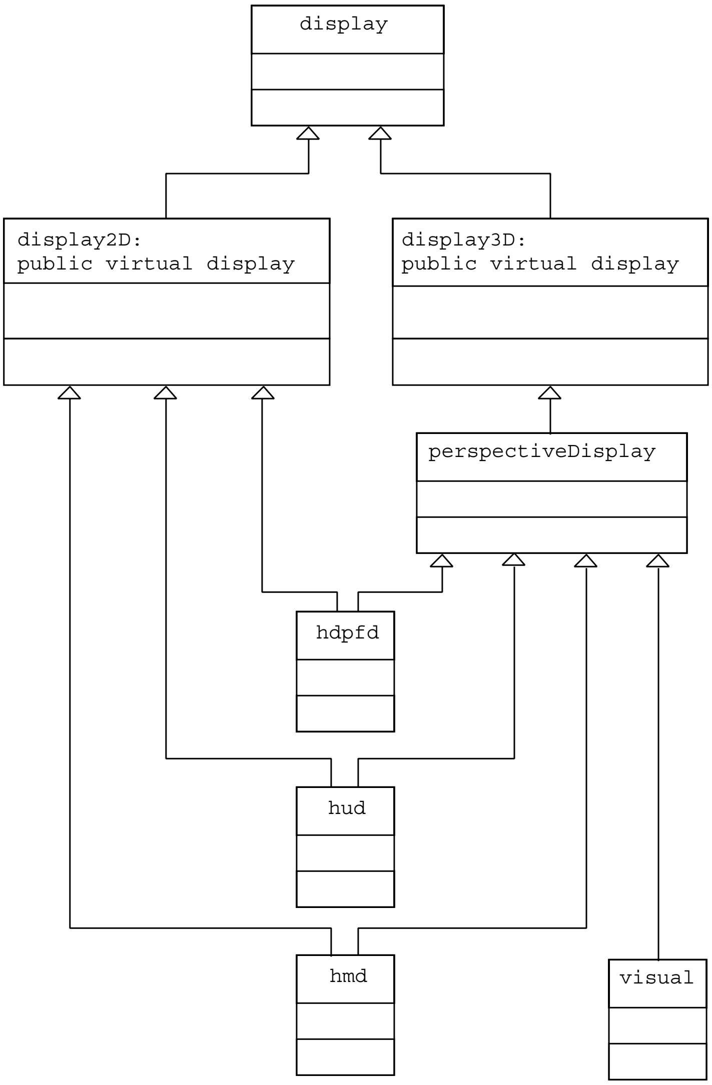


Abb. 4.30: Klassendiagramm der Klasse perspectiveDisplay

den Himmel hindurch fliegt. Wählt man den Radius größer als jede denkbare Flughöhe, kommt es zu Performance-Einbußen durch eine zu weit entfernt liegende *far clipping*-Ebene (siehe 4.4.2). Deswegen ist die Mitbewegung des Himmels sinnvoll, beim Zeichnen des Horizonts.

Die Null-linie des Anstellwinkels muss ständig deutlich dargestellt sein. Bei einem konventionellen künstlichen Horizont wird diese Null-linie durch einen Farbwechsel nachgebildet: Der *Boden* (alles, was unterhalb dieser Null-linie liegt) ist braun, der *Himmel* blau. Mit einer 3D-Geländedarstellung ist diese einfache Lösung nicht mehr möglich. Hier muss diese Grenze ständig durch eine weisse Gerade dargestellt werden. Das Konzept

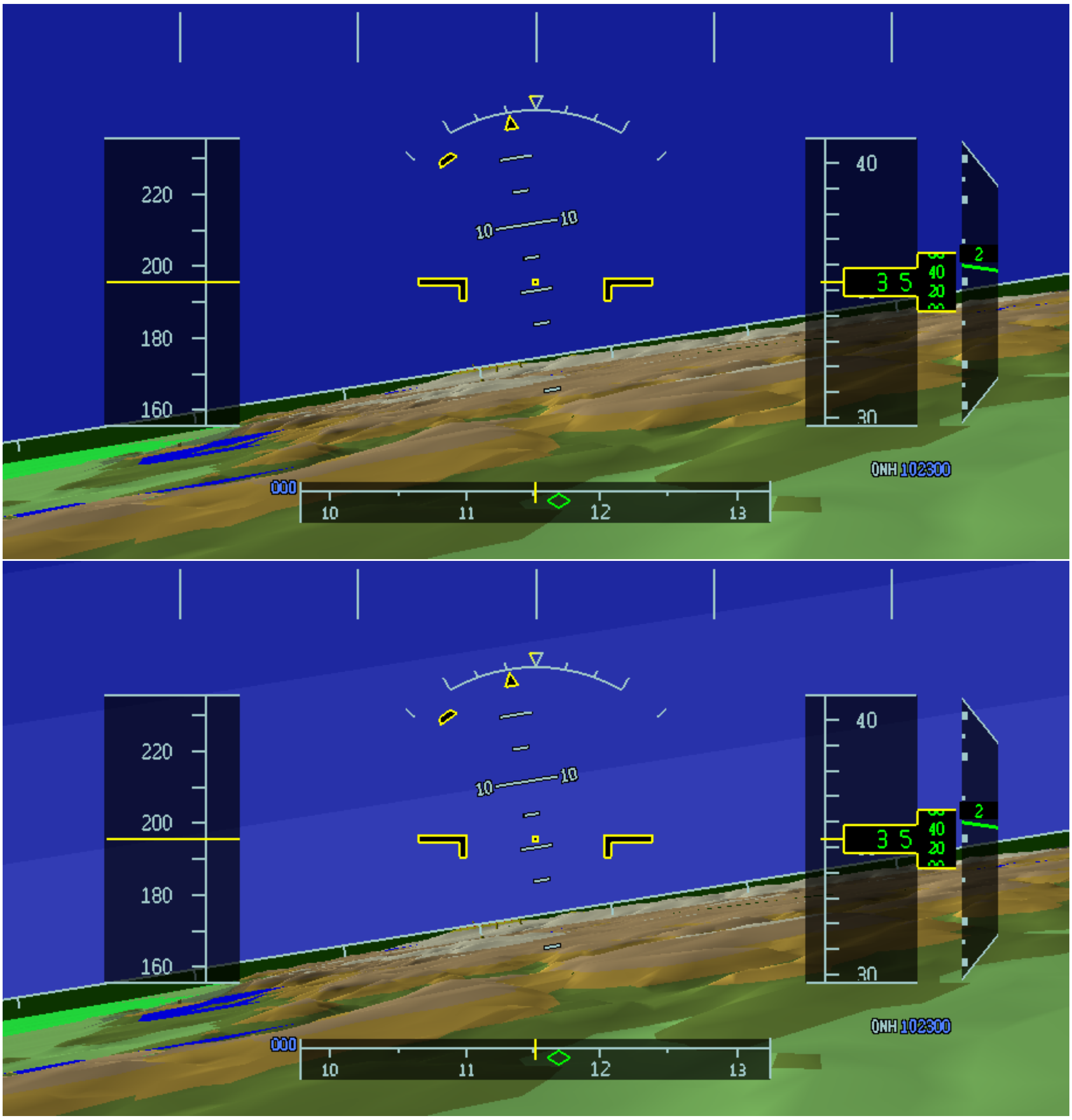
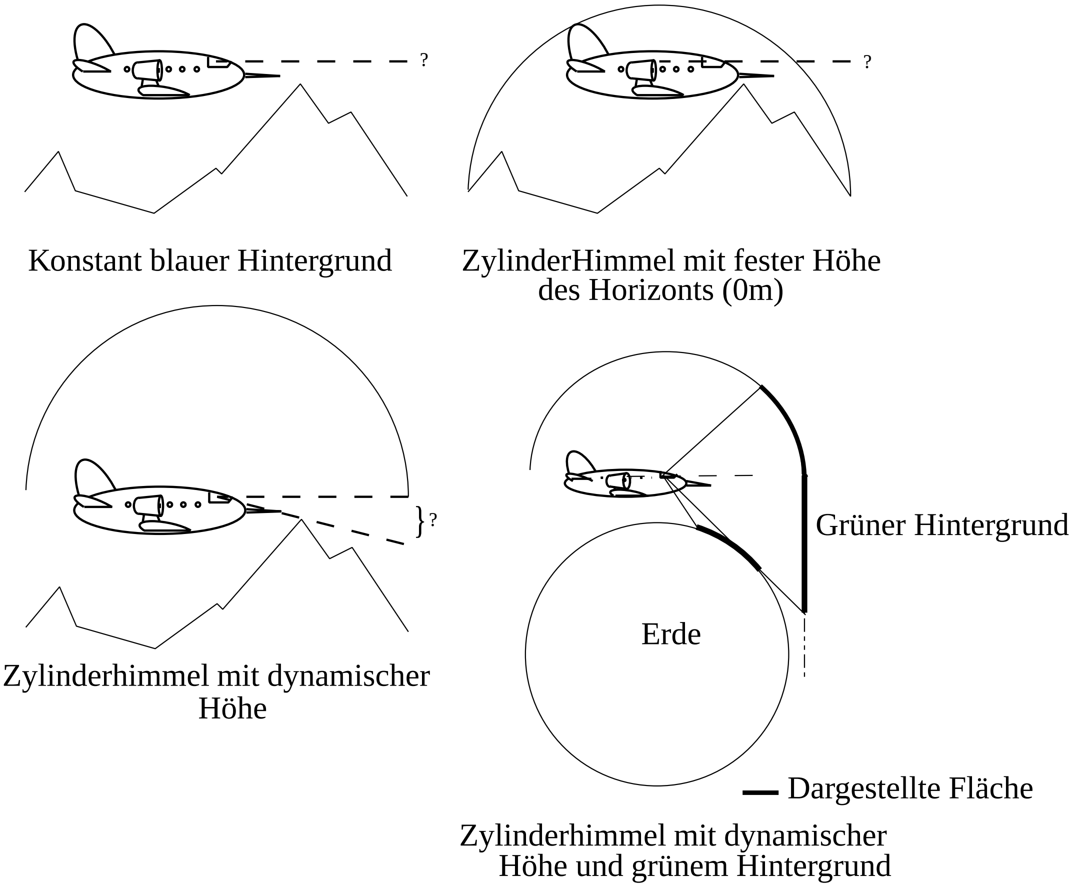


Abb. 4.31: Verschiedene Himmel in Hintergrund des HDPFD

des Farbunterschieds zwischen *Boden* und *Himmel* soll allerdings beibehalten werden. Die Verwendung des mitbewegten Himmels führt dabei allerdings zu einer Schwierigkeit.

Das Flugzeug befindet sich ständig im Mittelpunkt des Himmel-Halbzylinders. Dadurch liegt der Rand des Himmels immer auf einer Höhe mit der weißen Horizontlinie. Fliegt das Flugzeug unterhalb der höchsten Berge, verdecken diese den Rand des Himmel-Halbzylinders. Dies verbessert das Situationsbewusstsein des Piloten, der nun deutlich erkennen kann, dass er unterhalb der Berge fliegt. Fliegt das Flugzeug jedoch deutlich oberhalb der höchsten Berge, entsteht unterhalb der künstlichen Horizontlinie ein undefinierter Bereich. Dieser erstreckt sich von dieser Linie bis zum realen Horizont, der durch den Berührungspunkt einer durch den Flugzeugmittelpunkt führenden Erdtangente gebildet wird (siehe Abb. 4.32).

Es stellt sich nun die Frage, was man in diesem undefinierten Bereich darstellt. Als Lösung erhielt die Anzeige einen grünen Hintergrund, so dass der undefinierte Bereich im Display grün erscheint.



Konstant blauer Hintergrund

ZylinderHimmel mit fester Höhe des Horizonts (0m)

Zylinderhimmel mit dynamischer Höhe

Zylinderhimmel mit dynamischer Höhe und grünem Hintergrund

Abb. 4.32: Künstliche Horizonte

Der Himmel ist auf dem HDPFD das entfernteste Element. Er definiert die *Far Clipping*-Ebene (siehe Kapitel 4.4.2 51). Große Entfernungen muss man mit zahlenmäßig großen Werten berechnen. Die Rechnerungenauigkeiten spielen hier wieder eine Rolle. Daher darf man die *Far Clipping*-Fläche nicht genau hinter den Himmel setzen, sondern muss dazwischen einen gewissen Spielraum lasse.

hud

Ein HUD sieht dem HDPFD sehr ähnlich, aber hat keine 3D-Geländedarstellung. Eine 3D-Erweiterung des HUD würde eine Geländedarstellung mit einem *Wireframe-Modell*[9] oder einem Höhenlinien-Modell erfordern, um die Transparenz der Anzeige zu erhalten.

Auf jeden Fall braucht ein HUD, im Gegensatz zu HDPFD, keine Hintergrund-Farbe, damit der Hintergrund nicht das echte Gelände verdeckt. Daher wird auch kein Himmel definiert.

4.7.7 Unabhängige 3D-Elemente: drawObject3D

In der 3D-Welt der Flugführungsanzeige gibt es außer dem Gelände noch weitere 3D-Elemente. Kennzeichen dieser Elemente ist eine eigene Dynamik, die teilweise unabhängig von der Bewegung des Flugzeuges ist. Nachfolgend eine kleine Liste der 3D-Elemente:

- Die *intruder*: Die fremden Flugzeuge, die zur Warnung angezeigt werden,
- Der *followMeAircraft*: Ein Flugführungs-Symbol,
- Der *3D Flight Path Vector* und *3D Flight Path Vector Target*,
- Der Prädiktor: Ein Symbol zeigt die zukünftige Position des Flugzeugs, mit ca. 3-10 Sekunden Vorschau.
- Die *Route*: Ein Kanal, der den Sollweg des Flugzeuges anzeigt.

Die Klasse *display3D*, hat eine verkettete Liste von *drawObject3D*-Objekten, wobei die Instanzen der Liste Unterklassen der Klasse *drawObject3D* sind. Die Klasse *drawObject3D* ist demzufolge die Basisklasse der oben aufgeführte 3D-Elemente. Jede Art von 3D-Element wird durch eine eigene Unterklasse modelliert.

Die Klasse *drawObject3D* benutzt ein Datenbankmodell für die Darstellung der 3D-Elemente, wobei der Name des Modells durch die Unterklasse definiert ist.

---

[9] Eine Wireframe Darstellung ist eine Darstellung in der nur die Linie der Dreiecke der Polygone dargestellt werden. Die normale Darstellung füllt die Dreiecke.

Die Dynamik des 3D-Elementes wird in der Unterklasse mit der Methode *update* berechnet. Diese Methode ist als pur virtuelle Methode in *drawObject* definiert. So wird zum Beispiel *hdPfd* die update-Methode von jedem Element der *drawObject3D* Liste aufrufen.

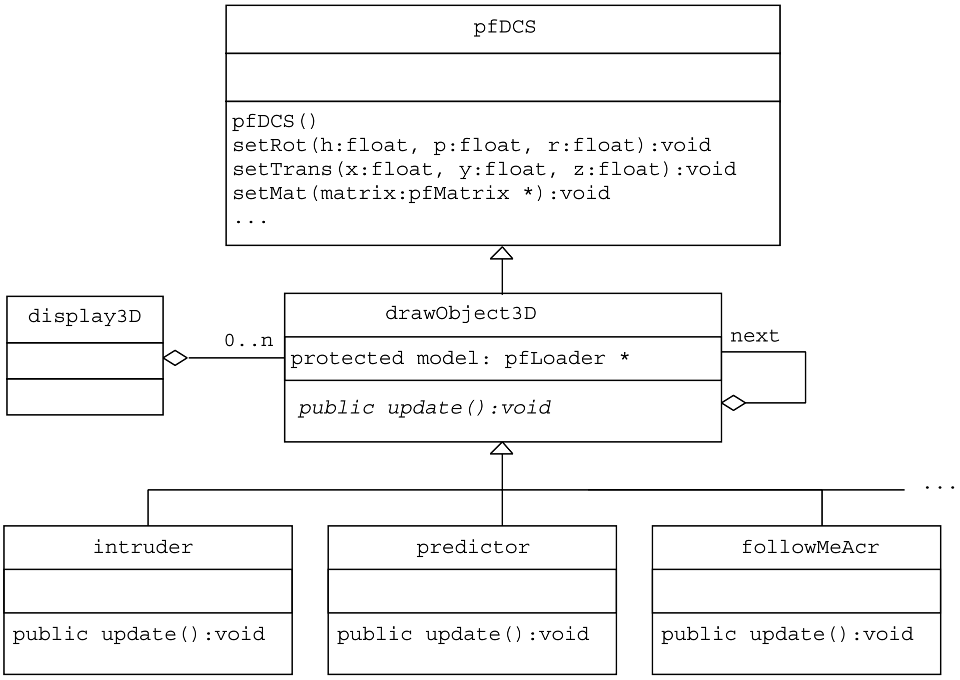


Abb. 4.33: Klassendiagramm der Klasse *drawObject3D*

4.7.8 Verwaltung der Datenbank

Es ist unmöglich, die Geländeinformation der ganzen Erde auf einmal zu laden. Wie erläutert wurde, ist eine Fläche für eine Computerdarstellung aus Dreiecksflächen aufgebaut. Je präziser die Darstellung ist, desto mehr Dreiecke sind nötig. Für jedes Dreieck braucht man nicht nur die Koordinaten der Eckpunkte, sondern auch mindestens einen Normalenvektor. Um die Beleuchtung zu berechnen, ist die Farbe der Dreiecke ebenfalls notwendig. Daraus folgt, dass der Arbeitsspeicher eines Rechners sehr schnell voll sein wird.

Aus diesem Grund wird die grafische Information des Geländes auf Massenspeichern abgelegt. Um die grafisch dargestellte Information zu beschränken und einen schnellen Zugriff auf geographisch spezifizierte Gebiet zu erlauben, sind die Daten in Kacheln organisiert, die jeweils ein festes geographisches Gebiet abdecken [May01a]. Auf Massenspeichern werden Dateien gespeichert, die einer Kachel entsprechen.

Über einen definierten Verzeichnissbaum kann auf jede Datei schnell zu gegriffen werden. Dazu wird ein Datenbankkonzept benutzt: Eine Datenbank ist eine Ansammlung von elektonisch gespeicherten Informationen (Daten), die thematisch zusammenhängen, und die unter verschiedensten Fragestellungen abgerufen werden können [HDR96]. Mit Hilfe einer externen Bibliothek und abhängig von der aktuellen Position findet man so schnell eine einzelne Datei.

Die Informationen in einer Datei sind ebenfalls in einem Baum geordnet. Es gibt Knoten (vergleichbar mit Verzeichnissen in einem Dateisystem) und Blätter (vergleichbar mit Dateien in einem Dateisystem). Ein Knoten beschreibt zum Beispiel das Gelände, ein anderer Knoten beschreibt die Hindernisse, die auf dem Gelände liegen. Unter dem Hindernisse-Knoten kann man weitere Knoten für verschiedene Arten von Hindernissen (Kirchen, Hochhäuser, Türme, beleuchtete oder nicht beleuchtete Hindernisse usw.) finden.

Dieses Kapitel wird zunächst erläutern, warum eine Simultanverarbeitung für die Datenbankverwaltung wichtig ist. Danach werden verschiedene Geländemodelle erläutert und schließlich die Realisierungen eines Konzepts erklärt, das dazu dient, die Geländeinformation zu laden, und um diese Information zu verwalten.

Simultanverarbeitung

Auf Grafikrechnern stehen oft mehrere Prozessoren zur Verfügung. In einem symmetrischen Parallel- oder Multiprozessorsystem arbeiten alle Prozessoren gleichberechtigt und gleichzeitig an den Aufgaben. Es ist meistens möglich, eine komplexe Software in einige nahezu unabhängige Teile, Prozesse, zu trennen. Diese Prozesse können auf einem Rechner, der mehrere Prozessoren hat, parallel laufen. Bei Rechnern, die nur einen Prozessor haben, wird die Parallelität jedoch nur vorgetäuscht. Tatsächlich verteilt der Prozessor die Rechenzeit so zwischen den verschiedenen Anwendungen, dass alle Anwendungen scheinbar gleichzeitig ausgeführt werden [HDR96]. Der Multitask-Kern des Betriebsystems teilt jedem Prozess eine begrenzte Zeit für die Nutzung der Prozessoren zu. Am Ende dieser Zeit, oder des Zyklus, stellt der Kern des Betriebsystems den Prozessor anderen Prozessen zur Verfügung. Der nächste Prozess ist derjenige, der das größte Vorrecht hat. Es gibt verschiedene Vorrechtsprinzipien. Jeder Multitask-Kern hat seine eigene Strategie.

Es wird unterschieden zwischen Multi-Thread und Multi-Prozess-Programmen. Ein über Multi-Prozessing abgespaltener Prozess verhält sich wie ein gänzlich selbständiges Programm, das sozusagen unabhängig vom Hauptprozess läuft.

Ein Thread ist eine Funktion, die parallel zur Hauptfunktion läuft. Zwei Threads (oder zwei Funktionen, die parallel laufen) können den gleichen Speicherbereich benutzen, wohingegen zwei Prozesse zwei getrennte Speicherbereiche benutzen.

Zwei Prozesse haben bis zum Zeitpunkt der Aufspaltung die gleiche Information in ihrem gemeinsamen Speicherbereich. Wenn sich danach eine Information bei einem Prozess ändert, wird der neue Wert in beiden Prozessen nicht mehr identisch sein. Daher allokiert man einen speziellen Speicherbereich vor der Aufspaltung der Prozesse, auf dem beide Prozesse nach der Aufspaltung zugreifen können. Dieser Speicherbereich gehört weder dem ersten Prozess noch dem zweiten, sondern normalerweise dem Betriebssystem [Rif94]. Dieser Speicherbereich steht jedem Prozess, der sich korrekt anmeldet, zur Verfügung. Auf diese Weise können zwei Prozesse Informationen in diesem *Shared-Memory*-Bereich austauschen.

Unter anderem gibt es im *Shared Memory* auch Informationen, um darauf zugreifende Prozesse zu synchronisieren. Als Beispiel einer Synchronisation kann man sich eine *Client/Server*-Software denken: Ein Client bittet mit Hilfe des *Shared-Memory* einen Verwalter um einen Dienst. Daraufhin wird eine Nachricht vom Verwalter zum Client geschickt. Erst wenn diese angekommen ist, kann der Client mit den Ergebnissen des Verwalters weiterarbeiten.

Dieses Beispiel zeigt den größten Vorteil des Multiprocessings. Ein Client kann andere Aufgaben bearbeiten, während der Server noch mit dem vom Client bestellten Dienst beschäftigt ist. Der Client ist also nicht blockiert.

Für die Flugführungsanzeige-Anwendung müssen die Anzeigen einerseits so oft wie möglich aktualisiert werden. Sie benötigen andererseits aber viel Zeit, neue Daten für die Geländedarstellung aus der Datenbank zu laden. Die Anzeigen dürfen aber keinesfalls dadurch blockiert werden, dass ein neuer Teil des Geländes geladen wird: Die 2D-Darstellung muss auf jeden Fall weiter aktualisiert und angezeigt werden.

Wie schon erwähnt wurde, ist es mit *OpenGL Performer* möglich, die Hauptteile der Software in mehrere unterschiedliche Prozesse aufzuteilen. Um eine Blockierung der Anzeige zu vermeiden, ist es daher empfehlenswert, den Prozess für das Laden der Daten als separaten Prozess laufen zu lassen. Für jede *Pipe* ist es zusätzlich möglich, die folgende Aufspaltung zu realisieren:

- einen *Application*-Prozess,
- einen *Culling*-Prozess,

- und einen *Drawing*-Prozess.

Allerdings ist dieser Schritt nur sinnvoll, wenn so viele Prozessoren wie Prozesse zur Verfügung stehen: Für eine Anwendung, die drei *Pipes* benutzt, sollte man günstigerweise $3 * 3 = 9$ Prozessoren haben. Im FSR wird mit Ausnahme der Außensicht nur eine *Pipe* benutzt. Die für die Flugführungsanzeige des FSR verwendeten Rechner besitzen ein bis zwei Prozessoren. Wie gerade erläutert wurde, ist es aber trotzdem interessant, auch bei nur einem Prozessor einen separaten Prozess für das Laden der Datenbank zu verwenden.

Verschiedene Geländemodelle

Da die Daten von einem Massenspeicher geladen werden, ist es möglich, unterschiedliche Modelle eines Geländeteils bereitzuhalten. Dies ermöglicht unter anderem Versuche zur Performance und Detaillierung der Anzeige.

OpenGL Performer erlaubt mehrere Dateiformate für die Datenbank. Um Aufwand zu sparen, wird hier nur ein Dateiformat benutzt. Im Rahmen dieser Arbeit wird das *Open Inventor* Format verwendet. *Open Inventor* ist eine grafische Bibliothek, die in [Wer94] beschrieben wurde. Diese Bibliothek besitzt ein eigenes Daten-Format. Diese Bibliothek ist *OpenGL Performer* sehr ähnlich, aber im Gegensatz zu *OpenGL Performer* erlaubt sie zur Zeit keine Parallelprozesse ohne zusätzlichen Code und kann nicht mit mehreren *Pipes* arbeiten.

Zur Zeit werden folgende Modelle benutzt (die Abbildungen der Anzeige 4.34 und 4.35 entstammen dem gleichen Flug, sie unterscheiden sich durch die Art der Datenbank):

- Trianguliertes-Rastermodell (oder reguläres Gitter-Modell) [Dat90]: Alle Punkte liegen auf einem Gitter. Mit diesem Modell werden viele unnötige Punkte definiert, da stets alle Punkte des Gitters gegeben werden, auch wenn das Gelände auf einer Ebene liegt. In 4.4.4 (Seite 57) wurde erläutert, dass die Reihenfolge der Punkte einen sehr großen Einfluss auf die Performance hat. Dieses Modell benutzt sehr viele Punkte, jedoch sind sie andererseits am besten für das Rendering geordnet.

  Da es zu viele unnötige Punkte gibt, ist dieses Modell nur ein Grundmodell, um andere zu generieren. Die Flugführungsanzeige könnte trotzdem solche Modelle benutzen. Die Anzahl der Polygone ist aber viel zu groß, und trotz der geeigneten Ordnung der Polygone würde die Performance zu schlecht werden.

- TIN[10] Modell: Unnötige Punkte des Rastermodells werden hier entfernt und das Modell kontrolliert dezimiert. Der Vorteil dieser Darstellungsart liegt in der be-

---

[10] Triangular Irregulate Network

sonderen Eignung für 3D-Visualisierungen, da normalerweise wesentlich weniger Flächen als bei regulären Gittern für die grafische Darstellung berechnet werden müssen und damit bei der Verwendung heutiger Rechnerleistung erst eine interaktive Betrachtung des Modells ermöglicht wird [May01a]. Viel Speicherplatz wird gespart, hingegen ist es schwieriger, die Punkte für das Rendering zu ordnen. Für dieses Modell wird *Gouraud Shading* als Farbverlauf benutzt.

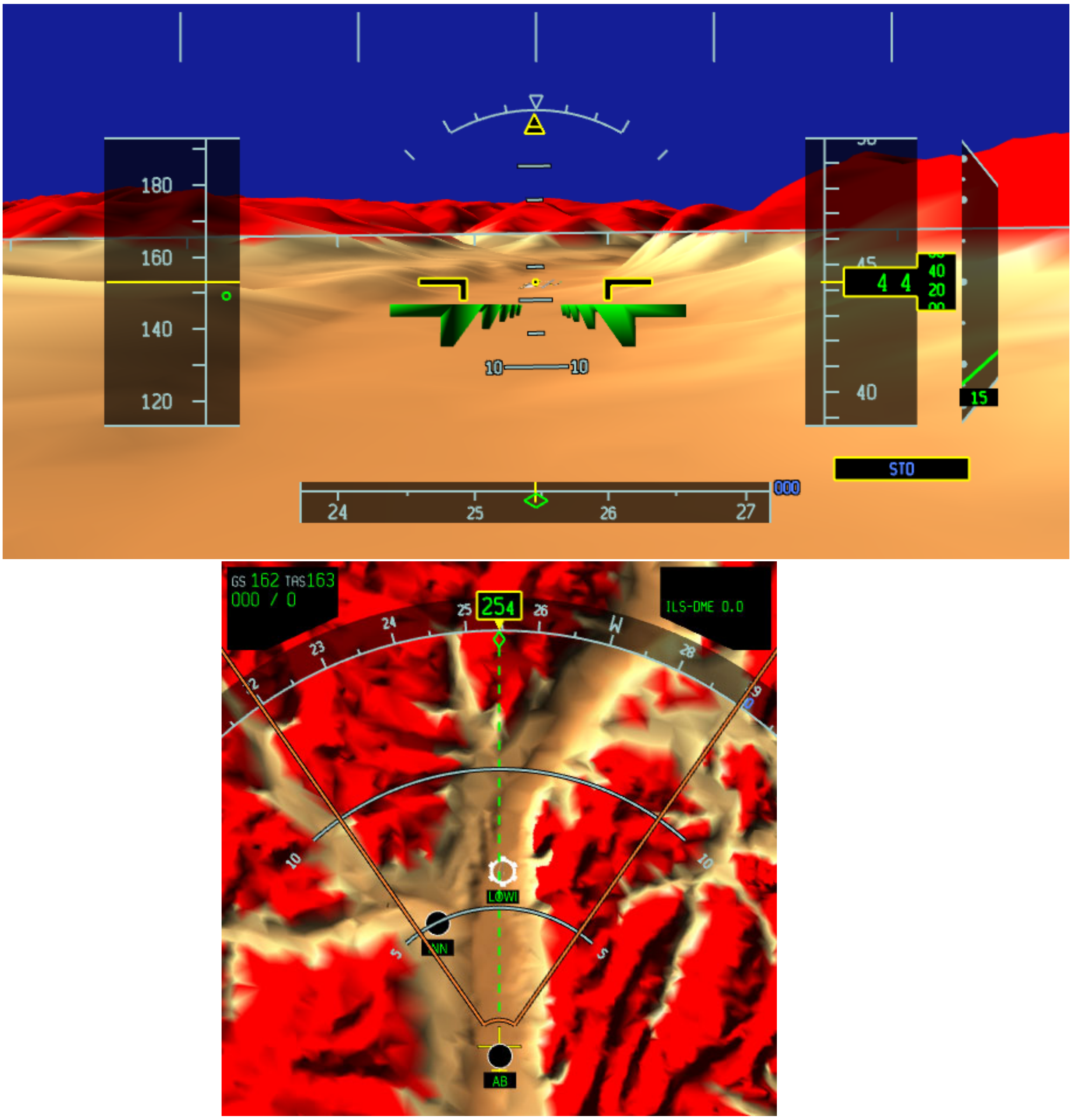


Abb. 4.34: TIN-Modell

- Schichtenmodell: Die Vektoren des Geländes werden auf diskreten Höhenstufen zusammengezogen. Dabei müssen zum Teil neu Vektoren aus vorhandenen interpoliert werden. Zum Beispiel werden alle Punkte zwischen 75 und $125m$ unter einem grafischen Knoten auf $100m$ Höhe zusammengefasst. Daher entstammen die Punkte nicht unbedingt direkt einem Rastermodell, sie sind teilweise nach einer Rechnung interpoliert. Dieses Modell benötigt einerseits weniger Punkte als das Rastermodell. Anderseits kann es sein, dass mehr Punkte als auf dem TIN-Modell generiert werden müssen. Im Gegensatz zum TIN Modell kann man aber die Punkte relativ einfach für die Optimierung des Renderings ordnen. Die Punkte sind bereits in Höhenschichten sortiert, man braucht sie nur hintereinander dem Kurvenverlauf folgend weiter zu ordnen. Die Optimierung ist nicht vergleichbar mit der des Rastermodells, weil die Punkte in dem Schichtenmodell nur für jede Schicht geordnet werden. Im Rastermodell sind sie jedoch für den gesamten Teil des Geländes richtig geordnet. Das Schichtenmodell benutzt aber normalerweise weniger Dreiecke. Außer der Leistung spricht die Qualität der Darstellung für das Schichtenmodell: Die Kontur des Geländes wird hier deutlicher dargestellt (siehe [Hel00]).

  Für dieses Modell wird das *Gouraud Shading* innerhalb jeder Schicht, jedoch nicht zwischen zwei Schichten benutzt. Zwei Schichten sind getrennte Objekte und es scheint so, als ob *Flat Shading* zwischen zwei Schichten verwendet würde. Der gestufte Farbverlauf verbessert hierbei die Erkennbarkeit des Geländes (siehe [Hel00]).

- Konturlinien-Modell: Dies ist eine Erweiterung des Schichtenmodells, bei dem die Grenze einer Schicht zu einer Linie ausgeprägt ist (siehe [Hel00]). [Web00] beschreibt Algorithmen, um Konturlinien aus einem Schichtenmodells und LOD zu geniereren. Die Punkte der Konturlinien sind bereits im Schichtenmodell definiert. Es werden keine zusätzlichen Punkte mehr benötigt. Da nicht nur Dreiecke dargestellt, sondern auch teilweise die Grenzen der Dreiecke dargestellt werden, wird mehr Zeit für die Berechnung dieses Modells in vergleich zum Schichtenmodell benötigt.

Einige Symbole müssen in der Darstellung eine Dynamik haben. Ein Funkfeuer zum Beispiel besitzt zur Identifikation seine Morsekennzeichnung und die zugehörige Frequenz. Diese Informationen werden auf einer zugehörigen orthogonalen Anzeige dargestellt. Sie sind zur Zeit in der Geländedatenbank enthalten und mit dem Funkfeuerssymbol verbunden. Erstens werden diese Informationen nur auf der orthogonal Anzeige dargestellt. Das

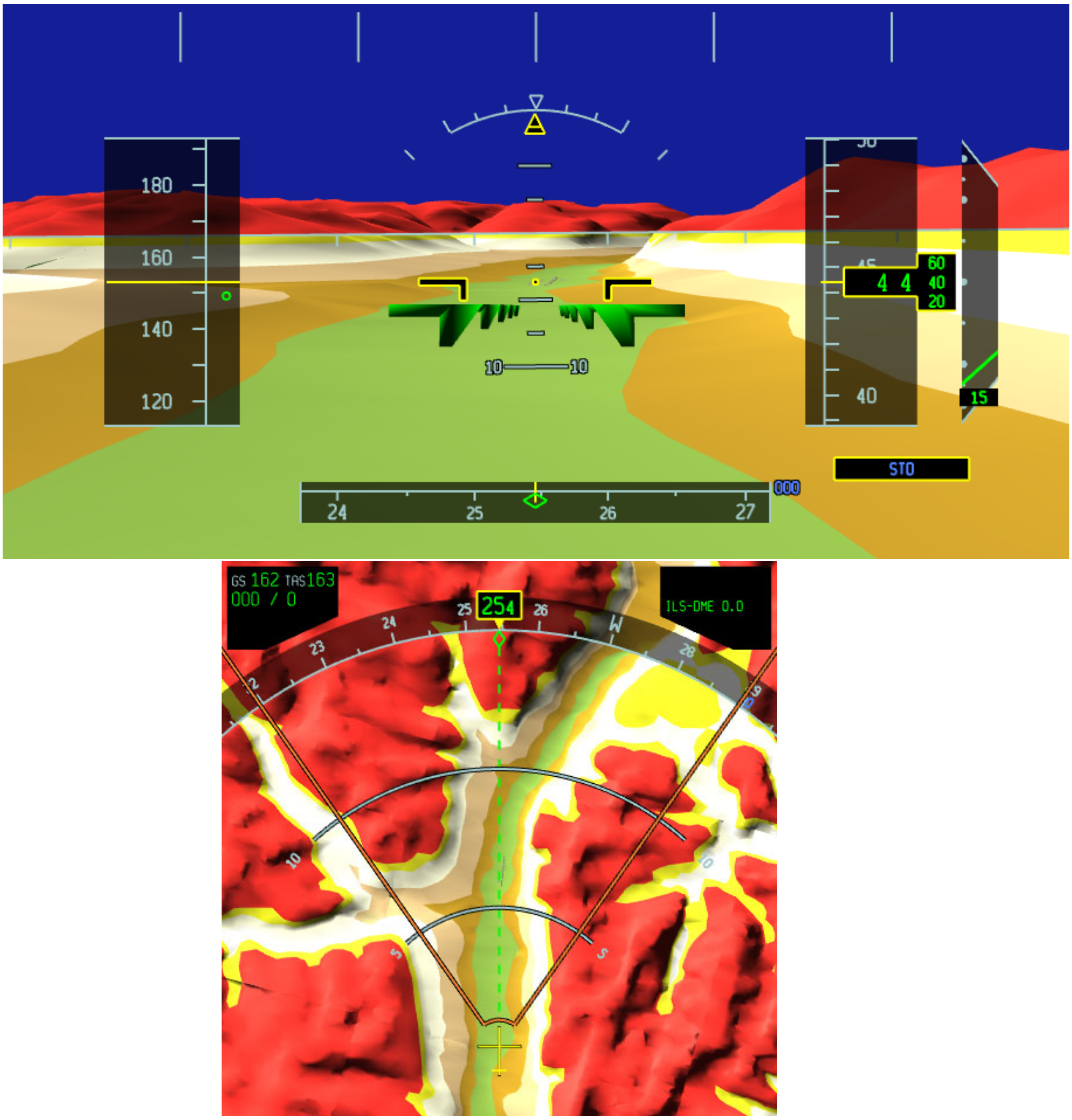


Abb. 4.35: Schichtenmodell

bedeutet, dass man eine Darstellung in der Datenbank für die orthogonale Anzeige, und eine andere für die perspektivische Anzeige braucht.

Wenn das Flugzeug seinen Kurs ändert, muss sich das Gelände mit den Funkfeuern auf der Anzeige drehen. Der Text der Funkfeuer muss jedoch immer eine konstante Richtung relativ zum Bildschirm haben. Er dreht sich deshalb entgegen der Geländedrehung. Sonst würde es bedeuten, dass beim Flug in Nordrichtung der Text lesbar wäre, aber dass in Südrichtung der Text auf dem Kopf würde, so dass er nahezu unlesbar wäre.

Dieses Beispiel zeigt, dass es im Geländeteil Symbole gibt, die eine Dynamik haben. Bezüglich der Informationen über die Dynamik, gibt es hier zwei Alternativen:

- Entweder weiss die Anzeigesoftware selbst, was die Symbole bedeuten und welcher Dynamik sie unterliegen,
- oder die Anzeigesoftware weiss nichts über die Bedeutung der Symbole, bekommt aber von der Datenbank Informationen oder Befehle über die anzuwendende Dynamik.

Die erste Lösung ist gänzlich unflexibel: Wenn man die Dynamik eines Symbols ändern möchte, muss man die Software ändern. Im Gegensatz dazu erlaubt die zweite Lösung mehr Flexibilität. Die Dynamik sollte in der Datenbank implementiert sein, mit dem Hinweis, welcher Teil des Geländes betroffen ist. Außer bei der Einführung eines zusätzlichen Befehls, braucht man die Software nicht zu ändern, wenn die Dynamik eines Elements geändert wird. Man braucht nur die Information in der Datenbank zu ändern.

Daher wird die Informationen über die Dynamik in der Datenbank abgelegt. Um die zugehörigen Befehle von der Datenbank zu lesen, und um auf diese Dynamik zu reagieren, ist eine Interpretation der Informationen in der Anzeigesoftware nötig. Dies wird in 4.7.8 erläutert.

Verwaltung der Simultanverarbeitung mit der Klasse exchangeDB

In 4.7.8 waren die Vorteile der Simultanverarbeitung und ihre Anforderungen erläutert worden. Dieses Unterkapitel wird die Implementierung erläutern.

Die 3D-Anzeigen definieren eine *Call-Back*-Methode für die *Application-Phase*. Diese Methode fragt unter anderem die aktuellen Koordinaten des Flugzeuges ab. Sie leitet diese Information an eine Datenbankbibliothek weiter. Diese Bibliothek gibt dann ein Feld zurück, das Pfad und Namen von neuen Dateien, die die Software lesen muss, enthält.

Eine Instanz der Klasse *exchangeDB* wird im *shared Memory* generiert. Diese Instanze bekommt von der *Application-Callback*-Methode der Anzeige die Information über die Aufgabe des Datenbankprozesses. Wenn eine neue Kachel - ein Teil des Geländes - geladen werden muss, wird *exchangeDB* die Information für den Datenbankprozess (dieser Prozess ist vom Hauptprozess getrennt) speichern. Zusätzlich wird eine weitere Information sofort gespeichert: Die neue Kachel steht noch nicht für den Hauptprozess zu Verfügung.

Auf der Seite des Datenbankprozesses wird die Klasse *exchangeDB* immer abgefragt. Wenn eine neue Kachel benötigt wird, muss dieser Prozess die Kachel auslesen. Wenn die Ladeaufgabe beendet ist, muss der Datenbankprozess die Klasse *exchangeDB* informieren: Wenn die Ladeaufgabe problemlos war, wird die neue Kachel für den Hauptprozess im *Shared Memory* gespeichert und eine Meldung geschickt. Da der Ladeprozess lange dauert, kontrolliert *exchangeDB*, ob die geladene Kachel die richtige ist. Nur in diesem Fall steht die Kachel wirklich für den Hauptprozess zu Verfügung. Dies wird durch eine Meldung in *exchangeDB* von der Datenbank zum Hauptprozess erreicht (siehe Abb. 4.36).

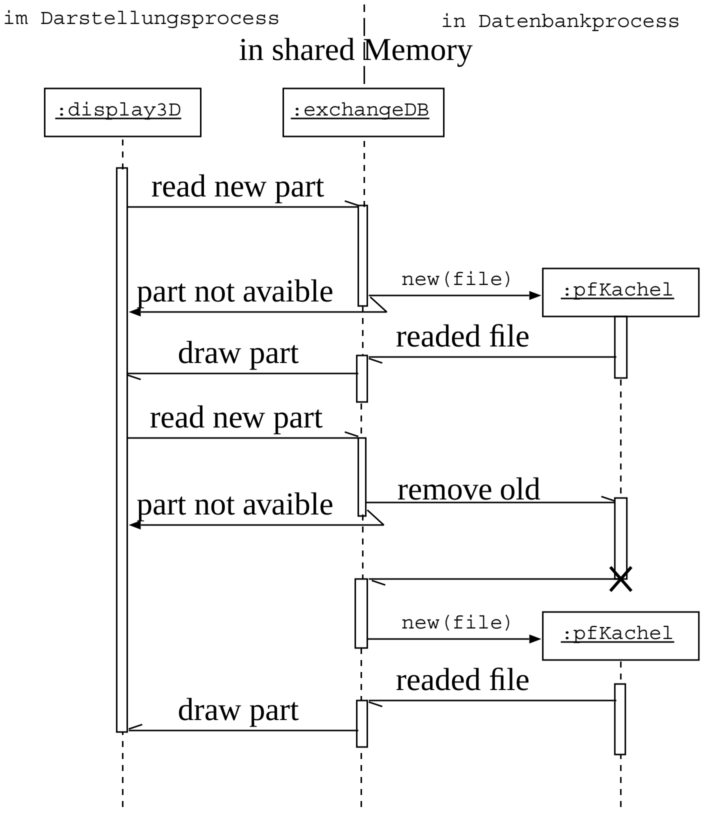


Abb. 4.36: Sequenzdiagramm: Verwaltung der Simultanverarbeitung

Es gibt so viele Instanzen der Klasse *exchangeDB*, wie Kacheln gezeichnet werden. Die Anzahl der Kacheln ist durch die Performance begrenzt. Die Klasse *exchangeDB* regelt also den Informationaustausch zwischen den beiden Prozessen und behält die Informationen über die Kachel. Die Instanz der Klasse *exchangeDB* muss während der ganzen Laufzeit der Software immer erhalten bleiben. Dies ist nicht der Fall für den Inhalt. Der Inhalt wird im nächsten Unterkapitel erläutert.

Verwaltung von Geländedateien mit der Klasse *pfKachel*

Das Gelände ist aufgeteilt in einzelne Kachelabschnitt. Jeder Abschnitt des Geländes wird aus einer *Open Inventor*-Datei gelesen. Die Klasse *pfKachel* wurde programmiert, um die Datei zu lesen. Diese Klasse erweitert den *Open Inventor*-Übersetzer von *OpenGL Performer*. Im Flug bewegt sich das Gelände und das Flugzeug ist ortsfest. Deswegen werden die Kacheln mit einem dynamischen Knoten modeliert. Dazu erbt *pfKachel* von der Klasse *pfDCS*.

Das World Geodetic System 1984 (WGS84) ist das weltweite Standardsystem, um jeden Punkt der Erde zu referenzieren. Dieses System modelliert die Erde als Geoid oder Ellipsoid, jeder Punkt auf der Erde wird durch zwei Winkel und seine Höhe definiert. Die zwei Winkel sind der geografische Längen- und Breitengrad. Mit diesem Winkel ist ein Punkt P_{geoid} auf einem Ellipsoid definiert. Damit wird eine Gerade durch diesen Punkt und durch den Erdmittelpunkt definiert. Der richtige Punkt auf der Erde P_{erde} ist dann durch einen Wert definiert, der aus dem Abstand zwischen den zwei Punkten P_{geoid} und P_{erde} auf der vordefinierten Gerade berechnet wird. Der Abstand $h = P_{erde} - P_{geoid}$ ist die Meeresspiegelhöhe des Punktes.

Dieses System referenziert also jeden Punkt mit zwei Winkeln und einer Entfernung. *OpenGL Performer* benutzt hingengen nur kartesische Koordinatensysteme. Das bedeutet, dass man die WGS84 Koordinaten in ein kartesisches System umrechnen muss. Solche Berechnungen für jeden Punkt in der Laufzeit oder in der Ladephase sind unannehmbar, sie würden zu viel Zeit benötigen.

Wenn jeder Punkt einer Kachel eine relative kartesische Koordinate zu einem Referenz-Punkt der Kachel hätte, wäre es nur nötig, die Berechnung dieser Referenz der Kachel zur Laufzeit durchzuführen. Dies ist zur Zeit noch eine Anforderung an die Datenbankentwickler. Die Koordinaten in der Datenbank, die für die Flugführungsanzeige zur Verfügung stehen, sind in tausendstel Bogensekunden gespeichert.

Der Verwalter der Kacheln, die Klasse *pfKachel*, muss die Kachel-Koordinaten in kartesische Koordinaten umrechnen.

Eine Methode der Klasse *pfKachel* berechnet abhängig vom aktuellen Breitengrad zwei Skalierungsfaktoren N und M. Diese Methode vereinfacht die Berechnung der kartesischen Koordinaten eines Punktes des Geoides.

Diese Skalierungsfaktoren N und M skalieren die gesamte Geländedarstellung. Diese Skalierung für die ganze Darstellung wird die Performance nicht derart verschlechtern wie eine Berechnung, die für jeden Punkt realisiert werden müsste. Exakt stimmt diese Berechnung allerdings nur für den überflogenen Punkt. Diese Umrechnung wird als vorläufige Lösung benutzt. Es ist eine Näherung, die in der Nähe der Pole nicht mehr ausreicht.

Verwaltung von Dynamik in der Datenbank

Die Klasse *pfKachel* muss nicht nur die Struktur des Geländes, sondern auch die Dynamik der Elemente aus der Datenbank lesen und speichern, damit der Hauptprozess auf diese Dynamik reagieren kann. Die Klasse *pfKachel* enthällt die Verwaltung der Datenbank-Dynamik.

Für die Anforderungen an die Datenbank war das Beispiel der Beschriftung eines Textes eines Funkfeuers gewählt worden: Man konnte sehen, dass es auf einer Kachel einige Symbole gibt, die eine konstante Richtung in der Darstellung auf dem Display aufweisen müssen. Um Zeit zu sparen, darf es nicht sein, dass man diese Symbole für jedes Bild im grafischen Baum suchen muss.

Um die konstante Richtung zu gewährleisten, wird eine Liste von Zeigern auf alle die Symbole erstellt, die eine entsprechende Dynamik besitzen. Diese Liste wird beim Laden erzeugt. Alle Knoten, die eine konstante Richtung haben müssen, haben in den Datenbank-Dateien einen Kommentarknoten. Im Kommentarknoten ist die Dynamik als Wort, für die Drehung *mupnN*, und einem Wert, hier 1 gespeichert, wenn das Element eine konstante Richtung haben soll. Damit auch mehrere Dynamiken gespeichert werden können, sind die Texteinträge der Kommentarknoten durch Semikola getrennt:

mupnN=1;scanN=0.025

ScanN wird für eine konstante Größe benutzt. Der Wert stellt die gleichbleibende Größe der Elemente in normierten Fensterkoordinaten dar.

Die Klasse *pfKachel* muss jeden Kommentarknoten lesen und interpretieren.

Am Beispiel des Kommandos *mupnN* wird nun die Implementierung der Datenbank-Dynamik erläutert.

Um die konstante Ausrichtung einzuhalten, ruft der Hauptprozess eine *callback*-Methode auf. Diese Methode muss den aktuellen Kurs abfragen, und jedes Element der Symbol-Liste in die Gegenrichtung drehen.

Die Klasse *pfFLMupnN* verwaltet Elemente, die immer mit einer konstanten Richtung gezeichnet werden sollen. Wenn *pfKachel* während des Ladeprozesses einen neuen Knoten findet, der unter anderem die Richtung beibehalten soll, wird ein Zeiger von *pfFLMupnN* in seine verkettete Liste eingefügt.

Dieses Prinzip wird für jede Dynamik angewandt. Die Klasse *pfFLCommand* ist die Basisklasse der Dynamikverwalter. In dieser Basisklasse wird für jeden Prozess, *Application-*, *Culling- und Drawingprozess* eine Defaultcallbackmethode vordefiniert. Diese Methoden dürfen nicht pur virtuelle sein: Beispielsweise überschreibt die Unterklasse *pfFLMupnN* nur die Methode für den *Drawing* Prozess, die beiden anderen Methoden haben für *pfFLMupnN* keine Implementierung (siehe Abb. 4.37).

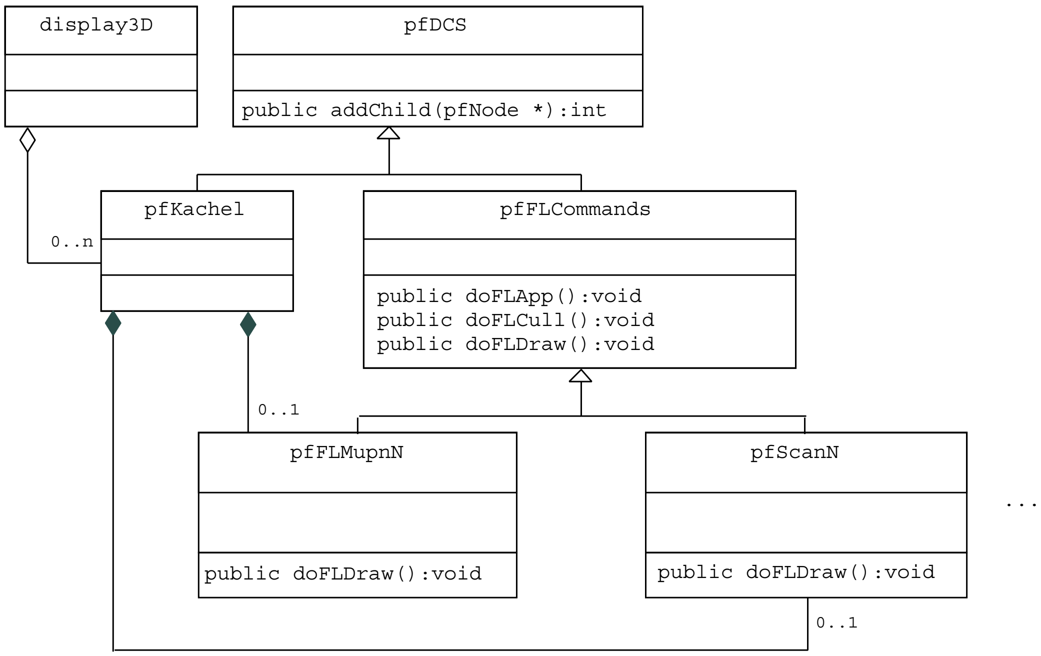


Abb. 4.37: Klassendiagramm der Verwaltungsklasse für die Datenbanklogik

Eine Kachel hat also zwei Bäume: Einen grafischen Teil, und einen logischen Teil, der mit den Unterklassen von *pfFLCommand* erzeugt wird. Dieser zweite Baum wird nie dargestellt, beeinflusst aber die Darstellung des grafischen Baums, denn seine Dynamik ändert die Darstellung des grafischen Baums. (siehe Abb. 4.38).

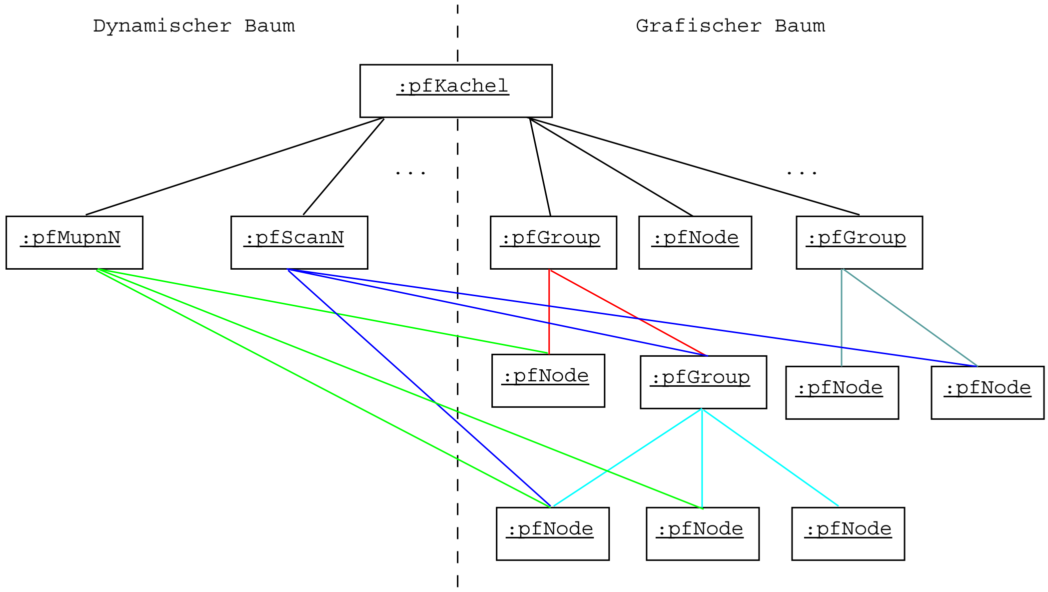


Abb. 4.38: Dynamischer und grafischer Baum

4.8 Kommunikationsschnittstelle

Es wurde erwähnt, dass Parameter für die Flugführungsanzeigen-Anwendung mit anderen Programmen ausgetauscht werden. Die Programme laufen nicht unbedingt auf demselben Rechner. Die Kommunikation der Anwendungen muss daher über das Netzwerk erfolgen. Die Anforderungen der Kommunikationsschnittstelle wurden in Kapitel 3.3 erläutert. Die Realisierung wird hier nach der Analyse von fertigen Systemen erklärt.

4.8.1 Analyse verteilter Kommunikationsstrukturen

Es gibt fertige Architekturen, um Werte zwischen verschiedenen Rechnern auszutauschen. Die bekannten Lösungen basieren auf einem Client/Server System. Hier wird eine kurze Erläuterung solcher Lösungen mit ihren Haupteinschränkungen gegeben.

Das Client-Server-Modell basiert auf einem einfachen, verbindungslosen Frage-/Antwort-Protokoll, das den beträchtlichen Overhead der verbindungsorientierten Protokolle wie etwa OSI oder TCP/IP vermeiden soll. Der Client sendet eine Nachricht an den Server und fordet einen Service an. Der Server erledigt diese Arbeit und liefert entweder die gewünschten Daten zurück, oder einen Fehlercode, falls die Anfrage nicht ausgeführt werden konnte [Tan95].

Um solche Dienste zu erfüllen, wurden Protokolle über der Schicht des Internet Protokolls (IP) entwickelt. Das RPC[11] Protokoll, oder das RMI[12] Protokoll, für OOP[13]-Anwendungen (siehe [PMG97]), ermöglichen einen entfernten Funktions- bzw. Methodenaufruf. Jeder Server im Netz stellt im Rahmen dieses Konzeptes eine Anzahl von Diensten zur Verfügung, die mit RPC angefordert werden können. Diese Funktionen sind als Prozeduren eines Programmes realisiert und können unter Angabe von Serveradresse, Programmnummer und Prozedurnummer angesprochen werden [Lip99] (siehe Abb. 4.39).

Zum Beispiel benutzt CORBA[14] normalerweise das RPC. Dies bietet dem Entwickler die Möglichkeit, ein verteiltes System mit Hilfe des objektorientierten Ansatzes zu modellieren und umzusetzen [GGM99]. Abbildung 4.39 zeigt die CORBA-Implementierung der Anfrage eines Dienstes.

Um *remote*-Funktionen[15] aufzurufen, braucht man spezielle Funktionen, die *Stubs*, die die

---

[11] Remote Procedure Call

[12] Remote Method Invocation

[13] Objektorientierte Programmierung

[14] Common Object Request Broker Architecture

[15] Funktionen, die auf einem anderen Rechner ausgeführt werden

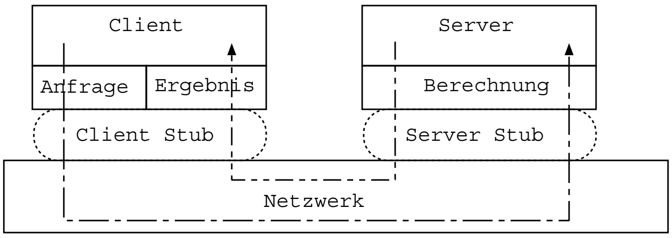


Abb. 4.39: Remote Procedure Calls

Informationen für den Aufruf und die Rückgabeparameter der *remote*-Funktion vorbereiten. Die Aufgaben des Client Stub sind [XI95]:

- Spezifikation der aufgerufen Prozedur, Zuordnung des Aufrufes zum Zielrechner,
- Darstellung der Parameter im Übertragungsformat,
- Dekodieren der Ergebnisse und Übergabe an den Client,
- Blockierung des Client aufheben. In Verbindung mit Threads ist allerdings eine asynchrone Realisierung eines *remote*-Funktionsaufrufs möglich [Sys].

Die Aufgaben des Server Stub sind folgende [XI95]:

- Dekodieren des Aufrufs und der Parameter, Bestimmung der Aufrufadresse der Prozedur (z.B. mittels einer Tabelle),
- Kodieren der Ergebnisse und Aufrufkennung,
- Quittung, Routing und Wiederholung von Übertragungspaketen.

Der Client Stub wird bei Corba *IDL*[16] *Stub* genannt, und der Server Stub, *IDL Skeleton*. Zwei wichtige Aspekte der Architektur werden im folgenden gezeigt [Sie96]:

- Sowohl die Implementierung des Clients und die des Objekts sind (im Fall des Servers) vom ORB[17] durch die IDL-Schnittstelle getrennt. Die Clients sehen nur die IDL-Objekte, nie direkt die Details der Implementierung. Dies gewährleistet die Austauschbarkeit der Implementierung unterhalb der Schnittstelle.

---

[16] *Interface Definition Language*

[17] Object Request Broker

- Eine Anfrage eines Client kann nicht direkt beim Objekt ankommen, sie muss immer vom ORB verwaltet werden. Die Form der Anfrage ist immer die gleiche, egal ob das Objekt lokal oder entfernt ist.

Die Schnittstellendefinition eines IDL-Objekts beschreibt für den Dienst die Operation, die Eingangs- und Ausgangsparameter sowie Ausnahme-ereignisse, die während der Berechnung entstehen könnten. IDL ist also eine *Absprache* mit den Clients eines Objekts. Die Clients und die Objekte sind voneinander durch drei Komponenten getrennt: einen *IDL Stub* bei dem Client, einen oder mehrere ORB und einen entsprechenden *IDL Skeleton* beim Objekt selbst. Es ist nicht möglich, bei CORBA direkt zur Implementierung eines Dienstes einen Server zu erreichen, es durchläuft immer Komponenten. Man muss die Client und Server Stubs wie zusätzliche Netzwerkschichten sehen. Wenn zwei Rechner etwas austauschen müssen, müssen die Informationen durch jede Schicht laufen, beim Absender wie beim Empfänger, wobei die Schichten beim Empfänger in der entgegengesetzten Richtung durchlaufen werden.

Diese Eigenschaft von CORBA widerspricht im Fall der Flugführungsanzeige der Forderung danach, dass Anforderungen von der Umgebung an die vorliegende Software gemacht werden, und nicht umgekehrt: Wenn diese verteilte Architektur benutzt wird, müssen andere Anwendungen, die beispielsweise Werte berechnen, mindestens das IDL-Skeleton benutzen, um Werte zu liefern.

Dasselbe Problem tritt auch bei anderen verteilten Kommunikationsstrukturen, wie dem HLA[18] System (siehe [HLA]) auf.

Unter den gegebenen Anforderungen an die Kommunikationsschnittstelle ist daher eine Eigenentwicklung notwendig. Das nächste Kapitel wird die Implementierung der Kommunikationsschnittstelle der Anzeigen-Anwendung erläutern.

4.8.2 Implementierung der Schnittstellenarchitektur

Die Informationen sind in verschiedenen *Strukturen* zusammengefasst. Natürlich würde die Verwendung einer klassischen C-Struktur zu einigen Unflexibilitäten oder Schwierigkeiten führen: Zunächst müsste man die Struktur für ein Projekt jedesmal neu definieren, wenn sich bei einem bestimmten Protokoll die Reihenfolge der Werte in der *Struktur* oder der Typ eines Wertes ändert. Deswegen erscheint eine Struktur besser, die nicht die Werte

---

[18] High Level Architecture

selbst, sondern Zeiger auf die Werte enthält. Solche *Strukturen* sind von der endgültigen Reihenfolge der Werte, die über das Netz ausgetauscht werden, unabhängig.

Die Anzeige-Anwendung benötigt mehrere Strukturen: Unter anderem eine Struktur für den Flugzustand und eine andere für Werte, die die Anzeige steuern (Zoomstufe, Positionierung der Skalen auf dem Bildschirm usw.). Diese verschiedenen Strukturen werden ihrerseits in einer verketteten Liste zusammengeführt. Jeder einzelne Wert, beispielsweise *positionX*, wird durch zwei Zeiger angesprochen: Der erste, der auf die entsprechende Struktur in der Liste verweist, der zweite, der auf den Wert innerhalb dieser Struktur verweist. Die Werte sind daher in einer Art Baum organisiert.

Hier wird eine sehr flexible Lösung zur Erzeugung dieser Strukturen gesucht. Eine Möglichkeit bestünde darin, die Struktur der Werte mit externen Dateien zur Laufzeit zu generieren. Die Anzeige-Anwendung würde dann eine Strukturbeschreibungsdatei lesen, in der die Namen der Werte (z.B. *positionX*) und ergänzende Informationen stünden, um damit die Zeiger auf die Werte setzen zu können.

So wäre es möglich, durch Änderung an einer Textdatei die Struktur zu ändern, ohne die Anwendung neu zu kompilieren. Wenn eine Funktion einen Wert benötigte, würde man einer Lesemethode den Namen und den erwarteten Rückgabetyp übergeben. Ein solcher Aufruf könnte zum Beispiel wie folgt lauten:

```
int positionX = get(“positionX”,typeInt);
```

Diese Lösung wäre sehr flexibel, hätte aber auch wesentliche Nachteile.

Zum einen dürfte eine vergleichsweise schlechte Performance erwartet werden, da zur Laufzeit, und nicht etwa in der Inititialisierungsphase der Software, der Baum nach den Namen der Werte, also nach Zeichenketten durchsucht werden müssen. Diese Suche müsste fehlertolerant sein, da nicht sichergestellt werden kann, dass der gesuchte Name im Baum *exakt* der Schreibweise in der Software entspricht, beispielsweise durch Tippfehler.

Die folgenden zusätzlichen Probleme treten zwar prinzipiell bei der Erstellung der Strukturen auf, können aber mit der beschriebenen Variante nur sehr schwierig gelöst werden:

- Es gibt Werte, die nicht in jedem Protokoll existieren. Um die Struktur herzustellen, muss man jedoch für jeden Wert einen Zeiger setzen. Jede Lösung muss daher einen Algorithmus enthalten, um diese Anforderung zu erfüllen, indem man z.B. den Zeiger für einen Wert, der im Protokoll nicht existiert, auf einen vordefinierten Wert umlenkt. In der beschriebenen Variante würde sich die Komplexität der Strukturbeschreibungsdatei dadurch beträchtlich erhöhen.

- Es gibt für bestimmte Protokolle Formatänderungen, z.B. Dezimalzahl in ganze Zahl. Die Strukturbeschreibungsdateien müssten solche Konvertierungen beschreiben können.

- Eine zusätzliche Logik für die Konvertierung einzelner Werte muss implementiert werden. Zum Beispiel könnte man einen Boolean benutzen, um zu beschreiben, ob das Flugzeug auf dem Boden ist oder nicht. Als Standard könnte man entscheiden, dass der Wert *True* ist, wenn sich das Flugzeug auf dem Boden befindet. Sollte ein Projektpartner einen anderen Standard verwenden (ist bei ihm dieser Wert *True*, wenn das Flugzeug fliegt), ist eine Konvertierung notwendig. Es ist sehr schwierig, eine entsprechende Konvertierungsvorschrift in die Strukturbeschreibungsdatei zu integrieren. Es gibt andere Konvertierungen, die viel komplexer sind als die des obigen Beispiels. Die Modelldatei müsste solche komplexen Konvertierungen beschreiben. Es ist weiterhin sehr aufwendig, eine Logik, die für alle denkbaren Konvertierungen gilt, auf diese Weise zu implementieren.

Aufgrund der schwerwiegenden Nachteile der beschriebenen Variante wurde in dieser Arbeit ein anderer Weg gegangen.

4.8.3 Systemüberblick der Kommunikationschnittstelle

Wie erwähnt werden die Werte in Strukturen zusammengefasst. In der vorliegenden Arbeit sind dabei die Adressen der Werte fest implementiert und jedes Protokoll erhält dafür eine eigene Implementierung (siehe Abschnitt 4.8.4 und 4.8.6). Für jede Struktur ist es möglich, für bestimmte Protokolle eine Übersetzungsmethode zu definieren, um die oben erläuterten Probleme zu lösen (siehe Abschnitt 4.8.6). Zwar erlaubt diese Lösung keine Änderung ohne erneute Kompilation, aber sie ist zur Laufzeit schneller (zumindest weil keine Suche im Baum nötig ist, um Werte zu liefern). In 4.8.10 wird erläutert, wie man die Struktur trotzdem schnell generieren und aktualisieren kann.

Das Setzen der Zeiger innerhalb einzelner Strukturen wird von der Standardstruktur-Klasse übernommen (siehe 4.8.4). Das Setzen der Zeiger auf die einzelnen Strukturen des Baums erfolgt in der *communicationKernel*-Klasse (siehe Abschnitt 4.8.8).

4.8.4 Standardstruktur-Klasse

Die Standardstruktur-Klasse versammelt Werte, die in einem logischen Zusammenhang stehen. Diese Werte sind, wie erläutert wurde, mit Zeigern referenziert. Diese Klasse muss die Zeiger und die Inhalte initialisieren.

Initialisierung der Zeiger

Diese Klasse kann eine Methode haben, die diese Zeiger setzt. Normalerweise sollte der Konstruktor der Klasse diese Aufgabe erfüllen. Leider darf ein Konstruktor in C++ nie virtuell sein. Die Initialisierung der Zeiger muss notwendigerweise mit einer virtuellen Methode, *initPointer*, bewirkt werden. Dies ergibt als größten Vorteil, dass eine Unterklasse der Standardstruktur-Klasse die Voreinstellungsmethode *initPointer* überschreiben kann (siehe Abb. 4.42).

Die Methode *initPointer* ist virtuelle, und daher ist besondere Sorgfalt darauf zu verwenden, in jeder Unterklasse, die diese Methode überschreibt, jeden Zeiger richtig zu setzen. Um die Fehlertoleranz zu verbessern, wird an verschiedenen Stellen (unter anderem während der Initialisierung des Inhalts der Zeiger) überprüft, ob die Zeiger richtig gesetzt sind. Andernfalls wird dies als Ausnahme behandelt, die Software wird eine Fehlermeldung ausgeben und beendet. Diese Fehlerüberprüfung ist natürlich nur möglich, wenn der Konstruktor jeden Zeiger zuerst mit NULL initialisiert.

Initialisierung des Zeigerinhalts

Der Inhalt der Werte, auf die die Zeiger verweisen, ist noch nicht definiert. Dafür gibt es eine *reset* Methode, die die Parameter mit Voreinstellungswerten initialisiert. Es ist immer möglich, diese Methode auch außerhalb der Initialisierungsphase zu benutzen, um die Werte wieder auf ihren Voreinstellungsstand zu setzen. Damit die Voreinstellungswerte nicht fest implementiert sein müssen, liest die *reset* Methode eine Datei (ini file). Die Klasse *initFile* (siehe Kapitel 4.6.9) verwaltet diese Aufgabe: Sie sucht in einer Datei ein bestimmtes Wort und hinter dem Symbol $'' =''$ wird ein Wert gelesen, welcher der *reset* Methode zurückgegeben wird.

Abbildung 4.40 zeigt auszugsweise eine Standardklasse, *navigation* die den Flugzustand des Flugzeugs beschreibt.

4.8.5 Standardreader, standardwriter Klassen

Bis jetzt wurde nur die Grundstruktur einer Kommunikationsklasse definiert. Die Software muss die Werte lesen oder schreiben. Eine Unterklasse soll nun Methoden zur Verfügung stellen, die die Werte der Parameter liefern. Bei der Klasse für den Flugzugstand kann man z.B. unter anderem erwarten:

- eine Methode, die die Koordinaten liefert,

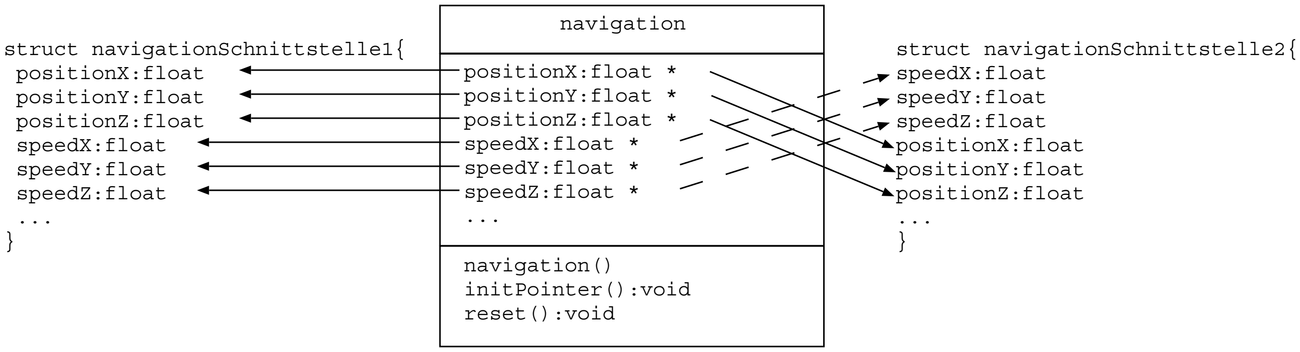


Abb. 4.40: Klassendiagramm der Standardstruktur

- und eine andere, die die Geschwindigkeit als Vektor ergibt.

Diese Unterklasse stellt die sogenannte Standardreader-Klasse (siehe Abb. 4.41) dar.

Damit diese Standardreader-Klasse noch unabhängig vom Protokoll bleibt, erbt sie von der *reader*-Klasse, die später in Abschnitt 4.8.8 erläutert wird, eine pur virtuelle Lesemethode, die *read* Methode. Später wird gezeigt, wie diese *read* Methode für jedes Protokoll oder jede Schnittstelle definiert wird.

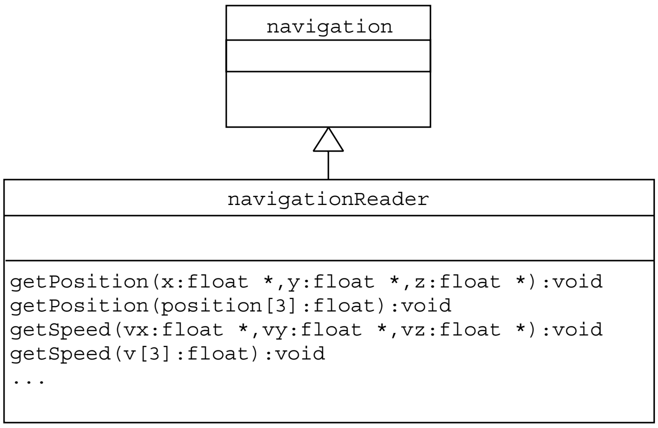


Abb. 4.41: Klassendiagramm des Standardreader

Eine zweite Klasse muss umgekehrt den von einer Software gelieferten Wert ins Netz schreiben, es handelt sich um die Standardwriter-Klasse. Die beiden Klassen ähneln sich stark, weil sie jeweils die gleichen Werte lesen und schreiben.

Wie bei der Standardreader-Klasse erbt eine Standardwriter-Klasse eine pur virtuelle Schreibmethode, die dann in einer Unterklasse definiert werden muss. Weil die Logik des writers und der des readers sehr ähnlich ist, wird im folgenden nur noch die reader-Seite betrachtet.

Außer im Schnittstellenteil kennt die ganze Software nur die Standardreader- und Standardwriter-Klassen zum Datenaustausch. Für die Software ist die Kommunikationsschnittstelle in einigen *Black Boxes* realisiert: Entweder liest man externe Werte nur mit einer Instanz einer Standardreader-Klasse, oder man schreibt Werte mit einer Instanz einer Standardwriter-Klasse. Der Rest der Implementierung der Kommunikation bleibt den Anwendern verborgen.

4.8.6 Schnittstellenstrukturreader Klasse

Für jede Schnittstelle und für jede Struktur gibt es eine Klasse, die die Werte vom Netz liest. Diese Klassen sind protokollspezifisch (siehe Abb. 4.42).

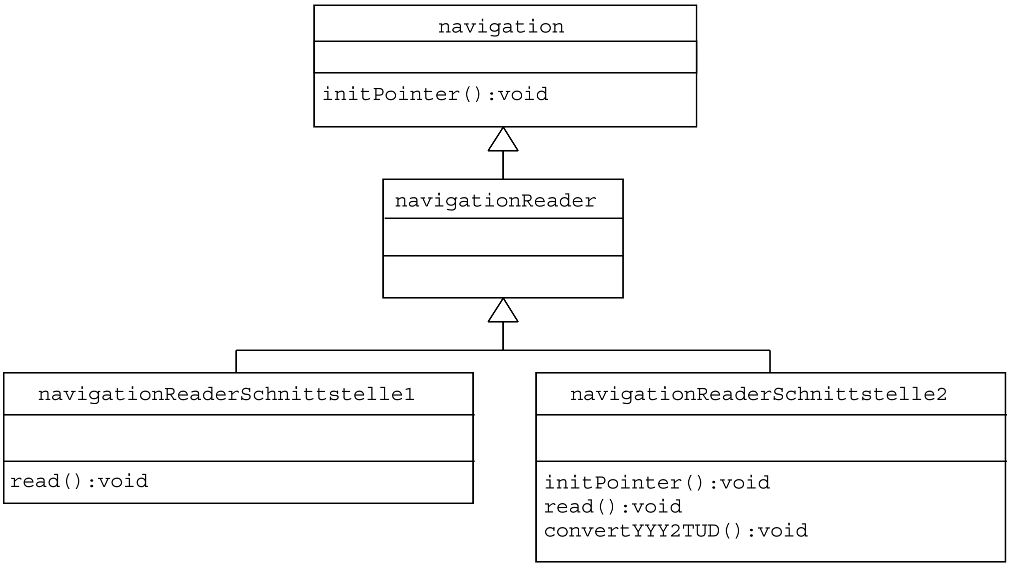


Abb. 4.42: Klassendiagramm des Schnittstellenstrukturreader

Wie erläutert wurde, muss für diese Klassen manchmal ein Übersetzer erzeugt werden: Man muss die Einheit und das Format der Werte an die Protokolleinheit bzw. das Format des Protokolls anpassen. Dies ist die Aufgabe der Methode *convertXXX2YYY*-, wobei XXX zum Beispiel das Standard-Format (TUD Format) darstellt und YYY ein anderes Format ist (siehe Abb. 4.42). Wenn die Methode *convertXXX2YYY* in der Schnittstellenstrukturreader-Klasse definiert ist, muss entsprechend die Methode -*convertYYY2XXX* auch in der Schnittstellenstrukturwriter-Klasse definiert sein.

4.8.7 Schnittstellenreader Klasse

Das Verfahren, die Schnittstelle eines Protokolls oder die Netzwerkkarte zu initialisieren, oder Werte aus dem Netz zu lesen, ist für jede Schnittstellenstrukturreader-Klasse einer bestimmten Schnittstelle nahezu identisch. Für jedes Protokoll wurde eine Klasse entwickelt, die diese Verfahren übernimmt. Diese *Schnittstellenreader-Klasse* (siehe Abb. 4.43) muss mit den zur Verfügung stehenden Protokoll-Funktionen arbeiten. Diese Klasse erkennt die Bedeutung der Werte, die sie verarbeitet, nicht. Sie *weiß* nur, *wie* sie diese Werte aus dem Netz verarbeiten muss. Sie ist eine Protokoll-reader-Klasse und hat eine direkte Verbindung mit dem Netzprotokoll. Alle Funktionalitäten zur Kommunikation, die im Protokoll nicht standardmäßig existieren, müssen in dieser Klasse implementiert sein.

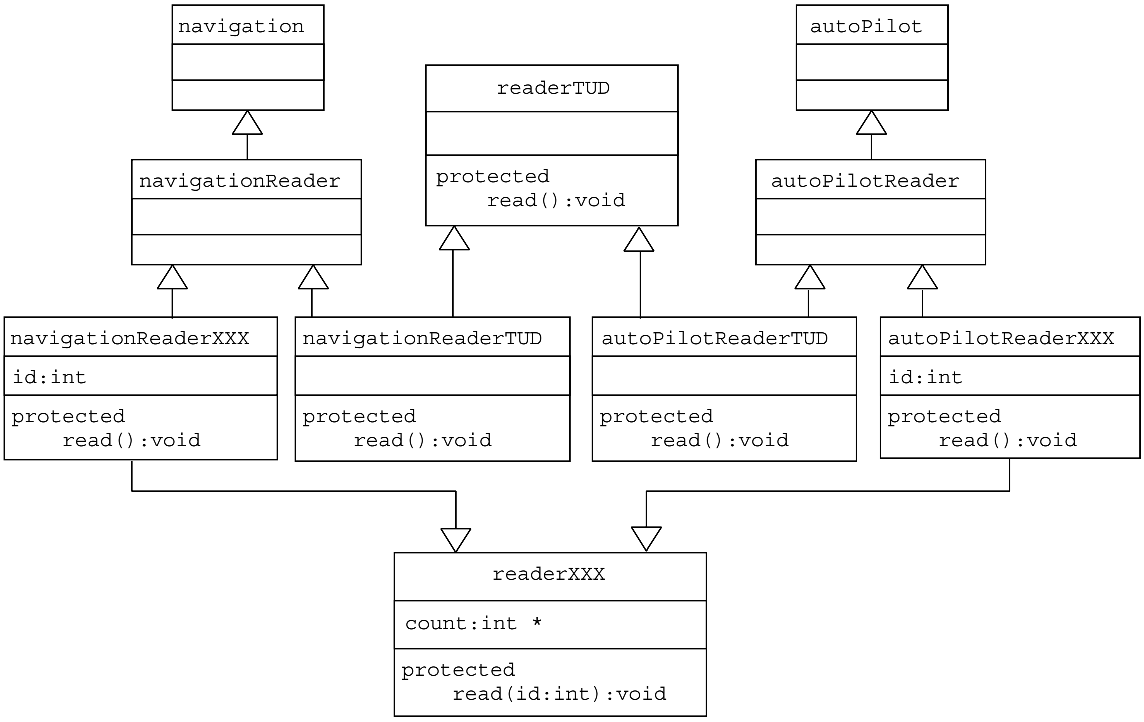


Abb. 4.43: Klassendiagramm der Kommunikationschnittstelle

Die Schnittstellenstrukturreader-Klasse erbt von mehreren Klassen. Einerseits erbt eine Schnittstellenstrukturreader-Klasse von einer Standardreader-Klasse, und anderseits von einer Schnittstellenreader-Klasse (siehe Abb. 4.43). Beispielsweise die Klasse *navigationReaderTUD*, sowie die Klassen *autopilotReaderTUD*, *navigationReaderXXX* und *autopilotReaderXXX* lösen die pur virtuelle Methode *read* der Klasse *reader* (siehe 4.8.8) auf durch einen Aufruf einer Methode des Schnittstellenreaders:

- Die *read*-Methode von *navigationReaderTUD* und *autopilotReaderTUD* rufen die *read*-Methode von *readerTUD* auf.
- Die *read*-Methode von *navigationReaderXXX* und *autopilotReaderXXX* rufen die *read* Methode von *readerXXX* auf. In diesem Beispiel braucht die *read* Methode des Protokolls XXX noch einen Identifizierungsparameter *id*. Der Grund dafür und die Nutzung des Identifizierungsparameters wird in 4.8.8 erläutert.

Man kann das gesamte Konzept als Erweiterung des Schichtenmodells sehen, wie es Abbildung 4.44 zeigt.

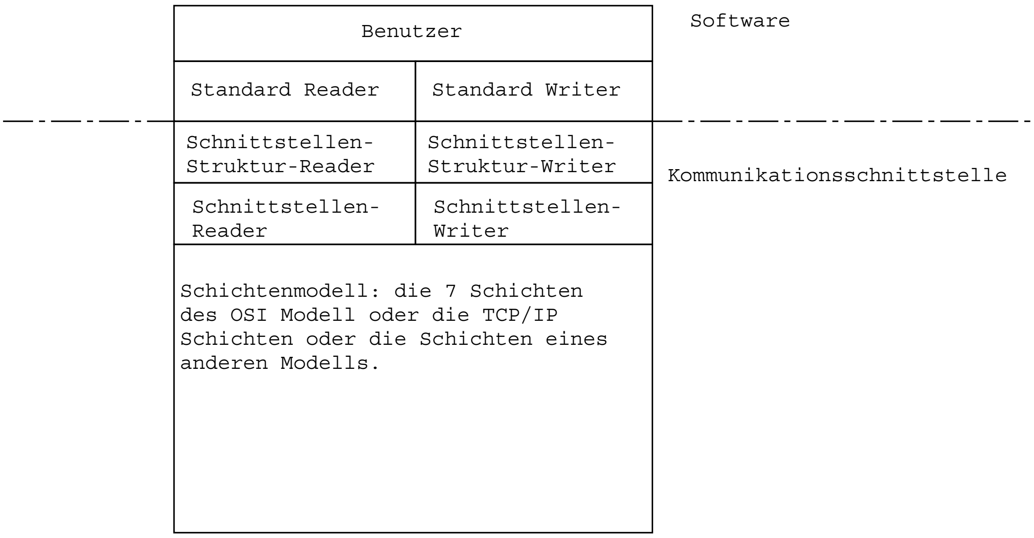


Abb. 4.44: Schnittstelle Konzept

4.8.8 Kommunikationskernel

Die Klasse *communicationKernel* bildet den Verwalter der gesamten Kommunikationsschnittstelle mit der Hauptforderung nach optimierter Performance. Die wesentlichen Probleme, mit denen diese Klasse dabei konfrontiert wird, sind folgende:

- Einzelne Werte der Standardstrukturen werden sicherlich in mehreren Methoden (oder Funktionen) der Applikation benötigt. Aufgabe des Kommunikationskernels ist zu vermeiden, dass von benötigten Schnittstellenstrukturreader mehrere Instanzen erzeugt werden.

- Es ist zu erwarten, dass verschiedene Funktionen stets die aktuellen Werte (vom letzten Update) benötigen. Es besteht die Gefahr, dass mehrere Entwickler in unterschiedlichen Methoden einen Leseprozess initiieren, um sicherzugehen, wirklich aktuelle Werte zu erhalten. Diese unkoordinierten Leseprozesse hätten folgende Nachteile:
  - zeitliche Inkonsistenz der Strukturen in unterschiedlichen Teilen der Software
  - und reduzierte Performance durch unnötige Leseprozesse.

Der Kommunikationskernel zentralisiert sämtliche Lese- und Schreibzugriffe und vermeidet so die beschriebenen Probleme.

reader-Klasse

Dieser Kommunikationskernel muss also wissen, welche Kommunikationsklassen schon instanziert wurden. Dafür hat er eine Liste von *readern* und eine Liste von *writern*. Eine verkettete Liste kann in *OOP* besonders zweckmäßig angewandt werden, wenn sämtliche enthaltenen Objekte auf einer einzigen Basisklasse beruhen. Dann ist es beim Durchschreiten der Liste möglich, *formal* immer die gleiche (virtuelle) read- oder write-Methode aufzurufen, ohne zu wissen, welches Objekt tatsächlich betroffen ist. Es wird daher für alle Strukturreader eine gemeinsame Basisklasse, die *reader*-Klasse, definiert. Die Instanzen einer Unterklasse von *reader* können schließlich in einer verketteten Liste angeordnet werden (siehe [Str00]). Eine *Strukturreader*-Klasse muss also von einer *standardStruktur* Klasse und einer *reader* Klasse erben (siehe Abbildung 4.45).

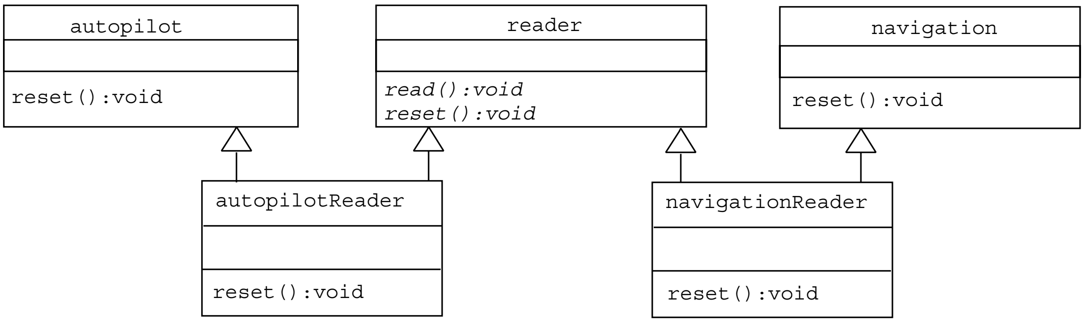


Abb. 4.45: Klassendiagramm Reader und Standardreader

Abbildung 4.46 verdeutlicht die Beziehungen zwischen dem Kommunikationskernel, den *readern*, den Strukturreadern, den *writern* und den Strukturwritern.

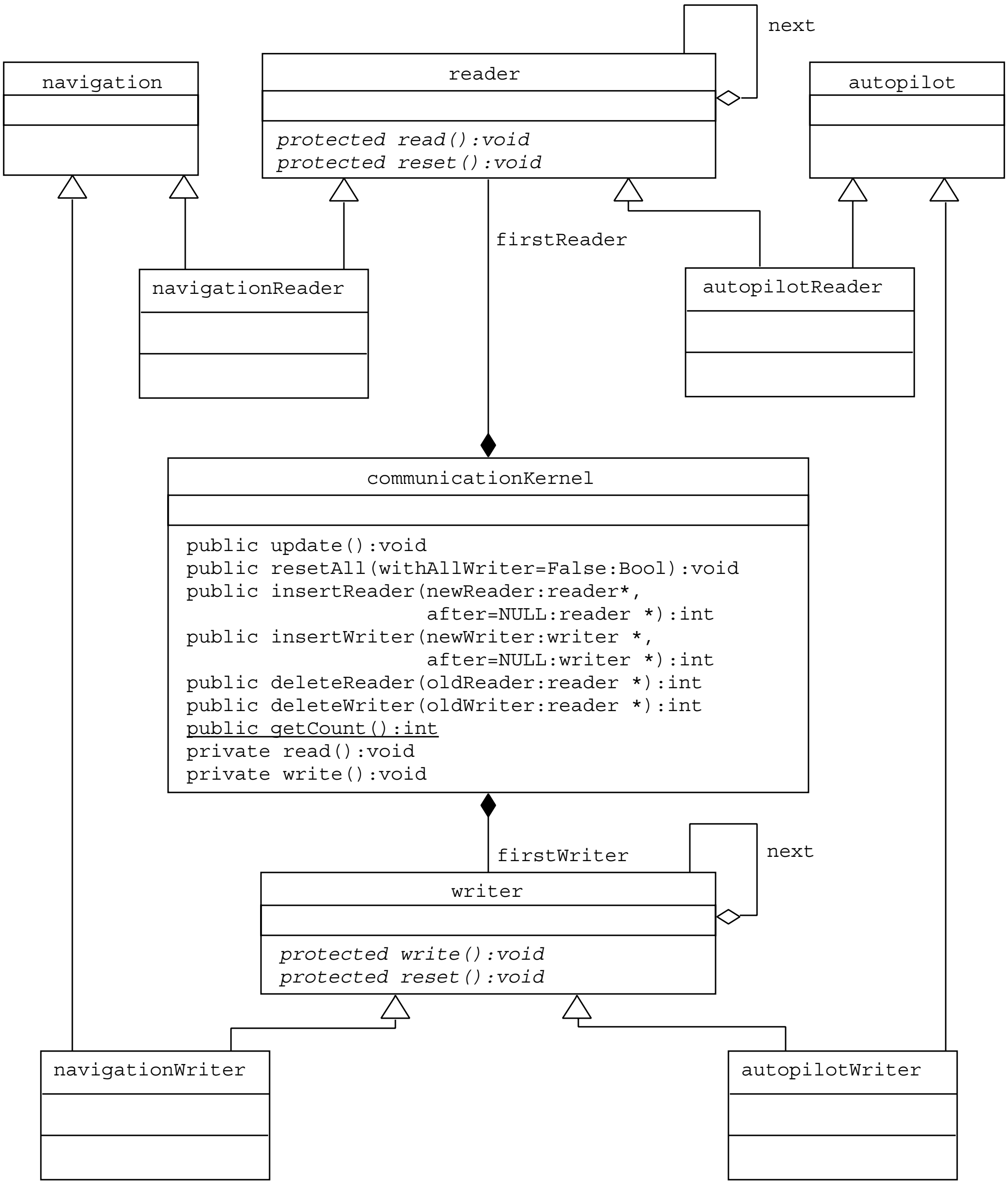


Abb. 4.46: Klassendiagramm des Kommunikationskernels

Die *reader*-Klasse muss eine virtuelle *read*- und *reset*-Methode haben. So ist es möglich, einmal pro Zyklus eine Aktualisierung für jede Klasse aufzurufen. Wenn die read-Methode *protected* ist, ist sichergestellt, dass die read-Methode nur von einer Instanz des *readers* oder von einer Unterklasse von *reader* aufgerufen wird und nicht durch eine externe Methode.

Der Kommunikationskernel ist keine Unterklasse der *reader*-Klasse. Daher könnte der Verwalter der Kommunikationsschnittstelle nicht die read-Methode aufrufen. Dies wäre problematisch, wenn es nicht die *friend*-Lösung in C++ gäbe. Eine *friend class* A einer Klasse B kann die privaten oder geschützten (*protected*) Teile der Klasse B benutzen und ändern. Man sollte diese Möglichkeit aber nur sehr vorsichtig einsetzen, da dies das Geheimnisprinzip einer Klasse untergräbt. Hier wurde es trotzdem verwirklicht: Der Kommunikationskernel darf alles von den von ihm verwalteten Objekten sehen. Mit dieser Möglichkeit wird die read-Methode bzw. write-Methode geschützt, und außer dem Kommunikationskernel kann niemand diese Methode aufrufen. So kann der Kommunikationskernel sicherstellen, dass die Strukturen einmal (und genau einmal) für jeden Zyklus gelesen werden. Er sichert auch die Konsistenz der Daten ab: Alle Daten werden zu einem bestimmten Moment des Zyklus gelesen, und sie entsprechen alle dem gleichen Stand.

Schnittstellenspezifische Strukturen

Bei manchen Kommunikationsschnittstellen werden einige Standardstrukturen in schnittstellenspezifischen, virtuellen Strukturen zusammengefasst. Diese schnittstellenspezifischen Strukturen werden mit einem Index, *identification*, gekennzeichnet. Die read-Methode einer Schnittstellenreader-Klasse (siehe 4.8.7) braucht, um solche schnittstellenspezifischen Strukturen zu lesen, diesen Index. Dazu liefert die Unterklasse Schnittstellenstrukturreader (siehe 4.8.6) ihre *identification* an die Schnittstellenreader-Klasse.

Der Kommunikationskernel ruft für jede einzelne Struktur eine Lese- bzw. Schreibmethode auf. Dies könnte jedoch dazu führen, dass einige der schnittstellenspezifischen Strukturen mehrfach gelesen bzw. geschrieben werden. Um dies zu vermeiden, besitzt der Kernel einen Zähler, *count*, der bei jedem Zyklus inkrementiert wird. Darüber hinaus verfügt jede Schnittstellenreader-Klasse über ein Feld von Zählern. Die Größe dieses Feldes hängt von der Anzahl solcher schnittstellenspezifischen Strukturen ab.

Wenn eine Schnittstellenreader-Klasse eine Struktur lesen muss, benutzt sie die *identification* als Index ihres Zählerfelds. Sie vergleicht den Inhalt des indizierten Zählerfelds mit dem Zähler des Kommunikationskernels. Wenn die beiden Zähler gleich sind bedeutet dies, dass diese schnittstellenspezifische Struktur bereits gelesen wurde und nicht noch einmal gelesen werden braucht. Andernfalls aktualisiert die Schnittstellenreader-Klasse die schnittstellenspezifische Struktur und kopiert dann den Zähler des Kommunikationskernels in ihr indiziertes Zählerfeld.

4.8.9 Generieren einer neuen Instanz eines Readers

Der Kernel hat also eine Liste von allen *readern* und allen *writern*, die für eine Software nötig sind. Der Kernel muss kontrollieren, dass ein gleiches Objekt nicht mehrmals definiert wird. Das heißt also, dass jedes Objekt für den Kernel eine eindeutige Signatur hat. Diese Signatur muss auch zeigen, ob das Objekt ein *reader* oder ein *writer* ist: In 4.8.11 (Seite 136) wird ein Beispiel gezeigt, bei dem in der gleichen Software Werte erst gelesen und dann geschrieben werden. Die *Type*-Klasse beschreibt den gesamten Vererbungsgrafen eines Objekts. Mit dieser Klasse kann der Kernel überprüfen, ob zum Beispiel ein *navigationReader* nicht schon vorher generiert wurde. Existiert ein *navigationReader* bereits, gibt der Kernel die Adresse dieses Objekts mit der Methode *getObjectOfType* oder *getObjectOfExactType* zurück. Andernfalls wird eine neue Instanz von *navigationReader* generiert. Diese *Type*-Klasse (siehe Abb. 4.47) bildet in etwa die Funktionalität des RTI[19] der Standardbibliothek von C++ nach. Dies war notwendig, weil das RTI derzeit nicht allen C++ Kompilern zur Verfügung steht und für diese Anwendung ein Ersatz gefunden werden musste.

Als Anforderung wurde beschrieben, dass die reader wie eine *Black-box* benutzt werden. Ein Benutzer braucht lediglich einen Standardreader, ohne zu wissen, mit welchem Protokoll die Werte gelesen werden. Dafür hat jeder *standardReader* eine Klassenmethode, die *build* Methode. Diese Methode fragt den Kernel, ob der *reader* nicht bereits generiert ist. Dazu gibt sie den *Typ* der gewünschten *standardReader*-Klasse an eine Suchmethode des Kommunikationkernels. Entweder gibt es schon ein Objekt, das eine Unterklasse von diesem Typ ist, oder die Klassenmethode *build* muss eine neue Instanz herstellen. Dazu muss diese Methode wissen, welches Protokoll verwendet wird. Für einige Schnittstellen sind Konfigurationsparameter nötig. Diese müssen als Parameter beim Aufruf der Software übergeben werden.

4.8.10 Generierung der Sourcecodes

Wenn ein Entwickler bislang nicht austauschbare Informationen lesen oder schreiben will, muss er entweder die Informationen in eine existierende Struktur einfügen oder eine neue Struktur schreiben. Er muss die folgenden Dateien dann entweder aktualisieren oder neu schreiben:

- *StrukT* (eine Klasse des Klassenmodells *StandardStruktur*),

---

[19] Run Time Type Information

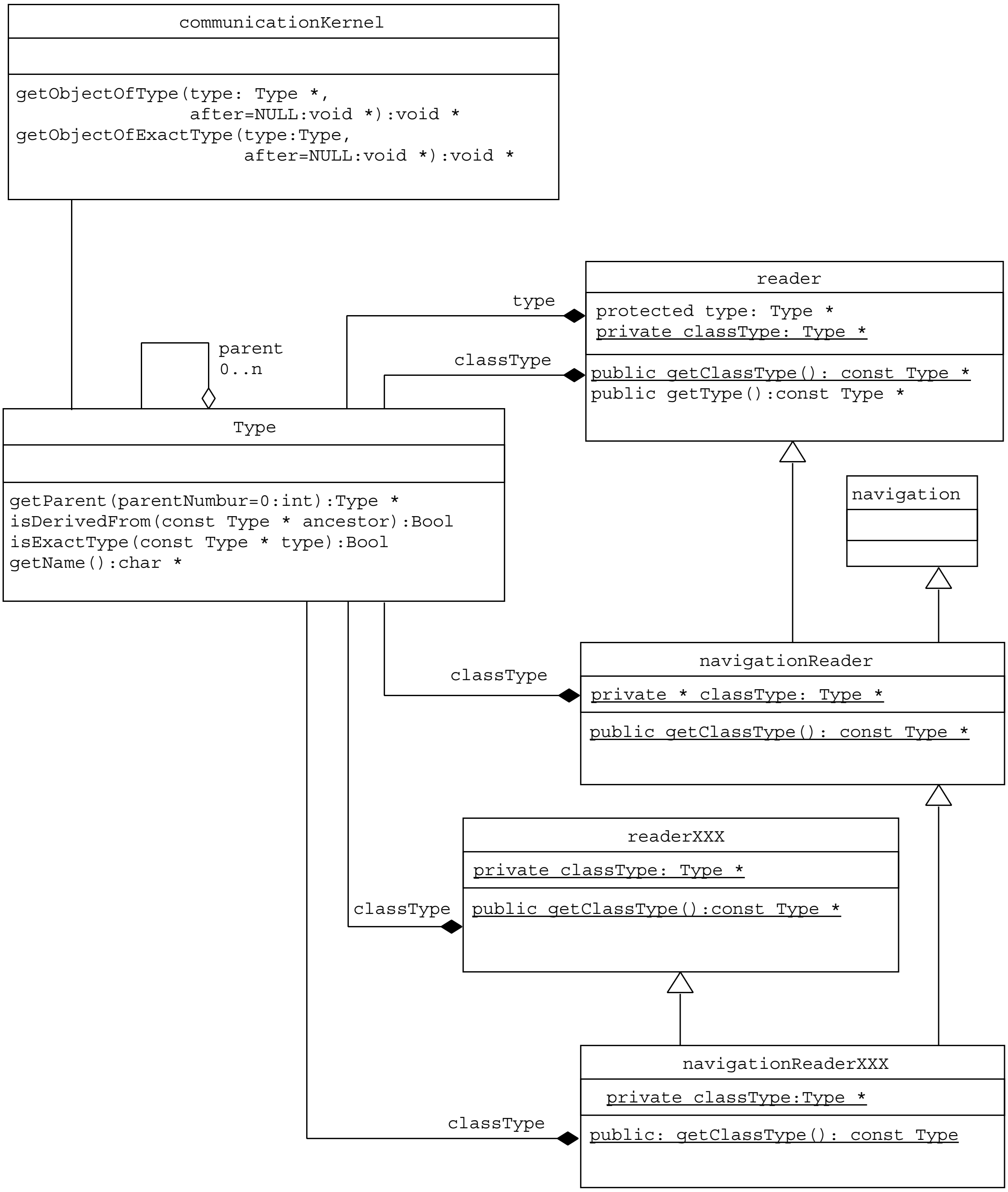


Abb. 4.47: Klassendiagramm der Klasse Type

- *StructReader* (eine Klasse des Klassenmodells *StandardReader*),
- *StructWriter* (eine Klasse des Klassenmodells *StandardWriter*).

Für jedes Protokoll, das gebraucht wird, muss er noch die Klassen

- *structReaderProtokolXXX* (eine Klasse des Klassenmodells *Schnittstellenstrukturreader*),
- und *structWriterProtocolXXX* (eine Klasse des Klassenmodells *Schnittstellenstrukturewriter*) schreiben.

Wenn der Entwickler neue Protokolle benutzen möchte, muss er die neuen Klassen

- *readerProtocolYYY* (eine Klasse des Klassenmodells *Schnittstellenreader*),
- *writerProtocolYYY* (eine Klasse des Klassenmodells *Schnittstellenwriter*)

schreiben. Und mindestens für jede Struktur, die diese Protokolle benutzt, muss er noch

- *structReaderProtocolYYY*
- und *structWriterProtocolYYY*.

erzeugen.

Zur Aktualisierung der Kommunikationsschnittstelle sind also Änderungen an einigen Dateien notwendig. Eine manuelle Aktualisierung wäre aufwendig und fehleranfällig. Die verschiedenen Klassen sollten also automatisch generieren werden. Dafür wird ein Script benutzt. Auf das Schreiben einer zusätzlichen Software konnte verzichtet werden. Es wurde auf die bewährte Software *sed*[20] zurückgegriffen. Es ist mit dieser Software möglich, Dateien zu ändern. Mit einem normalen Editor muss man die Änderungen tippen. Mit dem Editor sed braucht man die Änderungen nicht zu tippen, sondern die Änderungen werden mit Hilfe einer Datei generiert, in der die Änderungen beschrieben werden.

Um eine Klasse zu erzeugen, braucht man eigentlich nur ein Muster der betroffenen Klasse zu aktualisieren. Das Script muss den Namen im Muster mit dem Namen der Klasse ersetzen; innerhalb einer Datei gibt es mehrere Stellen, an denen dieser Name ersetzt werden muss.

Dazu werden einige Stellen definiert, an denen spezifische Informationen der Klasse eingefügt werden. Diese Stellen liegen immer zwischen zwei Kommentaren. Das Script sucht diesen Kommentar für die Generierung einer Klasse, aber auch für deren Aktualisierung.

---

[20] Stream EDitor

Es löscht zuerst alle Zeilen zwischen diesen Kommentaren. Daraufhin werden, abhängig von der Information der Klasse *standardstruktur* (siehe 4.8.4), neue Zeilen eingefügt. Diese neuen Zeilen hängen selbst immer von einem anderen Muster ab. Wenn z.B. die *standardstruktur* die Position des Flugzeugs $positionX$, $positionY$ und $positionZ$ als float besitzt, wird das Script, um die *get Methode* der Klasse Standardstructreader zu generieren, automatisch die Namen der Parameter hinter dem Wort *get* einfügen. Das heißt, hier wird das Script automatisch die folgenden Methoden in den Header der Klasse einfügen:

```
float getPositionX();
float getPositionY();
float getPositionZ();
```

Die Ini-Dateien, die die voreingestellten Werte enthalten, werden ebenfalls automatisch generiert. Bereits vorhandene Voreinstellungswerte werden dabei beibehalten, neu hinzugefügte Parameter werden vom Script zunächst zu Null gesetzt.

Mit diesem Script ist es möglich, neue Informationen in eine existierende Klasse einzufügen, unnötige Parameter zu löschen und auch alle Dateien einer neuen Klasse zu generieren. Der Header der *Standardstruktur*-Klasse (siehe Kapitel 4.8.4) wird nicht automatisch generiert: Der Softwareentwickler muss die Parameter, die er lesen und schreiben möchte, einfügen. Er muss zusätzlich den Typ der Parameter angeben. Aber der Softwareentwickler braucht dem Script normalerweise fast keine anderen Informationen zu geben, das Script kann mit diesen Informationen und den Musterdateien alle Klassen generieren.

Dieses Script erlaubt, dass ein Formatentwickler dieser Anzeigensoftware die Kommunikationsschnittstelle als *Blackbox* behandelt. Er sieht nur die *standardReader*- oder *standardWriter*-Klasse, und braucht weder zu verstehen, wie die ganze Kommunikationsschnittstelle aufgebaut ist, noch muss er diese Klassen ändern.

Für ein neues Protokoll muss man dennoch die Klasse *readerProtocolXXX* und *writerProtocolXXX* schreiben. Dabei ist es nötig, die Muster der Klassen *structReaderProtocolXXX* und *structWriterProtocolXXX* zu generieren. Dazu muss man das Script erweitern, um die Änderungen der neuen Muster zu integrieren.

4.8.11 Verwendung der Schnittstelle in anderen Anwendungen

Die Kommunikationsschnittstelle wird nicht nur in der Flugführungsanzeigessoftware benutzt, sondern auch in anderen Anwendungen. Das folgende Unterkapitel wendet sich diesen Anwendungen zu.

Externe Bedienung der 3D-Flugführungsanzeige

Als erste Anwendung sei die Bedienung der Anzeige erwähnt. Diese Anwendung liest Eingabewerte vom Eingabemedium oder simuliert sie. Zum Beispiel werden die Zoomstufen oder die Druckeinstellung für die Höheninformationen mit Hilfe von Knöpfen im Cockpit eingestellt. In einer Simulation oder der Realität werden die Werte der Hardware von der Bedienung gelesen und weiter zur Anzeige geschickt. Sollte es keine echte Hardwarebedienung geben, wird diese Anwendung die Werte nicht lesen, sondern mit Hilfe einer Bedienungsoberfläche die Werte selbst erzeugen.

Diese Anwendung ist im Rahmen der Entwicklung ebenfalls sehr wichtig: Es ist notwendig, zur Laufzeit beispielsweise die 2D-Elemente bewegen, skalieren oder abzuschalten zu können. Die endgültigen Werte bezüglich der Position im Fenster und der Größe werden danach in die betroffene Ini-Datei der 2D-Elemente geschrieben.

Die Abbildung 4.48 zeigt das Hauptfenster der Bedienungsanwendung und zwei Untermenüs für detaillierte Einstellungen.

Spur Player

Es muss möglich sein, echte oder simulierte Flüge reproduzieren zu können. Das heißt, dass Werte während eines Fluges oder zur Laufzeit einer Simulation in Dateien gespeichert werden. Eine Version von einem *Player* ist in 4.49 abgebildet. Diese Anwendung kann derzeit noch keine Werte speichern aber kann mehrere Formate anderer Anwendungen abspielen. Dies ist auch mit den verschiedenen Kommunikationsschnittstellen, die zur Zeit zur Verfügung stehen, möglich. Eine Erweiterung ist geplant, um Werte speichern zu können, auch mit der Möglichkeit, die zu speichernden Strukturen vor der Aufnahme auszuwählen.

Protokollübersetzer

In bestimmten Fällen, wie in der Beispielanwendung ATTAS-Flugkampagne 1997 [HKLP00], ist es nicht möglich, nur die eigenen Kommunikationsschnittstellen zu verwenden. Der Rechner B in Abbildung 4.50 wird als Brücke benutzt: Er liest die Flugzeugdaten vom Rechner A mit der *RFM-Kommunikationsschnittstelle* und schickt diese Werte über die interne Kommunikationsschnittstelle des FSR auf den Ethernet-Bus, damit die anderen Rechner C,D und E im Versuchsflugzeug die Werte auch empfangen können. Dies wurde mit der Protokollübersetzer-Software realisiert.

Testsoftware

Für die Entwickler wurde im Rahmen dieser Arbeit eine Testanwendung erzeugt. Mit

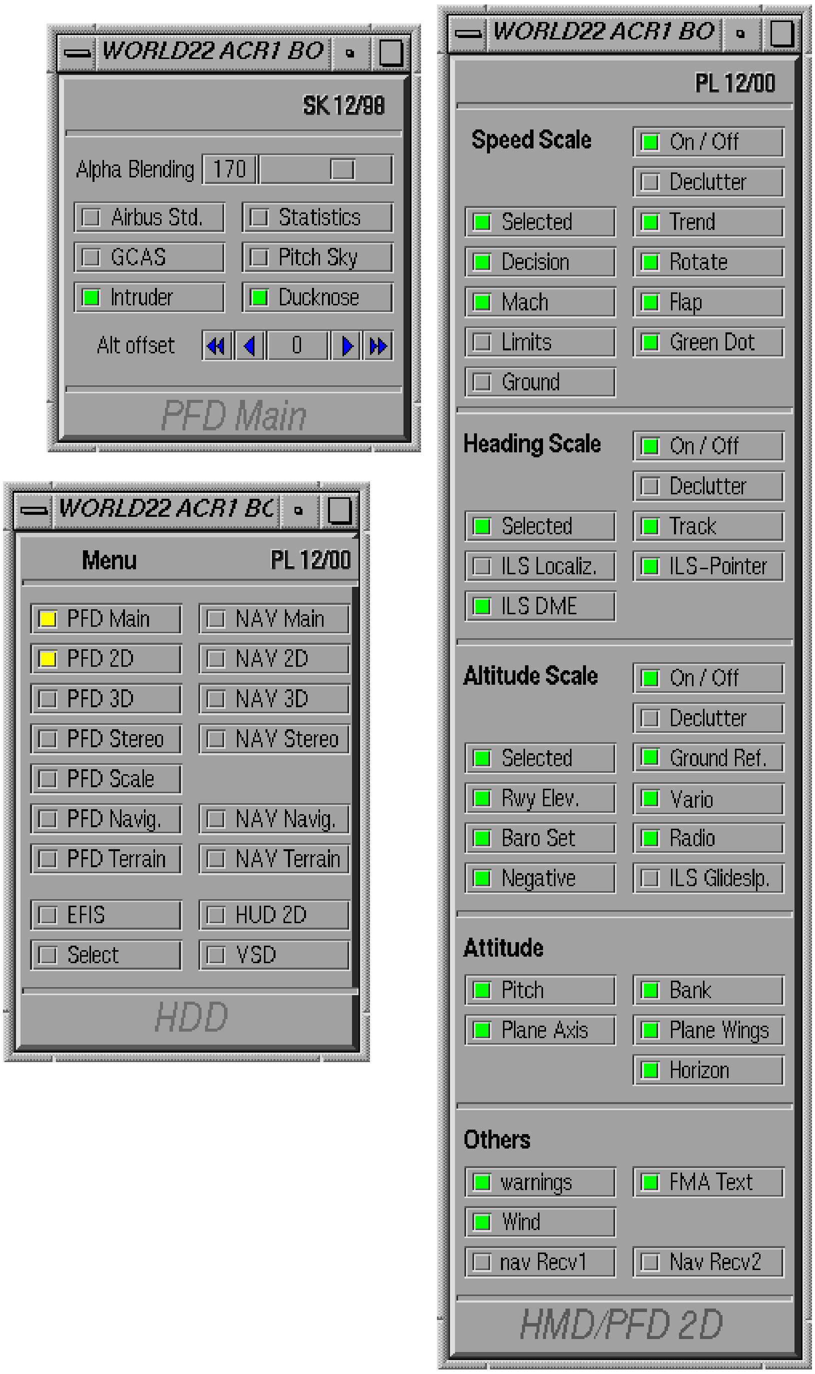
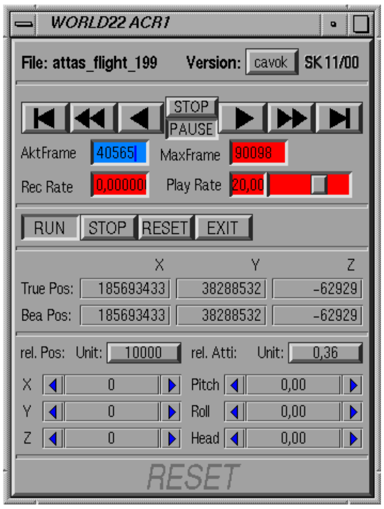


Abb. 4.48: Schnittstelle für den Formatentwickler

dieser ist es zum einen möglich, die unterschiedlichen Kommunikationsschnittstellen zu überprüfen. Dafür wird die Anwendung auf zwei Rechnern gestartet, wobei auf einem Rechner gesendet und von einem empfangen wird. Durch den Verzicht auf die komplexe Anzeigesoftware können so gezielt die Kommunikationsmethoden untersucht werden.



File: attas_flight_199 Version: cavok SK 11/00

AktFrame 40565 MaxFrame 90098

Rec Rate 0,00000 Play Rate 20,00

	X	Y	Z
True Pos:	185693433	38288532	-62929
Bea Pos:	185693433	38288532	-62929

rel. Pos: Unit: 10000 rel. Atti: Unit: 0,36

X 0 Pitch 0,00

Y 0 Roll 0,00

Z 0 Head 0,00

Abb. 4.49: Player

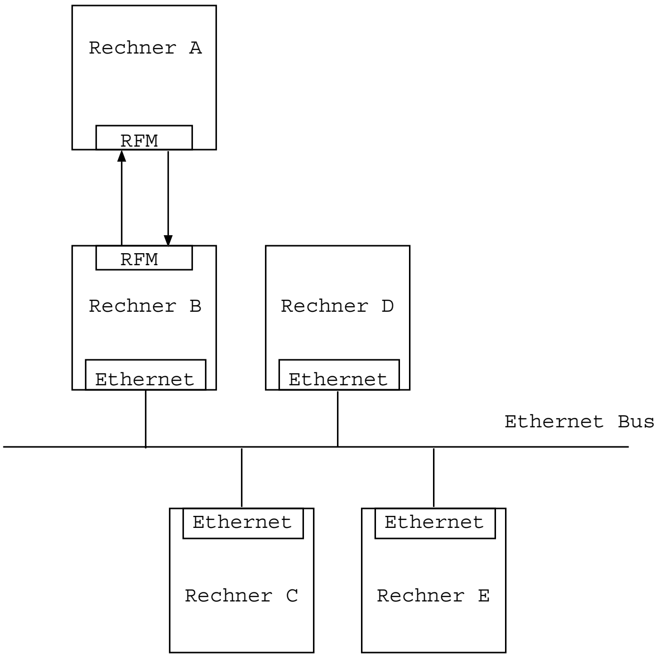
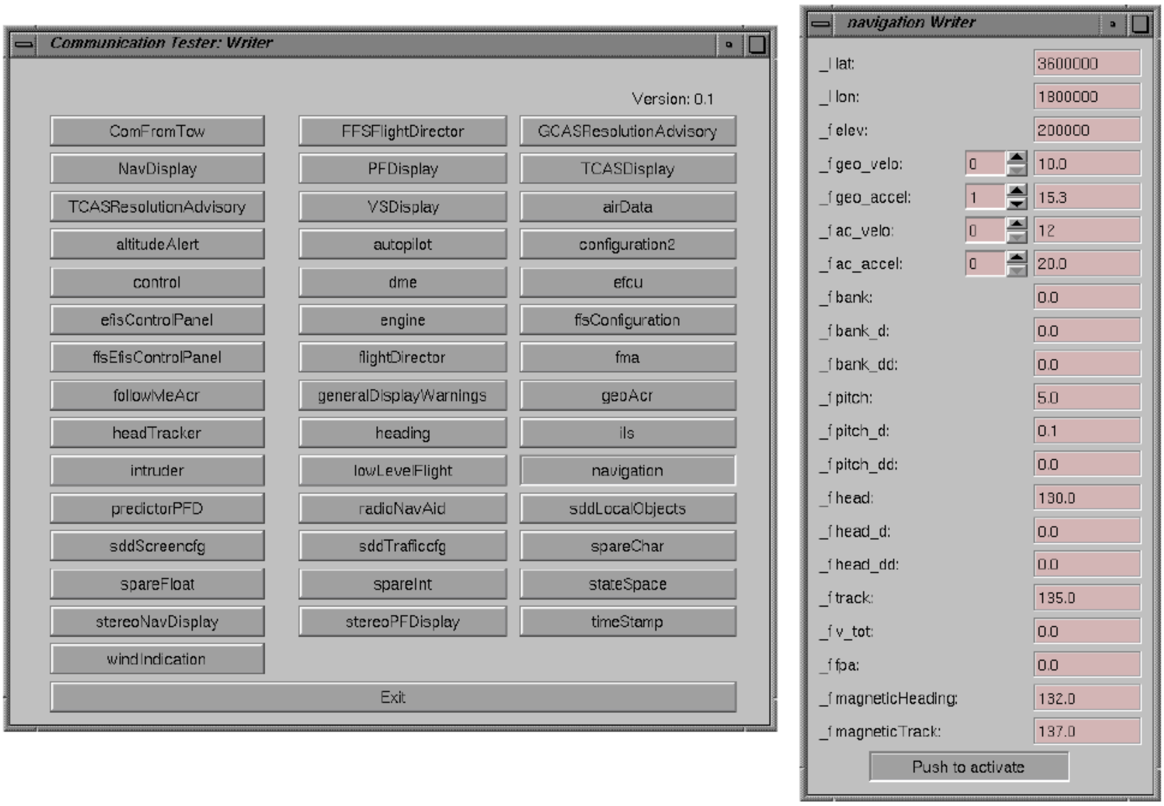


Abb. 4.50: ATTAS-Architektur

Während einer Integration der Flugführungsanzeige-Software in eine fremde Umgebung kann so beispielsweise die Korrektheit von Formatkonvertierungen überprüft werden.

Zum anderen dient die Testanwendung der gezielten Einflussnahme auf einzelne Werte, ohne eine Simulation oder aufgezeichnete Flugspuren manipulieren zu müssen. Beispielsweise lässt sich durch Vorgabe der *Indicated Air Speed* durch die Testanwendung die korrekte Reaktion der Geschwindigkeitsskala der Anzeige darauf überprüfen.

Abbildung 4.51 zeigt die Bedienungsoberfläche der Testanwendung.



_l lat: 3600000
_l lon: 1800000
_f elev: 200000
_f geo_velo: 0 10.0
_f geo_accel: 1 15.3
_f ac_velo: 0 12
_f ac_accel: 0 20.0
_f bank: 0.0
_f bank_d: 0.0
_f bank_dd: 0.0
_f pitch: 5.0
_f pitch_d: 0.1
_f pitch_dd: 0.0
_f head: 130.0
_f head_d: 0.0
_f head_dd: 0.0
_f track: 135.0
_f v_tot: 0.0
_f fpa: 0.0
_f magneticHeading: 132.0
_f magneticTrack: 137.0

Abb. 4.51: testerwriter

5 Verwendung der Flugführungsanzeige-Software in verschiedenen Projekten

Für jedes Projekt fasst eine Tabelle die Charakteristika des Projekts bezüglich der Anwendung zusammen:

- Die Art der im Projekt verwendeten Anzeige.
- Die Anzeige/Rechner-Konfiguration für den Kapitän und den First Officer. Es wird angegeben, wie viele Rechner für das Projekt pro Anzeige benutzt werden, und ob die Software gleichzeitig eine oder mehrere Anzeigen verwaltet. Wenn für den Kapitän zum Beispiel ein HDPFD und ein ND verwendet werden, werden beide entweder von einem Rechner A oder von zwei Rechnern, A und B, berechnet. Die Software kann theoretisch gleichzeitig alle Anzeigen verwalten (jeder Pilot hat eine getrennte Bedienung, die andere Werte zur Flugführungsanzeige schickt).
- Die Auflösung des Bildschirms. Es ist zu beachten, dass die Bildschirme manchmal rotiert sind.
- Die Fenstergröße in Pixel: $horizontal_g * vertikal_g$ und für die perspektivische Anzeige das Field of View in Grad $horizontal_{FOV} * vertikal_{FOV}$. In der Software werden die Werte wie folgt ineinander umgerechnet:

$$horizontal_g = vertical_g \frac{tan(\frac{horizontal_{FOV}}{2})}{tan(\frac{vertical_{FOV}}{2})} \tag{5.1}$$

- das Gelände-Modell,
- die Symbologie: 2D/3D-Elemente,
- die Kommunikationsschnittstelle und
- die externe Bedienung.

5.1 Projekt ISAWARE

Ziel von *ISAWARE*[1] war es, die Flugsicherheit durch die Schaffung eines ständigen und vorausschauenden Situationsbewusstseins des Piloten in allen Flugphasen zu verbessern.

---

[1] Increasing Safety through collision Avoidance WARning intEgration

Dazu wurde das Konzept des Integrated Situation Awareness System (ISAS) umgesetzt, bei dem alle Informationen über Flugplanung, Navigation, Wetter, Verkehr, Gelände und Gefahren zusammengeführt und entsprechend ihrer Priorität visuell per Display an den Piloten weitergegeben werden ([SLKM00], [KS01]). Das Projekt erfolgte in Zusammenarbeit mit verschiedenen Herstellern und Betreibern von Luftfahrzeugen in Europa.

Die Arbeit des Instituts für Flugsysteme und Regelungstechnik umfasste die Entwicklung einer synthetischen Anzeige zur Darstellung der Informationen. Das entwickelte System wurde in einen Simulator der NLR integriert und zusammen mit Piloten von Iberia, Air France, KLM und Eurocopter validiert und bewertet.

Die Flugführungsdisplay-Software wurde bei diesem Projekt dazu benutzt, um das HD-PFD und das HUD darzustellen.

Das ISAWARE-HDPFD ist eine quadratische Anzeige in Anlehnung an das HDPFD in heutigen Verkehrsflugzeugen.

Für einen Vergleich zwischen der herkömmlicher Symbologie und der weiterentwickelten Darstellung mit synthetischem Gelände sollte während der Laufzeit der Anwendung zwischen beiden Formaten umgeschaltet werden können.

Für die Geländedarstellung wird ein einheitliches Datenbankformat im gesamten Projekt benutzt.

Das HUD wird in einem rechteckigen Fenster dargestellt. Um die HUD-Darstellung auch ohne HUD-Hardware nutzen zu können, wurden die Videosignale der HUD-Darstellung und der Simulator-Außenssicht gemischt. Die 2D-Darstellung des HUD ist mit der des HDPFD nahezu identisch. Nur ist die HUD Symbologie einfarbig weiß. Um Konflikte durch normalerweise im HDPFD farbkodierte Symbole zu vermeiden, wurde der Inhalt der Symbologie teilweise vereinfacht.

Als im Gesichtsfeld auf der Combinerscheibe eingespiegelte Anzeige besitzt das HUD keine Geländedarstellung.

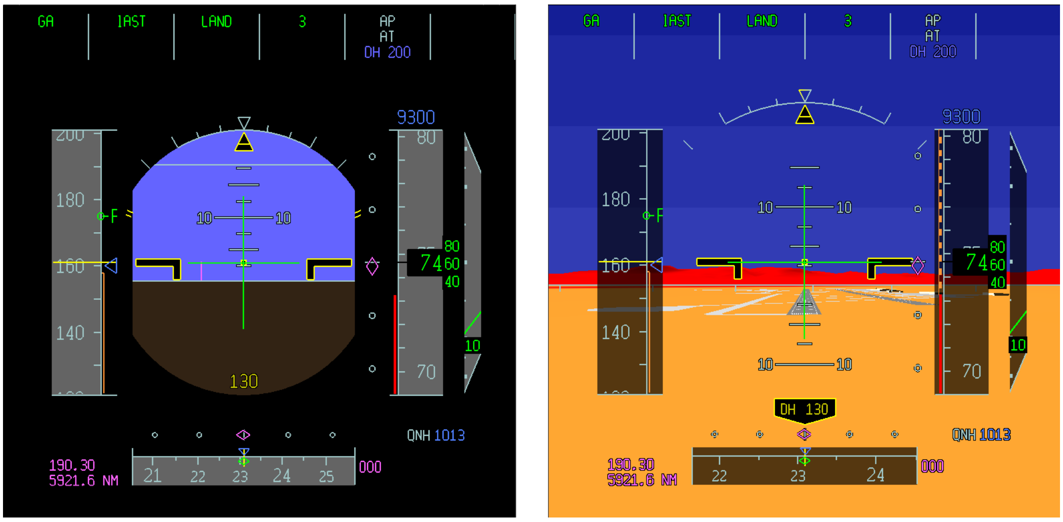
Die Anzeigen benötigen eine neue Kommunikationsschnittstelle, da die ISAWARE Datenübertragung mit sehr großen Strukturen vorgegeben war, die nicht der internen Struktur der Anzeigesoftware entspricht.

Eine Funktion dieser Schnittstelle liefert die Adresse einer Information in einer ISAWARE Struktur. Mit deren Hilfe werden die internen Strukturen angepasst.

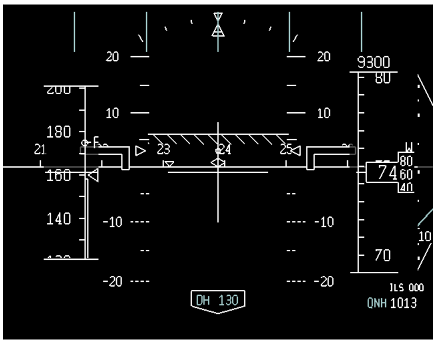
Übersetzermethoden sind nötig, um Datentypen zu konvertieren oder um Informationen an andere Formate zu adatpieren.

Die Bedienung der Anzeige wurde hauptsächlich mit Initialisierungsdateien (Typ: *.ini) und einer Anwendung der NLR realisiert. Nur während der internen Implementierung wurde die Bedienungsanwendung der TUD benutzt. Die Bedienungsanwendung liest die Struktur, bevor sie etwas an dieser Struktur ändert.

Die nächsten Abbildungen zeigen das Standard und *Enhanced* HDPFD so wie das HUD, die in ISAWARE entwickelt wurden.



Standard HDPFD Enhanced HDPFD



HUD

Abb. 5.1: ISAWARE Anzeigeformate

5.2 Projekt AWARD

Das Projekt *AWARD*[2] hat *Synthetic and Enhanced Vision Systems* (SVS/EVS) erprobt. Die Aufgabe der TU Darmstadt war die Erstellung der Synthetic Vision Anzeige

---

[2] All Weather ARrival and Departure

Anzeige	PFD	HUD
Rechner Kapitän	A	B
Rechner First Officer	C	–
Auflösung	$1024 * 768$	$640 * 480$
Fenstergröße/ Field Of View	$768 * 768/$ $60° * 60°$	$640 * 358/$ $48° * 28°$
Gelände	TIN Mexiko-Acapulco	–
Symbologie	ISAWARE farbig	ISAWARE monochrom
Kommunikationsschnittstelle	ISAWARE spezifisch	
Bedienung	TUD Software für die Entwicklung NLR Software für die Erprobung	
Versuchsumgebung	NLR Cockpit	
Besonderheit	–	

Tab. 5.1: Projekt ISAWARE

([GMK01],[MKG99]): Es handelt sich um eine Darstellung des Geländes mit Hilfe von Datenbankinformationen. Das SVS wurde im Flugsimulator der NLR und im Flugsimulator der TUD bewertet.

Für dieses Projekt musste die Software ein HDPFD und ein ND darstellen.

Die 2D-Darstellungen wurden verändert, um den Kontrast zur Geländedarstellung im Hintergrund der 2D-Elemente zu verbessern.

Die Geländedarstellung wird wie beim ISAWARE-Projekt nicht ständig angezeigt. Neben der Geländedarstellung werden Anflugkanäle und Taxiway-Informationen für die Rollphase gezeichnet.

Für die Kommunikationsschnittstelle wird eine interne Bibliothek der NLR (unterschiedlich von ISAWARE) benutzt.

Die Anwendung der NLR und die interne Bedienungssoftware der TUD wurden zur Bedienung der Flugführungsanzeige benutzt.

Die Abbildung 5.2 zeigt das AWARD-HDPFD und -ND während eines Anflugs und einer Rollphase auf dem Flughafen Frankfurt am Main. Besonders zu berücksichtigen ist der *Declutter*-Mode der Geschwindigkeit-, Kurs- und Höhenskala des HDPFDs während der Rollphase (rechtes Bild).

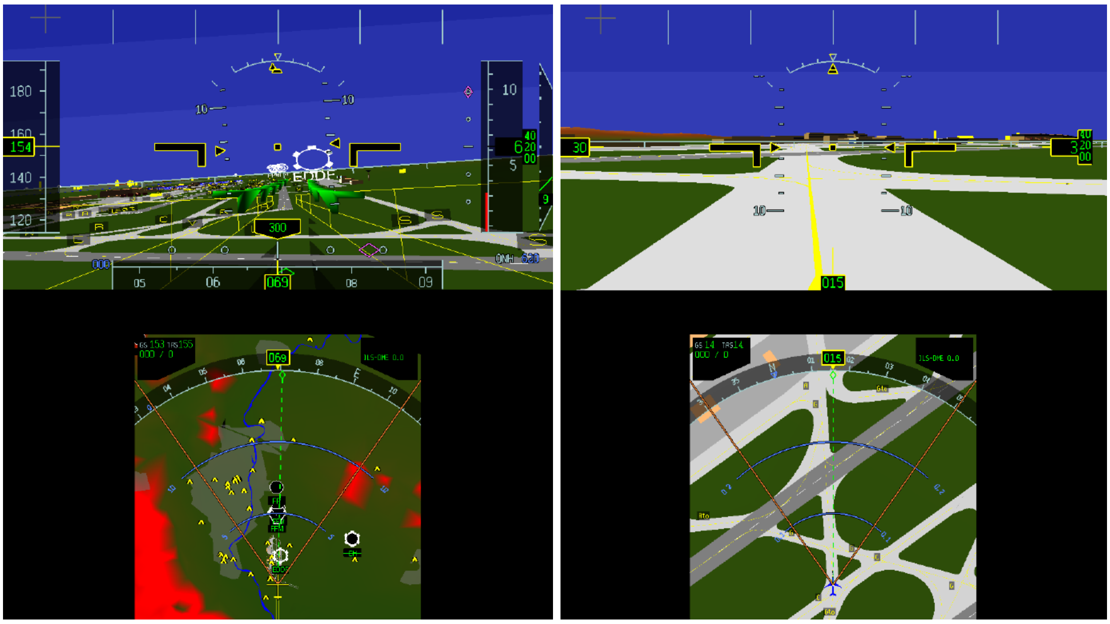


Abb. 5.2: AWARD HDPFD und ND

Anzeige	PFD	ND
Rechner Kapitain	A	B
Rechner First Officer	C	D
Auflösung	$640 * 480$ $1024 * 768$	
Fenstergröße/ Field Of View	$638 * 480/60° * 47°$ $765 * 576/60° * 47°$	$480 * 480$ $768 * 768$
Gelände	TIN Frankfurt	
Symbologie	AWARD	
Kommunications-schnittstelle	AWARD spezifisch	
Bedienung	TUD Software für die Entwicklung TUD und NLR Software für die Erprobung	
Versuchsumgebung	TUD und NLR Cockpit	
Besonderheit	Declutter Mode	–

Tab. 5.2: Projekt AWARD

5.3 ATTAS-Flugkampagne 1999

Das Projekt FFS VI[3] war ein nationales Forschungsvorhaben mit den Industriepartnern Diehl Avionik GmbH und EADS[4].

Die 2D-Darstellung war der des Projekts AWARD sehr ähnlich. Ein neuer Flight Director wurde zusätzlich generiert.

Im Gegensatz zu ISAWARE oder AWARD wurde das Gelände immer dargestellt, aber es wurden verschiedene Datenbankmodelle benutzt. Der Prädiktor konnte abhängig vom Flugmodus, der von EADS definiert wurde, aktiviert oder deaktiviert werden. Es war auch ein *Follow-me*-Flugzeug entwickelt worden, dieses wurde aber während der Flugkampagne 1999 nicht benutzt. Für den Flug wurden Flugkanäle von einer externen Anwendung generiert und danach wie eine Geländekachel gelesen und gezeichnet. Allerdings wurden immer nur die jeweils nächsten 6000m des Kanals im HDPFD dargestellt. Im ND wurde der Kanal komplett gezeigt.

Die Bedienung der Software wurde mit der Hardware des Flugzeugs und der internen Bedienungsanwendung der TUD durchgeführt.

Als Kommunikationsschnittstelle wurde mit EADS ein gemeinsames System entwickelt. Ein Rechner der TUD wurde mit einem Echtzeitsystem-Rechner der EADS an ein *ReFlectiv Memory* (RFM) Netzwerk gekoppelt. Dieses Netzwerk wird normalerweise als *Ring* benutzt, hier wurde es als *Point to Point* Netzwerk verwendet. Um sicherzustellen, dass die vom Rechner der TUD gelesene Struktur konsistent und vollständig war, wurde vor der Nutzung der Werte ein besonderer Kontrollmechanismus angewandt.

Außer der Anzeige für das HDPFD und das ND gab es verschiedene Anwendungen:

- die Bedienungsanwendung,
- eine Datenspeicheranwendung und
- eine externe HUD Anzeigeanwendung.

Jede Anwendung musste auf einem getrennten Rechner arbeiten. Die zwei letzten Anwendungen konnten nur mit der internen Kommunikation der TUD arbeiten, außerdem stellte die EADS nur eine RFM-Karte für die TUD zur Verfügung. Daher ergab sich die Notwendigkeit einen Protokollübersetzer zu entwickeln. Der Protokollübersetzer lief auf

---

[3] FlugFührungsSicht Phase VI
[4] European Aeronautic Defense and Space Compagny

dem gleichen Rechner wie das HDPFD und das ND. Der Rechner, auf dem HDPFD und ND berechnet wurden, war mit dem EADS Rechner verbunden. Er diente gleichzeitig als Netzwerkbrücke. Er las die Information vom RFM Netzwerk und schrieb sie in ein Ethernetnetzwerk mit der internen Kommunikationsschnittstelle der TUD. In Abbildung 4.50 Seite 138 wurde diese Konfiguration des Netzwerks bereits gezeigt.

Die Abbildung 5.3 zeigt zwei Verisonen des HDPFD und des ND während der ATTAS-Flugkampagne: Sie unterscheiden sich im Datenbankmodell.

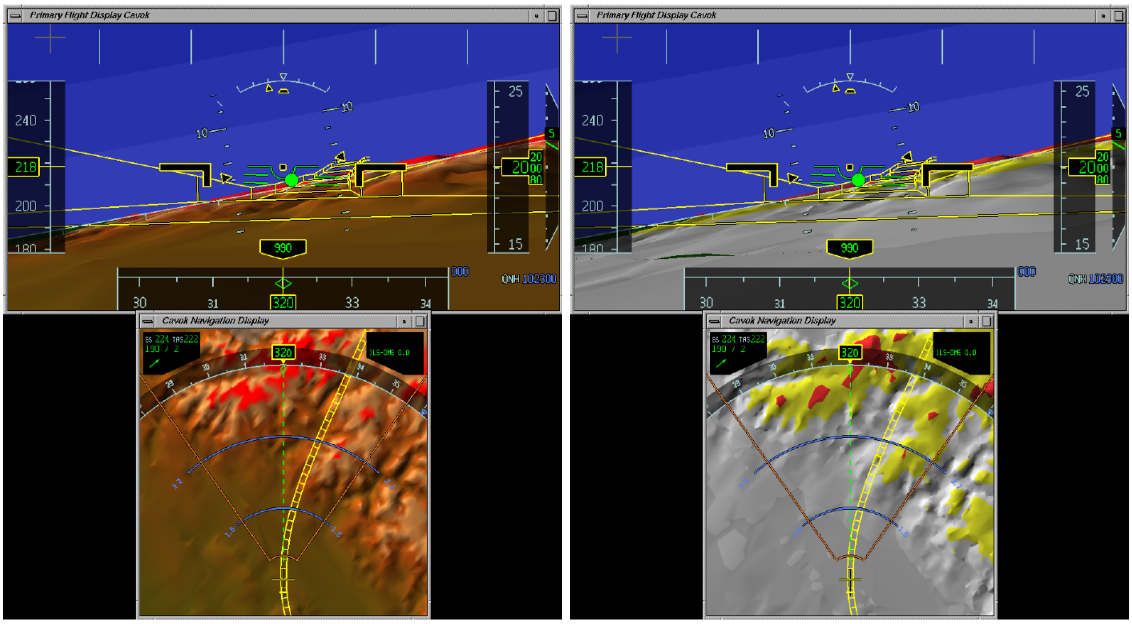


Abb. 5.3: ATTAS HDPFD ND

5.4 EADS-Flugsimulatoren

Für die EADS wurden die Anzeigen der ATTAS-Flugkampagne 1999 (FFS VI) für deren Flugsimulatoren portiert.

Eine neue Anzeige kam hinzu: das *Vertical Situation Display*, VSD. Da die flächenmäßigen Begrenzungen des Bildschirms im ATTAS Cockpit hinfällig waren, wurden die Anzeigen in größeren Fenstern dargestellt. Außerdem war die Anordnung der Fenster eine andere, siehe Abb. 5.4. Ansonsten gab es keine Änderungen an der Darstellung.

Die Kommunikationsschnittstelle wurde erweitert: Die Anzeigen mussten nicht nur mit RFM Netzwerk, sondern auch mit SCRAMNET arbeiten. SCRAMNET ist ein Ringnetzwerk und funktioniert ähnlich wie RFM. Es gab für das SCRAMNET zwei Arten von

Anzeige	PFD	ND
Kapitain Rechner	A	A
First Officer Rechner	–	–
Auflösung	$1024 * 1280$	
Fenstergröße/ Field Of View	$1013 * 584/$ $70° * 44°$	$654 * 654$
Gelände	TIN und graues Schichtenmodell Braunschweig, Harz und	
Symbologie	TUD und EADS Flight Director	TUD
Kommunikationsschnittstelle	2 Netze: RFM und TUD Kommunikation	
Bedienung	ATTAS und TUD Software	
Versuchsumgebung	ATTAS VFW614	
Besonderheit	Flugkanal	

Tab. 5.3: Projekt FFS Phase VI

Netzwerkkarten: eine mit VME Bus und eine mit PCI Bus. Die Kommunikationsschnittstelle musste diese Karten konfigurieren, um die Daten lesen zu können.

Die Abbildung 5.4 zeigt eine Darstellung des HDPFD, des ND und des VSD, die für den EADS Flugsimulator programmiert wurden.

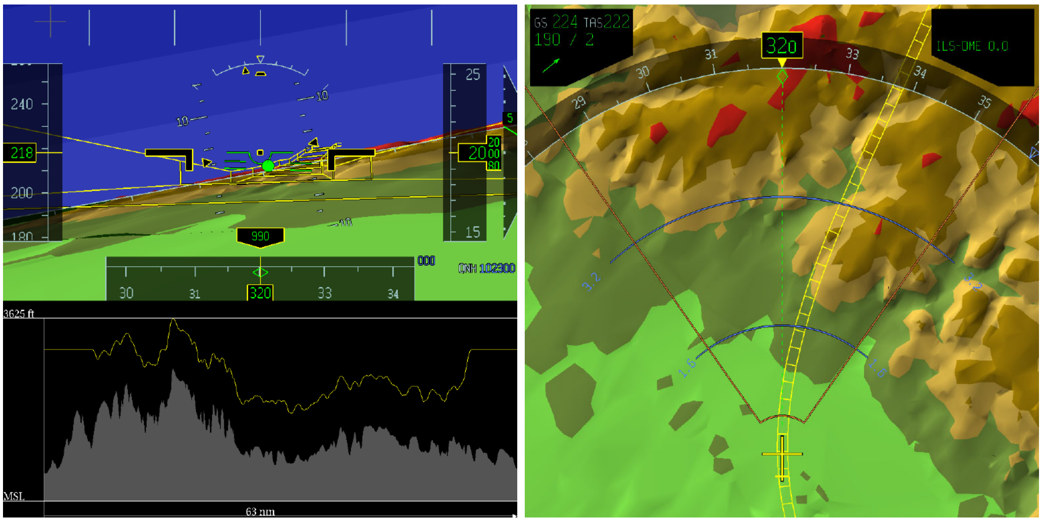


Abb. 5.4: EADS HDPFD ND und VSD

Anzeige	PFD	VSD	ND
Rechner Kapitain	A	A	A
Rechner First Officer	B	B	B
Auflösung	$2 * 1280 * 1024$		
Fenstergröße/ Field Of View	$1024 * 591/$ $70° * 44°$	$1024 * 344$	$1024 * 1024$
Gelände	TIN und Schichtenmodell		
Symbologie	TUD + EADS Flight Director	EADS	TUD
Kommunications-schnittstelle	RFM;Scramnet VME; Scramnet PCI		
Bedienung	EADS Flugsimulator und TUD Software		
Versuchsumgebung	EADS Flugsimulator		
Besonderheit	-		

Tab. 5.4: Projekt EADS Flugsimulator

5.5 Simulator des Fachgebiets *Flugsysteme und Regelungstechnik der TUD*

Der Flugsimulator des Fachgebiets *Flugsysteme und Regelungstechnik der TUD* stellt die höchsten Anforderungen an die Integrationen dieser Anzeigeanwendungen. Da sie sich wegen der zahlreichen Forschungsarbeiten laufend ändern und z.B. alle zuvor genannten Vorhaben im TUD-Simulator vorbereitet wurden, werden nur die wesentlichen Besonderheiten erwähnt.

Die Anzeigen werden auf LCD-Displays normalerweise entweder im Hochkant- oder im Querformat dargestellt. Es wird versucht, immer einen möglichst großen Teil des Bildschirms auszunutzen. Das HDPFD und das ND müssen manchmal getrennt und manchmal auf einem einzigen Bildschirm gezeigt werden. Ist Letzteres der Fall, werden die beiden Anzeigen auch von einem einzigen Rechner berechnet. Die beiden HDPFD für Kapitän und First Officer müssen auf zwei Bildschirmen dargestellt werden, können aber trotzdem von nur einem Rechner berechnet werden. Diese Konfiguration erspart einen Rechner. Die beiden HDPFD sind dann einfache Kopien, eine neue Berechnung durch die Software ist unnötig.

Die Bedienung entstammt der Bedienungsanwendung. Entweder liest sie die Information von der Hardware, oder die Informationen werden von der Bedienungsanwendung selbst berechnet. Die Anzeige liest auf jeden Fall immer die Werte der Bedienungsanwendung ohne Berücksichtigung des Ursprungs dieser Werte. Die Entscheidung, ob Werte berechnet oder von der Hardware geliefert werden, wird nicht in der Anzeigensoftware durchgeführt, sondern in der Bedienungsanwendung, die nicht so zeitkritisch ist. Schließlich ist es möglich, die Anzeigen des Piloten und des Copiloten entweder zusammen oder getrennt zu bedienen: die Bedienungsanwendung muss entweder die Werte zusammen für beide Piloten schicken, oder die Bedienungsanwendung wird zweimal gestartet, und beide müssen in diesem Fall die Anzeige jeweils eines Piloten bedienen.

Als Kommunikationsschnittstelle dient die interne Schnittstelle. Die Abbildung 5.5 zeigt das PFD und das ND bei einem Abflug in Juneau (Alaska).

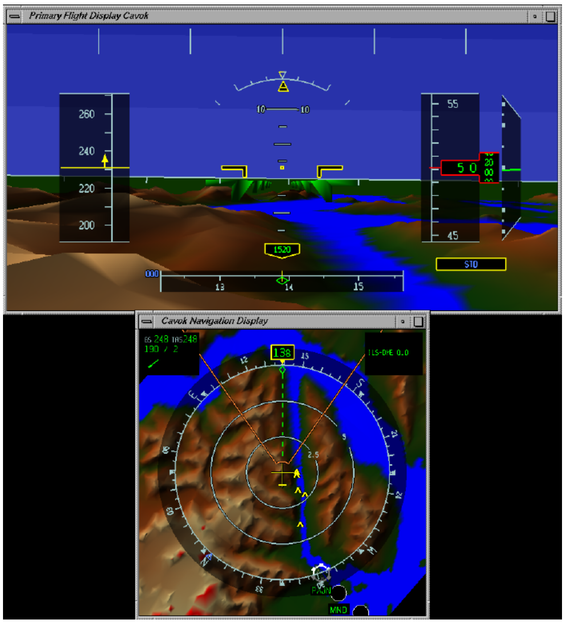


Abb. 5.5: Beispiel PFD ND im TUD-Simulator

Anzeige	PFD	HUD	Visual	ND	VSD
Kapitain Rechner	A	B	C	A oder D	- oder A
First Officer Rechner	E	–	C	E oder F	- oder E
Auflösung	$1024 * 768$	$640 * 480$	$3 * 1280 * 1024$	$1024 * 768$	$1024 * 768$
	$1280 * 1024$	$1280 * 1024$		$1280 * 1024$	$1280 * 1024$
	$1024 * 768$ S.[5]			$1024 * 768$ S.	
	$1120 * 840$ S.			$1024 * 768$ S.	
	$1280 * 1024$ S.			$1280 * 1024$ S.	
Fenstergröße/ Field Of View	$914 * 475/$ $70° * 40°$	$639 * 405/$ $40° * 26°$	$3 * 1280 *$ 1024/ $(3 * 70°) *$ $58.512°$	$475 * 475$	$1024 * 433$
	$768 * 480/$ $70° * 47.271°$			$540 * 540$	–
	$1010 * 525/$ $70° * 40°$			$768 * 768$	–
	$1033 * 768/$ $70° * 55°$			$768 * 768$	–
	$620 * 620/$ $60° * 60°$			$620 * 620$	–
	$300 * 300/$ $60° * 60°$			$300 * 300$	–
	$645 * 500/$ $70° * 57$			$500 * 500$	–
				$768 * 768$	$768 * 256$
Gelände	TIN, Schichten, Konturlinien,Textur Modell				
Symbologie	Von jedem Projekt				
Kommunikations-schnittstelle	TUD Intern				
Bedienung	TUD Software und TUD Flugsimulator				
Versuchsumgebung	TUD Flugsimulator				
Besonderheit	–				

Tab. 5.5: TUD Flugsimulator

---

[5] S.: Stereo

6 Zusammenfassung und Ausblick

Dreidimensionale Flugführungsanzeigen sollen das räumliche Situationsbewusstsein der Piloten verbessern. Weil eine erste Generation solcher Anzeigenandwendung an der TUD verschiedenen Beschränkungen unterlag, wurde es erforderlich, eine zweite Generation zu entwickeln, die auf der ganzen Erde einsatzfähig ist. Die neue Version, die in dieser Arbeit vorgestellt wurde, ermöglicht:

- die Flugführungsanzeigen in verschiedenen Umgebungen zu erproben
- eine einfache Erweiterung (z.B. 2D/3D Symbolik, Kommunikationsschnittstelle etc.),
- neue Anzeigenformate,
- und die einfache Erzeugung von Varianten für Forschungszwecke.

Aufgrund dieser Anforderungen wurde eine objektorientierte Softwarestruktur gewählt und gemäss dem internationalen Standard UML um gesetzt. Die Implementierung geschah in C++.

Die Software ist auf verschiedenen Hardwareplattformen und UNIX-Derivaten (IRIX, Linux) lauffähig. Um die Softwareentwickler bei der Erweiterung oder Aktualisierung der Kommunikationsschnittstelle zu unterstützen, wurde ein Skript entwickelt, das automatisch wesentliche Softwarekomponenten erzeugt.

Die neue Anwendung erfüllt die Anforderungen der Piloten. Formatentwickler haben die Möglichkeit, die Darstellung zu ändern:

- Diese Anwendung funktioniert mit verschiedenen Bildschirmauflösungen.
- Man kann den Bildschirm hochkant oder im Querformat benutzen.
- Es ist möglich, verschiedene Arten von zweidimensionalen Elementen zu verwenden.
- Es ist möglich diese verschiedenen Versionen der Elemente zu ändern, ohne dass man die Software wieder kompilieren muss.
- Die Farbe der Elemente ist nicht in der Software fest implementiert.
- Die Position eines Elementes kann zur Laufzeit geändert werden.

- Die Größe eines Elementes kann zur Laufzeit geändert werden.
- Elemente können zur Laufzeit ein- oder ausgeschaltet werden.
- Es ist möglich, verschiedene Arten der Geländedarstellung zu verwenden.
- Die dreidimensionalen Elemente wie der Flugprädiktor oder Fremdflugzeuge sind als Datenbankmodelle definiert; dies erlaubt es, verschiedene Darstellungsmodelle zu nutzen.
- Diese dreidimensionalen Elemente können ein- oder ausgeschaltet werden.
- Teile der Geländedarstellung werden aus der Datenbank zur Laufzeit geladen.
- Darstellungseigenschaften der Geländeelemente werden aus der Datenbank gelesen, dies erlaubt Änderungen, ohne dass man die Anwendung wieder kompilieren muss.
- Durch eine flexible Kommunikationsschnittstelle kann man die Anwendung in verschiedenen Umgebungen betreiben.

Diese Software besteht hauptsächlich aus vier Modulen:

- dem zweidimensionalen Grafikmodul,
- dem dreidimensionalen Grafikmodul,
- der Kommunikationsschnittstelle
- und dem Anzeigeverwalter.

Jedes dieser Module besitzt Untermodule, wie zum Beispiel verschiedene Arten von zweidimensonalen Elementen. Die Modularität der Anwendung erlaubt einerseits ein besseres Verständnis der Anwendung, andererseits kann die Software an verschiedene Umgebungen mit verschiedenen Anforderungen angepasst werden. Hierzu müssen nur kleine Module geändert werden. Das Konzept dieser Softwareentwicklung hat die folgende Erweiterung der existierenden Anzeigen ermöglicht:

- Die zweidimensionalen Elemente mit neuer Dynamik oder neuer Darstellung für verschiedene Projekte,
- Das VSD mit dynamischen Anzeigen

- Das HDPFD mit perspektvischer stereoskopischer Projektion [May03],
- Das ND mit paralleler stereoskopischer Projektion [May03],
- Das HUD mit stereoskopischer Projektion [Kai03].

Außerdem hat dieses Konzept neue Anzeigen realisierbar gemacht:

- Ein monoskopisches und stereoskopisches HMD [Len02] mit *Wireframe*-Geländedarstellung.
- Eine monoskopische und stereoskopische Radarlotsen-Anzeige [Cav01].

Es ist jetzt möglich, eine neue Dynamik der dreidimensionalen Geländeelemente als neue Klasse (oder neue Module) zu entwickeln.

Für jede dieser Änderungen werden mit Hilfe eines Skripts neue Kommunikationsstrukturen generiert oder aktualisiert. Dies erleichtert die Arbeit der Softwareentwickler. Klassen, die die zwei- und dreidimensionale Elemente modellieren, erleichtern ebenfalls die Erweiterung der Software.

Ausblick

Zur Zeit wird die stereoskopische Darstellung in verschiedenen Klassen verwaltet. Es wäre möglich, diese Verwaltung in einer Klasse zu zentralisieren. Verschiedene stereoskopische Darstellungen wären dann einfacher zu erweitern.

Um die Performance der Anwendung zu verbessern, wäre es denkbar, aktive *Datenbanken* zu benutzen. So wäre es möglich, den *Level-Of-Detail* der Geländedarstellung zu verbessern. Zur Zeit hat ein Teil der Geländedarstellung einen gemeinsamen Level Of Detail; mit einer dynamischen *Meshdatenbank* wäre es möglich, die Qualität der Darstellung innerhalb eines Teils zu verändern. Hier könnte man die *Active Surface Definition* von *OpenGL Performer* benutzen [Eck97]. Es wäre eine Erweiterung des LODs, und erlaubte unter anderem glatten Übergang zwischen zwei LOD-Stufen.

Literatur

[Ach97] W. Achtert. *Objektorientierte Software-Entwicklung von der Struktierung bis zur Migration*. Computerwochee-Verl. München (Schritenreihe: Informationsverarbeitung in der Praxis)(CW-Praxis), 1997.

[AG] Deutsche Lufthansa AG. *Airplane Operations Manual AOM A320 Family, Vol.1 (Flight Crew Operating Manual, Vol1) Systems Description*.

[ARE$^+$98] O. Albert, F. Roshani, K. Engels, C. Huth, F. Thurecht, und J. Schiefele. Das Forschungscockpit der TU Darmstadt ein Werkzeug zur Untersuchung neuer Cockpitkonzepte. In *Anthropotechnik. gestern-heute-morgen, 40. Fachausschusssitzung Anthropotechnik der DGLR*. Deutsche Gesellschaft für Luft- und Raumfahrt e.V., 1998.

[ari99] *Airborne Weather Radar with Forward Looking Windshear Detection Capability- ARINC Characteristic 708A-3*, 1999.

[Ben97] P. A. Bennett. Applications of display prototyping and rehosting tools to the development of simulator, flight training device, and cockpit Displays. In *American Institute of Aeronautics and Astronautics*. Virtual Prototypes Inc., 1997.

[Bre99] U. Breymann. *C++ Eine Einführung; 5. aktualisierte und erweiterte Auflage*. Carl Hanser Verlag München Wien, 1999.

[Bro94] R. Brockaus. *Flugregelung*. Springer-Verlag Berlin Heidelberg, 1994.

[Cav00] Cavok. Formatspezifikation und Abschlussbericht Flugführungssicht Phase VI. Interns Bericht der Fachgebiet Flugmechanik und Regelungstechnik Technische Universität Darmstadt, 2000.

[Cav01] Cavok. Bericht zum Forschungsauftrag JANE: ”Ermittlung von Design-Kriterien für ein stereoskopisches Radardisplays”. Interns Bericht der Fachgebiet Flugmechanik und Regelungstechnik Technische Universität Darmstadt, 2001.

[Dat90] *http://www.geog.ubc.ca/courses/klink/gis.notes/ncgia/toc.html*, 1990.

[Eck97] G. Eckel. *IRIS Performer Programmer’s Guide*, 1997.

[Eng01] K. Engels. *Realisierung und Untersuchung der Kommunikationsstruktur einer Simulationsarchitektur für einen verteilten Forschungssimulator*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 2001.

[fN90] DIN (Deutsches Institut für Normung). Luft- und Raumfahrt; Begriffe, Größen und Formelzeichen der Flugmechanik; Bewegung des Luftfahrzeuges gegenüber der Luft; ISO 1151-1: 1988 modifiziert. In *DIN 9300 Teil 1*. Beuth Verlag GmbH Berlin, 1990.

[Fol98] T. Foltis. Entwicklung eines flexiblen Formatkonzeptes für 4D-Flugführungsdisplays. Diplomarbeit, Technische Universität Darmstadt, 1998.

[FvDFH95] J. D. Foley, A. van Dam, S. K. Feiner, und J. F. Hughes. *Computer Graphics: Principles and Practice second Edition in C*. Addison-Wesley Publishing Company, 1995.

[GGM99] J. M. Geib, C. Gransart, und P. Merle. *CORBA Des concepts à la pratique-2ème édition*. Dunod, Paris, 1999.

[GMK01] M. Gross, U. Mayer, und R. Kaufhold. Design of a perspective flight guidance display for a synthetic vision system. In *Enhanced and Synthetic Vision 1998*. Spie Proc., 2001.

[GMR00] J. Groeneweg, A. Marsman, und W. Rouwhorst. *Ethernet I/F Managment and Programming*. NLR, 2000. Referenz: ISAWARE-DC-4.2.7/NLR-V1.02xxx Restricted.

[Ham98] M. Hammer. *Stereoskopische Informationsdarstellung am Beispiel eines Flugführungsdisplays*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 1998.

[HDR96] Horst, Dieter, und Radke. *Computer Lexikon*. Augustus Verlag, Augsburg, 1996.

[Hel00] A. Helmetag. *Improvement of Perception and Cognition in Synthetic Spatial Environments*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 2000.

[HGK96] M. Hammer, M. Gross, und R. Kaufhold. Natürliche Displays im Cockpit. In *DGLR- Anthropotechnik Tagung*. Deutsche Gesellschaft für Luft- und Raumfahrt e.V., 1996.

[Höh97] T. Höhle. Untersuchung zum Einsatz eines Tools zur Displaygenerieurng. Studienarbeit, Technische Universität Darmstadt, Fachgebiet Flugmechanik und Regelungstechnik, 1997.

[HKK01] H. Herold, M. Klar, und S. Klar. *GoTo Objektorientierungte*. Addison-Wesley München, 2001.

[HKLP00] M. Hammer, R. Kaufhold, P. M. Lenhart, und M. Purpus. Flugerprobung im Rahmen des Forschungsvorhabens Flugführungssicht April 1997/September 1997. Interns Bericht der Fachgebiet Flugmechanik und Regelungstechnik Technische Universität Darmstadt, 2000.

[HKP99] A. Helmetag, R. Kaufhold, und M. Purpus. 3D Flight Guidance Displays. In *CEAS*, 1999.

[HLA] High Level Architecture. http://hla.dmso.mil.

[Ind89] Airbus Industrie. *A320 Flight deck and systems breifing for pilots*, 1989.

[JL00] Deutsche Übersetzung: Frank Langenau J. Liberty. *Jetzt lerne ich C++*. München: Mart und Technik Verlag, 2000.

[Kai03] J. Kaiser. *Verwendung stereoskopischer Informationsdarstellung in durchsichtfähigen Anzeigen am Beispiel eines Head-Up Displays*. Dissertation in vorbereitung, TU Darmstadt, Fachbereich Maschinenbau, 2003.

[Kau98] R. Kaufhold. *Konzeption und Erprobung der Visualisierung von Geländeinformation in Flugführungsanzeigen*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 1998.

[KO01] M. Keanne und K. O'Neil. Head Up Guidance at easyJet. In *http://www.aatl.net/publications/easyJet-HUGS.htm*. Flight Deck International, 2001.

[KS01] J. Kaiser und G. Smietanski. The European project ISAWARE. In *Enhanced and Synthetic Vision 2001*. Spie Proc., 2001.

[Kub96] Prof. Dr.-Ing. Kubbat. *Skriptum zur Vorlesung Prozessdatenverarbeitung-Teil 2*, 1996.

[Len02] P. M. Lenhart. *Räumliche Darstellung von Flugführungsinformationen in Head-Mounted Displays*. Dissertation in vorbereitung, TU Darmstadt, Fachbereich Maschinenbau, 2002.

[Lip99] K. Lipinski. *Lexikon der Datenkommunikation*. MITP-Verlag GmbH, Bonn, 1999.

[May01a] L. May. *Generierung und Verifikation von Geländedatenbanken für Luftfahrzeuge*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 2001.

[May01b] U. Mayer. Stereoskopische Flugführungsanzeigen. In *Seminar SE 3.16 Enhanced Vision: Erweiterte Sichtsysteme für zukünftige Cockpits ud Leitstände*. Carl-Cranz-Gesellschaft e.V., 2001.

[May03] U. Mayer. *Validierung stereoskopischer Informationsdarstellung in einer exozentrischen Synthetic-Vision-Flugführungsanzeige*. Dissertation in vorbereitung, TU Darmstadt, Fachbereich Maschinenbau, 2003.

[Met75] W. Metzger. *Gesetze des Sehens*. Verlag Waldemar Kramer in Frankfurt am Main, 1975.

[Mey90] B. Meyer. *Objektorientierte Softwareentwicklung*. München Hanser, 1990.

[Mey97] B. Meyer. *Object-Oriented Software Construction Second Edition*. Prentice Hall PTR-Upper Saddle River New Jersey, 1997.

[MKG99] U. Mayer, J. Kaiser, und M. Gross. AWARD's Synthetic Vision Flight Guidance Displays as Basic Display for a Free Flight Environmet. In *IFAC-1999*, 1999.

[Mul99] M. Mulder. *Cybernetics of tunnel-in-the-sky displays*. Dissertation, TU Delft, 1999.

[Mur93] R. B. Murray. *C++ Strategies and Tactics*. Addison-Wesley USA, 1993.

[NDW93] J. Neider, T. Davis, und M. Woo. *OpenGL Programming Guide The Official Guide to Learning OpenGL, Release 1*, 1993.

[(NL96] Netherlands National Aerospace Laboratory (NLR). An Analysys of Controlled-flight-into-terrain (CFIT) Accifents of Commercial Operators 1988 through 1994. In *The Leading Edge*, 1996.

[Oes99] B. Oestereich. *Objektorientierte Softwareentwicklung- Analyse und Design mit der Unified Modeling Language- 4.,aktuallisierte Auflage*. R. Oldenbourg Verlag München, 1999.

[O'R93] J. O'Rourje. *Computational Geometry in C*. Cambrige University press, 1993.

[PMG97] E. H. Page, R. L. Mosse, und Jr. S. P. Griffin. WEB-Based Simultaion In SIMJAVA Using Remote Method Invocation. In *Proceedings of the 1997 Winter Simulation Conference*, http://msie.org/page/papers/wsc/wsc97/wsc97.html, 1997.

[Puj95] G. Pujolle. *Les Réseaux*. Eyrolles, 1995.

[Pur99] M. Purpus. *Die Rolle der Prädiktion in dreidimensionalen Flugführungsdarstellungen*. Dissertation, TU Darmstadt, Fachbereich Maschinenbau, 1999.

[Rif94] J. M. Rifflet. *La Programmation sous UNIX 3e édition*. Ediscience International, 1994.

[RLS95] J. M. Reising, K. K. Ligget, und T. J. Solz. A comparison of two head up Display formats used to fly curved instrument approaches. 1995.

[SB01] H. J. Siegert und U. Baumgarten. *Computer Lexikon*. Technische Universität München, 2001.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[SKP$^+$97] G. Smietanski, R. Kaufhold, M. Purpus, M. Gross, U. Mayer, K. Engels, J. Schiefle, und A. Helmetag. Anzeigespezifikation; HGG Hochleistungsgrafikgenerator. 1997.

[SLKM00] G. Smietanski, P. M. Lenhart, S. Kranz, und U. Mayer. Development of the second generation of a Synthetic Vision Displays. In *Enhanced and Synthetic Vision 2000*. Spie Proc., 2000.

[Str97] T. Strasser. *C++ Programmieren mit Stil-Eine systematische Einfühtrung*. Dpunkt.verlag Heidelberg, 1997.

[Str00] B. Stroustrup. *The C++ Programming Language-Special Edition*. AT&T, 2000.

[Sys] Saxonia Systems. Remote Procedure Call. In *Die Linuxfibel*, http://www.linuxfibel.de/rpc.htm.

[Tan95] A. S. Tanenbaum. *Verteilte Betriebssystem*. Prentice Hall, 1995.

[The97] E. Theunissen. *Integrated Design of a Man-Machine Interface for 4-D Navigation*. Dissertation, TU Delft, 1997.

[Web00] J. Weber. Weiterführende Optimierung der Geländedarstellung auf der Grundlage des Schichtenmodells. Studienarbeit, Technische Universität Darmstadt, Fachgebiet Flugmechanik und Regelungstechnik, 2000.

[Wer94] J. Wernecke. *The Inventor Mentor- PRogramming Object-Oriented 3D Graphics with Open Inventor, Release 2*, 1994.

[XI95] TUM Informatik XI. Verteilte Anwendungen basierend auf RPC. http://www11.informatik.tu-muenchen.de/lehre/lectures/ss1994/va/-chap_3/verte.html, 1995. TU München.

Index

Lebenslauf

22. August 1969	geboren in Paris 13ème als Sohn des Französischlehrer Jacques Smietanski und seiner Ehefrau Künstlerin Johanna, geborene Bächler
1975 – 1979	Grundschule in Paris
1979 – 1980	Grundschule in Brignac
1980 – 1989	Gymnasium in Clermont l’hérault
1989	Abitur
1989 – 1991	Diplôme Universitaire de Technologie Génie Electrique Informatique Industrielle in Montpellier
1991 – 1993	Maîtrise des Sciences et Techniques Microinformatique Industrielle et Automatique in Annecy
1993 – 1994	Wehrdienst
1994 – 1995	Diplôme d’Etude Supérieures Spécialisées Ingenièrie des Systèmes in Corte
1996	Mitarbeiter bei Informatique Development in Croissy Beaubourg
1997	Mitarbeiter bei Alten in Paris
1997 – 2002	Wissenschaftlicher Mitarbeiter am Fachgebiet Flugsysteme und Regelungstechnik der Technischen Universität Darmstadt
seit Aug. 2002	Mitarbeiter bei EADS Dornier in Friedrichshafen