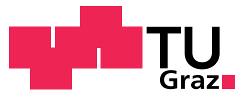


Andreas Schuller, Bakk.rer.soc.oec.

Entwicklung eines *Human Machine Interface* für die Bodenlagedarstellung zum Zweck der Flugsicherung

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Softwareentwicklung-Wirtschaft

eingereicht an der

Technischen Universität Graz

Betreuer

Ao. Univ.-Prof. Dipl.-Ing. Dr.techn. Erich Leitgeb

Institut für Hochfrequenztechnik

Graz, Juni 2019

Abstract

The strong increase in air traffic in recent decades has led to the development of computer-aided control systems. The use of an Advanced Surface Movement Guidance and Control System (A-SMGCS) gives air traffic controllers situational awareness over all aircraft and vehicle movements on the aerodrome even under bad weather conditions. The requirements posed on the Human Machine Interface (HMI) of an A-SMGCS have increased since introduction of such systems.

In this master thesis a prototype for the further development of the A-SMGCS HMI of ADB Safegate Austria is implemented to meet these growing requirements. The implementation of the existing A-SMGCS HMI is based on conventional GUI technology (Qt 3) and does not make use of graphics hardware acceleration. The developed prototype, however, uses Qt Quick (Qt 5) and OpenGL to leverage hardware-accelerated rendering. The functionality of the prototype includes the display of surface movement radar and track data in superimposed form with the airport map. The prototype is able to receive the A-SMGCS data provided by the A-SMGCS server via the network and to display the ground situation simultaneously in several windows for different areas of the airport.

The theoretical part of this thesis gives a short overview of the general concept of an A-SMGCS and its implementation by ADB Safegate Austria. The essential differences between the conventional GUI technology and Qt Quick are worked out and methods for the efficient texture data transfer with OpenGL – for the display of the surface movement radar image – are presented. Relevant work that dealt with the implementation of radar displays with the help of the graphics hardware is referred to.

The practical part is dedicated to the development of the HMI prototype on the basis of the described requirements and presents the essential aspects of the implementation. The factors identified in the analysis of the ACEMAX-HMI, which limit the performance of the ACEMAX HMI, are addressed in the design. Different approaches are shown and discussed concerning the presentation of the airport map, the surface movement radar image and the targets with Qt Quick.

The performance of the prototype is evaluated by means of a test bed developed in this master thesis. The HMI performance parameters defined in the ED-87C standard of the European Organisation for Civil Aviation Equipment (EUROCAE) are used for the evaluation. The results imply that the chosen approaches are suitable for implementing an HMI which is founded on the Qt Quick GUI technology and which meets the performance requirements for the A-SMGCS data considered in this thesis.

Kurzfassung

Die starke Flugverkehrszunahme hat in den letzten Jahrzehnten zur Entwicklung computergestützter Kontrollsysteme geführt. Durch Einsatz eines Advanced Surface Movement Guidance and Control System (A-SMGCS) können Fluglotsen auch bei schlechten Wetterverhältnissen die Flug- und Fahrzeugbewegungen am Flughafen verfolgen. Die Anforderungen an das Human Machine Interface (HMI) eines A-SMGCS haben sich seit Einführung derartiger Systeme erhöht.

In dieser Masterarbeit wird ein Prototyp für die Weiterentwicklung des A-SMGCS HMI der Firma ADB Safegate Austria implementiert, um diesen wachsenden Anforderungen gerecht zu werden. Die Implementierung des bestehenden A-SMGCS HMI basiert auf konventioneller GUI-Technologie (Qt 3) ohne Grafikkartenbeschleunigung. Der entwickelte Prototyp verwendet hingegen Qt Quick (Qt 5) und OpenGL und setzt damit auf hardwarebeschleunigtes Rendern. Die Funktionalität des Prototypen umfasst die Anzeige der Bodenradar- und Track-Daten, in überlagerter Darstellung mit der Karte des Flughafens. Der Prototyp ist in der Lage, die vom A-SMGCS Server zur Verfügung gestellten A-SMGCS Daten über das Netzwerk zu empfangen und die Bodenlage gleichzeitig in mehreren Fenstern, für unterschiedliche Bereiche des Flughafens, darzustellen.

Der theoretische Teil der vorliegenden Arbeit gibt einen kurzen Überblick über das allgemeine Konzept eines A-SMGCS und dessen Umsetzung durch die Firma ADB Safegate Austria. Es werden die wesentlichen Unterschiede zwischen der konventionellen GUI-Technologie und Qt Quick herausgearbeitet und Methoden zum effizienten Texturdatentransfer mit OpenGL – für die Anzeige des Bodanradarbildes – vorgestellt. Auf relevante Arbeiten, die sich mit der Umsetzung von Radar-Displays unter Zuhilfenahme der Grafikhardware befassten, wird ebenfalls Bezug genommen.

Der praktische Teil widmet sich der Entwicklung des HMI-Prototypen auf Basis der beschriebenen Anforderungen und legt die wesentlichen Aspekte der Implementierung dar. Die in der Analyse des ACEMAX-HMI identifizierten Faktoren, welche die Leistungsfähigkeit des ACEMAX HMI begrenzen, werden im Entwurf thematisiert. Für die Darstellung der Flughafenkarte, des Bodenradarbildes und der Ziele mit Qt Quick werden verschiedene Ansätze aufgezeigt und diskutiert.

Die Leistungsfähigkeit des Prototypen wird mittels einer in dieser Masterarbeit entwickelten Testumgebung geprüft. Zur Evaluierung werden die im ED-87C Standard der European Organisation for Civil Aviation Equipment (EUROCAE) definierten Leistungsparameter herangezogen. Die Ergebnisse lassen darauf schließen, dass die im Prototypen gewählten Ansätze geeignet sind, um ein HMI zu implementieren, das auf der GUI-Technlogie Qt Quick basiert und das den Performance-Anforderungen für die in dieser Arbeit betrachteten A-SMGCS Daten genügt.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum

Unterschrift

Danksagung

Ich möchte mich an dieser Stelle bei Herrn Dr. Konrad Köck und Herrn Dr. Arnold Maier von der Firma ADB Safegate Austria für die Ermöglichung dieser Masterarbeit bedanken. Danke an dieser Stelle auch an alle Arbeitskollegen für die gute Zusammenarbeit und deren Feedback.

Weiters bedanke ich mich bei meinem Betreuer Ao. Univ.-Prof. Dr. Erich Leitgeb, der mich während der Masterarbeit immer unterstützt und mit seinem Fachwissen gefördert hat.

Einen besonderen Dank möchte ich an meine Eltern Margit und Otto richten, die mir das Studium an der TU Graz ermöglichten. Sie und meine Schwester Sabine haben mich auch in schwierigen Phasen immer wieder ermutigt am Abschluß meines Studiums festzuhalten.

Nicht zuletzt gilt mein Dank Nina und allen weiteren Freunden, die mir während der Masterarbeit mit Rat und Tat zur Seite standen.

Graz, im Juni 2019 Andreas Schuller

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation und Aufgabenstellung 2
1.2 Aufbau der Arbeit 3

2 Problembeschreibung **4**
2.1 Flugsicherung 4
2.1.1 Organisationen und Behörden 4
2.1.2 Flugsicherungsdienste 6
2.1.3 Flugverkehrskontrolle 7
2.2 A-SMGCS im Allgemeinen 8
2.3 ACEMAX – Systemübersicht 10
2.3.1 Zielverfolgung 11
2.3.2 Bodenradardatenverarbeitung 12
2.3.3 HMI – Anwendersicht 13
2.3.4 HMI – Techniksicht 16
2.4 Anforderungen 20
2.4.1 Rahmenbedingungen 20
2.4.2 Funktionale Anforderungen 21
2.4.3 Nicht-funktionale Anforderungen 21
2.5 Analyse des bestehenden HMI 23
2.5.1 Messung der CPU-Last 23
2.5.2 Resultate und Schlussfolgerungen 25

3 Methoden und Technologien **27**
3.1 Qt 5 27
3.1.1 Konventionelles GUI-System 27
3.1.2 Qt Quick 28
3.2 Effizienter Texturdatentransfer mit OpenGL 32
3.3 Relevante Arbeiten 34

4 Entwurf und Implementierung **37**
4.1 Architektur des HMI-Prototypen 37
4.1.1 Datenschnittstellen und -quellen 37
4.1.2 Modulübersicht 38
4.2 Aufbau der Benutzeroberfläche 40
4.2.1 Fenster 41
4.2.2 View 41

4.3 Anzeige der Flughafenkarte . . . 43
4.3.1 Auswahl des Lösungsansatzes . . . 43
4.3.2 Implementierung . . . 45
4.4 Anzeige des Radarbildes . . . 46
4.4.1 Auswahl des Lösungsansatzes . . . 47
4.4.2 Implementierung . . . 49
4.5 Anzeige der Ziele . . . 53
4.5.1 Auswahl des Lösungsansatzes . . . 54
4.5.2 Implementierung . . . 55

5 Resultate **60**
5.1 ED-87C Parameter . . . 62
5.2 CPU-Last und Framerate . . . 63
5.3 Asynchroner Texturdatentransfer . . . 65

6 Fazit und Ausblick **67**

Abkürzungsverzeichnis **69**

Literaturverzeichnis **71**

Appendizes **74**
A.1 CPU-Last des ACEMAX-HMI – Testablauf und statistische Zusammenfassung 74
A.2 Ergebnisse der Leistungsbewertung des HMI-Prototypen . . . 76

Abbildungsverzeichnis

1.1 HMI des Produkts ACEMAX von ADB Safegate [1] . . . 1
1.2 HMI des Produkts INFOMAX von ADB Safegate [1] . . . 2

2.1 Flugverkehrskontrolle (Air Traffic Control, ATC) als Teil der Flugsicherungsbetriebsdienste . . . 6
2.2 Vergleich zwischen einem A-SMGCS-Display und einem klassischen Radarsichtgerät . . . 10
2.3 Funktionale Architektur des im Forschungsprojekt DEFAMM entwickelten A-SMGCS-Prototyps . . . 10
2.4 Client-Server-Architektur von ACEMAX . . . 11
2.5 Radardatenverarbeitung in ACEMAX . . . 13
2.6 Boden- und Luftlagedarstellung im ACEMAX-HMI . . . 14
2.7 Radarecho (orange) mit synthetischem Afterglow (gelb) . . . 15
2.8 Zieldarstellung . . . 16
2.9 Grafik-Stack des ACEMAX-HMI . . . 17
2.10 Komposition des Bodenlagebildes . . . 17
2.11 Verarbeitungsschritte zum Rendern des SMR-Bildes im ACEMAX-HMI . . . 19
2.12 Prinzipielle Render-Prozedur zur Zieldarstellung im ACEMAX-HMI . . . 20
2.13 Synthetisch generiertes SMR-Video . . . 24
2.14 Fensteranordnung und Anzeigeeinstellungen des ACEMAX-HMI bei der Durchführung der Performance-Tests . . . 25
2.15 Durchschnittliche CPU-Last des ACEMAX-HMI . . . 26

3.1 “Hello World” GUI-Anwendung mit Qt Quick . . . 29
3.2 Verarbeitungsschritte während eines Renderdurchgangs . . . 31
3.3 Synchroner Texturdatentransfer . . . 32
3.4 Asynchroner Texturdatentransfer unter Verwendung von PBOs im Round-Robin-Verfahren . . . 33
3.5 Asynchroner Texturdatentransfer mittels Worker-Thread und Context-Sharing 34

4.1 TCP/IP basierte Datenschnittstellen zwischen dem HMI-Prototypen und den zur Auswahl stehenden Datenquellen . . . 38
4.2 Modulübersicht des HMI-Prototypen . . . 39
4.3 GUI-Komponenten des Hauptfensters und eines Zoom-Fensters . . . 41
4.4 Realisierung der View-Komponente: Klassendiagramm und beispielhafter QML-Code . . . 43
4.5 Anwendung des Qt Quick-Items `Map` . . . 46
4.7 Fragment-Shader-Programm (GLSL) für die Erzeugung des SMR-Bildes . . . 52
4.8 Anwendung des Qt Quick-Items `SMRImage` . . . 53

4.9 Gegenüberstellung von Ansatz (a) und (b) zur Beschreibung der Ziele im Qt Quick-Szenengraphen 57
4.10 Anwendung des Qt Quick-Items `TrackView` 59

5.1 Testaufbau zur Performance-Evaluierung des HMI-Prototypen 60

1 Einleitung

Die Technik in der zivilen Luftfahrt hat sich seit ihrer Entstehung Anfang des 20. Jahrhunderts rasant weiterentwickelt, was nicht nur zu einem Anstieg in der Produktion von Flugzeugen führte, sondern auch die Zahl der Flugbewegungen deutlich ansteigen ließ. Während man 1960 noch unter zwei Millionen Flugbewegungen in Europa zählen konnte, sind es seit 2012 schon über zehn Millionen Flugbewegungen pro Jahr (siehe [9]). Diese starke Verkehrszunahme hat in den letzten Jahrzehnten zur Entwicklung computergestützter Kontrollsysteme geführt, die Fluglotsen bei ihrer Arbeit unterstützen sollen.

Zur Kontrolle der Bewegungen am Flughafen und des Verkehrs im angrenzenden Luftraum wurde das Konzept eines "Advanced Surface Movement Guidance and Control System" (A-SMGCS) [23] entworfen, das auch bei schlechten Wetterverhältnissen ein detailliertes Bild der Boden- und Luftlage garantieren soll. Das Konzept umfasst:

- **verschiedene Überwachungssensoren**, zur Positionsbestimmung und Identifikation von Flug- und Fahrzeugen;
- **ein Computersystem**, das nicht nur die Sensordaten verarbeitet, sondern zusätzliche Funktionen - wie Routenplanung und Konfliktkennung - bietet und den Fluglotsen ein Human Machine Interface (HMI) zur Verfügung stellt.

Ein solches Computersystem wird von der Firma ADB Safegate Austria [1] unter dem Namen ACEMAX entwickelt und vertrieben. Das HMI von ACEMAX, dargstellt in Abbildung 1.1, ist Gegenstand dieser Arbeit.

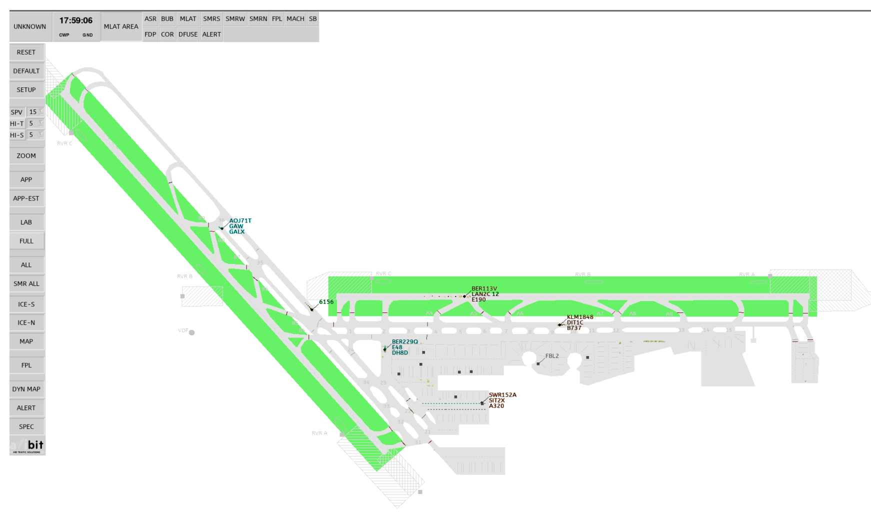


Abbildung 1.1: HMI des Produkts ACEMAX von ADB Safegate [1]

1.1 Motivation und Aufgabenstellung

Die Firma ADB Safegate Austria setzt bei der Implementierung ihrer Softwarelösungen auf das plattformübergreifende Software Developement Kit (SDK) Qt [7]. Die Codebasis von ACEMAX beruht auf der veralteten Version Qt 3 und wird daher auf Qt 5 umgestellt. Mit der Portierung soll auch ein Umstieg des von Qt nicht mehr weiterentwickelnden QWidget-Systems auf Qt Quick realisiert werden. Die neue GUI-Technologie baut auf OpenGL [38] auf und ermöglicht dadurch hardwarebeschleunigtes Rendern (Bildsynthese) der grafischen Benutzeroberfläche.

Man verspricht sich von der neuen Technologie unter anderem:

- ein moderneres GUI-Design,
- eine bessere Wiederverwendbarkeit der HMI-Komponenten zwischen verschiedenen ADB Safegate Austria Produkten und
- eine bessere HMI-Performance, gemessen an der Anzahl darzustellender Flug- und Fahrzeuge.

Erste Erfahrung mit Qt Quick konnte ADB Safegate Austria bereits bei der Implementierung eines neuen HMI für das Produkt INFOMAX [1] sammeln, siehe Abbildung 1.2. Der Einsatz von Komponenten des INFOMAX-HMI im ACEMAX-HMI, und vice versa, soll in Zukunft möglich sein.

Abbildung 1.2: HMI des Produkts INFOMAX von ADB Safegate [1]

Die Frage, inwieweit sich Qt Quick als Basistechnologie für das ACEMAX-HMI eignet, soll in dieser Arbeit beantwortet werden. Dazu soll ein HMI-Prototyp entworfen werden, der folgende Grundanforderungen erfüllt: Ein Hauptfenster und bis zu vier Zoom-Fenster, die Darstellung der Flughafenkarte, der Ziele und des Bodenradarbildes sowie die Ansichtstransformationen (Zoom, Verschieben und Rotieren) und die Darstellung von mindestens 500 Zielen pro Fenster. Auf die detaillierte Anforderungen wird in Kapitel 2 näher eingegangen.

1.2 Aufbau der Arbeit

Der erste Teil der vorliegenden Masterarbeit umfasst die Beschreibung der Problemstellung und erarbeitet alle für die Entwicklung des HMI-Prototyps notwendigen Grundlagen. Kapitel 2 geht zunächst auf die Flugsicherung als Anwendungsdomäne ein und erläutert das Konzept eines A-SMGCS. In weiterer Folge wird das von der Firma ADB Safegate Austria entwickelte A-SMGCS vorgestellt, wobei der Fokus auf das HMI gelegt wird. Schlussendlich wird die Aufgabenstellung durch Nennung der an den HMI-Prototyp gestellten Anforderungen präzisiert und die Problembeschreibung mit einer Analyse der Leistungsfähigkeit des bestehenden ACEMAX-HMI abgeschlossen. In Kapitel 3 werden Technologien und Methoden, die für die Entwicklung des HMI-Prototypen maßgeblich sind, vorgestellt und es werden Ansätze anderer relevanter Arbeiten zusammengefasst.

Der zweite Teil der Masterarbeit beschreibt die technische Umsetzung des HMI-Prototypen (Kapitel 4 bis 6). In Kapitel 4 wird der Entwurf und die Implementierung des HMI-Prototypen beschrieben. Es werden verschiedene Lösungsansätze aufgezeigt, das Design des HMI-Prototypen wird begründet und die wichtigsten Aspekte der Implementierung werden hervorgehoben. Die Leistungsfähigkeit des HMI-Prototypen wird in Kapitel 5 beurteilt und mit der des bestehenden HMI verglichen. In Kapitel 6 wird schließlich ein Fazit bezüglich der Eignung der im HMI-Prototypen implementierten Ansätze zur Entwicklung eines neuen ACEMAX-HMI gegeben, auf noch zu lösende Probleme hingewiesen und ein Ausblick auf die Weiterentwicklung des ACEMAX-HMI geworfen.

Der Anhang dokumentiert den Testablauf und die Testergebnisse der in dieser Arbeit durchgeführten Performance-Messungen.

2 Problembeschreibung

In diesem Kapitel wird zunächst eine kurze Übersicht der wichtigsten Organisationen, Behörden und Dienste der Flugsicherung gegeben. Anschließend wird das Konzept eines A-SMGCS erklärt und die Implementierung der Firma ADB Safegate Austria in seinen Grundzügen umrissen. Zum besseren Verständnis der Aufgabenstellung werden wesentliche Aspekte des bestehenden HMI im Detail diskutiert. Das Hauptaugenmerk richtet sich dabei auf die darzustellenden Daten, den Aufbau und die technische Umsetzung der grafischen Benutzeroberfläche. Insbesondere werden die darzustellenden Daten, der Aufbau und die technische Umsetzung der grafischen Benutzerschnittstelle erläutert. Abschließend wird die Problemstellung durch Auflistung aller Anforderungen konkretisiert und eine Analyse der Leistungsfähigkeit des bestehenden HMI durchgeführt.

2.1 Flugsicherung

Die Flugsicherung dient gemäß österreichischem Luftfahrtgesetz (§ 119 LFG) "der sicheren, geordneten und flüssigen Abwicklung des Flugverkehrs" [30]. Zur Vermeidung von Kollisionen in der Luft, zur effizienten, sicheren und kontrollierten Abwicklung von Flügen über Landesgrenzen hinweg, ist eine Kontrolle der sich am Boden und in der Luft befindlichen Luftfahrzeuge erforderlich. In diesem Kapitel werden die sich mit der Erfüllung dieser Aufgabe verantwortlich zeigenden Behörden und Organisationen, auf internationaler und nationaler Ebene, vorgestellt. Es wird auf die Flugsicherungsdienste und insbesondere auf den Flugverkehrskontrolldienst, der den engeren Rahmen dieser Arbeit vorgibt, eingegangen.

2.1.1 Organisationen und Behörden

Es folgt eine Übersicht über die wichtigsten Organisationen und Behörden auf internationaler, europäischer und nationaler Ebene, die für die Standardisierung, Forschung und gesetzliche Regelung im Bereich der Flugsicherung maßgebend sind.

2.1.1.1 Internationale Zivilluftfahrtorganisation

Die Regulierung des grenzübergreifenden Luftverkehrs begann mit dem Chicagoer-Abkommen über die internationale Zivilluftfahrt im Jahre 1944, das von 52 Staaten unterzeichnet wurde. Mit der Ratifizierung des Abkommens entstand die Internationale Zivilluftfahrtorganisation (International Civil Aviation Organisation, ICAO) [24], welche als Sonderorganisation der Vereinten Nationen geführt wird. Eine der Hauptaufgaben dieser Organisation ist die Entwicklung eines weltweit einheitlichen Systems für die zivile Luftfahrt. Für diesen Zweck

werden Standards und Empfehlungen erarbeitet. Die sogenannten "Standard and Recommended Practices" (SARPs) sind Teil des Abkommens und werden als Annexe publiziert, siehe auch Abbildung 2.1. SARPs erlangen ihre Rechtsverbindlichkeit allerdings erst dann, wenn sie in das nationale Recht der Mitgliedstaaten eingearbeitet wurden. Befolgt ein Mitglied gewisse SARPs nicht, müssen die daraus resultierenden Abweichungen vom betroffenen Staat veröffentlicht werden. Diese Abweichungen sind dann als "Non-standard Practise" in den ICAO-Dokumenten ersichtlich. (vgl. [16, S. 5 f.])

Annex 1 – Personnel Licensing
Annex 2 – Rules of the Air
Annex 3 – Meteorological Service for International Air Navigation
Annex 4 – Aeronautical Charts
Annex 5 – Units of Measurement to be used in Air and Ground Operations
Annex 6 – Operation of Aircraft
Annex 7 – Aircraft Nationality and Registration Marks
Annex 8 – Airworthiness of Aircraft
Annex 9 – Facilitation
Annex 10 – Aeronautical Telecommunications
Annex 11 – Air Traffic Services
Annex 12 – Search and Rescue
Annex 13 – Aircraft Accident and Incident Investigation
Annex 14 – Aerodromes
Annex 15 – Aeronautical Information Services
Annex 16 – Environmental Protection
Annex 17 – Security
Annex 18 – The Safe Transport of Dangerous Goods by Air
Annex 19 – Safety Management

Tabelle 2.1: Die 19 Anhänge des Abkommens (Stand Oktober 2014)

2.1.1.2 Europäische Organisation für Flugsicherheit

Die Europäische Organisation für Flugsicherheit (European Organisation for Safety of Air Navigation, EUROCONTROL) [10] wurde im Jahre 1960 von mehreren europäischen Staaten – darunter befanden sich unter anderem Deutschland, Frankreich und das Vereinigte Königreich – ins Leben gerufen. Die Ziele der Anfangszeit unterscheiden sich nicht wesentlich von den heutigen Zielen, nämlich die gemeinsame Organisation des Luftraumes und der Betrieb gemeinsamer Flugsicherungsdienste. Die Forschung und Entwicklung zur Steigerung der Sicherheit und Effizienz des Luftverkehrs in Europa ist eine weitere Kernaufgabe dieser Organisation. (vgl. [32, S. 13 ff.])

2.1.1.3 Vorgaben der Europäischen Union

Die Europäische Union (EU) stellt durch Verordnungen sicher, dass einheitliche Standards und Sicherheitsbestimmungen von jedem Mitgliedsland umgesetzt werden. Ein Anliegen ist

beispielsweise die Schaffung eines einheitlichen europäischen Luftraumes, dem sogenannten "Single European Sky". Das EU-Projekt "Single European Sky ATM Research Programme" (SESAR) soll dabei unter anderem für eine Erhöhung der Sicherheit, der Kapazität und der Umweltverträglichkeit sorgen. (vgl. [16, S. 6 f.])

2.1.1.4 Nationale Behörden und Organisationen

In Österreich und Deutschland sind das Verkehrsministerium und das Verteidigungsministerium für die Luftfahrt verantwortlich [16, S. 8]. Die Aufgaben der Flugsicherung werden, wie auch in vielen anderen Ländern, an untergeordnete Organisation oder Einrichtungen delegiert. Laut österreichischem Luftfahrtgesetz, LFG §120 Abs. 1, obliegt die Wahrnehmung der Flugsicherung in Österreich der Austro Control GmbH. Organisationen dieser Art werden als Flugsicherungsdienstbetreiber (Air Navigation Service Providers) bezeichnet.

Die Aufgaben der Flugsicherung werden organisatorisch auf mehrere Flugsicherungsdienste aufgeteilt, die vom Flugsicherungsdienstbetreiber per gesetzlichem Auftrag anzubieten sind. Die wichtigsten Flugsicherungsdienste werden nachfolgend zusammengefasst. Auf die Flugverkehrskontrolle, welche für diese Arbeit maßgeblich ist, wird am Ende des Kapitels noch gesondert eingegangen.

2.1.2 Flugsicherungsdienste

Die zu erbringenden Dienstleistungen gründen sich auf die von der ICAO erarbeiteten Standards und Empfehlungen und lassen sich, wie in Abbildung 2.1 grafisch dargestellt, in fünf große Dienstleistungsbereiche einteilen [16, S. 13]. Die nachstehende Beschreibung der Dienste bezieht sich auf die Ausführungen von Flühr [16, S. 13 ff.].

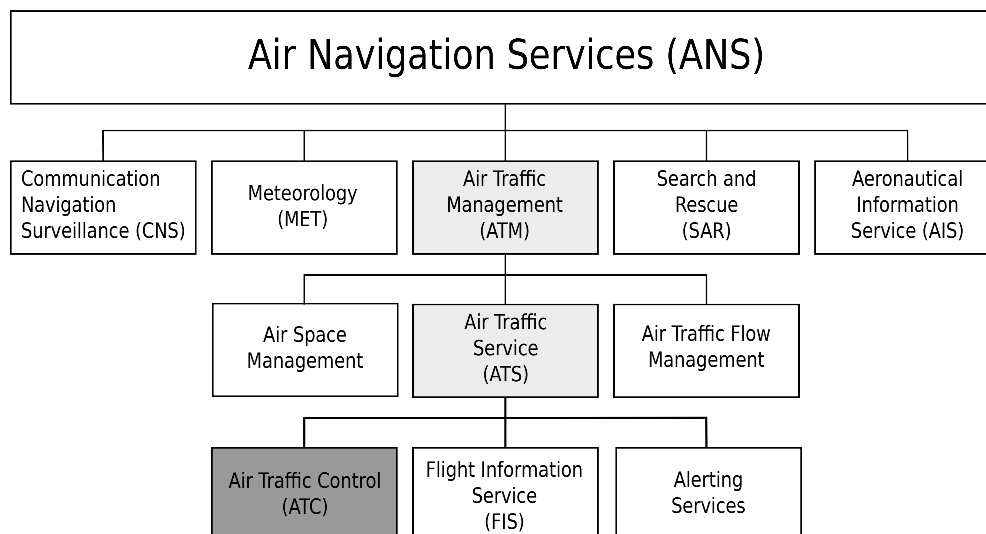


Abbildung 2.1: Flugverkehrskontrolle (Air Traffic Control, ATC) als Teil der Flugsicherungsbetriebsdienste; modifiziert übernommen aus [16]

Die als **Kommunikations-, Navigations- und Überwachungssysteme** (Communication Navigation Surveillance, CNS) zusammengefassten Dienste sichern den Betrieb der technischen Flugsicherungssysteme (e.g. Sprechfunksysteme, Radionavigationssysteme und Radaranlagen). Der **meteorologische Dienst** (Meteorology, MET) umfasst unter anderem Dienstleistungen zur Flugwetterüberwachung, zur Erstellung von Wettervorhersagen und zur Herausgabe von Warnungen über Wettererscheinungen.

Der **Flugberatungsdienst** (Aeronautical Information Service, AIS) sammelt und stellt Informationen bereit, die für eine sichere, geordnete und effiziente Flugdurchführung notwendig sind. Dies umfasst unter anderem die Entgegennahme, Verarbeitung und Weiterleitung von Flugplänen, die Herausgabe des Luftfahrthandbuch (Aeronautical Information Publication, AIP) und die Bekanntmachung von Anordnungen und Informationen über temporäre Änderungen des Luftfahrthandbuches (Notice to Airman, NOTAM). Der **Flugalarmdienst** (Search and Rescue, SAR) ist für die Suche und Rettung von Menschen nach einem Flugzeugunglück verantwortlich.

Das **Flugverkehrsmanagement** (Air Traffic Management, ATM) lässt sich noch weiter unterteilen in **Luftraummanagement** (Air Space Management), in **Verkehrsflussmanagement** (Air Traffic Flow Management, ATFM) und in **Flugverkehrsdienste** (Air Traffic Service, ATS). Das Luftraummanagement sorgt für eine Aufteilung des verfügbaren Luftraumes auf zivile nicht gewerbliche, zivile gewerbliche und militärische Nutzer. Das Verkehrsflussmanagement hingegen sichert den optimalen Verkehrsfluss im Luftraum unter Berücksichtigung der vorhandenen Kapazität.

Zu den Flugverkehrsdiensten gehören die **Flugverkehrskontrolle** (Air Traffic Control, ATC), der **Fluginformationsdienst** (Flight Information Service, FIS) und die **Alarmdienste** (Alerting Service). Der Fluginformationsdienst stellt, im Unterschied zum Flugberatungsdienst, Informationen während des Fluges bereit (z. B. über eine Sprechfunkverbindung). Zu den primären Aufgaben der Alarmdienste zählt die Benachrichtigung der Rettungsleitstellen und Notfallorganisationen im Falle eines sich in Not befindlichen Flugzeuges.

2.1.3 Flugverkehrskontrolle

Die Flugverkehrskontrolle ist "die Überwachung und Lenkung der Bewegungen im Luftraum und auf den Rollfeldern von Flugplätzen mit Flugplatzkontrolle zur sicheren geordneten und flüssigen Abwicklung des Luftverkehrs" [32, S. 71]. Die obersten Ziele dabei sind die Verhinderung von Kollisionen von Luftfahrzeugen mit anderen Luftfahrzeugen, mit Fahrzeugen oder Hindernissen auf dem Rollfeld.

Die Aufgabe der Flugverkehrskontrolle wird, entsprechend den einzelnen Phasen des Fluges, auf mehrere Kontrollstellen aufgeteilt. Die Untergliederung ist mitunter abhängig von der Größe des Flugplatzes und des Verkehrsaufkommens. Grundsätzlich lässt sich aber folgende Aufteilung treffen (vgl. [8, 16, 32]):

- Tower-Kontrolle (Tower Control): Diese Kontrollstelle ist örtlich im Kontrollturm (Tower) des Flugplatzes untergebracht und hat die Überwachung der Rollbahnen, der Rollwege, des Vorfeldes und der Parkflächen zur Aufgabe. In ihrem Verantwortungsbereich liegen die aktiven Start- und Landebahnen, das heißt sie lenkt und koordiniert die Bewegungen startender und landender Flugzeuge.

- Anflugkontrolle (Approach Control): Abfliegende Flugzeuge, die sich bereits in der Luft befinden, werden von der Tower-Kontrolle übergeben und von der Anflugkontrolle zu den Flugstraßen geführt. Anfliegende Flugzeuge werden entsprechend der festgelegten Anflugreihenfolge an den Flugplatz herangeführt und vor der Landung wiederum an die Tower-Kontrolle übergeben.
- Streckenflugkontrolle (Area Control): Für den En-route-Verkehr (Verkehr auf den Flugstraßen) ist die Streckenflugkontrolle verantwortlich.

Der Dienst in den Kontrollstellen wird von Fluglotsen (Air Traffic Control Officer, ATCO) durchgeführt. Die Tätigkeit eines Fluglotsen wird dabei durch zahlreiche technische Systeme unterstützt.

Für die Überwachung des Luftraumes und des Flugplatzes werden radargestützte und andere Sensoren eingesetzt. Die Sensordaten werden weiter verarbeitet und die daraus gewonnene Position wird zusammen mit dem Radarbild auf einem elektronischen Bildschirm, am Arbeitsplatz des Fluglotsen (Controller Working Position, CWP), dargestellt.

Fortschrittliche Systeme, wie das der Firma ADB Safegate Austria (siehe Abschnitt 2.3), sind in der Lage, Daten von mehreren Sensoren zu verarbeiten und bieten neben der Luft- und Bodendarstellung auch zusätzliche Funktionen, wie beispielsweise die Anzeige von Flugplaninformationen und die Ausgabe von Kollisionswarnungen.

2.2 A-SMGCS im Allgemeinen

Ein Bodenverkehrsleit- und Kontrollsystem (Surface Movement and Guidance System, SMGCS), das den Einsatz von Bodenmarkierungen, Beschilderungen, sowie schaltbaren Haltebalken (Stop-bars) und Rollweglichtern vorsieht, wurde bereits 1986 von der ICAO konzipiert [22]. Die Überwachung und Kontrolle der Bodenbewegungen erfolgt hierbei in erster Linie durch direkten Sichtkontakt und kann durch ein Bodenradar zusätzlich unterstützt werden. Bei eingeschränkter Sicht treten zur Gewährleistung der Sicherheit jedoch zusätzliche Verfahrensanweisungen in Kraft, die die Verkehrsabwicklung verlangsamen [23].

Das von der ICAO im Jahr 2004 veröffentlichte Konzept für ein *Advanced Surface Movement and Guidance System* (A-SMGCS) [23], soll hingegen eine flüssige und sichere Verkehrsabwicklung – insbesondere unter schlechten Sichtverhältnissen garantieren. Das Konzept beruht auf der funktionalen Architekur des im Forschungsprojekt Demonstration Facilities for Aerodrome Movement Management (DEFAMM) entwickelten A-SMGCS-Prototyps (vgl. Abbildung 2.3) und sieht vier verschiedene Services für ein A-SMCGCS vor:

- Surveillance-Service zur Positionsbestimmung, Identifikation und Zielverfolgung von Flug- und Fahrzeugen;
- Routing-Service zur automatischen Routenplanung;
- Guidance-Service zur Lenkung von Flugzeugen durch Steuerung der Rollfeldbeleuchtung entsprechend der ausgewählten Route und zur Weitergabe der Bodenlageinformation sowie anderer Information an das Flughafenpersonal, an Fahrzeuglenker und Piloten;
- Airport-Safety-Support-Service zur Konfliktwarnung (z. B. wenn sich ein Flugzeug auf der Landebahn eines sich im Landeanflug befindlichen Flugzeuges aufhält) und zur

Bereitstellung von Konfliktlösungen.

Das Surveillance-Service ist ein fixer Bestandteil (Basisfunktion) jedes A-SMGCS. Die dabei für die Überwachung des Luftraumes und der Bewegungen am Flughafen verwendeten technischen Systeme werden als Surveillance-Sensoren bezeichnet. Abhängig vom Funktionsprinzip können Surveillance-Sensoren neben der Lageinformation auch andere Informationen über ein Flug- oder Fahrzeug zur Verfügung stellen. Surveillance-Sensoren lassen sich dabei in kooperative und nicht kooperative Sensoren einteilen.

Als Kooperative Sensoren bezeichnet man jene, die von einem kooperativen Flug- oder Fahrzeug Daten empfangen (e.g. Kennung/Adresse des Transponders). Die Daten werden hierbei über einen an Bord befindlichen Transponder abgestrahlt. Beispiele für kooperative Sensoren sind Sekundärradarsysteme und Multilaterationssysteme (siehe [16, S. 158, 211]). Nicht kooperative Sensoren werden komplementär eingesetzt, um beispielsweise am Flughafen Hindernisse und Fahrzeuge ohne Transponder detektieren zu können. Als bedeutsames Beispiel für einen nicht kooperativen Sensor ist das Bodenradar (Surface movement radar, SMR) zu nennen, das für die Überwachung des Rollfeldes verwendet wird. Dieses zählt zur Klasse der Primärradaranlagen, die Objekte zwar orten aber nicht identifizieren können. Über die rotierende Antenne werden gebündelte hochfrequente Radarimpulse ausgesendet. Trifft ein Impuls auf ein Objekt wird ein Teil der Energie reflektiert. Das reflektierte Signal wird von der Radarantenne an einen Empfänger weitergeleitet und durch Laufzeitmessung des ausgestrahlten und wieder reflektierten Impulses (Radarecho) wird die Entfernung des Objektes bestimmt. Die Richtung des Objektes ergibt sich aus der Winkelposition der Antenne. (vgl. [32, S. 340 ff.]; [12, 14])

Gemäß der von der EUROCONTROL bereitgestellten Spezifikation [13], geht mit dem Surveillance-Service die Einführung eines automatisierten Systems zur synthetischen Repräsentation der Bodenlage einher. Das an der CWP vorgesehene HMI ermöglicht die Positionsbestimmung und die Identifikation von Flug- und Fahrzeugen auf den Bewegungsflächen und unterstützt somit die Fluglotsen in ihrer Arbeit. Zur relativen Lagebestimmung ist auch die geografische Repräsentation von Teilen des Flughafens im HMI erforderlich. Der Unterschied eines traditionellen Radarbildschirms zu dem eines A-SMGCS wird in Abbildung 2.2 deutlich. Das A-SMGCS-Display zeigt zusätzlich zum Bodenradarbild eine Flughafenkarte und verwendet eine synthetische Repräsentation zur Lagedarstellung und Identifikation von Flug- und Fahrzeugen.

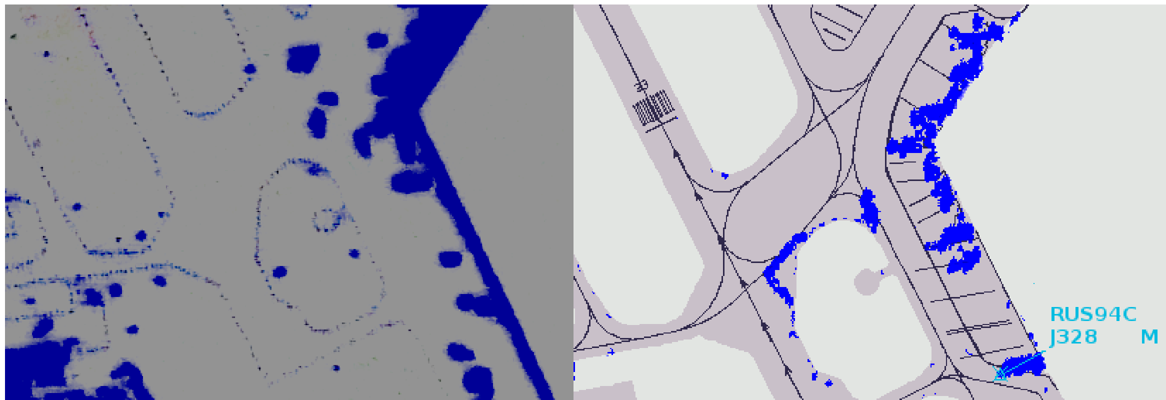


Abbildung 2.2: Vergleich der Darstellung und des Informationsgehaltes zwischen einem Radarsichtgerät (links) und einem A-SMGCS-Display (rechts)

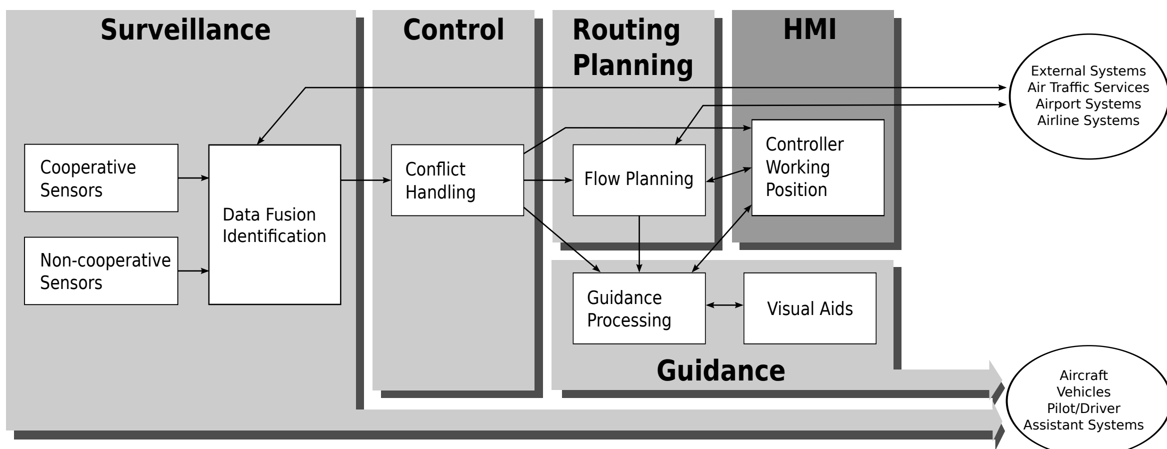


Abbildung 2.3: Funktionale Architektur des im Forschungsprojekt DEFAMM entwickelten A-SMGCS-Prototyps; modifiziert übernommen aus [23]

2.3 ACEMAX – Systemübersicht

ACEMAX, das von der Firma ADB Safegate Austria entwickelte und dieser Arbeit zugrundliegende A-SMGCS, basiert auf einer Client-Server-Architektur (vgl. Abbildung 2.4). Der ACEMAX-Server verfügt über Schnittstellen zu mehreren Sensoren und externen Systemen. Das ACEMAX-HMI (Client des Systems) implementiert die grafische Benutzerschnittstelle des Lotsen.

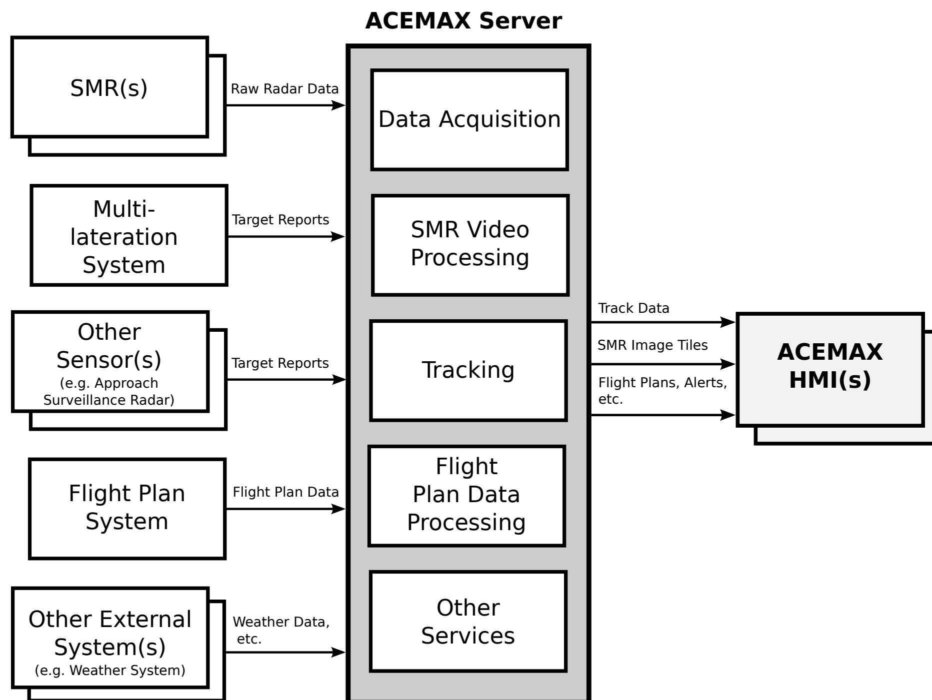


Abbildung 2.4: Client-Server-Architektur von ACEMAX

Die Kommunikation zwischen Server und Client erfolgt größtenteils über TCP/IP. Bei der Übertragung der Bodenradardaten an eine größere Anzahl an Clients kommt auch UDP-Mulitcast zum Einsatz. Auf der Anwendungsschicht (Schicht 7 des OSI-Modells) wird für den systeminternen Nachrichtenaustausch ein proprietäres Client-Server-Protokoll der Firma ADB Safegate Austria verwendet.

Der Server stellt dem Client verschiedene A-SMGCS-Daten zur Verfügung. Für die Bearbeitung der Aufgabenstellung ist es jedoch ausreichend, die Verarbeitung der Sensordaten zu betrachten.

2.3.1 Zielverfolgung

Die Positionsmeldungen der Sensoren werden vom ACEMAX-Server empfangen und verarbeitet. Die Kernaufgabe des Servers ist dabei die Zielverfolgung (Tracking). Der Tracker, also der für die Zielverfolgung zuständige Softwareprozess, führt die zeitlich aufeinanderfolgenden Beobachtungen der Sensoren zusammen und generiert daraus die Bewegungsspur eines Ziels [16, S. 154].

Der für ein detektiertes Ziel generierte Output des Systems wird als Track bezeichnet und beinhaltet folgende Hauptattribute:

- Track-ID: Die Kennung die zur eindeutigen Identifizierung des mit dem Ziel assoziierten Track-Objektes dient.

- Position: Die Position des Ziels wird einerseits in geographischen Koordinaten (geographische Breite und Länge) und andereseits in systemeigenen, kartesischen Koordinaten (durch Projektion der geographischen Koordinaten auf die Ebene) angegeben.
- Geschwindigkeitsvektor: Dieser gibt die Richtung und die Geschwindigkeit des Ziels an.
- Transponder-Daten: Die Daten die vom Transponder des Zieles übermittelt werden, wie z. B. die Kennung des Transponders, die Kennung des Fluges (Call-Sign) und die Flughöhe.

Die Track-Daten werden gesammelt (als Report) übertragen und periodisch, in einem konfigurierbaren Zeitintervall aktualisiert.

2.3.2 Bodenradardatenverarbeitung

Ein Radar besitzt eine definierte Winkel- und Entfernungsauflösung. Die Winkelauflösung entspricht dem Winkelinkrement der Radarantenne. Die Entfernungsauflösung gibt die minimale Distanz zwischen zwei Objekten an, die in ihrer Entfernung noch unterschieden werden können. Aus der Winkel- und Entfernungsauflösung ergibt sich ein polares Gitter. Innerhalb einer Gitterzelle, in weiterer Folge als Auflösungszelle bezeichnet, lässt sich die Position eines Objektes nicht näher bestimmen. (vgl. [16, S. 137 ff.], [35])

Die Itensitätswerte der detektierten Radarechos werden pro Azimutstellung als Datenvektors übertragen. Die Größe des Datenvektors entspricht der Anzahl der Auflösungszellen entlang des Radarstrahls und wird durch die zu erfassende Reichweite definiert. Jedes Vektorelement $v[i]$ ordnet dabei einer Auflösungzelle in Entfernung $\triangle R \cdot i$ einen Intensitätswert von 0 (kein Echo) bis I_{max} (maximale Intensität, e.g. 255) zu, wobei $\triangle R$ die Entfernungsauflösung des Radars angibt.

Die Azimutdaten werden im ersten Schritt der Verarbeitungskette gefiltert, um beispielsweise Clutter (unerwünschte Echos, z. B. bei Regen) und Rauschen zu unterdrücken. Nach erfolgter Filterung werden die aktualisierten Bereiche des in Polarkoordinaten vorliegenden Radarbildes vom Radarkoordinatensystem in das systemeigene, kartesische Koordinatensystem transformiert. Dieser Vorgang wird auch als Radar-Scan-Conversion bezeichnet [35].

Um neu abgetastete Bereiche effizient übertragen und zeitnah im ACEMAX-HMI aktualisieren zu können, wird das erzeugte Radarbild in rechteckige Kacheln (Tiles) unterteilt. Sobald ein Sektor vom Radar abgetastet wurde, werden alle dem Sektor zugehörigen Kacheln ausgesendet und die Anzeige aktualisiert. Die Sektorgröße ist einstellbar und kann somit mehr als ein Winkelinkrement des Radars umfassen.

Die prinzipielle Funktionsweise eines Bodenradars und die Verarbeitung der Radardaten durch den ACEMAX-Server sind in Abbildung 2.5 dargestellt. Der linke Teil der Abbildung zeigt die je Azimutstellung übermittelnden Radarintensitätswerte entlang des Radarstrahls. Die Verarbeitungsschritte die am ACEMAX-Server durchgeführt werden, um aus den Azimutdaten ein Radarbild zu erzeugen, sind in der Bildmitte zu sehen. Rechts daneben sind das genierte Radarbild und die für Übertragung durchgeführte Unterteilung des Bildes in Sektoren und Kacheln skizziert.

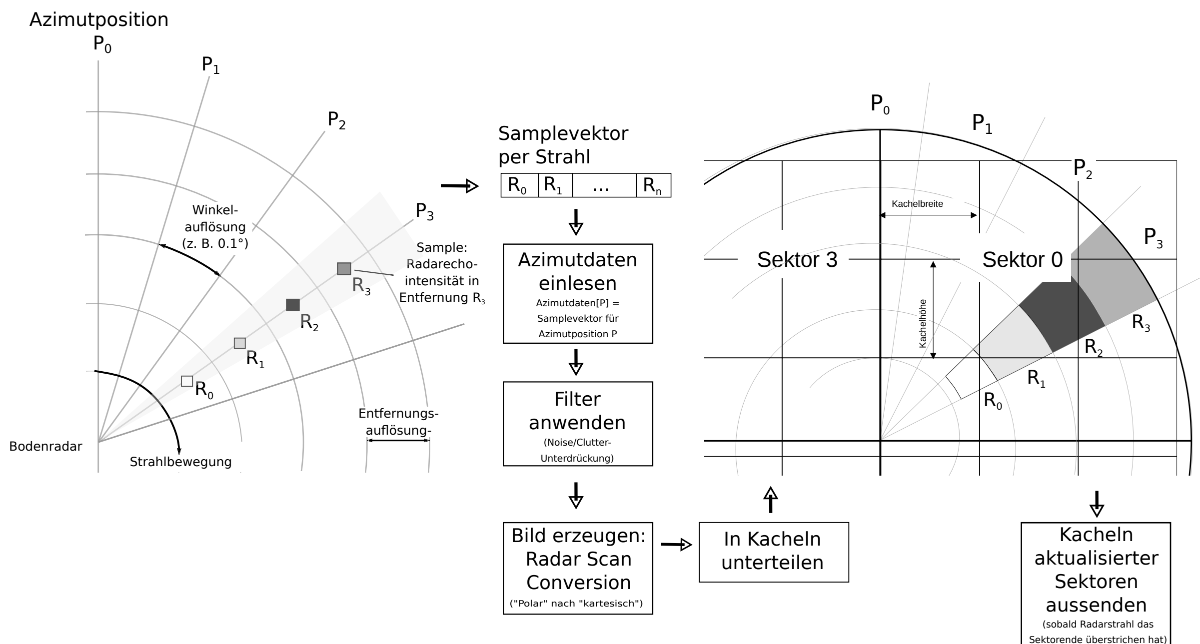


Abbildung 2.5: Vereinfachte Darstellung der Radardatenverarbeitung in ACEMAX

2.3.3 HMI – Anwendersicht

Das ACEMAX-HMI ist in Abbildung 2.6 ersichtlich. Das HMI besteht aus einem Hauptfenster in Vollbildmodus und mehreren Subfenstern. Die Bodenlage wird im Hauptfenster und in sogenannten Zoom-Fenstern (rechtes oberes und unteres Fenster) dargestellt, welche eine Detailansicht für einen vom Benutzer definierten Bereichs des Flughafens ermöglichen. Zusätzlich wird die Luftsituation im Nahverkehrsbereich des Flughafens (linkes Fenster) sowie die Zeit bis zur Landung (mittleres Fenster) von sich im Landeanflug befindlichen Flugzeugen visualisiert. Für jedes Fenster kann die Zoomstufe, der Blickwinkel (Rotation) und der sichtbare geografische Bereich mit der Maus verändert werden.

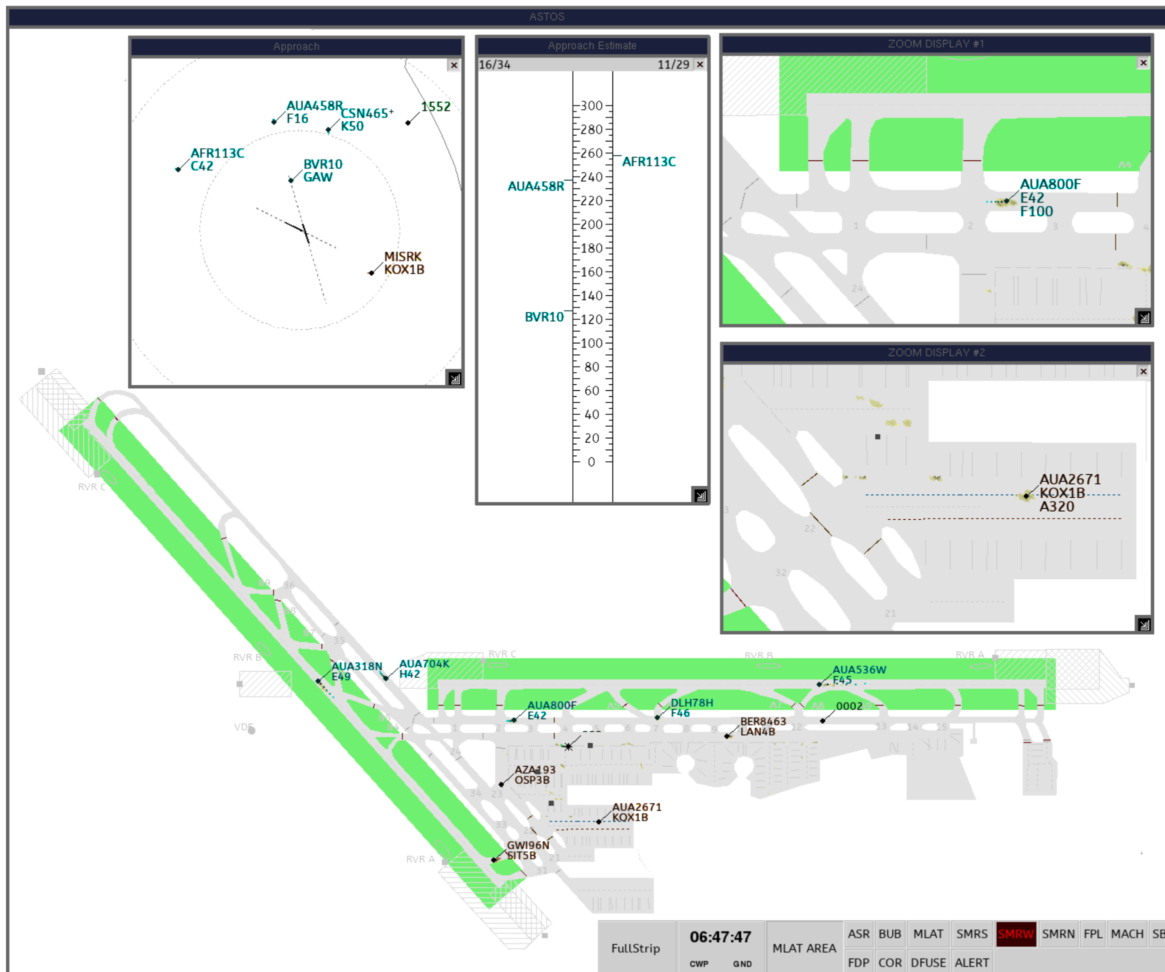


Abbildung 2.6: Boden- und Luftlagedarstellung im ACEMAX-HMI

Das Bodenlagebild setzt sich aus mehreren Bildebenen zusammen. Die unterste Ebene zeigt eine geografische Repräsentation des Flughafens (Flughafenkarte). In der darauffolgenden Ebene werden die Bodenradarechos dargestellt. Die oberste Ebene stellt die detektierte Position der Ziele grafisch dar.

2.3.3.1 Darstellung der Bodenradardaten

Die Visualisierung des Radarechos für ein vom Bodenradar erfasstes Ziel wird in Abbildung 2.7 gezeigt. Das aktuelle Echo des Ziels mit der Kennung "GWI4H" ist in orange dargestellt. Die Intensität des Radarechos entspricht dem Alphawert der korrespondierenden Pixel im Ausgabebild. Ein Radarecho mit maximaler Intensität verdeckt somit den Hintergrund vollständig, während Echos mit niedriger Intensität fast komplett transparent erscheinen.

Das Radarbild enthält zusätzlich auch ein sogenanntes synthetisches Afterglow [36, S. 117]. Der Begriff Afterglow entstammt der Verwendung von analogen Radarsichtgeräten mit

Phosphor-Leuchtschirmen [32, S. 343]. Die Nachleuchtdauer dieser Bildschirme führt dazu, dass illuminierte Pixel (Radarechos) erst nach mehreren Radarumläufen vollständig verblassen. Ziele lassen sich dadurch leichter verfolgen.

Um diesen Effekt synthetisch herzustellen, werden die Echos vorhergehender Radarumläufe nicht gelöscht, sondern beim nächsten Radarumlauf, beim Erreichen der gleichen Azimutstellung oder in fixen Zeitabständen in ihrer Intensität reduziert, was optisch zu einem verblassen des Echos führt. Die Verwendung einer eigenen Afterglow-Farbe verbessert noch zusätzlich die Unterscheidung zwischen dem aktuellen Echo und dem synthetischen Afterglow.

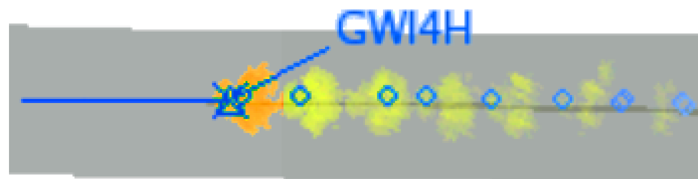


Abbildung 2.7: Radarecho (orange) mit synthetischem Afterglow (gelb)

2.3.3.2 Darstellung der Ziele

Die synthetische Zielrepräsentation ist in Abbildung 2.8 ersichtlich. Die Zielposition wird durch ein Symbol dargestellt, das in weitere Folge als Track-Symbol bezeichnet wird. Die Geschwindigkeit und Richtung des Ziels wird durch einen Geschwindigkeitsvektor (Speed-Vector) angezeigt. Die Bewegungsverfolgung wird durch die Darstellung der letzten N Zielpositionen, der sogenannten Track-History, erleichtert. Punkte der Positionshistorie werden dabei durch ein eigenes Symbol präsentiert, und erscheinen – analog zur Darstellung des synthetischen Afterglow – entsprechend ihres "Alters" abgedunkelt bzw. transparent.

Das Track-Symbol ist über eine Verbindungslinie, die Leader-Line, mit einem Label verbunden. Das Label kann neben der Kennung des Ziels auch zusätzliche Informationen enthalten, wie beispielsweise die Flughöhe und die Kennung des Transponders. Durch die Verknüpfung des Ziels mit einem Flugplan können auch wichtige Informationen aus dem Flugplan angezeigt werden.

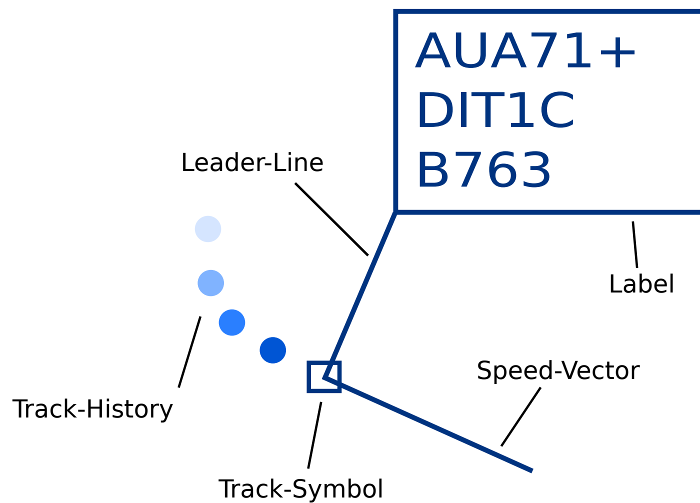


Abbildung 2.8: Zieldarstellung

2.3.4 HMI – Techniksicht

Die Implementierung des HMI basiert auf Qt 3 (siehe auch Kapitel 2.4.1). Für die grafische Anzeige wird die von Qt bereitgestellte Abstraktion der plattformspezifischen 2D-Grafikschnittstelle verwendet. Auf Qt aufbauend wurde von ADB Safegate Austria die Grafikbibliothek AVLayerView entwickelt, mit der sich die einzelnen Darstellungslayer implementieren und zu einem Bild zusammenführen lassen. Der gesamte Grafik-Stack ist in Abbildung 2.9 ersichtlich.

2.3.4.1 Grafikbibliothek AVLayerView

Das Widget `AVLayerView` und die Klasse `AVLayer` sind die zentralen Elemente der Grafikbibliothek. Zum Zeichnen des Layerinhaltes, implementierten Layer eine abstrakte Draw-Methode der Basisklasse `AVLayer`. Der Draw-Methode wird als Argument eine Instanz der Qt-Klasse `QPainter` übergeben, welche ein umfangreiches API zum Zeichnen von Vektor- und Rastergrafiken anbietet. Die Layer werden entsprechend ihrer Darstellungsreihenfolge zum Widget `AVLayerView` hinzugefügt.

Die Bildsynthese findet in einem Offscreen-Buffer (Qt-Klasse `QPixmap`) statt. Dabei werden die Draw-Methoden der nach Tiefe sortierten Layer sequentiell aufgerufen (*Painter's algorithm* [20, S. 1041]). Am Ende des Rendervorganges wird der Inhalt des Widgets mit dem Inhalt des Offscreen-Buffers überschrieben. Die Aktualisierung des von `AVLayerView` dargestellten Inhalts, in weiterer Folge als Update bezeichnet, ist ereignisgetrieben. Jeder Layer kann ein Update anfordern (Update-Request). Zeitlich dicht aufeinanderfolgende Update-Requests werden zusammengefasst und lösen daher nicht mehrere, sondern nur ein Update aus. Das Zeitfenster in dem Updates-Requests zusammengefasst werden beträgt standardmäßig 150 [ms] und definiert sogleich die maximal erzielbare Update-Rate.

Das Konzept anhand der in dieser Arbeit betrachteten Layer wird in Abbildung 2.10 dargestellt.

Die Arbeitsweise des Map-, SMR- und Track-Layers wird in weiterer Folge noch genauer beleuchtet.

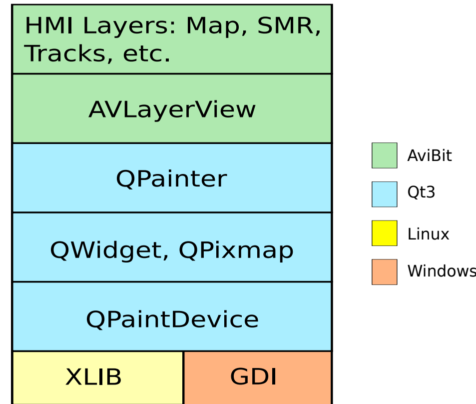


Abbildung 2.9: Grafik-Stack des ACEMAX-HMI

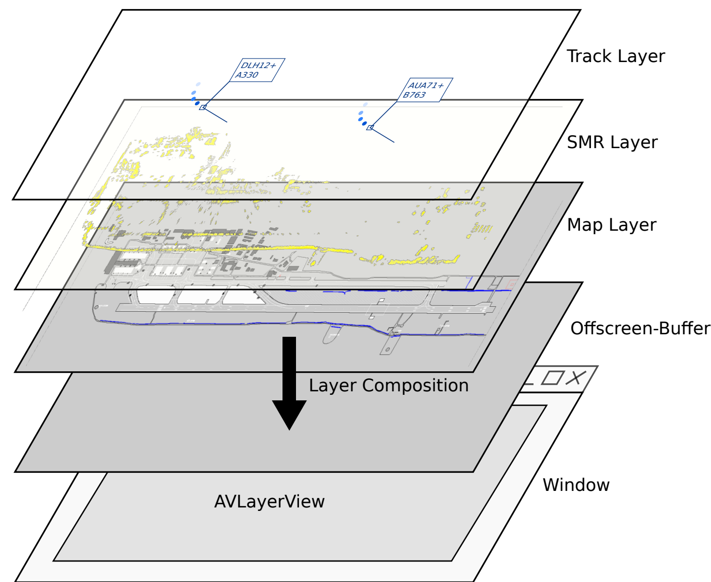


Abbildung 2.10: Komposition des Bodenlagebildes

2.3.4.2 Rendern der Flughafenkarte

Die Flughafengeländedaten werden in einem proprietären XML-basierten Vektorformat der Firma ADB Safegate Austria gespeichert. Die Koordinaten liegen im systemeigenen, zweidimensionalen, kartesischen Koordinatensystem vor. Das Koordinatensystem wird durch eine stereografische Projektion definiert, die geografische Koordinaten auf Basis des *World Geodetic System 1984* (WGS84) auf die Ebene projiziert (siehe auch [11]).

Das dem Vektorformat zugrunde liegende Objektmodell ist hierarchisch aufgebaut und besteht auf der obersten Ebene aus einer Liste von Gruppen. Jede Gruppe besteht wiederum aus einer Liste von Zeichenelementen. Es existieren Zeichenelemente für die folgenden geometrischen Primitive:

- Punkt
- Rechteck
- Polylinie
- Polygon
- Ellipse
- Textelemente
- Positionssymbole

Für jedes Element lassen sich visuelle Eigenschaften, wie Farbe, Linienstärke und Füllmuster definieren. Die Zeichenreihenfolge ist durch die Listenreihenfolge der Gruppen und der Listenreihenfolge der Elemente innerhalb einer Gruppe festgelegt.

Die Karte wird vom Map-Layer unter Berücksichtigung der Ansichtstransformation gezeichnet. Das Ergebnis wird in einem Bildpuffer zwischengespeichert, um bei einer Änderung in einem darüber liegenden Layer und bei gleichbleibender Ansichtstransformation die Karte nicht neu rendern zu müssen.

2.3.4.3 Rendern der Bodenradardaten

Wie in Kapitel 2.3.2 bereits erläutert, werden die Radardaten bereits am Server gerastert gerastert und in Kacheln unterteilt. Aktualisierte Bildbereiche werden über das ADB Safegate Austria Messaging-Protokoll an das HMI übertragen. Die weitere Verarbeitung im HMI ist in Abbildung 2.11 dargestellt.

Die vom Server empfangenen Bildkacheln werden vom Softwaremodul SMR-Image-Composer wieder zu einem vollständigen Bild zusammengesetzt. Die gerasterten Radardaten können dabei von mehreren Radarquellen stammen. Die Komposition des Radarbildes aus den Radardaten mehrerer Radarquellen erfolgt bei sich überlappenden Bereichen nach einer vorgegebenen Reihenfolge.

Der SMR-Image-Composer stellt dem SMR-Layer das erzeugte Bild und eine Liste von Rechtecken bereit, welche die aktualisierten Bildbereiche beschreiben. Die Teilbilder werden vom SMR-Layer mittels Bildtransformation in das View-Koordinatensystem, das sich gegenüber dem Systemkoordinatensystem um die vom Benutzer festgelegte Ansichtstransformation unterscheidet, übergeführt. Der Abbildungsvorgang des Teilbildes wird in inverser Richtung (Inverse-Mapping [17, S. 88]) ausgeführt. Das heißt, es wird zuerst der der sichtbare Bildbereich im View-Koordinatensystem berechnet, danach werden die korrespondierenden Bildpunkte im Ausgangsbild mittels Rückwärtstransformation bestimmt. Bei Verkleinerung des Teilbildes wird die Nearest-Neighbour-Interpolation verwendet. Zur Minimierung des Rechenaufwandes wird die Rückwärtstransformation für alle Bildpunkte des Fensters vorausberechnet und in einer Lookup-Tabelle gespeichert.

Die Teilbilder werden schließlich in einem der Grafikschnittstelle zugänglichen Bildpuffers

(Pixmap) zusammengeführt und gezeichnet. Unter Linux residieren Pixmaps im Speicherbereich des X-Server. Es ist daher im Regelfall kein direkter Speicherzugriff möglich. Um Bilddaten effizienter übertragen zu können, wird die MIT-Shared-Memory-Extension [33] des X-Servers verwendet. Der Pixmap-Speicher kann damit in den Adressraum des HMI eingeblendet werden.

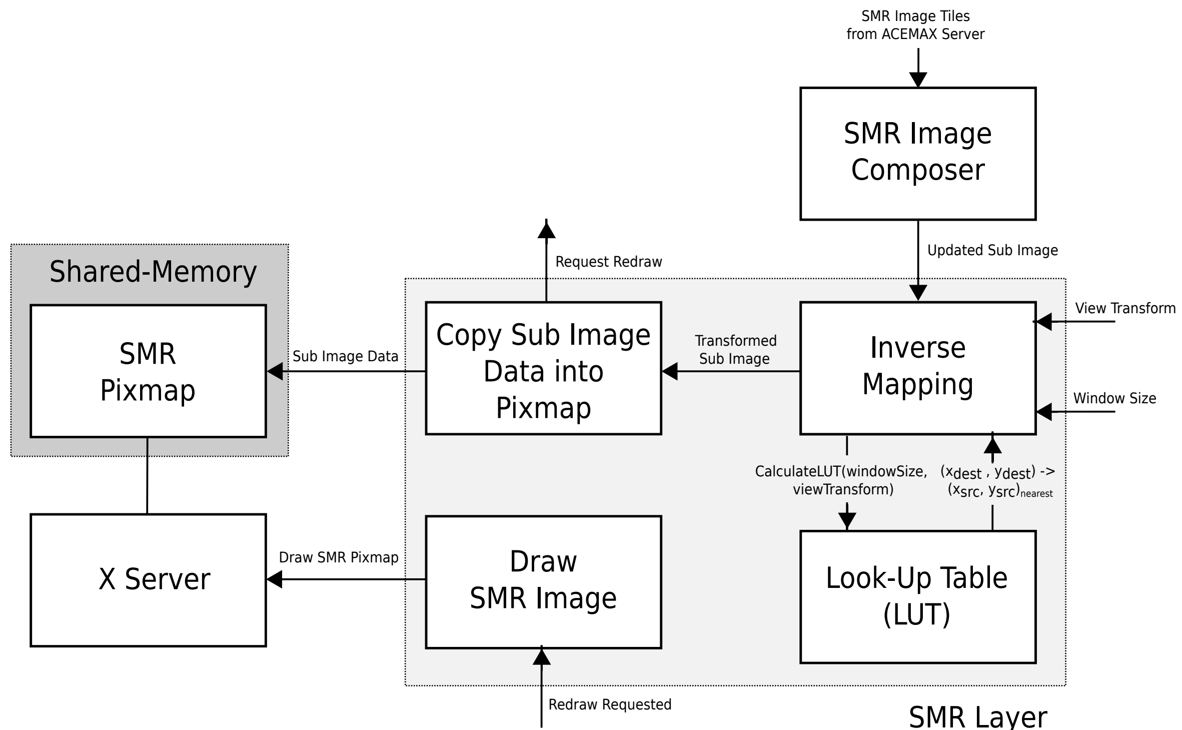


Abbildung 2.11: Verarbeitungsschritte zum Rendern des SMR-Bildes im ACEMAX-HMI

2.3.4.4 Rendern der Ziele

Die vom ACEMAX-Server in regelmäßigen Intervallen ausgesandten Track-Daten werden vom HMI empfangen und als Liste von Track-Objekten vorgehalten. Zwischen dem Empfang der Track-Daten und dem eigentlichen Rendervorgang passiert jedes Track-Objekt eine Filterkette (Architekturmuster “Pipes and Filters”, siehe auch [4, S. 53]), um beispielsweise ein Ziel mit einem Flugplan zu assoziieren oder um es gänzlich von der Darstellung auszunehmen.

Die Track-Objekte werden, nachdem sie die Filterkette durchlaufen haben, an den Track-Layer weitergereicht und gerendert. Eine prozedurale Beschreibung des Rendervorgangs ist in Abbildung 2.12 enthalten. Wie der Abbildung zu entnehmen ist, werden Tracks der Reihe nach gezeichnet. Hierbei bietet die Zeichenreihenfolge für den Betrachter keinen zusätzlichen Informationsgehalt (Darstellung von Track “A” über Track “B” hat keine Bedeutung) und beruht auf einem nicht ersichtlichen Sortierkriterium (Sortierung nach einer automatisch generierten, system-internen Kennung).

Der Track-Layer speichert das Render-Ergebnis nicht in einem Bildpuffer zwischen, dass

heißt, der Rendervorgang muss bei einer Änderungen eines anderen Layers wiederholt werden. Der Takt, in dem der Track-Layer neu gerendert wird, wird somit durch den Layer mit der höchsten Aktualisierungsrate, im Regelfall durch den SMR-Layer, vorgegeben.

```
1: modelViewMatrix ← view.modelViewMatrix
2: viewportRect ← view.rect
3: for all track in trackList do
4:     position ← modelViewMatrix · track.position // Transform current position
5:     // Draw only tracks that are inside the visible area and are not filtered
6:     if contains(viewportRect, position) = false or track.isFiltered then
7:         next
8:     end if
9:     for i = 0 to sizeof(track.history) do
10:        position ← modelViewMatrix · track.history[i] // Transform history point
11:        dimFactor ← i/(sizeof(track.history) + 1)
12:        // Draw the track history symbol. Symbol color
13:        // depends normally on the track type (e.g. aircraft vs. vehicle)
14:        drawHistorySymbol(track.trackType, position, dimFactor)
15:    end for
16:    // Draw track symbol. Symbol type and color normally depend on the track type
17:    drawTrackSymbol(track.trackType, position)
18:    speedVector ← modelViewMatrix · track.speedVector // Transform the speed
   vector
19:    // Draw speed vector line. Color normally depends on the track type
20:    drawSpeedVector(track.trackType, position, speedVector)
21:    // Draw leader line. Color depends normally on the track type
22:    drawLeaderLine(track.trackType, position)
23:    // Draw label. Label layout and color scheme depend normally on the track type
24:    drawLabel(track.trackType, position)
25: end for
```

Abbildung 2.12: Prinzipielle Render-Prozedur zur Zieldarstellung im ACEMAX-HMI

2.4 Anforderungen

In diesem Abschnitt wird die Problemstellung durch eine Kurzbeschreibung der bei ADB Safegate Austria vorherrschenden technischen Rahmenbedingungen sowie durch Auflistung der wichtigsten funktionalen und nicht-funktionalen Anforderungen, die an den HMI-Prototyp gestellt werden, noch weiter präzisiert. Zur Referenzierung im Verlauf dieser Arbeit werden die Anforderungen durchnummeriert (Anforderung R1 bis Rn).

2.4.1 Rahmenbedingungen

Software Als Programmiersprache wird C++ nach ISO/IEC 14882:2011 [5] verwendet. Das plattformübergreifende Application-Framework Qt [7] bildet das Fundament der entwickelten

Softwarelösungen. Ein Großteil der ADB Safegate Austria Software wurde auf Basis von Qt 3, Version 3 des Qt-Frameworks, entwickelt. Die Weiterentwicklung des Frameworks, das mittlerweilen bereits in Version 5 vorliegt, hat dazu geführt, dass die Codebasis sich in einem Migrationszustand befindet. Softwaresysteme, wie ACEMAX, werden schrittweise auf die neueste Qt-Version (Qt 5) portiert.

Betriebssystem Als Betriebssystem wird die Linux-Distribution CentOS 7 64-Bit [6] eingesetzt.

Hardware Die Hardware wird im Regelfall von ADB Safegate Austria geliefert. Die Firma ist dadurch in der Lage auf die neuesten Modelle zurückzugreifen. Für die Server und Client-Komponenten des Systems wird im Regelfall Server-Hardware (e.g. HP Proliant Servers) verwendet, in Ausnahmefällen kommen für Clients auch Desktop-PCs zum Einsatz. Die Server und Clients sind über Gigabit-Ethernet miteinander verbunden.

2.4.2 Funktionale Anforderungen

Anforderung R1 Der HMI-Prototyp soll aus einem Hauptfenster und mindestens 5 Zoom-Fenstern bestehen.

Anforderung R2 Das Bodenlagebild pro Fenster soll folgende Bildebenen umfassen:

- Ebene 0: Geografische Repräsentation des Flughafens (Flughafenkarte)
- Ebene 1: Darstellung der Bodenradardaten
- Ebene 2: Darstellung der Bodenziele

Anforderung R3 Der sichtbare Bereich soll, analog zum bisherigen HMI (siehe 2.3.3), mit der Maus einstellbar sein und die Ansichtstransformationen "Verschieben", "Zoom" und "Rotation" unterstützen.

Anforderung R4 Der HMI-Prototyp soll Echtdaten, aufgezeichnete und simulierte Daten verarbeiten können.

Anforderung R5 Die Zieldarstellung soll analog zum bisherigen HMI (siehe 2.3.3.2) erfolgen und soll sich daher aus den Komponenten Track-Symbol, Track-History, Speed-Vector, Leader-Line und Label zusammensetzen.

2.4.3 Nicht-funktionale Anforderungen

Anforderung R6 Der HMI-Prototyp soll ein Bodenradarbild für einen Erfassungsbereich von maximal 4500 x 4500 [m] und einer Entfernungsauflösung von 1.8 [m] darstellen können, das in 8 Sektoren unterteilt ist und das sektorweise, mit einer Radarumlaufzeit von 1 Sekunde, aktualisiert wird.

Anforderung R7 Der HMI-Prototyp soll im Ein-Sekunden-Intervall versendete Track-Updates verarbeiten können.

Anforderung R8 Es sollen mindestens 500 Tracks pro Fenster dargestellt werden können.

Anforderung R9 Für den Betrieb des HMI-Prototypen soll als Referenzsystem ein Desktop-PC mit Quad-Core-Prozessor (Intel i5-6500 CPU 3.20 GHz) und 32 GB RAM dienen. Der PC verfügt über eine NVIDIA Grafikkarte (NVIDIA Quadro P400), welche über PCIe 3.0 x16 an das System angebunden ist. Der HMI-Prototyp soll auf CentOS 7 (64 Bit) unter dem Fenstermanager IceWM [21] an 2 Bildschirmen mit einer Auflösung von 1920 x 1200 betrieben werden.

Anforderung R10 Alle für den HMI-Prototyp erforderlichen Basisbibliotheken sollen von Qt 3/4 nach Qt 5 portiert werden.

Anforderung R11 Der HMI-Prototyp soll auf Basis von Qt Quick entwickelt werden.

Anforderung R12 Die Performance des HMI-Prototypen soll auf dem in Anforderung R9 spezifizierten Referenzsystem unter den nachfolgenden Bedingungen getestet werden.

Flughafenkarte Für die Darstellung des Flughafens soll eine fiktive, von der Firma ADB Safegate Austria zur Verfügung gestellte Flughafenkarte verwendet werden.

Track-Daten Der ACEMAX-Server soll als Track-Datenquellen für den HMI-Prototypen fungieren. Die zur Track-Generierung notwendigen Sensor-Input-Daten sollen artifiziell erzeugt werden. Die Simulation soll 500 Tracks umfassen, welche sich in gleichförmiger geradliniger Bewegung befinden und räumlich im Umkreis von 2000 [m] um den Flughafenbezugspunkt (Ursprung des kartesischen Koordinatensystems) verteilt sind. Die erzeugten Tracks sollen über den gesamten Testverlauf hinweg erhalten bleiben. Die Track-Updates sollen im Ein-Sekunden-Intervall vom ACEMAX-Server ausgesendet werden.

SMR-Video Das SMR-Video soll, analog zu den Track-Daten, vom ACEMAX-Server empfangen werden. Das mittels artifizieller Input-Daten erzeugte SMR-Video soll einen Bereich im Radius von 2250 [m] um die Radarposition abdecken und eine Entfernungsauflösung von 1.8 [m] aufweisen. Die Radarposition soll sich zentral am Flughafen befinden. Die Umlaufzeit des fiktiven Radars soll 1 [s] betragen. Das SMR-Video soll vom ACEMAX-Server in 8 Sektoren aufgeteilt und sektorweise aktualisiert werden. Die Bildkacheln sollen eine Größe von 100 x 100 [Pixel] besitzen.

HMI-Konfiguration Das Hauptfenster soll maximiert auf dem ersten Bildschirm angezeigt werden. Die 5 Zoom-Fenster sollen mit einer Fenstergröße von 960 x 500 [Pixel] am zweiten Bildschirm angeordnet werden. In jedem Fenster soll der Flughafenbereich innerhalb von 2 [NM] um den Flughafenbezugspunkt sichtbar sein. Die Ziele sollen mit Speed-Vector und Track-History – die letzten fünf vorhergehenden Positionen – dargestellt werden. Das Label soll das Call-Sign, den Mode-S-Code und die Geschwindigkeit des Zieles zeigen.

2.5 Analyse des bestehenden HMI

Empirisch kann bei der Anzeige von 500 Tracks eine totale Überlastung des ACEMAX-HMI festgestellt werden. Die Prozessorauslastung stellt dabei den limitierenden Faktor dar.

Da mit der Neuentwicklung des HMI auch eine Performance-Verbesserung erzielt werden soll (siehe Anforderung R8 in 2.4), ist eine Analyse der derzeitigen Implementierung notwendig, um einerseits die Leistungsfähigkeit der neuen und alten Implementierung unter gleichen Testbedingungen vergleichbar zu machen und andererseits jene Verarbeitungsschritte zu identifizieren, die die meiste Prozessorzeit in Anspruch nehmen.

2.5.1 Messung der CPU-Last

Zur Analyse wird die vom HMI verursachte CPU-Last auf dem in Anforderung R9 (siehe 2.4) spezifizierten Referenzsystem mit der in Anforderung R12 angegebenen Testbedingungen gemessen. Für die Anzeige des SMR-Videos im HMI werden, wie in Abbildung 2.13 gezeigt, Radarechos simuliert, die sich in radialer Richtung in fixen Abständen wiederholen (konzentrische Kreise). Die Track-Anzeige und die Fensteranordnung auf den zwei Bildschirmen, mit einer Auflösung von jeweils 1920 x 1200, sind in Abbildung 2.14 ersichtlich. Um den Anteil der Track- bzw. SMR-Darstellung an der beanspruchten CPU-Zeit sichtbar zu machen, werden die untenstehenden Testkonfigurationen verwendet. Die Messung der CPU-Last einer Testkonfiguration wird für unterschiedliche Track-Anzahlen durchgeführt: *a*) keine, *b*) 125, *c*) 250 und *d*) 500 Tracks.

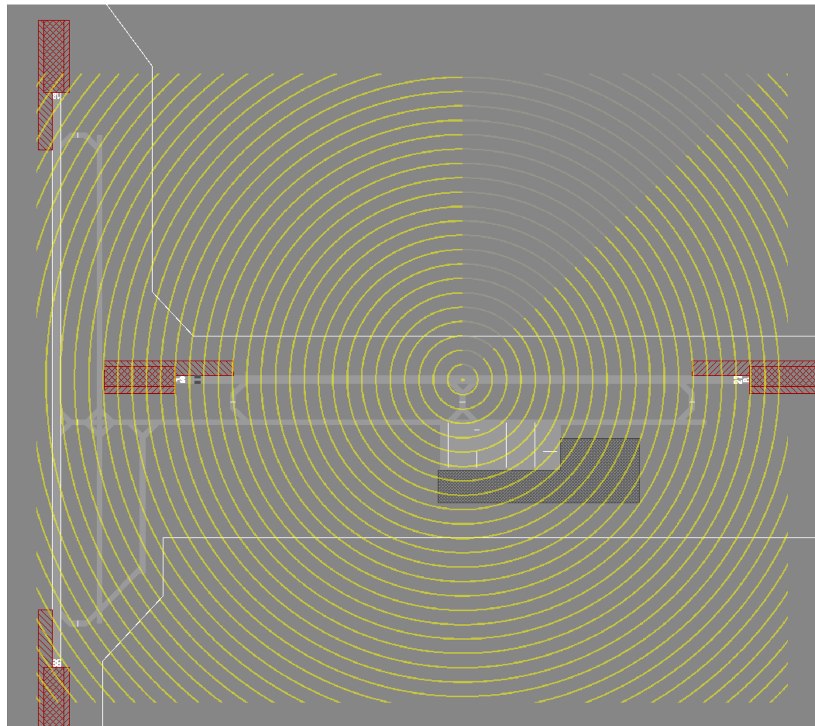


Abbildung 2.13: Synthetisch generiertes SMR-Video

Testkonfiguration "Tracks + SMR" In dieser Testkonfiguration wird das ACEMAX-HMI in Volllast betrieben. Es werden sowohl Track- als auch SMR-Daten angezeigt.

Testkonfiguration "Track-Datenverarbeitung" Diese Testkonfiguration dient zur Abschätzung des Anteils der Track-Datenverarbeitung (e.g. Verarbeitung der Server-Daten, Durchlauf der Filterkette) an der gesamten CPU-Last. Es werden keine SMR-Daten verarbeitet, der Layer zum Zeichnen der Track-Daten ist deaktiviert.

Testkonfiguration "Tracks" Zur Abschätzung des Anteils der Track-Anzeige an der gesamten CPU-Last werden in dieser Testkonfiguration nur Track-Daten angezeigt. Es werden keine SMR-Daten verarbeitet.

Testkonfiguration "SMR" In dieser Testkonfiguration wird zur Abschätzung des Anteils der SMR-Anzeige an der gesamten CPU-Last nur das SMR-Video angezeigt. Es werden keine Track-Daten verarbeitet. Zusätzlich wir die durch den SMR-Image-Composer (e.g. Komposition der Teilbilder, Erzeugung des synthetischen Afterglow) verursachte CPU-Last gemessen. Diese ist getrennt messbar, da der SMR-Image-Composer in einem eigenenen Thread betrieben wird.

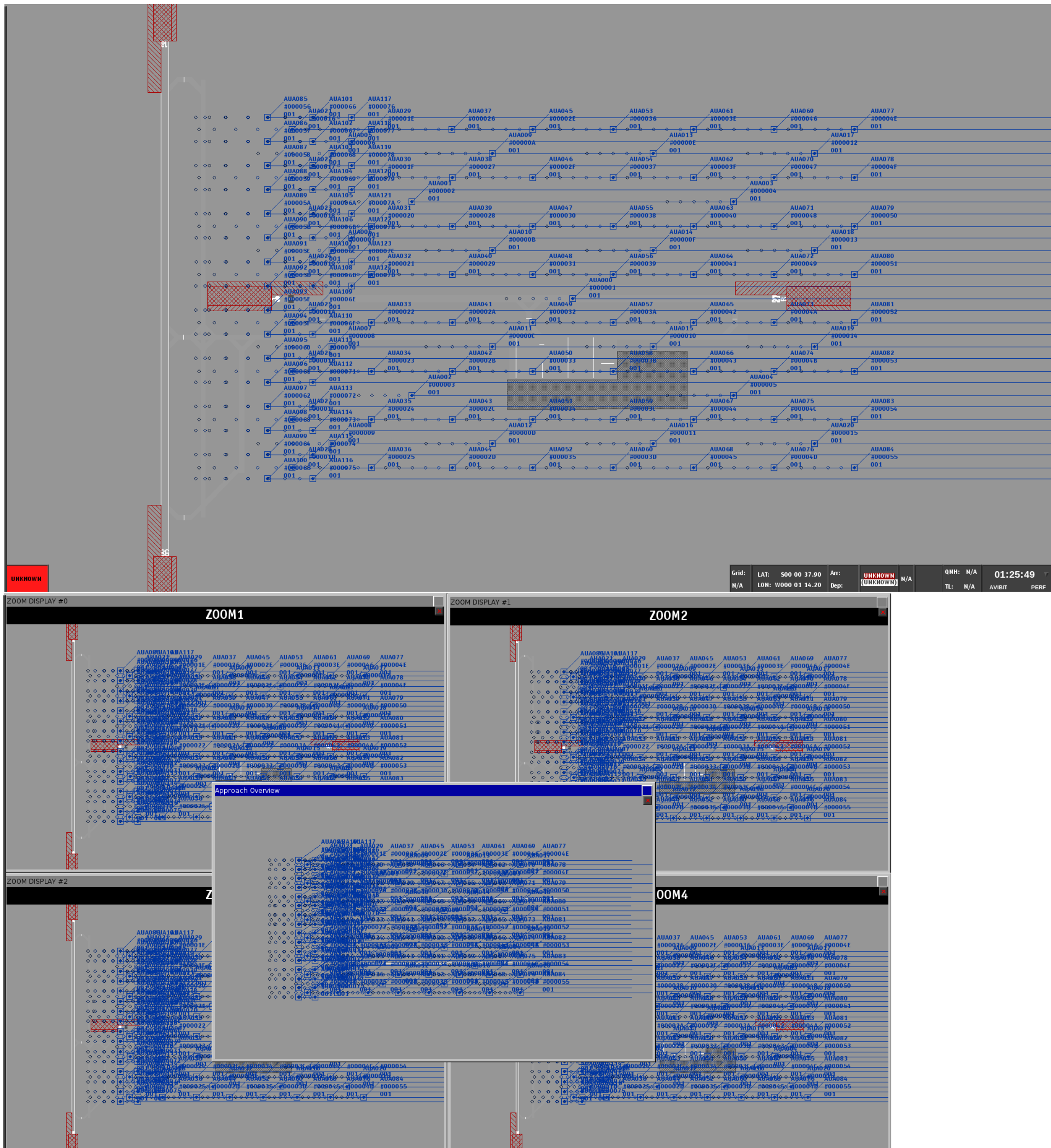


Abbildung 2.14: Fensteranordnung und Anzeigeeinstellungen des ACEMAX-HMI bei der Durchführung der Performance-Tests

2.5.2 Resultate und Schlussfolgerungen

Das Messergebnis ist in Abbildung 2.15 ersichtlich, die zugrundeliegenden Messdaten sind im Anhang A.1 enthalten. Das Balkendiagramm zeigt für jede Testkonfiguration die durchschnittliche CPU-Last des ganzen HMI sowie des Submoduls SMR-Image-Composer. Neben der

CPU-Last des HMI wird auch die durchschnittliche CPU-Last des X-Servers angegeben.

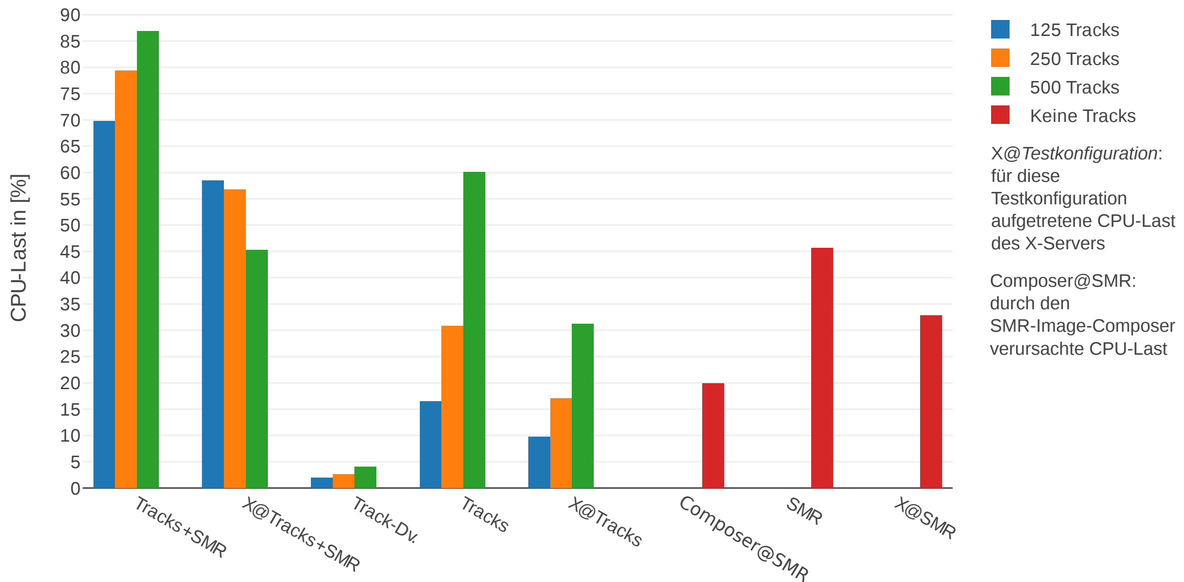


Abbildung 2.15: Durchschnittliche CPU-Last des ACEMAX-HMI (Wertebereich: 0 - 400 %) bei keinen, 125, 250 und 500 Tracks

Aus den Ergebnissen lässt sich erkennen, dass die CPU-Last in der Testkonfiguration "Tracks+SMR" bei zunehmender Track-Anzahl annähernd linear ansteigt. Die CPU-Last des X-Servers bricht bei 500 Tracks ein. Die Track-Anzeige für sich alleine betrachtet zeigt jedoch einen deutlichen Anstieg der CPU-Last des X-Servers. Daraus kann geschlossen werden, dass das kombinierte Bodenlagebild (Tracks u. SMR) aufgrund der höheren Bearbeitungszeit weniger oft aktualisiert wird.

Die Verdoppelung der Track-Anzahl in der Testkonfiguration "Tracks" führt beinahe zur Verdoppelung der CPU-Last. Die Track-Datenverarbeitung generiert vergleichsweise wenig CPU-Last. Daraus kann abgeleitet werden, dass der Großteil der CPU-Zeit beim Zeichnen der Track-Daten im Track-Layer verbraucht wird.

Der Vergleich der Messergebnisse "Composer@SMR" und "SMR" zeigt, dass die CPU-Last für den SMR-Image-Composer-Thread weniger als die Hälfte der gesamten CPU-Last der SMR-Anzeige beträgt. Analog zur Track-Anzeige wird ein Großteil der CPU-Zeit also beim Darstellen der SMR-Daten im SMR-Layer verbraucht. Die SMR-Anzeige trägt auch maßgeblich zur Auslastung des X-Servers bei.

Der Track-Layer und der SMR-Layer werden pro Fenster instanziiert. Die Verarbeitungschritte pro Fenster erfolgen jedoch sequentiell, da für das Zeichnen nur ein Thread in Verwendung ist (Multi-Threaded-Rendering mit `QPainter` ist in Qt 3 nicht möglich, siehe [47]).

3 Methoden und Technologien

In diesem Kapitel werden zunächst die mit dem GUI-Toolkit von Qt [7] verfügbaren Rendering-Möglichkeiten vorgestellt. Der Fokus liegt dabei auf Qt Quick, das gleichzeitig auch Teil der zu bearbeitenden Problemstellung ist. Danach werden Methoden zum effizienten Texturdatentransfer mit OpenGL erläutert, welche für die Darstellung des Bodenradarbildes im HMI-Prototypen von Bedeutung sind. Abschließend sollen relevante Arbeiten zur Realisierung eines Radar Display unter Verwendung moderner Grafikhardware zusammengefasst werden.

3.1 Qt 5

Das Qt Software Development Kit (SDK) dient der plattformübergreifenden Entwicklung von Anwendungen. Qt wurde 1995 erstmals veröffentlicht und war von seinen Entwicklern Haavard Nord und Eirik Chambe-Eng primär als C++-Toolkit für die objektorientierte Erstellung von Anwendungen mit grafischer Benutzeroberfläche (GUI) gedacht [3, S. XV ff.]. Seit seinen Anfängen als reines GUI-Toolkit wurde das Qt SDK fortlaufend um Funktionalität erweitert. Neben den Programmbibliotheken stellt Qt auch Werkzeuge für die Implementierung von Anwendungen zur Verfügung. Mit dem Qt Creator, der integrierten Entwicklungsumgebung (IDE) von Qt, lassen sich beispielsweise GUIs entwickeln und grafisch gestalten.

Qt 5, die aktuellste Version des SDK, bietet im Vergleich zu Qt 3, auf Basis dessen das ACEMAX-HMI der Firma ADB Safegate Austria entwickelt wurde, einige Neuerungen. Neben Erweiterungen des konventionellen Qt Widget-Systems steht mit Qt Quick auch eine neue GUI-Technologie zur Verfügung. Mit dem Qt Quick- und Qt 3D-Rendering-System setzt das GUI-Toolkit von Qt zudem auf hardwarebeschleunigtes Rendern von 2D und 3D Inhalten.

3.1.1 Konventionelles GUI-System

Der konventionelle Ansatz von Qt zur Erstellung einer GUI basiert auf die Verwendung von Widgets. Unter Widgets, Kunstwort aus den Wörtern “Window” und “Gadgets”, werden visuelle Elemente verstanden aus welchen sich Fenster der GUI [3, S. XV ff.] zusammensetzen. Qt bietet für die Standardkomponenten einer klassischen Desktop-Anwendung, wie z. B. Buttons und Texteingabefelder, vorgefertigte Widgets an. Widgets lassen sich verschachteln und es können dadurch auch neue, komplexere Widgets erstellt werden.

`QWidget` ist die zentrale Klasse des Qt Widget-Systems und stellt die Basisklasse für alle GUI-Komponenten dar. Vom Window-System erzeugte Events (e.g. Maus- und Keyboard-Events) werden über das Event-System von Qt an das betroffene Widget zur Verarbeitung weitergeleitet.

Ein Fenster mit verschachtelten UI-Komponenten wird als Baum von `QWidget` Objekten

repräsentiert. Das Fenster ist selbst ein `QWidget` und stellt die Wurzel des Baumes dar. Die Parent-Child-Beziehung zwischen den `QWidget` Objekten wird mittels Pointer auf das Parent-Widget hergestellt.

Die automatische Ausrichtung und Positionierung von Widgets wird über Layout-Klassen, wie `QGridLayout`, ermöglicht. Die Benutzeroberfläche lässt sich vollständig in C++ durch Instanziierung der erforderlichen Widget- und Layout-Klassen erstellen. Für komplexere Benutzeroberflächen bietet sich jedoch die grafische Erstellung mittels Qt Creator/Designer an. Das so erzeugte Design wird in einem Qt-spezifischen XML-Format gespeichert. Die Generierung des C++ Codes erfolgt mittels des Qt User-Interface-Compilers (UIC).

Die Klasse `QPainter` ist das Herzstück des Qt Paint-Systems. Es werden die wesentlichen Zeichenprimitive sowie komplexe Formen (`QPainterPath`) unterstützt, die in unterschiedlichen Linienstärken, Linienstilen und Füllmustern dargestellt werden können.

Gezeichnet wird auf einem `QPaintDevice`, welches eine abstrakte Zeichenfläche zum Zeichnen von 2D Inhalten repräsentiert. `QWidget` ist selbst ein `QPaintDevice` und es lässt sich daher direkt mit `QPainter` auf ein `QWidget` zeichnen. Mit QPixmap und `QImage` stehen `QPaintDevice` Implementierungen zur Verfügung, um in einen Offscreen-Buffer zu zeichnen.

Der eigentliche Rendervorgang (Bildsynthese) wird durch die vom `QPaintDevice` zur Verfügung gestellte `QPaintEngine` Implementierung durchgeführt. Die Standardimplementierung, welche z. B. zum Zeichnen von Widgets auf allen Plattformen zum Einsatz kommt, verwendet den Software-Rasterizer von Qt. Für hardwarebeschleunigtes Rendern steht durch Verwendung von `QOpenGLPaintDevice` auch eine Implementierung für OpenGL zur Verfügung.

3.1.2 Qt Quick

Mit Qt Quick steht dem konventionellen Qt Widget-System ein neuer Ansatz entgegen, um grafische Benutzeroberflächen zu erstellen. Die neue GUI-Technologie hat mit Qt 4.7 Einzug in Qt gefunden und sollte ursprünglich vor allem die Erstellung moderner grafischer Benutzeroberflächen auf mobilen Geräten mit Touch-Bedingung, wie Smartphones und Tablets, ermöglichen. Sie wird jedoch auch vollständig auf PCs, für die Erstellung von Desktop-Anwendungen unterstützt.

3.1.2.1 QML

Das Design und die Implementierung der grafischen Oberfläche erfolgt mit einer eigens dafür konzipierten deklarativen Sprache namens Qt Modeling Language (QML). Die Sprache basiert auf CSS und JavaScript. Im einfachsten Fall werden durch QML in CSS Syntax Properties von Objekten beschrieben. Es ist jedoch auch möglich, beispielweise JavaScript Methoden für die Implementierung eines Qt Signal-Handler zu verwenden oder JavaScript Ausdrücke bei der Wertzuweisung eines Property einzusetzen.

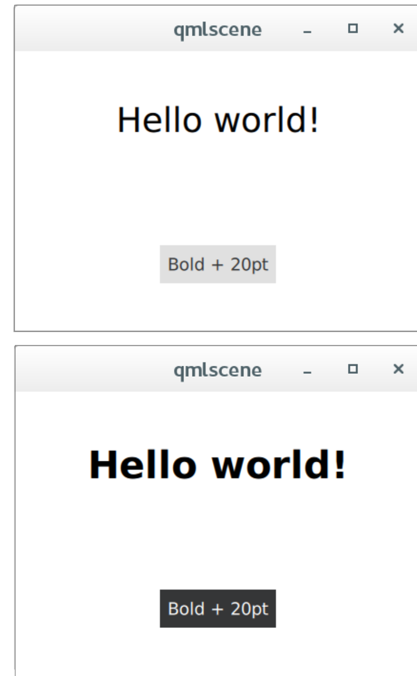
Zu den fundamentalen Konzepten von QML gehört der Property-Binding-Mechanismus. Dieser erlaubt ein Property A an den Wert eines anderen Property B zu binden, sodass auch bei Änderung von B, der Wert von A aktualisiert wird. Bei der Zuweisung von Property B wird somit automatisch das Binding zu A hergestellt. Ein QML Beispielcode, der die Zuweisung von

konstanten Werten als auch die Verwendung des Property-Binding-Mechanismus (Properties `font.pointSize` und `font.bold`) zeigt, ist in Abbildung 3.1 ersichtlich.

```
Rectangle {
    id: rect
    width: 450
    height: 300
    color: "white"
    ColumnLayout {
        id: layout
        anchors.fill: parent
        Text {
            id: text
            y: 10
            text: "Hello world!"
            color: "black"
            Layout.alignment: Qt.AlignHCenter
            font.pointSize: button.checked ? 20 : 18
            font.bold: button.checked
        }
        Button {
            id: button
            text: "Bold + 20pt"
            checkable: true
            Layout.alignment: Qt.AlignHCenter
        }
    }
}
```

Property Bindings: (markiert: `font.pointSize` und `font.bold`)

(a) QML Dokument



(b) Darstellung des QML Dokument mit dem Qt Tool `qmlscene`

Abbildung 3.1: "Hello World" GUI-Anwendung mit Qt Quick

Die QML und C++ Seite lassen sich über den durch die `QmlEngine` bereitgestellten `QQmlContext` miteinander verbinden. Über den `QQmlContext` können Instanzen von `QObject` abgeleiteten Klassen der QML Seite zur Verfügung gestellt werden. Umgekehrt wird beim Laden eines QML Dokuments in C++ ein `QObject` Pointer auf das Root-Objekt des geladenen QML Dokuments returniert. Aufgrund der hierarchischen Objektstruktur lassen sich somit alle Child-Objekte finden und als C++ Objekte ansprechen.

Das Data-Binding zwischen Objekten der C++ und QML Seite erfolgt über das Qt Meta-System. Auf `QObject` basierte C++ Klassen lassen sich über das Qt Meta-System als QML Type registrieren und können unter dem gewählten Namen in QML verwendet werden. C++ Klassenattribute, welche als Qt Property deklariert wurden, stehen auch in QML als Property zur Verfügung. Des Weiteren sind mit dem Makro `Q_INVOKABLE` ausgezeichnete Klassenmethoden und Qt Signals in QML verfügbar.

Analog zu `QWidget` im Qt Widget-System ist `QQuickItem` die Basisklasse für alle UI-Komponenten in Qt Quick und bietet ein ähnliches Interface für die Event-Verarbeitung von Benutzereingaben.

3.1.2.2 Szenengraph

Die grafische Darstellung eines `QQuickItem` wird im Gegensatz zum Qt Widget-System nicht imperativ festgelegt, also durch Absetzen von Zeichenkommandos, sondern deklarativ in einem Szenengraph, welcher durch einen Szenengraph-Renderer standardmäßig unter Verwendung von OpenGL gezeichnet wird. Dieser als Retained-Mode-Rendering [50, S. 23]) bezeichnete Ansatz erlaubt es dem Renderer, durch Vorliegen einer Beschreibung der gesamten zu zeichnenden Szene, Optimierungen durchzuführen. Zum Beispiel können statische Geometriedaten so einmalig OpenGL als Ressource zur Verfügung gestellt werden. Beim wiederholten Zeichnen kann die Grafikkarte somit in effizienter Weise auf die Geometriedaten zugreifen, z. B. durch Lesen vom Grafikkartenspeicher.

Für den Szenengraph-Renderer stehen bezüglich Threading eine “non-threaded” und eine “threaded” Variante zur Verfügung. In der “non-threaded” Variante erfolgt das Rendering im GUI-Thread (Main-Thread) der Applikation. Die “threaded” Variante hingegen verwendet einen eigenen Render-Thread pro Szenengraph. Jedes Fenster besitzt also in dieser Variante einen eigenen Render-Thread. Dadurch lässt sich das Rendering auf Multi-Core Prozessoren besser parallelisieren.

Die zeitliche Abfolge der Verarbeitunsschritte eines Renderdurchgangs zur Generierung eines neuen Bildes (Frame) ist in vereinfachter Darstellung in Abbildung 3.2 für beide Varianten ersichtlich. Durch Aufruf der Methode `QQuickItem::update()` wird signalisiert, dass ein `QQuickItem` sich neu darstellen möchte und daher eine Aktualisierung des Szenengraphen am Beginn eines neuen Renderdurchgangs erforderlich ist. In der “threaded” Variante wird zum Aktualisieren des Szenengraphen der GUI-Thread vom Szenengraph-Renderer zuerst blockiert. In weiterer Folge, gibt der Szenengraph-Renderer durch Aufruf der Methode `QQuickItem::updatePaintNode()` dem `QQuickItem` die Möglichkeit dessen Sub-Knoten/Baum im Szenengraph anzupassen. Nach erfolgter Aktualisierung des Szenengraphen wird der GUI-Thread in der “threaded” Variante wieder freigegeben. Die anschließende Traversierung des Szenengraphen entsprechend dem vom Szenengraph-Renderer implementierten Renderalgorithmus und das Absetzen der OpenGL Befehle, einschließlich Framebuffer-Swap, erfolgt in der “threaded” Variante im Render-Thread und entlastet somit den GUI-Thread.

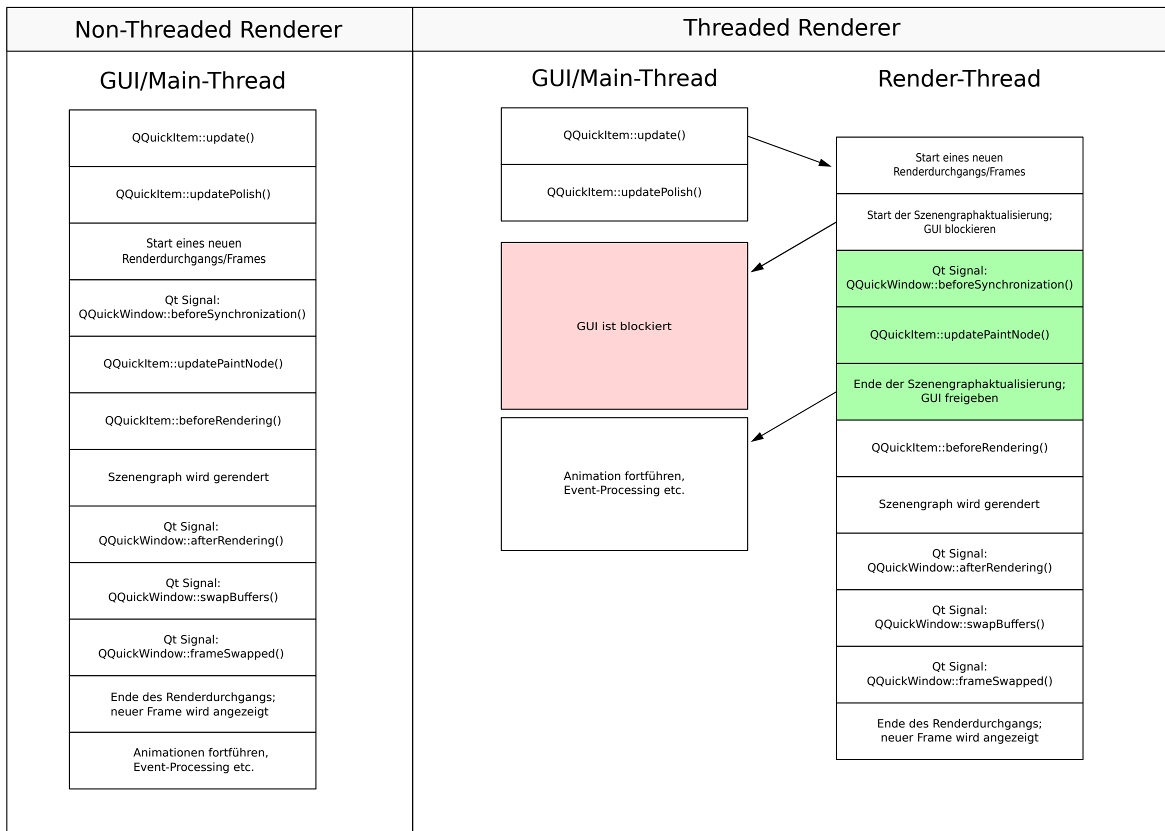


Abbildung 3.2: Verarbeitungsschritte während eines Renderdurchgangs ohne (links) und mit dediziertem Render-Thread (rechts); modifiziert übernommen aus der Qt-Dokumentation [7]

Bei der in dieser Arbeit vorliegenden Qt Version 5.9.1 ist das Szenengraph-API kein vollwertiger Ersatz für den Funktionsumfang von `QPainter` und entspricht im Wesentlich den von OpenGL gebotenen Funktionsumfang zur Darstellung von Vektorgrafiken. Es fehlen somit beispielsweise die Unterstützung für

- unterschiedliche Linienstile und Füllmuster,
- die Behandlung der Überschneidungsflächen (Line-Joins) von Liniensegmenten eines Linienzugs mit breiter Linienstärke und
- die Behandlung (Tesselierung) von konkaven Polygonen.

Der funktionalen Einschränkung wird durch den Einsatz des Qt Quick-Item `QQuickPaintedItem` Rechnung getragen. Es lassen sich durch Ableitung von `QQuickPaintedItem` Qt Quick-Items erstellen, welche auf konventionelle Weise mit `QPainter` gezeichnet werden können. `QPainter` operiert dabei auf einem Offscreen-Buffer, welcher schlussendlich als Textur dargestellt wird. Die Vorteile von Retained-Mode-Rendering gehen dadurch jedoch verloren.

3.2 Effizienter Texturdatentransfer mit OpenGL

Ein einfacher Ansatz um die Pixeldaten einer Textur mit OpenGL zu aktualisieren, ist die Verwendung von `glTexSubImage2D()`. Dieser Ansatz erfordert auf der Applikationsebene lediglich den Aufruf dieser OpenGL-Funktion, um die von der Applikation zur Verfügung gestellten Texturdaten vom CPU-Hauptspeicher in den GPU-Speicher zu transferieren (vgl. Abbildung 3.3). Der Vorgang erfolgt aus Sicht des Aufrufers synchron, das heißt, der Funktionsaufruf endet erst nachdem die Texturdaten transferiert wurden [48, S. 419]. Nach Venkataraman [48, S. 419] werden die Texturdaten vom OpenGL-Treiber in der Regel zunächst in einen vom Betriebssystem nicht-auslagerbaren Speicher, dem sogenannten Pinned-Memory, kopiert. Danach werden die Texturdaten in den GPU-Texturspeicher übertragen.

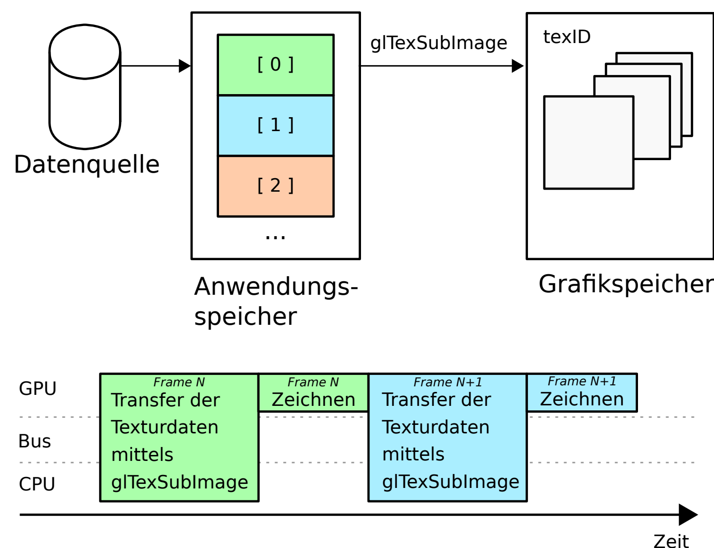


Abbildung 3.3: Synchroner Texturdatentransfer; modifiziert übernommen aus [48, S. 419]

Das zusätzliche Kopieren in den vom OpenGL-Treiber verwalteten Speicher und der blockierende Aufruf während der Zeit des Transfers der Texturdaten zur GPU sind als Nachteil dieser Methode zu sehen [48, S. 419]. Diese Nachteile lassen sich mittels OpenGL Pixel-Buffer-Objects (PBO) vermeiden. Die Texturdaten werden zunächst in das PBO übertragen. Ein PBO ermöglicht es, mittels Buffer-Mapping (e.g. `glMapBuffer()`) direkt auf den vom OpenGL Treiber reservierten Speicher zuzugreifen. Ein zusätzliches Kopieren der Daten kann somit vermieden werden [48, S. 420].

Das Speicherziel des PBO lässt sich durch den Verwendungshinweis (e.g. `GL_DYNAMIC_DRAW`) beeinflussen. Laut Hrabcak und Masserann [19, S. 395] kommt hierfür der CPU-Hauptspeicher (im Idealfall Pinned-Memory) oder der GPU-Speicher in Frage. Die Entscheidung obliegt letztendlich jedoch dem OpenGL-Treiber. Wird als Speicherort GPU-Speicher gewählt, so werden, nach Hrabcak und Masserann, die Daten mittels Direct-Memory-Access (DMA) in den GPU-Speicher übertragen. Sollte als Speicherort CPU-Hauptspeicher dienen, greift die GPU in diesem Fall über den Bus (e.g. über PCIe) auf das PBO zu [19, S. 393 ff.].

Die Programmausführung wird durch die Übertagung in den GPU-Speicher nicht angehalten. Um jedoch zu vermeiden, dass der Speicherinhalt des PBO während des Befüllens der Textur überschrieben wird, synchronisiert der OpenGL-Treiber den Zugriff und blockiert den Aufruf für den Schreibzugriff auf das PBO. Dies wird auch als implizite Synchronisation bezeichnet [19, S. 396 f.]. Hrabcak und Masserann [19, S. 398] sowie Venkataraman [48, S. 420] schlagen daher vor, mehrere PBOs im Round-Robin-Verfahren, wie in Abbildung 3.4 gezeigt, zu verwenden.

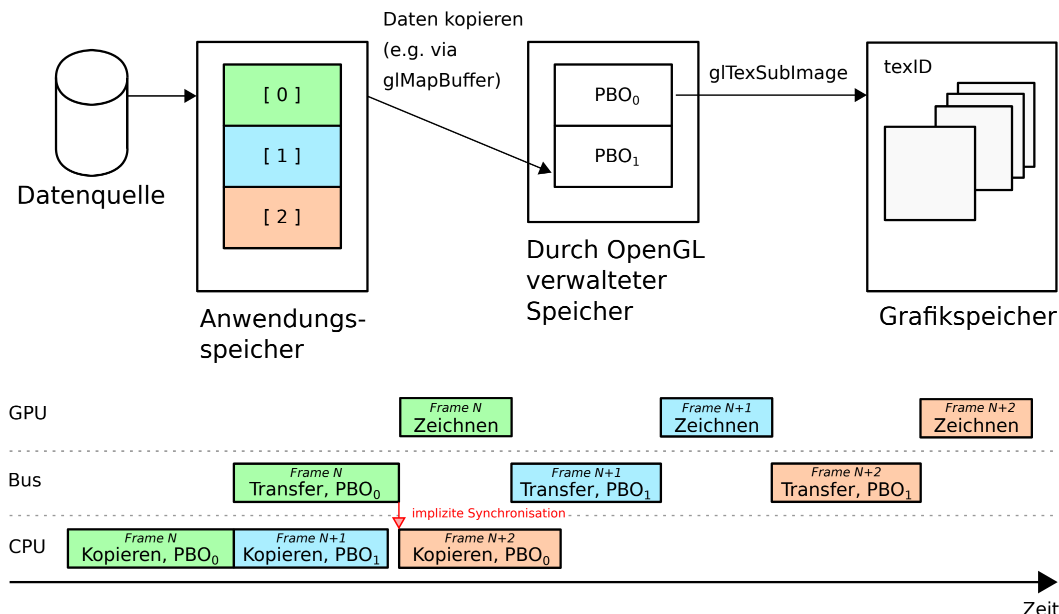


Abbildung 3.4: Asynchroner Texturdatentransfer unter Verwendung von PBOs im Round-Robin-Verfahren; modifiziert übernommen aus [48, S. 420]

Das Kopieren der Daten in das PBO lässt sich mithilfe eines Worker-Threads parallel zum Render-Thread der Applikation durchführen. Das Buffer-Mapping kann dabei durch den Render- oder Worker-Thread erfolgen. Im letzteren Fall ist für den Worker-Thread ein eigener OpenGL Context notwendig, welcher mit dem des Render-Thread mittels Context-Sharing verbunden werden müsste, um OpenGL Objekte miteinander teilen zu können.

Hrabcak und Masserann [19, S. 406] weisen jedoch darauf hin, dass die implizite Synchronisation über OpenGL Context-Grenzen hinweg nicht funktioniert und die Synchronisation explizit mit den von OpenGL zur Verfügung gestellten Synchronisationsobjekten (`glFenceSync()`) sichergestellt werden muss. Durch die Verwendung eines Render- und Worker-Thread mit separaten OpenGL Context lassen sich allerdings die in NVIDIA-Grafikkarten mit der Einführung der Fermi Architektur vorhandenen Dual-Copy-Engines [37] nutzen, sodass die Grafikkarte den Datentransfer zum GPU-Speicher parallel zur Abarbeitung der Render-Operationen durchführen kann [19, S. 421].

Eine schematische Darstellung eines Ansatzes, der den Texturdatentransfer parallel zum Rendern durch Einsatz eines Worker-Threads durchführt und die Nutzung von Dual-Copy-Engines ermöglicht, ist in Abbildung 3.5 ersichtlich. Die Texturen werden zwischen dem

Worker-Thread und dem Render-Thread geteilt. Während der Worker-Thread eine neue Textur vorbereitet, wird die aktuelle Textur vom Render-Thread verwendet. Die Umschaltung zwischen zu befüllender und zum Zeichnen verwendeter Textur muss synchronisiert mithilfe von OpenGL Synchronisationsobjekten erfolgen.

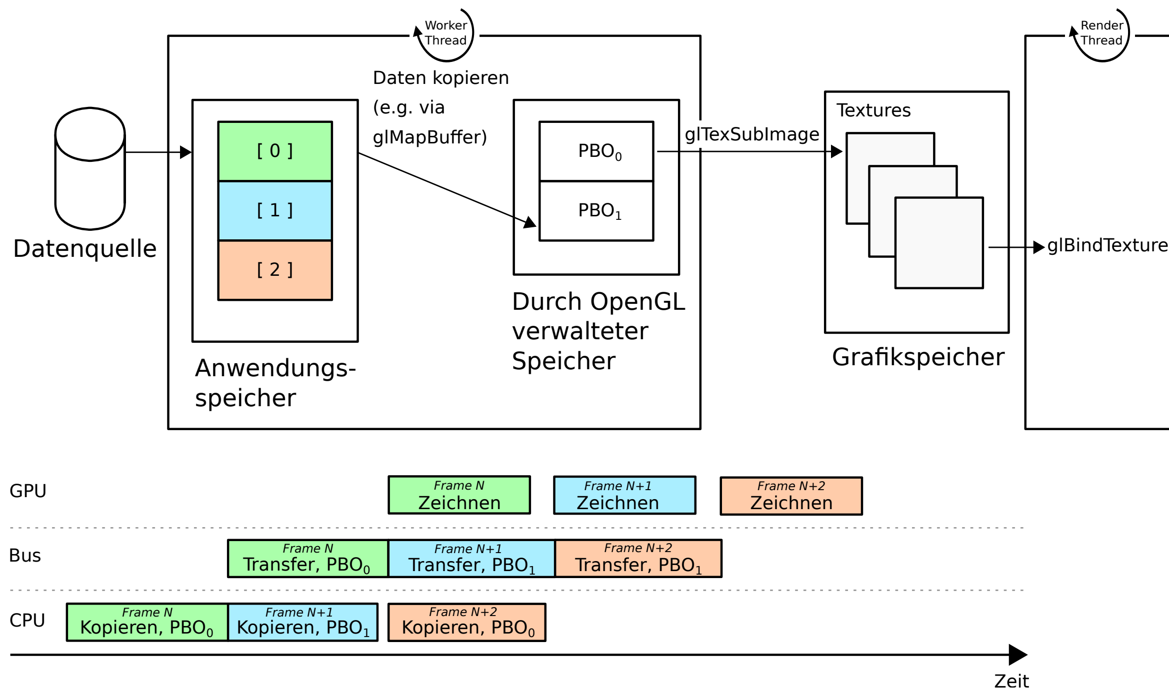


Abbildung 3.5: Asynchroner Texturdatentransfer mittels Worker-Thread und Context-Sharing; modifiziert übernommen aus [48, S. 422 f.]

3.3 Relevante Arbeiten

Die Möglichkeiten zur Nutzung moderner Grafikhardware haben auch bei der Realisierung von Radar-Displays ihre Anwendung gefunden. So stellen Stamatović et. al. in ihrer Arbeit [44] ein Radar-Display für die Luftraumüberwachung vor, das speziell für die Darstellung des Radarbildes auf OpenGL und Funktionen der Grafikhardware setzt.

Das Radar-Display umfasst unter anderem die Darstellung von geografischen Daten, wie z. B. Luftraumkarten, die Anzeige des Radarbildes mit synthetischen Afterglow und die Darstellung von Zielen. Für die Implementierung wird, wie auch in dieser Arbeit, auf das Qt SDK zurückgegriffen. Die grafische Umsetzung erfolgt jedoch mit dem Qt Graphics-View-Framework des konventionellen Qt-Widget-Systems.

Für die Radar-Scan-Conversion verwenden Stamatović et. al. [44] einen GPU-basierten Ansatz, der sich die Texture-Mapping-Funktion der Grafikkarte für die Konvertierung der Radardaten – von polar nach kartesisch – zunutze macht. Die Radarintensitätswerte werden dazu in Texturen gespeichert, deren Koordinatenachsen Azimut und Entfernung zugeordnet sind. Anschließend werden die aus der Azimutstellung des Radars resultierenden Kreissektoren

gezeichnet und mittels Texture-Mapping ausgefüllt. Das synthetische Afterglow wird mithilfe eines Fragment-Shader erzeugt, welcher die in den Texturen gespeicherten Radarintensitätswerten in regelmäßigen Zeitintervallen dekrementiert.

Die bei Low-End-Grafikkarten vorhandenen Einschränkungen hinsichtlich der Texturgröße – quadratisch und einer Zweierpotenz entsprechend – haben Stamatović et. al. durch Unterteilung der Texturen in rechteckige Blöcke Rechnung getragen. Die Blockgröße bestimmt somit die Kreissektorengröße in azimulater Richtung und die Kreissektorunterteilung in radialer Richtung. Sie besitzt ebenfalls Einfluss auf die erzielbare Framerate und Qualität der Radar-Scan-Conversion. Das Prinzip ist in Abbildung 3.6 ersichtlich. Ein Verfahren zur Bestimmung der optimalen Blockgröße in Abwägung der zu erzielenden Framerate und Qualität wird von Jevtić et. al. [25] beschrieben.

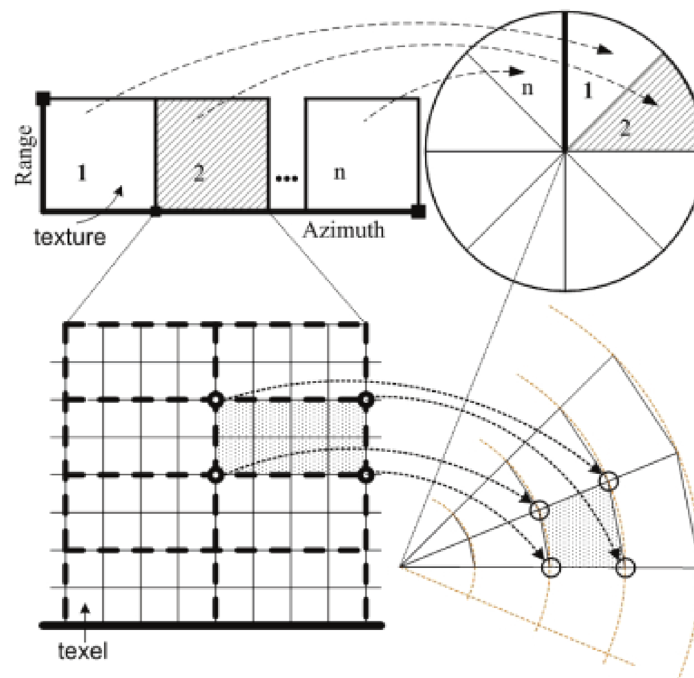


Abbildung 3.6: Prinzip zur GPU-basierten Radar-Scan-Conversion [25]

Der auf Texture-Mapping basierende Ansatz zur Radar-Scan-Conversion wird auch in der Arbeit von Sharma et. al. [43] dargestellt und hinsichtlich Durchsatz im Vergleich zu traditionellen CPU-basierten Algorithmen, als signifikant schneller bewertet. Pinzón und Espartero [41] haben gezeigt, dass sich dieser Ansatz auch für Echtzeitanwendungen mit hochauflösendem Radar eignet.

Pezhgorski und Lazarova [40] stellen einen alternativen GPU-basierten Ansatz vor. Für die Vorverarbeitung der Radardaten wird dabei auf die Open Computing Language (OpenCL) gesetzt. Die daraus resultierenden, polaren Radardaten werden OpenGL unter Verwendung einer OpenCL-Extension als Textur zur Verfügung gestellt. Die Radar-Scan-Conversion wird nicht mittels Texture-Mapping von Kreissektoren, sondern durch Koordinatenrücktransforma-

tion in einem Fragment-Shader realisiert. Der Radarintensitätswert an den ermittelten polaren Koordinaten wird anschließend durch Texture-Sampling der über OpenCL zur Verfügung gestellten Textur entnommen. Der gewählte Ansatz verfügt laut Pezhgorski und Lazarova über eine ausreichend hohe Framerate um in Echtzeitanwendungen zum Einsatz zu kommen.

In dem von der Firma ADB Safegate Austria entwickelten A-SMGCS wird die Radar-Scan-Conversion serverseitig durchgeführt und ist zudem in der Lage, das Bild von mehreren Radars zu fusionieren und in ein gemeinsames, geografisches Koordinatenbezugssystem zu transformieren. Im Unterschied zu den hier vorgestellten Arbeiten zur Realisierung eines Radar-Displays, wird daher in dieser Arbeit nicht auf einen GPU-basierten Ansatz zur Radar-Scan-Conversion eingegangen.

4 Entwurf und Implementierung

Wie in Abschnitt 2.5 zu sehen, ist der Aufwand für die Bildsynthese zu hoch, um auf dem Referenzsystem mehr als 250 Ziele pro Fenster darstellen zu können. Beim Entwurf des Prototypen ist also besonderes Augenmerk auf den Rendering-Prozess zu legen. Qt Quick bietet mit dem auf OpenGL und einem Szenengraph basierten Unterbau gute Voraussetzungen für ein performantes, hardwarebeschleunigtes Bodenlagedisplay.

Qt Quick ist im Gegensatz zu 3D-Grafik-Toolkits, wie OpenSceneGraph [39], jedoch nicht für die Verwendung als universell einsetzbares hochperformantes Grafik-Toolkit konzipiert, sondern dient primär der Realisierung moderner grafischer 2D-Benutzeroberflächen. Die Schnittstelle zwischen QML und C++ stellt gegenüber der direkten Ausführung in C++ einen Umweg dar und erhöht dadurch die Ausführungszeit von z. B. Zuweisungen und Funktionsaufrufen. Die Synergie und Flexibilität, die man durch die reguläre Verwendung von Qt Quick gewinnt, kann somit auf Kosten der HMI-Performance gehen, welche an der Anzahl gleichzeitig darstellbarer Ziele gemessen wird.

Die Problemstellung kann mit Qt Quick auf unterschiedlichen Abstraktionsebenen gelöst werden. So ermöglicht Qt Quick die Erstellung von neuen Szenengraph-Nodes und Qt Quick-Items, aber auch die direkte Verwendung von OpenGL. Die Möglichkeit mehrere Varianten testen, analysieren und gegenüberstellen zu können wird im Design des Prototypen verankert. Der Prototyp stellt somit auch eine Testplattform dar.

4.1 Architektur des HMI-Prototypen

Im Folgenden wird die Architektur des HMI-Prototypen beschrieben. Der Blick wird zuerst auf die zu implementierenden Datenschnittstellen gelenkt. Danach wird die Architektur in seinen Grundzügen beschrieben und auf einzelne Module eingegangen.

4.1.1 Datenschnittstellen und -quellen

Das HMI soll, wie in Anforderung R4 (siehe Abschnitt 2.4) verlangt, sowohl Echtdaten als auch synthetisch generierte Testdaten verarbeiten können. Für die Interprozesskommunikation verwendet die Firma ADB Safegate Austria ein proprietäres, TCP/IP basiertes Messaging-Protokoll (siehe Kapitel 2). Echtdaten und Echtdaten und synthetische Daten unterscheiden sich hinsichtlich des zur Übertragung verwendeten Message-Typs nicht. Zur Erfüllung dieser Forderung muss der Prototyp daher:

- das proprietäre ADB Safegate Austria Messaging-Protokoll unterstützen, und
- eine TCP/IP Verbindung zu einem beliebigen Server mit der Adresse `<host:port>` aufbauen können.

Es werden zwei getrennte TCP/IP-Verbindungen vorgesehen, um die Track- und SMR-Daten von verschiedenen Datenquellen über verschiedene TCP/IP-Ports empfangen zu können. Die Datenschnittstellen des HMI-Prototypen und die möglichen Datenquellen für den Empfang von Echtdaten und simulierten Daten sind in Abbildung 4.1 illustriert. Für autarkes Testen verfügt der Prototyp auch über eine prozessinterne Track-Datenquelle. In der Abbildung wird auf Spezialfälle, in denen der ACEMAX-Server anstelle von realen Sensordaten mit simulierten Daten gespeist wird, zugunsten der übersichtlicheren Darstellung verzichtet.

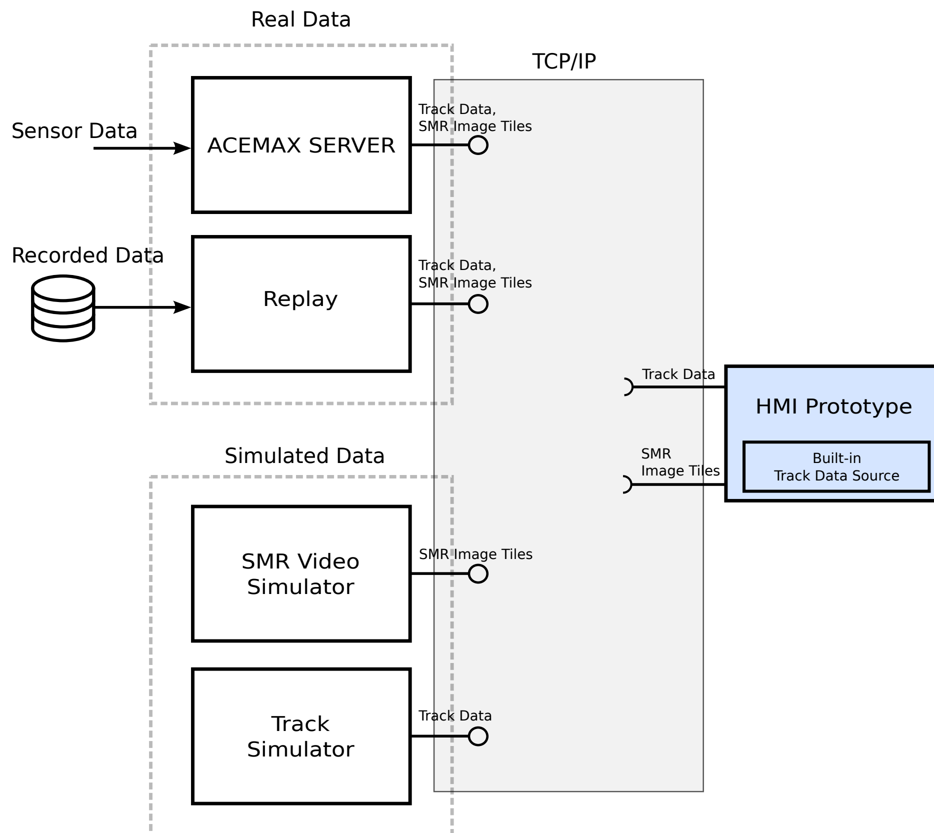


Abbildung 4.1: TCP/IP basierte Datenschnittstellen zwischen dem HMI-Prototypen und den zur Auswahl stehenden Datenquellen

4.1.2 Modulübersicht

In Abbildung 4.2 wird eine Modulübersicht gegeben. Dargestellt werden die nach funktionalen Gesichtspunkten angeordneten Module des HMI-Prototypen und die Abhängigkeiten zu den ADB Safegate Austria und Qt 5 Bibliotheken. Unter Modul wird hier die thematische Gruppierung von Klassen und Funktionen verstanden. Der Prototyp wird demnach in sechs Module unterteilt.

Der Anwendungskern und alle Klassen und Funktionen, die sich keinem anderen Modul zuordnen lassen, sind im Modul “`Application`” enthalten. Das Modul “`Quick GUI`” umfasst

alle Komponenten der mit Qt Quick realisierten GUI. Der Zweck des Moduls “`Configuration`” ist die Persistenz von Konfigurationsparametern. Es stellt anderen Modulen eine Schnittstelle zum Lesen und Schreiben der Konfigurationsdaten bereit.

Das Modul “`Map Data`” lädt die Flughafenkarten von einer Datenquelle (e.g. Datei) und stellt diese der Anwendung zur Verfügung. Die Track- und SMR-Daten werden vom Modul “`Track Data`” beziehungsweise vom Modul “`SMR Video`” über das Netzwerk empfangen, verarbeitet und der Anwendung zur Verfügung gestellt. Für die Rekonstruktion des beim Versenden in Bildkacheln zerlegten SMR-Bildes verwendet das Modul “`SMR Video`” den SMR-Image-Composer, an dessen Ausgang ein sich kontinuierlich aktualisierendes Bild zur Verfügung steht.

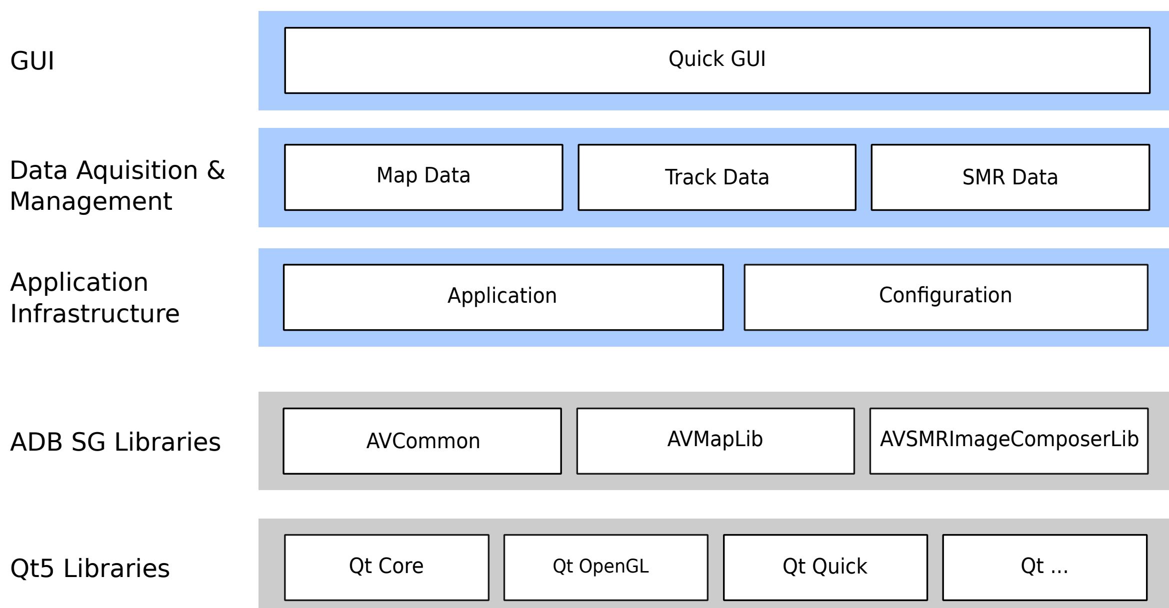


Abbildung 4.2: Module und Abhängigkeiten zu den ADB Safegate Austria und Qt Bibliotheken des HMI-Prototypen

Die Entwicklung des Prototyps basiert auf Qt 5.9.1. Die Verwendung von Qt 5 erfordert die Portierung aller vom Prototyp verwendeten, auf Qt 3 basierenden ADB Safegate Austria Bibliotheken. In Tabelle 4.1 wird die Codebasis angeführt und eine kurze Beschreibung der Bibliotheken gegeben.

Die Separierung von Model und View wird durch Qt forciert. In Qt verankert ist die als “Model-View” bekannte, modifizierte Form des Entwurfsmusters Model-View-Controller (MVC) [4, S. 124]. Die Controller-Komponente, die auf die Benutzereingaben reagiert, wird dabei mit der View-Komponente kombiniert. Zusätzlich sieht Qt die Verwendung einer Delegate-Komponente vor, mit der das Rendern und Editieren von Items individuell an die Model-View-Paarung angepasst werden können. Die Aufteilung im Sinne des MVC-Entwurfsmusters wird auch im Design des HMI-Prototypen beherzigt. C++ Klassen, die als Model fungieren, stehen dadurch in QML als QML Type parat.

Bibliothek	Codebasis	Beschreibung
`AVCommon`	Qt 5	Sammlung von Basisbibliotheken, welche unter anderem Logging-, Messaging- und Networking-Funktionalität zur Verfügung stellen; mit enthalten ist auch ein Framework zum Laden und Speichern von Konfigurationsparametern.
`AVMapLib`	Qt 3	Bibliothek zum Laden und Rendern von Flughafengeländedaten.
`AVSMRImageComposerLib`	Qt 3	Beinhaltet das Modul SMR-Image-Composer zur Komposition des SMR-Bildes aus den SMR-Daten mehrerer Radarquellen, siehe auch Abschnitt 2.3.4.3.

Tabelle 4.1: Beschreibung der verwendeten ADB Safegate Austria Bibliotheken

ADB Safegate Austria verfügt über ein eigenes Framework für die persistente Speicherung von Konfigurationsparametern. Die Konfigurationsdaten lassen sich objektorientiert durch Ableitung von einer Basisklasse des Frameworks abbilden. Diese Konfigurationsklassen können Standarddatentypen als auch Containerdatentypen beinhalten und lassen sich auch verschachteln. Für die Speicherung wird ein Textformat verwendet, Dateien lassen sich daher von Hand mithilfe eines Standardtexteditors bearbeiten.

4.2 Aufbau der Benutzeroberfläche

Das GUI soll, wie in Anforderung R1 beschrieben, aus einem Hauptfenster und mehreren Zoom-Fenstern bestehen. Haupt- und Zoom-Fenster stellen zwei verschiedene Fensterklassen dar. Zerlegt man die Fensterklassen in ihre Grundkomponenten, so lässt sich die Anzeige der Bodenlage als ein zentrales Element beider Fensterklassen identifizieren. Diese Komponente wird in weiterer Folge als View bezeichnet. Während ein Zoom-Fenster nur aus der View-Komponente besteht, verfügt das Hauptfenster zusätzlich noch über eine Toolbar und eine Sidebar. Der Sachverhalt ist auch in Abbildung 4.3 dargestellt.

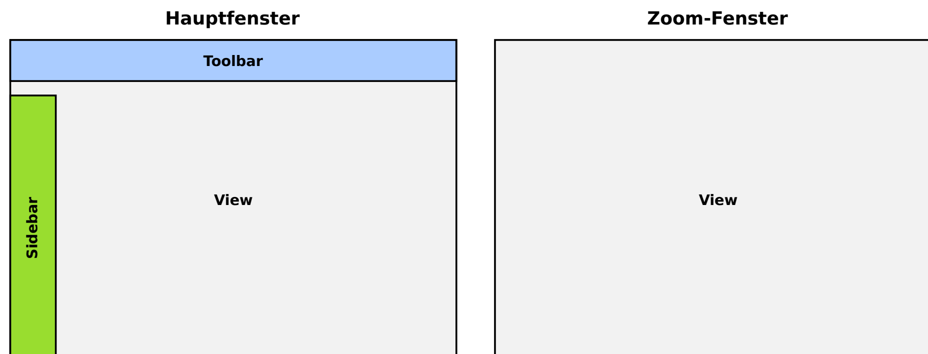


Abbildung 4.3: GUI-Komponenten des Hauptfensters und eines Zoom-Fensters

4.2.1 Fenster

Es gibt in Qt Quick zwei Möglichkeiten um das Hauptfenster und die Zoom-Fenster zu erzeugen:

- in C++: durch Instanziierung dedizierter Klassen (e.g. `QQuickView`),
- in QML: durch Verwendung der QML-Typen `ApplicationWindow` und `Window`.

Die QML-Variante hat den Vorteil, dass die in klassischen Desktopanwendungen übliche Unterteilung des Hauptfensters unterstützt wird. Das Hauptfenster wird also bereits in die Bereiche Hauptinhalt, Menü, Statusanzeige und Toolbar unterteilt. Die C++ Klasse, welche den QML-Typ `ApplicationWindow` implementiert, ist nicht Teil der öffentlichen API. Diese Funktionalität ist daher nur in QML verfügbar.

Der Nachteil der QML-Variante ist, dass auf die Erstellung des Fensters – insbesondere auf die Erstellung des OpenGL-Kontextes – kein Einfluss genommen werden kann. Diese Einschränkung ist für die Entwicklung des Prototyps, bei der mehrere Lösungsvarianten getestet werden, zu restriktiv. Aus diesem Grund werden Fenster direkt in C++ erzeugt.

Es wird für alle Fenster eine `QQmlEngine`-Instanz verwendet. Somit teilen alle Fenster den selben `QmlContext` und dadurch auch alle über den Kontext zu Verfügung gestellten C++ Objekte.

4.2.2 View

Der für die Anzeige der Bodenlage verantwortliche Teil des Fensters (View) wird als eigenes Qt Quick-Item (`QuickLayerView`) realisiert. Die Größe eines `QuickLayerView`-Items ist durch den Einsatz von Qt Quick-Layouts an die Fenstergröße gekoppelt. Der sichtbare geografische Ausschnitt wird durch eine affine Transformationsmatrix (View-Matrix) beschrieben. Zur Manipulation der View-Matrix stehen folgende Operationen zur Verfügung:

- Verschiebung um die Distanz (d_x, d_y):

$$M_{view} = M_{view_{-1}} \cdot M_T(d_x, d_y)$$

- Zoom um den Faktor k mit dem Zentrum $(origin_x, origin_y)$:

$$M_{view} = M_{view_{-1}} \cdot M_T(-origin_x, -origin_y) \cdot M_S(k, k) \cdot M_T(origin_x, origin_y)$$

- Rotation um den Punkt $(origin_x, origin_y)$ um den Winkel θ:

$$M_{view} = M_{view_{-1}} \cdot M_T(-origin_x, -origin_y) \cdot M_R(\theta) \cdot M_T(origin_x, origin_y)$$

Wobei:

- $M_T(tx, ty)$, $M_S(sx, sy)$, $M_R(\theta)$ die affinen Transformationsmatrizen [17, S. 88] für Translation, Skalierung und Rotation darstellen,
- mit M_{view-1} und M_{view} die View-Matrix vor bzw. nach der Ansichtsänderung bezeichnet wird, und
- $(x', y', 1) = (x, y, 1) \cdot M_{view}$ die Transformation eines Punktes mit den Koordinaten (x, y) angibt (post-multiply).

Das `QuickLayerView`-Item verwaltet, wie in Abbildung 4.4 dargestellt, eine Liste von Layern vom Typ `QuickLayerViewLayer`, analog zur Widget-basierten Implementierung (siehe Abschnitt 2.3.4). Die Z-Ordnung der Layer erfolgt nach den Regeln von Qt Quick. Die Ordnung ist somit standardmäßig durch die Deklarationsreihenfolge bestimmt und lässt sich durch die Angabe einer z-Koordinate ändern.

Neue Layer können in C++ durch Ableitung von der Basisklasse `QuickLayerViewLayer` implementiert werden. Es lassen sich jedoch auch neue Layer in QML definieren, indem ein beliebiges Qt Quick-Item als Kindelement von `QuickLayerViewLayer` fungiert.

Die View-Matrix wird von `QuickLayerView` vorgehalten und von `QuickLayerViewLayer` als Basistransformation (Qt-Klasse `QSGTransformNode`) des zugehörigen Subbaumes im Szenengraphen gesetzt. Die Ansichtstransformation wird somit auch an die Kindelemente von `QuickLayerViewLayer` vererbt. Die Vererbung der View-Matrix mittels Transformatinsknoten im Szenengraph lässt sich pro Layer über das boolsche Attribut
`QuickLayerViewLayer.applyViewTransform` deaktivieren und bietet so die Möglichkeit, die View-Matrix selektiv anzuwenden.

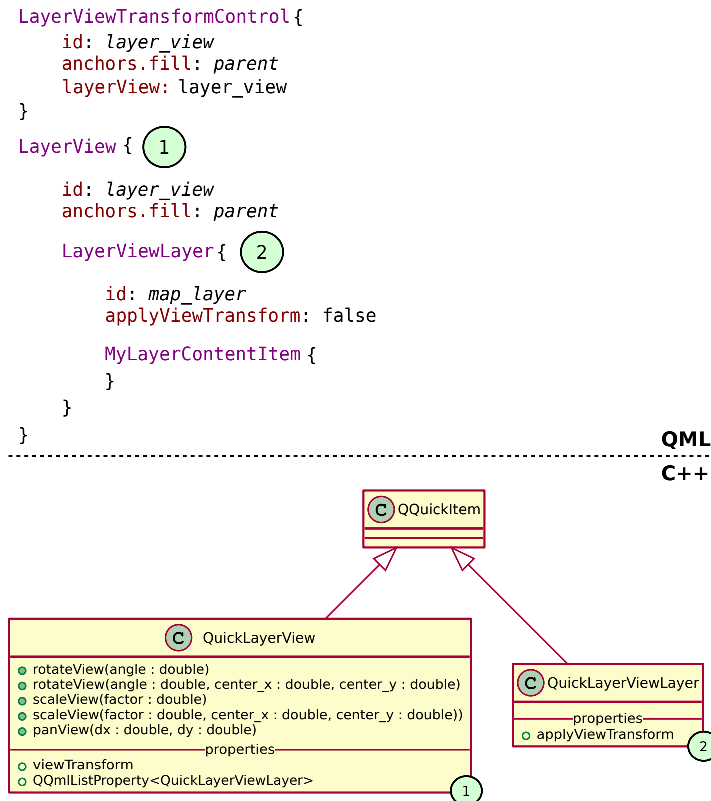


Abbildung 4.4: Realisierung der View-Komponente: Klassendiagramm und beispielhafter QML-Code

4.3 Anzeige der Flughafenkarte

Die Flughafengeländedaten liegen in einem Vektorformat vor (siehe Abschnitt 2.3.4.2). Die bestehende Bibliothek zum Laden und Rendern der Flughafenkarte wurde in Qt 3 implementiert und verwendet für die Darstellung die Qt-Klasse `QPainter`.

Wie in Abschnitt 3.1 erläutert, stellt `QPainter` eine imperative Grafik-API bereit, während Qt-Quick auf eine deklarative Grafik-API setzt. Es muss also ein Weg gefunden werden, der entweder beide Paradigmen vereint, oder die Karte im Szenengraph abzubilden vermag.

4.3.1 Auswahl des Lösungsansatzes

Im Folgenden werden die identifizierten Möglichkeiten zum Rendern der Flughafenkarte mit Qt Quick, zusammengefasst gegenübergestellt und innerhalb dieser eine Auswahl getroffen.

4.3.1.1 Ansatz "Szenengraph"

Die Flughafenkarte stellt eine statische Szene dar. Die Verwendung eines Szenengraphs hätte gegenüber dem imperativen Ansatz den Vorteil, mit dem Szenengraphen Optimierungen

durchzuführen, die die Anzahl der Zeichenbefehle minimieren (Batch-Processing). Die statischen Geometriedaten können zudem auf der Grafikkarte gespeichert werden und müssen nicht pro Frame neu übertragen werden.

Allerdings bietet der Qt-Quick-Szenengraph in Qt 5.9.1 im Vergleich zu `QPainter` nur einen sehr eingeschränkten Funktionsumfang, was die Darstellung von Vektorgrafiken betrifft. Der Szenengraph bietet lediglich einen Geometrieknoten, der sich auf die in OpenGL verfügbaren geometrischen Primitive beschränkt. Gegenüber der Verwendung von `QPainter` ist also zusätzlicher Implementierungsaufwand nötig, um alle Kartenelemente inklusive ihrer visuellen Eigenschaften zu unterstützen. Ein Szenengraph-basierter Ansatz müsste unter anderem

- konkave Polygone unterstützen,
- Ellipsen und Kurven approximieren (z. B. durch Linien),
- verschiedene Füllmuster und Linienstile unterstützen und
- Text darstellen können.

4.3.1.2 Ansatz "QQuickPaintedItem"

Qt Quick bietet mit der C++ Klasse `QQuickPaintedItem` eine Möglichkeit, um den `QPainter`-basierten Code zu integrieren. Der Render-Prozess ist zweistufig: Mit `QPainter` wird zuerst in einen Offscreen-Buffer gezeichnet und danach wird der Inhalt als Textur angezeigt.

Das API der Qt-Klasse `QQuickPaintedItem` lässt für den Offscreen-Buffer dabei die Wahl zwischen einem OpenGL Frame-Buffer-Object (FBO) oder einem `QImage` (CPU-seitiger Bildpuffer). Die OpenGL-FBO-Variante erlaubt den Rendervorgang vollständig auf der Grafikkarte auszuführen. Die Verwendung von `QImage` bedarf hingegen des Einsatzes von Sofware-Rendering und erfordert zusätzlich die Bereitstellung der im CPU-Hauptspeicher liegenden `QImage`-Daten als OpenGL Textur. Dieser zusätzliche Schritt fällt vor allem dann ins Gewicht, wenn sich der darzustellende Inhalt ständig ändert.

Beide Varianten unterscheiden sich zudem hinsichtlich des eingesetzten Antialiasing-Verfahrens. Die OpenGL-FBO-Variante greift auf die von der Grafikkarte zur Verfügung gestellten Postfiltering-Verfahren, wie z. B. Multisampling-Antialiasing (MSAA) [2, S. 128 ff.], zurück. Das führt abhängig von der Anzahl der Samples zu einem deutlich größerem Grafikkartenspeicherverbrauch [26]. Soll eine höhere Antialiasing-Qualität erreicht werden, ohne dabei die Ressourcen der Grafikkarte Übermaß zu beanspruchen, wäre der `QImage`-Variante Vorzug zu geben (Sofware-Renderering mit Prefiltering-Antialiasing).

4.3.1.3 Ansatz "OpenGL und QPainter"

Der Qt-Quick-Scenegraph-Renderer stellt zwei Einstiegspunkte zur Verfügung. Der erste Einstiegspunkt, ermöglicht die Ausführung von OpenGL-Code noch bevor die Qt-Quick-Szene gezeichnet wird. Die Karte lässt sich somit direkt unter Verwendung der Qt-Klassen `QPainter` und `QOpenGLPaintDevice` in den Default-Framebuffer zeichen.

4.3.1.4 Bewertung der Lösungsansätze

Der Ansatz “Szenengraph” ist aufgrund der im Vergleich zu `QPainter` im Funktionsumfang eingeschränkten Szenengraph-API implementationstechnisch am aufwändigsten. Der in Anspruch genommene Grafikkartenspeicher ist unabhängig von der Fenstergröße und im Normalfall wesentlich geringer als beim Ansatz “`QQuickPaintedItem`”. Zudem ist zu erwarten, dass dieser Ansatz dann am effizientesten ist, wenn das Neuzeichnen der Karte bei unveränderten Geometriedaten aber sich ändernder Ansichtstransformation erfolgt. Die Geometriedaten müssen nämlich von Frame zu Frame nicht neu übertragen und können direkt in den Default-Framebuffer gezeichnet werden.

Der Ansatz “OpenGL und QPainter” lässt sich nicht als `QQuickItem` realisieren und erfordert somit eine Sonderbehandlung. Die Geometriedaten müssen von Frame zu Frame neu übertragen werden.

Beim Ansatz “`QQuickPaintedItem`” ist das Neuzeichen der Karte – unter der Annahme, dass die Karte zugunsten der Darstellungsqualität neu gezeichnet und nicht das gerasterte Bild skaliert wird – nur bei einer Änderung der Karte oder der Ansichtstransformation erforderlich. Aufgrund der im Regelbetrieb gleichbleibenden Ansichtstransformation und der statischen Natur der Karte, kann somit in der Mehrzahl der Fälle auf die bereits gerasterte Karte, die als Textur im Grafikspeicher liegt, zurückgegriffen werden. Im Gegensatz zu den beiden anderen Ansätzen, ist der Grafikkartenspeicherverbrauch abhängig von der Fenstergröße.

Im Zuge dieser Arbeit wird der Ansatz “`QQuickPaintedItem`” implementiert. Er lässt die Verwendung des bereits existierenden `QPainter`-basierten Zeichencodes zu und ist dadurch der ökonomisch beste Ansatz.

4.3.2 Implementierung

Die Karte wird durch das Qt Quick-Item `Map` dargestellt. Die C++-Implementierung ist dank des Einsatzes von `QQuickPaintedItem`, nicht sehr aufwändig. Erfordert sie im Wesentlichen doch nur die Ableitung von `QQuickPaintedItem` und die Implementierung der Methode `QQuickPaintedItem::paint()`.

Die Funktionalität zum Laden und Zeichnen der Flughafenkarte wird durch die Bibliothek `AVMapLib`) zur Verfügung gestellt, welche von Qt 3 nach Qt 5 zu portieren ist. Die C++-Klasse `AVMap` dieser Bibliothek repräsentiert die Flughafenkarte im Speicher. Sie besteht aus Gruppen (`AVDrawingGroup`), die wiederum Zeichenelemente (von `AVDrawingElement` abgeleitete Klassen) beinhalten. Vor der Portierung werden Unit-Tests für alle Klassen dieser Bibliothek erstellt, um das Verhalten zu konservieren und um etwaige, durch die Portierung verursachte Fehler zu vermeiden.

Ein Klassendiagramm und ein QML-Codeausschnitt, der den Einsatz von Qt Quick-Item `Map` als Layer zeigt, ist in Abbildung 4.5 ersichtlich. Die darzustellende Karte, die Hintergrundfarbe und die zu verwendende Ansichtstransformation lassen sich in QML als Qt Property definieren.

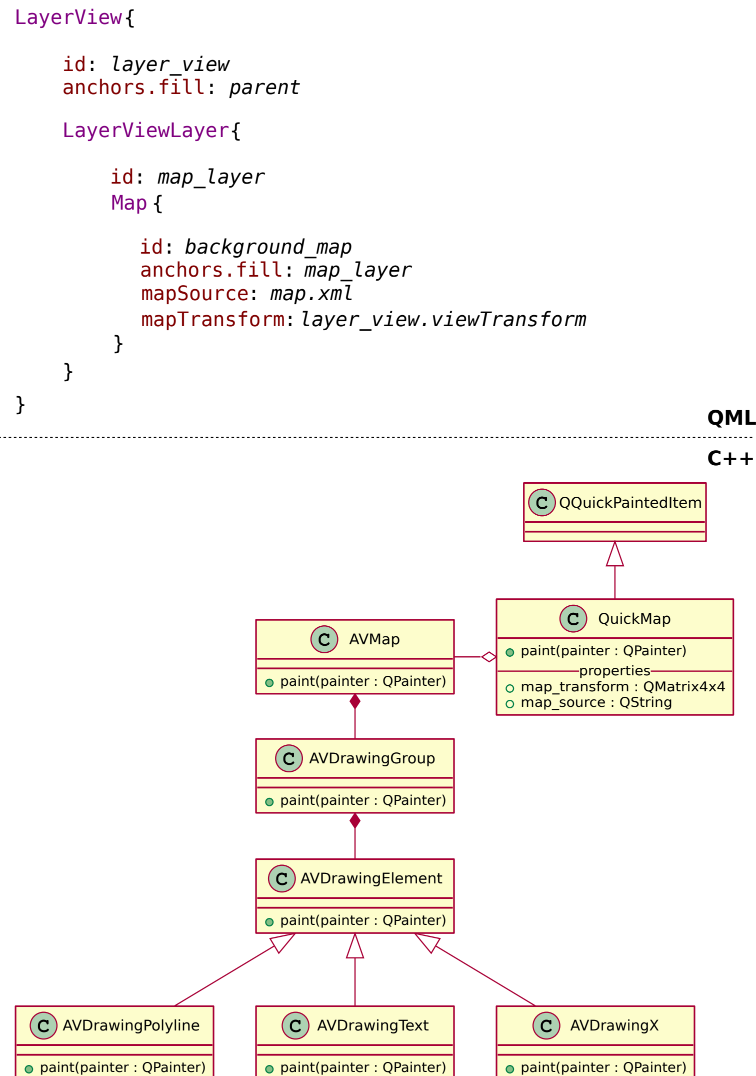


Abbildung 4.5: Anwendung des Qt Quick-Items `Map`: Klassendiagramm (unten) und beispielhafter QML-Code (oben)

4.4 Anzeige des Radarbildes

Die Anzeige des Bodenradarbildes erfordert im bestehenden HMI einen zweistufigen Prozess (vgl. Abschnitt 2.3.4.3). Stufe 1 umfasst die Komposition der vom ACEMAX-Server übertragenen Bildkacheln und die Erzeugung des Afterglow-Effektes. Dieser Teil wird in einem eigenen Thread betrieben. Die Darstellung des Bodenradarbildes in Stufe 2 erfolgt zur Gänze im Haupt-Thread in dem die gesamte grafische Benutzeroberfläche realisiert wird.

Die Radarbildanzeige trägt, wie in Abschnitt 2.5 festgehalten, deutlich zur CPU-Last bei, wobei sich der Aufwand gleichmäßig auf beide Stufen aufteilt (siehe Testkonfiguration "SMR-Datenv." bzw. "SMR-Anzeige"). Die Verarbeitungszeit der ersten Stufe hat aufgrund der

parallelen Ausführung und der asynchronen Kommunikation zwischen den Stufen keinen Einfluss auf die Zeit, die zum Rendern des Bodenlagebildes benötigt wird. Der Render-Prozess lässt sich also nur durch Verringerung der Verarbeitungszeit von Stufe 2 beschleunigen.

4.4.1 Auswahl des Lösungsansatzes

Fasst man die in Abschnitt 2.3.4.3 erläuterte Funktionsweise des SMR-Layer zusammen, kann Stufe 2 in zwei Verarbeitungsschritte aufgeteilt werden. In Verarbeitungsschritt 1 werden die aktualisierten im Fenster sichtbaren Bereiche des am Ausgang von Stufe 1 vorliegenden Radarbildes mittels einer Lookup-Tabelle in das Ansichtskoordinatensystem des Fensters transformiert. Die transformierten Teilbilder werden in Verarbeitungsschritt 2 in den Bildpuffer (SHM-Pixmap) des SMR-Layers, der das Radarbild in der vom Benutzer festgelegten Ansicht speichert, kopiert. Der SMR-Layer fordert schlussendlich ein View-Update an, um den Bildpuffer zu "zeichnen". Bei einer Ansichtsänderung muss die Lookup-Tabelle für die neue Ansichtstransformation neu berechnet und die Verarbeitungsschritte 1-2 müssen für den gesamten sichtbaren Bereich des Radarbildes erneut ausgeführt werden.

Verarbeitungsschritte parallelisieren Die Verarbeitung von Stufe 2 aller Fenster wird im bestehenden HMI sequentiell im GUI-Thread durchgeführt. Es ist folglich anzunehmen, dass eine parallele Ausführung die CPU-Last des GUI-Threads reduziert und die Reaktivität der GUI erhöht. Dies kann bereits durch den Einsatz der "Threaded-Render-Loop" von Qt Quick, eigener Render-Thread pro Fenster, realisiert werden.

Verarbeitungsschritte optimieren Die Bildtransformation wurde im bestehenden HMI bereits auf Basis einer Lookup-Tabelle zur Verbesserung der Laufzeit implementiert. Wie in Abschnitt 2.3.4.3 dargestellt, wird beim Skalieren die Nearest-Neighbour-Interpolation verwendet. Die dafür beanspruchte CPU-Zeit kann jedoch voraussichtlich durch eine GPU-seitige Bildtransformation weiter reduziert werden. Beim Hinauszoomen kann zudem, aufgrund der Harwareunterstützung, bilineare Interpolation zur Reduktion von Aliasing-Effekten angewendet werden. Allerdings muss für diesen Ansatz der GPU das in Systemauflösung vorliegende Radarbild (Stufe 1) zur Verfügung gestellt werden. Dies lässt sich mit dem in Abschnitt 3.2 erläuterten Verfahren in einer effizienten Art und Wiese bewerkstelligen. Das Radarbild kann dadurch in einen der GPU-zugänglichen Speicher asynchron zum Rendervorgang mittels eines dedizierten Threads übertragen werden.

Datenmenge reduzieren Verarbeitungsschritt 2 implementiert im Allgemeinen die Übertragung der Bilddaten in einem dem Grafiksystem zugänglichen Speicherbereich. In Qt Quick stellt OpenGL die Schnittstelle zum Grafiksystem dar. Die zu an OpenGL zu übermittelnden SMR-Daten liegen als 32-Bit Bild am Ausgang von Stufe 1 vor. Die Radarechointensität ist im Alphakanal enthalten. Der RGB-Wert dient zur farblichen Unterscheidung zwischen Afterglow und dem aktuellen Intensitätswert. Der Afterglow-Effekt lässt sich jedoch, wie in Abschnitt 3.3 beschrieben, auch auf der Grafikkarte erzeugen. Die zur übertragende Datenmenge könnte dadurch auf ein Viertel (8-Bit statt 32-Bit) reduziert werden.

Alle drei Möglichkeiten lassen sich kombinieren. Wobei zwei zentrale Ansätze hinsichtlich der Durchführung der Bildtransformation abgeleitet werden können: Der erste Absatz führt die Bildtransformation analog dem bestehenden HMI CPU-seitig durch. Der zweite Ansatz nützt die Möglichkeiten der Grafikhardware zur Bildtransformation.

Um die Machbarkeit des zweiten Ansatzes prüfen zu können, ist für den SMR-Daten-Transfer zur GPU eine Analyse der erforderlichen Datentransferrate notwendig. Die zu übertragende Datenmenge ist abhängig vom erforderlichen Erfassungsbereich, von der Entfernungsauflösung des Radars, der Farbtiefe und der Aktualisierungsrate des Bildes. Laut Anforderung R6 (Abschnitt 2.4) soll der Prototyp einen Erfassungsbereich 4500 [m] $\times$ 4500 [m] bei einer Entfernungsauflösung von 1.8 [m] unterstützen.

Für das ganze Bodenradarbild ohne sektorweise Aktualisierung ergibt sich bei einer Farbtiefe von 32-Bit und einer Umlaufzeit von 1 Sekunde eine Datenrate von:

$$(4500/1.8)^2 \cdot 4 \approx 24 \text{ [MiB/s]}$$

Unter Berücksichtung einer sektorweisen Aktualisierung mit 8 Sektor-Updates pro Sekunde und einer Verwendung einer einzigen Bildkachel mit der Größe des Bodenradarbildes (Worst-Case-Szenario) ergibt sich die Datenrate zu:

$$(4500/1.8)^2 \cdot 4 \cdot 8 \approx 191 \text{ [MiB/s]}$$

Die tatsächlich zu erwartende Datenmenge hängt indes von der Anzahl und Größe der Bildkacheln, die einen Sektor mehr oder weniger gut approximieren, ab und liegt somit zwischen dem Minimalwert von rund 24 [MiB/s] und dem Maximalwert von rund 191 [MiB/s]. Darüber hinaus kann durch Reduktion der Farbtiefe die Datenmenge auf ein Viertel reduziert werden.

Maßgeblich für die Betrachtung der maximal erzielbaren Datentransferrate ist die Anbindung der GPU an den PC. Es lässt sich dabei die Unterscheidung zwischen diskreten und integrierten GPUs treffen [19, S. 393]. Diskrete GPUs sind üblicherweise über PCIe mit dem PC verbunden. Integrierte GPUs sind im Gehäuse der CPU untergebracht (*on die*) und können dadurch direkt auf den Hauptspeicher zugreifen. Nach Hrabcak und Massernan ist daher der Zugriff auf Daten im Hauptspeicher schneller als der Zugriff über PCIe, jedoch haben diskrete Grafikkarten einen schnelleren On-Board-Speicher [19, S. 393].

Die Grafikkarte im Referenzsystem (siehe Anforderung R9, Abschnitt 2.4) ist eine diskrete Grafikkarte (Nvidia Quadro P400) und ist über PCIe 3.0 x16 angebunden. Die maximale Datentransferrate pro Richtung beträgt demnach 15.75 [GB/s], jedoch ist die im Sinne von Nutzdaten transferierbare Datenmenge mitunter abhängig von der Form des Datenverkehrs (e.g. viele kleine Pakte versus ein großes Paket) und der zusätzlich zu übertragenden Daten (Protokoll-Overhead) [49].

Die pro Sekunde zu übertragende Datenmenge wird im Vergleich zur maximal erzielbaren Datentransferrate als hinreichend klein beachtet. Dies ermöglicht es, nach dem Ansatz zu verfahren, die Bildtransformation auf der Grafikkarte durchzuführen.

Im Vergleich dazu ist die erforderliche Datentransferrate für den bisherigen Ansatz (CPU-seitige Bildtransformation) abhängig von der Fenstergröße und der Anzahl der Fenster,

jedoch unabhängig von der Größe des Erfassungsbereiches. Der bisherige Ansatz benötigt, bei gleicher Farbtiefe von 32-Bit und einer höheren Pixelanzahl des SMR-Bildes, verglichen mit der Pixelanzahl aller Fenster, weniger Bandbreite. Die Bildtransformation und die Neuberechnung der Lookup-Tabelle (z. B. beim Zoomen) beanspruchen jedoch zusätzliche CPU-Zeit. In dieser Arbeit wird die GPU-seitige Bildtransformation implementiert. Die zur Übertragung und zum Rendern der Daten verwendeten Methoden lassen sich für beide Ansätzen verwenden. Die CPU-seitige Bildtransformation kann, sofern sich neue Anforderungen ergeben, zu einem späteren Zeitpunkt als alternativer Zweig implementiert werden.

4.4.2 Implementierung

Die Implementierung der Stufe 1 für die Verarbeitung der SMR-Daten wird vom bestehenden HMI übernommen. Das Modul SMR-Image-Composer wird, wie in weiterer Folge noch beschrieben, modifiziert, um die zu übertragende Datenmenge verringern zu können. Es werden keine weiteren Änderungen vorgenommen, da die Verarbeitungszeit dieser Stufe (wie bereits erwähnt) keinen Einfluss auf die Render-Zeit besitzt und es seitens ADB Safegate Austria keinen Anlass zur Änderung des zugrundeliegenden Konzepts gibt. Die weiteren Ausführungen befassen sich mit der Umsetzung von Stufe 2 mit Qt Quick.

Fasst man die Erkenntnisse aus dem vorigen Abschnitt zusammen, so muss das Design folgende Punkte adressieren:

1. Die pro Fenster durchzuführende Bildtransformation soll von der Grafikkarte ausgeführt werden.
2. Die SMR-Daten sollen pro Sektoraktualisierung, unabhängig von der Anzahl der Fenster, nur einmal zur Grafikkarte übertragen werden.
3. Der Transfer der SMR-Daten zur Grafikkarte soll nach Möglichkeit nicht im Render-Thread stattfinden.

4.4.2.1 SMR-Bild als Textur

Die Darstellung des SMR-Bildes, einschließlich der dafür notwendigen, ansichtsspezifischen Bildtransformation, lässt sich durch Texture-Mapping [50, S. 303] realisieren: Die bereits als Bild vorliegenden SMR-Daten stehen als OpenGL-Textur zur Verfügung und werden als texturiertes Rechteck dargestellt. Die Lage und Größe des Rechtecks ergibt sich aus der horizontalen und vertikalen Ausdehnung des Erfassungsbereichs. Der Ansichtstransformation wird im Vertex-Shader Rechnung getragen.

Der durch diesen Ansatz notwendige, fensterübergreifende Zugriff auf die Textur stellt bei der Verwendung des unter Linux standardmäßig verwendeten Multi-Threaded-Variante des Qt Quick-Szenengraph-Renderer ein technisches Problem dar. Jedem Qt Quick-Fenster wird dabei ein eigener OpenGL-Kontext und Render-Thread zugewiesen. Die dem OpenGL-Kontext eines Qt Quick-Fensters zugehörigen OpenGL-Ressourcen sind jedoch für andere Qt Quick-Fenster nicht verfügbar. Allerdings lassen sich OpenGL-Kontexte mithilfe von plattformspezifischen OpenGL-Funktionen verbinden, um OpenGL-Ressourcen zu teilen [19, S. 405]. Mit `QOpenGLContext::setShareContext()` bietet Qt 5 hierfür bereits eine plattformübergreifende C++ API an.

4.4.2.2 Asynchroner Texturdatentransfer

Um den Datentransfer möglichst parallel zum Rendervorgang durchführen zu können, ist ein zusätzlicher Thread zum Aktualisieren der Textur notwendig, welcher in weiterer Folge als Feeder-Thread bezeichnet wird. Die Verwendung von mehreren Threads erfordert jedoch einen synchronisierten Lese- und Schreibzugriff. Der Einsatz von z. B. Mutex-Objekten zur Thread-Synchronisation ist nicht ausreichend, da von einer asynchronen Ausführung der OpenGL-Befehle auszugehen ist [48, S. 424].

Dieses Problem lässt sich durch den Einsatz von Synchronisationsobjekten, welche ab OpenGL Version 3.2 verfügbar sind, lösen. Für die Synchronisierung des Texturzugriffs bieten sich Synchronisationsobjekte vom Typ Fence an. Das Fence-Objekt verfügt über einen Zustand, der entweder *unsignaled* (Ausgangszustand) oder *signaled* sein kann. Der finale Zustand *signaled* wird gesetzt, wenn das bei der Erstellung des Fence-Objekts spezifizierte Ereignis eingetreten ist. Als Ereignis lässt sich z. B. die vollständige Abarbeitung aller GPU-Befehle definieren, die vor dem Setzen des Fence-Objekts abgesetzt wurden. Ein Fence ist somit vergleichbar mit einem GPU-Befehl im GPU-Befehlspuffer, dessen Abarbeitung sich auf Applikationsebene abfragen lässt (vgl. Wright et al. [42, S. 494]).

OpenGL stellt die Funktionen `glClientWaitSync()` bzw. `glWaitSync()` zur Verfügung, um den Zustand eines Synchronisationsobjektes abzufragen oder um auf den Ereigniseintritt zu warten [42, S. 495 ff.]. Die Funktion `glClientWaitSync()` lässt sich dazu verwenden, die Programmausführung im laufenden Thread bis zum Ereigniseintritt zu blockieren. Hingegen sorgt der Aufruf der Funktion `glWaitSync()` dafür, dass der OpenGL-Treiber bis zum Ereigniseintritt keine weiteren OpenGL-Befehle in den GPU-Befehlspuffer einsetzt. Die Programmausführung wird dadurch nicht blockiert und es können weitere OpenGL-Befehle abgesetzt werden, die jedoch erst nach dem Erreichen des Synchronisationspunktes verarbeitet werden.

Die Synchronisationsmechanismen ermöglichen zwar den sicheren Zugriff auf die Textur, allerdings können dadurch Wartezeiten im Feeder-Thread entstehen, wenn das Rendern der Textur mit dem Zeitpunkt ihrer Aktualisierung zusammenfällt. Der Feeder-Thread muss in diesem Fall warten bis die Textur nicht zum Rendern verwendet wird.

In Anlehnung an Venkataraman [48, S. 423], werden statt einer Textur zwei Texturen eingesetzt, die vom Feeder-Thread und den Render-Threads alternierend verwendet werden (Round-Robin-Verfahren). Das Prinzip wird in Abbildung 4.6 dargestellt. Durch die Verwendung ein zusätzlicher Texturen lässt sich der Vorgang zum Aktualisieren der Texturdaten und der Rendervorgang weiter parallelisieren. Die Umschaltung zwischen den Texturen lässt sich ereignisgesteuert – in der Abbildung durch strichlierte Pfeile angedeutet – implementieren und fügt sich dadurch sehr gut in das Qt-Konzept, der asynchronen Thread-Kommunikation via *Signal/Slot*-Verbindungen, ein.

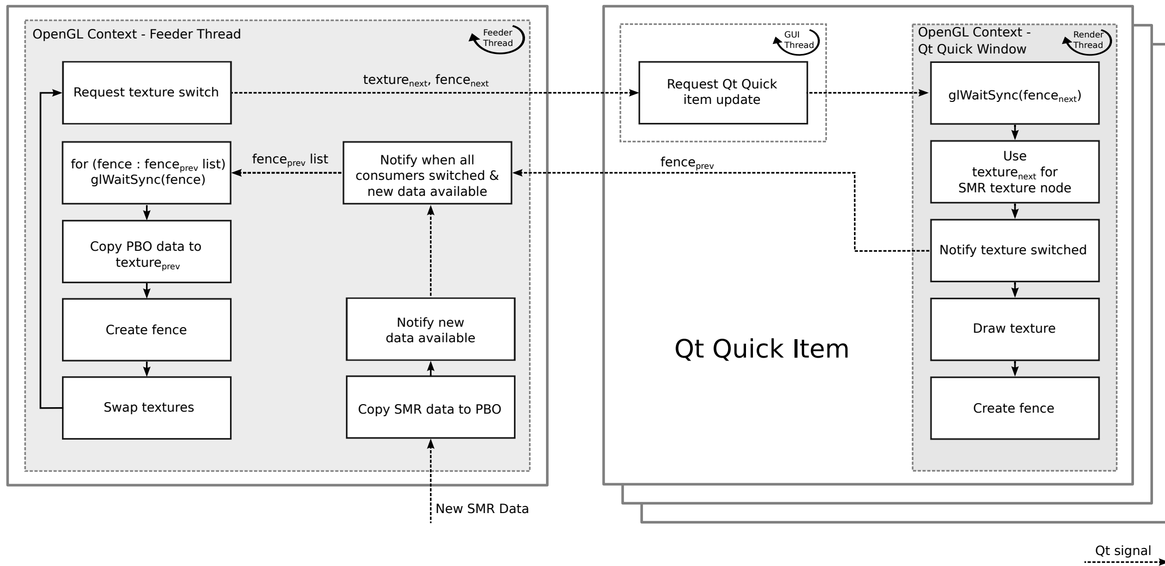


Abbildung 4.6: Asynchroner SMR-Texturdatentransfer im Round-Robin-Verfahren

4.4.2.3 Reduktion der Datenmenge

Wie in Abschnitt 4.4.1 aufgezeigt, kann die Farbtiefe des am Ausgang von Stufe 1 zur Verfügung stehenden Bildpuffers von 32-Bit auf 8-Bit reduziert werden, indem der Afterglow-Effekt auf der GPU generiert wird (siehe auch 3). In dieser Arbeit wird ein erster Schritt in diese Richtung gemacht. Der Afterglow-Effekt wird weiterhin CPU-seitig generiert, die Farbtiefe wird jedoch von 4 auf 2 Farbkanäle, zu jeweils 8-Bit, reduziert. Der erste Farbkanal gibt die Radarechointensität an, der zweite Farbkanal kodiert das Alter des Radarechos und dient somit zur Unterscheidung zwischen dem aktuellen Radarecho und dem Afterglow. Das Radarechoalter wird durch die Werte 0 (Afterglow) und 255 (aktueller Wert) kodiert.

Die derart kodierten Radardaten werden in einer OpenGL-Textur, bestehend aus den zwei 8-Bit-Farbkanälen Rot und Grün (OpenGL-Texturformat `GL_RG8`), gespeichert. Aus den Texturdaten wird mit Hilfe eines Fragment-Shader-Programms das SMR-Bild erzeugt. Ein beispielhaftes Fragment-Shader-Programm ist in Abbildung 4.7 ersichtlich. Die Farben für das aktuelle Radarecho (`smrColor`) und das Afterglow (`smrHistoryColor`) werden dem Shader als Uniform-Variablen zur Verfügung gestellt. Das Programm bestimmt für die vorliegenden Position zunächst mittels Texture-Sampling den RG-Wert des betroffenen Texels (`smrTextureSample`). Dabei gibt die R-Komponente die Intensität und die G-Komponente das Alter des Radarechos an.

Anschließend wird der RGBA-Wert des Fragments bestimmt. Die Radarechointensität legt den Alpha(A)-Wert des Fragments fest (Visualisierung mittels Transparenzeffekt, siehe Abschnitt 2.3.3.1). Der RGB-Wert des Fragments wird mittels linearer Interpolation zwischen den Farben `smrColor` und `smrHistoryColor` bestimmt. Die Gewichtung erfolgt entsprechend dem Alter des Radarechos, dessen Wert nach erfolgter Normierung und Interpolation (Texture-Filtering) im Intervall $[0, 1]$ liegt.

Für die Vergrößerung und Verkleinerung des SMR-Bildes wird, wie auch im bestehenden ACEMAX-HMI, standardmäßig die Nearest-Neighbour-Interpolation verwendet. Der vorgestellte Ansatz ermöglicht jedoch auch die Verwendung von bilinearer Interpolation. Das Texture-Filtering-Verfahren lässt sich durch OpenGL zudem für das Verkleinern und Vergrößern des SMR-Bildes unterschiedlich festlegen.

```
void main()
{
    vec4 smrTextureSample = texture2D(texture, texCoord);
    // calculate fragment color by mixing history and normal
    // color. The factor is given by the g-component
    // of the texure, which is either 0 or 1 in the source
    // image. When using bilinear filtering, the sampled value
    // is the interval [0, 1].
    vec3 color_rgb = mix(smrHistoryColor.rgb, smrColor.rgb,
                         smrTextureSample.g);
    // use radar intensity stored in the r-component of the
    // texture as alpha, use pre-multiplied alpha
    color = vec4(color_rgb * smrTextureSample.r,
                 smrTextureSample.r);
};
```

Abbildung 4.7: Fragment-Shader-Programm für die Erzeugung des SMR-Bildes

4.4.2.4 Anwendung

Das SMR-Bild wird durch das Qt Quick-Item `SMRImage` (C++ Klasse `QuickSMRImage`) dargestellt. Der Einsatz dieses Items als Layer ist in Abbildung 4.8 ersichtlich. Die von einer SMR-Datenquelle bereitgestellten SMR-Daten werden dem Item über eine Datenschnittstelle (C++ Schnittstellenklasse `QuickSMRImageTextureStream`) als Textur zur Verfügung gestellt. Die Auswahl der SMR-Datenquelle erfolgt in QML durch das Setzten des Property `SMRImage.source` und ist im HMI-Prototypen auf einzige SMR-Datenquelle beschränkt. Ein und dieselbe SMR-Datenquelle lässt sich von mehreren `QuickSMRImage`-Instanzen verwenden. Die Ansichtstransformation wird in diesem Fall vom Layer an `SMRImage` "vererbt" (siehe Abschnitt 4.2.2).

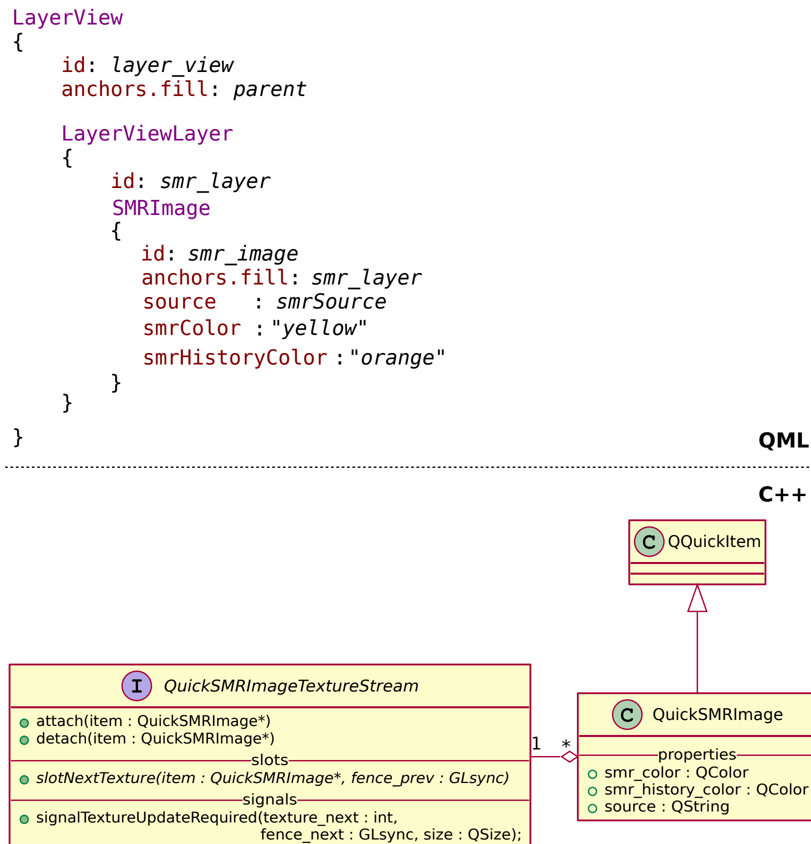


Abbildung 4.8: Anwendung des Qt Quick-Items `SMRImage`: Klassendiagramm (unten) und beispielhafter QML-Code (oben)

4.5 Anzeige der Ziele

Die Analyse in Abschnitt 2.5 zeigt, dass die Track-Darstellung einen erheblichen Anteil an der CPU-Auslastung des GUI-Threads des ACEMAX-HMI besitzt und dadurch die Reaktivität des HMI maßgeblich beeinflusst. Die CPU-Auslastung steigt mit zunehmender Track-Anzahl nahezu linear an. Die Entlastung des GUI-Threads, durch Verlagerung des Zeichenaufwands auf die Grafikkarte, ist somit ein Designziel für die Track-Darstellung.

Die synthetische Darstellung eines Zieles setzt sich, wie in Abschnitt 2.3.3.2, aus folgenden Komponenten zusammen: Label mit Leader-Line, Track-Symbol, Speed-Vector und Track-History. Das Label, welches aus mehreren Textfeldern besteht und die Track-History, welche die Darstellung mehrerer Symbole umfasst, sind dabei die am aufwändigsten zu zeichnenden Komponenten.

4.5.1 Auswahl des Lösungsansatzes

Qt Quick eröffnet, wie auch beim Rendern der Flughafenkarte und des Radarbildes, verschiedene Lösungsansätze zur Darstellung der Ziele. Die identifizierten Lösungsansätze werden nachfolgend zusammengefasst und abschließend bewertet.

4.5.1.1 Ansatz "Per Track"

Der Ansatz ein Ziel als Qt Quick-Item zu modellieren, drängt sich aufgrund der bereits mit Qt Quick verfügbaren Funktionalität auf. Das Label eines Zieles kann durch Einsatz von Qt Quick-Basistypen realisieren werden, z. B. durch die Verwendung der QML Basistypen `Text`, `GridLayout` und `Rectangle`. Funktionalität, die sich aus der Interaktion des Anwenders mit dem Label ergibt (e.g. Verschieben des Labels), lässt sich mit dem Basistyp `MouseArea` umsetzen.

Das Track-Symbol und die Track-History-Symbole können als Textur, z. B. mithilfe des QML Basistyps `Image`, abgebildet werden. Qt Quick bietet in Qt 5.9.1 kein dediziertes Qt Quick-Item für die Darstellung von Linien an. Für den Speed-Vector und die Leader-Line wäre daher ein solches Item zu implementieren.

Die Instanziierung der Qt Quick-Items kann wahlweise in C++ oder in Qt Quick, z. B. durch Einsatz des QML Basistyps `Repeater`, erfolgen. Mittels Property-Binding können die Qt Quick-Items mit den Track-Daten aus dem C++ Backend verknüpft werden.

4.5.1.2 Ansatz "Gesamte Track-Szene"

Anstatt jeden Track einzeln durch Qt Quick-Items zu modellieren, kann die gesamte Track-Szene auch durch ein einziges, eigens implementiertes Qt Quick-Item dargestellt werden. Dabei ist das Item selbst für die Erstellung und Aktualisierung der Qt-Quick-Szenengraph-Nodes verantwortlich, wodurch die Verwendung von Property-Bindings vermieden werden kann. Qt Quick lässt jedoch in Qt 5.9.1 einen Szenengraph-Node für die Anzeige von Text vermissen. Ein solcher Szenengraph-Node müsste für diesen Lösungsansatz implementiert werden.

4.5.1.3 Ansatz "Gesamte Track-Szene via OpenGL"

In Qt 5.9.1 ist Qt Quick auf Kompatibilität mit OpenGL 2.0 ausgelegt. Der Renderingalgorithmus des Qt Quick-Szenengraph-Renderer lässt sich nicht individuell anpassen und somit nicht für das Problem optimieren. Die direkte Verwendung von OpenGL hätte daher den Vorteil, einen auf das Problem optimierten Renderingalgorithmus zu konzipieren und die in neueren OpenGL Versionen zur Verfügung stehenden Funktionalitätserweiterungen auszunützen. So bietet sich beispielweise Instancing [46], Core-Funktionalität ab OpenGL 3.1, für die Darstellung der Track-Symbole und Track-History-Symbole an.

Der Qt Quick-Szenengraph-Renderer stellt einen Einsprungpunkt zur Verfügung, um nach dem Rendern der Qt Quick-Szene, eigenen OpenGL Code auszuführen. In Bezug auf Erweiterbarkeit ist jedoch ein Ansatz zu bevorzugen, der die Darstellung weiterer Qt Quick-Items

oberhalb der Zieldarstellung zulässt. Qt Quick sieht dafür mit `QQuickFramebufferObject` und `QSGRenderNode` zwei Möglichkeiten vor.

Die C++ Klasse `QQuickFramebufferObject` von Qt macht es möglich, Qt Quick-Items mit OpenGL zu rendern. Gezeichnet wird zuerst in ein OpenGL FBO. In einem zweiten Schritt wird die mit dem OpenGL FBO verbundene Textur dargestellt. Dieses zweistufige Verfahren stellt im Vergleich zum direkten Zeichnen in den Default-Framebuffer einen Zusatzaufwand dar. Gezeichnete und sich weniger häufig aktualisierende Inhalte lassen sich dadurch jedoch auch cachen. Mit dem Qt Quick-Szenengraph-Node `QSGRenderNode` kann im Gegensatz zu `QQuickFramebufferObject` direkt in den Default-Framebuffer gezeichnet werden. Die Verwendung des OpenGL Depth-Buffers ist jedoch eingeschränkt, Schreibzugriffe sind laut Qt-Dokumentation [7] zu vermeiden.

4.5.1.4 Bewertung der Lösungsansätze

Die Eigenschaften von Tracks können sich teils sekündlich ändern, wie beispielsweise die Position (Track-Symbol und Track-History) und die Geschwindigkeit (Speed-Vector). Der Aufwand für das Data-Binding zwischen QML und C++ ist daher ein nicht zu vernachlässigender Faktor und steigt mit der Anzahl an verwendeten Qt Quick-Items. Es ist daher zu erwarten, dass sich der intuitive Ansatz “Per Track” im Vergleich zum Ansatz “Gesamte Track-Szene”, bei dem alle Tracks durch ein einziges Qt Quick-Item dargestellt werden, als ineffizienter herausstellt. Hingegen erfordert der Ansatz “Per Track” für die Label-Interaktion und für das Rendering und die Anordnung der Label-Textfelder weniger Implementierungsaufwand.

Der Ansatz “Gesamte Track-Szene via OpenGL” bietet, aufgrund des direkten Einsatzes von OpenGL, das größte Optimierungspotential. Obgleich mit der Verwendung von `QSGRenderNode` und `QQuickFramebufferObject` auch Einschränkungen verbunden sind. Ein auf einer höheren Abstraktionsebene basierender Ansatz wird jedoch als hinreichend peformant gesehen und in dieser Arbeit der Vorzug gegeben.

Die Vorzüge von Ansatz “Per Track” und “Gesamte Track-Szene” lassen sich kombinieren. So lässt sich das Label im Ansatz “Gesamte Track-Szene” mit den QML Basistypen implementieren. Der Implementierungsaufwand kann dadurch reduziert werden. Der Aufwand für das Data-Binding kann in Kauf genommen werden, da sich die meisten Textfelder im Label nicht sekündlich aktualisieren. Dieser kombinierte Ansatz wird in dieser Arbeit implementiert.

4.5.2 Implementierung

Für die Beschreibung der Track-Szene im Szenengraph bietet sich an, jeden Track einzeln abzubilden, um eine vertikale Staffelung der Ziele wie im ACEMAX-HMI zu erhalten. Die Zerlegung der Zieldarstellung in seine Komponenten führt infolgedessen zu mehreren Szenengraph-Nodes pro Track. Die Aufgabe, die Vielzahl an Szenengraph-Nodes effizient mit OpenGL zu rendern, obliegt dabei dem Qt Quick-Szenengraph-Renderer. Durch dessen Batching-Mechanismus werden Szenengraph-Nodes mit gleichartigen Geometrietyp und Materialien gebündelt, um OpenGL Zustandsänderungen zu minimieren, die Geometriedaten gesammelt zu übertragen und zu zeichnen.

Der dem Batching-Mechanismus des Qt Quick-Szenengraph-Renderer zugrunde liegende

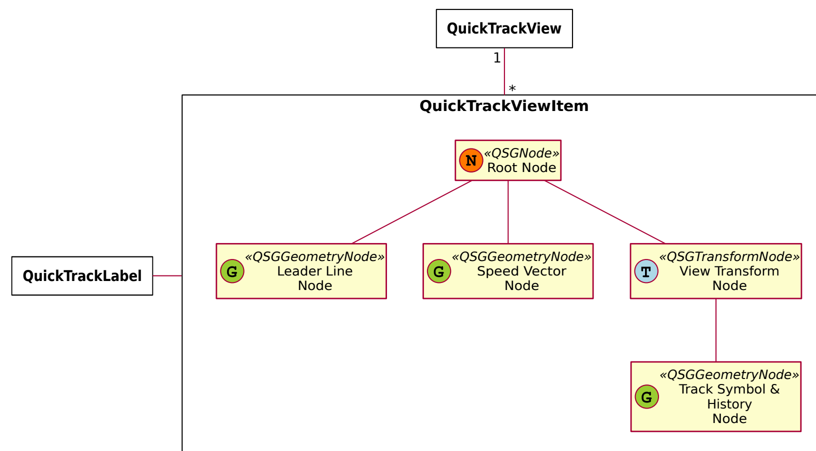
Optimierungsansatz kann bereits bei der Beschreibung der Ziele im Szenengraph angewendet werden, um den Rendervorgang CPU-seitig effizienter zu gestalten. Jedoch ist die vom Qt Quick-Szenengraph-Renderer eingesetzte OpenGL Blending-Funktion `GL_ONE`, `GL_ONE_MINUS_SRC_ALPHA` nicht kommutativ. Die Zeichenreichenfolge von nicht opaken Komponenten der Zieldarstellung ist daher entscheidend. Techniken zur Order-Independent-Transparency [28, S. 895] würden den direkten Einsatz von OpenGL erfordern. Qt adressiert dieses Problem, indem der Batching-Mechanismus nur dann angewendet wird, wenn es keine geometrische Überlappung gibt (vgl. Qt-Dokumentation "Qt Quick Scene Graph OpenGL Renderer" [7]). Als nicht opak einzustufen sind einerseits der Label-Text aufgrund von Antialiasing und andererseits die Track-History-Symbole aufgrund des mittels Transparenz erzeugten Afterglow-Effekts.

Um auch im Falle von geometrischer Überlappung der Ziele die Anzahl der OpenGL Zeichenbefehle und Zustandsänderungen zu minimieren, wird ein alternativer Ansatz zur vertikalen Staffelung gewählt. Die Zieldarstellung teilt sich bei diesem Ansatz komponentenweise auf mehrere Darstellungsebenen auf. Eine Darstellungsebene zeigt dabei die ihr zugeordneten Darstellungskomponenten von allen Zielen an. Die Zeichenreihenfolge der Ziele muss folglich für alle Darstellungsebenen gleich sein, um eine kohärente Anzeige zu gewährleisten.

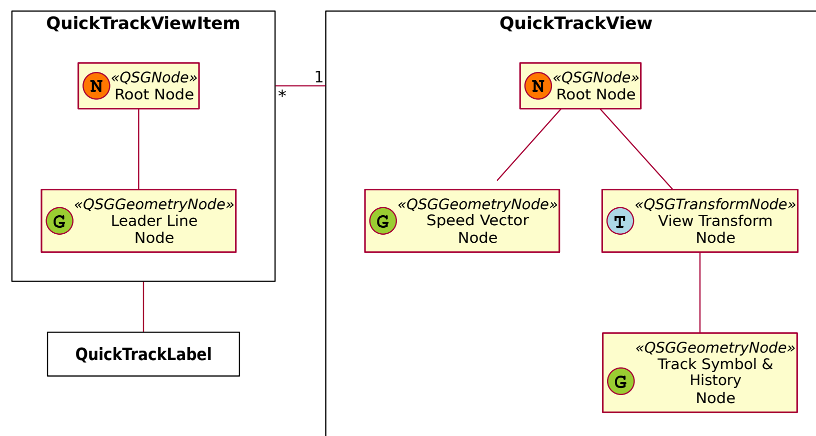
Für die Umsetzung dieses Ansatzes werden folgende Darstellungsebenen verwendet:

- Darstellungsebene 0: Track-Symbole und Track-Histories
- Darstellungsebene 1: Speed-Vectors
- Darstellungsebene 2: Labels mit Leader-Lines

Die Darstellungsebenen 0 und 1 lassen sich dadurch jeweils mit einem einzigen Szenengraph-Node beschreiben, indem die Geometriedaten in einem Array zusammengefasst werden. Die Beschreibung der Darstellungsebene 2 im Szenengraph muss weiterhin pro Track erfolgen, da das Label in QML realisiert wird. Der Unterschied zur eingangs beschriebenen Einzelabbildung von Tracks wird in Abbildung 4.9 veranschaulicht. Ansatz (a) realisiert hierin die Zeichenreihenfolge im ACEMAX-HMI, während Ansatz (b) der Implementierung im HMI-Prototypen entspricht. Hinsichtlich der Zeichenreihenfolge wird beispielsweise durch Ansatz (b), im Gegensatz zur Darstellung im ACEMAX-HMI, das Label eines zuerst gezeichneten Zieles nicht vom Track-History-Symbol eines anderen Zieles verdeckt.



(a) Beschreibung der Zieldarstellungskomponenten pro Track



(b) Beschreibung der Zieldarstellungskomponenten mittels komponentenweiser Aufgliederung in 3 Darstellungsebenen

Abbildung 4.9: Gegenüberstellung von Ansatz (a) und (b) zur Beschreibung der Ziele im Qt Quick-Szenengraphen

4.5.2.1 Track-Symbole/History

Die Track-Symbole und die Track-History-Symbole werden als texturierte Vierecke dargestellt. Die Vierecks- und Texturkoordinaten können anwendungsseitig berechnen werden, OpenGL bietet hierfür jedoch effizientere Methoden an:

- Instancing [46]: die Geometrie des Vierecks muss anwendungsseitig nur einmal beschrieben werden und kann wiederholt an allen Symbolpositionen verwendet werden.
- Geometry-Shader: die Vierecks- und Texturkoordinaten lassen sich durch einen Geometry-Shader aus den Symbolpositionen erzeugen.
- Point-Sprites [28, S. 519 ff.]: die Symbole können als texturierte, quadratische Punkte (OpenGL-Primitiv `GL_POINT`) gezeichnet werden, die Punktgröße lässt sich im Vertex-Shader definieren.

Für die Implementierung wird einfachheitshalber auf Point-Sprites gesetzt. Die Implementierung eines eigenen Szenengraph-Nodes, als Subklasse von `QSGRenderNode`, ist dadurch nicht erforderlich.

Die Track-Symbole und Track-History-Symbole von allen Zielen werden durch einen einzigen Szenengraph-Node vom Basistyp `QSGGeometryNode` beschrieben. Neben der Symbolposition wird auch die Art des Zieles (e.g. landendes vs. abfliegendes Flugzeug) in den Vertex-Daten spezifiziert, um unterschiedliche Farben, Symbolgrößen und -arten zu unterstützen. Zur Unterscheidung zwischen der aktuellen Position und den N vorhergehenden Positionen der Track-History ist noch ein weiteres Vertex-Attribut notwendig. Pro Symbol werden somit folgende Vertex-Attribute übertragen:

- x-Koordinate des Symbols
- y-Koordinate des Symbols
- Track-Typ
- Positionsindex i: 0 = aktuelle Position, >0 = i-te Track-History-Position

Die Texturen für die unterschiedlichen Symbole werden in einem OpenGL 2D-Texture-Array vorgehalten. Die Bestimmung des Symbols anhand der in den Vertex-Daten abgelegten Informationen obliegt dem Vertex-Shader. Dazu wird eine Lookup-Tabelle verwendet, welche für jeden Track-Typ Informationen für das zu verwendende Symbol bereitstellt. Die Lookup-Tabelle wird als QpenGL Uniform-Buffer-Object (UBO) dem Vertex-Shader zugänglich gemacht. Folgende Symbolinformationen werden in der Lookup-Tabelle vorgehalten:

- Symbolfarbe
- 2D-Texture-Array-Index für das Track-Symbol
- Größe des Track-Symbols
- 2D-Texture-Array-Index für das Track-History-Symbol
- Größe des Track-History-Symbols

4.5.2.2 Speed-Vector

Zur Abbildung der Speed-Vectors aller Ziele wird ein Szenengraph-Node vom Basistyp `QSGGeometryNode` mit Geometrietyp `GL_LINES` verwendet. Der Startpunkt eines Speed-Vector entspricht der Track-Position. Der Endpunkt wird auf Basis der konfigurierten Prädiktionszeit CPU-seitig berechnet. Die Start- und Endpunkte der Speed-Vectors werden dem Szenengraph-Node unter Berücksichtigung der Zeichenreihenfolge als Array übergeben.

4.5.2.3 Label mit Leader-Line

Das pro Ziel instantiierte Qt Quick-Item `TrackViewItem` stellt das Label und die Leader-Line dar. Die Leader-Line wird durch den Basistyp `QSGGeometry` im Szenengraph abgebildet. Das in QML implementierte Label wird hingegen als Qt Quick-Component C++-seitig geladen. Das `TrackViewItem` hält eine Referenz auf ein Track-Datenobjekt, welches die Track-Daten in Form von Qt Properties zur Verfügung stellt. Mithilfe von Property-Bindings lassen sich Label-Textfelder mit dem Track-Datenobjekt verknüpfen.

4.5.2.4 Anwendung

Die Ziele werden durch das Qt Quick-Item `TrackView` (C++ Klasse `QuickTrackView`) dargestellt (Abbildung 4.10 illustriert dessen Anwendung als Layer). Die über das proprietäre Messaging-Protokoll empfangenen Track-Daten werden durch die C++ Klasse `TrackDataManager` vorgehalten. Für jeden Track existiert eine Instanz der Klasse `TrackData`. Das `TrackDataManager`-Objekt wird als QML-Context-Property dem `TrackView`-Item zugänglich gemacht.

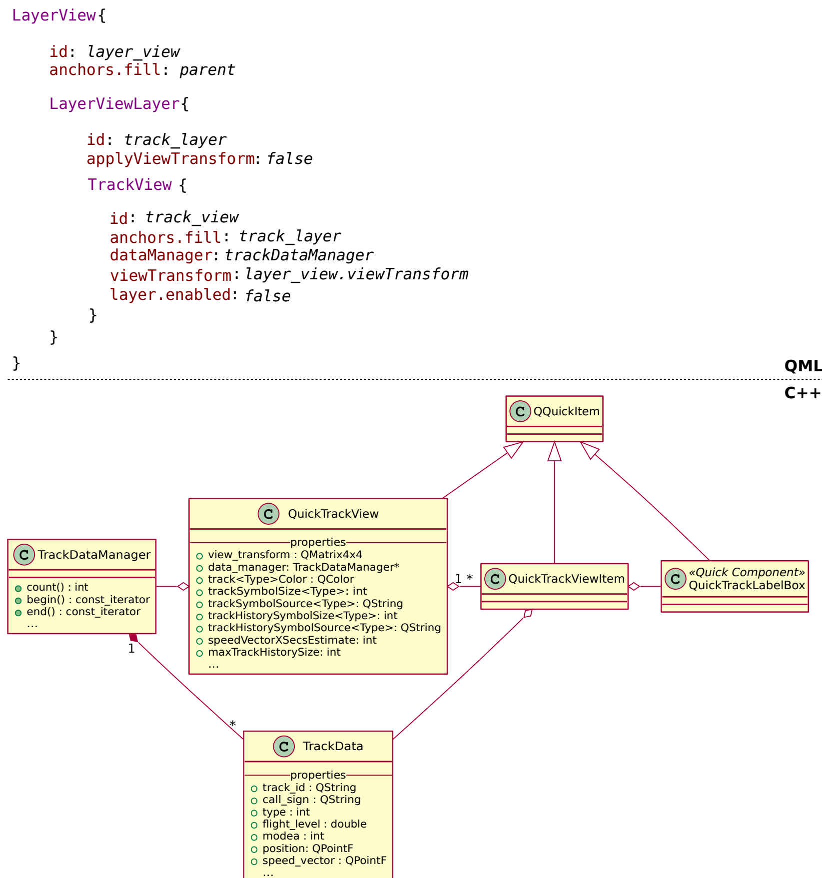


Abbildung 4.10: Anwendung des Qt Quick-Items `TrackView`: Klassendiagramm (unten) und beispielhafter QML-Code (oben)

5 Resultate

Zur Leistungsbewertung des HMI-Protypen wird die in Abbildung 5.1 ersichtliche Testumgebung verwendet. Sie besteht aus zwei PCs, einem Server- und einem Client-PC, welche zeitsynchronisiert und über ein Gigabit-LAN (1-Gbit/s-Ethernet) miteinander verbunden sind. Der mit simulierten Track- und SMR-Video-Daten gespeiste ACEMAX-Server wird am Server-PC betrieben. Der Client-PC dient zur Ausführung des HMI Prototypen und entspricht dem in Anforderung R9 (siehe 2.4.3) definierten Referenzsystem. Die Track- und SMR-Video-Daten werden über TCP/IP zwischen dem ACEMAX-Server und dem HMI-Prototypen übertragen.

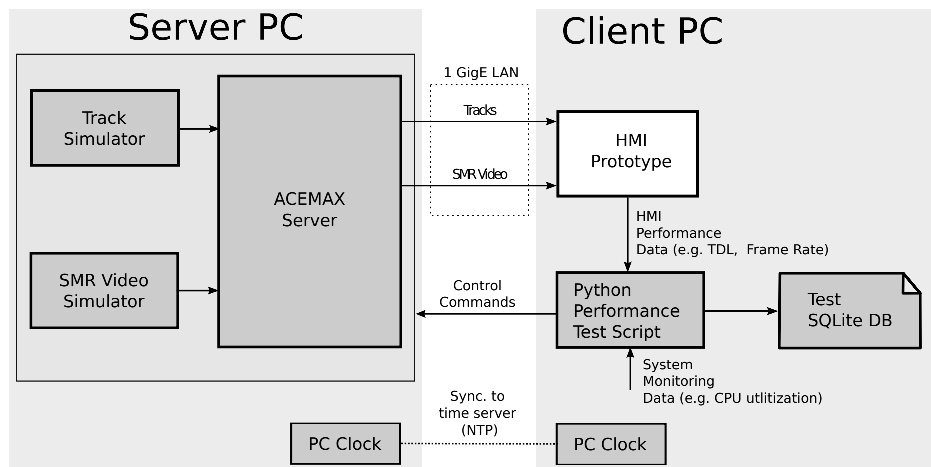


Abbildung 5.1: Testaufbau zur Performance-Evaluierung des HMI-Prototypen

Die Evaluierung basiert auf den unter Anforderung R12 (siehe 2.4.3) definierten Testbindungen und wird zusätzlich zum Basisszenario mit 500 Tracks pro Fenster auch für 750 und 1000 Tracks pro Fenster durchgeführt. Zur Leistungsbewertung werden folgende Parameter gemessen:

- CPU-Last
- Target display latency (TDL) nach ED-87C [45, S. 21]
- Information display latency (IDL) nach ED-87C [45, S. 29]
- Response Time to Operator Input (RTOI) nach ED-87C [45, S. 29]
- Framerate

Ein in dieser Masterarbeit entwickeltes Python-Testscript steuert den Testablauf und die Simulation des Testszenarios. Es startet den HMI-Prototypen und zeichnet die oben genannten Performance-Parameter in einer SQLite-Datenbank(-DB) auf. Das CPU-Last des

HMI-Prototypen wird mittels dem Python-Modul `psutil` ermittelt, die restlichen im HMI-Prototypen gemessenen Parameter, wie z. B. TDL und Framerate, werden in einer CSV-Datei zwischengespeichert und abschließend vom Python-Testscript in die SQLite-DB transferiert. Mit diesem Python-Testscript lässt sich ein Testlauf automatisiert wiederholen und für unterschiedliche Anzahlen von Tracks ausführen. Die aufgezeichneten Daten werden mittels Jupyter Notebook [27], eine Open-Source-Webapplikation für interaktive Datenanalysen, ausgewertet.

Das Hauptfenster und die fünf Zoom-Fenster des HMI-Prototypen sind – für einen Testlauf mit 1000 Tracks – in Abbildung 5.2 und 5.3 zu sehen. Neben der Darstellung der Flughafenkarte, der Ziele und des Bodenradarbildes, ist im Screenshot des Hauptfensters die horizontale Toolbar, die einklappbare Sidebar und ein vom Benutzer selektiertes Ziel in der oberen rechten Fensterecke erkennbar. In jedem Fenster wird außerdem die Framerate (Zahl in gelber Schriftfarbe) am rechten oberen Fensterrand angezeigt.

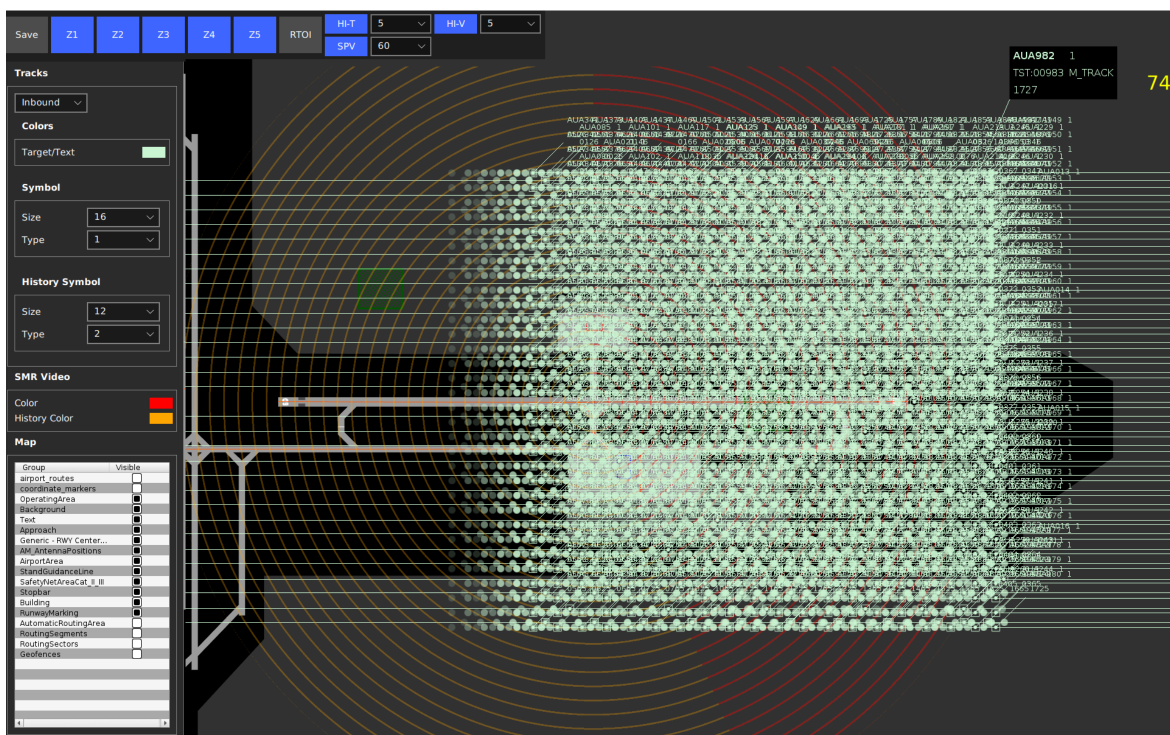


Abbildung 5.2: Das Haupfenster des HMI-Prototypen

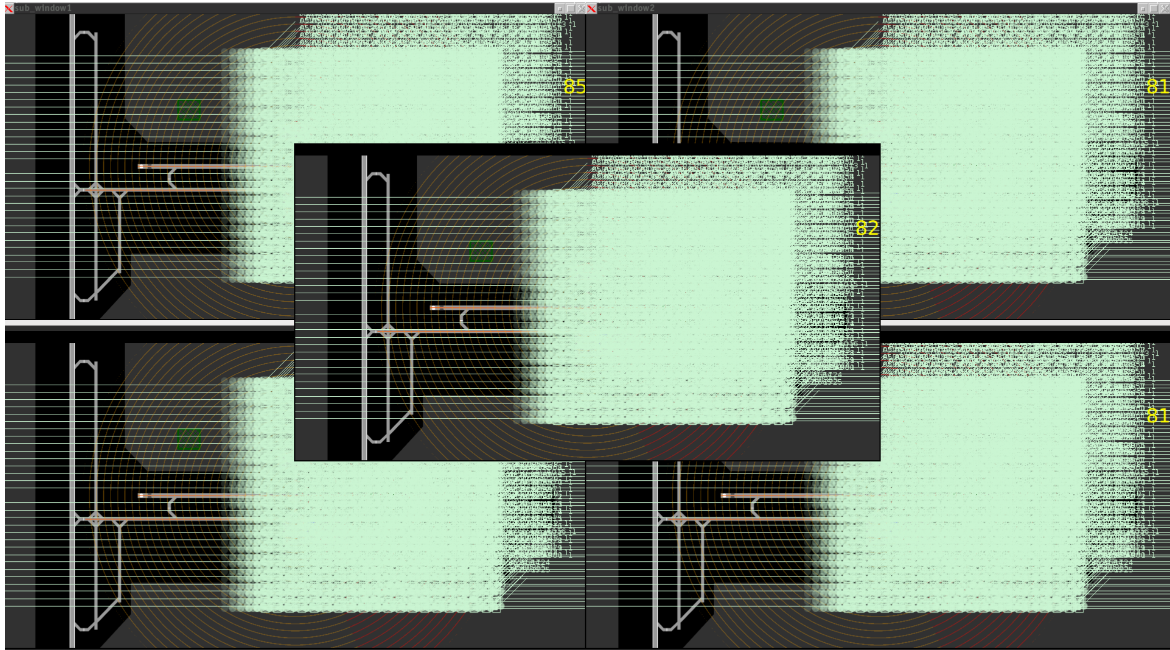


Abbildung 5.3: Die fünf Zoom-Fenster des HMI-Prototypen

5.1 ED-87C Parameter

Die Latenzzeiten TDL, IDL und RTOI werden im ED-87C Standard der European Organisation for Civil Aviation Equipment (EUROCAE) als wichtige HMI-Performance-Parameter ausgewiesen. Die IDL wird auf Basis der Bodenradarbildanzeige bestimmt. Als Startzeitpunkt der TDL-Messung fungiert der vom Tracker gesetzte Aktualisierungszeitstempel (Time of Report) eines Tracks. Die IDL wird ausgehend vom Empfangszeitpunkt einer Bildkachel gemessen. Die zu ermittelnde Latenzzeit ergibt sich aus der Verarbeitungszeit bis zum Beginn des Rendervorgangs (`QQuickWindow:beforeRendering()`) und der Renderzeit, einschließlich Framebuffer-Swap. Die Renderzeit enspricht dabei dem Maximalwert aus CPU-seitiger und GPU-seitiger Renderzeit. Für die GPU-seitige Messung werden OpenGL Timer-Query-Objekte [31, S. 493] eingesetzt.

Zur Messung der RTOI wird automatisiert mittels des X11-Programms `xdotool` auf einen dedizierten Qt Quick Button geklickt. Der HMI-Prototyp misst als Reaktion auf den Klick schließlich die Latenzzeit. Die Zwischenablage dient dabei als Kommunikationsmittel zum Austausch des für die Latenzzeitmessung notwendigen Startzeitpunkts des Klickvorganges. Die Messung wird mehrmals während eines Testlaufs wiederholt.

Die Einhaltung der vom ED-87C Standard geforderten Grenzwerte (siehe Tabelle 5.1) für die Latenzzeiten TDL, IDL und RTOI ist bei einer Track-Anzahl von mindestens 500 Tracks nachzuweisen. Die gemessenen Latenzzeiten sind in Abbildung 5.4 dargestellt und liegen auch bei 1000 Tracks deutlich unter den Grenzwerten.

Parameter	Anforderung	Maß
TDL	$\leq$ 500 ms	Maximalwert
IDL	$\leq$ 500 ms	Maximalwert
	$<$ 250 ms	Arithm. Mittelwert
RTOI	$\leq$ 500 ms	Maximalwert
	$<$ 250 ms	Arithm. Mittelwert

Tabelle 5.1: ED-87C Grenzwerte für die Latenzzeiten TDL [45, S. 25], IDL und RTOI [45, S. 30]

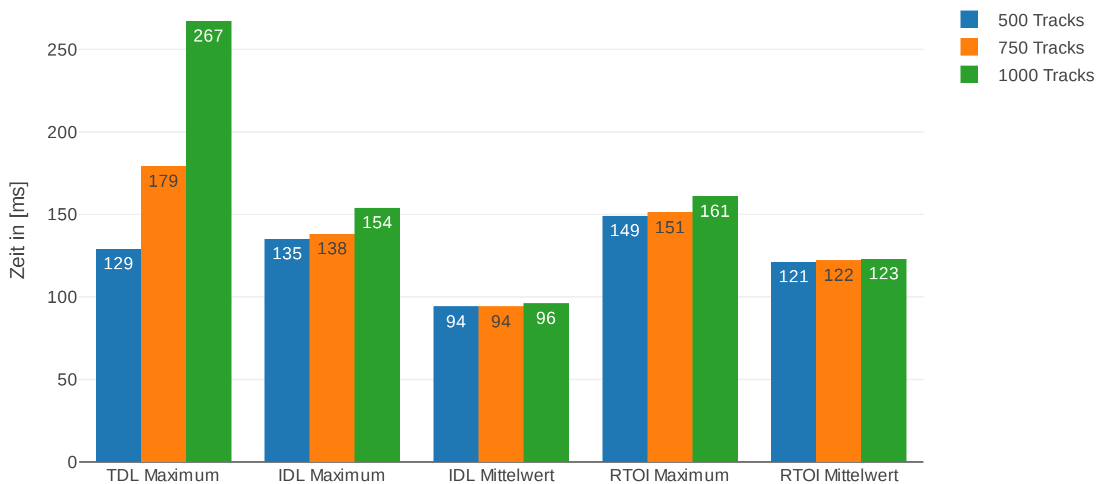


Abbildung 5.4: Latenzzeiten TDL (Maximalwert), IDL und RTOI (Maximalwert und arithmetischer Mittelwert) bei 500, 750 und 1000 Tracks

5.2 CPU-Last und Framerate

Die Messresultate in Abbildung 5.5 zeigen (von links nach rechts) die gesamte durch den HMI-Prototypen hervorgerufene CPU-Last, die CPU-Last des GUI/Main-Thread sowie die über alle Render-Threads gemittelte CPU-Last je Render-Thread. Die gesamte CPU-Last ist bei 500 Tracks mit rund 61 % um mehr als ein Viertel geringer als die des bestehenden ACEMAX-HMI (vgl. Abbildung 2.15). Auch bei einer Verdoppelung der Track-Anzahl auf 1000 Tracks pro Fenster bleibt die CPU-Last mit rund 73 % unterhalb des für das ACEMAX-HMI gemessenen Wertes von 86 % bei 500 Tracks. Die CPU-Last des GUI/Main-Thread beträgt weniger als Zehntel der gesamten CPU-Last. Die restliche Auslastung verteilt sich

auf die 6 Render-Threads (ein Render-Thread pro Fenster) und auf andere, im Diagramm nicht separat ausgewiesene Threads (e.g. Thread für den SMR-Texturdatentransfer).

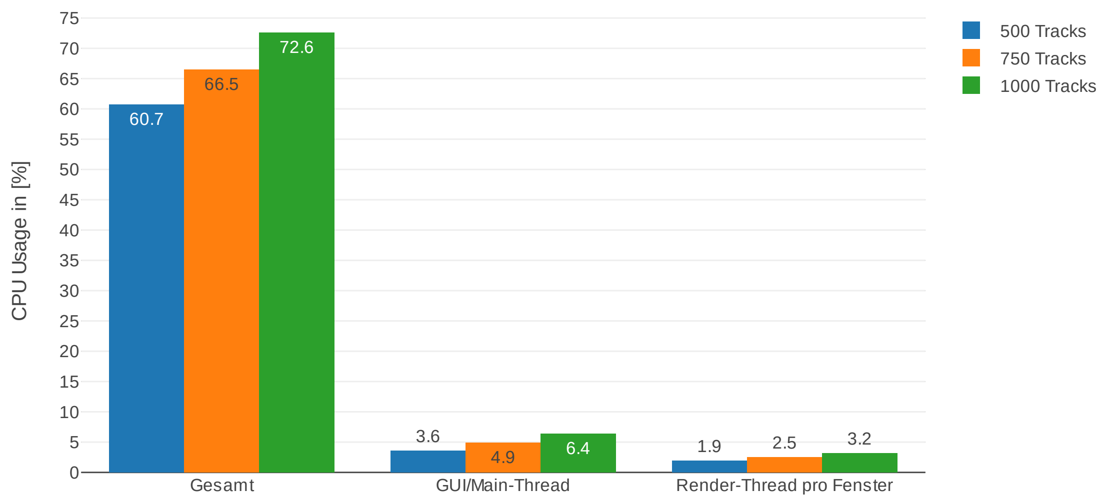


Abbildung 5.5: Durchschnittliche verursachte CPU-Last (Wertebereich: 0 - 400 %) des HMI-Prototypen bei 500, 750 und 1000 Tracks

Zur Messung der erzielbaren Frameraten wird der HMI-Prototyp in einem Modus betrieben, in dem die Anzeigeaktualisierung nicht inputgetrieben, sondern kontinuierlich erfolgt. Das heißt, ein neuer Rendervorgang beginnt direkt im Anschluss an dem vorhergehenden Rendervorgang. Die Messung erfolgt ohne vertikaler Synchronisation (VSync), um die maximal erzielbare Framerate bestimmen zu können. Das Messresultat ist in Abbildung 5.6 ersichtlich und zeigt, dass das Rendern einen vergleichweise kleinen Anteil (mittlere Framezeit < 13 ms) an den gemessenen Latenzzeiten besitzt.

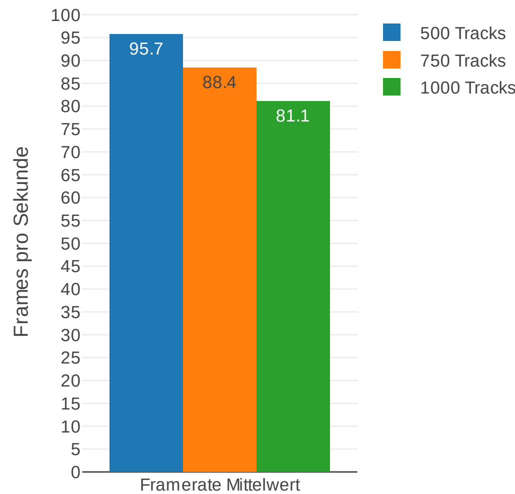


Abbildung 5.6: Mittelwert der pro Fenster gemessenen Framerate bei 500, 750 und 1000 Tracks

5.3 Asynchroner Texturdatentransfer

Die verwendete NVIDIA Grafikkarte (Quadro P400) verfügt über Dual-Copy-Engines und lässt sich durch die in Abschnitt 3.2 erläuterte und im HMI-Prototypen implementierte Architektur, eigener Thread mit dediziertem OpenGL-Kontext für den Transfer der SMR-Texturdaten, verwenden. Der Nachweis wird mithilfe des Tools GPUView [15] erbracht (vgl. Venkataraman [48, S. 421 ff.]). Dieses Tool ist nur für Windows erhältlich, sodass der HMI-Prototyp für diesen Zweck unter Windows betrieben werden muss. Das Auflösen von OpenGL-Funktionszeiger unter Windows wird durch Hilfsklassen von Qt, wie z. B. `QOpenGLFunctions` und `QOpenGLFunctions_3_3_Core`, wesentlich erleichtert.

Ein für den HMI-Prototypen erzeugtes Zeitverlaufsdiagramm ist in Abbildung 5.7 ersichtlich. Das Diagramm zeigt im oberen Teil die in den GPU-Hardware-Queues "3D" und "Copy" abgearbeiteten Aufgaben, welche in Form von Rechtecken dargestellt werden. Während die horizontale Zeitachse die zeitliche Abfolge und die Dauer einer Aufgabe angibt, zeigt die vertikale Achse die sich zum selben Zeitpunkt in der Queue befindlichen Aufgaben (unterste Aufgabe wird zuerst abgearbeitet). Die GPU-Hardware-Queues werden aus den zwei darunter dargestellten, mit "Transfer SMR-Texturdaten" und "Rendern" bezeichneten, CPU-seitigen Queues des HMI-Prototypen befüllt. Die Verwendung der GPU-Hardware-Queue "Copy" für den Transfer der SMR-Texturdaten belegt, dass der HMI-Prototyp in der Lage ist, die Dual-Copy-Engine zu nutzen.

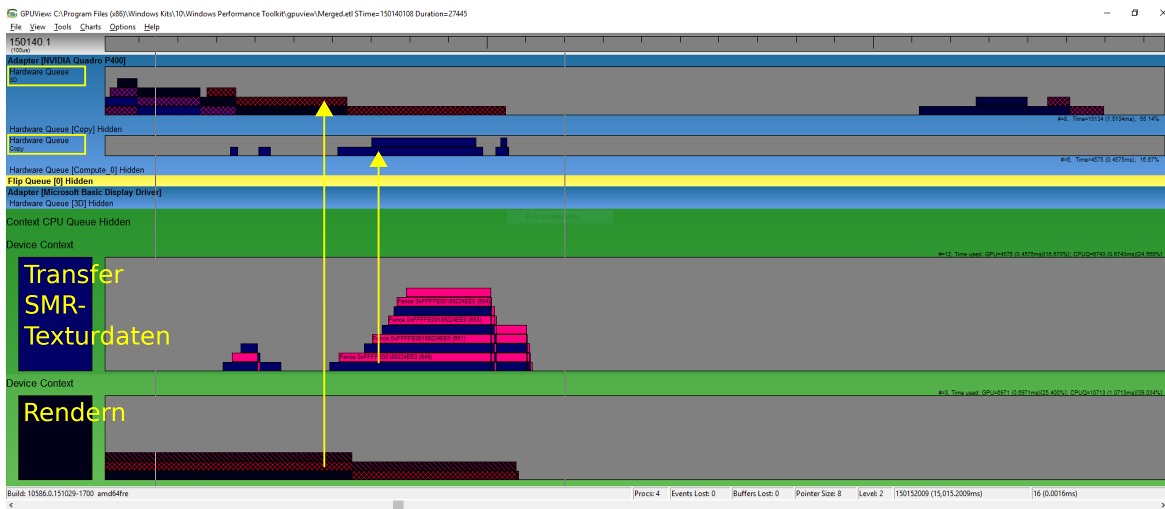


Abbildung 5.7: Mittels GPUView [15] generiertes Zeitverlaufsdiagramm, das die Nutzung der in der NVIDIA Grafikkarte verbauten Copy-Engine (Hardware-Queue "Copy") nachweist

6 Fazit und Ausblick

Als Fazit kann festgehalten werden, dass der in dieser Arbeit entwickelte Prototyp zur Verwirklichung eines neues Bodenlage-HMI auf Basis von Qt Quick die Latenzanforderungen des ED-87C Standards, unter der für den Nachweis gewählten Testkonfiguration, erfüllt. Die Latenzobergrenzen können auch bei einer Verdoppelung der vom Standard geforderten Mindesttrackanzahl, von 500 auf 1000 Tracks, deutlich eingehalten werden (Abbildung 5.4).

Für die Darstellung der Flughafenkarte, des Bodenradarbildes und der Ziele mit Qt Quick wurden verschiedene Ansätze aufgezeigt und diskutiert (Kapitel 4). Die in Abschnitt 2.5 identifizierten Faktoren, welche die Leistungsfähigkeit des bestehenden ACEMAX-HMI begrenzen, konnten im Entwurf adressiert und entgegenwirkt werden.

Die Verlagerung der fensterspezifischen Ansichtstransformation des Radarbildes auf die Grafikkarte und die zur Bereitstellung des Radarbildes als Textur mittels asynchronen Texturdatenstransfers (Abschnitt 4.4.2.2 und 3.2) hat maßgeblich zur Entlastung des GUI-Thread beigetragen (vgl. Abbildung 5.5 mit 2.15) und schafft dadurch die Voraussetzungen, um auch bei Volllast die Reaktivität des HMI auf Benutzereingaben zu gewährleisten.

Die Performance-Auswirkungen von Property-Bindings (Abschnitt 3.1.2.1), welche bei der Beschreibung des Labels eines Ziels mit Qt Quick entstehen, wurden aufgrund des statischen Label-Inhaltes (Anforderung R12, Abschnitt 2.4) nicht in Betracht gezogen. Die Performance des in dieser Arbeit verfolgten Ansatzes zur Zieldarstellung (Abschnitt 4.5.1) wäre daher bei sich ständig ändernden Label-Inhalten, wie z. B. Zeitzählern, noch zu prüfen.

Der implementierte Ansatz zur Darstellung der Flughafenkarte mit Qt Quick (Abschnitt 4.3.1) lässt sich auch auf andere, in dieser Arbeit nicht realisierte Layer des ACEMAX-HMI anwenden, welche auf der Grafikbibliothek AVLayerView (Abschnitt 2.3.4.1) beruhen. Ein oder mehrere solcher Layer könnten durch ein `QQuickPaintedItem` unter Verwendung des bestehenden `QPainter`-Codes umgesetzt werden. Der Grafikspeicherverbrauch beim Einsatz mehrerer `QQuickPaintedItem` Objekte in Bildschirm-/Fenstergröße und der bei sich oft aktualisierenden Layern, ineffiziente, zweistufige Rendervorgang stellen jedoch nicht vernachlässigbare Faktoren dar. Die Aktivierung von Anti-Aliasing für ein `QQuickPaintedItem` (multisampled OpenGL FBO) kann den Grafikspeicherverbrauch noch weiter steigen lassen.

Der für die Leistungsbewertung entwickelte Performance-Test kann in Zukunft auch für das ACEMAX-HMI verwendet werden. Er eignet sich einerseits als Regressionstest, um nach Änderungen eine Verschlechterung der Performance durch Vergleich mit den Ergebnissen eines vor den Änderungen durchgeführten Tests erkennen zu können. Andererseits lässt sich damit die Erfüllung der ED-87C Anforderungen nachweisen.

Die Erstellung von zusätzlichen Unit-Tests für die von Qt 3 nach Qt 5 zu portierenden Bibliotheken der Firma ADB Safegate Austria (Abschnitt 4.1.2) hat sich bewehrt. Die durch Verhaltensänderung von Qt-Klassen bedingten Fehler konnten so leichter gefunden werden. Wie in 5.3 gezeigt wurde, kann der HMI-Prototyp auch unter Windows betrieben werden.

Die Verwendung der Qt-Hilfsklassen für OpenGL, wie z. B. `QOpenGLFunctions`, hat das Entwickeln von plattformunabhängigen OpenGL-Code wesentlich erleichtert.

Ausblick

Das ACEMAX-HMI wird stetig um weitere Funktionalität erweitert, sodass eine komplette Neuentwicklung zum Zeitpunkt des Verfassens dieser Arbeit als nicht möglich erscheint. Es ist jedoch technisch möglich, die konventionelle Widget-basierte GUI-Technologie zusammen mit Qt Quick zu verwenden. Die Darstellung der Bodenlage kann dadurch mit Qt Quick auf Basis der im HMI-Prototyp implementierten Qt Quick-Items erfolgen, während alle anderen Widget-basierten Dialoge und Fenster bestehen bleiben können.

Mit Qt 5.10 und des darin eingeführten QML-Basistypen `Shape` eröffnet sich ein alternativer Ansatz zur Umsetzung bestehender ACEMAX-HMI-Layer mit Qt Quick, um die oben genannten Nachteile durch Einsatz von `QQuickPaintedItem` zu vermeiden. Dieser neue Ansatz erlaubt es, Vektorgrafiken mittels Pfad-Elementen unter Angabe der Darstellungseigenschaften (e.g. Linienstil) deklarativ zu beschreiben. Zum Rendern der Pfad-Elemente stellt Qt zwei Methoden zur Auswahl: mittels Triangulierung der für die Darstellung der Kontur und Füllung eines Pfades erzeuten Geometrieobjekte oder durch Einsatz der Nvidia-spezifischen OpenGL-Erweiterung `NV_path_rendering` [29].

Durch den Wegfall des in Fenstergröße vorliegenden OpenGL FBO pro Layer ist zu erwarten, dass der Grafikspeicherverbrauch im Vergleich zur Verwendung von `QQuickPaintedItem` wesentlich geringer ausfällt. Es ist auch anzunehmen, dass bei gleichbleibenden Eigenschaften des zu zeichnenden Vektorgrafikobjekts die Renderzeit, anders als bei `QQuickPaintedItem` (Texturaktualisierung erforderlich), sich invariant gegenüber einer Änderung der Ansichtstransformation zeigt. Wie auch für den QML-Basistypen `Text` werden jedoch keine C++-Klassen zur Verfügung gestellt (Stand Qt 5.10), um mittels C++ Pfad-Elemente im Szenengraph zu erzeugen.

Abschließend sei noch die Open-Source-Bibliothek QNanoPainter [18] als weitere alternative Möglichkeit zu `QQuickPaintedItem` erwähnt. Die QNanoPainter-API ist für die Verwendung mit Qt ausgelegt, besitzt einen ähnlichen Funktionsumfang wie `QPainter`, um Vektorgrafiken mit OpenGL zu zeichnen (unter Zuhilfenahme der Open-Source-Bibliothek NanoVG [34]) und lässt sich z. B. in Verbindung mit `QSGRenderNode` im Qt-Quick-Szenengraphen verwenden.

Abkürzungsverzeichnis

A-SMGCS Advanced Surface Movement Guidance and Control System
AIP Aeronautical Information Publication
AIS Aeronautical Information Service
API application Programming Interface)
ATC Air Traffic Control
ATFM Air Traffic Flow Management
ATM Air Traffic Management
ATS Air Traffic Service
CNS Communication Navigation Surveillance
CPU Central Processing Unit
CWP Controller Working Position
DEFAMM Demonstration Facilities for Aerodrome Movement Management
DMA Direct Memory Access
FBO Frame Buffer Object
FIS Flight Information Service
GCC Gnu Compiler Collection
GLSL OpenGL Shading Language
GPU Graphics Processing Unit
GUI Graphical User Interface
LUT Lookup Table
HMI Human Machine Interface
IDL Information Display Latency
MET Meteorology
MSAA Multisample Antialiasing
MVC Model View Controller
NOTAM Notice to Airman
OpenCL Open Computing Language
OpenGL Open Graphics Library
PBO Pixel Buffer Object
PCIe Peripheral Component Interconnect Express
QML Qt Modeling Language

RTO Response Time to Operator Input
SMGCS Surface Movement Guidance and Control System
SMR Surface Movement Radar
TDL Target Display Latency
UBO Uniform Buffer Object
UIC User Interface Compiler
VBO Vertex Buffer Object
WGS84 World Geodetic System 1984
SAR Search and Rescue

Literaturverzeichnis

[1] ADB Safegate Austria, https://adbsafegate.com, Zugriffsdatum: 14.07.2018.

[2] T. Akenine-Moller, E. Haines und N. Hoffman, *Real-time rendering*, 3rd Aufl., A. K. Peters, Ltd., Natick, MA, USA, 2008, ISBN 978-1-56881-424-7.

[3] J. Blanchette und M. Summerfield, *C++ gui programming with qt 4*, Prentice Hall PTR, 2008, ISBN 978-0-13-235416-5.

[4] F. Buschmann, *Pattern oriented software architecture: a system of patters*, John Wiley&Sons, 1999.

[5] C++ Standards Committee and others, *ISO/IEC 14882:2011: Information technology — Programming languages — C++*, 2011, ISO Norm.

[6] CentOS, http://www.centos.org/, Zugriffsdatum: 20.06.2018.

[7] The Qt Company, https://www.qt.io/, Zugriffsdatum: 02.06.2018.

[8] A. Cook, *European Air Traffic Management: Principles, Practice, and Research*, Ashgate, 2007, ISBN 978-0-7546-7295-1.

[9] European Organisation for the Safety of Air Navigation (EUROCONTROL), *Mid-Term Forecast Flights 2012-2018*, 2012.

[10] European Organisation for the Safety of Air Navigation (EUROCONTROL), https://www.eurocontrol.int/, Zugriffsdatum: 10.06.2018.

[11] European Organization for the Safety of Air Navigation (EUROCONTROL), *WGS 84 Implementation Manual (Edition 2.4)*, 1998.

[12] European Organization for the Safety of Air Navigation (EUROCONTROL), *Definition of A-SMGCS Implementation Levels (Edition 1.2)*, 2010.

[13] European Organization for the Safety of Air Navigation (EUROCONTROL), *Specification for Advanced-Surface Movement Guidance and Control System (A-SMGCS) Services (Edition 1.0)*, März 2018.

[14] European Organization for the Safety of Air Navigation (EUROCONTROL), *Surveillance: Summary of Terminology*, https://ext.eurocontrol.int/lexicon/index.php/Main_Page, Zugriffsdatum: 01.06.2018.

[15] M. Fisher und S. Pronovost, *GPUView*, https://graphics.stanford.edu/~mdfisher/GPUView.html, Zugriffsdatum: 10.06.2019.

[16] H. Flühr, *Avionik und Flugsicherungstechnik*, Springer, 2010, ISBN 978-3-642-01612-7.

[17] R.C. Gonzalez und R.E. Woods, *Digital Image Processing*, Pearson Education, 2008.

[18] K. Grönholm, *QNanoPainter*, https://github.com/QUItCoding/qnanopainter/, Zugriffsdatum: 30.06.2019.

[19] L. Hrabcak und A. Masserann, *Asynchronous buffer transfers*, OpenGL Insights (P. Cozzi und C. Riccio, Hrsg.), CRC Press, July 2012, `http://www.openglinsights.com/`, S. 391–414, ISBN 978-1-4398-9376-0.

[20] J.F. Hughes und J.D. Foley, *Computer graphics: Principles and practice*, Addison-Wesley, 2013, ISBN 978-0-321-39952-6.

[21] IceWM (X11 Window Manager), `https://github.com/bbidulock/icewm/`, Zugriffsdatum: 10.06.2018.

[22] International Civil Aviation Organization (ICAO), *Manual of Surface Movement Guidance and Control Systems (SMGCS)*, 1. Aufl., 1986, ICAO Doc 9476.

[23] International Civil Aviation Organization (ICAO), *Advanced Surface Movement Guidance and Control System (A-SMGCS) Manual*, 1. Aufl., 2004, ICAO Doc 9830.

[24] International Civil Aviation Organization (ICAO), `http://www.icao.org`, Zugriffsdatum: 01.06.2018.

[25] M. Jevtic, M. Tatarevic, K. Markovic, T. Pajic und M. Stamatovic, *Performance trade-offs in GPU-assisted radar scan converter*, Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), 2013 11th International Conference on, Bd. 2, IEEE, 2013, S. 561–564.

[26] J. Jimenez, D. Gutierrez, J. Yang, A. Reshetov, P. Demoreuille, T. Berghoff, C. Perthuis, H. Yu, M. McGuire, T. Lottes, H. Malan, E. Persson, D. Andreev und T. Sousa, *Filtering approaches for real-time anti-aliasing*, ACM SIGGRAPH 2011 Courses (New York, NY, USA), SIGGRAPH '11, ACM, 2011, ISBN 978-1-4503-0967-7.

[27] Jupyter Project, `https://jupyter.org/`, Zugriffsdatum: 10.06.2019.

[28] J. Kessenich, G. Sellers und D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V*, 9. Aufl., Addison-Wesley Professional, 2016, ISBN 978-0-13-449549-1.

[29] Mark J. Kilgard und Jeff Bolz, *Gpu-accelerated path rendering*, ACM Trans. Graph. **31** (2012), Nr. 6, 172:1–172:10, ISSN 0730-0301, `http://doi.acm.org/10.1145/2366145.2366191`.

[30] Luftfahrtgesetz 1957 (LFG), In: BGBl 253/1957 idF BGBl I 92/2017.

[31] C. Lux, *The opengl timer query*, OpenGL Insights (P. Cozzi und C. Riccio, Hrsg.), CRC Press, July 2012, `http://www.openglinsights.com/`, S. 493–502, ISBN 978-1-4398-9376-0.

[32] H. Mensen, *Moderne Flugsicherung - Organisation, Verfahren, Technik*, 3. Aufl., VDI-Buch, Springer, 2004, ISBN 978-3-540-20581-4.

[33] MIT X Consortium, *The MIT Shared Memory Extension*, `http://www.x.org/releases/current/doc/xextproto/shm.html`, Zugriffsdatum: 11.07.2018.

[34] M. Mononen, *NanoVG*, `https://github.com/memononen/nanovg/`, Zugriffsdatum: 30.06.2019.

[35] K. Nickerson und S. Haykin, *Scan conversion of radar images*, Aerospace and Electronic Systems, IEEE Transactions on **25** (1989), Nr. 2, 166–175.

[36] A. Norris, A.D. Wall, A.G. Bole und W.O. Dineley, *Radar and arpa manual: Radar and target tracking for professional mariners, yachtsmen and users of marine radar*, Elsevier Science, 2005, ISBN 978-0-08-048052-7.

[37] NVIDIA, *NVIDIA Quadro Dual Copy Engines*, 2010, `https://www.nvidia.com/docs/IO/40049/Dual_copy_engines.pdf`, Zugriffsdatum: 01.06.2018.

[38] Open Graphics Library, `https://www.opengl.org/`, Zugriffsdatum: 01.06.2018.

[39] Open Scene Graph, `http://www.openscenegraph.org/`, Zugriffsdatum: 10.06.2018.

[40] V. Pezhgorski und M. Lazarova, *Real time gpu accelerated radar scan conversion and visualization*, Proceedings of the 18th International Conference on Computer Systems and Technologies (New York, NY, USA), CompSysTech'17, ACM, 2017, S. 249–256, ISBN 978-1-4503-5234-5, `http://doi.acm.org/10.1145/3134302.3134339`.

[41] D. L. G. Pinzón und J. M. P. Espartero, *Real time scan conversion implementation for high resolution radars*, 2016 17th International Radar Symposium (IRS), Mai 2016, S. 1–4.

[42] Sellers, G. and Wright, R.S. and Haemel, N., *OpenGL SuperBible: Comprehensive Tutorial and Reference*, OpenGL, Pearson Education, 2013, ISBN 978-0-13-336508-5.

[43] P. Sharma, V. Nagabhushanam, R. Bhandarkar, L. Ramakrishnan und R. Prakash Reddy, *Radar display gpu coding with the graphics api*, Proc of 9th International Radar Symposium India (IRSI-13), Bangalore, India, 2013.

[44] M. Stamatovic, M. Jevtić, U. Kisić und M. Tatarević, *Design and implementation of a modern radar display for air surveillance applications*, 2012 20th Telecommunications Forum (TELFOR), November 2012, S. 1520–1523.

[45] The European Organisation for Civil Aviation Equipment (EUROCAE), *Minimum Aviation System Performance Specification for Advanced Surface Movement Guidance and Control Systems (A-SMGCS) Levels 1 and 2*, Jänner 2015, EUROCAE ED-87C.

[46] The Khronos Group, Inc., *OpenGL ARB_draw_instanced Specification*, 2011, `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_draw_instanced.txt`, Zugriffsdatum: 02.11.2018.

[47] The Qt Company, *Thread Support in Qt (Qt3)*, `https://doc.qt.io/archives/3.3/threads.html`, Zugriffsdatum: 02.07.2018.

[48] S. Venkataraman, *Fermi asynchronous texture transfers*, OpenGL Insights (P. Cozzi und C. Riccio, Hrsg.), CRC Press, July 2012, `http://www.openglinsights.com/`, S. 415–430, ISBN 978-1-4398-9376-0.

[49] Wikipedia, *PCI Express*, `https://en.wikipedia.org/w/index.php?title=PCI_Express&oldid=849548332`, Zugriffsdatum: 11.07.2018.

[50] R. S. Wright, B. Lipchak und N. Haemel, *OpenGL Super Bible*, Addison Wesley, 2007.

Appendizes

A.1 CPU-Last des ACEMAX-HMI – Testablauf und statistische Zusammenfassung

Dieser Anhang enthält eine Beschreibung des Testablaufs und eine statistische Zusammenfassung der Messergebnisse zu der in Kapitel 2.5.1 durchgeführten Analyse des ACEMAX-HMI.

Für jede Testkonfiguration wurde folgender Testablauf 10-mal ausgeführt:

1. Die Testkonfiguration anpassen,
2. die Simulationsumgebung und das HMI starten und
3. die durchschnittlichen CPU-Last des ACEMAX-HMI und des X-Servers pro Sekunde über einen Zeitraum von 1 Minute mittels dem Linux-Programm `pidstat` aufzeichnen.

	Tracks+SMR	X@Tracks+SMR	Track-Dv.	Tracks	X@Tracks
mean	69.8	58.4	1.9	16.5	9.8
std	0.4	0.7	0.1	0.1	0.5
min	69.4	57.3	1.8	16.3	9.3
25 %	69.5	58.1	1.9	16.5	9.3
50 %	69.6	58.5	1.9	16.5	9.8
75 %	70.0	58.7	1.9	16.5	9.9
max	70.5	59.4	2.1	16.7	10.9

Tabelle A.1: ACEMAX-HMI: CPU-Last (Wertebereich: 0 - 400 %) bei 125 Tracks

	Tracks+SMR	X@Tracks+SMR	Track-Dv.	Tracks	X@Tracks
mean	79.4	56.8	2.6	30.8	17.1
std	0.4	0.4	0.1	0.2	0.4
min	78.7	55.7	2.5	30.7	16.3
25 %	79.2	56.6	2.5	30.7	16.8
50 %	79.4	56.8	2.6	30.8	17.0
75 %	79.7	57.1	2.6	30.9	17.4
max	79.8	57.2	2.7	31.2	17.6

Tabelle A.2: ACEMAX-HMI: CPU-Last (Wertebereich: 0 - 400 %) bei 250 Tracks

	Tracks+SMR	X@Tracks+SMR	Track-Dv.	Tracks	X@Tracks
mean	86.8	45.3	4.0	60.1	31.2
std	0.6	0.8	0.1	0.6	0.6
min	86.2	44.4	3.8	59.5	30.6
25 %	86.3	44.8	4.0	59.9	30.8
50 %	86.6	44.9	4.0	60.0	31.1
75 %	87.2	46.0	4.1	60.3	31.2
max	88.0	46.5	4.1	61.5	32.8

Tabelle A.3: ACEMAX-HMI: CPU-Last (Wertebereich: 0 - 400 %) bei 500 Tracks

	SMR-Image-Composer	SMR	X@SMR
mean	19.9	45.7	32.8
std	0.4	0.4	0.6
min	18.9	45.0	31.8
25 %	19.7	45.5	32.2
50 %	19.9	45.7	32.9
75 %	20.3	45.9	33.3
max	20.4	46.4	33.6

Tabelle A.4: ACEMAX-HMI: CPU-Last (Wertebereich: 0 - 400 %) ohne Tracks (nur SMR)

A.2 Ergebnisse der Leistungsbewertung des HMI-Prototypen

Die nachfolgenden Tabellen enthalten eine statistische Zusammenfassung der für die Leistungsbewertung des HMI-Prototypen durchgeführten Messungen (siehe Kapitel 5). Für jede Testkonfiguration wurden 10 Testläufe durchgeführt. Die Dauer eines einzelnen Testlaufs betrug 120 Sekunden. Die durchschnittliche CPU-Last des HMI-Prototypen pro Sekunde wurde mithilfe der Python Bibliothek `psutil` aufgezeichnet.

	TDL	IDL	RTOI	Framerate
mean	77.7	94.0	120.6	95.7
std	15.2	20.5	17.5	12.1
min	47.0	50.0	19.0	1.0
25 %	64.0	73.0	120.0	95.0
50 %	78.0	101.0	124.0	97.0
75 %	89.0	112.0	128.0	99.0
max	129.0	135.0	149.0	105.0

Tabelle A.5: HMI-Prototyp: Latenzzeiten in [ms] und Framerate (Frames pro Sekunde) bei 500 Tracks

	TDL	IDL	RTOI	Framerate
mean	112.8	93.6	121.9	88.4
std	16.6	20.5	18.9	6.3
min	73.0	51.0	18.0	1.0
25 %	100.0	73.0	121.0	87.0
50 %	114.0	101.0	125.0	89.0
75 %	124.0	111.0	132.0	90.0
max	179.0	138.0	151.0	95.0

Tabelle A.6: HMI-Prototyp: Latenzzeiten in [ms] und Framerate (Frames pro Sekunde) bei 750 Tracks

	TDL	IDL	RTOI	Framerate
mean	154.3	95.7	123.3	81.1
std	34.1	20.7	20.3	2.3
min	102.0	51.0	20.0	72.0
25 %	127.0	76.0	122.0	80.0
50 %	141.0	103.0	126.0	81.0
75 %	183.0	113.0	134.0	83.0
max	267.0	154.0	161.0	88.0

Tabelle A.7: HMI-Prototyp: Latenzzeiten in [ms] und Framerate (Frames pro Sekunde) bei 1000 Tracks

	Gesamt	GUI/Main-Thread	Render-Thread pro Fenster
mean	60.7	3.6	1.9
std	5.4	1.4	0.2
min	32.6	1.5	1.0
25 %	59.9	2.6	1.8
50 %	61.4	3.1	1.8
75 %	63.2	4.2	1.9
max	70.6	8.8	2.7

Tabelle A.8: HMI-Prototyp: CPU-Last (Wertebereich: 0 - 400 %) bei 500 Tracks

	Gesamt	GUI/Main-Thread	Render-Thread pro Fenster
mean	66.5	4.9	2.5
std	5.7	1.8	0.3
min	37.0	2.8	1.3
25 %	65.2	3.6	2.4
50 %	67.0	4.3	2.5
75 %	69.2	5.7	2.7
max	78.4	10.8	3.5

Tabelle A.9: HMI-Prototyp: CPU-Last (Wertebereich: 0 - 400 %) bei 750 Tracks

	Gesamt	GUI/Main-Thread	Render-Thread pro Fenster
mean	72.6	6.4	3.2
std	5.2	2.2	0.3
min	42.1	2.5	1.8
25 %	70.6	4.8	3.0
50 %	72.7	5.7	3.2
75 %	75.1	7.5	3.3
max	87.9	13.9	4.2

Tabelle A.10: HMI-Prototyp: CPU-Last (Wertebereich: 0 - 400 %) bei 1000 Tracks