IV Visualizer: Qualitätssicherung

Fraunhofer-Institut für Optronik, Systemtechnik und Bildauswertung IOSB
Josua Benjamin Eyl, Lukas Friedrich, Robin Köchel, Nathaniel Till Hartmann, Max Bretschneider
Betreuende Mitarbeiter: Mickael Cormier M.Sc., Stefan Wolf M.Sc.

Inhaltsverzeichnis

1 Vorwort

Dieses Qualitätsheft dient als Zusammenfassung der genutzten Maßnahmen der Qualitätssicherung des Projekts IV_Visualizer. Dieses wird im Rahmen des Wintersemesters 23/24 im Auftrag des Fraunhofer IOSB entwickelt. Die Betreuer sind Mickael Cormier und Stefan Wolf. Es werden wichtige Werkzeuge zur Codeanalyse vorgestellt, beschrieben und die Ergebnisse dieser dokumentiert.

2 Reflexion

In diesem Kapitel werden die abgesprochenen funktionalen, sowie nicht-funktionalen Anforderungen des Pflichtenhefts aufgeführt und der erreichte Fortschritt an diesen aufgezeichnet.

Der Status der Anforderungen wird wie folgt kategorisiert:

- Fertig(grün)
- in Entwicklung(orange)
- noch nicht fertig(rot)
- verworfen(grau)

2.1 Funktionale Anforderungen

Hier ein Überblick über den Bearbeitungszustand aller funktionalen Anforderungen

Pflicht	Beschreibung	Fortschritt	Bemerkung
FA10	Anzeigen eines Streams		funktionstüchtig, aber nicht fehlerfrei
FA20	Erstellen, laden und löschen einzelner Projekte		
FA30	Hinzufügen eines Linien-Diagramms		
FA40	Anzeigen von Logdaten		
FA50	Sortieren der Logdaten		
FA60	Speicherung der Projekteinstellungen		Einstellungen liegen in Config-Dateien
FA70	Bis zu 72 Stunden im Videostream zurückspulen		
FA80	Stream pausieren		
FA90	Betrachten einzelner Frames		
FA100	Spezielle Bereiche können hinzugefügt werden		
FA110	Spezielle Bereiche können entfernt werden		
FA120	Konfiguration des Streams		
FA130	Video mit veränderter Geschwindigkeit abspielen		
FA140	Exportieren von Videostreams		
FA150	Unbegrenztes zurückspulen im Video mit nur Annotationen		
FA160	Filtern der Logdaten nach Error-, Debug- und Warning-Nachrichten		
FA165	Die Länge der Speicherung von Annotationen und Videos kann angepasst werden		

Tabelle 1: Zustand funktionale Anforderungen

2.2 Nicht funktionale Anforderungen

Hier ein Überblick über den Bearbeitungszustand aller nicht-funktionalen Anforderungen

Pflicht	Beschreibung	Fortschritt	Bemerkung
NA10	Die Software wird in C++ implementiert		
NA20	Für die Implementierung wird das Anwendungsframework Qt verwendet		
NA30	Die Architektur ist objektorientiert und offen für Erweiterung		
NA40	Die Benutzeroberfläche soll übersichtlich und intuitiv aufgebaut sein		
NA50	Das System kann ohne Beeinträchtigung 10 Videostreams zeitgleich aufzeichnen		durch Hardwareeinschränkungen nicht testbar
NA60	Das System kann ohne Beeinträchtigung 4 Videostreams mit Annotation zeitgleich wiedergeben		durch Hardwareeinschränkungen nicht vollständig testbar
NA70	Videostreams mit bis zu 30 FPS können ohne Beeinträchtigung des Systems verarbeitet werden		
NA80	Das System läuft auch bei fehlerhaften Eingaben stabil		

Tabelle 2: Zustand nicht-funktionale Anforderungen

2.3 optionale Anforderungen

Hier ein Überblick über den Bearbeitungszustand aller optionaler Anforderungen

Pflicht	Beschreibung	Fortschritt	Bemerkung
FA170	Zum Öffnen des Visualizers und für den Zugriff auf die Daten wird ein Passwort benötigt		
FA180	Die Annotationen heben sich, abhängig von ihrer Klasse, voneinander ab		
FA190	Hinzufügen eines Kuchendiagramms		

Tabelle 3: Zustand optionale Anforderungen

3 Qualitätssicheung während der Implementierung

In diesem Kapitel werden Mittel zur Qualitätssicherung vorgestellt und beschrieben, die bereits während der Implementierungsphase genutzt wurden.

3.1 CI/CD-Pipeline

Die CI/CD-Pipeline ist eine Pipeline, die in GitLab zur Codeüberprüfung eingesetzt werden kann. Die CI/CD-Pipeline besteht aus verschiedenen Stufen, die jeweils den Code auf bestimmte Anforderungen überprüfen. Im Falle des Projekts IV_Visualizer wurden folgende Stufen benutzt:

3.1.1 Frontend

- Der Build-Prozess überprüft, ob der Code ohne Fehler des Kompilierers erstellbar ist
- die Dokumentation wird von Doxygen erzeugt

3.1.2 Backend

- Der Build-Prozess überprüft, ob der Code ohne Fehler des Kompilierers erstellbar ist
- durch Unit-Tests wird die Funktionalität einzelner Komponenten überprüft
- die Dokumentation wird von Doxygen erzeugt

Durch die CI/CD-Pipeline wird bei jedem Push auf einen Branch sichergestellt, dass der Code für die erforderlichen Standards eingehalten werden bzw. nicht erfüllte Standards identifiziert werden können.

3.2 Issues

Für die Verwaltung von Aufgaben stellt GitLab Issues zur Verfügung, welche mit Labels versehen werden können, um diese genauer zu kategorisieren. Diese Issues waren ein zentraler Bestandteil der Entwicklung des Programms IV_Visualizers und wurden mit dem Fortschritt des Projektes immer häufiger und intensiver genutzt, um die Aufgabenverteilung des Entwicklerteams zu organisieren. Innerhalb eines Issues können Fortschritt und Zeitaufwand dokumentiert werden.

3.3 Branches

Für die Implementierung der Anforderungen/Features und Umsetzung der Issues wurde die Funktion von Branches ausgiebig genutzt. Jeder Entwickler hat für jede Aufgabe immer einen Branch vom letzten Entwicklungsstand erstellt, das Feature implementiert und den Branch nach dem Abschluss der Implementierung und Review auf den Development-Branch gemerged.

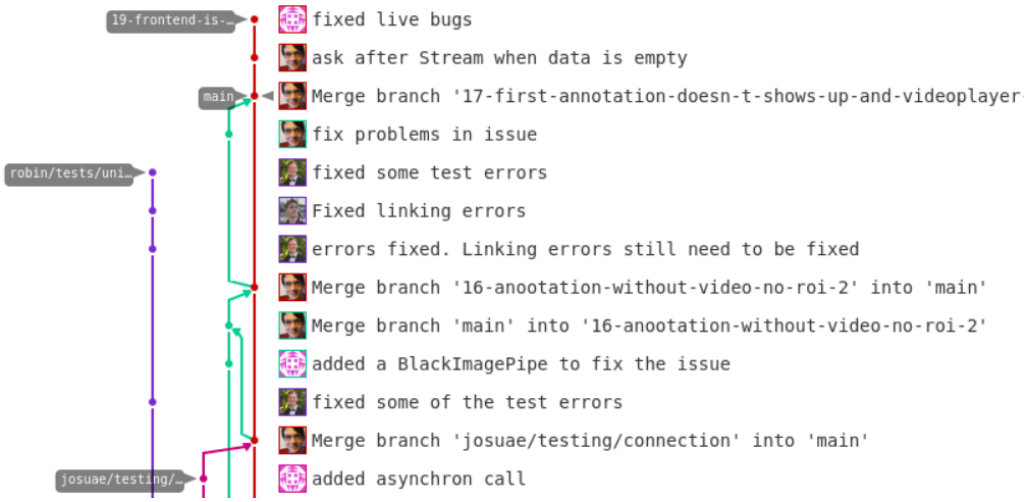

Abbildung 1: commit Graph

4 Statische Code Analyse

In diesem Kapitel werden Mittel zur statischen Codeanalyse vorgestellt und beschrieben.

4.1 Doxygen

Die Dokumentation des Programmcodes wurde mit Doxygen automatisch in ein HTML-Dokument generiert, welches aus der Dokumentation aller Klassen und Methoden besteht.

4.2 CLOC

Um die insgesamt geschriebenen Zeilen Code zu messen, wurde das Werkzeug CLOC benutzt. Dieses zählt alle Zeilen Code im Programm und unterteilt diese in Leer-, Kommentar- und Codezeilen. Außerdem zählt das Werkzeug die Anzahl der benutzten Dateien.

5 Testszenarien

Folgend wird die Ausführung der manuellen Tests dokumentiert. Die Resultate werden wie folgt kategorisiert.

- wie erwartet (grün)
- mit Fehlern (orange)
- nicht ausführbar (rot)

5.1 Testszenarien aus dem Pflichtenheft

Szenario	Beschreibung	Ergebnis	Testfälle
S10	Erstelle und konfiguriere einen neuen Videostream. Spule diesen an den Anfang und pausiere ihn dort		T10c, T70a, T70b, T80, T120, T150, T260
S20	Füge eine Kamera hinzu und spiele den Stream mit zweifacher Geschwindigkeit und ohne Annotationen ab.		T10a, T20a, T20b, T60, T270, T240, T130
S30	Setzen einer Maske und anzeigen lassen der Logdaten, sowie die Filterung von diesen		T40, T100a, T160
S40	Exportiere den gesamten Videostream mit Annotationen. Springe zudem innerhalb des Videos ein Frame nach vorne		T140, T90, T300
S50	Zeige einen Videostream korrekt mit Annotationen, die Annotationen werden bei Verlassen des Bildschirmrandes nicht weiter angezeigt. Bei fehlerhafter Positionsmeldung der KI wird die Annotation nicht angezeigt		T170, T180, T190, T280, T310
S60	Konfiguriere einen Videostream und reagiere auf fehlerhafte Eingabe Konfigurationseinstellungen		T180, T210
S70	Handhabung von Unterbrechungen in einem Videostream		T190, T200,T290, T300
S90	Verwaltung von Projekt-Konfigurationen bei unsachgemäßem Beenden des Programms		T210, T220
S100	Einstellen und Aktualisieren von Kamera- und KI-Parametern		T200, T220, T230

Tabelle 4: Zustand der Testszenarien

6 Tests

In diesem Kapitel werden Mittel zum Testen des Programmcodes aufgelistet und beschrieben. Des Weiteren werden Funktionen zur Übersicht über Tests vorgestellt. Generell wurde das Hauptaugenmerk nicht auf Unit-Tests gelegt. Vielmehr wurde durch manuelles Testen, oft von Hand, Fehler festgestellt und behoben. Aufgrund der hohen Komplexität des Programms wurde erst die Funktionalität des Projekts sichergestellt, bevor eine hohe Testabdeckung angestrebt wurde. Weil das Programm zu Beginn der Qualitätssicherung noch nicht vollumfänglich betriebsbereit war, wurde das und das Beheben von z.B. Segmentation Fehlern zur obersten Priorität und alles Weitere (Erweiterung des Visualizers, Unit-Tests schreiben, etc.) wurde untergeordnet.

6.1 Google Tests

Google Test ist eine Bibliothek zum Testen von auf C++ basierendem Code. In manchen Fällen ist es nicht mit geringem Aufwand möglich, Unit-Tests über das Qt Test Framework durchzuführen.

6.2 Frontend

Alle QWidget Klassen, unter anderem der Ordner GraphicalUserInterface wurden nicht mithilfe von Unit-Tests getestet, sondern von Hand.

source/BackendConnector		17.8 %	106 / 594	40.0 %	34 / 85
source/BackendStub		11.3 %	14 / 124	18.2 %	4 / 22
source/DataStreams		24.9 %	46 / 185	17.5 %	7 / 40
source/Export		11.2 %	21 / 188	21.1 %	4 / 19
source/ExportSettings		54.5 %	102 / 187	100.0 %	10 / 10
source/FrameData/Config		61.8 %	126 / 204	77.6 %	38 / 49
source/FrameData/Data		27.7 %	92 / 332	47.9 %	35 / 73
source/GraphicalUserInterface		0.0 %	0 / 376	0.0 %	0 / 52
source/Log		12.5 %	18 / 144	40.0 %	8 / 20
source/Pipeline		27.6 %	58 / 210	37.1 %	13 / 35
source/VideoPlayer		0.0 %	0 / 395	0.0 %	0 / 58
source/ViewType		44.4 %	28 / 63	58.3 %	14 / 24
source/main		38.1 %	107 / 281	37.9 %	11 / 29

Abbildung 2: frontend Testabdeckung

6.3 Backend

DBManager.cpp		92.8 %	374 / 403	96.0 %	24 / 25
DataManager.cpp		24.4 %	19 / 78	21.4 %	3 / 14
Encoder.cpp		0.0 %	0 / 127	0.0 %	0 / 11
FSManager.cpp		25.5 %	35 / 137	42.9 %	3 / 7

Abbildung 3: Backend - Model Tests

MessageConverter.cpp		7.4 %	20 / 272	5.6 %	1 / 18
MessageHandler.cpp		2.5 %	7 / 284	8.7 %	2 / 23

Abbildung 4: Backend - Controller Test