BA
Dokumentation

Sport live image stabilization in the cloud

Semester: Frühling 2025

Version: 1.0
Date: 2025-06-12
Git Version: a3d57ea

Projekt Team:	David Stäheli
	Cedric Christen
Projektbetreuer:	Norbert Frei
Co-Betreuer:	Adrian Kretz
Industrie-Partner:	Martin Stöck, Matthäus Alberding
Experte:	Wiesner Tobias

Fachbereich Informatik
OST – Ostschweizer Fachhochschule

Inhaltsverzeichnis

Teil I

Einleitung

Abstract

Ziel dieser Bachelorarbeit ist die Prüfung der Machbarkeit einer cloudbasierten Softwarelösung zur Bildstabilisierung. Der Fokus liegt bei der Stabilisierung eines Sport Video Streams in der Cloud, der von einer beliebigen Quelle kommt. Die Verarbeitung soll möglichst zeitnah sein, damit der Benutzer den Stream ohne grosse Verzögerung anschauen kann. Dazu wurde ein Limit von maximal 10 Sekunden Latenz für die Stabilisierung gesetzt. Zusätzlich ist die Lösung mit dem neuen, robusten und schnellen Video Streaming Protokoll SRT (Secure Reliable Transport) umgesetzt worden.

Unterschiedliche Verfahren zur Bildstabilisierung wurden auf ihre Eignung für den Einsatz in der Cloud geprüft. Insbesondere im Hinblick auf Performance, Effizienz und Skalierbarkeit. Hierzu kamen verschiedene Bibliotheken und Algorithmen zum Einsatz, deren Stärken und Schwächen anhand von Prototypen in Python und C++ miteinander verglichen wurden. Die eigentliche Umsetzung erfolgte auf Basis von Python und OpenCV, unterstützt durch Containerisierung für eine flexible und schnelle Bereitstellung. Wegen der breiten Unterstützung für wissenschaftliche Bibliotheken und die Vereinfachung der Entwicklung von Prototypen wurde der Vergleich der verschiedenen Algorithmen in Python durchgeführt. Die vielversprechendsten Algorithmen wurden zusätzlich durch gezieltes Anpassen von Parametern, wie dem Smoothing Window, weiter optimiert, um die Stabilisierung qualitativ so gut wie möglich zu gestalten. Das System wurde so konzipiert, dass mehrere parallele Videostreams verarbeitet werden können, ohne dass es zu grossen Qualitätseinbussen kommt. Die entwickelte Applikation ist darauf ausgelegt, dynamisch skaliert zu werden als Microservice und unterschiedliche Anforderungen an die Verarbeitungstiefe abzudecken. Die Leistungsfähigkeit der Lösung wurde sowohl in lokalen Testumgebungen als auch auf einer cloudbasierten Infrastruktur von AWS validiert. Neben der Bildoptimierung wurde auch Wert daraufgelegt, die anfallenden Kosten und die Skalierbarkeit der Lösung zu analysieren, um einen praxistauglichen Ansatz für den produktiven Einsatz zu schaffen.

Die Resultate der Arbeit zeigen, dass eine Auslagerung der Bildstabilisierung in die Cloud grundsätzlich technisch möglich ist. Allerdings erreicht die entwickelte Lösung hinsichtlich Qualität nicht das Niveau von integrierten Systemen wie beispielsweise einer GoPro Hero 7, die speziell für Bildstabilisierung optimiert wurde. Ein wesentlicher

Faktor ist zudem der hohe Rechenaufwand, der den Betrieb in einer Cloud-Umgebung kostenintensiv macht. Besonders die Verarbeitung mehrerer Videostreams gleichzeitig führt zu einem signifikanten Anstieg des Ressourcenbedarfs. Gleichzeitig lassen die Erkenntnisse aus den Tests darauf schliessen, dass alternative Lösungsansätze, beispielsweise durch die direkte SRT Server Implementierung in C++ mit integrierter Bildstabilisierung, potenziell bessere Ergebnisse in Bezug auf die Latenz liefern könnten.

Management Summary

Ausgangslage

Die Aufgabenstellung für das Projekt ist es, die Machbarkeit von Bildstabilisierung in der Cloud zu untersuchen. Dabei soll geprüft werden, ob eine Auslagerung der Bildstabilisierung auf Cloud Computing eine realistische Lösung darstellen kann, bezüglich der Latenz aber auch den anfallenden Kosten.
Die Qualität der Videos wird für diese Arbeit auf maximal 720p (eine Auflösung von 1280x720 Pixel) festgelegt, was bei der Bildstabilisierung eine einheitliche Vergleichsbasis schafft. Alle erörterten Ergebnisse fallen unter diese Limitierung und Videos mit einer grösseren Auflösung können möglicherweise nicht mit den gewünschten Anforderungen stabilisiert werden.
Als Cloud Provider wird der Hauptfokus auf AWS gelegt, damit für eine allfällige Umsetzung bereits vorhandene Infrastrukturen genutzt werden können. Falls jedoch alternative Anbieter einen grossen Vorteil bezüglich Kosten oder anderen Optionen bieten, dürfen diese in Betracht gezogen werden.

Vorgehen

Zu Beginn wurde eine Literaturrecherche durchgeführt, um einen Überblick über die Thematik zu erhalten. Dabei wurden verschiedene Algorithmen und deren Vor- und Nachteile untersucht, sowie bestehende Bibliotheken, Projekte oder ähnliche Umsetzungen in Betracht gezogen und ausgewertet.
Auf Basis dieser Informationen wurde ein Konzept erstellt, welches vorsieht, dass zuerst Prototypen lokal erstellt werden, um die verschiedenen Algorithmen zu testen und deren Vor- und Nachteile zu evaluieren. Auch soll ein Algorithmus als Python sowie C++ Version erstellt werden, um die Performance der beiden Sprachen und deren Implementierungen der Bibliotheken zu vergleichen.
Aufgrund der Prototypen ist die Entscheidung gefallen, dass mit Python und der Bibliothek OpenCV die Umsetzung für die Cloud Version erfolgen soll. Dies aufgrund der einfacheren Handhabung und der guten Performance der Bibliothek.
Da in AWS ein EC2 Instance mittels einer Vorlage erstellt werden kann, wäre das aus Sicht der Skalierbarkeit eine einfache Option, bei Bedarf, zu skalieren.

Die Container Umsetzung, um eine einfache und schnelle Skalierbarkeit zu ermöglichen, konnte leider nicht mit AWS ECS Service getestet werden. Ein Container-Image ist jedoch erstellt worden und kann für zukünftige Tests verwendet werden.

Research

Der Fokus der Literaturrecherche liegt bei den bestehenden Algorithmen und Bibliotheken, welche für die Bildstabilisierung verwendet werden können. Da AWS als Cloud Provider gesetzt ist, sind auch die bestehenden Cloud Services von AWS (siehe Abschnitt 4.1) in Betracht gezogen worden.
Die Algorithmen wurden in zwei Gruppen unterteilt. Die erste Gruppe sind Algorithmen, welche in Echtzeit verwendet werden können, und somit für die Cloud Version in Betracht kommen. Die zweite Gruppe sind Algorithmen, welche nicht in Echtzeit verwendet, werden können und somit für dieses Projekt nicht in verwendet werden.
Für die Aufgabenstellung gibt es zum Zeitpunkt der Bachelorarbeit keine komplette Lösung, welche alle Anforderungen erfüllt.

Prototypen

Die Prototyen sind kurze Programme, welche in Python geschrieben sind, um die Theorie von den Algorithmen in der Praxis zu testen. Dies bietet eine ideale Möglichkeit schnell und einfach zu evaluieren, ob die gewählte Strategie funktioniert und zeigt rasch auf, ob es grundsätzlich Performanz mässig möglich ist, die gewählten Algorithmen in der Cloud zu verwenden.
Auch sind grundlegende Tests mittels Prototypen gemacht worden, was das Encoding und Decodieren von Streams angeht. Da es keine Bibliothek für Python gibt, welche direkt einen SRT-Stream decodieren kann, wurde dies mittels Prototyen und FFmpeg getestet. Dies ist eine grundlegende Voraussetzung, um die Bildstabilisierung in der Cloud zu ermöglichen und hätte bei keinem Erfolg den Wechsel zu C++ zur Folge gehabt.

Python Implementation

Nach den Prototypen kam die eigentliche Implementation des Projektes. Dieses ist modular aufgebaut und lässt sich durch Befehlszeilen steuern.
Somit konnte der Vergleich verschiedener Algorithmen in kurzer Zeit erfolgen und gibt auch die flexibilität für die Skalierung. Ausserdem wurden noch mehrere visuelle Unterstützungstools dazu genommen wie ein Trajektorie Plotter und auch eine Veranschaulichung der gefunden Merkmale in einem Video.

Cloud-Architektur

Die Cloud-Architektur ist so einfach wie möglich gehalten und wie in der Abbildung 1 dargestellt, bestehend aus zwei IVS-Kanäle (Up-Stream und der stabilisierte Down-Stream) und einer EC2 Instanz dazwischen.

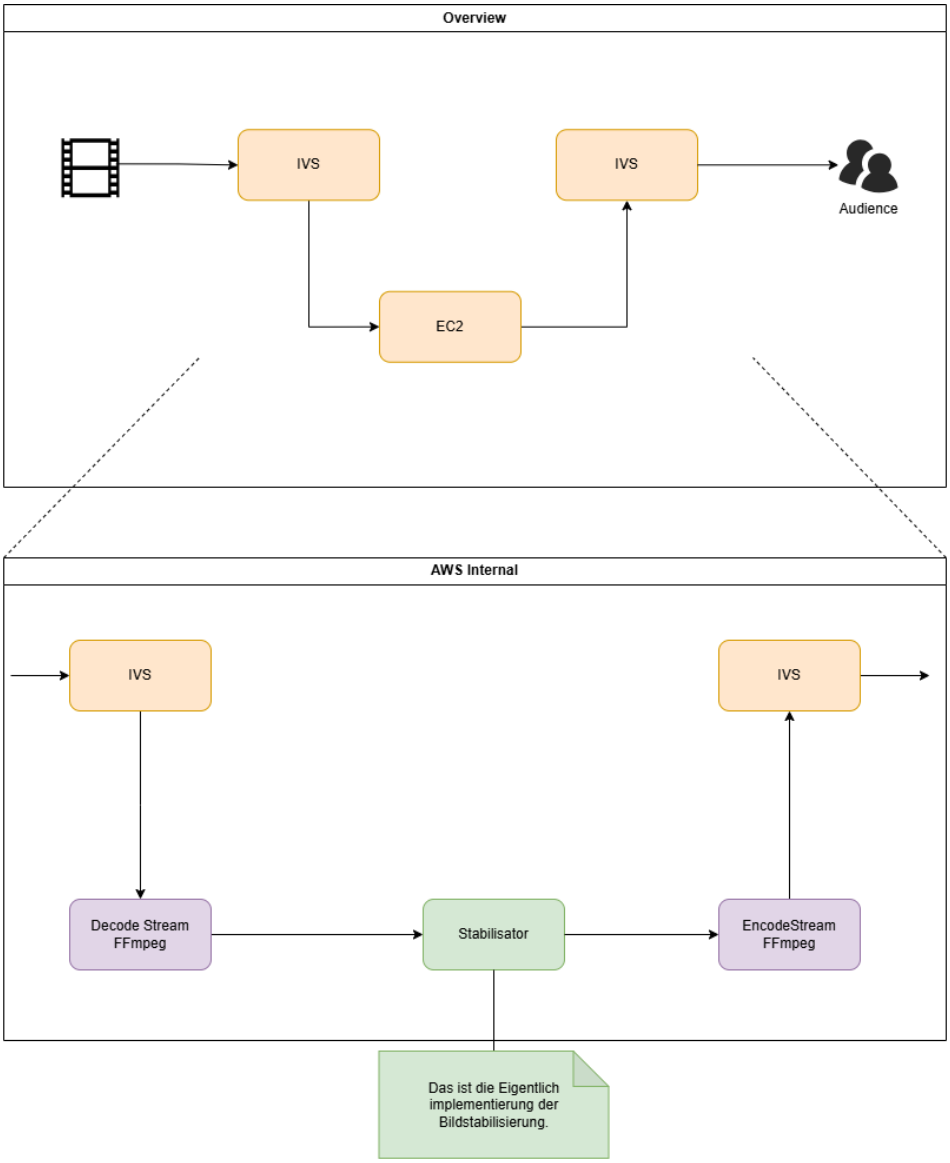


Abbildung 1: Cloud-Architektur

Der Original-Stream wird über den ersten IVS-Kanal in die Cloud übertragen, wo er von einer EC2-Instanz empfangen und verarbeitet wird. Nach der Stabilisierung des Streams erfolgt die Weiterleitung über einen zweiten IVS-Kanal.

Der stabilisierte Stream kann von dort aus beliebig abgerufen werden, wobei die IVS-Infrastruktur automatisch eine skalierbare Bereitstellung für beliebig viele Clients gewährleistet.

Auswertung

Mit der Good Features to Track Methode können sehr gute Merkmale in einem Bild gefunden und dann mit Lucas-Kanade den Bewegungsfluss berechnet werden.
Bei den Glättungslgorithmen sollte jeweils der Bestmögliche Algorithmus für ein spezifisches Szenario ausgewählt werden. Die im Projekt untersuchten Algorithmen zeigten, dass Gauss und Konvolution ein sehr solides Resultat liefern. Ausserdem kann auch auf das Smoothing der Trajektorie verzichtet werden und direkt mit der Trajektorie das Bild stabilisiert werden.

Ergebnisse

Die Stabilisation war ein Erfolg und das resultierende Video zeigt eine verbesserte Qualität. Jedoch ist die Qualitätserhöhung nicht ansatzweise auf einem Level wie zum Beispiel bei der GoPro.
Auch sind die Kosten für die Stabilisation in der Cloud verglichen zum Streaming-Service sehr hoch, weswegen evaluiert werden muss, ob es sich lohnt, eine cloudbasierte Stabilisation überhaupt einzubauen.
Damit die Bilder von den Algorithmen flüssig verarbeitet werden können, wird etwas mehr als 4 CPU-Cores und auch mehr als 4 GB benötigt. Die entsprechende EC2 Instanz verursacht jedoch im Verhältnis zu dem IVS Service über 90% der Kosten pro Stunde (siehe Unterabschnitt 6.6.5).

Ausblick

Für die Zukunft bieten sich mehrere vielversprechende Optimierungsansätze an.
Der gezielte Einsatz unterschiedlicher Algorithmen könnte die Bildstabilisierungsqualität weiter steigern.
Eine szenarienabhängige Konfiguration, beispielsweise für Selfie-Videos oder statische Kameras, verspricht zusätzlich verbesserte Ergebnisse. Auch eine Parallelisierung im bestehenden Queue-System wäre möglich, um die Verarbeitungsgeschwindigkeit zu erhöhen, wobei die Wirtschaftlichkeit sorgfältig geprüft werden muss.
Der Einsatz von Gyrodaten bietet die Perspektive, Rechenaufwand zu reduzieren und damit Kosten zu senken.
Zudem kann geprüft werden, ob ein eigener SRT-Server für den Up-Stream die bisherige IVS-Infrastruktur ersetzen kann, um die Latenz signifikant zu verringern.

Aufgabenstellung

Im Rahmen der Bachelorarbeit, kurz BA, für die Ostschweizer Fachhochschule (OST) soll untersucht und prototypisch umgesetzt werden, inwiefern sich die Bildstabilisierung von Videostreams bei Sportveranstaltungen in die Cloud auslagern lässt.

Ziel ist es, durch die Entkopplung der Bildstabilisierung vom Endgerät (z.B. Action-Cams oder Smartphones) eine längere Akkulaufzeit und eine höhere Geräteflexibilität zu ermöglichen, ohne die Qualität des Streams wesentlich zu beeinträchtigen.
Konkret soll eine Lösung entwickelt werden, die Videostreams in die Cloud (idealerweise über AWS) überträgt, dort stabilisiert und dem Endnutzer mit geringer Latenz wieder zur Verfügung stellt. Dabei sollen verschiedene Streaming-Protokolle unterstützt und eine stabile Videoqualität gewährleistet werden.
Die technische Umsetzung erfolgt mit modernen Cloud- und Container-Technologien.

Teil II

Produkt Dokumentation

Kapitel 1

Domainanalyse

1.1 Domain

Die Problem-Domäne hat den Fokus auf der Stabilisierung von Bildern, bzw. einem Video oder Video-Stream. Ein Video kann grundsätzlich von dem Aufnahmegerät direkt stabilisiert werden, was zur Folge hat, dass alle Videos-Streams eine unterschiedliche Qualität der Bilder aufweisen. Auch kann es sein, dass ein Aufnahmegerät nicht über eine Stabilisierung verfügt, was zu einem Verwackelten Bild führt.
Um dem entgegenzuwirken, sowie eine Verlängerung der Batterielaufzeit zu erzielen, soll geprüft werden, ob eine Auslagerung der Bildstabilisierung auf Cloud Computing eine machbare Lösung darstellen kann.
Die Bildstabilisierung soll möglichst in Echtzeit erfolgen, was bedeutet, dass nur Algorithmen in Frage kommen, welche eine geringe Latenz aufweisen und mindestens 30 Bilder pro Sekunde stabilisieren können.

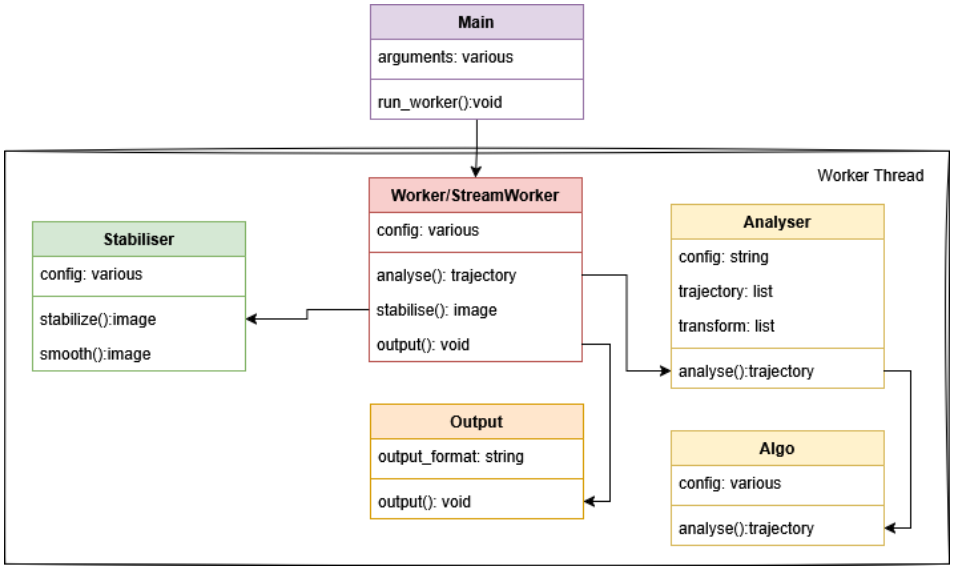


Abbildung 1.1: Domain Analyse

Die Klassen in Abbildung 1.1 haben folgende Aufgaben sowie Anforderungen. Das Programm und deren Klassen sowie Werte haben nur während der Ausführung ihre Laufzeitumgebung ihre Gültigkeit. Es werden keine Daten persistiert, sondern nur verarbeitet. Somit entfällt ein allfälliges Datenbankmodell.

- Main: Hier wird der Stream Thread/Prozess gestartet mit der entsprechenden Konfiguration.
- Worker/StreamWorker: Das ist die Hauptklasse, welche ein Bild mit der Unterstützung der anderen Klassen durcharbeitet.
- Analyser: Analysiert ein Bild und berechnet die Trajektorie.
- Algo: Berechnet die Transformation von nachfolgenden Bildern.
- Stabiliser: Hier wird die Trajektorie gesmoothed und das Bild stabilisiert.
- Output: Das erfolgreich Stabilisierte Bild wird hier and das gewünschte Format ausgegeben.

1.2 Architektur

Die Cloud Architektur des Programms ist in der Abbildung 1 dargestellt, wo die zwei IVS-Kanäle sowie die EC2 Instanz zu sehen sind.
Der Aufbau ist der folgende:

1. Der Original-Stream wird über den ersten IVS-Kanal in die Cloud übertragen.

2. Der Stream wird von der EC2-Instanz empfangen und verarbeitet. Dieser wird im Abschnitt 1.3 genauer erklärt.

3. Nach der Stabilisierung des Streams erfolgt die Weiterleitung über einen zweiten IVS-Kanal.

Danach kann der stabilisierte Stream von dort aus beliebig abgerufen werden.
Für die Entwicklung und das Testen wurde die Architektur auch in vereinfachter Form auf einem lokalen Rechner verwendet. Um die IVS API von AWS lokal zu simulieren, haben wir einen Container namens srt-live-server(SLS) verwendet, welcher ausführlicher im Unterabschnitt 4.2.1 erklärt wird.
Dieser Container wird als lokaler SRT-Server verwendet, welches die IVS API lokal simuliert und die Entwicklung sowie das Testen der Applikation lokal ermöglichen. Dies erlaubt es, dieselbe Architektur wie in der Cloud lokal zu simulieren und somit die Entwicklung und das Testen zu vereinfachen.

1.3 Programmfluss

Der Programmfluss (Abbildung 1.2) ist in drei unterschiedliche Teile unterteilt: Input, Stabiliser, Output. Zuerst wird der Stream konsumiert und die Daten zu einem Bild decodiert und an eine Queue weitergegeben. Als nächstes wird das Bild von der Queue genommen, Analysiert, falls nötig die Transformation geglättet und auf das Bild angewendet. Zum Schluss wird das Stabilisierte Bild in eine neue Queue gesendet. Der Letzte Teil nimmt das Stabilisierte Bild, encodiert es und schickt es an den verlangten Ort. Diese Parallelisierung erlaubt es, Arbeiten gezielt auszuführen und bei allfälligen kurzen Unterbrüchen eine möglichst stabile Verbindung aufrechtzuerhalten. Für den Lokalen Test wurde nur der Stabilisationsteil verwendet und simple Input und Output Methoden noch hinzugefügt.

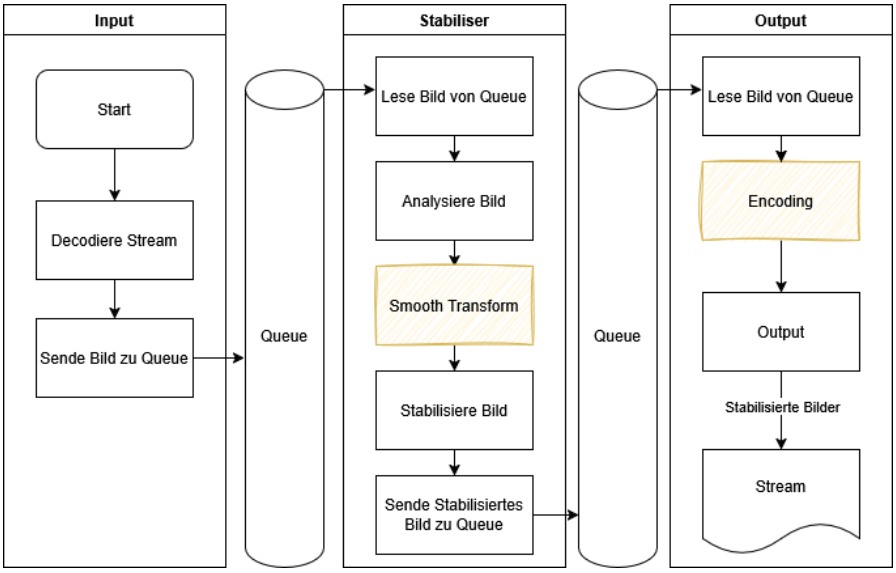


Abbildung 1.2: Programmflussdiagramm

1.3.1 Input

Beim Input wird der Stream konsumiert, decodiert und in "reine" Bilddaten umgewandelt, damit diese weiterverarbeitet werden können.

Start

Der eigentliche Start des Programms erfolgt über den init Handler, welcher dann die verschiedenen Threads/Prozesse startet, sowie die Konfigurationen für den Stream festlegen. Alle Einstellungen werden als Argumente an das Programm übergeben und werden beim Starten des Programms gesetzt.

Decodiere Stream

Der Stream wird mit dem Programm FFmpeg konsumiert und auch decodiert, um die einzelnen Bilder zu extrahieren. Die Bilder werden in das "raw" Videoformat decodiert, welches nur die Bilddaten und keine Metadaten oder zusätzliche Komprimierung enthält. Anschliessend werden die reinen Bilddaten in eine Queue geschrieben für die weitere Verarbeitung.

1.3.2 Stabiliser

Der Stabiliser ist der Kern des Projektes, bei welchem ein Bild genommen wird und das stabilisierte Bild, der Konfiguration entsprechend, weitergegeben wird.

Analysiere Bild

Die besten Merkmale eines Bildes werden gesucht und mit den Merkmalen des vorherigen Bildes verglichen, um daraus die Transformation und Trajektorie zu berechnen.

Smooth Transform

Die Transformation und Trajektorie werden gegebenenfalls geglättet um ein weicheres Ergebnis zu erhalten.

Stabilisiere Bild

Das Einzelbild wird mithilfe der berechneten Transformationsmatrix angepasst. Anschliessend erfolgt ein leichter Zoom, um das Auftreten schwarzer Ränder im Bild zu vermeiden.

1.3.3 Output

Der Output ist der letzte Teil des Programms, welcher das Stabilisierte Bild nimmt und es encodiert und an den verlangten Ort sendet.

Encoding

Nach der Verarbeitung des Bildes, müssen die Bilddaten wieder in ein Videoformat encodiert werden. Dies geschieht mit dem Programm FFmpeg, welches die Bilddaten in ein Videoformat encodiert. Die Bilder kommen mit dem Format mjpeg, welches zu einem mpegts Stream encodiert wird. Dies ist hartcodiert, da es mit dem AWS Service IVS kompatibel sein muss.

Teil III

Projekt Dokumentation

Kapitel 2

Projektplan

2.1 Ablauf

Wir entschieden uns für die Benutzung von Scrum+. Dabei handelt es sich um eine Kombination von Scrum mit dem Rational Unified Process (RUP). Grund dafür ist die Möglichkeit, eine Langzeitplanung des RUP mit der Kurzzeitplanung von Scrum kombinieren zu können. Das ermöglicht uns, einen groben Zeitplan zu erstellen, aber immer noch genügend Freiraum für allfällige Änderungen zu haben.

2.1.1 Iteration

Die Iterationen der einzelnen Sprints wurden in zwei Wochenabständen gehalten. Dies ermöglichte es uns, jeweils am Ende eines Sprints, ein Meeting mit den Stakeholdern zu führen.

2.1.2 Zeiterfassung und Tickets

Die Zeiterfassung und Tickets werden über Jira gemanagt. Die Zeiten werden auf 15 Minuten gerundet.

2.1.3 Rollen

Scrum Master

Überblick über die Aufgaben des Scrum Masters:

- Organisiert Scrums und Sprints
- Leitet Scrum Meetings
- Überprüft das Meetingprotokoll
- Unterstützt den Product Owner mit der Verwaltung des Backlogs

Diese Rolle wird übernommen von: Cedric Christen

Product Owner

Überblick über die Aufgaben des Product Owners:

- Verwaltung des Backlogs mit Priorisierungen
- Meetings organisieren
- Kommuniziert mit dem Berater und Stakeholdern
- Übergibt die Dokumente zeitgerecht

Diese Rolle wird übernommen von: David Stäheli

Project Manager

Überblick über die Aufgaben des Project Managers:

- Überprüft, dass die Leitfäden eingehalten werden
- Überprüft, dass die Qualität zufriedenstellend ist

Diese Rolle ist aufgeteilt in zwei Unterrollen:

- Code-Manager ist Cedric Christen
- Infrastruktur-Manager ist David Stäheli

Entwickler

Überblick über die Aufgaben des Entwicklers:

- Arbeitet an Arbeitspunkten

Diese Rolle wird übernommen von: Beiden

2.2 Leitfaden für den Code

2.2.1 Python

Die Prototypen und die allgemeine Implementierung sind beide in Python geschrieben, weswegen der Leitfaden für beide Module der Gleiche ist. Der Leitfaden bezieht sich auf PEP 8 - Style Guide for Python Code.[26] Die Formatierung des Codes wird mit dem Tool black [10] durchgeführt.

Die wichtigsten Regeln sind:

- Snake case für Methoden, Dateinamen und Variablen.
- Camel case für Klassen.
- Einrückung von 4 Leerschlägen.

Dies wird überprüft mit dem ruff[11] Linter Check in der Pipeline.

- Konfigurationsdateien werden von PEP 8 ignoriert.
- Zeilenlänge wird auf 88 Zeichen gesetzt.
- Der erste Prototyp wird ignoriert

Die genaue Konfiguration kann in pyproject.toml gefunden werden.

2.2.2 C++

Für Performancevergleiche wird auch ein C++ Prototyp umgesetzt. Der Leitfaden bezieht sich grundsätzlich auf Google C++ Style Guide[8], kann jedoch davon abweichen, da für die Prototypen die Funktionalität des Codes im Vordergrund stand.

Die wichtigsten Regeln sind:

- Snake case für Dateinamen und Variablen.
- Camel case für Klassen, Methoden.
- Einrückung von 4 Leerschlägen.

Da das C++ Projekt nur für den Performancevergleiche verwendet wird, wurde auf die Überprüfung verzichtet. Jedoch wäre das Tool clang-format [24] Linter Check ideal um in einer Build Pipeline einzubinden und die Überprüfung durchzuführen.

2.3 Leitfaden für die Dokumentation

Der Leitfaden bezieht sich auf WRITE THE DOCS.[25]
Dies bedeutet, dass zum Beispiel Hyperlinks direkt eingefügt werden sollen und nicht als “Click here“ oder ähnliches. Die Sprache sollte ausserdem einheitlich sein in der Dokumentation.

2.4 Phasen

Die Phasen der Implementierung und Evaluierung wiederholen sich für die Lokalen und die AWS-Applikation.

1. Inception 17.02.2025 - 02.03.2025
2. Elaboration 03.03.2025 - 09.03.2025
3. Implementation 10.03.2025 - 06.04.2025
4. Evaluation 07.04.2025 - 27.04.2025
5. Implementation 28.04.2025 - 25.05.2025
6. Evaluation 26.05.2025 - 08.06.2025
7. Abgabe 09.06.2025 - 13.06.2025

2.5 Meilensteine

Meilensteine sind jeweils auf den Freitag gesetzt.

M1 Prototyp

Datum: 16.03.2025
Ziel:

1. PoC erstellt
2. Grober Zeitplan erstellt

M2 Lokale Applikation

Datum: 27.04.2025
Ziel:

1. Lokale Applikation erstellt
2. Vergleiche von Algorithmen gemacht

M3 AWS Server Applikation

Datum: 01.06.2025
Ziel:

1. AWS-Stabilisierung läuft
2. Tests von AWS und Lokal gemacht

2.6 Geplante Releases

Lokale Applikation

Release Date: 27.04.2025
Die Lokale Applikation beinhaltet folgende Funktionalitäten:

- Verschiedene Modulare Algorithmen zum Austauschen und Vergleichen.
- Verschiedene Videos zum Testen

AWS Release

Release Date: 01.06.2025
Der Final Release setzt den vorherigen Release in die AWS-Umgebung:

- Einspeisung in AWS-System
- Streaming möglich von A bis Z

2.7 Langzeitplan / Roadmap

Das Projekt hat am 17.02.2025 angefangen und wird am 13.06.2025 mit der Abgabe der Dokumentation und der Präsentation abgeschlossen. Der Plan ist folgender, wie in Abbildung 2.1 beschrieben:

Abbildung 2.1: Roadmap

Kapitel 3

Qualitätsmessung

3.1 Definition of Done

Die folgenden Kriterien beschreiben unsere Definition, wann eine Arbeit als fertiggestellt definiert wird.

- Erwartete Funktionalitäten sind vorhanden.
- Alle automatisierten Tests sind erfolgreich.
- Alle Qualitätssicherungen sind durchlaufen.
- Dokumentation, Projektplan und Zeiterfassung sind auf dem neuesten Stand.

3.2 Code Qualität

Mittels linting und testing wird die Qualität des Codes sichergestellt. Dies wird durch die CI/CD Pipeline automatisiert durchgeführt, welche bei jedem Push, Pull Request oder Tag ausgeführt wird.

3.2.1 Qualitätssicherungen

Alle Qualitätssicherungen sind durchlaufen und die folgenden Kriterien wurden erfüllt:

- Ruff Codequalitätstest bestanden.
- Black Codeformatierung check bestanden.

3.3 Test Konzept

Das Testkonzept beschreibt die Teststrategie, Testtechniken, Testumgebung, Testergebnisse.

3.3.1 Funktionalitäten zum Testen

- Stabilisierung von Videos
- Stabilisierung von Video-Streams

3.3.2 Test Strategie

Die Strategie für das Testen beinhaltet folgende Punkte:

- Identifizierung der Testfälle.
- Ausführung der Tests. Die Tests werden manuell durchgeführt.
- Reporting von Fehlern und Testergebnissen.
- Wiederholung der Tests, bis alle Fehler behoben sind.

3.3.3 Test Techniken

Folgende Testtechniken werden in unserem Projekt verwendet:

- Automatisch
  - Build-Tests
- Manuell
  - Systemtests
  - Integrationstests
  - Akzeptanztests

3.3.4 Test Umgebung

3.3.5 Lokal

Als lokale Testumgebung wurden 2 verschiedene Geräte verwendet, welche beide Windows 11 als Betriebssystem haben.

Gerät 1

Wurde für das Testen der Performance der Algorithmen verwendet.

- OS: Windows 11 Home
- Processor: 12th Gen Intel(R) Core(TM) i7-12700H 2.70 GHz
- Installed RAM: 16.0 GB (15.7 GB usable)
- System type: 64-bit operating system, x64-based processor
- VRAM: 4 GB
- Python: Version 3.10.11

Gerät 2

Wurde für den Performance vergleich von C++ und Python verwendet.

- OS: Windows 11 Pro
- Processor: Intel(R) Core(TM) Ultra 9 185H 5.1GHz
- Installed RAM: 32.0 GB (31.6 GB usable)
- System type: 64-bit operating system, x64-based processor
- Shared VRAM: 18 GB
- Python: Version 3.11.4

3.3.6 Server

Die Applikation ist für den Betrieb als Container oder direkt auf einem Server ausgelegt. Für die Ausführung auf einem Server sind folgende Voraussetzungen nötig:

- OS: Ubuntu 24.04.2 LTS
- Python: Version 3.12.3
- FFmpeg: Version 6.1.1
- Poetry: Version 2.1.3

Die Applikation wurde auf einem AWS EC2 Server mit Ubuntu 24.04.2 LTS getestet und funktioniert einwandfrei. Für die Tests wurde Poetry via pipx installiert, um die benötigten Abhängigkeiten zu installieren und die Applikation zu starten.

3.3.7 Container

Die Container-Umgebung wurde mit Docker und Docker Compose getestet und ist für die Ausführung auf einem Server oder lokal ausgelegt.

Lokale Container-Umgebung

- Docker Desktop: Version 4.42.0
- Docker: Version 28.2.2
- Docker Compose: Version 2.36.2-desktop.1

3.3.8 Test-Ergebnisse

Die Testergebnisse spielen eine zentrale Rolle in diesem Projekt und werden daher ausführlich in Kapitel 6 dargestellt.

Kapitel 4

Research

4.1 AWS

Der Videostream ist mit der aktuellen AWS-Architektur eingerichtet, daher ist der bevorzugte Cloud-Anbieter für dieses Projekt AWS. Alle recherchierten Dienste haben den Hauptfokus auf AWS, und andere Dienste werden nur in Betracht gezogen, wenn sie explizit erwähnt werden und entweder Kostenvorteile oder bedeutende andere Vorteile gegenüber AWS bieten.

4.1.1 Aktuelles Setup

Das aktuelle Setup verwendet den Amazon IVS Service für das Video Streaming.
Aktuell wird nur ein Kanal verwendet, welcher als Up- und Down-Stream dient.

4.1.2 Mögliche Stabilisierungsanwendung

Es gibt zwei verschiedene für das Projekt relevante Varianten, um den Stream zu verarbeiten.

AWS Media Services

Die AWS Media Services zur Stabilisierung des Streams zu nutzen, wäre sozusagen eine built-in Variante.
Jedoch gibt es keinen Dienst, der den Stream stabilisieren kann. Aktuell sind nur Dienste verfügbar, die den Stream in ein anderes Format, unterschiedliche Qualitätsstufen konvertieren oder einen Filter darauf anwenden. Daher wird diese Variante nicht weiterverfolgt, da es keine Möglichkeit gibt, den Stream zu stabilisieren oder einen selbst entwickelten Programmcode darauf anzuwenden.

EC2 oder ECS Instanz

Durch die Verwendung von AWS EC2- oder ECS-Instanzen können wir den Stream mit einem selbst entwickelten Stabilisierungsalgorithmus stabilisieren.
Nachdem ein Videostream beim IVS-Dienst eingetroffen ist, wird der Stream von einer EC2- oder ECS-Instanz konsumiert. Diese Instanz kann dann den Stream stabilisieren und zurück an den IVS-Dienst senden, der ihn anschliessend an die Zuschauer weiterleitet.
Da das Verarbeiten von Video Streams sehr rechenintensiv ist, soll die EC2-Instanz auch dieser Anforderung gerecht werden und auf CPU und RAM optimiert sein. Am besten dafür eignen sich EC2-Instanzen der C5i oder C6i Serie [1], welche auf hohe Rechenleistung optimiert sind, wie unteranderem Videokodierung. Somit wird eine EC2-Instanz der C6i Serie verwendet, welche 8 vCPUs und 16 GB RAM hat und auf Linux basierend ist.

4.2 Video Stream

Für den Prototypen wird OBS Studio verwendet, um den Video-Stream von der Webcam zu erstellen, mit welchem der Stabilisierungsalgorithmus getestet werden kann. OBS-Studio ist eine optimale Wahl als Streaming-Software, da es eine Vielzahl von Protokollen unterstützt, was die Möglichkeit gibt, verschiedene Streaming-Protokolle zu testen. Damit der Stream auch Lokal getestet werden kann, wird der Stream an den SRT-Server gesendet. Danach kann dieser von unserem Algorithmus stabilisiert und zurück an den Streaming-Server gesendet, lokal gespeichert oder direkt via OpenCV angezeigt werden.

4.2.1 Stream Server

Basierend auf unserer Recherche fiel die Wahl auf das GitHub-Projekt srt-live-server, mit dem wir den Stream zu AWS IVS simulieren. Dies ist eine OpenSource Container-Lösung basierend auf dem Code von Edward-Wu, in kurz auch SLS gennant. Der SLS ist ein Streaming-Server, der das SRT-Protokoll (Unterabschnitt 4.2.2) verwendet, um beispielsweise ein Video-Stream zu übertragen.
Mit der dockerisierten Version kann somit einfach Lokal ein SRT-Server gestartet werden, welcher den Stream von OBS Studio empfängt und von einem Client wie unserem Algorithmus oder einem Player wie VideoLAN Client wiedergegeben werden kann.
Die Implementierung des OpenSource SLS hat folgende Eigenheiten, welche abweichend von der IVS API sind:

- Der Up-Stream ist immer über /input/live/<stream_name > erreichbar.
- Der Down-Stream ist immer über /output/live/<stream_name > erreichbar.
- Der Standard-Port ist 1935, kann aber über die Umgebungsvariable PORT angepasst werden.

4.2.2 SRT Protokoll

Das SRT-Protokoll [12] (Secure Reliable Transport) ist ein offenes Transportprotokoll zur zuverlässigen und latenzarmen Übertragung von Audio- und Videodaten über ein Netzwerke wie das Internet. Es basiert auf UDP, ergänzt um Fehlerkorrektur, Verschlüsselung und einstellbare Pufferung, um auch bei Paketverlust eine stabile Übertragung zu gewährleisten. SRT ist dabei formatunabhängig und kann beliebige Datenströme sicher transportieren.

4.2.3 OBS Studio

OBS Studio ist ausgerichtet um einen möglichst ruckelfreien und qualitativ hochwertigen Stream zu erstellen. Dies führt jedoch dazu, dass die Verzögerung des Streams höher ist und sich im Bereich von 2-5 Sekunden befindet.

4.2.4 FFmpeg

FFmpeg wird hauptsächlich verwendet um den Stream zu encodieren und decodieren, da diese Opensource Software eine breite Verwendung hat und sehr viele Codecs implementiert sind. Die Unterstützung für das Encodieren und Decodieren von SRT-Streams ist ebenfalls bereits integriert.
Für lokale Test ist diese Software jedoch perfekt geeignet, um Streams zu erstellen, da die Verzögerung mittels Optimierung der Parameter auf ca. 0.5 Sekunden reduziert werden kann. Dies mittels FFmpeg zu ermitteln war sehr wichtig, da bei OBS Studio alle Möglichkeiten ausgeschöpft wurden, jedoch immer noch Verzögerungen im Bereich von ca. 5 Sekunden vorhanden sind.

```
ffmpeg -f dshow -rtbufsize 512M -i video="Integrated␣Camera":
    audio="Microphone␣Array␣(Intel␣Smart␣Sound␣Technology␣for␣
    Digital␣Microphones)" -vcodec libx264 -preset ultrafast -tune
    zerolatency -x264-params keyint=30:min-keyint=30:scenecut=0 -
    acodec aac -ar 44100 -b:a 128k -pix_fmt yuv420p -f flv rtmp://
    localhost:1935/stream
```

Listing 4.1: Beispiel FFmpeg Stream mit Webcam

4.3 Algorithmus

Hier vergleichen wir verschiedene Algorithmen und was deren Fähigkeiten laut der aktuellen Forschung und Fachzeitschriften ist. Unser Fokus liegt auf Algorithmen, welche der Anforderung für Streaming gerecht werden. Dies bedeutet, dass mindestens 30 FPS erreicht werden müssen, was auch in der Tabelle durch die Spalte Echtzeitfähig wiederspiegelt wird.

Methode	**FPS**	**Echtzeitfähig**	**Hardware**
OIS/EIS (GoPro, DJI)	60+ FPS	Ja	Spezial-Hardware
MeshFlow (FFmpeg/vidstab) [15]	30+ FPS	Ja	CPU-basiert
PwStableNet (STN) [29]	30+ FPS	Ja	GPU benötigt
Optical Flow + Kalman [2]	30-60 FPS	Ja	CPU-basiert
DUT (Deep Learning) [27]	14 FPS	Nein	GPU empfohlen
DIFRINT (Deep Iterative Frame Interpolation) [4]	15 FPS	Nein	GPU

Tabelle 4.1: Vergleich der Stabilisierungsmethoden

4.4 VidStab

VidStab ist eine Bibliothek von FFmpeg-Filtern zur Videostabilisierung, die auf lokal gespeicherte Videodateien angewendet wird. Sie ist darauf ausgelegt, vollständige Videos nachträglich zu verarbeiten und eignet sich nicht für die Stabilisierung von Videodaten, die fortlaufend aufgenommen oder übertragen werden.
Der Prozess der Stabilisierung erfolgt in zwei Schritten: 1. Analyse und 2. Transformation.

1. **Analyse**: Der Vidstabdetect-Filter analysiert die Bewegung im Video und erstellt eine Transformationsdatei.

```
ffmpeg -i VID20250309102654-unstabilized.mp4 -vf
   vidstabdetect=shakiness=10:accuracy=15:show=1 out2.mp4
```

Listing 4.2: Analyse der Videobewegung

2. **Transformation**: Der Vidstabtransform-Filter wendet die Transformationen an, um das Video zu stabilisieren.

```
ffmpeg -i out2.mp4 -vf vidstabtransform=smoothing=30:zoom=5:
   input="transforms.trf" stabilized2.mp4
```

Listing 4.3: Anwendung der Stabilisierung

3. **Vergleich**: Nebeneinanderstellung des Original- und des stabilisierten Videos.

```
ffmpeg -i out.mp4 -i stabilized.mp4 -filter_complex "[0:v:0]
    pad=iw*2:ih[bg];␣[bg][1:v:0]overlay=w" sidebyside.mp4
```

Listing 4.4: Erstellung eines Vergleichsvideos

Parameter-Erklärung:

- `shakiness=10`: Bestimmt, wie stark das Video als verwackelt erkannt werden soll (1-10)
- `accuracy=15`: Genauigkeit der Bewegungserkennung (1-15)
- `smoothing=30`: Stärke der Glättung der Bewegungstrajektorie
- `zoom=5`: Zoom-Faktor in Prozent, um schwarze Ränder zu vermeiden

Der 3. Punkt ist optional und dient dazu, die Ergebnisse zu vergleichen. Dieser Befehl erzeugt ein Video, dass das Originalvideo und das stabilisierte Video nebeneinander anzeigt.

4.5 Merkmale erkennen

Um eine Bildstabilisierung zu erhalten, muss ein Vektor gefunden werden, welcher den Unterschied von zwei aneinander liegenden Bildern darstellt. Um dies zu ermöglichen werden Merkmale von Bildern gesucht und diese dann Bild für Bild miteinander verglichen.

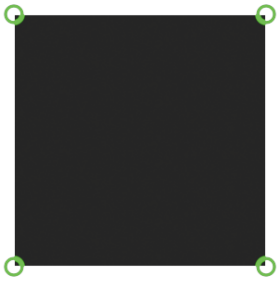


Abbildung 4.1: Merkmale Bild 0

Bei einem schwarzen Quadrat wären das zum Beispiel die Ecken. (Abbildung 4.1) Um Merkmale in einem Bild zu finden, gibt es verschiedene Methoden.

4.5.1 Features from accelerated segment test (Fast)

Der Fast-Algorithmus[13] geht durch jedes Pixel eines Bildes, schaut sich die 16 Pixel Nachbaren in einem Kreis an und berechnet ob es sich dabei um eine Ecke handelt. Falls eine vorgegebene Anzahl der 16 Pixel einen hohen Kontrast aufweisen, verglichen mit dem Ursprungspixel, handelt es sich um eine Ecke. Das führt dazu, dass Fast sehr effizient und schnell arbeitet.
Eine Erweiterung des Algorithmus beinhaltet das Filtern von zu nahen Ecken. Falls sich mehrere Ecken in einem vorgegebenen Radius befinden, kann man diese auf einen Ecken reduzieren um Duplizierungen zu vermeiden.
Der Fast Algorithmus hat aber auch Nachteile. Er findet nicht alle Ecken und hat keine Einstufungen in der Qualität der gefundenen Ecken.
Mit KI kann der Algorithmus noch weiter verbessert werden, was jedoch zu einer Verlangsamung führt.

4.5.2 Harris

Der Harris-Algorithmus[18] berechnet die Horizontale und Vertikale Ableitungen, filtert die Komponente mit einem Kreisförmigen 5 zu 5 Gaussian Filter und berechnet eine Metrik, welche die Qualität des Eckens aufzeigt. Mit der Qualität kann ein Schwellenwert gesetzt werden, um nur qualitativ höhere Ecken zu behalten.
Harris ist weniger effizient und langsamer als der Fast Algorithmus, jedoch findet er mehr Ecken und schlechte Ecken können herausgefiltert werden. Nachteil von Harris ist, dass die Parameter je nach Umgebung angepasst werden müssen.

4.5.3 Scale-invariant feature transform (SIFT)

Der SIFT[16] Algorithmus ist ein sehr komplexer Algorithmus und wird darum in vereinfachter Form erklärt.
Zuerst wird das Bild mehrfach verkleinert und gleichzeitig unscharf gemacht. Dadurch soll erreicht werden, dass gefundene Merkmale, welche nur eine kleine Einwirkung auf das ganze Bild haben, rausgefiltert werden können. Danach werden die Ableitungen berechnet wie beim Harris Filter, aber für jede Verkleinerungsform. Diese Ableitungen werden danach miteinander verglichen und Merkmale gefunden.
Der SIFT-Algorithmus ist ein sehr mächtiger Algorithmus, jedoch langsamer als Harris oder Fast. Der Zusatznutzen von SIFT ist die Invarianz zu Koordinatentransformationen wie Lokation, Translation und Rotation. Dieser Zusatz wird jedoch nicht für das Projekt benötigt.

4.6 Bewegungsfluss (en. Optical Flow) Berechnen

Nachdem die Merkmale gefunden wurden, muss der Bewegungsfluss berechnet werden. Dabei handelt es sich um den Vektor, welcher die Differenz von Bild 1 und Bild 2 darstellt.

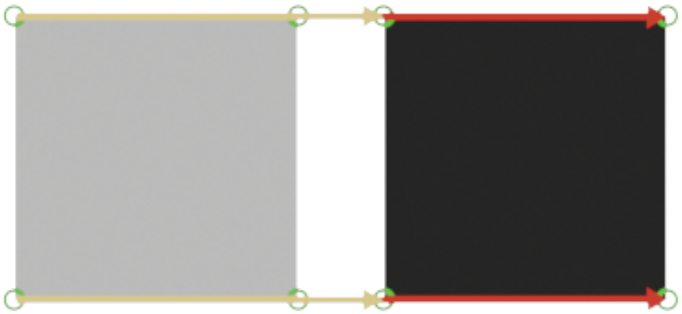


Abbildung 4.2: Transform Bild 0 zu Bild 1

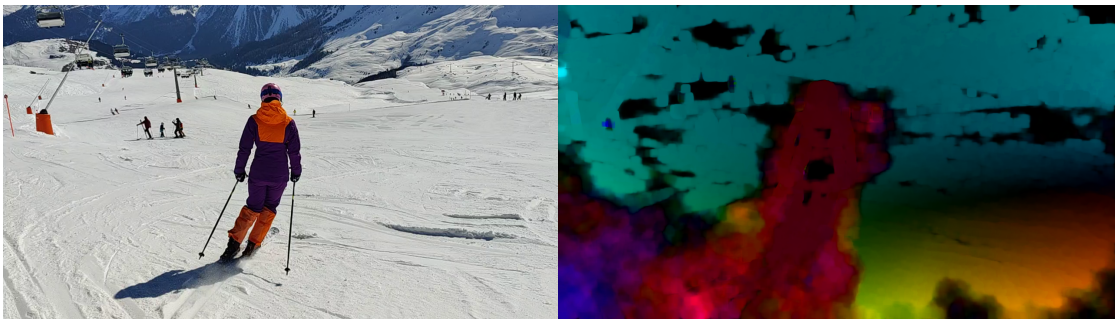
Falls sich das schwarze Quadrat um einige Pixel nach rechts bewegt, bewegen sich die neuen Merkmale genau so weit. (Abbildung 4.2) Somit gibt es nun zwei Sets von Merkmalen. Die Merkmale von Bild 0 und die Merkmale von Bild 1. Vergleicht man diese Merkmale miteinander, kann die Transformation berechnet werden, was den Vektor von der Differenz darstellt. Für die Berechnung testeten wir zwei verschiedene Methoden.

4.6.1 Lucas-Kanade Methode

Die Lucas-Kanade Methode[17] nimmt an, dass zwei aufeinanderfolgende Bilder die gleichen Merkmale aufzeigen und dadurch der Bewegungsfluss berechnet werden kann. Mithilfe einer Ableitung werden die Merkmale von Bild 0 genommen und in der näheren Umgebung nach den gleichen Merkmalen Ausschau gehalten. Das Resultat liefert dann eine Liste von Merkmalen von Bild 1, welche auch in Bild 0 gefunden wurden. Die Lucas-Kanade Methode liefert keinen dichten Fluss. Der Vorteil davon ist, dass es Robuster gegenüber Rauschen ist.

4.6.2 Farneback

Der Farneback-Algoritmus[5] ist ein sehr komplexer Algorithmus und wird vereinfacht erklärt. Verglichen mit der Lucas-Kanade Methode benötigt er keine Vordefinierten Merkmale, da der Algorithmus sich alle Pixel anschaut. Zuerst arbeitet er ähnlich wie der SIFT-Algorithmus indem er die Bilder Verkleinert um auch grössere Bewegungen besser zu erkennen. Danach berechnet er wie sich ein Pixel verglichen zum Nachbar verändert haben könnte. Also falls eine Person im ersten Bild in der Sonne ist und dann im nächsten den Schatten betrat, verändert sich die Farbe des Pixels. Danach versucht er jeden Pixel mit leichten Bewegungen zum gleichen Ort zu bewegen, wie im vorherigen Bild. Dies ermöglicht es, von jedem Pixel einen Vektor zum vorherigen Bild zu bekommen.



(a) Original Bild (b) Dichter Bewegungsfluss

Abbildung 4.3: Dichter Bewegungsfluss Vergleich

Das Ergebnis ist ein dichter Fluss. Der Farneback-Algorithmus ist ein sehr Hardware intensiver Algorithmus, welcher mit der Hilfe von GPUs beschleunigt werden kann.

4.6.3 Transformation VS Trajektorie

Bisher wurde nur die Transformation angeschaut, welche die Differenz von einem Bild zum nächsten darstellt. Falls es jedoch mehr als nur zwei Bilder gibt, reicht das nicht aus.

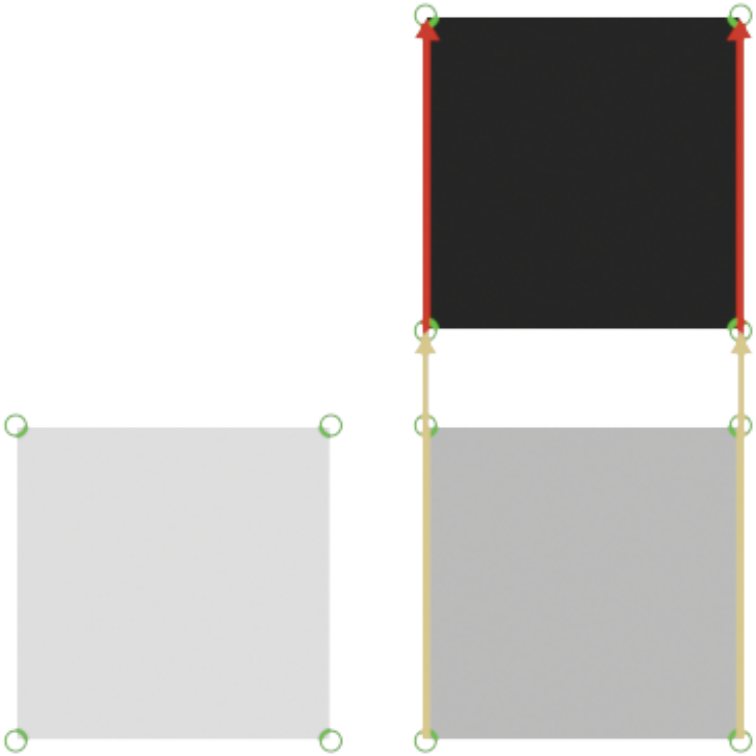


Abbildung 4.4: Transform Bild 1 zu Bild 2

Die erhaltene Transformation von Bild 1 zu Bild 2 sagt aus, dass sich das Quadrat um einige Pixel nach oben bewegt hat. (Abbildung 4.4)

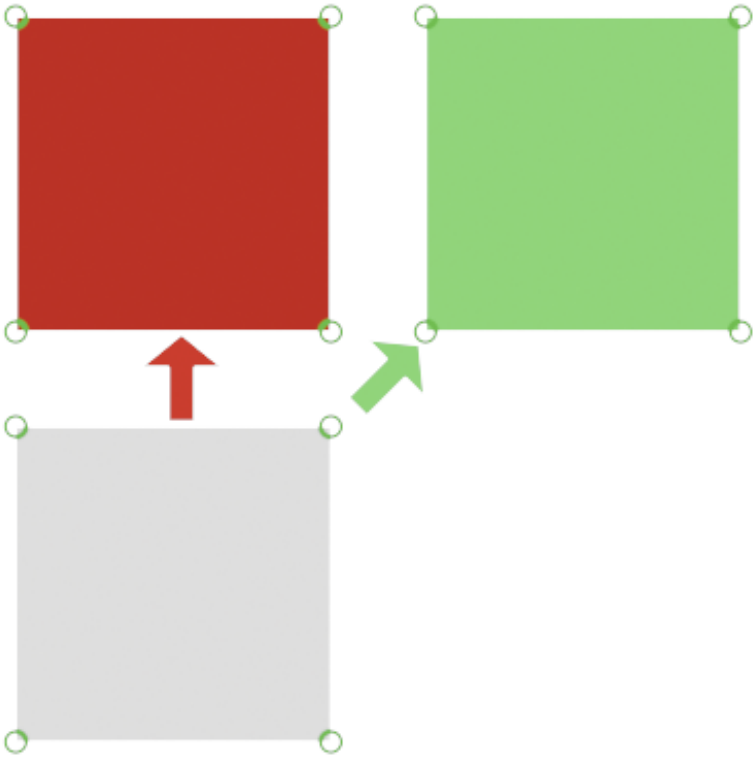


Abbildung 4.5: Anwendung Transformation Bild 1

Wird diese Transformation jedoch auf das Bild angewandt, erscheint das Quadrat an der falschen Stelle. Dargestellt in Rot. Hier kommt die Trajektorie ins Spiel. Die Trajektorie beschreibt eine fortlaufende Bahn eines Punktes im Bild, wodurch der Bewegungsverlauf des Punktes nachvollzogen werden kann. Somit nehmen wir die Momentane Transformation und addieren diese mit der letzten Trajektorie. Im Beispiel (Abbildung 4.4) angewendet Resultiert das zu der grünen Transformation und somit zum richtigen Ergebnis.

4.7 Bewegungsfluss zu Transformation

Bisher wurde der Bewegungsfluss von jedem Merkmal oder sogar Pixel berechnet. Aber bei einer Bildstabilisation ist es nicht möglich, das Bild mit verschiedenen Vektoren zu verschieben. Deswegen muss ein Weg benutzt werden, der alle Transformationen zusammennimmt und eine Trajektorie zurückgibt.

4.7.1 Numpy Mean

Die simpelste Methode dafür ist es einfach den Durchschnitt von allen Transformationen zu nehmen. Problem dabei ist, dass dadurch sehr viele Ausreisser dazu genommen werden, weil jedes Merkmal die gleiche Gewichtung hat. Ausserdem fehlt die Rotationsinformation und das Bild kann somit nur verschoben und nicht rotiert werden.

Abbildung 4.6: Beispiel Rasteransicht

Numpy Mean kann verbessert werden, indem das Bild zuerst in ein Raster aufgeteilt wird, siehe Abbildung 4.6, und dann am Ende wieder zusammengenommen wird. Dadurch vergrössert sich die Gewichtung zu grösseren Objekten, was in den häufigsten Fällen dem Hintergrund entspricht. Der Vorteil von Numpy ist die Geschwindigkeit und Effizienz.

4.7.2 Estimate Affine

Die estimateAffinePartial2D[28] Funktion nimmt zwei Listen von Punkten und berechnet die Optimale Transformation. Dies wird ermöglicht durch einen robusten auf der Grundlage von RANSAC[6] aufgebauten Algorithmus. Danach wird mit dem Levenberg-Marquardt[14] Algorithmus die Fehlerrate noch weiter vermindert. EstimateAffinePartial2D filtert Ausreisser aus und gibt zusätzlich zur Verschiebung noch ein Rotationsgrad zurück.

4.8 Glättung

Um die Stabilisation zu verbessern, muss eine Glättung auf die Trajektorie angewandt werden, was die orange Linie darstellt in Abbildung 4.7. Dies ermöglicht es der Transformation geschmeidigere Bewegungen zu machen, statt ein Standbild zu erhalten. Dies ist vor allem nötig bei Videos, bei welchen die Kamera sich mit dem Geschehen mitbewegt. Es gibt auch die Möglichkeit die Glättung zu ignorieren und direkt die Trajektorie anzuwenden. Da das Projekt mit Stream Daten arbeitet, wird ein Algorithmus gebraucht, welcher fortlaufend eine Glättung ausführen kann.

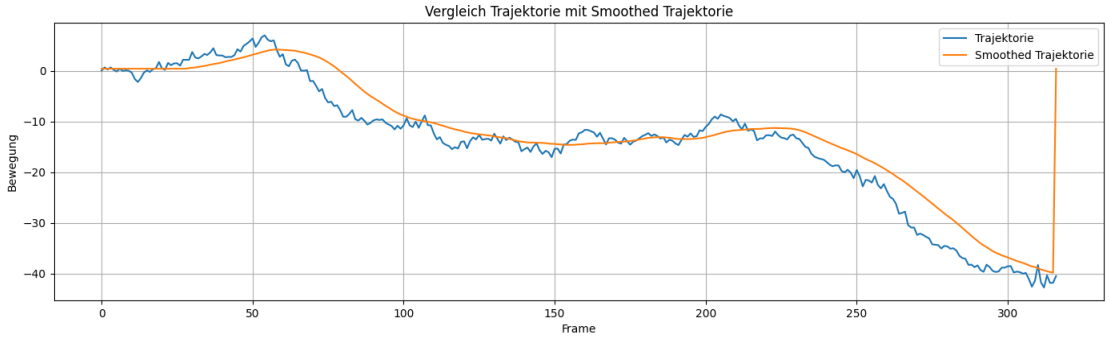


Abbildung 4.7: Geglättete Trajektorie Beispiel

4.8.1 Smoothing Window

Um eine erfolgreiche Glättung zu erreichen, wird ein Puffer benötigt, welcher zuerst die Bilder sammelt, bevor diese geglättet werden können. Grund dafür ist, dass eine Glättung, welches nur Daten aus der Vergangenheit beinhaltet nicht gut auf abrupte Bewegungen in der Zukunft reagieren kann und somit die Glättung hinter der eigentlichen Bewegung hinterherhängt. Aus diesem Grund gibt es das Smoothing Window, welches eine vordefinierte Länge besitzt und die Glättung selbst dann um dessen Länge zurückhängt.

4.8.2 Gleitender Mittelwert

Mit einem Gleitenden Mittelwert[9] kann mit nur einer kleinen Menge an Bilder ein Mittelwert berechnet werden. Dafür wird ein vordefinierte Fenstergrösse genommen, die Daten aufsummiert und danach der Durchschnitt genommen. Dies ermöglicht eine schnelle und effiziente Berechnung des Mittelwertes.
Es gibt verschiedene Variationen davon.

Expanding

Die Expanding Methodik [20] berücksichtigt die ganze Vergangenheit. Für ein Video, bei welcher die entfernte Vergangenheit kein Einfluss auf die Gegenwart hat, ist diese Methode nicht gut geeignet.

Exponential

Bei der Exponentiellen Methodik [19] wird noch eine Gewichtung mitgegeben, mit welcher gezeigt wird wieviel entferntere Bilder vom Aktuellen Bild gewichtet werden sollen.

4.8.3 Konvolution

Ein anderer Weg ist es eine Konvolution[22] zu benutzen. Hierbei werden zwei Zeitstrahlen aufeinandergelegt und deren Mittelwert zurückgegeben. Dies ermöglicht eine Zeitnahe Berechnung einer Glättung. Konvolution kann mit einer Box erfolgen, welches die Glättung gleichmässig verteilt. Es kann auch ein Gaussian Filter[7] angewandt werden, welcher die Distanz gewichtet und somit entferntere Bilder weniger Einfluss gibt.

4.9 Affine Abbildung

Mit einer Affinen Abbildung [3] wird die Trajektorie auf das Bild angewandt. Dabei handelt es sich um eine Transformationsmatrix mit zwei verschiedenen Formen. Die erste Form is für den Fall, dass es keinen Rotationsinformationen für die Transformation beinhaltet:

$$\begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \end{bmatrix}$$

Wobei x und y die Trajektorie der jeweiligen Koordinaten sind.

Die zweite Form berücksichtigt auch die Rotation, was ermöglicht, dass das stabilisierte Bild auch gedreht werden kann um allfällige Kamerarotationen auszugleichen:

$$\begin{bmatrix} cos(a) & -sin(a) & x \\ sin(a) & cos(a) & y \end{bmatrix}$$

Wobei a der Winkel und x und y die Trajektorie der jeweiligen Koordinaten sind.

4.10 3D

3D Algorithmen können das Stabilisieren enorm verbessern. Die meisten dieser Algorithmen benutzen leider DeepLearning welche nicht Echtzeitfähig sind.
Ein manueller Versuch wurde gestartet, jedoch sehr schnell gestoppt, da ohne die intrinsischen Kameraparameter keine effiziente und zuverlässige 3D-Transformation möglich ist.

Kapitel 5

Projekt

5.1 Prototyp

Der Prototyp wurde bereits sehr früh umgesetzt um die Frage, welches Streaming Protokoll verwendet werden sollte, zu beantworten. Mit dem SRT-Protokoll hatten wir bereits früh Erfolg.

5.2 Audio

Der Fokus des Projektes ist es, ein guten Stabilisationsalgorithmus zu finden. Deswegen wurde die Audio-Spur nach dem Demultiplexen nicht mehr berücksichtigt. Diese kann jedoch hinzugefügt werden, indem die Audiospur beim Multiplexen dem Stream wieder hinzugefügt wird. Dafür muss zusätzlich der Zeitstempel des Audio-Streams mit dem Video-Stream synchronisiert werden und den Queues mitgegeben werden.

5.3 Python Implementation

Das Lokal laufende Tool wurde komplett neu und möglichst modular aufgebaut. Zuerst wurden die einzelnen Schritte voneinander getrennt. Somit trennten wir das Analysieren und das Stabilisieren. Darauf folgte die Trennung vom Output damit dieser je nach Bedarf ausgewechselt werden kann. In unserem Testvideo passierte es, dass die Stabilisation des Bildes eine Person als Fokus nahm. Falls diese Person dann an den Rand des Bildes kommt, stabilisierte das Tool diese zurück in die Mitte. Was dazu führte, dass etwa die hälfte des Bildes ein schwarzer Rand war. Mit einer neuen Funktionalität, welche die maximale Stabilisation und somit wie weit das Bild sich verschieben darf, konnte die Stabilisation in einem zufriedenstellenden Rahmen gehalten werden. Danach fügten wir ein Zoom ein, welcher den schwarzen Rand wegschneidet.

(a) Kein Zuschneiden und Kein Limit

(b) Kein Zuschneiden aber mit Limit

(c) Limitiert und zugeschnitten

(d) Original

Abbildung 5.1: Vergleich Limitierung und Zuschneidung

Das Resultat in der Abbildung 5.1 war um einiges besser, aber fühlte sich immer noch schlechter an, als das ursprüngliche Video. Deswegen wurde ein Mechanismus eingebaut, welcher das Bild jeden Frame um einen Pixel in die Mitte zurückbewegt. Dadurch bleibt das Bild nicht an einer Wand kleben und stoppt das Stabilisieren. Im späteren Verlauf des Projektes stellte sich jedoch heraus, dass diese Funktionalität nicht geeignet war. Sie verhinderte eine geschmeidige Bewegung und verursachte mehr Ruckler als zuvor und wurde deswegen deaktiviert.

5.3.1 Raster

Als nächsten Schritt fügten wir den raster-basierte Analyseablauf hinzu. Dieser sollte verursachen, dass nicht nur die grössten Features genommen und übergewichtet werden sollten, sondern ein Durchschnitt von allen im Raster gefundenen. Nach dem Aufteilen in ein Raster wird eine einzelne Kachel genommen und überprüft ob diese überhaupt relevant ist, indem der Kontrast der Kachel angeschaut wird. Nur die Kacheln, welche durch den Filter kommen, werden dann analysiert und genau gleich wie die bisher existierende Analyse benutzt.

5.3.2 Estimate Affine 2D Algorithmus

Bisher war das Analysemodul aufgeteilt in Raster und Vollbild. Jedoch um spezifischere Algorithmen testen zu können, benötigte es benötigte es einen neuen Layer. Deswegen fügten wir den Analysemodulen noch ein Algorithmus Modul hinzu, welches ermöglicht die Algorithmen vereinfacht auszutauschen. Dies verhinderte das Duplizieren von Code, so dass wir nicht immer eine Version für Raster und eine Version für Vollbild benötigten.
Danach bauten wir den Estimate Affine 2D Algorithmus ein. Der Algorithmus fügt die Möglichkeit hinzu, die Rotation des Bildes zu erhalten. Somit würde das Bild nicht nur verschoben, sondern auch falls nötig rotiert werden. Die Stabilisation musste dafür leicht angepasst werden um eine Rotation zu erlauben und ausserdem das zuschneiden des Bildes musste auch erweitert werden. Das Resultat mit der Affinen Transformation war leider viel verwackelter als mit der direkten Methode. Deswegen entschieden wir uns vorerst mit der Numpy Mean Methode weiterzumachen.

5.3.3 Performance Checker (PC)

Mit einem laufenden Programm war es nun Zeit die Laufzeit zu überprüfen. Für das wurde die Performance Checker Klasse entworfen, welche an alle Objekte mitgegeben wird. Diese ist sehr simpel und enthält einfach einen Dictionary. PC enthält zwei Hauptfunktionen. Zum einen der Start einer Zeitmessung und dann das Ende der Messung. Dort berechnet PC die Dauer des Auftrags. Falls es wiederholende Schritte gibt, zum Beispiel das Analysieren jedes Frames, dann werden die Zeiten angesammelt und zum Schluss der Durchschnitt berechnet. Am Ende der Stabilisierung gibt PC dann alle Zeiten auf der Konsole zurück. Durch diese Implementierung war es nun sehr einfach die Algorithmen miteinander zu vergleichen und zu überprüfen, wo genau es zum Stocken kommt.

5.3.4 Gleitender Mittelwert mit verschiebbaren Mittelpunkt

Der Puffer für die Stabilisation wird als Smoothing Window mitgegeben und geht nur in die Zukunft. Deswegen entschieden wir uns auch die Vergangenheit anzuschauen und änderten den Stabilisationsalgorithmus, damit das Programm beidseitig die Glättung anwenden kann. Jedoch stellten wir schnell fest, dass die Ergebnisse um einiges schlechter waren und beim Nachforschen fiel uns auf, dass die komplette Vergangenheit bereits beim vorherigen Verfahren berücksichtigt wurde. Somit verwarfen wir dieses Verfahren.

5.3.5 Pandas Stabilisierer

Als nächsten Schritt fügten wir weitere Glättungsalgorithmen hinzu. Als erstes der Pandas Rolling Algorithmus, welcher genau das gleiche Ergebnis liefert, wie der Manuelle Weg, welcher mit Numpy gemacht wurde. Danach ergänzten wir noch den Pandas Expanding und Pandas Exponential Algorithmus für die Glättung, stellten jedoch keine grossen Unterschiede fest.

5.3.6 Sharpening

Wegen dem Abschneiden der Ränder und der dabei nötige Zoom wird das Ergebnis, wie in Abbildung 5.2a zusehen ist, um einiges verschmierten. Deswegen entschieden wir uns einen Schärfefilter zu verwenden, um dem Bild die Details ein wenig zurückzugeben.

(a) Keine Schärfung

(b) Starke Schärfung

(c) Mittlere Schärfung

Abbildung 5.2: Schärfungsvergleich

Beim ersten Versuch setzten wir die Schärfefilter zu hoch ein, was zum Resultat in Abbildung 5.2b führte. Bei der Rotation der Kamera waren dadurch Linien sichtbar durch die Schärfung. Aber nach der Anpassung der Feineinstellungen, erreichten wir ein zufriedenstellendes Ergebnis wie in Abbildung 5.2c zu sehen ist.

5.3.7 Resizer

Bisher arbeiteten wir mit 720p Videos, weil das die Mindestauflösung war, welche unsere Kameras ermöglichten. Deswegen entwickelten wir ein kleines Tool, welches mit FFmpeg die Auflösung auf 480p runterschraubt, damit wir auch einen Vergleich mit der kleineren Auflösung haben beim Geschwindigkeitstest.

5.3.8 Transformation Files und Merkmalsansicht

Um die Unterschiede zwischen den einzelnen Algorithmen besser darstellen zu können, entschieden wir uns die Transformation und Trajektorie Daten des Programmes in ein Text File zu speichern. Dies ermöglicht uns bei der Auswertung diese miteinander zu Vergleichen und allfällige Fehler zu finden. Die Implementation davon verlief reibungslos.
Im gleichen Ansatz erstellten wir auch die Merkmalsansicht, welche ermöglicht, die gefundenen Merkmale des Analysers auf dem Bild zu repräsentieren.

5.3.9 Weitere Merkmalerkennungsalgorithmen

Die Erkennung der Merkmale wurde bisher von Good Features to Track übernommen und wir wollten testen, ob andere manuelle Ansätze eine Bessere Effizienz herausholen können. Deswegen testeten wir den FAST, Harris und SIFT-Algorithmus, welche jedoch nicht sehr gute Resultate lieferten und deswegen nicht in das Kernprogramm eingespeist wurden.

5.3.10 Kein Smoothing

Bisher war die Qualität des Stabilisierten Video nicht sehr zufriedenstellend. Unser verdacht lag an den Glättungsalgorithmen, da diese die geglättete Transformation zurückgaben und nicht die geglättete Trajektorie. Aber auch beim Versuch die Trajektorie zurückzugeben, war das Ergebnis schlechter als zuvor. Deswegen entschieden wir uns einen Stabilisierer hinzuzufügen, welcher die Glättung komplett weglässt und einfach nur die Trajektorie zurückgibt.
Das Ergebnis war nicht zufriedenstellend. Aber eine Hoffnung blieb uns. Die Berechnung der Transformation war bisher vom vorherigen Bild zum nächsten. Was bedeutet, dass dadurch eine Transformationsschnur gebildet wird, welche in die Zukunft schaut und nicht zum Anfang.
Somit entwickelten wir einen Switch, welcher über ein Argument gesteuert wird, ob das Programm die Transformation vom nächsten Bild nehmen sollte oder vom vorherigen. Für die Transformation ohne Rotation bedeutete dies lediglich, dass die Negation genommen wird und um ein Bild verschoben.
Durch diese Änderung war der Kein Smoothing-Test erfolgreich und unserer Meinung nach sogar besser als die bisherigen Algorithmen. Gleichzeitig konnten wir durch diese Änderung das Problem mit dem Estimate Affine 2D Algorithmus lösen, welchem die falsche Reihenfolge der Bilder mitgegeben wurde.

5.3.11 Konvolution

Die Idee, Konvolution für das Smoothing zu benutzen, kam auf. Weswegen wir diesen Stabilisierer hinzufügten. Zur gleichen Zeit unterstützten uns unsere Betreuern mit einen Gauss Algorithmus, welcher mit Konvolution das Smoothing auch verbessern sollte. Der Vorteil des Gauss Algorithmus ist es, dass dieser kein Limit braucht. Die Implementation der Algorithmen lief reibungslos in das Programm, dank dessen Modularen Umsetzung und Steuerung über Befehlszeilenargumente.

5.3.12 Trajektorie Plotten

Gegen Ende des Projektes erstellten wir ein Tool, welches die Transformation und Trajektorie Daten einliest und Diagramme zurückgibt. Das Tool selbst ist nur ein Prototyp und dient als Hilfe für die Dokumentation. Ein Beispiel davon kann im Research, in der Abbildung 4.7 gefunden werden. Das Einlesen der Transformationsdateien war zu Beginn nicht sehr einfach, da wir eine Liste von Zahlen in ein Textfile speicherten und diese nachher mit Python wieder einlesen mussten. Im Nachhinein wäre es besser gewesen, wir hätten den Filetyp auf JSON geändert, aber bis wir zu diesem Schluss gekommen waren, hatten wir das Tool bereits fertiggestellt.

5.3.13 Docker

Die Applikation kann direkt auf ein System, welches die Anforderungen erfüllt wie in (Unterabschnitt 3.3.7) beschrieben, installiert werden. Jedoch ist es einfacher, die Applikation in einem Docker Container zu betreiben. Dies vereinfacht die Installation und ermöglicht auch gleich die Skalierung der Applikation, falls mehrere Streams gleichzeitig verarbeitet werden sollen.
Das Docker File ist nach dem Multi-Stage Build Prinzip aufgebaut, welches das finale Image so klein wie möglich hält. Es wird im ersten Schritt ein Basis-Image mit Python 3.11-slim geladen, FFmpeg mittels apt-get und alle Python Bibliotheken mit poetry installiert. Dieses Image wird als builder bezeichnet und wird für das Erstellen der Applikation verwendet. Hier sind alle Abhängigkeiten für den Build mitinstalliert, welche jedoch für die Ausführung der Applikation nicht mehr benötigt werden.
Das zweite Image ist das runtime Image, welches nur die Applikation und runtime relevanten Bibliotheken und Abhängigkeiten enthält. Dieses ist ebenfalls auf Python 3.11-slim aufgebaut und die FFmpeg Bibliothek wird mit apt-get installiert. Die Applikation, sowie die benötigten Abhängigkeiten, werden jedoch vom Builder-Image kopiert. Dadurch wird das Finale Image ca. 30% kleiner, als wenn das Builder-Image direkt als Runtime-Image verwendet wird.
Die Applikation im Docker Container wird mit den gleichen Argumenten, wie direkt auf dem lokalen System, gestartet. Dies erlaubt es, verschiedene Konfigurationen zu verwenden und verschiedene Streams gleichzeitigt mittels gestarteten Containers zu verarbeiten, da die jeweilige Stream URL dem Container mitgegeben wird. Somit kann

auch die Skalierung der Applikation einfach umgesetzt werden, indem mehrere Container gestartet werden.

5.4 C++

Ein Prototyp in C++ wurde umgesetzt, um Vergleiche im Bezug der Performance mit der Python Version zu machen. Ein entscheidender Grund für die Auswahl von C++ war, dass das SRT-Protokoll ebenfalls in C++ implementiert wurde. Somit können wir potenzielle Engpässe feststellen, die Python eventuell haben könnte. Python zählt hinsichtlich Ausführungsgeschwindigkeit und Speicherverbrauch nicht zu den effizientesten Programmiersprachen. Dies wird unter anderem durch die Vergleichstests auf der Webseite "The Computer Language Benchmarks Game" veranschaulicht.

5.4.1 Umgebung und Abhängigkeiten

Für die Entwicklung des C++ Prototypen wurde der g++ Compiler mit der Version 11.4.0 unter Ubuntu 22.04 verwendet. Die für die Stabilisierung notwendige Bibliothek opencv4 muss in der Entwicklungsumgebung installiert sein. OpenCV[23] muss mit der Version 4.5.4 oder höher installiert sein. Cmake muss die Version 3.10 oder höher aufweisen für dieses Projekt. Die FFmpeg Bibliothek wird nicht benötigt, da beim C++ Prototypen der SRT-Stream direkt konsumiert wird über die SRT-Bibliothek.
Die Installation der Abhängigkeiten wird mit folgenden Befehl gemacht:

sudo apt install libopencv-dev

5.4.2 Prototypen

Es wurden 2 Prototypen in C++ gemacht, um einen Eindruck zu erhalten, wie die Verarbeitunsgeschwindigkeit gegenüber Python ist.

1. Prototyp Der erste Prototyp liest eine Video Datei aus, stabilisiert dieses Video und speichert die verarbeite Version. Somit können wir in Python und C++ einen direkten Geschwindigkeitsvergleich machen mit den gleichen Video-Dateien.

2. Prototyp Dank der nativen Einbindung der SRT-Bibliothek in C++ kann der zweite Prototyp den Stream direkt verarbeiten, was theoretisch eine optimierte Performance ermöglicht. Durch die direkte Einbindung des SRT-Streams kann ein zusätzliche FFmpeg subprocess vermieden werden.
Diese Implementierung diente dem Vergleich der Latenz des Streams gegenüber der Python-Version.

5.5 AWS Services

5.5.1 AWS CLI

Die AWS CLI ist ein Befehlszeilen-Tool, welches es ermöglicht, AWS-Dienste über die Befehlszeile zu verwalten. Diese wird unter anderem auch dazu benötigt, sich bei der AWS Elastic Container Registry (ECR) anzumelden, um Docker-Images zu pushen und zu pullen.
Hier wird die Konfiguration der AWS CLI mittels SSO (Single Sign-On) beschrieben, um sich bei AWS zu authentifizieren und die benötigten Berechtigungen zu erhalten.

```
aws configure sso --profile ost-ba
SSO session name (Recommended): ost-ba
SSO start URL [None]: https://d-99676e93b2.awsapps.com/start
SSO region [None]: eu-central-1
SSO registration scopes [sso:account:access]: sso:account:access
```

Mit dem erfolgreichen Login kann nun die AWS CLI mit dem Profil “ost-ba“ verwendet werden, um auf die AWS-Dienste zuzugreifen.

5.5.2 AWS Elastic Container Registry (ECR)

Die AWS Elastic Container Registry (ECR) ist ein verwalteter Docker-Container-Registry-Dienst, der es ermöglicht, Docker-Images zu speichern, zu verwalten und bereitzustellen.

```
aws ecr get-login-password --region eu-central-1 --profile ost-ba |
    docker login --username AWS --password-stdin 788180922866.dkr.ecr.
    eu-central-1.amazonaws.com
```

Unsere Application wurde als Container Image in der AWS ECR registriert, um sie später im AWS ECS (Elastic Container Service) zu verwenden.

5.5.3 AWS EC2

Die AWS EC2 (Elastic Compute Cloud) ist ein Web-Service, der es ermöglicht, virtuelle Server in der Cloud zu starten und zu verwalten.
Dieser wird im Projekt verwendet, um direkt auf der EC2-Instanz unseren Code auszuführen und zu testen.

Die Instanz kann über das Web-Interface oder über die AWS CLI mit dem folgenden Befehl gestartet werden. Dabei wird ein Amazon Machine Image (AMI) verwendet, welches Ubuntu 24.04 LTS enthält.

```
aws ec2 run-instances --image-id ami-{id} --count 1 --instance-type
    t2.micro --key-name ost-ba-key --security-group-ids sg-0
    a1234567890abcdef --subnet-id subnet-0a1234567890abcdef --profile
    ost-ba
```

Das Profil "ost-ba" wurde zuvor mit der AWS CLI konfiguriert, um sich bei AWS zu authentifizieren.
Benötigte Pakete, welche auf der EC2 Instanz installiert werden müssen, um das Projekt auszuführen:

- Python 3.11 or higher
- pip3 (Python Package Installer)
- poetry (für die Paketverwaltung)
- FFmpeg (für die Videoverarbeitung)

Um die Pakete zu installieren, können die folgenden Befehle verwendet werden. Dazu muss zuerst die EC2-Instanz per SSH verbunden werden und der Source Code entweder via SCP auf die Instanz kopiert oder der Code aus dem Git-Repository geklont werden. Für das Klonen eines Git-Repositorys werden jedoch Zugangsdaten (Credentials) benötigt auf der AWS-Instanz.

```
# SCP Befehl zum Kopieren des Codes auf die EC2 Instanz
scp -i /path/to/your/key.pem -r /path/to/your/code ubuntu@your-ec2-
    instance-ip:/home/ubuntu/
# SSH Verbindung zur EC2 Instanz herstellen
ssh -i /path/to/your/key.pem ubuntu@your-ec2-instance-ip
# Auf der EC2 Instanz die Pakete installieren und das Projekt
    einrichten
sudo apt install python3 python3-pip pipx ffmpeg
pipx install poetry
# erstelle ein Verzeichnis fuer das Projekt
mkdir ~/cvs
cd ~/cvs
tar -xzf cvs-python.tar.gz
# Konfiguration von Poetry, um virtuelle Umgebungen im
    Projektverzeichnis zu erstellen
poetry config virtualenvs.in-project true
# poetry config virtualenvs.create false
poetry install
```

5.5.4 AWS ECS

Leider konnten wir die Applikation nicht auf AWS ECS (Elastic Container Service) deployen, da wir nicht die nötigen Berechtigungen hatten, um die Container zu starten. Auch wurde die Zeit für die Problemlösung zu knapp und wir mussten uns daher auf die EC2-Instanz beschränken.

5.6 AWS IVS Probleme

Wir hatten zu Beginn das Problem, dass wir nicht die richtigen Keyframes (siehe Abbildung 5.3) für den Stream gesetzt haben, wodurch der Stream abgehackt und nicht flüssig lief.

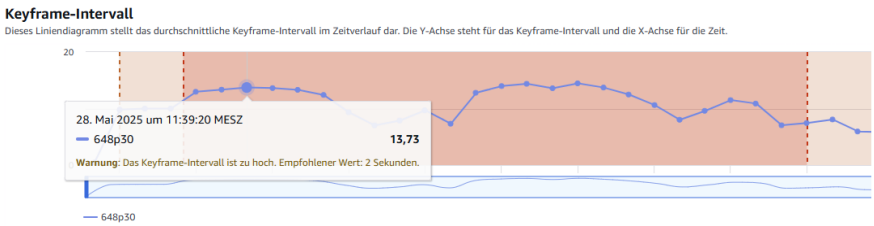


Abbildung 5.3: AWS IVS Keyframe Problem

Auch wurde die Auflösung des Streams nach dem Glätten nicht korrekt hochskaliert, was zu dem speziellen Format von 648p30 führte. Dies war jedoch nicht das Hauptproblem, da die Keyframes nicht korrekt gesetzt waren, sondern nur ein implementierungstechnisches Problem.

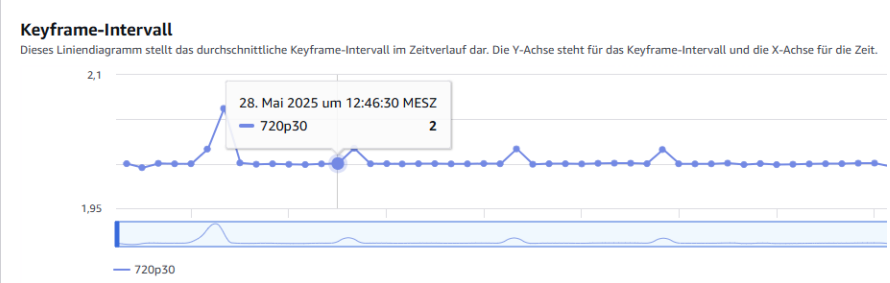


Abbildung 5.4: AWS IVS Keyframe Problem gelöst

Durch das setzen der Parameter `-g` (Keyframe-Intervall) und `keyint_min` (minimales Intervall) auf jeweils den Wert 60 beim FFmpeg-Encoding, konnte das Problem erfolgreich behoben werden. Siehe Abbildung 5.4 für das Ergebnis. Auch wurde die Auflösung des Streams auf 720p hochskaliert.

5.7 Aufnahme Testvideo

Damit die verschiedenen Algorithmen und deren Qualität verglichen werden können, mussten Videos für verschiedene Szenarien aufgenommen werden. Damit kann auch die Effektivität der einzelnen Algorithmen für das jeweilige Szenario optimiert, verglichen und bewertet werden. Zusätzlich zu den Szenarien wurde auch ein reales Video auf einem Pumptrack aufgenommen. Damit die Qualität der Stabilisierung beurteilt

werden kann, sind zusätzlich zwei GoPro (GoPro 7 und GoPro 9) Aufnahmen gleichzeitig gemacht worden.

5.7.1 Szenarien

Die Szenarien wurden mit einer Handykamera aufgenommen mit einer Auflösung von 720p zu 30 FPS. Es sind insgesamt 6 Szenarien aufgenommen worden.

- **Szenario 1 (Forest):** Statische Kamera, welche auf ein Waldstück fokussiert ist. Ausser den Windbewegungen der Bäume passiert nichts.
- **Szenario 2 (Ski):** Bewegende Kamera, welche auf eine Person fokussiert ist, während dem Skifahren.
- **Szenario 3 (Slowing Train):** Statische Kamera, welche auf einen Zug fokussiert ist. Der Zug ist weit weg, ein kleines Objekt und fährt durch das Bild.
- **Szenario 4 (Train Far):** Statische Kamera, welche auf einen Zug fokussiert ist. Der Zug ist nah, ein grosses Objekt und fährt durch das Bild.
- **Szenario 5 (Selfie Bycicle):** Kamera am Bike montiert, welche die Person selbst auf dem Bike filmt. Die Person fährt mit dem Bike über einen Kiesweg.
- **Szenario 6 (Pumptrack):** Kamera am Bike montiert, welche nach vorne über einen Pumptrack filmt.

5.7.2 Pumptrack (Szenario 6)

Das reale Video wurde mit einem Handy (Oneplus 6), einer GoPro 7 und einer GoPro 9 aufgenommen. Die Handykamera hat eine Auflösung von 720p zu 30 FPS, die GoPro 7 hat eine Auflösung von 720p zu 60 FPS und die GoPro 9 eine Auflösung von 1080p zu 60 FPS.
Dabei wurde für die GoPro 7 und 9 die jeweils beste Stabilisierung aktiviert, welche die GoPro bietet. Die Handykamera lief ohne Stabilisierung, was für die Tests der Algorithmen verwendet wird.
Alle drei Kameras wurden am Mountain-Bike montiert, wie in Abbildung 5.5 zu sehen ist. Somit konnten die Aufnahmen unter den gleichen Bedingungen gemacht werden, was für den Vergleich der Qualität der Stabilisierung wichtig ist.

Abbildung 5.5: Reales Video auf einem Pumptrack bike Setup

Kapitel 6

Evaluation

6.1 Vergleich C++ und Python

Das Projekt wurde in Python geschrieben, da es eine einfache und schnelle Entwicklung ermöglicht. Um einen Eindruck zu bekommen, wie viel schneller eine C++ Implementierung wäre und ob es eventuell gar keine Option sein kann, eine Python Implementierung zu benutzen, wurde ein Test durchgeführt.
Dazu wurde ein Testvideo genommen, welches eine Person beim Skifahren zeigt und durch beide Prototypen geschickt wurde und am Ende gespeichert. Dies wurde jeweils 10 Mal durchgeführt, um die Geschwindigkeit zu messen. Dies wurde mit dem Tool Hyperfine[21] Version 1.19.0 durchgeführt, was ein Benchmarking Tool für die Linux Kommandozeile ist. Das tool führt den mitgegebenen Befehl 10-mal aus und gibt die durchschnittliche Zeit, die Varianz sowie Min und Max zurück.

```
→ demo git:(stream-out) hyperfine './vs-stream-mp4.o ../ski1.mp4'
Benchmark 1: ./vs-stream-mp4.o ../ski1.mp4
  Time (mean ± σ):     19.040 s ±  0.344 s    [User: 36.453 s, System: 16.257 s]
  Range (min … max):   18.179 s … 19.353 s    10 runs
```

Abbildung 6.1: Hyperfine Benchmark C++ Ergebnisse

Die Ergebnisse des Benchmarks für den C++ Prototypen sind in Abbildung 6.1 dargestellt.

```
(env) → src git:(main) X hyperfine "python main.py -al affine -st convolve -an full -fp custom -f ski1 -r 720 -of video"
Benchmark 1: python main.py -al affine -st convolve -an full -fp custom -f ski1 -r 720 -of video
  Time (mean ± σ):     22.504 s ±  0.636 s    [User: 42.385 s, System: 24.810 s]
  Range (min … max):   20.940 s … 23.538 s    10 runs
```

Abbildung 6.2: Hyperfine Benchmark Python Ergebnisse

Die Ergebnisse des Benchmarks für die Python Implementierung sind in Abbildung 6.2 dargestellt.
Es zeigt sich, dass die C++ Implementierung im Durchschnitt etwa 15% schneller ist als

die Python Implementierung. Dies wird mit der Formel `(Python Zeit - C++ Zeit) / Python Zeit * 100` berechnet, wobei jeweils die durchschnittlichen Zeiten verwendet werden.

$$\left(\frac{22.504s - 19.04s}{22.504s}\right) \times 100 = 15.39\%$$

Wir können dadurch annehmen, dass ein Grossteil der Python Bibliotheken im Hintergrund höchst wahrscheinlich in C++ implementiert ist. Daher ist die Geschwindigkeit von Python in diesem Fall nicht so schlecht, wie zuerst die Annahme war. Mit einer C++ Implementierung könnte die Geschwindigkeit nochmals gesteigert werden, jedoch auf Kosten der Entwicklungszeit und Flexibilität.

6.1.1 Video Auflösung

Ein Test wurde durchgeführt, um zu evaluieren, ob 720p oder 480p Videos benutzt werden sollten. Dafür wurde die Geschwindigkeit von beiden Auflösungen miteinander verglichen und auch dessen Videoqualität. 480p ist im Durchschnitt um 2.2-mal schneller als 720p, was ein enormer Geschwindigkeitsgewinn ist. Jedoch leidet die Qualität aber zu stark darunter. Gewisse Bewegungen verwirren den Bewegungsfluss Algorithmus, dass dieser den Bewegungsfluss nicht identifizieren kann und deswegen das Video weniger stabilisiert ist als das Originalvideo. Diese Ruckler stören den Zuschauer sehr, da diese sehr kurze und schnelle Bewegungen sind und im Vergleich dazu hat 720p geschmeidigere Bewegungen. Aus diesem Grund wurde trotzdem die 720p Auflösung für die weiteren Tests genommen.

6.1.2 Zoomfaktor

Das Projekt unterstützt verschiedene Zoomfaktoren, welche die Qualität der Stabilisierung verbessern können.

(a) Original

(b) 50 Pixel Zoom

(c) 100 Pixel Zoom

Abbildung 6.3: Vergleich Zoomlevels

Je grösser der Zoomfaktor ist, desto stabiler ist das schlussendliche Bild. Zumindest in der Theorie. Der Zoomfaktor ist mit der Limitierung der Trajektorie verbunden, was bedeutet, dass je grösser der Faktor ist, desto mehr kann das Bild stabilisiert werden. Jedoch zeigte sich in einem Test, dass ein grösserer Zoomfaktor die Qualität des Bildes nur beschränkt verbessert. Das Problem lag daran, dass der erhöhte Zoom die leichten Ruckler, welche nach dem Stabilisieren blieben, um einiges verstärkten.
Ausserdem war der Zoom sehr schnell bereits zu stark und zeigte nur noch einen kleinen Teil des Bildes. (siehe Abbildung 6.3)

Aus diesem Grund wurde entschieden, dass der Zoomfaktor bei 50 Pixel gehalten werden würde, was bei einem 720p Video bedeutet, dass aus den 1080 zu 720 Pixel, 50 Pixel auf jeder Seite weggeschnitten werden, was zu einem 980 zu 620 Resultat führt. Dies entspricht etwa 22%. Danach wird ein resizing ausgeführt, um wieder die gleiche Auflösung zu bekommen.

6.1.3 Smoothing Window

Das Smoothing Window ist ein essenzieller Bestandteil der Stabilisation, welches ein Gleichgewicht zwischen Latenz und Videoqualität darstellt. Je grösser das Smoothing Window desto mehr Daten können benutzt werden um die Trajektorie zu glätten und ein stabileres Video kann erzeugt werden. Die Latenz erhöht sich jedoch genauso. Ausserdem verlangsamt sich der Glättungsalgorithmus je mehr Daten dieser bekommt.

In der Praxis zeigte sich, dass ein zu hohes Smoothing Window die Qualität des Videos nicht erhöhte. Grund dafür ist, dass die Stabilisation sich nie festigen konnte, weil entweder noch die vorherige Bewegung geglättet wird oder bereits die nächste. Aus diesem Grund wurde ein allgemeines Ideal für die Testvideos gefunden, welches bei 30 Bildern lag.

6.2 Merkmalerkennungsalgorithmen

6.2.1 Geschwindigkeitsvergleich

Das Projekt beinhaltet vier Hauptalgorithmen, welche miteinander verglichen werden müssen. Dabei handelt es sich um Fast, Harris, SIFT und Good Features to Track.

Testverfahren

Im Test wurde ein Testvideo genommen, welches einen anhaltenden Zug in einem Bahnhof aufzeigt. Im Hintergrund waren vereinzelte Bäume und Häuser sichtbar.
Das Video beinhaltet 970 Frames, welche alle durch den Stabilisierer geschickt wurden. Für jeden Testfall wurde das Video fünf Mal durchgelaufen, um allfällige Stockungen zu vermindern. Schlussendlich bedeutet dies, dass der Durchschnitt das Resultat von 970*5 und somit 4850 Berechnungen ist.
Die Geschwindigkeit wurde mit dem Algorithmus selbst, aber auch mit der Lucas-Kanade Algorithmus getestet, denn ein schneller Merkmalerkennungsalgorithmus kann trotzdem um einiges langsamer sein in der Bildflussberechnung.
Die Algorithmen wurden angepasst, damit sie im Durchschnitt 100 Merkmale erkennen.

Testresultat

Im Resultat Abbildung 6.4 ist schnell erkennbar, dass Farneback und der SIFT Algorithmus um einiges zu langsam sind. Es soll angemerkt sein, dass im Durchschnitt ein Algorithmus nicht langsamer als 34 Millisekunden sein sollte, da er sonst die Geschwindigkeit von 30 Bilder pro Sekunde nicht erreichen kann.

Alle Messungen sind in	Good Features to Track		FAST		Harris		SIFT		Farneback
Milisekunden (ms)	Merkmalerkennung	Lucas-Kanade	Merkmalerkennung	Lucas-Kanade	Merkmalerkennung	Lucas-Kanade	Merkmalerkennung	Lucas-Kanade	Algorithmus
Durchlauf 1	27.457	5.363	0.697	8.834	28.589	6.885	209.947	8.554	667.086
Durchlauf 2	29.042	8.273	0.642	8.319	36.261	9.662	209.332	8.903	610.999
Durchlauf 3	28.394	6.877	0.652	8.386	28.784	6.477	205.791	8.526	603.84
Durchlauf 4	21.615	5.694	0.733	8.064	25.633	5.521	201.913	8.766	609.22
Durchlauf 5	22.388	8.799	0.92	8.595	27.079	5.214	244.251	7.691	607.289
Total	25.7792	7.0012	0.7288	8.4396	29.2692	6.7518	214.2468	8.488	619.6868
Summiertes Total	32.7804		9.1684		36.021		222.7348		

Abbildung 6.4: Geschwindigkeitsvergleich Merkmalerkennungsalgorithmen

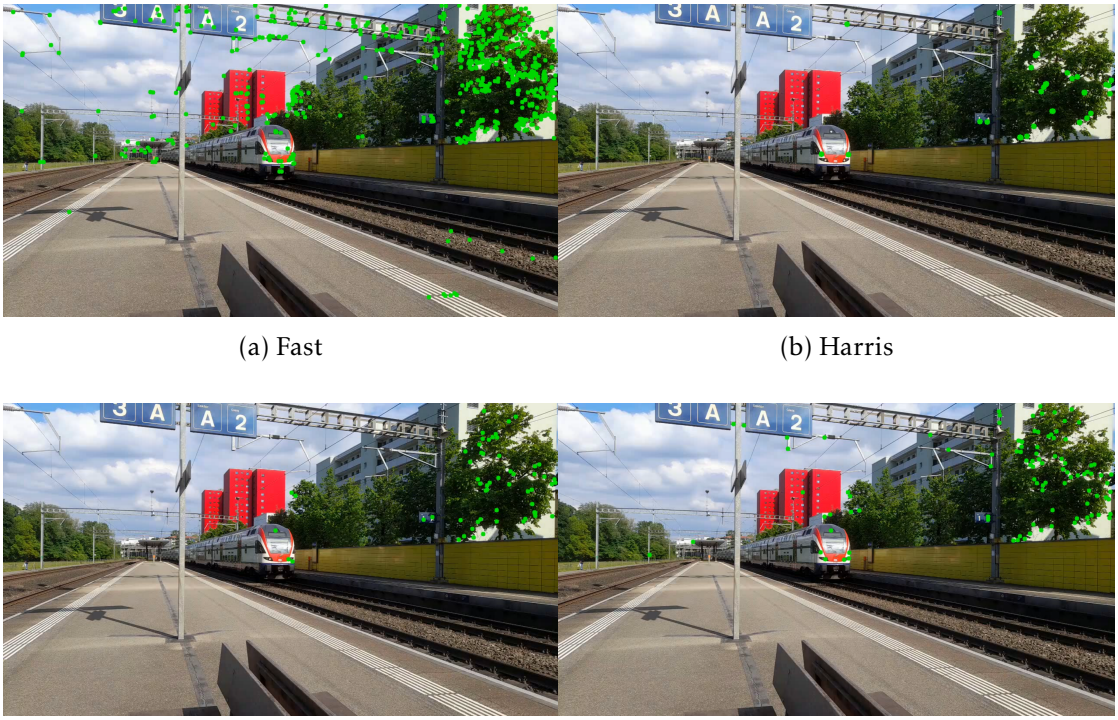
Der Farneback-Algorithmus berechnet den dichten Fluss, was bedeutet, dass er keine Merkmale nimmt, sondern alle Pixel des Bildes. Ein anderes Problem des Algorithmus besteht darin, dass er zu sehr mit dem Hintergrund auskompensieren will. Wenn das Video zum Beispiel eine Person zeigt, welche mit einem Selfie Video am Skaten ist, dann würde der Farneback-Algorithmus versuchen das Bild durchgehend zur Seite zu bewegen. Er ist um einiges besser geeignet, falls es sich um eine statische Kamera handelt.

Der SIFT-Algorithmus Over Engineered das Resultat für unseren Fall. Der Algorithmus gibt einen Rotation und Skalierungsneutrales Ergebnis zurück, welches für die Bildstabilisation nicht benötigt wird. Der Vorteil des SIFT-Algorithmus besteht in der Möglichkeit die Anzahl Merkmale auf eine fixe Zahl zu fixieren was die Berechnung des Bildflusses weniger fluktuiert.

Der Fast Algorithmus, wie der Name bereits andeutet, resultierte zum schnellsten Ergebnis. Die Effizienz des Algorithmus beherbergt aber auch Nachteile. Er kann nicht differenzieren zwischen einem qualitativ gutem Merkmal und einem Schlechteren. Dazu mehr im Merkmalvergleich.

Good Features to Track und der Harris Algorithmus erzielten ein beinahe gleiches Ergebnis. Beide Algorithmen funktionieren auf der gleichen Basis, dass sie die Ableitung von ihrem Nachbarschaft nehmen. Auch wenn der Harris im Testfall leicht unter der Gewünschten 34 Millisekunden sind, kann dieser noch mit Feinanpassungen beschleunigt werden.

6.2.2 Merkmalsvergleich



(a) Fast (b) Harris

(c) SIFT (d) Good Features to Track

Abbildung 6.5: Merkmalsvergleich

Bei allen Algorithmen in der Abbildung 6.5 ist zu sehen, dass vor allem der Baum auf der rechten Seite den Fokus erhalten hat.

Der Fast Algorithmus beherbergt alle Merkmale, welche die anderen auch besitzen, obwohl der Filter gesetzt wurde, dass dieser nur sehr wenige Merkmale nehmen sollte. Ausserdem ist auch erkennbar, dass die Punkte verglichen mit dem Good Features to Track und Harris Algorithmus leicht verschoben sind. Während beim Fast Algorithmus die Punkte genau beim Ecken der Objekte sind, sind diese bei den anderen etwas mehr auf das Objekt selbst zurückgestuft. Die zu vielen Merkmale verursacht bei der Stabilisation, dass kein Optimaler Bewegungsfluss gefunden werden kann und das Resultat dann verwackelter ist, wegen Überstabilisation, als das Original.

Der Harris-Algorithmus hat in der Implementation des Projektes das Problem, dass er die gleichen Merkmale mehrfach mit wenigen Pixel Verschiebung findet. Deswegen sieht man etwas grössere grüne Punkte in der Abbildung 6.5b als bei den anderen. Diese Verdoppelung der Merkmale vermindert die Anzahl anderer zuverlässigen Merkmale und lässt die Stabilisation einen zu grossen Bias haben.

Beim SIFT-Algorithmus sind die Punkte sehr stark auf der rechten Seite fixiert. Die Merkmale selbst sind sehr ähnlich mit dem Good Features to Track Algorithmus, aber es fehlt die Varianz in den Merkmalen. Bei der Stabilisation bedeutet dies, dass der SIFT-Algorithmus nicht sehr robust auf Änderungen reagieren kann und die Qualität deswegen unvorhersehbar ist.

Good Features to Track enthält ein gutes Gleichgewicht in der Auswahl der einzelnen Merkmale und der Anzahl. Der Algorithmus fokussierte sich nicht nur auf der rechten Seite, sondern suchte auch einige Merkmale, welche sich auf der linken Seite befinden. Für die Stabilisation ist dieser Weg ideal und das beste Endergebnis kann dadurch gefunden werden.

6.2.3 Merkmalerkennungsalgorithmen Fazit

Mit dem Vergleich der Algorithmen ist klar, dass der SIFT- und Farneback-Algorithmus wegen der Geschwindigkeit wegfallen. Auch wenn die Qualität des SIFT-Algorithmus ein gutes Ergebnis geliefert hat, ist die Geschwindigkeit ein zu grosses Problem. Es soll aber angemerkt sein, dass diese Algorithmen auch parallelisiert werden können, indem es mehrere Queues gibt, welche dann von mehreren SIFT oder Farneback Tools abgearbeitet und mit einem Zeitstempel wieder in eine weitere Queue geschickt werden. Dies würde jedoch die Rechenkosten um einiges erhöhen, da beide Algorithmen bereits eine höhere Leistung benötigen.

Der Fast-Algorithmus ist zwar der schnellste, aber mit der unzuverlässigen Merkmalen, welche gefunden werden, fällt auch dieser raus. Die Stabilisation von dem Algorithmus hat eine schlechtere Qualität als das Originalvideo. Eine Möglichkeit den Algorithmus zu verbessern, wäre es eine KI dazuzuschalten, welche die Merkmale filtert. Dies würde jedoch die Geschwindigkeit wiederum senken und wurde im Projekt nicht getestet.

Harris und Good Features to Track Algorithmus sind beide auf einer ähnlichen Qualität. Good Features to Track benutzt intern einen ähnlichen Algorithmus wie Harris. Der grosse Unterschied der beiden Methoden ist das Filtern. Good Features to Track hat bereits eingebaute Algorithmen, welche die besten Merkmale aus den Gefundenen rausliesst und nur diese zurückgibt, während Harris dieses Feature nicht besitzt. Es soll aber angemerkt sein, dass Good Features to Track von OpenCV auch mit dem Harris Algorithmus benutzt werden kann, jedoch für den Testfall nicht benutzt wurde, um die Unterschiede besser zu erkennen.

Das Fazit der Tests ist somit, dass Good Features to Track der geeignetste Algorithmus ist mit Harris als möglicher Ersatz.

6.3 Bewegungsfluss Algorithmen

Das Projekt beinhaltet zwei verschiedene Berechnungsarten für den Bewegungsfluss: Farneback und Lucas-Kanade. Farneback wurde bereits mit den Merkmalerkennungsalgorithmen getestet und als ungeeignet eingestuft, weswegen dieser wegfällt.
Dies bedeutet, dass für das Projekt nur der Lucas-Kanade in Frage kommt. Es wurde zu Beginn auch 3D Methoden begutachtet, welche jedoch bereits schnell als nicht funktionstüchtig eingestuft wurden.

6.4 Bewegungsfluss zu Transformation

Auch für die Berechnung des Bewegungsflusses zu der Transformation verfügt das Projekt nur zwei Algorithmen: Numpy mean und Estimate Affine.

6.4.1 Geschwindigkeitsvergleich

Testverfahren

Das Testverfahren wurde gleich gehalten wie für die Merkmalerkennungsalgorithmen.

Testresultat

Alle Messungen sind in Milisekunden (ms)	Numpy Mean	Estimate Affine
Durchlauf 1	0.168	0.144
Durchlauf 2	0.092	0.146
Durchlauf 3	0.108	0.159
Durchlauf 4	0.083	0.197
Durchlauf 5	0.192	0.162
Total	0.1286	0.1616

Abbildung 6.6: Geschwindigkeitsvergleich Optical Flow zu Transformation

Die Geschwindigkeit von beiden Algorithmen sind beinahe gleich wie in der Abbildung 6.6 sichtbar ist.

6.4.2 Qualitätsvergleich

Numpy Mean ist ein sehr simpler Algorithmus, welcher einfach die X und Y Transformation von den einzelnen Merkmalen nimmt und von dieser dann den Durchschnitt berechnet. Dies bedeutet jedoch, dass Ausreisser nicht ausgeschlossen werden, was zu

etwas ruckeligerem Ergebnis führt. Ausserdem beinhaltet der Algorithmus keine Informationen zur Rotation, was dem Bild eine verminderte Verbesserung der Qualität gibt.

Bei der Estimate Affine Methode werden Ausreisser ausgefiltert und zusätzlich berechnet er die Rotation der Merkmale. Dies führt zu einem verbesserten Ergebnis in der Stabilisation.

6.4.3 Bewegungsflussberechnungs Fazit

Der Numpy Mean hat den einzigen Vorteil, dass er schneller ist als der Estimate Affine Algorithmus. Was jedoch nur ein minimaler Unterschied ist im Vergleich zu den Merkmalerkennungsalgorithmen und somit nicht gross in Gewichtung fällt.
Der Vorteil von Estimate Affine ist dabei um einiges grösser. Nicht nur die Informationen der Rotation werden dabei zurückgegeben, sondern gleichzeitig auch Ausreisser gefiltert. Somit ist der Estimate Affine der geeignetere Algorithmus.

6.5 Glättung

6.5.1 Geschwindigkeitsvergleich

Testverfahren

Das Testverfahren wurde gleich gehalten wie für die Merkmalerkennungsalgorithmen.

Testresultat

Alle Messungen sind in Milisekunden (ms)	Konvolution	Gauss	Kein Smoothing	Numpy Rolling Average
Durchlauf 1	0.913	0.524	0	0.956
Durchlauf 2	0.728	0.716	0	0.75
Durchlauf 3	0.939	0.508	0	0.834
Durchlauf 4	0.735	0.541	0	0.565
Durchlauf 5	0.849	0.873	0	1.163
Total	0.8328	0.6324	0	0.8536

	Pandas Rolling Average	Pandas Expanding	Pandas Exponential
Durchlauf 1	3.024	2.866	2.709
Durchlauf 2	3.107	2.997	2.809
Durchlauf 3	3.06	2.816	2.995
Durchlauf 4	3.085	2.85	3.045
Durchlauf 5	3.402	2.844	2.912
Total	3.1356	2.8746	2.894

Abbildung 6.7: Geschwindigkeitsvergleich Glättungsalgorithmen

Das Projekt beinhaltet verschiedenste Glättungsalgorithmen, wie in Abbildung 6.7 ersichtlich ist.

Kein Smoothing gibt direkt die Trajektorie zurück und ignoriert den Glättungsprozess, weswegen er auch der schnellste Algorithmus ist. Obwohl er nichts macht, ist das Ergebnis trotzdem nicht schlecht.

Danach kommt der Gauss-Algorithmus. Der Algorithmus ist ein sehr schneller Algorithmus, welcher mit einer Gaussverteilung eine Konvolution mit den Trajektorien Daten ausführt.

Als nächstes kommt die Konvolution. Die Konvolution verhält sich sehr ähnlich wie der Gauss Algorithmus ist jedoch ein wenig langsamer. Im Gegensatz zum Gauss Algorithmus wird jedoch eine Box benutzt.

Durch den manuellen Weg mit Numpy direkt einen Rolling Average zu machen, statt diese mit Pandas Library, erzielte ein grosser Geschwindigkeitsgewinn. Das Resultat ist das gleiche wie von Pandas.

Die Pandas Algorithmen brauchen um einiges mehr an Zeit. Zuerst wurde gedacht, dass dies an der Transformation zu einem Pandas Dataframe liegen könnte, jedoch braucht das nur Durchschnittliche 0.25 Millisekunden. Es ist davon auszugehen, dass Pandas bei einem kleineren Datenset, welches wir besitzen, nicht so gut optimieren kann wie Numpy.

Fazit des Geschwindigkeitsvergleiches ist, dass die Pandas Algorithmen etwas ungeeigneter sind als die anderen, aber hinsichtlich zu der totalen Laufzeit es trotzdem keinen zu grossen Unterschied macht.

6.5.2 Transformation und Trajektorie Vergleich

Testverfahren

Um die Transformation und Trajektorie zu vergleichen, wird ein anderes Testvideo genommen als die bisherigen Tests. Dabei handelt es sich um eine Aufnahme von einem Handy, welches in einem Wald ein Standaufnahme macht. Die einzigen Bewegungen in dem Video sind die leichten Bewegungen der Blätter im Wald und die Instabilität von einer Handaufnahme.

Testresultat

In den Diagrammen wird nur die X Transformation berücksichtigt, um eine einfachere Darstellung garantieren zu können.

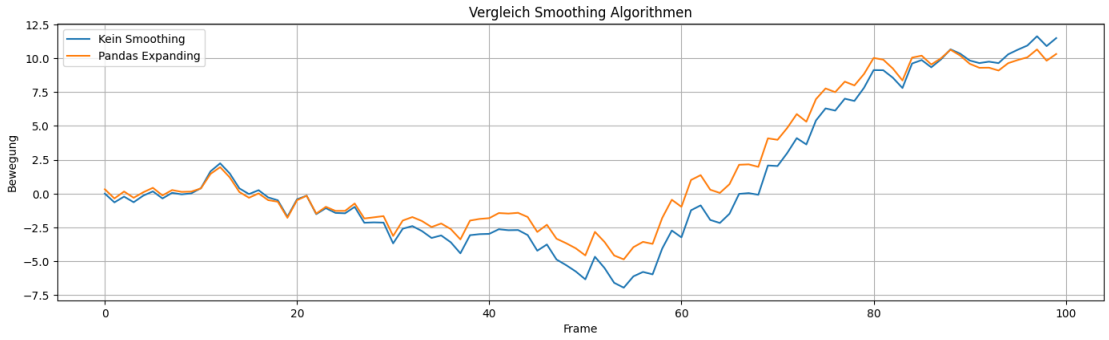


Abbildung 6.8: Transformationsvergleich *Kein Smoothing* und Pandas Expanding

Die Algorithmen in der Abbildung 6.8 für *Kein Smoothing* und Pandas Expanding tendieren weniger dazu um den 0 Punkt zu stabilisieren. Bei *Kein Smoothing* liegt es daran, dass dieser direkt die Trajektorie zurückgibt und somit bei einer Bewegung der Kamera sich vom Startpunkt wegbewegt.
Pandas Expanding versucht ein wenig zurück zum 0 Punkt zu gehen, jedoch um einiges zu wenig.

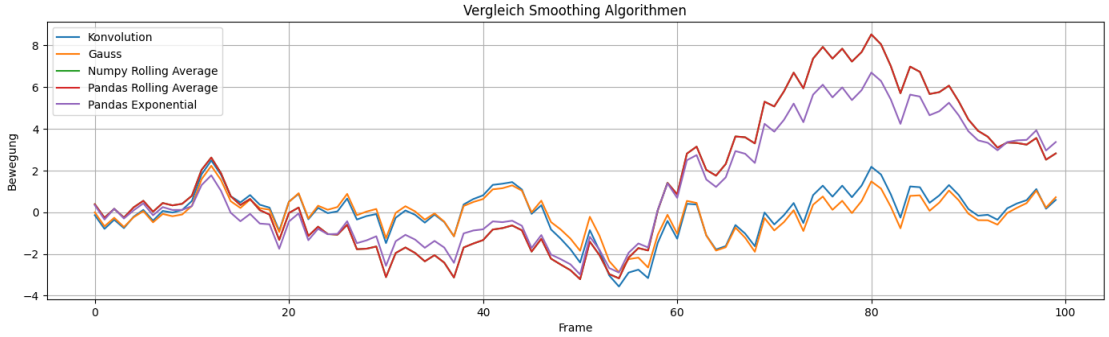


Abbildung 6.9: Transformationsvergleich restliche Algorithmen

Pandas Exponential und der Gauss Glättungsalgorithmus sind sehr ähnlich (siehe Abbildung 6.9) und tendieren mehr um den 0 Punkt zu bleiben.
Pandas Rolling Average und Numpy Rolling Average sind genau gleich, was den Erwartungen entspricht.

Konvolution geht mehr Richtung dem 0 Punkt im Gegensatz zu den Rolling Averages aber nicht so stark wie Exponential.

Im Grunde kann gesagt werden, dass die einzelnen Bewegungen von einem Bild zum nächsten bei allen sehr ähnlich sind und der grösste Unterschied liegt darin, dass gewisse Algorithmen mehr oder weniger zum 0 Punkt gehen.

6.5.3 Benutzertests

Es ist sehr schwierig die Qualität der verschiedenen Glättungsalgorithmen mit den Diagrammen zu vergleichen. Deswegen wurde ein Benutzertest gemacht, welcher fünf Personen befragte, welcher Algorithmus am besten aussieht. Die Pandas Algorithmen wurden für den Test weggelassen, um die Anzahl der Tests zu vermindern. Grundlage für die Auswahl wurde die Geschwindigkeit genommen, welche die Pandas Algorithmen um einiges zurückhängen im Vergleich zu den anderen.

Testverfahren

Für den Test wurden den fünf Testpersonen jeweils ein Zip geschickt. In dem Zip hatte es ein Excel File, bei welchem eine vordefinierte Struktur zur Verfügung stand um das Testen der Testpersonen zu vereinfachen und das Bewerten der Resultate zu beschleunigen. Die Tests sind in 5 Glättungsalgorithmen aufgeteilt:

- *FFmpeg*
- *Konvolution*
- *Gauss*
- *Kein Smoothing*
- *Rolling Average*

Der FFmpeg wurde dazu genommen um einen Vergleich zu haben mit einem externen nicht von uns entwickeltem Stabilisationsprozess. Er kann nur ganze Videos verarbeiten, weswegen er nicht für einen Stream geeignet ist.

Jeder Algorithmus hat die gleichen 6 Videoszenarien, welche bereits im Unterabschnitt 5.7.1 beschrieben sind.

Die Beurteilung erfolgt in einem Punktesystem, bei welchem 1 die schlechteste und 10 die beste Stabilisation repräsentiert. Ausserdem wurden noch die originalen und somit nichtstabilisierten Videos mitgeschickt, welche als Vergleich dienen und bei der Bewertung 5 Punkte erreichen würde, damit eine Verbesserung oder Verschlechterung der Videoqualität gezeigt werden kann.

Testresultat

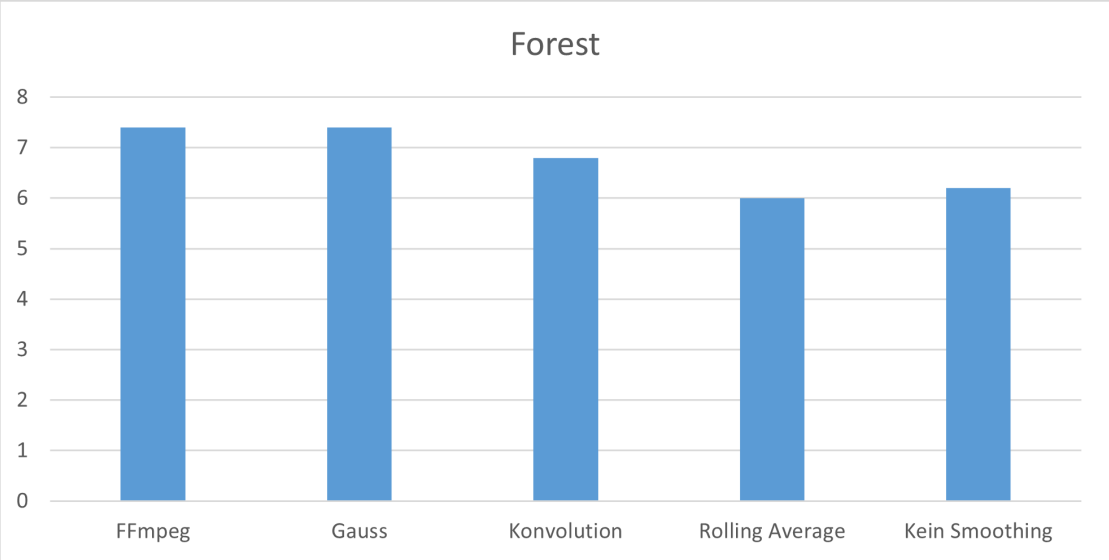


Abbildung 6.10: Benutzertest Forest

FFmpeg, Gauss und Konvolution haben eine beinahe gleiche Bewertung bekommen, wie aus der Abbildung 6.10 zu lesen ist. FFmpeg wurde beim Waldvideo am wenigsten zugeschnitten, hat aber trotzdem sehr gut stabilisiert.

Beim Rolling Average und *Kein Smoothing* kommt es öfters vor, dass, durch das Schärfen des Bildes, Wellen im Bild sichtbar sind. Dies hat den Ursprung, dass grössere Bewegungen durch diese Algorithmen verursacht werden.

Im Allgemeinen kann man sagen, dass alle Algorithmen ein besseres Ergebnis, als das Original brachten und das schärfen des Bildes das grössere Problem war.

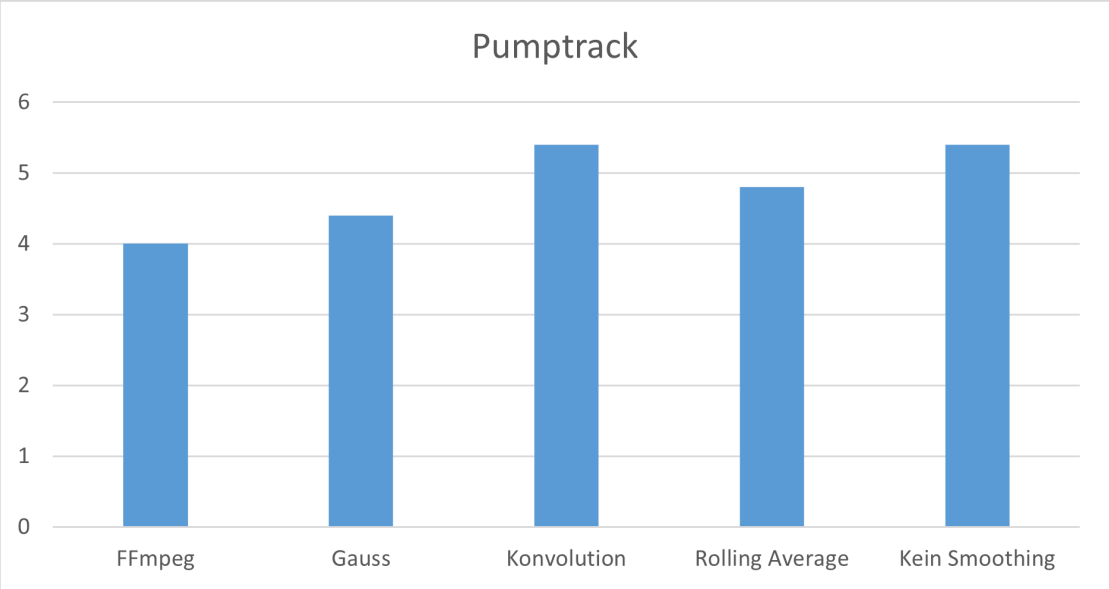


Abbildung 6.11: Benutzertest Pumptrack

FFmpeg, Gauss und Rolling Average verschlechtern die Videoqualität, siehe Abbildung 6.11, im Vergleich zum Original. Die vielen Bewegungen und abrupten Schwenkungen der Kamera macht das Stabilisieren sehr schwierig.

Auch die Konvolution und *Kein Smoothing* geben nur ein leicht besseres Ergebnis. Hier Punkten beide durch ihre etwas weniger starke Stabilisierung.

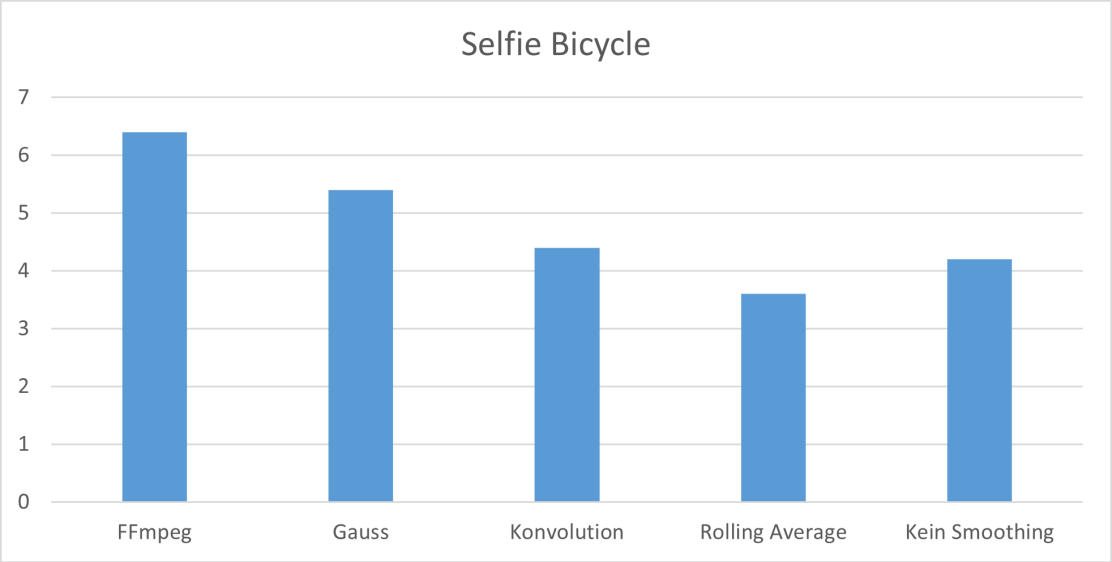


Abbildung 6.12: Benutzertest Selfie Bicycle

FFmpeg, siehe Abbildung 6.12, ist hier klar der Beste. Er ruckelt am wenigsten verglichen mit den anderen Stabilisationen, jedoch verursacht er starke Verzerrungen.

Gauss ist der einzige andere Algorithmus, welcher noch ein besseres Ergebnis als das Original lieferte.

Die Ruckler, welche vom Fahrrad auf dem Kies verursacht werden, ist für die meisten Algorithmen eine zu hohe Herausforderung. Dazu kommt noch der bewegende Hintergrund, welcher die Algorithmen zusätzlich verwirrt, da er mehr als die Hälfte des Bildes in Anspruch nimmt.

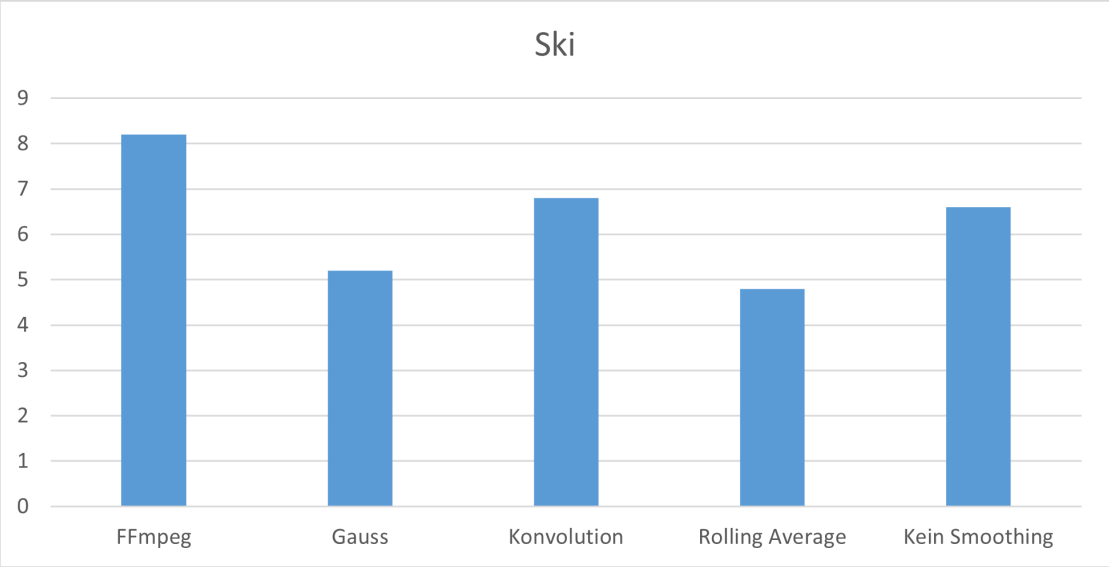


Abbildung 6.13: Benutzertest Ski

FFmpeg in der Abbildung 6.13 konnte die periodischen Schwingbewegungen am besten einlesen, was keine Überraschung ist, da er das ganze Video von Beginn an Analysieren kann.

Die Konvolution und der *Kein Smoothing* Weg zeigten eine bemerkbare Verbesserung, trotz der Limitierung der Trajektorie, welche bei dem Video schnell erreicht wurden.

Der Gauss und Rolling Average Algorithmus hingegen sind qualitativ auf dem gleichen Level wie das Original. Es soll aber angemerkt sein, dass das Original bereits eine sehr geschmeidige Bewegung beinhaltet, welches dem tiefen Kontrast des Videos und die Bewegungen des Rutschens der Skis zu verdanken ist.

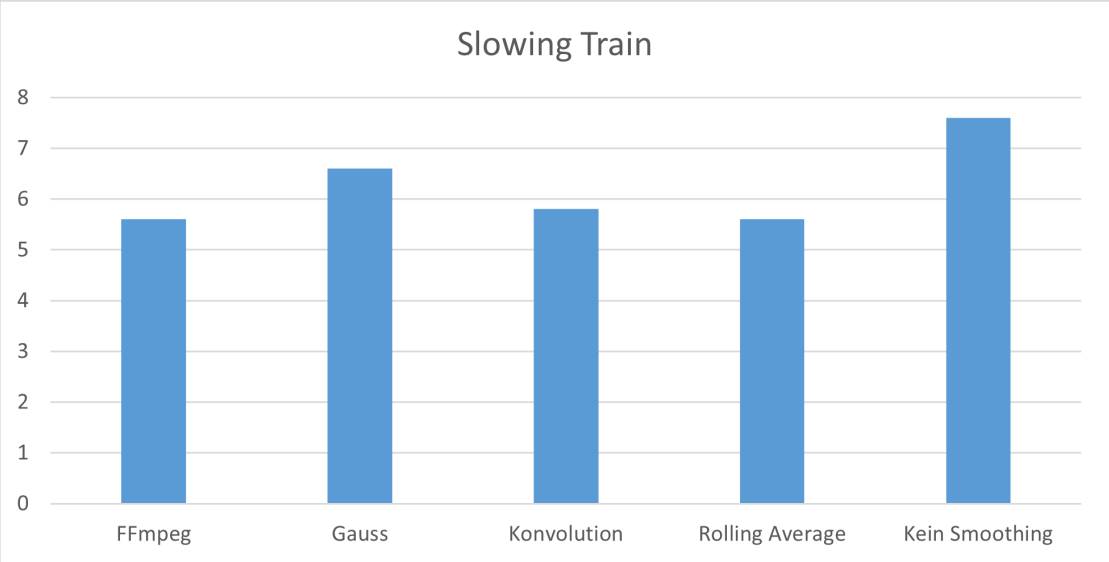


Abbildung 6.14: Benutzertest Slowing Train

Kein Smoothing in der Abbildung 6.14 zeigte die beste Stabilisierung mit den wenigsten Rucklern, mit Gauss als naher Zweitplatzierter.

Konvolution und Rolling Average hatten immer noch vermehrt Ruckler, welche klar sichtbar waren im Video.

FFmpeg startet sehr gut stabilisiert, jedoch sobald die Kamera sich aufhört zu bewegen, versuchte der Algorithmus dem Zug nach zu stabilisieren was zu unnötigen Bewegungen führte.

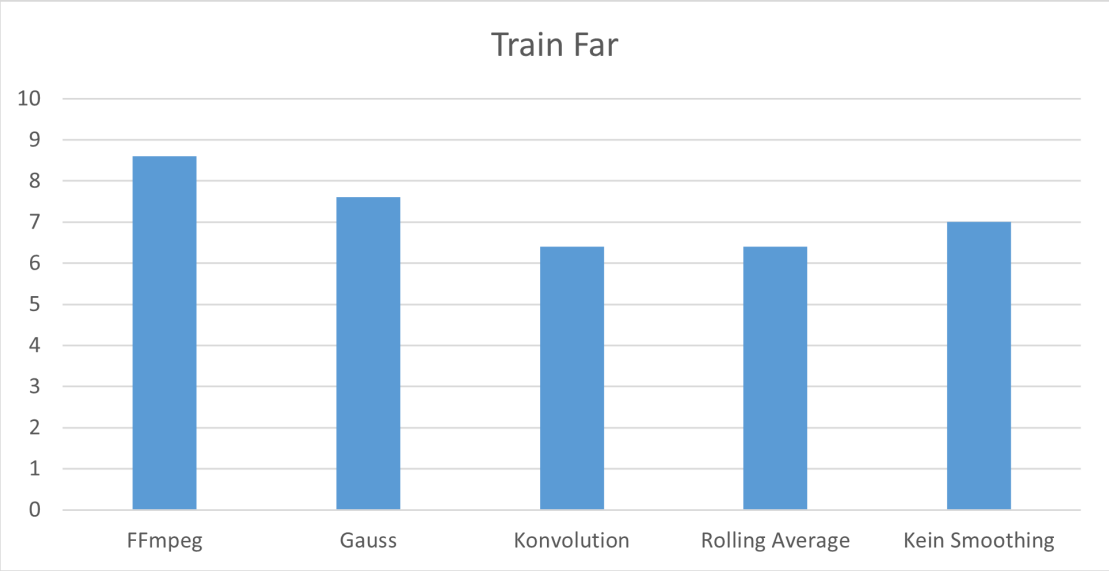


Abbildung 6.15: Benutzertest Train Far

FFmpeg in der Abbildung 6.15 zeigte eine solide Stabilisierung, welche dieses Mal nicht versuchte dem Zug nach zu stabilisieren.

Gauss zeigte auch eine gute Stabilisierung, nahe gefolgt von *Kein Smoothing*. Konvolution und Rolling Average ruckelten um einiges mehr, aber alle Algorithmen zeigten eine klare Verbesserung verglichen zum Originalvideo.

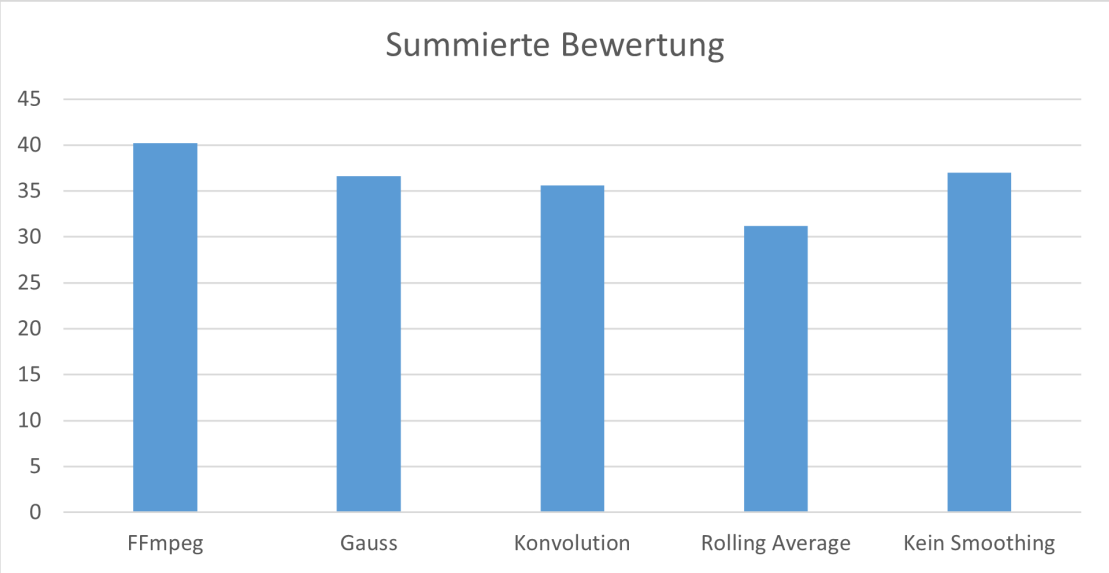


Abbildung 6.16: Summierte Bewertung der Benutzertests

Summiert man alle Punkte zusammen erhält man das Ergebnis, wie es in der Abbildung 6.16 zu sehen ist.
FFmpeg hat die meisten Punkte erhalten und stabilisiert die meisten Videos am besten. Jedoch hatte auch dieser seine schlechten Zeiten, welche bei zu starken Bewegungen oder bei zu grossen Beweglichen Objekten klar wird.

Gauss, Konvolution und *Kein Smoothing* haben beinahe die gleiche Punktzahl bekommen. Alle hatten ihre Spezialgebiete und sie können je nach Bedarf und Szenario miteinander ausgewechselt werden, um ein optimales Ergebnis zu erhalten.

Rolling Average hängt etwas hinterher. Es ist ein stabiler Algorithmus, welcher jedoch nie die höchste Punktzahl erhielt, sondern meistens in der Mitte oder etwas darunter war.

Glättung Fazit

Da FFmpeg nicht fähig ist, Streamingdaten zu verarbeiten, fällt dieser Leider weg. Qualitativ hatte er das beste Ergebnis geliefert und zeigt auch, wie gut eine Stabilisation mit nur Bild funktionieren kann.

Als nächste fällt auch Rolling Average aus dem Rennen. Geschwindigkeitsmässig war er bereits der langsamste, wenn man die Pandas Algorithmen weglässt, aber auch die Bewertung von dem Algorithmus waren die schlechtesten.

Konvolution und Gauss sind beinahe gleich. Beide funktionieren auch mit Konvolution mit dem einzigen Unterschied, dass Gauss eine Gaussverteilung benutzt bei der Konvolution, während der Konvolution Algorithmus mit einer Box arbeitet. Die Geschwindigkeit wie auch Qualität von beiden Algorithmen sind beinahe gleich.

Kein Smoothing ist ein überraschender Weg, um mit wenig Zeit trotzdem ein gutes Ergebnis zu erzielen. Er hat die höchste Geschwindigkeit von allen, da er die Trajektorie direkt zurückgibt ohne zu glätten und gleichzeitig ist das stabilisierte Bild trotzdem auf gleichem Level wie Gauss und Konvolution.

Im Fazit kann gesagt werden, dass Gauss, Konvolution und *Kein Smoothing* alle gleich benutzt werden können. Die Entscheidung sollte für die jeweiligen Videoszenarien gemacht werden, statt eine Allgemeinlösung zu nehmen. Auch der Geschwindigkeitsvorteil von *Kein Smoothing* ist nur geringfügig, da die anderen Algorithmen alle auch unter einer Millisekunde sind und somit keinen grösseren Einfluss auf das ganze haben.

6.6 AWS Tests

In der AWS-Umgebung sind die Tests nur mit Streams gemacht worden, um die Machbarkeit, Geschwindigkeit und Qualität zu testen. Dazu wurde der Stream von einem Handy genommen, welches mittels der Haivision Play Pro einen Stream auf AWS IVS sendet. Alternative wurde auch OBS-Studio benutzt, um einen Stream zu senden. Alle Streams wurden in 720p mit 30 FPS und eine Bitrate von ca. 2.5 Mbit/s gesendet.

6.6.1 AWS Instanz

Da die Kosten eine Rolle spielen, wurden die Tests zunächst mit einer möglichst kleinen Instanz (t2.micro) ausgeführt. Jedoch zeigte sich sofort, dass die Instanz zu klein war und die Ausführung der Applikation nicht möglich ist.
Die Nachforschungen im Abschnitt 4.1.2 zeigen, das eine c6i Instanz am besten geeignet ist für rechenintensive Aufgaben, wie Videokodierung, deswegen wurde eine c6i.xlarge und c6i.2xlarge Instanz benutzt.

Eigenschaft	c6i.xlarge	c6i.2xlarge
vCPUs	4	8
RAM	8 GiB	16 GiB
Kosten	0.194 USD pro Stunde	0.388 USD pro Stunde
Stabilität	Stösst an ihrer Grenze, führt zu einem ruckeligem Bild	Stabiles Bild bei den Tests

Tabelle 6.1: Vergleich der AWS Instanzen c6i.xlarge und c6i.2xlarge

Die c6i.xlarge Instanz stiess während den Tests an ihre Leistungsgrenze. Dies äusserte sich in einer dauerhaft hohen CPU-Auslastung, wie in Abbildung 6.17 dargestellt. Infolge der begrenzten Rechenressourcen war es nicht möglich, alle eingehenden Frames im vorgesehenen 30-FPS-Takt zu verarbeiten. Dadurch füllte sich die Warteschlange kontinuierlich, was zu einer ruckelnden Darstellung führte. Im Gegensatz dazu konnte die leistungsstärkere c6i.2xlarge Instanz die gesamte Verarbeitungslast bewältigen. Alle Frames wurden rechtzeitig stabilisiert, wodurch ein flüssiges und stabiles Bildresultat erzielt wurde.

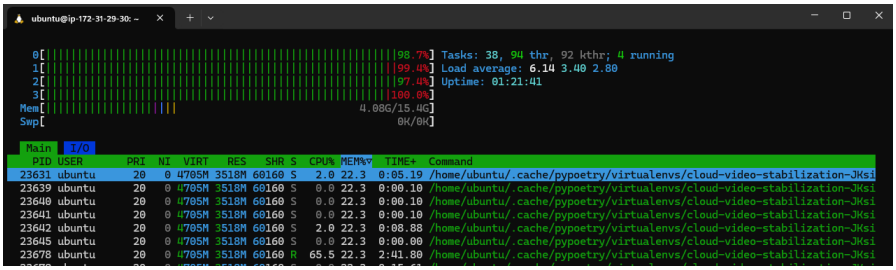


Abbildung 6.17: CPU/RAM Last der c6i.xlarge Instanz

6.6.2 Queue Grösse

Bei 10000 Bildern für die Queue Grösse, läuft das Memory bei einer EC2 Instanz c6i.xlarge mit 8GB Ram und 4 CPUs voll. Gleiches ist auch bei der t3.xlarge mit 16GB RAM und 4 CPUs der Fall. Wird die Queue Grösse auf 1000 Bilder reduziert, pendelt sich der Memory Verbrauch zwischen 3.7 - 4.2 GB ein, siehe Abbildung 6.17. Die CPU-Auslastung der 4 CPUS liegt bei 100% und der Stream läuft mit 30 FPS bei 720p, jedoch wird der Stream immer wieder kurz unterbrochen, da die CPU nicht schnell genug ist, um die Bilder zu verarbeiten.

6.6.3 Geschwindigkeitstests

Die Geschwindigkeitstests wurden mit dem GNU Time Programm durchgeführt. Dieses Befehlszeilenprogramm misst die Zeit, welche einen Befehl braucht, um ausgeführt zu werden. Dabei wird die Zeit von der Ausführung des Befehles bis zum Ende gemessen. Auch wird die Benutzer Zeit, die System Zeit, CPU-Last, die Speicherbenutzung sowie die Anzahl Kontext Switches gemessen. Dadurch wird ein gutes Bild der verschiedenen Kennzahlen gegeben, welche für die Performance wichtig sind.

Nr	CPU-Last (%)	Laufzeit (min:s)	User Time (s)	System Time (s)	RAM max (GB)	Minor PF	Voluntary CS	I/O (R/W)
1	264	1:49	243.06	46.03	4.65	5,808,763	334,985	72/64
2	289	3:20	516.13	64.95	4.67	4,662,444	460,899	0/96
3	232	7:09	723.27	276.15	3.24	26,660,107	1,504,369	0/96
4	411	4:02	820.78	178.00	3.53	25,039,634	1,112,383	0/144
5	232	5:09	510.10	210.17	2.22	22,892,340	1,043,891	0/192

Tabelle 6.2: Zusammenfassung der AWS Performance Tests

In der Abbildung 6.18 ist ein Beispiel für die Ausgabe von GNU Time zu sehen, auf welchen die oben genannten Kennzahlen basieren. Die Werte in der Tabelle 6.2 sind aus mehreren Tests aggregiert und geben einen Überblick über die Performance der Anwendung auf der AWS EC2 Instanz.

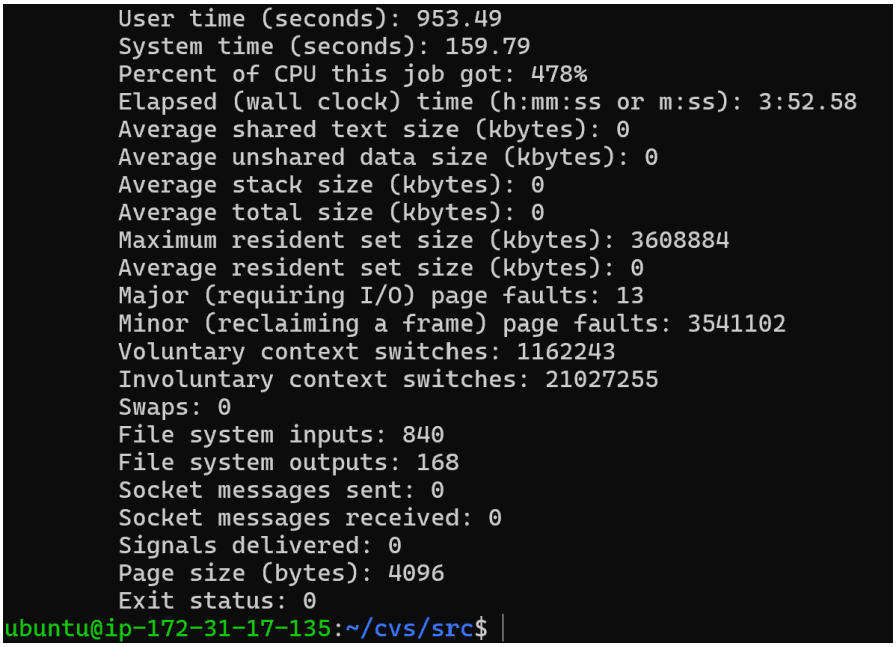


Abbildung 6.18: Beispielausgabe von GNU Time für einen Test

Kurz zusammengefasst kann über die Applikation folgendes gesagt werden: Die Anwendung nutzt mehrere CPU-Kerne effizient, arbeitet komplett im RAM ohne Festplattenzugriffe und erzeugt sehr viele Threads und Speicherzugriffe. Dies ist typisch für hochparallele, speicherintensive Bild- oder Videobearbeitung.
Es sind keine I/O Engpässe erkennbar, da ausschliesslich die Nutzung von RAM und CPU die Performance limitiert. Hier einige Interessante Kennzahlen, welche aus der Tabelle 6.2 entnommen wurden:

- **Parallelisierungsgrad (Effizienz der Multicore-Nutzung)**
  - **Kennzahl:** (User Time + System Time) / Laufzeit = Durchschnittliche Anzahl gleichzeitig genutzter CPU-Kerne.
  - **Nutzen:** Zeigt, ob und wie effizient die Anwendung mehrere Kerne nutzt (Werte über 100% zeigen echte Parallelisierung).
- **CPU-Bound vs. I/O-Bound**
  - **Kennzahl:** CPU-Last, Major Page Faults, File System Inputs/Outputs.
  - **Nutzen:**
    - **CPU-bound:** Hoher CPU-Anteil, wenige Page Faults, wenig I/O → Rechenlast dominiert, was hier der Fall ist.
    - **I/O-bound:** Viele I/O-Operationen, Major Page Faults, niedrige CPU-Auslastung → Warten auf Datenzugriffe bremst.

- **Speicherauslastung**
  - **Kennzahl:** Maximum Resident Set Size (RAM max).
  - **Nutzen:** Zeigt, wie viel RAM im Peak benötigt wird, ob Engpässe auftreten und ermöglicht den Vergleich verschiedener Algorithmen.

6.6.4 Container Vergleich

Der Vergleich mit AWS ECS oder AWS App Runner konnte wir nicht durchführen, da der Benutzeraccount nicht die nötigen Rechte hatte, um die Container zu starten. Jedoch wurde der Funktionstest mit Docker auf dem lokalen Rechner durchgeführt. Dieser ergab grundsätzlich ein gleichwertiges Bild wie die direkte Ausführung der Applikation auf dem lokalen Rechner. Da die Applikation lokal auf einer Windows Docker Umgebung getestet wurde, ist die Performance jedoch nicht vergleichbar mit der auf einer Linux Umgebung.

6.6.5 AWS Kosten

Das Verhältnis der Stabilisierungskosten zu den Streaming Kosten auf AWS IVS ist extrem. In den Tests macht die Stabilisierung 96% der Gesamtkosten aus. Der Streaming Service IVS kostet hingegen nur 4% der Gesamtkosten. Dies Wird sehr deutlich in Abbildung 6.19 dargestellt, wobei Rot der Balken für den IVS Service ist.

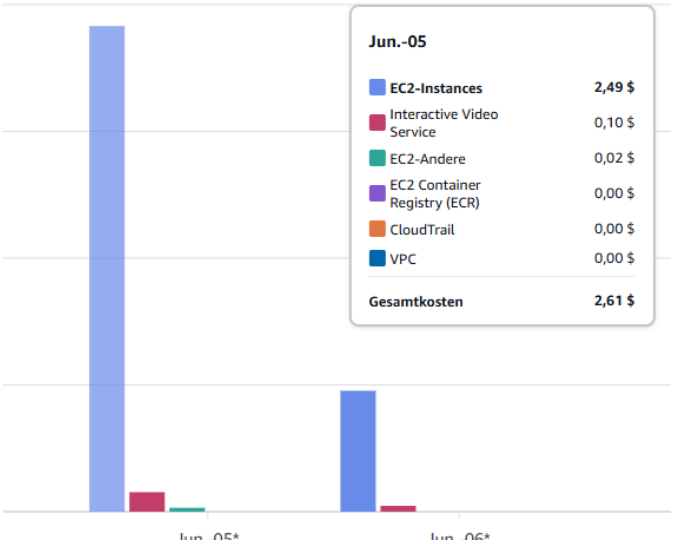


Abbildung 6.19: Kosten für AWS IVS vs EC2 Streaming

6.7 Schlussfolgerung

Die Evaluation hat gezeigt, dass eine Echtzeit-Videostabilisation mit 30 FPS technisch möglich ist, jedoch eine hohe Rechenleistung erfordert. Python bietet trotz geringerer Geschwindigkeit dank einfacher Entwicklung und C++-basierter Bibliotheken eine praktikable Basis.
Good Features to Track erwies sich als bester Kompromiss bei der Merkmalerkennung. Für die Bewegungsfluss- und Transformationsberechnung lieferten *Lucas-Kanade* und *Estimate Affine* die besten Ergebnisse. Beim Glätten überzeugten *Gauss*, *Konvolution* und *Kein Smoothing* je nach Szenario.
Die Tests in der AWS-Umgebung bestätigten die grundsätzliche Machbarkeit der Lösung im Cloud-Streaming-Kontext, wiesen jedoch auf die Notwendigkeit leistungsfähiger Instanzen hin, um eine ruckelfreie Verarbeitung zu gewährleisten. Besonders die Instanz *c6i.2xlarge* konnte ein stabiles Resultat liefern, während kleinere Instanzen schnell an ihre Grenzen stiessen.
Insgesamt zeigt die Evaluation, dass eine Echtzeit-Videostabilisation mit den richtigen Algorithmen und ausreichend Rechenleistung realisierbar ist. Sind die Kosten jedoch entscheidend, sollte eine offline Stabilisierung mit dem Aufnahmegerät in Betracht gezogen werden, da diese eine bessere Qualität und Stabilität bei geringeren Kosten bietet.

Kapitel 7

Ausblick

Da sich das Projekt derzeit in der Phase eines Proof of Concept befindet, bestehen zahlreiche Möglichkeiten zur Optimierung und Weiterentwicklung.

- Weitere Algorithmen können getestet werden, welche ein potenziell schnelleres oder qualitativ besseres Ergebnis erzielen. Ein Weg wäre, einen 3D-Algorithmus zu benutzen und zu prüfen, ob dieser besser abschneidet. Dafür müssen jedoch bei der Übertragung auch die intrinsischen Kameraparameter mitgeschickt werden.
- Beim Testen wurde klar, dass verschiedene Szenarien eine andere Konfiguration benötigen. Zum Beispiel sollte ein Video, bei welchem die Kamera sich nicht bewegt, sondern die Objekte darin, anders konfiguriert werden als ein Selfie-Video. Dazu gehören:
  - Smoothing Window
  - Zoomfaktor
  - Videoauflösung
  - Algorithmen-spezifische Konfigurationen

  Diese verschiedenen Szenarien können optimiert werden und somit mit szenarienspezifischen Einstellungen stabilisiert werden, wodurch eine bestmögliche Qualität erreicht wird.
- Zum andere kann man auch die Stabilisation parallelisieren. Das Projekt selbst hat bereits ein Queue System. Es wird ein Zeitstempel oder ein anderer Weg nötig sein, um die Reihenfolge der Frames nicht zu verlieren, aber sobald dieser vorhanden ist, können mehrere Stabilisierer die Queue abarbeiten und die 30 Frames pro Sekunde können dadurch besser erreicht werden. Natürlich erhöht das auch die Kosten in AWS und muss deswegen evaluiert werden, ob es sich lohnt.
- Gyrodaten ermöglichen eine robustere Bildstabilisierung, da Bewegungen – etwa entlang der X-Achse – bereits vor der Bildverarbeitung erkannt werden können.

Bei ausreichend hoher Qualität der Gyrodaten und einem überzeugenden Stabilisierungsergebnis kann unter Umständen vollständig auf die Bildanalyse verzichtet werden, was Rechenleistung spart. Die Übertragung der Gyrodaten kann effizient und zuverlässig über das latenzoptimierte SRT-Protokoll erfolgen.

- Um die Übertragungslatenz zu verringern, könnte der erste IVS-Service (Up-Stream) durch die Stabilisierungsapplikation mit integriertem SRT-Server ersetzt werden. Dadurch liesse sich das Pufferung im IVS-Service vermeiden, das je nach Modus zwischen 5 und 30 Sekunden beträgt. Auch das zusätzliche Pufferung von etwa einer Sekunde, das im aktuellen Setup mit zwei IVS-Services beim Auslesen des Streams entsteht, könnte dadurch entfallen.
- Eine weitere Möglichkeit zur Optimierung der Stabilisierung besteht darin, das aktuell verwendete Pixelformat von BGR24 (Blau-Grün-Rot, 24 Bit pro Pixel) auf das bereits im Stream genutzte YUV420p-Format umzustellen. Da YUV420p speichereffizienter ist, könnte dies die CPU-Last deutlich reduzieren und somit die Performance verbessern, sofern das Ergebnis der Stabilisierung zufriedenstellend ist.

Die genannten Optimierungsmöglichkeiten bieten viel Potenzial, um die Bildstabilisierung in der Cloud weiter zu verbessern und an die spezifischen Anforderungen verschiedener Anwendungsfälle anzupassen. Die Implementierung dieser Ideen könnte die Effizienz und Qualität der Bildstabilisierung erheblich steigern und möglicherweise die Kosten im Rahmen halten.

Teil IV

Verzeichnisse

Literaturverzeichnis

[1] Inc. Amazon Web Services. Ec2 c6i instances, 08-06-2025. URL: https://aws.amazon.com/de/ec2/instance-types/c6i/.

[2] Wenbo Bao, Xiaoyun Zhang, Li Chen, and Zhiyong Gao. Kalmanflow: Efficient kalman filtering for video optical flow. In *2018 25th IEEE International Conference on Image Processing (ICIP)*, pages 3343–3347. IEEE, 2018. doi:10.1109/ICIP.2018.8451564.

[3] M. Berger, M. Cole, and S. Levy. *Geometry I*. Universitext. Springer Berlin Heidelberg, 2009. URL: https://books.google.ch/books?id=5W6cnfQegYcC.

[4] Jinsoo Choi and In So Kweon. Difrint: Deep iterative frame interpolation for full-frame video stabilization. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3732–3736. IEEE, 2019. doi:10.1109/ICCVW.2019.00463.

[5] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In Josef Bigun and Tomas Gustavsson, editors, *Image Analysis*, pages 363–370, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[6] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, June 1981. doi:10.1145/358669.358692.

[7] C. Forbes, M. Evans, N. Hastings, and B. Peacock. *Statistical Distributions*. Wiley Series in Probability and Statistics - Applied Probability and Statistics Section Series. Wiley, 2010. URL: https://books.google.ch/books?id=0A4AJZsn0pMC.

[8] Google. Google c++ style guide, 24-04-2025. URL: https://google.github.io/styleguide/cppguide.html.

[9] Rob Hyndman. *Moving Averages*, pages 866–869. 01 2010. doi:10.1007/978-3-642-04898-2_380.

[10] Ambv Software Inc. Black, 31-05-2025. URL: https://black.readthedocs.io/en/stable/.

[11] Astral Software Inc. Ruff, 31-05-2025. URL: https://astral.sh/ruff.

[12] Haivision Systems Inc. Srt: Secure, reliable, transport, 11-06-2025. URL: https://github.com/Haivision/srt.

[13] A. Leonardis, A.P.H.B. Aleš Leonardis, and A. Pinz. *Computer Vision - ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006 : Proceedings*. Computer Vision: ECCV 2006. Springer, 2006. URL: https://books.google.ch/books?id=ITvZ9HDPizcC.

[14] KENNETH LEVENBERG. A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944. URL: http://www.jstor.org/stable/43633451.

[15] Shuaicheng Liu, Ping Tan, Lu Yuan, Jian Sun, and Bing Zeng. Meshflow: Minimum latency online video stabilization. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer vision - ECCV 2016*, Lecture notes in computer science, pages 800–815, Cham, 2016. Springer. URL: https://link.springer.com/chapter/10.1007/978-3-319-46466-4_48, doi:10.1007/978-3-319-46466-4{\textunderscore}48.

[16] D.G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999. doi:10.1109/ICCV.1999.790410.

[17] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, page 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[18] Krystian Mikolajczyk and Cordelia Schmid. An affine invariant interest point detector. In Anders Heyden, Gunnar Sparr, Mads Nielsen, and Peter Johansen, editors, *Computer Vision — ECCV 2002*, pages 128–142, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[19] Pandas. pandas.dataframe.ewm, 02-06-2025. URL: https://pandas.pydata.org/docs/user_guide/window.html#exponentially-weighted-window.

[20] Pandas. pandas.dataframe.expanding, 02-06-2025. URL: https://pandas.pydata.org/docs/user_guide/window.html#expanding-window.

[21] David Peter. Hyperfine: A command-line benchmarking tool, 08-06-2025. URL: https://github.com/sharkdp/hyperfine.

[22] S.W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997. URL: https://books.google.ch/books?id=rp2VQgAACAAJ.

[23] OpenCV team. Opencv, 08-06-2025. URL: https://opencv.org.

[24] The Clang Team. Clangformat, 24-04-2025. URL: https://clang.llvm.org/docs/ClangFormat.html.

[25] Write the Docs. Documentation principles, 28-02-2025. URL: https://www.writethedocs.org/guide/writing/docs-principles/.

[26] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. Pep 8 – style guide for python code, 28-02-2025. URL: https://peps.python.org/pep-0008/.

[27] Yufei Xu, Jing Zhang, Stephen J. Maybank, and Dacheng Tao. Dut: Learning video stabilization by simply watching unstable videos. URL: https://arxiv.org/pdf/2011.14574v3.

[28] Kota Yamaguchi. cv.estimateaffinepartial2d, 02-06-2025. URL: https://amroamroamro.github.io/mexopencv/matlab/cv.estimateAffinePartial2D.html.

[29] Minda Zhao and Qiang Ling. Pwstablenet: Learning pixel-wise warping maps for video stabilization. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 2020. doi:10.1109/TIP.2019.2963380.

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

CI/CD (Continuous Integration/Continuous Deployment) beschreibt Prozesse und Praktiken in der Softwareentwicklung, die eine schnelle und automatisierte Bereitstellung von Anwendungen ermöglichen. 22

CPU Central Processing Unit. 27

Docker Eine Plattform zur Automatisierung der Bereitstellung von Anwendungen in Containern. 25

FFmpeg Eine vollständige, plattformübergreifende Lösung zum Aufnehmen, Konvertieren und Streamen von Audio und Video `https://ffmpeg.org/`. 5, 13, 24, 28, 44, 45, 46, 48, 62, 63, 65, 66, 67, 68, 69

FPS frames per seconds (Bilder pro Sekunde). 29

GPU Graphical Processing Unit. 33

OBS Studio Open Broadcaster Software Studio, eine Open-Source-Software für Videoaufzeichnung und Live-Streaming. 27, 28

OpenCV Open Computer Vision Library, eine Open-Source-Bibliothek für Computer Vision. 27

OS Operating System, auf deutsch Betriebssystem. 24

Poetry Ein Tool zur Verwaltung von Python-Projekten und deren Abhängigkeiten. 24

RAM Random-Access Memory. 27

SLS Edward Wu's SRT Live Server, ein Streaming-Server, der das SRT-Protokoll verwendet `https://github.com/Edward-Wu/srt-live-server`. 12, 27

SRT Secure Reliable Transport. 5

UDP User Datagram Protocol. 28

VideoLAN Client VLC Media Player, ein freier und quelloffener Multimedia-Player, der eine Vielzahl von Audio- und Videoformaten abspielen kann https://www.videolan.org/. 27

Hilfsmittelverzeichnis

Aufgabenbereich	Tools
Literatur-Recherche und Verwaltung	Google, swisscovery, Citavi, ChatGPT
Datenanalyse und Visualisierung	Python, Excel, ChatGPT
Übersetzung	DeepL, Google Translate, ChatGPT
Diagramme und Grafiken	draw.io, Blender, GIMP
Textoptimierung, Rechtschreibe und Grammatikprüfung	Word, DeepL, ChatGPT
Zusammenarbeit und Projektmanagement	GitLab, Teams, Outlook, Jira
Video- und Streaming	OBS Studio, FFmpeg, DaVinci Resolve, Haivision Play Pro
DevOps	Docker, AWS, GitLab CI/CD