Oskar-Maria-Graf-Gymnasium
Neufahrn b. Freising

Abiturjahrgang
2017

S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Numerik

Leitfach: **Informatik**

Thema der Arbeit:

Extrapolation eines drei-dimensionalen Modells aus zwei-dimensionalen Fotos

Verfasser/in: **Lina van Brügge**

Kursleiter/in: **StR Sven Baumer**

Abgabetermin:
(2. Unterrichtstag im November)

08. November 2016

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 61 (7) GSO = Summe:2 (gerundet)	

Datum und Unterschrift der Kursleiterin bzw. des Kursleiters

Inhaltsverzeichnis

1 Einführung

1.1 Einleitung in die Numerik

Numerik[1] beschäftigt sich mit alltäglichen, mathematischen Problemen wie zum Beispiel die Berechnung eines schwingenden Fadenpendels und versucht, diese durch ein Verfahren zu lösen, entweder per Hand oder mithilfe eines Computers. Folgende Punkte müssen dabei berücksichtigt werden:

- **Interpretierbarkeit:** Der Anwender soll die Möglichkeit haben, das Ergebnis zu verstehen.
- **Effizienz:** Das Verfahren soll das Problem möglichst in einer angemessenen Zeitspanne lösen.
- **Genauigkeit:** Das Ergebnis des Verfahrens soll dabei in einem vorher festgelegten Toleranzbereich liegen.

Man unterscheidet zwischen **approximierenden** (nähernden) und **iterativen** (wiederholenden) Verfahren.

Die Kreiszahl π lässt sich beispielsweise unmöglich exakt berechnen. Stattdessen errechnet man eine Näherung wie $3,14159$, um π für den Anwender in eine lesbare Form zu bringen.

Beim Newton-Verfahren hingegen berechnet man exakt die Nullstellen einer Funktion; allerdings gelingt dies meist nur durch viele Wiederholungen. Damit das Verfahren effizient bleibt, setzt man hierfür einen Computer ein.

1.2 Einleitung in Computeranimationen

Computeranimationen, wie man sie heutzutage überall in Filmen oder PC-Spielen findet, werden immer realistischer und ihr Aussehen oder ihre Bewegungen immer menschenähnlicher. Dies wird durch moderne Bewegungserkennung/Bewegungserfassung, sogenanntes Motion Capture erreicht. Hierbei wird ein Schauspieler mit verschiedenen Kameras in einem Studio gefilmt. An seinem Körper oder in seinem Gesicht sind währenddessen Marker fixiert, die entweder aktiv oder passiv Signale emittieren. Passive Marker sind beispielsweise weiße oder schwarze Punkte, aktive Marker emittieren Infrarot-Licht oder Ähnliches. Die von den Kameras erfassten Markersignale werden anschließend von einem Computer analysiert, weiterverarbeitet und auf ein Skelett übertragen, welches die Bewegung des Schauspielers später in der Animation nachahmt.

[1] Neuss 2010.

Abbildung 1: Mark Hamill bei dem Dreh für die Zwischensequenzen in dem Computerspiel „Star Citizen“, Quelle: pcgameshardware.de[2]

Leider ist dieses Verfahren sehr aufwendig, da zusätzlich zu den Aufnahmen und der Übertragung auf das Skelett manuell ein Modell erstellt werden muss, auf das die Bewegungen projiziert werden.

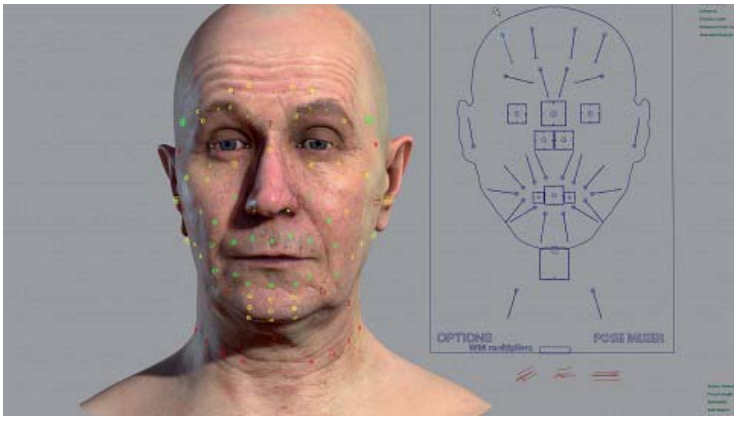

Abbildung 2: Computermodell von Mark Hamills Gesicht für die Figur Steve Colton in „Star Citizen“, Quelle: golem.de[3]

1.3 Ziel der Arbeit

In dieser Seminararbeit soll daher mithilfe eines iterativen Verfahrens eine alternative Technik zum manuellen Erstellen des Computermodells entwickelt werden. Diese Technik soll ein dreidimensionales und wirklichkeitsgetreues Modell eines Gesichts aus drei herkömmlichen, zweidimensionalen Fotos von Webcams erstellen können. Der zu entwickelnde Algorithmus sollte dabei auch in der Lage sein, bei verschiedenen Gesichtsausdrücken verschiedene Modelle des gleichen Gesichtes zu erstellen. Das Formel zum Finden der Punkte wurde dabei eigenständig hergeleitet.

In einem weiteren Schritt, der allerdings nicht mehr Teil dieser Seminararbeit sein wird, könnte man den Algorithmus auf kürzere Video-Sequenzen erweitern und somit

Gefühlsausdrücke modellieren oder Interpolation zwischen Gesichtern ausdrücken. Der für diese Arbeit implementierte Algorithmus benötigt allerdings noch viele Optimierungen, damit die in Kapitel 1.1 genannten Punkte erfüllt werden (siehe auch Kapitel 5).

2 Grundlage dieser Arbeit

In dieser Arbeit wurde sich beim Kamera-Aufbau an „High-Quality Single-Shot Capture of Facial Geometry: Implemention Details“[4] orientiert, allerdings in einem kleineren Umfang. Dort wird ein Verfahren mithilfe von sieben Kameras getestet, die in einem Halbkreis aufgestellt werden. Nach einer Kalibrierung der Kameras an die Lichtverhältnisse wird eine Computer-Maske anhand der Hautfarbe der fotografierten Person erstellt, um den Hintergrund der Fotos zu löschen. Anschließend werden immer zwei nebeneinander liegende Fotos miteinander verglichen, um zusammen gehörende Pixel zu finden (Matching). Dieses Matching wird in mehreren Schritten durchgeführt. Zuerst werden die Pixelfarben grob mithilfe eines Block-Matching-Algorithmus[5] sortiert. Die gefunden Pixel auf den zwei Fotos werden anschließend mit ihren genauen Farbwerten verglichen. Bei einer sehr hohen Ähnlichkeit werden die Tiefendaten dieser Pixel ausgerechnet und zusätzlich zu den Farbwerten gespeichert. Die Übereinstimmung wird nun noch einmal auf ihre Richtigkeit überprüft. Ist dieser Test positiv, wird an dessen Stelle ein Punkt in den Raum gesetzt. Nachdem alle Fotos immer paarweise miteinander verglichen wurden, wird in mehreren Schritten zwischen den entstandenen Punkten solange inter- und extrapoliert, bis ein exaktes Modell des Gesichtes entstanden ist.

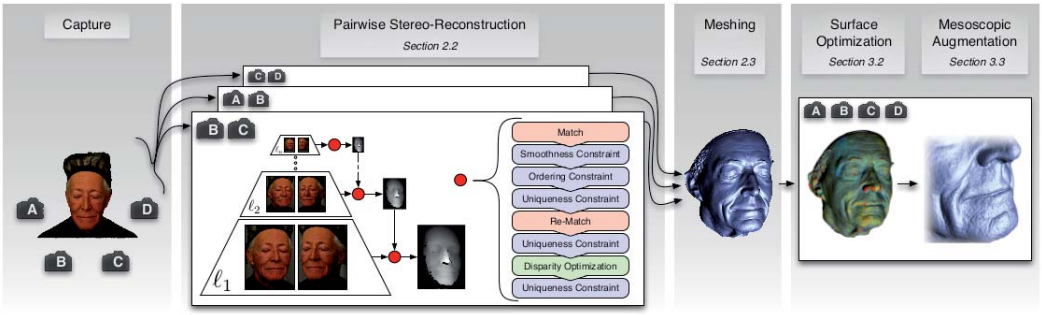

Figure 2: The proposed system - *The subject is captured with multiple cameras. This figure shows a four-camera setup, but the system can incorporate an arbitrary number of cameras.*

Abbildung 3: Darstellung des Verfahrens mit vier Kameras, Quelle: Beeler u. a. 2010

[4] Beeler u. a. 2010.
[5] *Block-matching algorithm.*

3 Das Verfahren

3.1 Die Ausgangslage

Die Grundlage für den zu implementierenden Algorithmus bilden drei Fotos, die von drei Webcams aus verschiedenen Blickwinkeln aufgenommen werden. Eine Kamera ist frontal zum Objekt ausgerichtet, die anderen beiden Kameras haben einen Abstand von 30 cm und stehen im 45° Winkel zu der Front-Kamera an den Seiten.

Exemplarischer Aufbau der Kameras und deren ungefähren Abstand zum Objekt

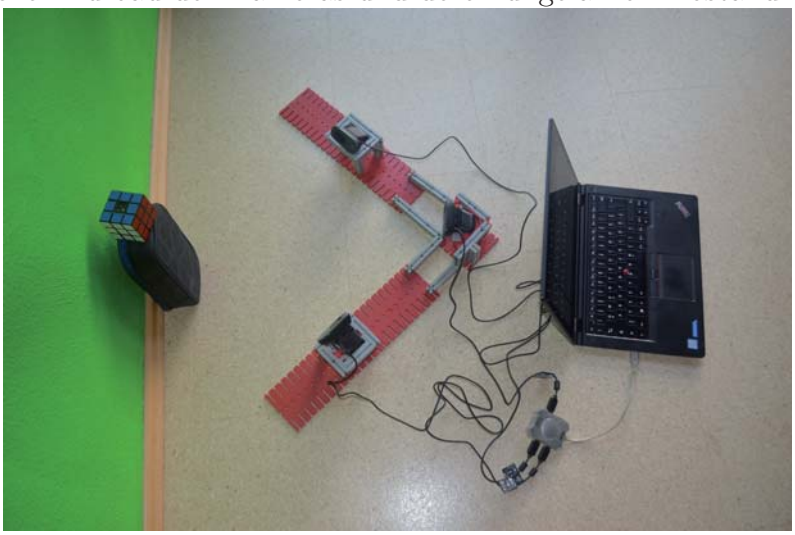

Abbildung 4: Aufsicht, Quelle: eigenes Foto

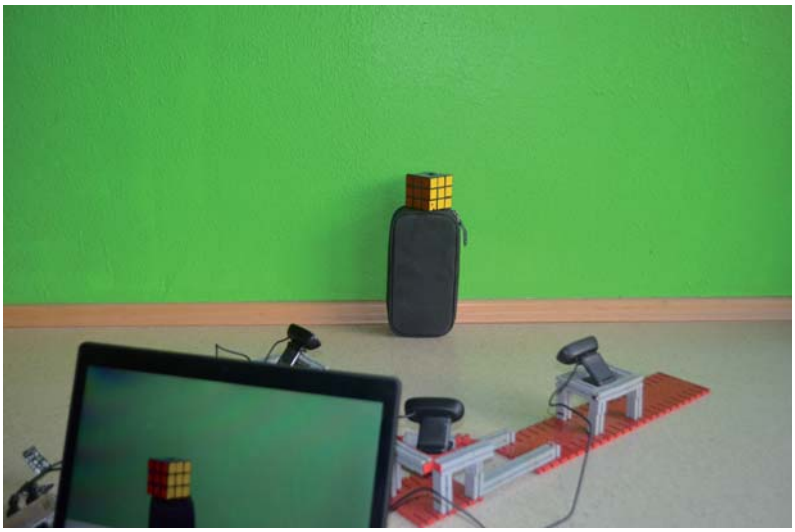

Abbildung 5: Frontsicht, Quelle: eigenes Foto

Um den hergeleiteten Algorithmus an verschiedenen Objekten zu testen, wurden drei Fotoreihen aufgenommen, zuerst von einem Zauberwürfel, dann von einem Gesicht und zum Schluss von einem Kuscheltier.

Abbildung 6: Foto des Zauberwürfels von rechts, frontal und von links,
Quelle: eigene Fotos, bearbeitet

Abbildung 7: Fotos des Gesichts von rechts, frontal und von links,
Quelle: eigene Fotos, bearbeitet

Abbildung 8: Fotos des Kuscheltiers von rechts, frontal und von links,
Quelle: eigene Fotos, bearbeitet

3.2 Übersicht über das Verfahren

Bei der Erstellung des Computer-Modells sind drei Aspekte von zentraler Bedeutung, welche im weiteren Verlauf dieser Arbeit näher erläutert werden:

- **Entfernen des Hintergrundes:** Da das Objekt nur erkannt werden kann, wenn der komplette Hintergrund gelöscht ist, muss dieser zuerst aus den Bildern möglichst vollständig entfernt werden (siehe Kapitel 3.3).
- **Verkleinern der Datenmenge:** Da bei einem HD-Bild mit $1920 * 1080$ Pixeln die zu überprüfende Datenmenge sehr groß ist und um falsche Zuordnungen zu vermeiden, wird anschließend das Bild skaliert, um die Datenmenge zu verringern (siehe Kapitel 3.4).
- **Auswählen des passenden Pixels:** Durch einen Abgleich der Farbwerte (RGB-Werte) aller Pixel aus der Vorauswahl (siehe Kapitel 3.4) wird der Pixel mit der höchsten Übereinstimmung ausgewählt (siehe Kapitel 3.5).

3.3 Entfernen des Hintergrundes

Um den Hintergrund möglichst komplett aus den Fotos zu entfernen, wurde bei den Aufnahmen ein Greenscreen[6] benutzt. Mithilfe von OpenCV[7], einer Open Source Bibliothek, die auf Bildverarbeitung in Echt-Zeit sowie ihre Weiterverarbeitung spezialisiert ist, lässt sich dieser leicht löschen.

3.3.1 Maske erstellen

Für das Entfernen wird das RGB-Foto in drei 8-Bit-Fotos aufgeteilt, in denen die Graustufen jeweils den Rot-, Grün- und Blau-Anteilen im ursprünglichen Foto entsprechen.

Abbildung 9: Fotos des Zauberwürfels in Rot-,Grün- und Blau-Anteile aufgeteilt, Quelle: eigene Fotos

Nun wird mithilfe der OpenCV-Funktion `threshold()` die Werte der Pixel überprüft. Liegen sie über einem, zuvor durch Tests ermittelten, Wert, werden sie auf den Maximalwert, d. h. weiß, gesetzt, ansonsten auf den Minimalwert, d. h. schwarz. Dabei ist zu beachten, dass das blaue Bild invertiert[8] werden muss, da der Blauanteil im Greenscreen zu hoch ist und sonst alles außer dem grünen Hintergrund entfernt werden würde. Die bearbeiteten Bilder werden wieder zusammengefügt und anschließend als Maske für das ursprüngliche Bild benutzt. Ist der RGB-Wert des Pixels in der Maske komplett weiß, so wird der Wert des Pixels auch im ursprünglichen Bild auf weiß gesetzt.

Abbildung 10: Maske und daraus resultierendes Foto mit gelöschten Greenscreen, Quelle: eigenes Foto

[6] Ein Greenscreen ist ein einfarbiger, meist grüner Hintergrund, der sich leicht entfernen lässt.
[7] *OpenCV*.
[8] Wird ein Bild invertiert, wird dessen Negativ erstellt.

3.3.2 Fehlerbehebung

Durch dieses Verfahren kommt es zu unvermeidbaren Falscherkennungen, da manche Pixel zu nah an den Farbwerten des Greenscreens liegen und deshalb ebenfalls entfernt werden. Aus diesem Grund wird anschließend wieder mit der Funktion `threshold()` eine zweite Maske mit möglichen Falscherkennungen erstellt und an dessen weißen Stellen mithilfe der Funktion `inpaint()` interpoliert wird, um die ungefähre Pixel-Farbe wieder herzustellen. Diese Methode kann allerdings nicht alle Leerstellen und auch die Farbe nicht komplett rekonstruieren. Je größer und komplexer dabei das Bild ist, desto eher werden Pixel falsch entfernt.

Abbildung 11: Foto vor der Fehlerbehebung, Maske für `inpaint()` und reproduziertes Foto, Quelle: eigene Fotos

3.4 Verkleinern der Datenmenge

Da die zu verarbeitende Pixelanzahl bei drei HD-Bildern selbst für heutige Computer so hoch ist, dass die Laufzeit für einen Abgleich extrem lang wäre, muss die Datenmenge verkleinert werden. Dies kann durch sogenannte epipolare Geometrie[9] erreicht werden (siehe auch Beeler u. a. 2010). Man geht dabei davon aus, dass alle möglichen Pixel ausgehend von einem beliebigen Pixel auf einem Bild auf dem anderen Bild eine Linie bilden, auch Epipolarlinie genannt. Durch das Suchen der Pixel auf dieser Linie wird die Anzahl der möglichen Übereinstimmungen für den ausgewählten Pixel im anderen Bild auf einen Bruchteil reduziert. Sollte es für den Pixel im Anfangs-Bild keine Übereinstimmung geben, lässt sich auch keine Epipolarlinie auf dem anderen Foto finden.

3.4.1 Datenreduzierung durch Erzeugung von Strahlen

Im hier angewandten Verfahren wird zum Reduzieren der Datenmenge auf Strahlen zurückgegriffen. Treffen sich zwei erzeugte Strahlen, die jeweils von der Kamera aus durch den ausgewählten Pixel führen, im Raum, so liegt der eine Pixel auf der gesuchten Epipolarlinie des anderen. Der Schnittpunkt der Strahlen wird in einer Liste gespeichert und ausgewertet (siehe Kapitel 3.5). Die Ortsvektoren der Startpunkte und die Richtungs-

[9] *Epipolargeometrie*

vektoren der Strahlen werden dabei anhand eines virtuellen Modells durch eine Visualisierung in dem Freeware-Geometrie-Programm GeoGebra ausgelesen und berechnet.

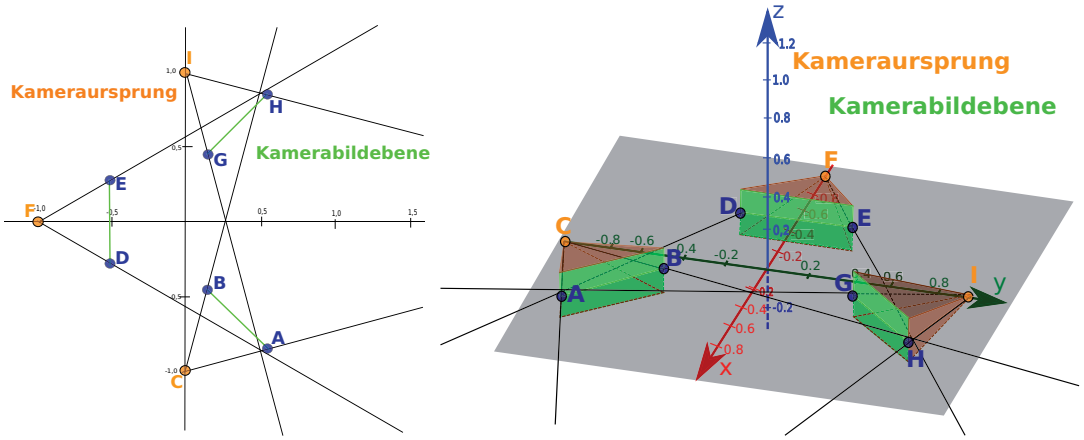

Abbildung 12: Visualisierung in GeoGebra, links Aufsicht, rechts Frontsicht, Quelle: eigene Darstellung

Der Ursprung der Strahlen ist jeweils der Kameraursprung, das heißt entweder der Punkt C(0/1/0), der Punkt E(−1/0/0) oder der Punkt B(0/1/0). Der Richtungsvektor $\vec{v}$ wird mithilfe von zwei for-Schleifen und von dessen abhängige Laufvariablen i und j erzeugt und kann durch folgende Formel beschrieben werden:

$$\vec{v} = \vec{o} + \vec{x} \cdot |\vec{x}| \cdot i + \vec{y} \cdot |\vec{y}| \cdot j - \vec{u} \tag{1}$$

Dabei berechnet man die x-Koordinate des Pixels durch den Vektor $\vec{o}$, der auf die obere, linke Ecke zeigt, zuzüglich den normalisierten Verbindungsvektor zwischen der oberen, linken und der oberen, rechten Ecke der Kamerabildebene ($\vec{x}$). Die y-Koordinate lässt sich von der oberen, linken Ecke der Kameraebene anhand des normalisierten Verbindungsvektors zwischen der oberen, linken und der unteren, linken Ecke ($\vec{y}$) berechnen. Der daraus resultierende Vektor ist der Ortsvektor des Pixels. Um jetzt den Verbindungsvektor mit der Kamera zu erhalten, muss man noch den Kameraursprung ($\vec{u}$) abziehen (siehe Abbildung 13).

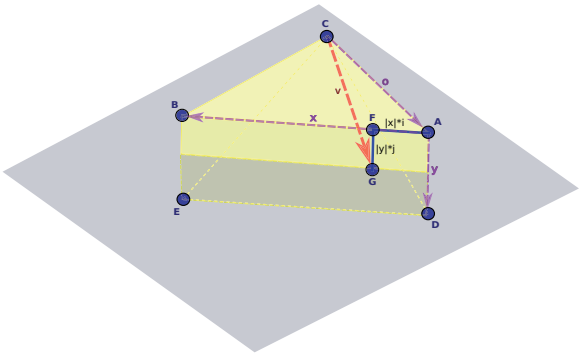

Abbildung 13: Berechnung des Richtungsvektor, Quelle: eigene Darstellung

3.4.2 Abstand der beiden Strahlen

Nun wird überprüft, ob sich die Strahlen (in Abbildung 14: u, f) treffen. Meistens stehen diese windschief zueinander, da die Pixel nicht auf der gleichen Epipolarlinie liegen und somit sich nicht entsprechen können. Die Vektoren der Strahlen bestehen aus drei 32-Bit-Gleitkommazahlen (float-Werten). Wegen Gleitkomma-Ungenauigkeiten kann es deswegen auch zu keinem Treffer kommen, selbst wenn sich die Strahlen treffen müssten. Aus diesem Grund berechnet dieser Algorithmus einen Verbindungsvektor, der den kürzesten Abstand zwischen den beiden Strahlen beschreibt und überprüft, ob dessen Betrag unter einem bestimmten Wert liegt, der je nach gewünschter Genauigkeit festgelegt werden kann. Wählt man beispielsweise einen sehr großen Überprüfungswert, so erhält man durch die größere Ungenauigkeit auch deutlich mehr mögliche Übereinstimmungen als bei einem kleineren Wert.

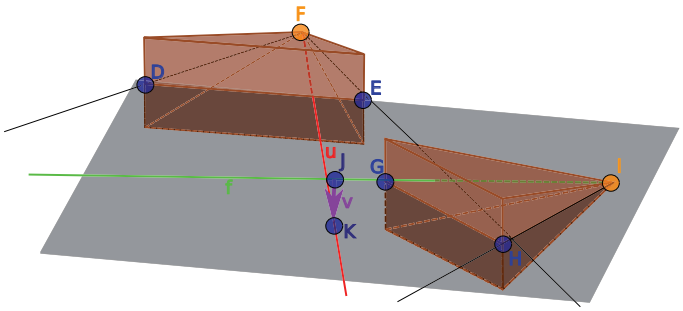

Abbildung 14: Visualisierung der zwei Strahlen und dessen Verbindungsvektor, Quelle: eigene Darstellung

Allgemein lässt sich der Verbindungsvektor mit folgender Formel beschreiben.

$$\vec{v} = (\lambda \cdot \vec{u} + \vec{x}) - (\mu \cdot \vec{f} + \vec{y}) \tag{2}$$

Dabei handelt es sich bei $\vec{u}$ und $\vec{f}$ um die Richtungsvektoren der Strahlen mit deren Länge λ und μ und bei $\vec{x}$ und $\vec{y}$ um die Ortsvektoren der Kameraursprünge.

3.4.3 Länge des Verbindungsvektors

Die Länge des Vektors $\vec{v}$ muss minimal sein, um zu überprüfen, ob der Pixel auf derselben Epipolarlinie liegt. Dafür berechnet man den Betrag von $\vec{v}$ in Abhängigkeit von λ und μ (siehe Formel 3).

$$|\vec{v}| = |(\lambda \cdot \vec{u} + \vec{x}) - (\mu \cdot \vec{f} + \vec{y})| \tag{3}$$

$$|\vec{v}| = \lambda(\vec{u}) + \mu(\vec{f}) - 2\lambda\mu\vec{u} \cdot \vec{f} + (\vec{x}) + (\vec{y}) - 2\vec{x} \cdot \vec{y} + 2(\lambda\vec{u} \cdot \vec{x} - \lambda\vec{u} \cdot \vec{y} - \mu\vec{f} \cdot \vec{x} + \mu\vec{f} \cdot \vec{y})$$

Der Vektor kann genauso als Funktion in Abhängigkeit von λ und μ gesehen werden, die

man ableiten kann. Diese Ableitungsfunktion kann man nun Null setzen, da die Funktion an dieser Stelle ihren Tiefpunkt besitzt (siehe Formel 4 und 5).

$$\partial_\lambda d(\lambda,\mu) = 2(\vec{u}) \cdot \lambda - 2\vec{u} \cdot \vec{f} \cdot \mu + 2(\vec{u} \cdot \vec{x} - \vec{u} \cdot \vec{y}) = 0 \tag{4}$$

$$\partial_\mu d(\lambda,\mu) = 2(\vec{f}) \cdot \mu - 2\vec{u} \cdot \vec{f} \cdot \lambda + 2(\vec{f} \cdot \vec{y} - \vec{f} \cdot \vec{x}) = 0 \tag{5}$$

Der $\cos\varphi$ ergibt sich aus dem Schnittwinkel der beiden Strahlen und vereinfacht die Formel zur Berechnung von λ und μ (siehe Formel 6).

$$\cos\varphi = \frac{\vec{v} \cdot \vec{u}}{|\vec{v}| \cdot |\vec{u}|} \tag{6}$$

Setzt man Formel 4 und 5 gleich und berechnet nun den Winkel φ (siehe Formel 6), ergibt sich daraus Formel 7, mit der man λ und μ ausrechnen kann.

$$\begin{pmatrix}\lambda\\ \mu\end{pmatrix} = \begin{pmatrix} 2\vec{u} - 2\vec{u} \cdot \vec{f} \\ -2\vec{u} \cdot \vec{f} + 2\vec{f}\end{pmatrix}^{-1} \begin{pmatrix} -2(\vec{u} \cdot \vec{x} - \vec{u} \cdot \vec{y}) \\ 2(\vec{f} \cdot \vec{x} - \vec{f} \cdot \vec{y}\end{pmatrix} \tag{7}$$

Die Formel für die Berechnung von λ lautet demnach:

$$\lambda = -\frac{\vec{u} \cdot \vec{x} - \vec{u} \cdot \vec{y}}{|\vec{u}| \cdot \sin\varphi} + \frac{\cos\varphi \cdot (\vec{f} \cdot \vec{x} - \vec{f} \cdot \vec{y})}{|\vec{u}| \cdot |\vec{f}| \cdot \sin\varphi} \tag{8}$$

Analog dazu lautet die Formel zur Berechnung von μ:

$$\mu = \frac{\vec{f} \cdot \vec{x} - \vec{f} \cdot \vec{y}}{|\vec{f}| \cdot \sin\varphi} - \frac{\cos\varphi \cdot (\vec{u} \cdot \vec{x} - \vec{u} \cdot \vec{y})}{|\vec{f}| \cdot |\vec{u}| \cdot \sin\varphi} \tag{9}$$

Mithilfe von λ und μ lässt sich nun die Länge des Verbindungsvektors berechnen. Ist diese kleiner als $0,1$, so wird ein Punkt (Vertex) mit dem Koordinaten der Mitte des Verbindungsvektors und dem RGB-Wert des Pixels des rechten bzw.linken Bildes in einer Liste gespeichert.

3.5 Auswählen des passenden Pixels

Gemäß Kapitel 3.4 werden alle möglichen Vertices in einer Liste gespeichert. Nun muss man aus der Liste den Vertex mit der größten Übereinstimmung finden. Dies erreicht man durch sogennantes Pixel Matching. Hierbei wird der RGB-Wert des Pixels im mittleren Bild mit den RGB-Werten der Pixel, die auf der Epipolarlinie zum Pixel im mittleren Bild liegen, verglichen und der Wahrscheinlichste ausgewählt.

Dafür greift man wie schon in Kapitel 3.3 beschrieben auf den RGB-Wert des Pixels im mittleren Bild zu und übergibt diesen zusammen mit der Liste von möglichen Pixeln der Funktion `findBestVertex()` (siehe Abbildung 15). Diese geht nun mithilfe einer for-

Schleife die Vertices in der Liste durch und vergleicht deren RGB-Werte mit denen des einzelnen Pixels. Immer wenn ein Vertex mit hoher Übereinstimmung gefunden wurde, wird dessen Nummer gespeichert, und, falls kein Pixel mit besserer Übereinstimmung gefunden wurde, am Ende der Funktion zurückgegeben. Dieser Pixel wird anschließend in eine weitere Liste gespeichert, die alle überprüften Vertices der beiden Bilder enthält. Die Liste wird am Schluss des Algorithmus in eine PLY-Datei[10] geschrieben, die dann von dem Grafik-Programm Gnuplot angezeigt werden (siehe Abbildung 16).

4 Der Algorithmus

4.1 Allgemeines zum Algorithmus

Das in Kapitel 3 erläuterte Verfahren wurde in der Programmiersprache C++ geschrieben. C++ ist eine objektorientierte Sprache. Dies erleichtert den Umgang mit den einzelnen Punkten des Objektes und das Ausrechnen ihrer Tiefendaten mithilfe von Vektoren. Außerdem trägt die starke Typisierung in Datentypen und Funktionen zur Lesbarkeit des Algorithmus bei. Ein weiterer Grund, der C++ hierfür prädestiniert, ist die Open Source Bibliothek OpenCV, da sie einen schnellen Zugriff auf einzelne Pixel in den Fotos sowie effiziente Algorithmen zur Weiterverarbeitung ermöglicht (siehe auch sKapitel 3.3). Zusätzlich bietet die Vektorklasse aus „3D-Spieleprogrammierung mit DirectX 9 und C++“[11] hervorragende Möglichkeiten, um die Datenmenge zu verkleinern (siehe Kapitel 3.4.2).

4.2 Die Implementierung

Nachdem ein umsetzbares Verfahren zum Finden der Vertices gefunden wurde, lässt sich das Verfahren vor allem auch durch OpenCV und die Vektorklasse (siehe Kapitel 4) leicht in Quellcode übersetzen.

Die Vektorklasse spielt in dem Algorithmus eine zentrale Rolle. Aus zwei Vektoren wird beispielsweise eine Klasse Strahl (in der Implementierung Ray genannt) gebildet, mit der man mithilfe der Funktion `intersect()` den Abstand der beiden Strahlen (siehe Kapitel 3.4.2) ausrechnen kann. Auch bilden Vektoren die Grundlage der Klasse Camera, in denen die Koordinaten des Kameraursprungs und der Kamerabildebene gespeichert werden (siehe Kapitel 3.4.1). Die Bild-Klasse (Picture) steht separat zu den anderen Klassen, da sie nur zum Bearbeiten der Fotos gedacht ist; so ist die Methode `faceCapture()` für das Entfernen des Hintergrundes im Bild (siehe Kapitel 3.3) zuständig. Erst nach dem Entfernen spielen die anderen Klassen für das Verfahren eine Rolle. In `compare()` werden zum

[10] *PLY (file format).*
[11] Scherfgen 2006.

Schluss die Vertices aus Kapitel 3.5 gefunden. Probleme bei der Implementierung bereitete nur die Speicherplatzverwaltung. Da nicht alle Zeiger auf durch `compare()` erstellte Vertices zuverlässig gelöscht wurden, kam es durch die Masse an gefundenen Vertices und der daraus folgenden Speicherüberlastung oft zu einem Absturz des Programms.

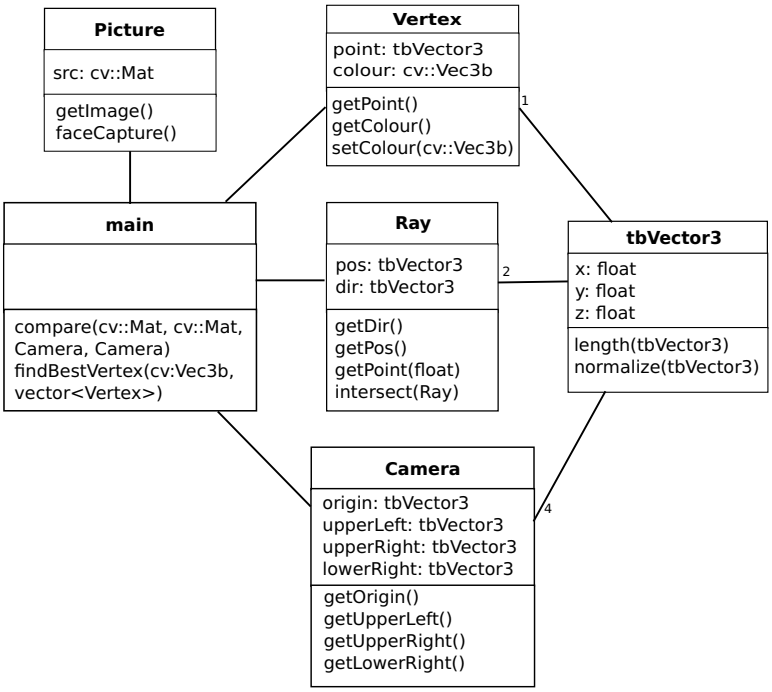

Abbildung 15: vereinfachtes Klassendiagramm des Algorithmus, Quelle: eigene Darstellung

4.3 Ergebnisse des Algorithmus

Das aus dem Algorithmus entstehende Modell sieht man in Abbildung 16. Es erfüllt leider nicht das in Kapitel 1.3 genannte Ziel, allerdings könnte man mit einigen Optimierungen genauere Modelle erstellen. Ursachen und deren Lösungsvorschläge finden sich unter Kapitel 5 und 6.

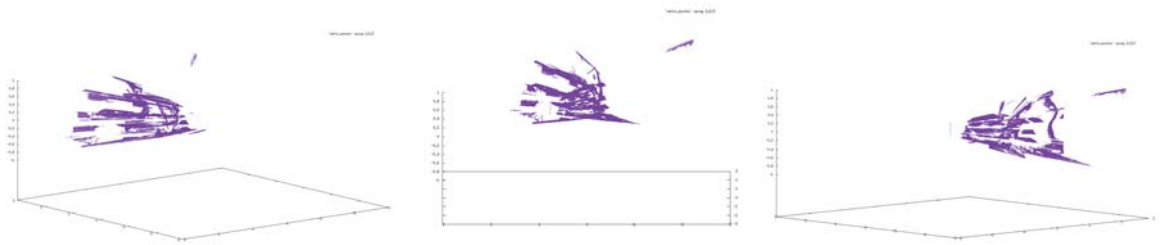

Abbildung 16: Durch den Algorithmus berechnetes Modell des Gesichtes von links, der Front und rechts, Quelle: eigenes Modell

5 Problembehandlung

Der Algorithmus ist in seinen Berechnungen nicht sehr genau. Dies hat verschiedene Ursachen. Zum einen ist es auf das verwendete Equipment für die Fotos zurückzuführen, zum anderen auch auf kleinere Rechenfehler und andere Ungenauigkeiten im Algorithmus, deren Optimierung allerdings den Umfang dieser Arbeit übersteigen würden.

5.1 Ungenauigkeiten aufgrund der Fotos

Der Hauptgrund für Ungenauigkeiten im Modell entsteht durch die Fotos. Diese sind zwar in HD-Qualität aufgenommen, allerdings nur von einer einfachen Webcam mit Aufnahmen bis zu maximal drei Megapixeln, die die Farben dadurch nicht so genau abbilden wie zum Beispiel eine Spiegel-Reflexkamera.

Abbildung 17: Vergleich der benutzen Webcam (rechts) mit einer Spiegel-Reflexkamera (links), Quelle: eigene Fotos

Auch der Aufnahmeort der Bilder spielt für das Pixel Matching eine Rolle. Zwar wurde ein Teil der Fotos in einem Greenscreen Studio aufgenommen, allerdings konnte dabei nicht garantiert werden, dass das Objekt von allen Seiten gleich ausgeleuchtet wird, sodass es zu kleinen Farbunterschieden kommen konnte.

Auch war es schwierig sicherzustellen, dass die Kameras genau den gleichen Abstand zwischen einander oder auch die exakt gleiche Ausrichtung haben. So passen die Bilder nicht komplett zusammen, was ein Pixel Matching und das finden der Epipolarlinien zusätzlich erschwert. Diese Fehlerquelle hätte man eliminieren können, indem man für die Kameras Ausrichtungen und Abstände mithilfe eines festen Gestells fixiert hätte.

5.2 Ungenauigkeiten aufgrund des Algorithmus

Zusätzlich zu der Problemquelle „Foto“ gibt es in den Berechnungen mithilfe des Algorithmus Ungenauigkeiten. Teilweise resultiert dies aus Kapitel 5.1. Da selbst übereinstimmende Pixel durch abweichenden Lichteinfall oder veränderte Kameraausrichtung voneinander abweichen können, kann nicht nach einem Pixel mit demselben Farbwert gesucht werden,

sondern immer nur nach einem Pixel mit ähnlichem Farbwert, was die Wahrscheinlichkeit eines falschen Matchings erhöht.

Wie schon in Kapitel 3.4.2 erwähnt, arbeitet die Vektorklasse außerdem mit 32-Bit float-Werten und nicht mit 64-Bit double-Werten, welche zwar genauer wären, allerdings auch mehr Speicherplatz benötigen. Deshalb kann es zu Rechenfehlern kommen, die ein Gleichsetzen der beiden Strahlen und anschließendes Auflösen unmöglich machten. So wurde in einer früheren Version des Algorithmus versucht, λ und μ anstatt mit Formel 8 und 9 mithilfe der Formel 10 auszurechnen, wobei durch Rundungsfehler allerdings nie eine komplette Übereinstimmung gefunden wurde.

$$\vec{x} + \lambda \cdot \vec{u} = \vec{y} + \mu \cdot \vec{f} \tag{10}$$

Ein weiteres Problem ergab sich aus der ursprünglich geplanten Verwendung des Freeware-Grafikprogramms Blender, das ursprünglich für diese Seminararbeit verwendet werden sollte. Die PLY-Datei konnte zwar angeblich in Blender importiert werden, allerdings lagen alle gefunden Punkte auf einer Geraden, die schräg im Raum lag. Dabei bildeten die gefundenen Punkte immer dieselbe Gerade, und zwar egal, welche der Bilder in den Algorithmus geladen wurden. Lediglich die Lücken lagen an unterschiedlichen Stellen. Dies lässt sich auf einen Fehler im PLY-Plugin zurückführen. Aus diesem Grund wurde auf das etwas ältere Programm Gnuplot umgestiegen.

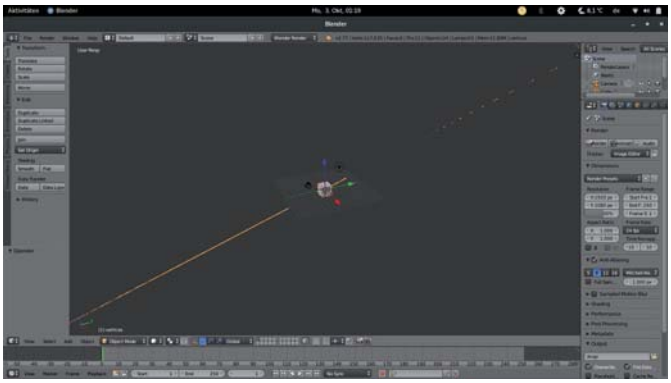

Abbildung 18: Angezeigte Punkte des Würfels in Blender, Quelle: eigene Darstellung

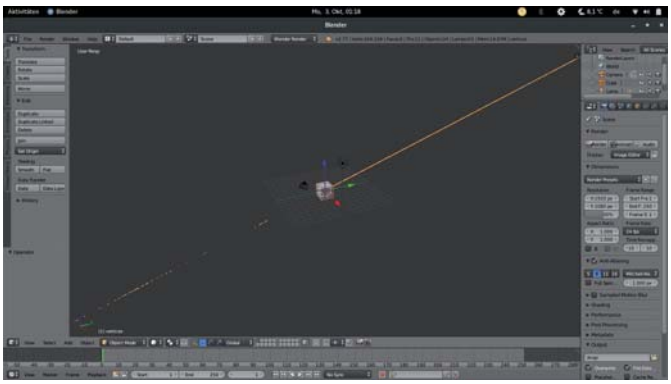

Abbildung 19: Angezeigte Punkte des Gesichtes in Blender, Quelle: eigene Darstellung

6 Ausblick

Wie schon in Kapitel 1.3 erwähnt, könnte dieser Algorithmus mit einigen Erweiterungen und Verbesserungen, wie z. B. besseren Aufnahmen mit einer Spiegelreflex-Kamera anstelle der hier genutzten Web-Kameras, der Verwendung von 64-Bit double-Werten statt 32-Bit float-Werten bei den Vektorrechnungen oder dem Einsatz von mehr als drei Kameras, genauere oder komplette Modelle erstellen, was allerdings auf Kosten der Laufzeit vonstatten gehen würde. Bereits bei drei HD-Bildern benötigte der verwendete Computer über zwei Tage Rechenlaufzeit, um das Modell zu erstellen, um das Modell zu erstellen. Man könnte zusätzlich den Algorithmus auf kurze Videosequenzen erweitern, indem man die Sequenz in einzelne Fotos aufteilt und diese dann einzeln von dem Algorithmus berechnen lässt. Auch ließe sich mit dem gegeben Algorithmus ein farbiges Modell mit den gespeicherten Farbwerten der Vertices erstellen. Da sich in Gnuplot allerdings keine Farben anzeigen lassen, wurde in dieser Arbeit darauf verzichtet. Auch gibt es Möglichkeiten, die Farberkennung zu verbessern. Beispielsweise beschreibt die Methode des Colour Mappings[12] eine Technik, wie Farben eines Fotos auf die beiden anderen Fotos übertragen werden könnten.

[12] Kagarlitsky 2009.

Anhang

A Quellenverzeichnis

Primärquellen

Scherfgen, David (2006): *3D-Spiele-Programmierung mit DirectX 9 und C++*. München, Wien: Hanser.

Monographien & Fachartikel

Beeler, Thabo u. a. (2010): *High-Quality Single-Shot Capture of Facial Geometry: Implemention Details*. Zürich: Department of Computer Science.

Kagarlitsky, Sefy (2009): „Consitent Color Mappings of Images Acquired Under Various Conditions“. Diss. Efi Arazi School of Computer Science.

Neuss, Nicolas (2010): *Einführung in die Numerische Mathematik für Studierende der Fachrichtungen Informatik und Ingenieurswesen*. Vorlesungsscript. Karlsruher Institut für Technologie.

Internetquellen

Bayer, Thilo (2015): *Star Citizen: Squadron 42 - Hinter den Kulissen mit Mark Hamill*. URL: `http://www.pcgameshardware.de/vidimg/2015/11/66434.jpg` (besucht am 06. 10. 2016).

Block-matching algorithm. URL: `https://en.wikipedia.org/wiki/Block-matching_algorithm` (besucht am 06. 10. 2016).

Epipolargeometrie. URL: `https://de.wikipedia.org/wiki/Epipolargeometrie` (besucht am 06. 10. 2016).

OpenCV. URL: `http://opencv.org/` (besucht am 06. 10. 2016).

PLY (file format). URL: `https://en.wikipedia.org/wiki/PLY_(file_format)` (besucht am 06. 10. 2016).

Steinlechner, Peter (2015): *Erste Szenen aus der Kampagne von Star Citizen*. URL: `http://video.golem.de/teaser/1/1/16101/medium-480-s42fat-snap.jpg` (besucht am 06. 10. 2016).

B Abbildungsverzeichnis

C Quellcode

main

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include "Picture.h"
#include "Ray.h"
#include "Camera.h"
#include "Vertex.h"
#include "tbVector3.h"

using namespace cv;
using namespace std;

Vertex* findBestVertex(Vec3b first, std::vector<Vertex> innerList) {
  int save = 0;
  Vec3b second, difference = {(uchar)20, (uchar)20, (uchar)20};
  for(int i = 0; i < innerList.size(); i++) {
    second = innerList.at(i).getColour();
    if((first[0] <= second[0] + difference[0] && first[1] <= second[1] - difference[1] &&
      first[2] <= second[2] + difference[2]) || (first[0] >= second[0] - difference[0] &&
      first[1] >= second[1] + difference[1] && first[2] >= second[2] - difference[2])) {
      save = i;
      difference = first - second;
    }
  }
  return new Vertex(innerList.at(save).getPoint(), first);
}

vector<Vertex*>* compare(Mat one, Mat two, Camera mid, Camera outer)
{
  vector<Vertex*>* list = new vector<Vertex*>();
  Ray midRay, outerRay;
  Vec3b first, second, white = {(uchar)255, (uchar)255, (uchar)255};
  //Verbindungsvektoren zuwischen den Ecken der Kamera
  tbVector3 x = mid.getUpperRight() - mid.getUpperLeft();
  float pixelLength = tbVector3Length(x) / one.size().width;
  x = tbVector3Normalize(x);

  tbVector3 y = mid.getLowerRight() - mid.getUpperRight();
  float pixelHeight = tbVector3Length(y) / one.size().height;
  y = tbVector3Normalize(y);

  tbVector3 xOut = outer.getUpperRight() - outer.getUpperLeft();
  xOut = tbVector3Normalize(xOut);

  tbVector3 yOut = outer.getLowerRight() - outer.getUpperRight();
  yOut = tbVector3Normalize(yOut);

  for(int i = 0; i < one.size().width; i++) {
    for(int j = 0; j < one.size().height; j++) {
```

```
      vector<Vertex> innerList = vector<Vertex>();
      //überspringt jeden Pixel, der nicht zu dem Objekt gehört durch faceCapture-Methode
      first = one.at<Vec3b>(j, i); //Farbe des Pixels bei x=i, y=j
      if(first == white) {
        continue;
      }

      //Vektor auf die Stelle des i,j-ten Pixel
      tbVector3 v = (mid.getUpperLeft() + x * (i * pixelLength + pixelLength/2)
      + y * (j * pixelHeight + pixelHeight/2)) - mid.getOrigin();

      //Strahl mit Kamera als Ursprung und v als Richtung
      midRay = Ray(mid.getOrigin(), v);

      for(int k = 0; k < two.size().width; k++) {
        for(int l = 0; l < two.size().height; l++) {
          second = two.at<Vec3b>(l, k); //Farbe des Pixels bei x = k, y = l
          if(second == white) {
            continue;
          }
          //Vektor auf die Stelle des k,l-ten Pixel
          tbVector3 w = outer.getUpperLeft() + xOut * (k * pixelLength + pixelLength/2)
          + yOut * (l * pixelHeight + pixelHeight/2) - outer.getOrigin();

          //Strahl mit Kamera als Ursprung und v als Richtung
          outerRay = Ray(outer.getOrigin(), w);

          //Vektor der Schnittpunkte der beiden Strahlen (falls vorhanden)
          tbVector3* intersection = midRay.intersect(outerRay);

          if(intersection == nullptr) {
            delete intersection;
            continue;
          }

          //erstellt einen Vertex an der Stelle des Schnittpunktes mit dessen RGB-Wert
          Vertex point = Vertex(*(intersection), second);
          delete intersection;

          innerList.push_back(point);
          }
        }

        if(innerList.size() > 0) {
          list->push_back(findBestVertex(first, innerList));
        }
        cout << "Größe List: " << list->size() << endl;
      }
    }

  return list;
}

int main(int argc, char** argv)
{
  Camera right = Camera(tbVector3(0, -1, 0), tbVector3(0.15, -0.45, 0.12),
  tbVector3(0.55, -0.85, 0.12), tbVector3(0.55, -0.85, -0.12));
  Camera mid = Camera(tbVector3(-1, 0, 0), tbVector3(-0.51, 0.28, 0.12),
```

```
tbVector3(-0.51, -0.28, 0.12), tbVector3(-0.51, -0.28, -0.12));
Camera left = Camera(tbVector3(0, 1, 0), tbVector3(0.55, 0.85, 0.12),
tbVector3(0.15, 0.45, 0.12), tbVector3(0.15, 0.45, -0.12));

char* imgPath = nullptr;
if(argc < 2)
{
  imgPath = (char*) "mitte.jpg";
}
else
{
  imgPath = argv[1];
}
Picture midPic(imgPath);
imgPath = (char*) "links.jpg";
Picture leftPic(imgPath);
imgPath = (char*) "rechts.jpg";
Picture rightPic(imgPath);

midPic.faceCapture();
leftPic.faceCapture();
rightPic.faceCapture();

midPic.decrease(2);
leftPic.decrease(2);
rightPic.decrease(2);

vector<Vertex*>* list = compare(midPic.getImage(), leftPic.getImage(), mid, left);
vector<Vertex*>* list2 = compare(midPic.getImage(), rightPic.getImage(), mid, right);

for(int i = 0; i < list2->size(); i++) {
  list->push_back(list2->at(i));
}
delete list2;

for(int i = 0; i < list->size(); i++) {
  Vertex* v = list->at(i);
  cout << v->getPoint().x << " " << v->getPoint().y << " " << v->getPoint().z << endl;
}

ofstream vertices("vertices.ply");
vertices << "ply\n";
vertices << "format ascii 1.0\n";
vertices << "element vertex " << list->size() << "\n";
vertices << "property float x\n property float y\n property float z\n";
vertices << "end_header\n";

for(int i = 0; i < list->size(); i++) {
  vertices << list->at(i)->getPoint().x << " " <<
  list->at(i)->getPoint().y << " " << list->at(i)->getPoint().z << endl;

}

vertices.close();

for(int i = 0; i < list->size(); i++)
{
  delete list->at(i);
```

```
  }
  delete list;

}
```

Picture

```
#ifndef PICTURE_H
#define PICTURE_H

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <opencv2/core/mat.hpp>
#include <vector>
#include <opencv2/photo.hpp>

class Picture
{
public:

  void PictureIn(cv::Mat);
  Picture(char *);
  ~Picture();

  cv::Mat getImage();

  //delete (white space) everthing else than the face
  void faceCapture();

private:
  cv::Mat src;

};

#endif

#include "Picture.h"

using namespace cv;
using namespace std;

//load the picture of imgPath in src
Picture::Picture(char * imgPath) {
  src = imread(imgPath);
  if (src.empty()) {
    printf("Could not load image file!");
    exit(1);
  }
}

void Picture::PictureIn(Mat in) {
  src = in;
}

Picture::~Picture() {
  src.release();
```

```
}

Mat Picture::getImage() {
  return src;
}

void Picture::faceCapture() {
  vector<Mat> channels;
  Mat save = src.clone();
  cv::split(src, channels);

  //delete the greenscreen in R, G, B plane
  cv::threshold(channels[1], channels[1], 115, 255, THRESH_BINARY);
  cv::threshold(channels[2], channels[2], 145, 255, THRESH_BINARY_INV);
  cv::threshold(channels[0], channels[0], 90, 255, THRESH_BINARY);

  //use it as a mask for the source image to delete the whole screen in the BGR-image
  Mat mask;
  merge(channels, mask);

  Vec3b intensity;

  Vec3b white = { (uchar)255, (uchar)255, (uchar)255};

  src.at<Vec3b>(0, 0) = white;


  for (int x = 1; x < src.size().width; x++) {
    for(int y = 0; y <  src.size().height; y++) {
      intensity = mask.at<Vec3b>(y, x);

      if(intensity == white) {
        src.at<Vec3b>(y, x) = white;
      }
    }
  }
  imwrite("beforeInterpolation.jpg", src);

  //make a grayscale mask for inpaint() to repair the lost parts of the face
  vector<Mat> saveChannels;
  cv::split(save, saveChannels);

  cv::threshold(saveChannels[2], saveChannels[2], 135, 255, THRESH_BINARY_INV);
  cv::threshold(channels[2], channels[2], 100, 255, THRESH_BINARY_INV);
  mask = saveChannels[2] + channels[2];
  cv::threshold(mask, mask, 100, 255, THRESH_BINARY_INV);

  inpaint(src, mask, src, 15, INPAINT_TELEA);
  imwrite("interpolation.jpg", src);
  imwrite("mask2.jpg", mask);

}
```

Vertex

```
#ifndef VERTEX_H
#define VERTEX_H
```

```
#include <opencv2/core/core.hpp>
#include "tbVector3.h"


class Vertex {

public:

  Vertex();
  Vertex(tbVector3, cv::Vec3b);

  tbVector3 getPoint();
  cv::Vec3b getColour();
  void setColour(cv::Vec3b);

private:

  tbVector3 pos;
  cv::Vec3b colour;

};

#endif
```

```
#include "Vertex.h"

Vertex::Vertex() {
  pos = tbVector3();
  colour = {(uchar)0, (uchar)0, (uchar)0};
}

Vertex::Vertex(tbVector3 v, cv::Vec3b colour) {
  pos = v;
  this->colour = colour;
}

tbVector3 Vertex::getPoint() {
  return pos;
}

cv::Vec3b Vertex::getColour() {
  return colour;
}

void Vertex::setColour(cv::Vec3b colour) {
    this->colour = colour;
}

//arithmetische Operatoren für cv::Vec3b
inline cv::Vec3b operator - (const cv::Vec3b& a, const cv::Vec3b& b) {return cv::Vec3b(a
    [0]-b[0],a[1]-b[1],a[2]-b[2]);}
inline cv::Vec3b operator + (const cv::Vec3b& a, const cv::Vec3b& b) {return cv::Vec3b(a
    [0]+b[0],a[1]+b[1],a[2]+b[2]);}
```

Ray

```
#ifndef RAY_H
#define RAY_H
```

```
#include "tbVector3.h"

class Ray {

public:
  Ray();
  Ray(tbVector3, tbVector3);

  tbVector3 getPos();
  tbVector3 getPoint(float);
  tbVector3 getDir();

  tbVector3* intersect(Ray);


private:
  tbVector3 pos;
  tbVector3 dir;
};

#endif
```

```
#include "Ray.h"
#include <iostream>

Ray::Ray() {
  pos = tbVector3();
  dir = tbVector3();
}

Ray::Ray(tbVector3 pos, tbVector3 dir) {
  this->pos = pos;
  this->dir = tbVector3Normalize(dir);
}

tbVector3 Ray::getPos() {
  return pos;
}

tbVector3 Ray::getDir() {
  return dir;
}

tbVector3 Ray::getPoint(float f) {
    return dir * f + pos;
}

tbVector3* Ray::intersect(Ray r) {
  float cosPhi = tbVector3Dot(dir, r.getDir()) / (tbVector3Length(pos) * tbVector3Length(
      r.getPos()));
  float sqSinPhi = 1 - pow(cosPhi, 2);
  float v = -(tbVector3Dot(dir, pos) - tbVector3Dot(dir, r.getPos())) / (pow(
      tbVector3Length(dir), 2) * sqSinPhi)
        + cosPhi * (tbVector3Dot(r.getDir(), pos) - tbVector3Dot(r.getDir(), r.getPos()))
        / (tbVector3Length(pos) * tbVector3Length(r.getPos()) * sqSinPhi);
  float u = (tbVector3Dot(r.getDir(), pos) - tbVector3Dot(r.getDir(), r.getPos())) / (pow
```

```
      (tbVector3Length(r.getDir()), 2) * sqSinPhi)
        - cosPhi * (tbVector3Dot(dir, pos) - tbVector3Dot(dir, r.getPos()))
        / (tbVector3Length(pos) * tbVector3Length(r.getPos()) * sqSinPhi);
  tbVector3 connection = r.getPoint(u) - getPoint(v);
  if(tbVector3Length(connection) < 0.1) {
    tbVector3 tmp = getPoint(v) + connection / 2;
    return new tbVector3(tmp);
  }
  return nullptr;
}
```

Camera

```
#ifndef CAMERA_H
#define CAMERA_H

#include "tbVector3.h"

class Camera {

public:
  Camera(tbVector3, tbVector3, tbVector3, tbVector3);

  tbVector3& getOrigin();
  tbVector3& getUpperLeft();
  tbVector3& getUpperRight();
  tbVector3& getLowerRight();


private:
  tbVector3 origin;
  tbVector3 upperLeft;
  tbVector3 upperRight;
  tbVector3 lowerRight;

};

#endif

#include "Camera.h"

Camera::Camera(tbVector3 origin, tbVector3 upL, tbVector3 upR, tbVector3 lowR) {
  this->origin = origin;
  upperLeft = upL;
  upperRight = upR;
  lowerRight = lowR;
}

tbVector3& Camera::getUpperRight() {
  return upperRight;
}
tbVector3& Camera::getUpperLeft() {
  return upperLeft;
}
tbVector3& Camera::getLowerRight() {
  return lowerRight;
}
tbVector3& Camera::getOrigin() {
```

```
  return origin;
}
```

tbVector3

```
#ifndef TBVECTOR3_H
#define TBVECTOR3_H
#include <cmath>
// Die 3D-Vektor-Klasse
class tbVector3
{
public:
  // Variablen
  union
  {
    struct
    {
      float x;  // Koordinaten
      float y;
      float z;
    };

    struct
    {
      float u;  // Texturkoordinaten
      float v;
      float w;
    };

    float   c[3];   // Array der Koordinaten
    //D3DVECTOR vD3DVector; // Vektor in Form eines Direct3D-Vektors
  };

  // Konstruktoren
  tbVector3()                                                    {}
  tbVector3(const tbVector3& v) : x(v.x), y(v.y), z(v.z)                    {}
  tbVector3(const float f) : x(f), y(f), z(f)                          {}
  tbVector3(const float _x, const float _y, const float _z) : x(_x), y(_y), z(_z)
      {}
  tbVector3(const float* pfComponent) : x(pfComponent[0]), y(pfComponent[1]), z(
      pfComponent[2]) {}
  //tbVector3(const D3DVECTOR& v) : vD3DVector(v)                    {}

  // Casting-Operatoren
  operator float* ()          {return (float*)(c);}
  //operator D3DVECTOR& ()        {return vD3DVector;}
  //operator const D3DVECTOR& () const  {return vD3DVector;}

  // Zuweisungsoperatoren
  tbVector3& operator = (const tbVector3& v)  {x = v.x; y = v.y; z = v.z; return *this;}
  tbVector3& operator += (const tbVector3& v) {x += v.x; y += v.y; z += v.z; return *this
      ;}
  tbVector3& operator -= (const tbVector3& v) {x -= v.x; y -= v.y; z -= v.z; return *this
      ;}
  tbVector3& operator *= (const tbVector3& v) {x *= v.x; y *= v.y; z *= v.z; return *this
      ;}
  tbVector3& operator *= (const float f)    {x *= f; y *= f; z *= f; return *this;}
```

```
  tbVector3& operator /= (const tbVector3& v) {x /= v.x; y /= v.y; z /= v.z; return *this
      ;}
  tbVector3& operator /= (const float f)    {x /= f; y /= f; z /= f; return *this;}
};

// Arithmetische Operatoren
inline tbVector3 operator + (const tbVector3& a, const tbVector3& b)  {return tbVector3(a
    .x + b.x, a.y + b.y, a.z + b.z);}
inline tbVector3 operator - (const tbVector3& a, const tbVector3& b)  {return tbVector3(a
    .x - b.x, a.y - b.y, a.z - b.z);}
inline tbVector3 operator - (const tbVector3& v)           {return tbVector3(-v.x, -v.y,
    -v.z);}
inline tbVector3 operator * (const tbVector3& a, const tbVector3& b)  {return tbVector3(a
    .x * b.x, a.y * b.y, a.z * b.z);}
inline tbVector3 operator * (const tbVector3& v, const float f)     {return tbVector3(v.x
    * f, v.y * f, v.z * f);}
inline tbVector3 operator * (const float f, const tbVector3& v)     {return tbVector3(v.x
    * f, v.y * f, v.z * f);}
inline tbVector3 operator / (const tbVector3& a, const tbVector3& b)  {return tbVector3(a
    .x / b.x, a.y / b.y, a.z / b.z);}
inline tbVector3 operator / (const tbVector3& v, const float f)     {return tbVector3(v.x
    / f, v.y / f, v.z / f);}

// Vergleichsoperatoren
inline bool operator == (const tbVector3& a, const tbVector3& b) {if(a.x != b.x) return
    false; if(a.y != b.y) return false; return a.z == b.z;}
inline bool operator != (const tbVector3& a, const tbVector3& b) {if(a.x != b.x) return
    true; if(a.y != b.y) return true; return a.z != b.z;}

// Funktionen deklarieren
inline float    tbVector3Length(const tbVector3& v)                          {return sqrtf
    (v.x * v.x + v.y * v.y + v.z * v.z);}
inline float    tbVector3LengthSq(const tbVector3& v)                        {return v.x *
    v.x + v.y * v.y + v.z * v.z;}
inline tbVector3  tbVector3Normalize(const tbVector3& v)                        {return v
    / sqrtf(v.x * v.x + v.y * v.y + v.z * v.z);}
inline tbVector3  tbVector3NormalizeEx(const tbVector3& v)                      {return v
    / (sqrtf(v.x * v.x + v.y * v.y + v.z * v.z) + 0.0001f);}
inline tbVector3  tbVector3Cross(const tbVector3& v1, const tbVector3& v2)           {
    return tbVector3(v1.y * v2.z - v1.z * v2.y, v1.z * v2.x - v1.x * v2.z, v1.x * v2.y -
    v1.y * v2.x);}
inline float    tbVector3Dot(const tbVector3& v1, const tbVector3& v2)           {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;}
inline float    tbVector3Angle(const tbVector3& v1, const tbVector3& v2)         {
    return acosf((v1.x * v2.x + v1.y * v2.y + v1.z * v2.z) / sqrtf((v1.x * v1.x + v1.y *
    v1.y + v1.z * v1.z) * (v2.x * v2.x + v2.y * v2.y + v2.z * v2.z)));}
inline tbVector3  tbVector3InterpolateCoords(const tbVector3& v1, const tbVector3& v2,
    const float p) {return v1 + p * (v2 - v1);}
inline tbVector3  tbVector3InterpolateNormal(const tbVector3& v1, const tbVector3& v2,
    const float p) {return tbVector3NormalizeEx(v1 + p * (v2 - v1));}

inline bool tbVector3EqualRounded(const tbVector3& a, const tbVector3& b) {
  if(a.x > b.x + 10 || a.x < b.x - 10)  return false;
  if(a.y > b.y + 10 || a.y < b.y - 10) return false;
  if(a.z > b.z + 10 || a.z < b.z - 10) return false; return true;
}

#endif
```

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

…....., den …....
Ort Datum

…....
Unterschrift des Verfassers