UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Online-Panoramaerstellung für VR

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Katharina Krämer

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	MSc. Bastian Krayer (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐

Harschbach, 30.09.2019 K. Krämer

(Ort, Datum) (Unterschrift)

Zusammenfassung

Innerhalb dieser Arbeit wird die Theorie des Video-Seethroughs anhand einer Panoramaerstellung aus mehreren Kamerabildern verschiedener Perspektiven grundlegend dargestellt. Darauf basierend wurde ein System konzipiert und umgesetzt, bei dem Videostreams durch perspektivische Verzerrung zu einem Panoramabild zusammengesetzt werden. Anschließend wird dieses auf die Innenseite eines Zylinders projiziert, in dessen Mitte sich die virtuelle Position des Betrachters befindet. Schließlich sollen die entstandenen Videopanoramen in einer VR-Brille dargestellt werden. Innerhalb der Implementierung werden außerdem einige Optimierungen vorgestellt, unter anderem solche, die das System - über die Aufgabenstellung hinaus - echtzeitfähig machen. Des Weiteren wird das erarbeitete System bewertet und mit zwei anderen Verfahren verglichen.

Abstract

In this thesis, the theory of video seethrough is fundamentally presented on the basis of a panoramic view from several camera frames of different perspectives. Based on this, a system was designed and implemented in which video streams are put together into a panoramic image by perspective distortion. This is then projected onto the inside of a cylinder with the virtual position of the viewer in the middle. Finally, the resulting video panoramas will be displayed in VR glasses. Within the implementation some optimizations are also presented, among others those that make the system real-time capable beyond the task. Furthermore, the developed system will be evaluated and compared with two other methods.

Inhaltsverzeichnis

1 Einleitung

Anwendungen der erweiterten Realität (engl.: *augmented reality*) und der virtuellen Realität (engl.: *virtual reality*) werden heutzutage immer häufiger in moderne visuelle Systeme integriert. Beliebte Beispiele sind Rückfahrassistenzen für Fahrzeuge, interaktive Gebrauchsanweisungen für komplizierte Gerätschaften, oder auch Computerspiele. Durch den Einsatz von Kamerasystemen können 360°-Videos oder VR-Touren durch real existierende Gebäude aufgenommen werden, wobei der Betrachter das Gefühl hat, er würde sich in gewisser Weise tatsächlich an diesem Ort befinden. Verstärkt wird dieses Gefühl von der Interaktivität mit der virtuellen Umgebung, da er sich frei darin umsehen kann.
Die Technik der Online-Panoramen kann dazu genutzt werden, schwer zugängliche oder schwer einsehbare Zonen zu visualisieren und die Wahrnehmung des Menschen dahingehend zu erweitern. Besonders eindrucksvoll wird ein solches sogenanntes Video-Seethrough durch den Einsatz von Head-Mounted Displays (HMDs), welche den Betrachter noch stärker in die dargestellte Szene eintauchen lässt. Zumal Menschen 80 Prozent der Umwelteindrücke über die Augen erhalten. Ein Viertel des Gehirns beschäftigt sich daraufhin mit der Analyse dieser visuellen Eindrücke [SPDT16]. Umso wichtiger ist der Einsatz solcher Video-Seethroughs in Situationen, in welchen das Visuelle ohne Kamerasysteme schwer wahrnehmbar ist, wie zum Beispiel bei Panzerfahrzeugen oder auch Drohnen.

Es soll in diesem Kontext ein Online-Panorama entstehen, welches so schnell, wie möglich auf Veränderungen in der Welt reagiert. Das bedeutet, dass der Betrachter durch die Kameraaufnahmen eine Panoramaansicht erhält, die kontinuierlich neu berechnet wird und sich den neuen Begebenheiten durch die Berechnung immer neuer Parameterwerte anpassen kann, um die Kamerabilder korrekt zu einem Panorama verschmelzen zu lassen. Mit einer anschließenden Darstellung auf einer Virtual Reality-Brille wird ein immersives Video-Seethrough erzeugt.

2 Grundlagen

2.1 Video-Seethrough

Video-Seethrough ist eine Technik, bei der Kameras eingesetzt werden, welche die Umgebungsinformationen meist als Videostream aufnehmen und dem Betrachter angezeigt werden. Dieser nimmt die Welt also nicht direkt wahr, sondern indirekt durch die Kameraaufnahmen. Dieses Prinzip wird beispielsweise bei digitalen Videokameras verwendet, wobei die Aufnahme nicht nur durch das Okular, sondern auch auf einem Display betrachtet werden kann. Solche Bildinformationen können aber auch genutzt werden, um durch Hinzufügen virtueller Objekte eine erweiterte

Realität zu erzeugen oder durch das Aufeinanderlegen und Verschmelzen mehrerer Kamerabilder eine Panoramasicht zu ermöglichen. Man unterscheidet zwischen kopfverbundenem und handverbundenem Video-Seethrough [Mü19]. Dabei handelt es sich um die verschiedenen Darstellungsweisen der Bilder, entweder als Head Mounted Display oder als Handheld, wie Handys, Tablets oder Videokameras.

2.2 Monoskopie und Stereoskopie

Bei der monoskopischen Darstellung einer Szene betrachtet man gewöhnliche zweidimensionale Abbildungen ohne Tiefeneindruck. Selbst, wenn mit der Fluchtpunkttechnik gezeichnet wird, kann der Mensch die räumliche Anordnung nur erahnen, ohne tatsächlich einen Raum in der Bildfläche auszumachen. Unsere Augen nehmen jeweils ein um ca. 10 cm verschobenes Bild wahr [Bor94]. Diese Bilder werden zwar nur minimal, aber dennoch von einer jeweils anderen Position aufgenommen und sind dadurch nicht nur durch einen Offset der Position, sondern auch perspektivisch verschieden. Die Verschmelzung dieser beiden Bilder im Gehirn verhilft dazu Schlüsse über Anordnung von Gegenständen im dreidimensionalen Raum zu erzielen, der uns in der physischen Wirklichkeit umgibt. Mit verschiedenen stereoskopischen Techniken kann dieser Eindruck auch durch zweidimensionale Abbildungen erzielt werden. Damit entsteht für den Betrachter ein Raumeindruck, obwohl diese Raumtiefe rein physikalisch gar nicht existiert. Beispielsweise wird dazu ein Bildschirm für jeweils ein Auge verwendet. Dieser sollte möglichst groß gehalten werden, sodass eine gute Immersion durch Abdeckung des Gesichtsfeldes erreicht wird [Bor94]. Sobald jedoch der Abstand der Bildschirmmittelpunkte zu groß wird (d.h. größer als der Augenabstand), tritt ein Divergenz-Problem auf (siehe Abb. 1). Die Augen können nicht adäquat divergieren, wie es die Bildschirmmittelpunkte vorgeben. Eine Lösung wäre das Kippen der Bildschirme, was aber Unschärfen verursacht. Daher ist die Methode des sogenannten Image Shiftings beliebter, bei welcher die aktiven Zentren beider Bilder auf dem Bildschirm verschoben werden [Bor94] (siehe Abb. 2). Diese Methode wird beispielsweise auch von *Google VR* [car] verwendet, das in dieser Arbeit als VR-Darstellungssystem zum Einsatz kommt.

Wie bereits beschrieben benötigt man für einen echten räumlichen Eindruck jedoch nicht nur eine horizontale Verschiebung der Bilderpaare für das reche und linke Auge. Viel mehr müssen diese sogenannten Stereopaare oder stereoskopischen Halbbilder als perspektivisch verschobene Ansichten präsentiert werden, wie zum Beispiel in Abbildung 3.

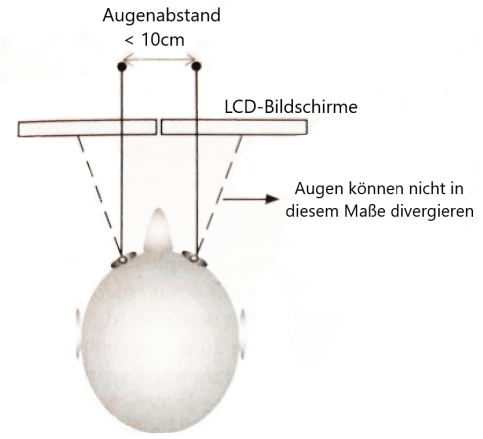


Abbildung 1: Normaler Augenabstand und das Problem der Divergenz [Bor94]

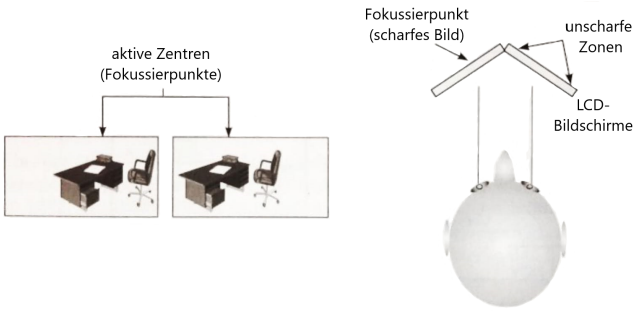


Abbildung 2: Verfahren zur Behebung des Divergenz-Problems. Links: Image Shifting; rechts: Kippen der Bildschirme

Abbildung 3: stereoskopische Darstellung eines Polarfuchses aus einer Google VR Expedition [car]

In Abbildung 3 ist die Ansicht auf einen Polarfuchs für das rechte und linke Auge dargestellt. Es lässt sich erkennen, dass das rechte Auge die linke Flanke des Fuchses besser einsehen kann als das linke Auge. Die zweidimensionale Projektion des Objekts ist dadurch breiter (*rechts*: 5.5 LE, *links*: 6.0 LE). Diese Ansichten imitieren die Sinneseindrücke, die das Augenpaar hätte, wenn man sich tatsächlich an dem gerenderten oder gefilmten Ort befinden würde.

In dieser Arbeit wird zum Rendern jedoch lediglich eine einfache zweidimensionale Textur verwendet, die nur eine Perspektive darstellen kann. Durch Image Shifting kann zumindest ein kleiner dreidimensionaler Eindruck mit einer VR-Brille erzeugt werden. Der Schwerpunkt dieser Arbeit liegt auf einem möglichst effizienten Zusammenfügen einzelner Kamerabilder zu einem Panoramabild. Das Erstellen und Verwenden korrekter Stereopaare kann Gegenstand weiterführender Aufgabenstellungen oder Erweiterungen sein (siehe Kapitel 6).

2.3 Kamerakalibrierung

Nach der Funktionsweise einer Lochkamera bilden alle Kameras Objekte des dreidimensionalen Raumes auf eine zweidimensionale Fläche ab [Pau16]. Jedoch erfolgt diese Abbildung nicht ohne Fehler, da die Bauteile der Kamera, wie Linse oder Lage des Bildsensors minimale Abweichungen des Idealzustands aufweisen. Mit einer Kalibrierung ist es auch möglich die stark verzerrten Bilder von Weitwinkelobjektiven zur Weiterverarbeitung zu verwenden, wenn diese durch die Kalibrierung vorher entzerrt werden.

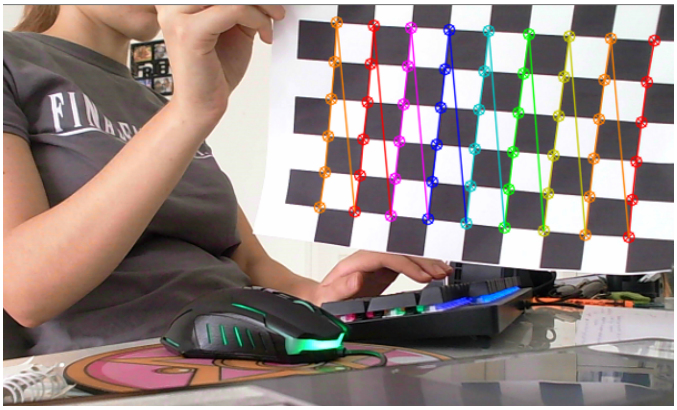


Abbildung 4: farbliche Darstellung der inneren Knotenpunkte eines Schachbretts; detektiert durch die OpenCV-Funktion *findChessboardCorners()* [ope]

Um eine Kamera zu kalibrieren, benötigt man ein zweidimensionales

Kalibrierungsmuster (z.B. ein Schachbrettmuster). Dabei wird die reale Größe der Kacheln auf dem Papier mit der Größe der Kacheln auf den Testaufnahmen in einen mathematischen Zusammenhang gebracht und die intrinsischen Kameraparameter werden bestimmt. Diese beschreiben den Zusammenhang zwischen dem Kamera- und dem Bildkoordinatensystem (wie z.B. Brennweite und optisches Zentrum). Zur Kalibrierung werden zwei Arten der Distorsion untersucht.

Die radiale Verzeichnung von Kamerabildern ist meist eine Tonnenverzerrung (auch bekannt unter dem *Fischaugeneffekt*). Ihre Korrektur lässt sich durch folgende Formel beschreiben [ope]:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{1}$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2}$$

Der Pixelwert an der Position (x, y) im verzerrten Eingabebild wird nach $(x_{corrected}, y_{corrected})$ im Ergebnisbild transformiert.

Eine tangentiale Verzerrung tritt auf, wenn das Objektiv nicht genau parallel zur Bildsensorebene steht. Diese Verzerrung kann wie folgt korrigiert werden [ope]:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{3}$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{4}$$

Die Parameter k_1, k_2, p_1, p_2 und k_3 aus den Formeln (1) - (4) sind die Distorsionskoeffizienten als Zeilenmatrix:

$$Distortion_{coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3) \tag{5}$$

Mit den noch unbekannten intrinsischen Parametern lässt sich folgende Kameramatrix aufstellen, die nach dem Lochkameraprinzip Punkte aus dem $\mathbb{R}^3$-Raum in den $\mathbb{R}^2$-Raum projiziert [ope]. Es sei dazu angemerkt, dass der Weltkoordinaten-Ursprung in der Kamera liegt (siehe Abb. 5) :

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{6}$$

Dabei ist (X, Y, Z) der Punkt aus dem $\mathbb{R}^3$-Raum, der nach $\mathbb{R}^2$ projiziert wird. In der zweidimensionalesn Ebene hat er dann die Koordinaten (x, y, w). In der Matrix ist c das optische Zentrum in Pixelkoordinaten und f die Brennweite. w ist die homogene Koordinate, mit der x und y nach der Projektion in den $\mathbb{R}^2$-Raum normalisiert werden. Diese Normalisierung mit der w -Koordinate des projizierten 3D-Punktes ist notwendig, weil so alle Punkte auf einer Projektionsgeraden auf ein und dem selben 2D-Punkt

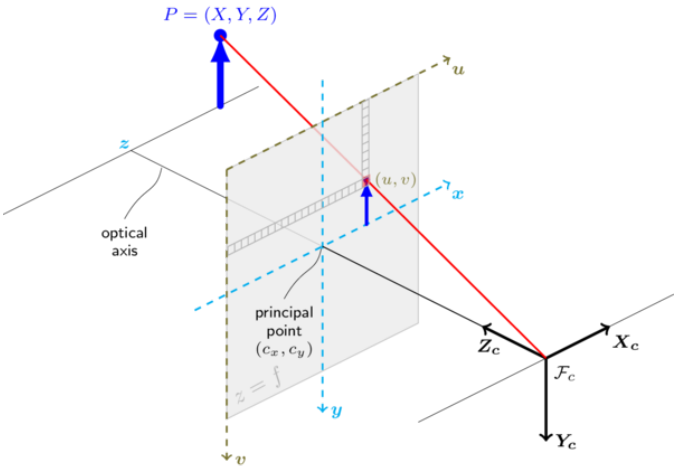


Abbildung 5: Das Lochkameraprinzip in geometrischer Darstellung [ope]

projiziert werden.
Die Kalibrierung umfasst das Berechnen der unbekannten Parameter in (5) und (6). Diese Berechnung besteht aus dem Lösen von Gleichungssystemen, wobei pro aufgenommenem Testbild mit dem Kalibrierungsmuster eine Gleichung hinzu kommt. Je mehr Bilder aus verschiedenen Positionen und mit verschiedenen Rotationen aufgenommen werden, desto genauer wird die Kalibrierung schlussendlich.

2.4 Homographie

Um ein Panorama aus verschiedenen Kameraquellen zu entwickeln, muss die Bildebene (u, v-Ebene) der jeweils rechten und linken Kamera auf die Ebene der mittleren Bildebene abgebildet werden. Das erfolgreiche Verschmelzen verschiedener Kamerabilder mit überlappenden Bildausschnitten setzt daher voraus, dass die geometrischen Beziehungen zwischen den Kameraanordnungen berücksichtigt werden [Sch12].

Es wird also angenommen, dass Punkt $X_1 = (u_1, v_1, 1)$ in einem Bild und Punkt $X_2 = (u_2, v_2, 1)$ in einem anderen Bild den gleichen Punkt in der realen Szene auf verschiedene Projektionsebenen abbilden (siehe Abb. 6). Mit einer Matrix H, auch Homographie genannt, werden die korrespondierenden Pixelkoordinaten folgendermaßen in Verbindung gebracht $X_2 = H * X_1$, wobei :

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{7}$$

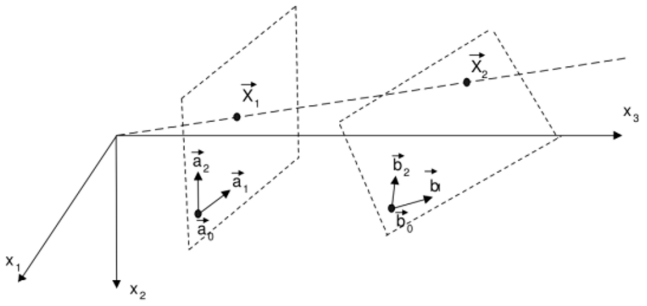


Abbildung 6: Ebenen im $\mathbb{R}^3$ sollen aufeinander abgebildet werden durch eine Transformationsmatrix H [Sch12]

Wenn diese Homographie auf alle Pixel angewendet werden, erhält man eine perspektivisch verzerrte Version des originalen Kamerabildes [ope].
Eine Voraussetzung dafür ist, dass die Aufnahmen von der gleichen Szene stammen, um die Homographie aus Punktkorrespondenzen zu errechnen, oder (falls bekannt) aus einem Winkel, durch den sich die Aufnahmen unterscheiden. Solange diese Bedingung erfüllt ist, können sich die Projektionsebenen der Kameras in Position und Neigungswinkel unterscheiden. Falls sich die Aufnahmen nur durch eine Rotation unterscheiden und der Winkel bekannt ist, besteht die Homographie aus der Rotationsmatrix R, wobei gilt $X_2 = R * X_1$.
Das Bildverarbeitungssystem, was in dieser Arbeit vorgestellt wird, soll jedoch dynamisch in Rotation und Translation sein, solange sich die Bildbereiche noch ausreichend überlappen. Ausreichend bedeutet in diesem Fall, dass genug Bildinformationen im Überlappungsbereich ist, um eine korrekte Homographie berechnen zu können (praktische Umsetzung und Untersuchung in Kapitel 4.2.2) . Eine solche Homographie soll schnellstmöglich immer wieder neu und richtig berechnet werden, sodass das Panorama dynamisch ist und sich den veränderten Bedingungen anpassen kann. Doch stellt sich dann die Frage, wie Bildkorrespondenzen gefunden werden, wenn Rotation und Translation der Bildebenen zu keinem Zeitpunkt sicher bekannt sind. Hier muss mit einem merkmalsbasierten Imagestitching gearbeitet werden, wie zum Beispiel mit dem SIFT bzw. SURF Algorithmus. Aus den gefundenen Bildkorrespondenzen kann direkt eine Homographie, beispielsweise mit einer Singulärwertzerlegung, berechnet werden [MV07].

2.5 SIFT/SURF Algorithmus

Manche Merkmale, wie Kanten oder auch Ecken können simpler detektiert werden, als charakteristische und markante Regionen in Bildern, welche nicht so einfach zu interpretieren sind. Der SIFT-Algorithmus (engl. *Scale*

Invariant Feature Transformation) dient der Detektion dieser markanten Keypoints. Diese sollen erkannt und mit einem SIFT-Merkmal beschrieben werden, damit sie wiederum mit Keypoints in anderen Bildern vergleichbar sind. So können Korrespondenzen zwischen den Kameraaufnahmen gefunden werden und darauf basierend Homographien berechnet werden. Ohne eine entsprechende Eingrenzung ordnet der SIFT Algorithmus einem 1000 x 2000 Pixel Bild normalerweise zwischen 10.000 und 20.000 SIFT-Merkmale zu [Pri15]. Diese Merkmale kennzeichnen Extremwerte des Laplacian of Gaussian, der mit verschiedenen σ Werten auf das Bild angewendet wird. Er fungiert als Blobdetektor in einen sogenannten Skalenraum eingebettet, weswegen er bis zu einem gewissen Grad invariant gegen Skalierung, Orientierung, Kameraposition und Beleuchtung ist. Er ist daher besonders geeignet, um gemeinsame Merkmale aus Bildern von verschiedenen Kamerapositionen, Skalierungen und unterschiedlicher Helligkeitsanpassung zu finden [Low04]. Der SURF Algorithmus (*engl. Speeded UP Roboust Features*) liefert die gleichen Ergebnisse ca. doppelt so schnell [BTVG06]. Im sogenannten Skalenraum (engl. *scale space*) kann ein Bild scheinbar aus verschiedenen Entfernungen betrachtet werden. Bei einer höheren Skala wird das Bild von einem weiter entfernten Standpunkt betrachtet und der Informationsgehalt sinkt mit der Auflösung. Bei der sogenannten Scale-space Extrema Detection wird eine spezielle Filterung verwendet, bei der Merkmale verschiedener Größen, mit Hilfe verschieden großer Fenster, detektiert werden. Diese Detektion wird für die verschiedenen Skalierungsstufen (auch *Oktaven*) des Ausgangsbildes ausgeführt, wobei jede höhere Oktave eine halb so große Auflösung hat, wie die jeweils darunter liegende (siehe Abb. 7).

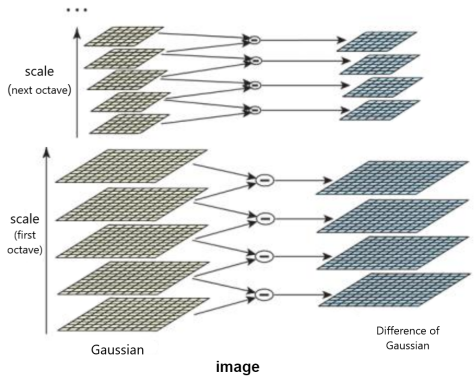


Abbildung 7: Mehrfache Anwendung des DoG Filters in jeder Oktave [ope]

Innerhalb einer Oktave wird das Ausgangsbild mit verschieden großen Gauß-Kernen gefaltet. Zwischen diesen Ergebnisbildern wird zwecks des DoG (engl. *Difference of Gaussian*) die Differenz gebildet. Auf diesen

Ergebnisbildern wird nach lokalen Maxima gesucht, indem jeder Pixel mit seinen acht nächsten Nachbarn und den jeweils neun Nachbarn der nächst oberern und nächst unteren Ebene der Oktave verglichen wird (siehe Abb. 8). Wenn dieser Pixelwert ein lokales Extremum ist, dann ist er ein potentieller Keypoint [ope].

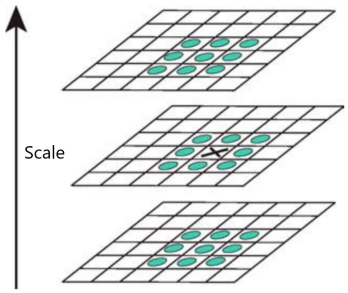


Abbildung 8: Suche von lokalen Extrema: Vergleich eines Pixels mit seinen 26 Nachbarn [ope]

Danach erhält man eine Ergebnisliste mit den Einträgen (x, y, σ). An der Stelle (x, y) mit dem Skalierungswert σ ist somit ein potentieller Keypoint. Im Algorithmus selber wird aus Effizienzgründen statt LoG (engl. *Laplace of Gaussian*) der DoG verwendet. Potentielle Keypoints werden auf ihren Kontrast und auf Kanteneigenschaften untersucht. Punkte mit zu geringem Kontrast werden verworfen, ebenso wie Punkte, die auf einer Kante liegen. Denn Kantenpunkte haben in ihren Eigenschaften wenig Individualität und könnten so leicht mit anderen Punkten verwechselt werden. Zur Kantendetektion wird ein Verfahren ähnlich des Harris Eckendetektors verwendet. Dazu wird eine Eigenwertdekomposition der folgenden Matrix durchgeführt [HS88]:

$$\begin{pmatrix} \sum(I_x(x,y))^2 & \sum I_x(x,y)I_y(x,y) \\ \sum I_x(x,y)I_y(x,y) & \sum(I_y(x,y) \end{pmatrix} \tag{8}$$

Als Ergebnis dieser Eigenwertdekomposition erhält man zwei Eigenwerte. Falls beide einen Schwellwert überschreiten, dann befindet sich im untersuchten Feld eine Ecke. Falls nur ein Wert größer ist, handelt es sich um eine Kante und der Keypoint wird verworfen. Um die Keypoints weiter zu stabilisieren, wird die Hauptorientierung mit den umliegenden Helligkeitsgradienten erstellt [Fri10]. Schließlich werden die Merkmalsvektoren erzeugt. Jedes Merkmal kennzeichnet sich durch die Bildposition (x, y), seiner Hauptorientierung und einem 128-dimensionalen Merkmalsvektor (auch *Deskriptor*). Daraufhin werden die Merkmale verschiedener Bilder auf Basis des euklidischen Abstandes, oder durch einen Suchbaum

verglichen und Korrespondenzen werden gefunden [ope].

3 Systemdesign

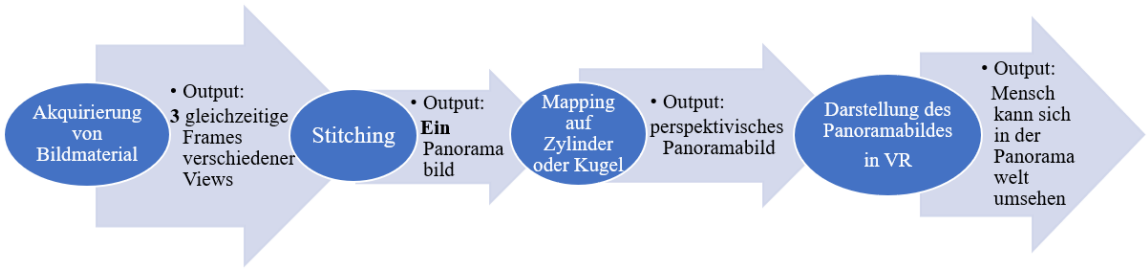


Abbildung 9: Prozess der Verarbeitung der Videokamerabilddaten von der Aufnahme bis zur Präsentation des Panoramabildes als Flussdiagramm

Wie in Abbildung 9 dargestellt, kann der Bildverarbeitungsprozess als Pipeline dargestellt werden, beginnend mit der Akquirierung der Kameraframes. Durch ein merkmalsbasiertes Verfahren sollen diese Kamerabilder zusammengesetzt, also korrekt übereinander gelagert werden. Der Output ist das fertige Panoramabild, welches in einem nächsten Schritt auf einen Zylinder oder eine Kugel gemappt werden soll, sodass der Betrachter das Gefühl hat sich tatsächlich in dem Raum zu befinden. Dieser Effekt wird durch das Verwenden einer VR-Brille unterstützt.

3.1 Kamerakonzeption und Videoaufnahme

Die Kamerakonfiguration besteht aus drei unabhängigen Webcams, welche einen manuellen Fokus und jeweils einen Öffnungswinkel von ca. $75°$ haben. Abbilding 10 zeigt die Berechnung des Öffnungswinkels.

Da dieser Öffnungswinkel bereits größer ist, als der eines Normalobjektivs, ist eine Kamerakalibrierung als Verzeichnungskorrektur sinnvoll. Ein weiteres Merkmal dieses Kamerakonzepts besteht darin, dass jede der 3 Kameras eine eigene Position hat und damit auch die Welt aus ihrer eigenen einzigartigen Perspektive aufnimmt (siehe Abb. 11). Die Konzeptidee besteht darin, die Frames der Kameras K_l und K_r so zu verzerren, dass sie sich mit der Perspektive der Frames von K_m zusammenfügen lassen. So liegt die eigentliche Betrachterposition hinter der von K_m. Dies lässt sich jedoch mit mehr als einer Kamera nicht realistisch nachbauen, da jede Kamera stets eine eigene Perspektive hat. Stattdessen werden die Überlappungsbereiche der Kameras nach korrespondierenden Merkmalen untersucht. Beispielhaft beträgt der Überschneidungswinkel zweier Kamerabilder $30°$, um ein Panorama mit einem Öffnungswinkel von $165°$ zu erzeu-

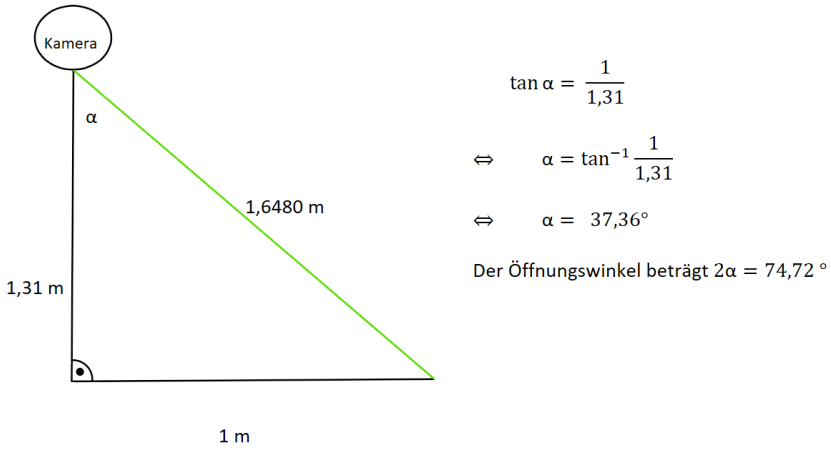


Abbildung 10: Berechnung des Öffnungswinkels α

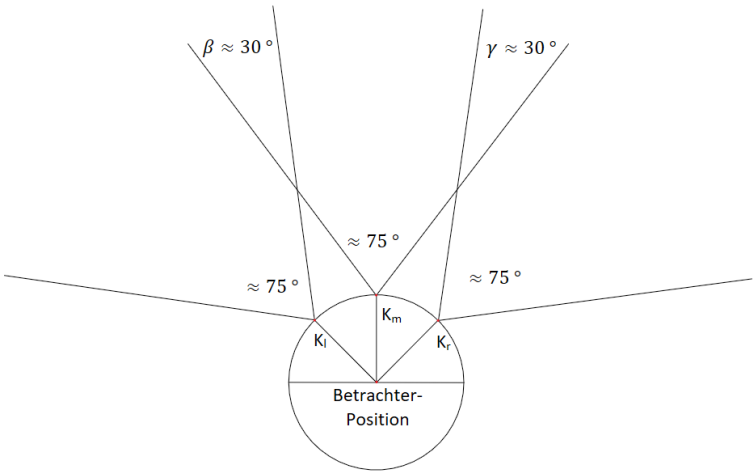


Abbildung 11: Kamerasetup ausgehend von der realen Betrachterposition

gen. Da der SIFT-Algorithmus ein merkmalsbasiertes Verfahren ist, sollte der Überschneidungsbereich der Kamerabilder nicht zu klein sein, um sinnvolle Merkmale finden zu können. Der optimale Winkel hängt jedoch auch davon ab, wie viele Merkmale und Details in der Umgebung der Kameras sichtbar sind. Sind viele Details vorhanden, kann das Sichtfeld dadurch maximiert werden, dass die Winkel β und γ minimiert werden. Das SIFT-Verfahren ermöglicht es, dass diese beiden Winkel keineswegs gleich oder ähnlich sein müssen. Bis zu einem gewissen Grad ist die Kamerakonstellation in Translation, Rotation und Skalierung variabel, solange die Kameras nicht grundlegend ihren Platz mit einer der anderen Kameras tauschen. Die linke Kamera muss stets links neben der mittleren Kamera sein und analog gilt dies für die rechte Kamera auf der rechten Seite. Ist

dies nicht der Fall, so kann zwischen den Bildern kein sinnvolles Matching stattfinden.
Die Videoaufnahme erfolgt durch die Programmierbibliothek *OpenCV*, welche auch Algorithmen für Bildverarbeitung und maschinelles Sehen zur Verfügung stellt. Die Frames, die mit OpenCV ausgelesen werden, können daraufhin analysiert und weiterverarbeitet werden.

Abbildung 12: *Loetad* Webcam

3.2 VR-Anbindung

Zur Darstellung des Panoramas in VR wird die fertige Panoramatextur auf die Innenseite eines Zylinders projiziert, damit bei Betrachtung ein räumlicher Eindruck entsteht. Die Kameraposition in der virtuellen Szene ist dabei in der Mitte des Zylinders. Von dort aus kann sich der Betrachter durch eine VR-Brille im Raum umsehen, der auf der Zylinderinnenwand dargestellt ist. Als Display dient ein Smartphone, das mit einem Gyroskop-Sensor ausgestattet ist, damit die Blickrichtung in Echtzeit verfolgt werden kann. Mit Hilfe einer VR-Brille für Smarphones (siehe Abb. 13) kann der Betrachter in das Geschehen eintauchen. Wie bereits im Grundlagenkapitel 2.2 erwähnt, handelt es sich bei der Darstellung jedoch um eine monoskopische Darstellung.

Abbildung 13: VRBox als Smarphonehalterung, welche mit Linsen für die VR-Ansicht ausgestattest ist

4 Implementierung

4.1 Programmierumgebung und Bibliotheken

Für die Umsetzung dieser Arbeit wurde ein Intel(R) Core(TM) i5-6200U Prozessor mit 2.30 GHz verwendet. Der Installierte Arbeitsspeicher beträgt 7,87 GB. Die Grafikkarte ist eine NVIDIA Geforce 940M. Zur Implementierung gehört zunächst die Wahl von Entwicklungsumgebungen, sowie nützlichen Bibliotheken, welche im Folgenden beschrieben werden.

Blender [ble] Mit Blender ist es möglich 3D-Modelle zu erstellen. In dieser Arbeit wurde in Blender die Panoramaplane aus einem extrudierten Kreis erstellt. Es ist möglich die Modelle in viele verschiedene Dateiformate zu exportieren (z.B. *.blend, .fbx, .obj*).

Unity [uni19] Als Gameengine ist Unity neben der Unreal Engine 4 eine mächtige Entwicklungsumgebung, die viele nützliche Funktionen bietet. Sie unterstützt das Einbinden von selbst programmierten Erweiterungen, sogenannten Plugins, welche bei der Optimierung in dieser Arbeit eine Rolle spielen. Das Verwenden von Software Development Kits, wie zum Beispiel das von Google VR, um ein Smartphone als VR Brille verwenden zu können, ist ebenfalls möglich. Auch können 3D-Modelle, die mit Blender erstellt wurden, einfach als FBX -Datei imporitert und verwendet werden.

Visual Studio Community [vis] Visual Studio ist eine Entwicklungsumgebung, um bspw. mit C++ oder C♯ für Webanwendungen, mobile Apps oder auch Windows-Desktop-Anwendungen zu programmieren. Für Unity wird es standardmäßig installiert, um Skripte oder Plugins zu erstellen. Vorteilhaft ist die Syntax Korrektur und das Einbinden von hilfreichen Bibliotheken durch einen Linker und einen Compiler.

OpenCVSharp [shi] OpenCV ist eine Programmierbibliothek und stellt Funktionen für Bildverarbeitung und maschinelles Sehen zur Verfügung. OpenCVSharp ist ein sogenannter *Wrapper*, also eine OpenCV-Schnittstelle für C♯. Nach dem Einfügen der entsprechenden DLLs (engl. *dynamic link library*) in Unity lassen sich in C♯ alle Funktionen nutzen, die auch das native OpenCV in C++ zu bieten hat.

Google VR SDK [car] Google VR ist die Technologie hinter Google Daydream und Google Cardboard, womit man mit dem Smartphone Cardboard-Demos oder 360 Grad Videos in VR betrachten kann. Das

Google VR Software Development Kit unterstützt die Entwicklung von Virtual Reality Anwendungen für das Smartphone in Unity. Es ist möglich verschiedene Smartphones zu verwenden und passend zur VR-Brille eine Einstellung zu wählen. Der Bildschirm wird daraufhin in der VR-Ansicht zwei Bereiche unterteilt, die jeweils die Sicht eines Auges darstellen.

Unity Remote 5 [uni] Mit der Android App Unity Remote 5 dient das Smartphone sowohl als Display des Unity Szenenoutputs aus Sicht der Kamera, als auch als Sensor für die Kopfbewegungen des Benutzers. Es findet also ein Datentransfer zwischen Computer und Smartphone über ein USB-Kabel in beide Richtungen statt, was das Smartphone als Datenbrille überhaupt erst sinnvoll macht.

4.2 Erstellung und Darstellung des Panoramas

4.2.1 Kameraaufnahmen und Kamerakalibrierung

Der erste Schritt, um ein Panorama zu erstellen, besteht in der Aufnahme der Kamerabilder. Diese werden zunächst entzerrt. Es ist nicht notwendig die drei verschiedenen Kameraquellen zu synchronisieren, da die zeitliche Differenz beim Auslesen der Bilder zur Verarbeitung gering genug ist.

```
public class camtest : MonoBehaviour
{
    // create matrix for frame
    Mat frame1 = new Mat(new Size(640, 480), MatType.CV_8UC3);
    VideoCapture cap1;
    // Start is called before the first frame update
    void Start()
    {
        //assign camera source slot to video capture
        cap1 = new VideoCapture(2);
        if (!cap1.IsOpened()) { print("Video cannot be captured!"); }
    }
    // Update is called once per frame
    void Update()
    {
        //read pixelinformation
        cap1.Read(frame1);
    }
}
```

Abbildung 14: Auslesen und Speichern der Kamerabilder

OpenCVSharp stellt eine Klasse *VideoCapture* zur Verfügung, mit welcher ein beliebiger Kameraeingang meist mit dem Index $0, 1, 2$ oder 3 wählbar ist. Die ausgelesenen Pixelwerte werden in einer Matrixstruktur zwischengespeichert.

Zu Beginn einer Kamerakalibrierung werden die realen Koordinaten durch die tatsächlichen Maße des Kalibrierungsmusters, wie z.B. die Kachelgröße eines Schachbrettmusters, in ein Raster eingetragen. Die Kachelgröße multipliziert mit dem entsprechenden Rasterindex ergibt eine absolute Raumposition in x und y Richtung (siehe Abb. 15).

```
// returns Mat of world space corner points
Mat createknownBoardPosition(Size boardSize, float squareEdgeLength)
{
            Point3f[,] objects = new Point3f[boardSize.Height, boardSize.Width];
            for (int i = 0; i < boardSize.Height; i++)
            {
                for (int j = 0; j < boardSize.Width; j++)
                {
                    objects[i, j] = new Point3f
                    {
                        X = j * squareEdgeLength,
                        Y = i * squareEdgeLength,
                        Z = 0.0f

                    };
                }
            }

    MatOfPoint3f objectPoints = new MatOfPoint3f(boardSize.Width * boardSize.Height, 1, objects);
    return objectPoints;
}
```

Abbildung 15: Berechnung der Schachbrettkoordinaten in der realen Welt in der 2D-Ebene

```
// returns List of all corners in image space
List<Mat> getChessboardCorners(List<Mat> images, bool showResults = false)
{
    List<Mat> allFoundCorners = new List<Mat>();
    foreach (Mat img in images)
    {
        Mat points = new Mat();
        bool found = Cv2.FindChessboardCorners(img, new Size(9, 6), points,
            ChessboardFlags.AdaptiveThresh | ChessboardFlags.NormalizeImage);
        if (found)
        {
            allFoundCorners.Add(points);
        }
        if (showResults)
        {
            Cv2.DrawChessboardCorners(img, new Size(9, 6), points, found);
            Cv2.ImShow("Looking for corners", img);
        }
    }
    return allFoundCorners;
}
```

Abbildung 16: *OpenCV*-Methode zum Finden der Eckpunkte des Schachbretts; Code auf Grundlage von [Lec]

```
Cv2.CalibrateCamera(worldSpaceCornerPoints, checkerboardImageSpacePoints,
    boardSize, cameraMatrix, distortionCoeffs, out rVec, out tVec);
```

Abbildung 17: *OpenCV*-Methode zur Kamerakalibrierung

Des Weiteren werden in den Bildaufnahmen die Eckpunkte des Schachbretts gesucht und die jeweiligen Punkte werden gespeichert (siehe Abb.16). Je mehr dieser Aufnahmen gemacht und analysiert werden, desto genauer wird die Kalibrierung.
Schlussendlich wird aufgrund der erhaltenen Koordinaten die OpenCVSharp-Methode *CalibrateCamera()* (Abb. 17) ausgeführt, in welcher das Gleichungssystem gelöst wird, um die benötigten Distorsionskoeffizienten, sowie die intrinsischen Parameter zu erhalten (siehe Grundlagenkapitel 2.3).

4.2.2 Stitching

Innerhalb des Stitchingverfahrens werden in einem nächsten Schritt die Keypoints der drei Eingangsbilder mit dem SURF-Algorithmus identifiziert. Sobald die Deskriptoren der SURF-Merkmale erstellt wurden, können diese mit Hilfe eines Descriptor-Matchers zwischen den Bildern verglichen werden (siehe Abb. 18). Vorzugsweise geschieht dies mit der k-nearest-neighbour-Methode, welche die k-besten Übereinstimmungen zwischen den Keypoints findet.

```
//detection of keypoints in left and middle img
surf.DetectAndCompute(middleImg, null, out keypoints1, descriptors1);
surf.DetectAndCompute(leftImg, null, out keypoints3, descriptors3);
//matching of equal keypoints
matches_left = matcher.KnnMatch(descriptors1, descriptors3, 2, null, false);
```

Abbildung 18: Anwendung des SURF-Algorithmus

Die Keypoints und deren Matches lassen sich visuell darstellen. Abbildung 19 zeigt die Entwicklung und Optimierung der Matches.

In Abbildung 19 a) sind die ungefilterten Matches sichtbar, wie sie aus der *KnnMatch*-Methode resultieren. Offensichtlich sind viele Matches falsch zugeordnet und sind daher ungeeignet, um eine korrekte Transformationsmatrix zu finden. In einem späteren Schritt wird der RANSAC-Algorithmus verwendet, der Ausreißer Samples aus der Menge der Matches entfernen. Dennoch ist es sinnvoll, falls möglich, im Vorhinein schon solche falschen Matches zu beseitigen.

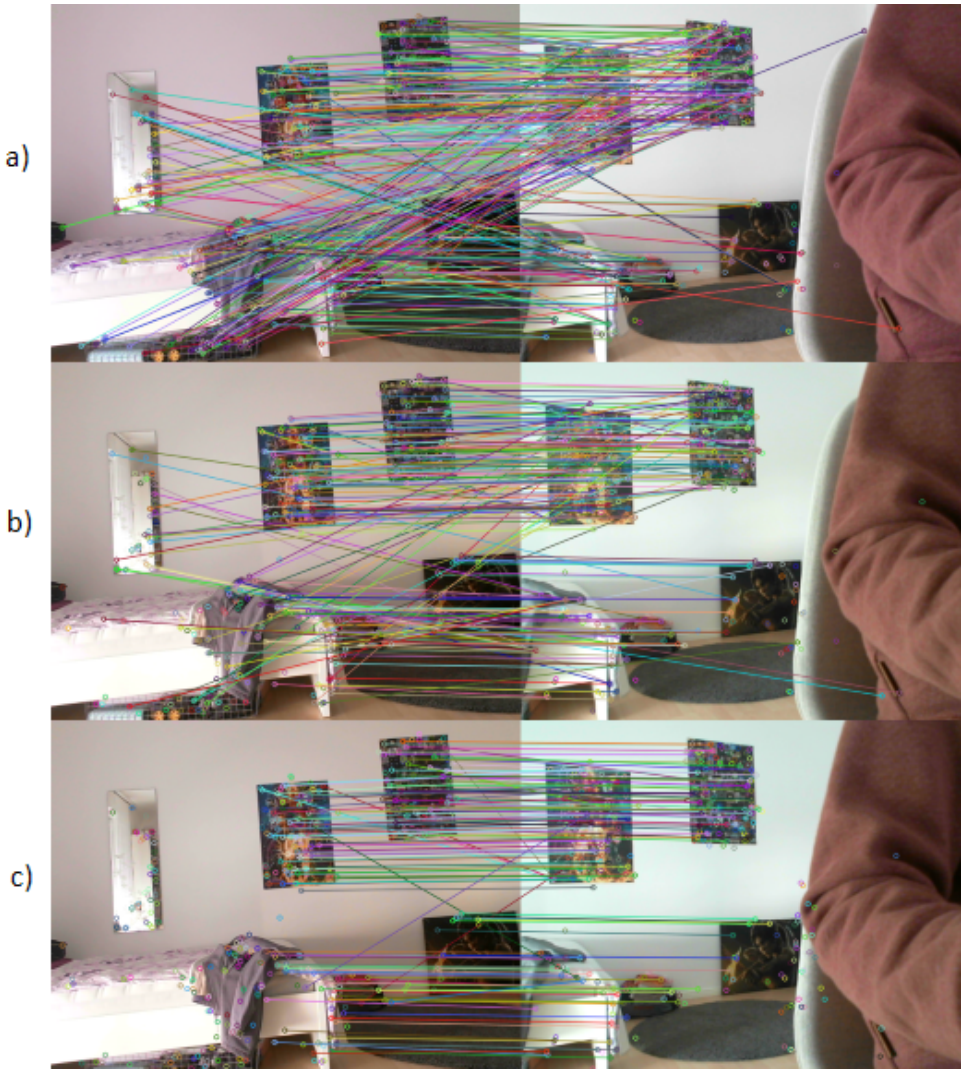


Abbildung 19: Darstellung der Keypoints und ihrer Korrespondenzen: a) ungefilterte Matches; b) gefilterte Matches mit kleinstem Distanz-Wert; c) gefilterte Matches mit kleinstem Distanzwert und realistischem horizontalen Abstand

Abbildung 19 b) zeigt das Ergebnis einer Filterung durch den Vergleich der Merkmalsabstände. Für jeden Keypoint werden die zwei besten Matches gespeichert. Der bessere Match hat einen kleineren Distanzwert und aufgrund der Wahl eines Schwellwerts $thresh_b$, kann dieser Distanzwert als Gültigkeitsbedingung noch weiter verringert werden:

$$matches[i][0].Distance < thresh_b * matches[i][1].Distance \tag{9}$$

In Abbildung 19 c) ist eine äußerst effektive Verbessung zu beobachten, welche jedoch nur funktioniert, wenn die Rotation der Kamera um die y-Achse der Kameras weiter eingeschränkt wird. Denn an dieser Stelle wird überprüft, ob der horizontale Abstand der Matches sinnvoll ist. Ist ein Toleranzbereich vorhersagbar, kann auch hier ein Schwellwert $thresh_c$ gewählt werden, sodass alle Matches entfernt werden, die größer sind als dieser Tolleranzbereich:

$$horizontalMatchDistance <= middleImg.Width * thresh_c \quad (10)$$

Nun wird die Homographie für die beiden äußeren Kamerabilder berechent. Falls an dieser Stelle nach diesen Filterungen nicht wenigstens vier Matches valide sind, so wird in der Implementierung die Berechnung der Homographie abgebrochen, da es nicht möglich ist, diese mit so wenigen Informationen fortzuführen. Es wird bis zum nächsten Berechnungsaufruf die bisherige Homographie weiter genutzt. Mit den nötigen Informationen kann die OpenCVSharp-Funktion $FindHomography()$ jedoch eine Homographie berechnen (siehe Abb. 20). Mit RANSAC werden solche Ausreißer-Matches eliminiert, die nicht durch die bisherige Filterung entfernt wurden.

```
//add keypoints of valid matches to findhomography()
valid_right.ForEach(delegate (DMatch match)
{
    //we only concider the keypoints of valid matches
    dstp.Add(keypoints1[match.QueryIdx].Pt);
    srcp.Add(keypoints2[match.TrainIdx].Pt);
});
src_con = srcp.ConvertAll(new System.Converter<Point2f, Point2d>(Point2fToPoint2d));
dst_con = dstp.ConvertAll(new System.Converter<Point2f, Point2d>(Point2fToPoint2d));

temp_homography = Cv2.FindHomography(src_con, dst_con, HomographyMethods.Ransac);
```

Abbildung 20: Berechnung der Homographie

Anschließend werden die Bilder durch die jeweilige Homographie perspektivisch verzerrt. Jedoch ist es so, dass sich dabei nicht immer eine räumlich sinnvolle Verzerrung ergibt (siehe Abb. 21). Das kann passieren, wenn nicht genügend valide Matches gefunden wurden. Es gibt daher viele verschiedene Homographien, welche das lineare Gleichungssystem erfüllen, was durch die Korrespondenzen aufgestellt wurde. Des Weiteren kann es vorkommen, dass die homogene Koordinate der neuen 2D-Koordinaten fälschlicherweise negativ wird. Die homogene Koordinate muss jedoch in jedem Fall positiv sein, da sie den Z-Wert des ursprünglichen 3D-Punktes beschreibt (siehe Formel (6)). Wenn dieser Abstand negativ wäre, würde der Punkt hinter der Kamera liegen, was er aber nicht

tut, da er von der Kamera abgebildet wird. Auch nach Transformationen durch die Homographie kann sich dieses Vorzeichen nicht ändern, da die homogene Koordinate dann stets 1 beträgt, um eine Verschiebung entlang der Koordinatenachsen durchführen zu können.

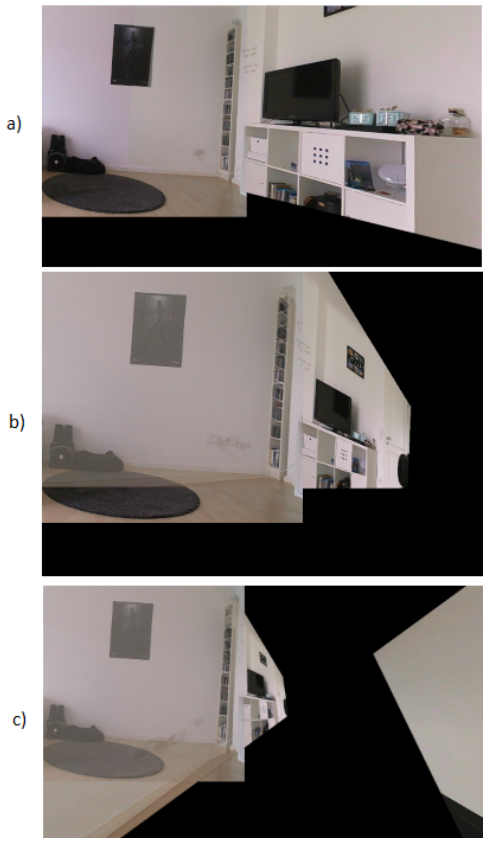


Abbildung 21: Verzerrungen der Perspektive der rechten Bilder: a) valide Homographie; b) invalide Homographie; c) invalide Homographie, die falsche homogene Koordinaten bei neuen Bildpunkten verursacht

Um immer eine richtige Berechnung zu erhalten, muss die resultierende Homographie auf verschiedene Kriterien geprüft werden. Falls die Determinante der Homographie zu nahe an 0 ist, ist diese Matrix sicher falsch, da ihre Vektoreinträge nahezu linear abhängig sind und aus diesem Grund mit einer starken Verzerrung zu rechnen ist. Das Gleiche gilt für einen zu großen Wert, da die Determinate der inversen Homographie dann zu nah an der 0 wäre. Es können also alle Homographien ausgeschlossen werden, deren Determinanten außerhalb von $[1/n, n]$ liegen [VL01]. Im Paper wurde n=10 gewählt. Für einen weiteren Filter für korrekte Homographien werden nur die Eckpunkte der Eingangsbilder mit der Homographie transformiert. Dann wird überprüft, ob die resultiernenden

Punkte in einem Tolleranzbereich liegen. Folgende Überprüfung wird für die Verzerrung des rechten Bildes vorgenommen, wobei P jeweils für einen Eckpunkt des Bildes steht. Angemerkt sei, dass sich der Ursprung in der oberen linken Ecke des mittleren Panoramas befindet. Die transformierten Punkte des rechten Bildes befinden sich relativ dazu:

$$P_{obenLinks}.Y < P_{untenLinks}.Y \tag{11}$$
$$P_{obenRechts}.Y < P_{untenRechts}.Y \tag{12}$$
$$P_{untenLinks}.X < P_{untenRechts}.X \tag{13}$$
$$P_{obenLinks}.X < P_{obenRechts}.X \tag{14}$$

Die Homographie für das linke Kamerabild würde entsprechend spiegelverkehrt überprüft werden. Mit Hilfe der resultierenden Homographie H werden anschließend die kompletten Bildpunkte der Frames transformiert ($P_{neu} = H * P_{alt}$) und in einer eigenen Matrix zwischengespeichert.

4.2.3 Blending

Die verzerrten Teilpanoramen aus dem rechten und dem linken Bild werden mit dem mittleren Kamerabild verblendet. Das Panoramabild, was entstehen soll, besteht aus verschiedenen konstruierten Bildbereichen: aus schwarzen Bereichen, in welchen keine Bildinformation verfügbar ist, aus den Bereichen der linken, rechten und mittleren Bildquelle, sowie den Überlappungsbereichen, in welche Information aus zwei Bildern einfließen und jeweils mit 0.5 gewichtet werden (siehe Abb. 22).

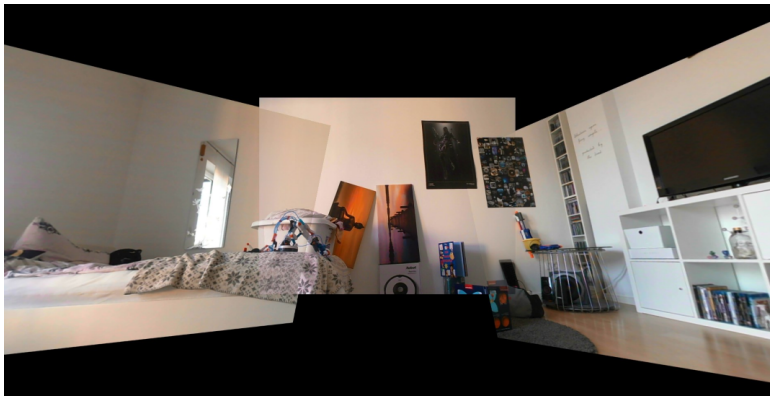


Abbildung 22: Panorama mit dem Blendingfaktor 0.5

Da die Homographien die beien äußeren Bilder in die Perspektive des mittleren Bildes transformieren, ist der obere, linke Eckpunkt des linken Bildes der Ursprung der transformierten Bilder. Wenn man dort

den Nullpunkt annimmt, so wird jedoch ca. $\frac{2}{3}$ des linken Panoramateils abgeschnitten sein. Daher ist es notwenidig neben der Homographie als Transformationsmatrix ebenso eine horizontale und vertikale Translation ($hOffset$ und $vOffset$) als Transformationsmatrix miteinzubeziehen, die alle Panoramabereiche betrifft. Anschließend werden alle Pixel des Ergebnispanoramas betrachtet und überprüft, ob sich an dieser Stelle eines der Panoramapixel befindet. Wenn dies zutrifft, dann wird dieser Pixelwert übernommen, wenn nicht, dann bleibt der Pixel schwarz (siehe Abb. 23). Sind an einer Pixelposition die Pixel aus zwei Inputbildern nicht schwarz, so handelt es sich um einen Überlappungsbereich.

```
if (j < horr_offset && (leftpanidx[i, j].Item0 != 0
    && leftpanidx[i, j].Item1 != 0
    && leftpanidx[i, j].Item2 != 0))
{
    panorama_final.Set<Vec3b>(i, j, leftpanidx[i, j]);
}
```

Abbildung 23: Blending: Entscheidung, ob Pixel aus linkem Frame gesetzt wird

Unglücklicherweise entsteht durch die OpenCVSharp-Methode $warpPerspective()$ um das gewarpte Bild herum ein dunkler Rand, der nicht ganz schwarz ist (d.h. nicht in jedem Farbkanal 0), jedoch so abgedunkelt ist, dass er beim Blending stört. Anscheinend werden die Warpkanten in der Methode etwas geblurt, weswegen der Übergang in die schwarzen Bereiche durch einen solchen Rahmen umschlossen werden. So müssen bei jeder Pixelposition zusätzlich die Nachbarpixelwerte überprüft werden. Falls diese schwarz sind, so wird auch der aktuelle Pixel ignoriert (siehe Abb. 24).

Abbildung 24: störender schwarzer Rand nach *warpPerspective()*

4.2.4 Mapping

Zum Mapping wird zunächst eine Fläche benötigt, auf welche die fertige Panoramatextur projiziert werden kann. Dazu kommt entweder die Innenfläche eines Zylinders oder auch einer Kugel zum Einsatz. In dieser Arbeit wird dazu ein Halbzylinder ohne Boden und Deckel verwendet (siehe Abb. 25). Die aus dem Panorama entstandene Textur wird auf die Innenseite dieses Halbzylinders gemappt, um einen Raum zu simulieren, in dem sich der Betrachter nach Belieben umschauen kann, während er sich in der Mitte der Darstellung befindet. Die horizontalen Linien werden bei der Projektion gekrümmt, während alle vertikalen Linien gerade bleiben.

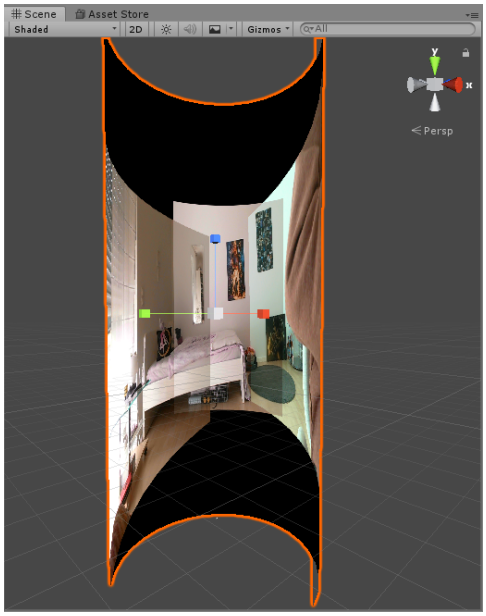


Abbildung 25: Blender-Modell mit Panoramatextur auf der Innenseite in Unity

Dazu müssen die uv-Koordinaten beim Erstellen des Meshs in Blender entweder entsprechend angepasst werden, oder die Anordnung der Vertices muss im Nachhinein entgegengesetzt rotiert angeordnet werden und die Normalen invertiert werden. So kann die Panoramatextur auf die Innenfläche gemappt werden. Mit Hilfe von Google VR und Unity Remote 5 können diese Bildinformationen dann auf eine Datenbrille gebracht werden (siehe Abb. 26).

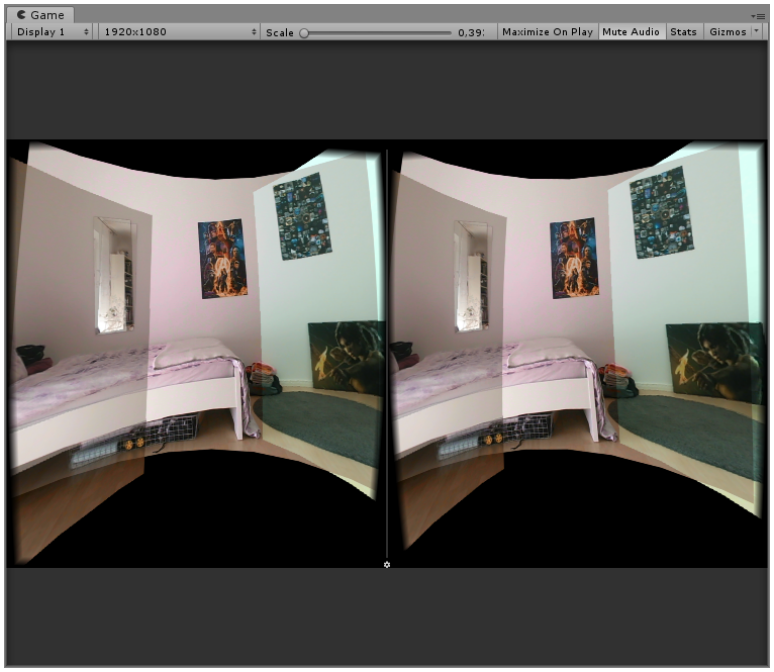


Abbildung 26: Ansicht des Betrachters in VR mit Google VR

4.3 Optimierungen

Im Folgenden wird das vorgestellte Bildverarbeitunsgsystem zur Erstellung von Online-Videopanoramen in den Bereichen optisches Aussehen der Panoramen (Kapitel 4.3.1 und 4.3.2) und Programmlaufzeit (Kapitel 4.3.3, 4.3.4 und 4.3.5) optimiert.

4.3.1 Helligkeitsangleich

Die zu überlagernden Kameraaufnahmen unterscheiden sich nicht nur durch ihre Positionen und Ausrichtungen, sondern auch durch ihre Helligkeitsanpassung abhängig von der einstrahlenden Lichtintensität. Die Kameras passen sich den Lichtverhältnissen so an, dass möglichst wenige Details verloren gehen, die sich im Fokus befinden. Bisher wurden die Eingangsbilder im Überlappungsbereich mit jeweils 0.5 gewichtet. Jedoch hat das einen farblichen Stufeneffekt zufolge, der das Panorama sehr unwirklich wirken lässt (siehe Abb. 26). Daher ist es sinnvoller eine Blendingtechnik zu verwenden, die pro Bildposition einen eigenen Gewichtungsfaktor mit Hilfe von linearer Interpolation und den Strahlensätzen berechnet.
In einem ersten Schritt wird auf die transformierten Eckpunkte des linken und rechten Frame zugegriffen, um die jeweiligen Steigungen der Verbindungsstrecken zu berechnen (siehe Abb. 27 und 28) . Dabei ist noch

einmal zu beachten, dass die transformierten Eckpunktkoordinaten als Ursprung die linke obere Ecke des mittleren Frames haben und nicht die linke obere Ecke des ganzen Panoramabildes, da sich die Homographie auf die Perspektive dieses mittleren Frames bezieht. Von den gewöhnlichen Pixelkoordinaten des Panoramabildes muss daher in den folgenden Formeln immer $vOffset$ und $hOffset$ abgezogen werden.

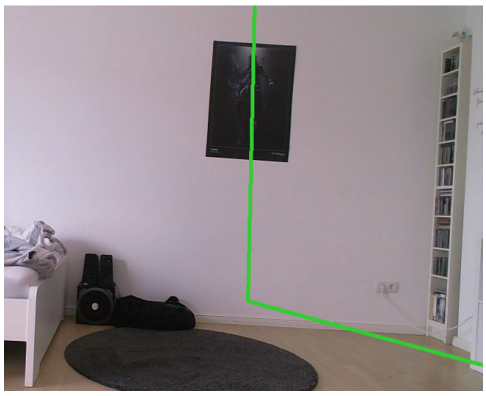


Abbildung 27: Umrisse in grün bilden das Ergebnis der Homographie für das rechte Teilpanorama

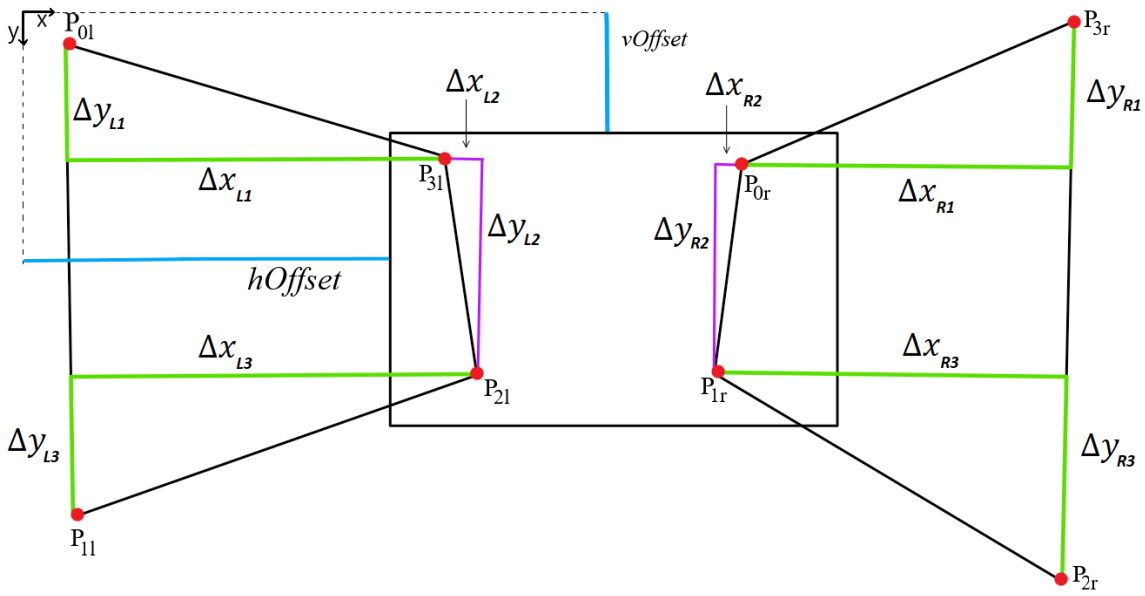


Abbildung 28: Steigungen der Verbindungsstrecken zur Berechnung dynamischer Blendinggewichte

Damit die Pixelwerte im Überschneidungsbereich interpoliert werden können, muss zunächst die Breite dieses Bereichs für jede Zeile berechnet

werden. Für jedes betrachtete Pixel kann durch seine Position für die Interpolation ein prozentualer Anteil zwischen 0 und 1 berechnet werden. Dazu müssen zunächst die Werte Δx und Δy ermittelt werden (siehe Abb. 28). Nun muss die vertikale Distanz zwischen einem Eckpunkt und dem betrachteten Pixel berechnet werden, woraus sich ein Prozentsatz bezogen auf den Abstand zwischen den Eckpunkten ergibt. Da die Breite des Überlappungsbereichs an den Eckpunkten P_{3l} und P_{2l} bekannt ist, kann durch den berechneten Prozentsatz und die Steigung für jede Pixelzeile die korrekte Breite berechnet werden. Durch den x-Wert jedes Pixelortes kann für jeden Pixelwert durch die berechnete Breite eine Gewichtung berechnet werden, sodass die Übergänge fließend werden und sich dynamisch an neue Kameratransformationen anpassen können. Diese Berechnung wird für drei verschiedene Steigungen durchgeführt, das bedeutet für all jene Kanten, die das mittlere Panoramabild schneiden.
Beispielhaft beginnt die Berechnung bei P_{3l} mit dem Speichern der aktuellen Breite $P_{3l}.x$. Die Breiten der jeweils darüber liegenden Zeilen hängen von der Steigung $\frac{\Delta y_{L1}}{\Delta x_{L1}}$ ab. Für eine beliebige Zeile y oberhalb von P_{3l} berechnet sich die Breite b_y durch:

$$b_y = P_{3l}.x - \frac{P_{3l}.y - (y - vOffset)}{\Delta y_{L1}} * \Delta x_{L1} \tag{15}$$

Die Breiten b_y zwischen P_{3l} und P_{2l} werden folgendermaßen bestimmt:

$$b_y = P_{3l}.x - \frac{y - vOffset - P_{3l}.y}{\Delta y_{L2}} * \Delta x_{L2} \tag{16}$$

Dabei ist stets zu beachten, dass Δx_{L2} immer auch ein negatives Vorzeichen haben kann, je nach Lage von P_{3l} und P_{2l}. Denn es gilt $\Delta x_{L2} = P_{3l}.x - P_{2l}.x$. Im Beispiel in Abbildung 28 tritt dieser Fall auf. Die Breiten b_y unter P_{2l} werden durch folgende Formel beschrieben:

$$b_y = P_{2l}.x - \frac{y - vOffset - P_{2l}.y}{\Delta y_{L3}} * \Delta x_{L3} \tag{17}$$

In den Formeln (15), (16) und (17) wurden jeweils die Prozentsätze ermittelt, wie weit die betrachtete Zeile y sich von dem Ausgangspunkt (z.B.P_{3l}) entfernt hat im Bezug auf den absoluten $\Delta y_{L1...L3}$ Wert. Da es sich bei dem Steigungsdreieck um ein rechtwinkliges handelt, lassen sich darauf die Strahlensätze anwenden. Der berechnete Prozentsatz lässt sich daher auch auf $\Delta x_{L1...L3}$ beziehen, um den Prozentwert der Breitenänderung zu berechnen.
Nun kann die Gewichtung W jedes Pixels p für das mittlere Teilpanorama in dieser Zeile im Überlappungsbereich berechnet werden:

$$Wmiddle_p = \frac{p.x - hOffset}{b_{p.y}} \tag{18}$$

$$Wleft_p = 1 - Wmiddle_p \tag{19}$$

Analog wird diese Technik verwendet, um die Gewichtungen für das rechte Teilpanorama zu ermitteln. In Abbildung 29 ist dieser dynamische fließende Übergang erkennbar, wobei der Anschaulichkeit wegen der Anteil des mittleren Bildes auf 0 gesetzt wurde.

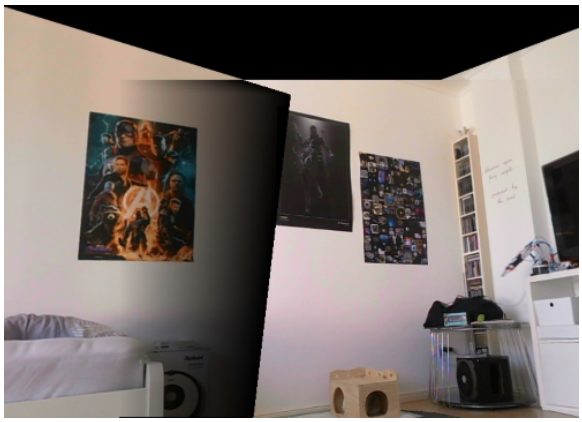


Abbildung 29: Blending mit dynamischen Pixelwertgewichten für das linke und mittlere Bild (Pixelwerte für mittleres Bild auf 0 gesetzt)

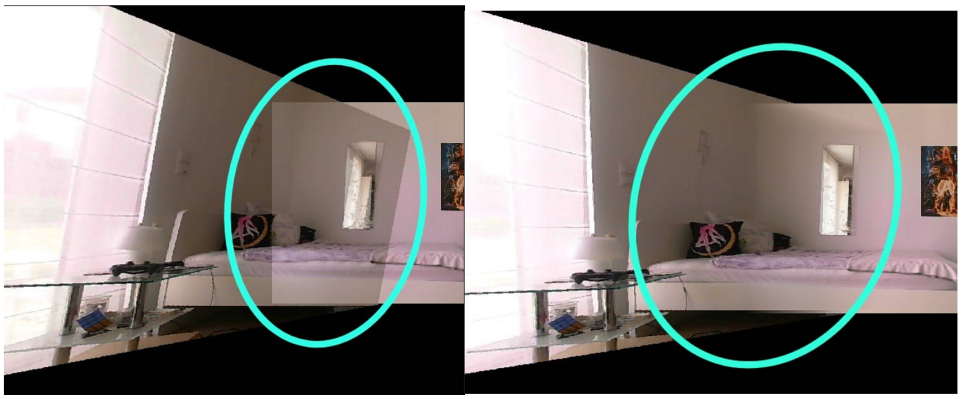


Abbildung 30: links: Blending mit konstanter Pixelwertgewichtung 0.5
rechts: optimiertes Blending mit dynamischen Pixelwertgewichten

Dieses Blendingverfahren schafft einen horizontal verlaufenden fließenden Farbübergang (siehe Abb. 30). Dies kann auch angewendet werden, um einen vertikalen Helligkeitsangleich zu erzielen.
Dazu werden jedoch nicht die Zeilenbreiten des Schnittbereichs gesucht, sondern die Höhen der Spalten, damit eine vertikale Gewichtung berechnet werden kann, bei der jede Spalte eine eigene absolute Höhe hat, nach der diese Gewichtung bemessen wird. Zur Berechnung der Höhe pro Spalte werden die y-Werte der Punkte auf $\overline{P_{0l}P_{3l}}$ und $\overline{P_{1l}P_{2l}}$ gesucht, welche die

Spalte schneiden. Auch diesmal werden diese Werte für die eine beliebige Spalte x mit dem Strahlensatz berechnet:

$$y_{up} = P_{3l}.y - \frac{P_{3l}.x - (x - hOffset)}{\Delta x_{L1}} * \Delta y_{L1} \tag{20}$$

$$y_{down} = P_{2l}.y - \frac{(x - vOffset) - P_{2l}.x}{\Delta x_{L3}} * \Delta y_{L3} \tag{21}$$

Die gesuchte Spaltenhöhe wird dann folgendermaßen berechnet:

```
//compute height
float height = 480;
if (y_up >= 0) {
    height -= y_up;
}
if (y_down < 480) {
    height -= (480 - y_down);
}
```

Abbildung 31: Berechnung der absoluten Höhe einer Spalte eingegrenzt durch den Überlappungsbereich

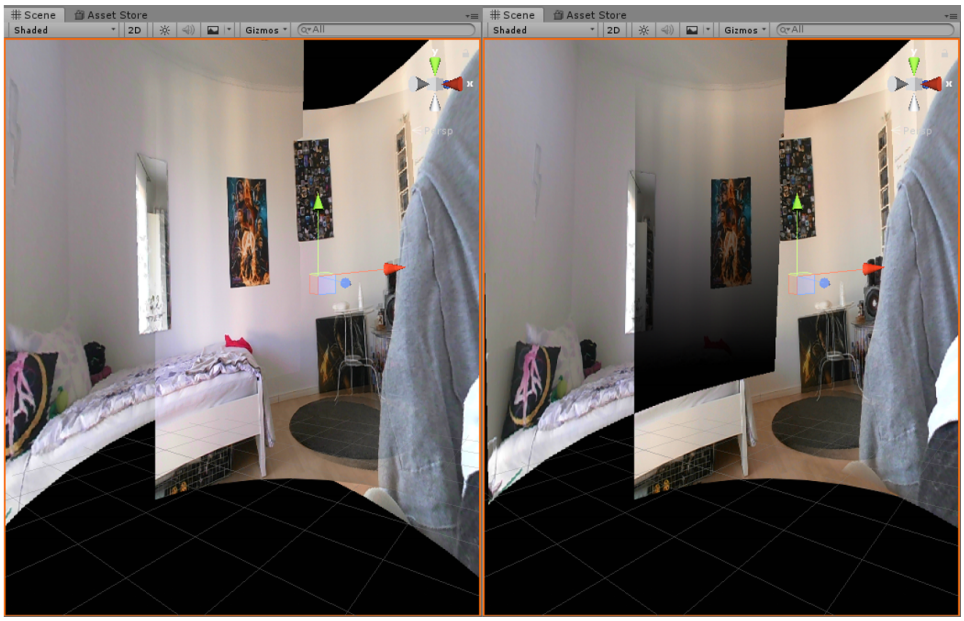


Abbildung 32: vertikaler Gewichtungsverlauf zwischen mittlerem und linken Bild (rechts: Pixelwerte für mittleres Bild auf 0 gesetzt)

Basierend auf dieser Höhe wird ein Gewicht für den aktuellen y-Wert berechnet (siehe Abb. 32). Je nachdem, ob sich der Punkt P_{3l} über oder P_{2l} unter dem mittleren Bild befindet, wechseln die Gewichte für das linke und

mittlere Bild. Denn mal ist das linke Bild oben und bleicht nach unten hin langsam aus und mal ist es das mittlere Bild.

4.3.2 Blending bei ungünstiger Kameraposition

Die Panoramakonzeption sieht vor, dass sich jeweils nur zwei Kamerabilder schneiden und verblendet werden müssen. Dennoch soll die Software so fehlertolerant und die Bewegung der Kameras so frei wie möglich sein. Selbst, wenn der Field of View nicht optimal ausgenutzt wird, soll berücksichtigt werden, dass alle drei Inputframes eine Schnittmenge haben können (siehe Abb. 33).

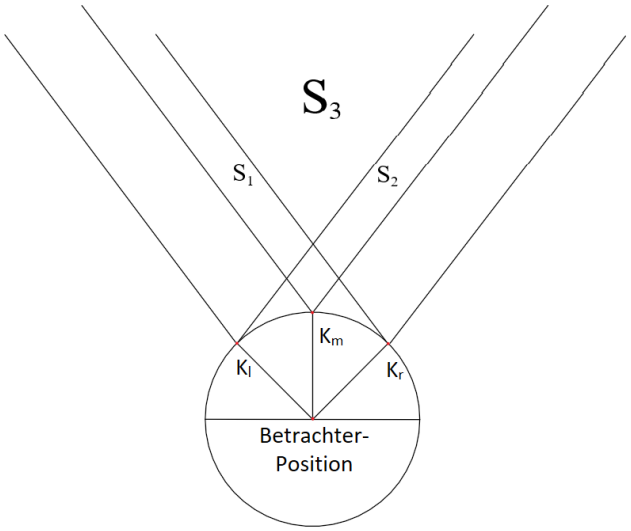


Abbildung 33: Die Kameraausrichtungen verursachen einen Schnitt aller drei Sichtbereiche

Das erste Problem besteht in der Ermittlung der Breite $b_{tripple}$ in jeder Zeile des Schnittbereiches zwischen allen drei Kameras (siehe Abb. 34). Da die Breiten b_{left} und b_{right} jedoch schon aus den Strahlensatztermen des bisherigen Blendingverfahrens hervorgehen, ist $b_{tripple}$ definiert durch:

$$b_{tripple} = b_{left} + b_{right} - middleImg.Width \tag{22}$$

Das zweite Problem besteht in der Einbeziehung des mittleren Bildes. Zunächst werden daher die beiden äußeren Bilder jeweils mit dem mittleren Bild verblendet mit den bekannten Gewichten (aus Formeln 18 und 19): $Wleft$ und $Wmiddle_l$ (für S_1), sowie $Wright$ und $Wmiddle_r$ (für S_2).

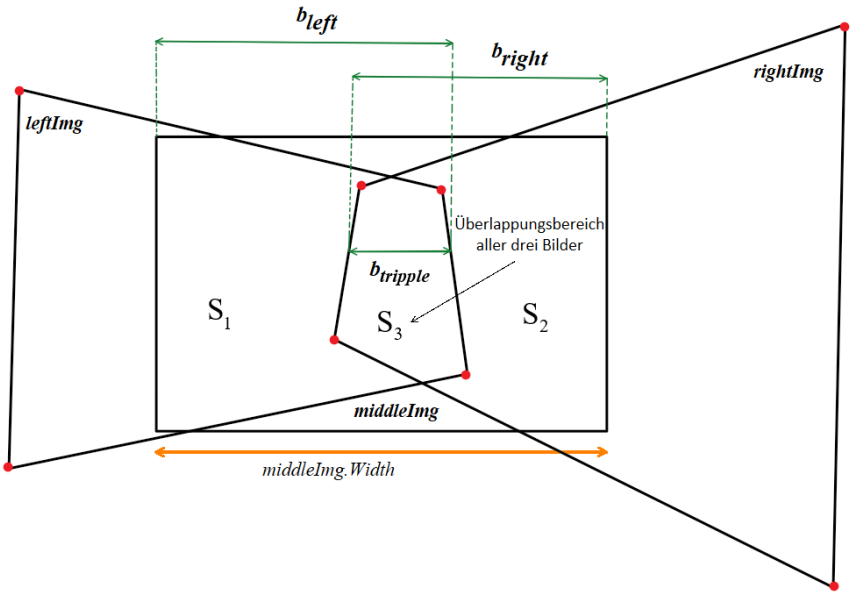
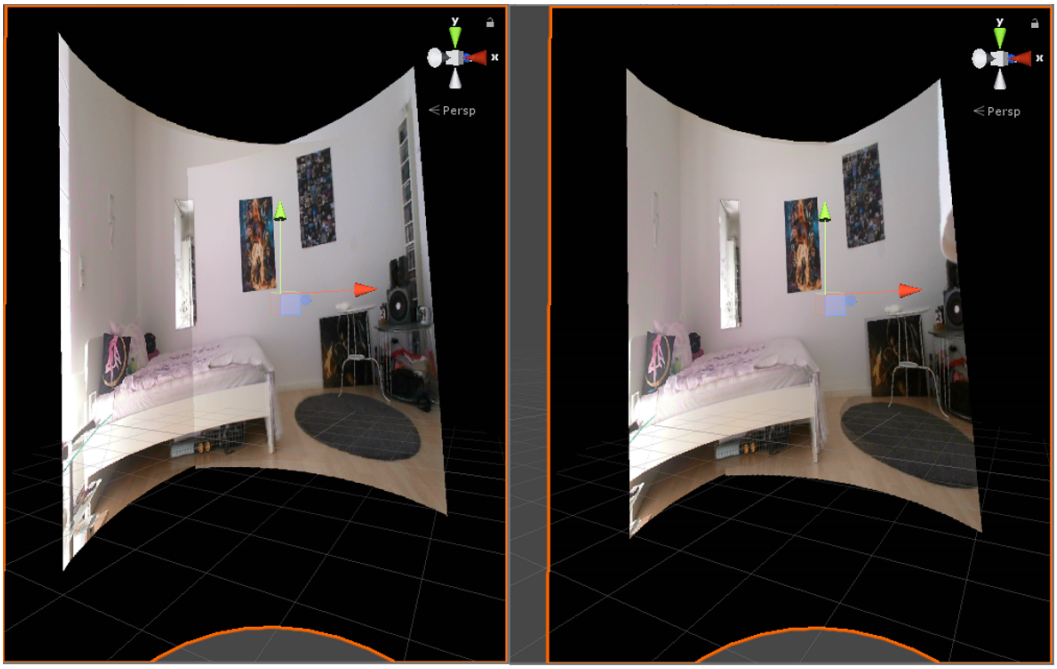


Abbildung 34: Berechnung von $b_{tripple}$ aus b_{left} und b_{right}



a) b)

Abbildung 35: Auswikungen der optischen Optimierung: a) vorher: Farbkante, b) nachher: fließender Übergang

Anschließend werden die resultierenden Pixelwerte nochmal bzgl. S_3 mit den neuen Gewichten $Wright_{new}$ und $Wleft_{new}$ geblendet, wobei p der betrachtete Pixel ist und es gilt:

$$Wright_{new} = \frac{p.x - hOffset + b_{right} - middleImg.Width}{b_{tripple}} \tag{23}$$

$$Wleft_{new} = 1 - Wright_{new} \tag{24}$$

Abbildung 35 a) zeigt das unangepasste Blending bzgl $S3$, wohingegen Abbildung 35 b) das Ergebnis der soeben erläuterte Methode zeigt.

4.3.3 Beschleunigung auf der CPU

Ohne jegliche Optimierungen schafft der Algorithmus die Berechnung nur knapp unter einem Frame pro Sekunde. Durch Verbesserungen der Algorithmen kann die Verarbeitungsgeschwindigkeit des Programms auf bis zu 7 FPS gesteigert werden. Im Folgenden werden die Optimierungen dazu beschrieben.
OpenCVSharp bietet die Möglichkeit auf einzelne Pixelorte per $At()$ und $Set()$ zum Lesen und Schreiben von Pixelwerten zuzugreifen. Diese Methoden sind effektiv, wenngleich sehr ineffizient, da zumindest im Debug-Modus bei jedem Pixelzugriff die Indizes auf Validität überprüft werden. Doch auch im Release-Modus ist die Ausführung der $At()$ und $Set()$ Funktionen im Vergleich zu Indexern, die man durch typspezifische OpenCVSharp Matrizen erstellen kann nur halb so schnell (siehe Abb. 36). Das Panoramabild ist ca. 2.000.000 Pixel groß bei einer Auflösung von 480 x 640 pro Kamerabild. Bei einer Berechnung auf der CPU ist es außerdem ressourcensparend, wenn man nur über die Panoramabereiche iteriert, die auch die erwarteten Bildinformationen enthalten, also nicht schwarz sind. Die schwarzen Bereiche des Panoramabildes haben in den drei Farbkanälen jeweils den Wert 0. Bei dieser Maßnahme ist die Schleifenbedingung für das Inkrement bzw. Dekrement der Zeilenvariable, dass der Farbwert nicht 0 sein darf. Ist dies der Fall, so wird die aktuelle Zeile nicht länger untersucht. Ein entscheidender Nachteil dieses Optimierungsverfahrens ist jedoch, dass bei einer schwachen Beleuchtung, oder bei stark Licht absorbierenden Objekten schwarze Streifen entstehen können, da man durch den Algorithmus annimmt es sei keinerlei Information mehr in der Zeile vorhanden und so zeichnet er direkt in der nächsten Zeile weiter. Auch können die Ecken des Panoramas nicht gänzlich dargestellt werden. Wenn auch ein schwerwiegender Fehler, wie er in Abbildung 37 zu sehen ist, so führt diese Optimierung auch wieder zu einer Verdopplung der Bildfrequenz.

CPU-Geschwindigkeitsoptimierung			Optische Optimierung	Bildfrequenz
Indexer statt At() und Set()	Optimierte Iteration	Parallelisierung	Helligkeitsangleich	
				0,3 FPS
x				0,7 FPS
	x			0,6 FPS
		x		0,3 FPS
x	*x*			1,7 - 3,7 FPS
x	*x*	*x*		3,0 - 6,7 FPS
x	*x*	*x*	*x*	3,0 - 5,0 FPS

Abbildung 36: FPS-Angaben der CPU-Optimierungen bei einer Homographiebe-rechnung alle 10 Frames

Eine Parallelisierung der Schleifen ist als alleinige Optimierung recht wenig effektiv. Jedoch kann sie in Kombination mit den anderen Methoden den Algorithmus erneut in der Bildfrequenz verdoppeln mit $3,0$ bis $6,7$ FPS. Schwankungen der FPS lassen sich dadurch erklären, dass alle 10 Frames eine neue Homographie berechnet wird. Ersichtlich ist auch, dass die Berechnung der dynamischen Gewichtungen im Helligkeitsangleich nicht besonders zeitaufwändig sind (siehe Abb. 36).

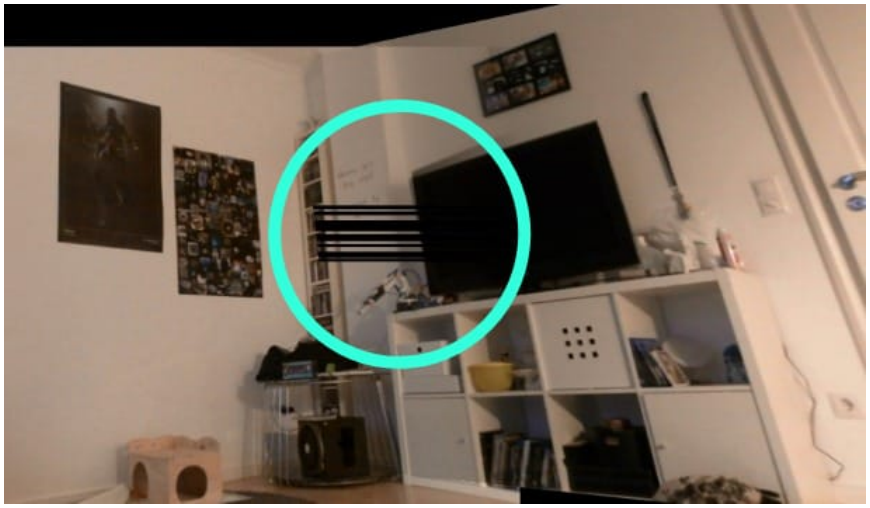


Abbildung 37: Fehler bei zu dunklen Bildbereichen

4.3.4 Beschleunigung auf der GPU

Um ein echtzeitfähiges System zu entwickeln, muss man letztendlich auf die Architektur der GPU zurückgreifen. Der Grafikprozessor ist darauf spezialisiert sehr viele, aber einfache Grafikberechnungen parallel zu bearbeiten. Unity bietet dazu eine Auswahl an verschiedenen Shadern, wie zum Beispiel den Computeshader. Mit einem solchen Shader kann die perspektivische Verzerrung, sowie das Blending deutlich schneller berechnet werden, da die Arbeit auf viele sogenannte *ThreadGroups* aufgeteilt wird, welche auch nochmal viele Threads enthalten (siehe Abb. 38).

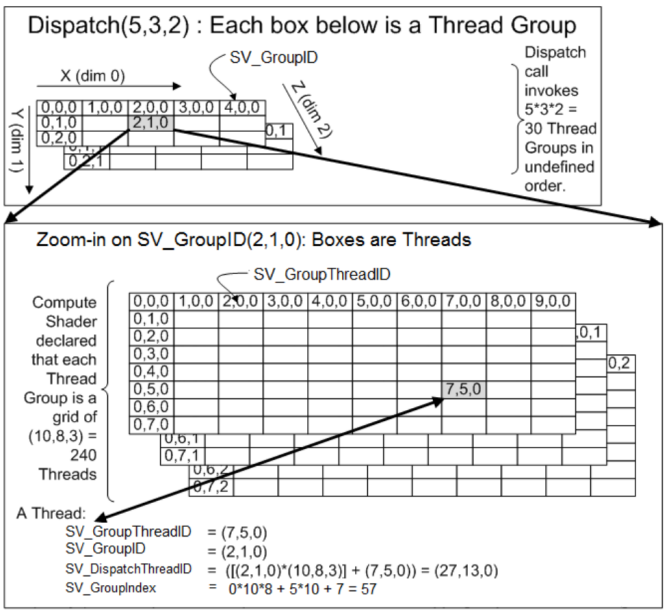


Abbildung 38: SV DispatchThreadID [dis]

Mit $Dispatch(x, y, z)$ wird von der CPU aus festgelegt, dass $x * y * z$ ThreadGroups erzeugt werden sollen. Im Computeshader selbst wird durch $[numthreads(u, v, w)]$ bestimmt, wie viele Threads in einer ThreadGroup enthalten sind, und zwar $u * v * w$. Standardmäßig sind 64 Threads in einer Gruppe (siehe Abb. 39).

Wenn man die Anzahl der Threads an die Größe der Panoramatextur anpassen möchte, so werden die Texturdimensionen durch die Threadanzahl pro ThreadGroup geteilt (siehe Abb. 40). So erhält man im Computeshader pro Pixel einen einzelnen Thread, den man durch eine eindeutige ThreadID einzeln mit $ThreadID.xy$ ansprechen und dem Pixel an dieser Stelle einen Wert zuschreiben kann.
Im Shader wird die Programmiersprache HLSL verwendet, weswegen es

```
// each threadgroup has 64 threads
[numthreads(8,8,1)]
void CSMain(uint3 id : SV_DispatchThreadID)
{
    // TODO: insert actual code here!
}
```

Abbildung 39: Iteration über das Panorama per Threads

```
blend.Dispatch(kernel2, 1920 / 8, 960 / 8, 1);
```

Abbildung 40: Festlegung der ThreadGroups

nicht möglich ist OpenCV Funktionen zu verwenden. Die Idee ist also die aktuellen zwei Homographien für rechtes und linkes Teilpanorama an den Shader zu übergeben und das perspektivische Verzerren, auch *Warpen*, der Texturen auf der GPU auszuführen. Dazu greift man mit Hilfe der Shader-Threads auf das Ergebnispanoramabild zu, das anfangs noch schwarz ist. Für jeden Pixel wird mit der inversen Homographie berechnet, wo der Pixel sich im unverzerrten Originalbild befinden würde. Am resultierenden Pixelort wird der Wert in diesem Originalbild gelesen und an der aktuellen Position im Ergebnisbild eingetragen (siehe Abb. 41). Wenn der errechnete Pixelort außerhalb der originalen Bilddimensionen liegt, wird er ignoriert. Liegt ein Pixelort zwischen zwei ganzzahligen Indizes, so wird der Pixelwert interpoliert.

```
//apply left homography and draw left panimg
org_left.xyz = mul(inverse_lefthomography, float3(threadid.x,threadid.y,1));
org_left.x /= org_left.z;
org_left.y /= org_left.z;
if (org_left.y < 480 && org_left.y >= 0 && org_left.x < 640 && org_left.x >= 0) {
    leftpan = 1;
}
else {
    leftpan = 0;
}
```

Abbildung 41: GPU Pendant zu *OpenCV warpPerspective()*

In der Implementierung wird erstmal kein Farbwert eingetragen, sondern durch einen Boolean-Wert (im Beispiel $leftpan$) gespeichert, ob der Pixel das geprüfte Teilpanorama trifft. Diese Überprüfung wird für alle drei Originaltexturen durchgeführt. Falls $middlepan$ und $leftpan$ oder $rightpan$ 1 sind, so müssen die entsprechenden Pixelwertgewichtungen berechnet werden. Auch hier wird das optimierte Blendingverfahren für den Helligkeitsangleich verwendet. Für den Fall, dass alle drei Booleanwerte 1 sind (siehe Kapitel 4.3.2).

Ein weiteres Geschwindigkeitsproblem bildet jedoch auch das Umwandeln der Texturformate. Die Texturdaten werden in OpenCV spezifischen Matrizen ($OpenCV.Mat$) gespeichert. Jedoch braucht Unity eine Textur der Klasse $Texture2D$, um das fertige Panorama mappen zu können. Es muss also zwangsläufig irgendwo im Algorithmus eine solche Umwandlung stattfinden. Es gibt nun zwei Methoden, wie die Texturen auf der Graphikkarte überschrieben werden können.
Die erste Methode wäre auf der CPU die rohen Texturdaten der $OpenCV.Mat$ zu verwenden, um die Textur auf der GPU mit der $Apply()$-Methode zu aktualisieren (siehe Abb. 42).

```
// update texture
m_MainTexture.LoadRawTextureData(panorama_final.Data, 1920 * 960 * 3);
m_MainTexture.Apply();
```

Abbildung 42: Updaten der Texturen per *Apply()*

Eine andere Methode ist es, die Texturdaten durch ein natives C++ Plugin direkt über OpenGL durch $glTexSubImage2D$ auf den Grafikprozessor zu laden (siehe Abb. 45). Alle Plugin-Funktionen müssen zunächst in das Unity-Skript importiert werden (siehe Abb.43). Dazu wird auf der CPU Seite für jedes Kamerabild eine Texture2D erstellt und dessen Adresse an das Plugin übergeben (vgl. Abb. 44). Im Plugin werden OpenCV.Mat Objekte erstellt, in welche die Frames gelesen werden. Diese Daten werden zum Updaten durch OpenGL genutzt. Durch $GL.IssuePluginEvent()$ wird dieser Rendering-Updateauftrag in eine Warteschlange gestellt (siehe Abb. 46). Dies ist notwendig, da diese Texturupdates von Unitys Rendering Thread ausgeführt werden müssen, sodass überhaupt etwas gezeichnet werden kann.

```
[UnmanagedFunctionPointer(CallingConvention.Cdecl)]
public delegate void MyDelegate(string str);
[DllImport("Plugin")]
public static extern void SetDebugFunction(IntPtr fp);
[DllImport("Plugin")]
static extern IntPtr GetRenderEventFunc();
[DllImport("Plugin")]
static extern void SetTextureFromUnity(int texhandle1, int texhandle2, int texhandle3);
[DllImport("Plugin", CallingConvention = CallingConvention.Cdecl)]
static extern IntPtr GetLeftMat(ref int height, ref int width);
[DllImport("Plugin", CallingConvention = CallingConvention.Cdecl)]
static extern IntPtr GetRightMat(ref int height, ref int width);
[DllImport("Plugin", CallingConvention = CallingConvention.Cdecl)]
static extern IntPtr GetMiddleMat(ref int height, ref int width);
```

Abbildung 43: Einbindung der Plugin Funktionen

```
//send textures adresses to plugin
SetTextureFromUnity(leftimg_tex.GetNativeTexturePtr().ToInt32(),
                    middleimg_tex.GetNativeTexturePtr().ToInt32(),
                    rightimg_tex.GetNativeTexturePtr().ToInt32());
```

Abbildung 44: Setzen der Textureadressen im Plugin

```
// Plugin function to handle a specific rendering event
static void UNITY_INTERFACE_API OnRenderEvent(int eventID)
{
    //update texture data
    cap1 >> leftmat;
    glBindTexture(GL_TEXTURE_2D, g_tex_left);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 640, 480, GL_BGR, GL_UNSIGNED_BYTE, leftmat.data);

    cap2 >> middlemat;
    glBindTexture(GL_TEXTURE_2D, g_tex_middle);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 640, 480, GL_BGR, GL_UNSIGNED_BYTE, middlemat.data);

    cap3 >> rightmat;
    glBindTexture(GL_TEXTURE_2D, g_tex_right);
    glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 640, 480, GL_BGR, GL_UNSIGNED_BYTE, rightmat.data);


}
```

Abbildung 45: Texturupdate mit OpenGL

```
GL.IssuePluginEvent(GetRenderEventFunc(), 1);
```

Abbildung 46: Einreihen in die Warteschlange des *Unity* Renderthreads

Nun müssen jedoch auch die $OpenCVSharp.Mat$ -Objekte im C♯ Skript mit den Rohdaten aus dem Plugin gefüllt werden, damit u.a. der SIFT-Algorithmus durchgeführt werden kann (siehe Abb. 47). Eine Möglichkeit ist, die Daten von der GPU mit einem $AsyncGPUReadback$ zu lesen, was aber erstens nicht optimal ist, da es asynchron zum aktuellen Datenbestand wäre und zweitens nicht OpenGL unterstützt. Daher ist es sinnvoller einen $char$-Pointer auf die Rohdaten vom Plugin zurück an das C♯-Skript zu übergeben und die OpenCVSharp-Matrizen zu füllen (siehe Abb. 48).

Im Hinblick auf die Performance ist $LoadRawTextureData()$ mit $Apply()$ in dieser Situation der Panoramaerstellung sogar schneller mit bis zu 112 FPS. Wohingegen man mit der Verwendung von $glTexSubImage2D()$ mit OpenGL auf höchstens 90 FPS kommt. Dennoch kann $LoadRawTextureData()$ in anderen Situationen deutlich langsamer sein, da es nur auf dem Hauptthread von Unity ausgeführt werden kann und den Renderthread so beim Laden blockiert. Dies wird problematisch, wenn außer der Berechnung der Panoramatextur noch weitere Berechnungen vom Hauptthread getätigt werden müssen. Außerdem verwendet $LoadRawTextureData()$ eine Speicherkopie, welche bei einem Laufzeit-

```
IntPtr ptr = GetLeftMat(ref height, ref width);
byte[] arr = new byte[height * width * 3];
Marshal.Copy(ptr, arr, 0, arr.Length);
Marshal.Copy(arr, 0, leftimg_mat.Data, arr.Length);

ptr = GetMiddleMat(ref height, ref width);
Marshal.Copy(ptr, arr, 0, arr.Length);
Marshal.Copy(arr, 0, middleimg_mat.Data, arr.Length);

ptr = GetRightMat(ref height, ref width);
Marshal.Copy(ptr, arr, 0, arr.Length);
Marshal.Copy(arr, 0, rightimg_mat.Data, arr.Length);
```

Abbildung 47: Füllen der Matrizen für *OpenCVSharp* Methoden

```
extern "C"  UNITY_INTERFACE_EXPORT uchar*  UNITY_INTERFACE_API GetLeftMat(int* height, int* width) {
    *height = leftmat.rows;
    *width = leftmat.cols;
    return leftmat.data;
}
```

Abbildung 48: Übergeben der *OpenCV* Matrix Adresse an das Unityskript

Texturupdate zu Verzögerungen führen kann [uni19]. Ebenfalls muss nach diesem Funktionsaufruf auch immer $Apply()$ aufgerufen werden, welches alle Änderungen auf der GPU aktualisiert. Auch diese Operation ist potentiell kostspielig, weshalb zwischen den Aufrufen möglichst viele Pixeländerungen gemacht werden sollten [uni19]. Jedoch hilft dieser Ratschlag seitens des Unity Manuals nicht, wenn hoch-frequentierte Textur-Updates mit $Apply()$ durchgeführt werden müssen. Im Falle der vorgestellten Panoramaerstellung ist das native Plugin möglicherweise nur deshalb langsamer, da die Auflösung der drei Kamerabilder nur 480 x 640 beträgt. Wird die Auflösung erheblich größer, ist das native Plugin womöglich besser als Textur Update-Verfahren geeignet.

4.3.5 Threadpools und Coroutinen

Selbst nach der Optimierung mit Hilfe der Berechnungen auf der GPU gibt es Einbrüche der FPS, da die OpenCV-Methode DetectAndCompute() und KnnMatch(), welche die SURF-Merkmale finden und diese Merkmale mit anderen vergleichen, lange dauern. Das Matchen der Merkmale zwischen zwei Bildern dauert im Schnitt 20 Millisekunden ($\hat{=}$ 50 FPS). Da das für die rechte und linke Seite berechnet wird, muss mit einem Gesamtaufwand von 40 Millisekunden gerechnet werden. Das Bestimmen der SURF-Merkmale dauert ca. 130 Millisekunden ($\hat{=}$ 7.6 FPS). Diese Merkmale müssen für drei Bilder berechnet werden. Also beträgt der Aufwand ca. 390 Millisekunden. Insgesamt beanspruchen diese OpenCV-Funktionen dann 430 Millisekunden ($\hat{=}$ $2,32$ FPS). Durch das Verwenden von Coroutinen

können diese Berechnungen zeitlich auf mehrere Frames verteilt werden. Bis die Berechnung der neuen Homographien beendet ist, wird so lange die aktuelle Homographie genutzt. Dennoch kann die Laufzeit der einzelnen OpenCV Funktionen nicht beeinflusst werden, oder durch Coroutinen auf mehrere Frames aufgeteilt werden. Diese Funktionen können somit nicht dermaßen im Hintergrund laufen, dass die FPS-Zahl davon nicht beeinträchigt würde. Wenn die SURF-Merkmale berechnet werden, erreicht der Algorithmus für einen Moment höchstens ca. 7 FPS. Für dieses Problem bieten Threadpools eine Lösung (siehe Abb. 49). Nur durch Parallelisierung lässt sich der Einbruch auf unter 10 FPS vermeiden. Wenn für jeden 10. Frame unter Verwendung von Threadpools eine neue Homographie berechnet wird schwankt die Leistung zwischen 60 und 90 FPS (unter Verwendung des nativen Plugins). Es ergibt sich also ein System, das auf die Veränderung der Kameraanordnung bzgl. Translation, Rotation und Skalierung zeitnah reagiert.

```
if (noteveryframe % 40 == 0)
{
    ThreadPool.QueueUserWorkItem(findmiddlekeypoints);
    ThreadPool.QueueUserWorkItem(findRightHomography);
    ThreadPool.QueueUserWorkItem(findLeftHomography);
}
```

Abbildung 49: Threadpools;parallelisierte Berechnung der Homographien alle 40 Frames

5 Bewertung

5.1 Möglichkeiten und Grenzen des technischen Setups

Eine 360°-Rundumsicht ist mit diesem Panoramasystem nicht möglich, da sich alle Frames durch eine Homographie perspektivisch auf genau einen Frame beziehen. Je weiter sich die Perspektiven der Bilder von diesem Bezugsframe wegbewegen, desto stärker werden diese verzerrt. Diese Vorgehensweise wäre für ein Setup von mehr als 180° nicht geeignet.
Ein Panorama aus verschiedenen Kamerabildern zu entwickeln bringt jedoch die Möglichkeit mit sich, verschiedene Perspektiven miteinander zu verschmelzen. Auch, wenn sich die Kameras nicht unbedingt auf engem Raum befinden müssen. Die einzige wichtige Anforderung ist, dass es einen ausreichend großen Überlappungbereich gibt, der genügend Merkmale bietet, an dem sich der Algorithmus orientieren kann. Dennoch gibt es auch kritische Situationen, für die es bisher keine Lösung zu geben scheint. Bei optimaler Ausrichtung laufen die Geraden durch die optischen Zentren in einem Knotenpunkt hinter den Kameras zusammen. Wie bereits in Kapi-

tel 3.1 erläutert, liegt dadurch die eigentliche Betrachterposition hinter den Kameras. Das bringt den Nachteil mit sich, dass unter anderem tote Winkel entstehen und perspektivische Konstellationen entstehen können, die sich nicht simpel ineinander transformieren lassen. Es folgen 4 Szenarien, bei denen das Stitching aus perspektivischen Gründen schon gar nicht korrekt erfolgen kann. Die folgende Beschreibung nimmt Bezug auf Abbildung 50.

Abbildung 50 *Szenario 1* zeigt eine Situation, in der sich hintereinander gereihte Objekte auf der optischen Achse befinden, welche die Betrachterposition schneidet, jedoch nicht die Kamerapositionen. Eigentlich würde der Betrachter nur das vorderste Objekt sehen können, da dies die anderen auf dieser Achse verdeckt, nicht jedoch aus allen anderen Perspektiven. Daher ist durch die Kameras ein leicht seitlicher Blick auf die Objekte zu sehen, jedoch von der jeweils anderen Seite. Dies macht das Stitching der Frames dieses Szenarios äußerst schwierig.

Abbildung 50 *Szenario 2* stellt eine ähnliche Situation dar. Hier handelt es sich ebenfalls darum, dass ein Objekt von einer anderen Perspektive aus dargestellt wird. Die linke Seite des Objekts ist das einzige, was im linken Kamerabild überhaupt davon sichtbar ist. Im rechten Bild ist eben diese Seite fast gar nicht zu sehen. Es ist unmöglich sinnvolle Matches zu finden.

Abbildung 50 *Szenario 3* wird ein Objekt auf einer optischen Achse von K_m völlig von einem anderen Objekt verdeckt. Das Objekt wird aber durchaus von der anderen Kamera abgebildet. Nur der blaue Zylinder kann an dieser Stelle richtige Keypoint-Matches hervorrufen.

Abbildung 50 *Szenario 4* zeigt die Auswirkungen des toten Winkels zwischen den Kameras. Objekte in diesem Bereich werden gar nicht erst dargestellt, sofern sie komplett in diesem Bereich liegen, oder werden nur zum Teil abgebildet, wie die beiden Zylinder. Da diese Objekte sehr nah sind und jeweils nur von einer Kamera aufgezeichnet werden, können sie selbst keine Keypoint-Matches hervorrufen und verdecken zusätzlich noch einen Großteil der eigentlichen Überlappungsfläche zwischen den Kamerabildern.

Alle Szenarien haben gemeinsam, dass diese negativen Auswirkungen der unterschiedlichen Perspektiven besonders groß sind, je näher Gegenstände vor der Kamera stehen.

Falls durch andere Gegenstände im Raum, die nicht von diesen perspektivischen Nachteilen betroffen sind, doch sinnvolle Matches und damit eine sinnvolle Homographie entsteht, so entstehen Artefakte beim Blending der Teilpanoramen, wie zum Beispiel durchsichtige und in sich verschobene Objekte (siehe Abb. 52).

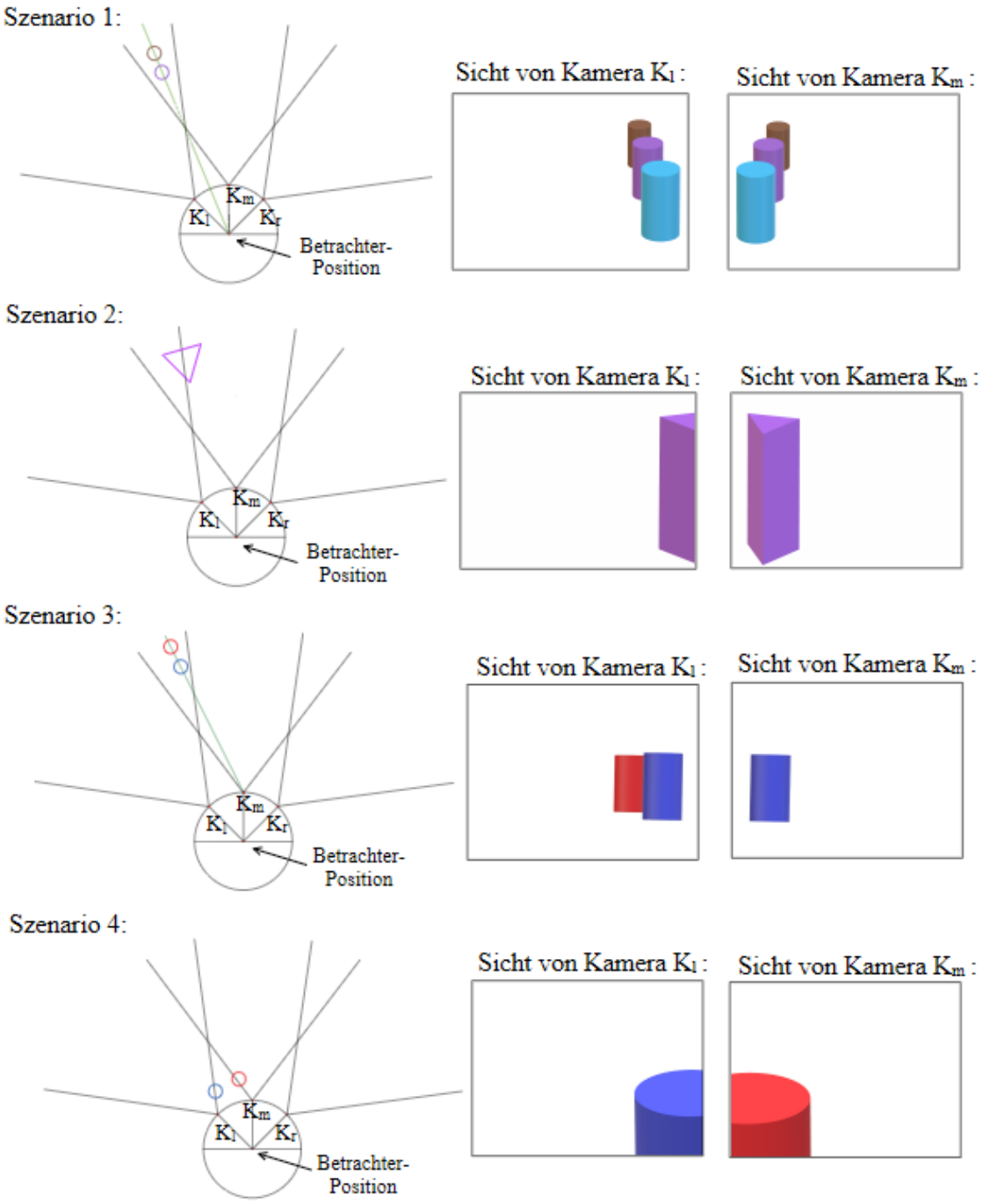


Abbildung 50: Szenario 1-4 zeigen die Grenzen des technischen Setups bzgl. perspektivischer Eigenschaften der Kameraaufnahmen bei bestimmten Objektkonstellationen

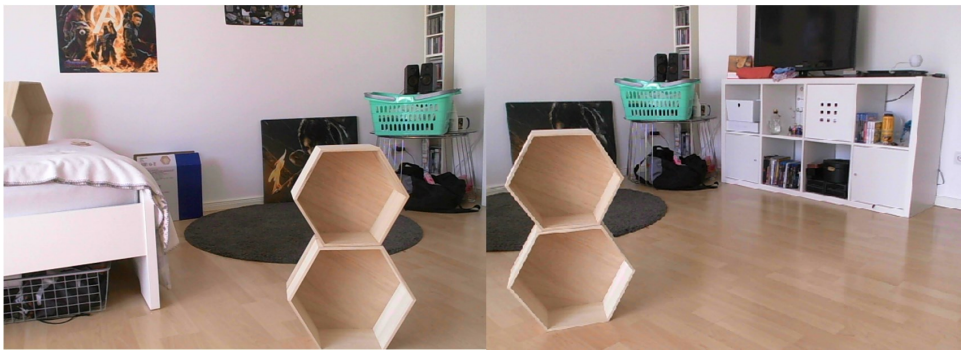


Abbildung 51: Holzregal ist zu nah am Kamerasetup und verursacht perspektivisches Problem; links: Bild der mittleren Kamera, rechts: Bild der rechten Kamera

Abbildung 52: Blendingfehler durch perspektivisches Problem; links: Blending mit konstanter Gewichtung 0,5; rechts: Blending mit linearer Interpolation

Dieses Beispiel (siehe Abb. 51) zeigt, dass die dargestellten Perspektiven des Holzregals aus dieser Nähe zu unterschiedlich sind, um sinnvoll gematcht und geblendet zu werden (ähnlich zu *Szenario 2*). Zudem werden Hintergrundobjekte unterschiedlich verdeckt, was das Regal an den Rändern durchsichtig erscheinen lässt (ähnlich zu *Szenario 3*). Dieser Effekt kann durch das optimierte Blending jedoch etwas abgeschwächt werden, wie auf der rechten Seite in Abbildung 52 zu sehen ist.
Die Stärke des Setups liegt also definitiv darin weiträumige und großflächige Aufnahmen zu verschmelzen. Daher ist ein Einsatz bei Drohnen oder Panzerfahrzeugen durchaus denkbar.

5.2 Vergleich mit anderen Verfahren

Ein anderes Verfahren, um ein Panorama zu erzeugen, ist zum Beispiel das *Mosaicing*. Der Unterschied zu diesem Verfahren ist, dass die Aufnahmen

diesmal von einem einzigen Point of View aufgenommen werden. Eine Kamera rotiert dabei um ihre eigene Achse und nimmt dabei kontinuierlich Bilder auf. Diese Aufnahmen werden dann ebenfalls zu einem Panorama zusammengesetzt, wobei jedoch nur die mittleren vertikalen Streifen für das Panorama verwendet werden [PBE99].

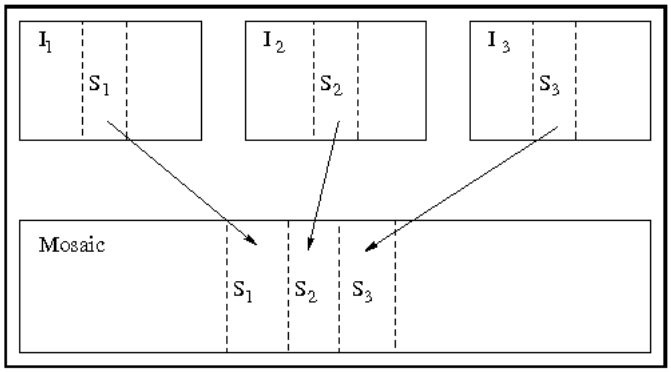


Abbildung 53: Prinzip des Mosaicings [PBE99]

Es ist möglich mit *Mosaicing* ein stereoskopisches Panorama zu erzeugen mit nur einer Kamera. Dazu wird die Tatsache ausgenutzt, dass die Kamera rotiert. Nicht der mittlere vertikale Bildstreifen wird diesmal als Bildfragment gewählt, sondern für jedes Auge ein rechts und links daneben liegender Streifen.

Ein weiteres Verfahren ist die Panoramaerstellung durch eine Tonnenverzerrung. Dazu wird jeder Frame durch eben diese Verzerrung bearbeitet, sodass nicht die vollständige Perspektive transformiert wird. Lediglich die Überlappungsbereiche sind davon stark betroffen. Diese sind dann so gekrümmt, dass das Bild jeweils an seinen Nachbarn gestitcht werden kann [Sze06].

Abbildung 54: drei rohe Bilder dreier Kameras aus verschiedenen Positionen

Ein Nachteil ist, dass die unterschiedlichen Perspektiven nicht in eine einzige Perspektive transformiert werden und somit die Anordnung der Objekte verfälscht wird. In Abbildung 55 ist dies vor allem an den verbogenen Bordsteinen zu sehen. Ein entscheidender Vorteil ist jedoch, dass

Abbildung 55: Panoramaerstellung aus drei Bildern mit *PanoramaStudio 3 Pro* [pan]

dieses Verfahren leicht genutzt werden kann, um ein 360°-Panorama aus bspw. sieben Kamerabildern zu erstellen.

6 Fazit und Ausblick

Das vorgestellte Online-Panorama für VR bietet die Möglichkeit ein Video-Seethrough zu entwickeln, um eine möglichst große bildliche Darstellung von beschränkt einsehbaren Gebieten zu erzeugen. Durch ein merkmalsbasiertes Verfahren ist eine Dynamik der Kamerapositionen erfüllbar. Das Verfahren ist bis zu einem gewissen Grad tolerant und unempfindlich gegen Skalierung, Rotation und Translation. Durch Nutzung der GPU-Architektur kann darüber hinaus die Echtzeitfähigkeit des Systems sichergestellt werden. Das macht das System in der Praxis besser nutzbar und bietet die Möglichkeit weitere Systemfunktionen, wie diverse Augmented Reality-Tools zu implementieren.

Wie bereits im Grundlagenkapitel 2.2 beschrieben, wird zum Darstellen auf einer VR-Brille das Google VR SDK verwendet. Mit Hilfe des Image Shiftings wird ein räumlicher Eindruck angedeutet, kann aber nicht gänzlich hergestellt werden. Dazu wären 6 statt 3 Kameras nötig, um stereoskopische Darstellung für jedes Auge aufzunehmen.

Ein weiterer Ansatz zur Optimierung wäre die Stabilität der Homographie noch weiter zu erhöhen. Je öfter eine neue Homographie berechnet wird, desto schneller passt sich das System einer neuen Kameraposition an. Jedoch werden auch in diesem Intervall neue Homographien berechnet, selbst, wenn sich die Kamerapositionen nicht verändern. Was man eigentlich erwarten würde ist, dass das Panorama gleich bleibt und die Homographien sich nicht ständig ändern. Dennoch gibt es für einen Pool aus Korrespondenzen mehr als eine passende Homographie, die das Eingangsbild immer ein bisschen anderes verzerrt, als die vorherige oder die nachfolgende Homographie. Diese unterscheiden sich jedoch nur wenig voneinander und das Panorama sieht in jedem Fall stimmig aus. Es wäre

jedoch wünschenswert, wenn es ein stabiles und konsistentes Kriterium gäbe, dass immer ein und die selbe Homographie bei einem gleichen Frameinput erzeugt. Jedoch variieren selbst schon die resultierenden Keypoints, die sich aus dem SURF-Algorithmus ergeben, bei gleichem Input. Das führt natürlich auch zu einer Variation an Matchpoints. Dieser Faktor fördert die Vielfältigkeit der Homographien auch nochmal zusätzlich. Es kann einen störenden Effekt haben, wenn die Bilder sich alle 10 Frames bei ca. 80 FPS - 100 FPS unterschiedlich verzerren. Selbst, wenn diese Unterschiede nur minimal sind, ist es angenehmer, wenn die Homographie bspw. nur alle 40 Frames neu berechnet wird. In einem Echtzeitsystem ist dies immer noch schnell genug, um sich an neue Situationen anzupassen.

Ein weiterer Punkt ist eine optimale Blending-Technik zu finden. Bisher ist das Blending mit einem Helligkeitsangleich nur in die vertikale Richtung, oder in die horizontale Richtung möglich. Die Kombination der Gewichte für horizontales und vertikales Blending durch eine Min()- oder Max()Funktion, Mittelwertfilter oder Multiplikation, führen durch einen zu starken Helligkeitskontrast immer noch zu unerwünschten Kanten.

Max()	1	0,75	0,5	0,25	0
1	1	1	1	1	1
0,75	1	0,75	0,75	0,75	0,75
0,5	1	0,75	0,5	0,5	0,5
0,25	1	0,75	0,5	0,25	0,25
0	1	0,75	0,5	0,25	0

Abbildung 56: Max()-Verknüpfung

Mean()	1	0,75	0,5	0,25	0
1	1	0,875	0,75	0,625	0,5
0,75	0,875	0,75	0,625	0,5	0,375
0,5	0,75	0,625	0,5	0,375	0,25
0,25	0,625	0,5	0,375	0,25	0,125
0	0,5	0,375	0,25	0,125	0

Abbildung 57: Mittelwert-Verknüpfung

Die Tabellen in Abbildung 56 und 57 modellieren einen einfachen Fall. Die Zahlen in weiß unterlegten Zeilen und Spalten sind jeweils die horizontalen und vertikalen Gewichte, die es zu kombinieren gilt. In

rot sind dann die Pixelgewichte aufgeführt, die bei einer Max()- oder Mittelwertverknüpfung auftreten würden. Die Max()-Verknüpfung bietet sich an, da von links nach rechts und von oben nach unten ein fließender Übergang hergestellt wird. Kritisch ist jedoch die Kante, die sich von rechts oben nach rechts unten bildet, sowie die Kante, die sich von links unten nach recht unten abzeichnet. Noch kritischer ist die Lösung mit der Mittelwertverknüpfung zu betrachten. Möchte man Gewicht 0 und 1 gleichmäßig mischen, so erhält man $0,5$. Dieser Wert entspricht weder dem einen noch dem anderen Gewicht wirklich. So entstehen überall an den Grenzregionen Kanten, da die Kontraste zwischen den Gewichten zu hoch sind.
Dennoch erhält man mit dem entwickelten Verfahren des horizontalen Helligkeitsangleichs eine angenehme Rundumsicht. Sofern die Kameras nur um ihren eigenen up-Vektor rotiert sind und sich nicht durch ihre Neigung zu stark vertikal nach oben oder unten verschieben, ist ein vertikaler Helligkeitsangleich auch nicht zwingend von Nöten.

Abbildungsverzeichnis

Literatur

[ble] *Blender*. `https://www.blender.org`. – [Online; accessed 28-09-2019]

[Bor94] BORMANN, Sven: *Virtuelle Realität Genese und Evaluation Addison-Wesley S.76 -81*. 1994

[BTVG06] BAY, Herbert ; TUYTELAARS, Tinne ; VAN GOOL, Luc: *SURF: Speeded Up Robust Features*. 2006

[car] *GoogleVR Google Cardboard*. `https://play.google.com/store/apps/details?id=com.google.samples.apps.cardboarddemo&gl=DE`. – [Online; accessed 17-09-2019]

[dis] *SV DispatchThreadID*. https://msdn.microsoft.com/de-de/vstudio/ff471566(v=vs.80). – [Online; accessed 18-09-2019]

[Fri10] FRIES, Carsten: *Objekterkennung mit SIFT-Merkmalen Studienarbeit Hochschule für angewandte Wissenschaften Hamburg*. 2010

[HS88] HARRIS, Chris ; STEPHENS, Mike: *Ä Combined Corner and Edge Detector". Alvey Vision Conference. 15.* 1988

[Lec] LECAKES, George D.: *OpenCV Basics Camera Calibration*. https://www.patreon.com/glecakes. – [Online; accessed 17-09-2019]

[Low04] LOWE, D.G.: *Distinctive Image Features from Scale-Invariant Keypoints International Journal of Computer Vision (2004) 60: 91. https://doi.org/10.1023/B:VISI.0000029664.99615.94.* 2004

[Mü19] MÜLLER, Stefan: *Augmented Reality Vorlesung „Virtuelle Realität und Augmented Reality"*. 2019

[MV07] MALIS, Ezio ; VARGAS, Manuel: *Deeper understanding of the homography decomposition for vision-based control [Research Report] RR-6303, INRIA. 2007, pp.90. ffinria-00174036v3f.* 2007

[ope] *OpenCV Documantation*. https://docs.opencv.org. – [Online; accessed 17-09-2019]

[pan] *PanoramaStudio 3 Pro*. https://www.tshsoft.de/de/panostudiopro_index. – [Online; accessed 17-09-2019]

[Pau16] PAULUS, Dietrich: *Bildverarbeitung 1 Folie 14 und 15*. 2016

[PBE99] PELEG ; BEN-EZRA: *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149) Stereo panorama with a single camera*

[Pri15] PRIESE, Lutz: *Computer Vision Einführung in die Verarbeitung und Analyse digitaler Bilder Kapitel 12.2*. 2015

[Sch12] SCHLESINGER, D.: *Bildverarbeitung: 3D-Geometrie tu-dresden*. 2012

[shi] *OpenCVSharp Documentation*. github.com/shimat/opencvsharp/wiki. – [Online; accessed 17-09-2019]

[SPDT16] SIMM, Michael ; PROF. DR. THANOS, Solon: *SEHEN – (K)EIN SELBSTVERSTAENDLICHES WUNDER*. https://www.dasgehirn.info/wahrnehmen/sehen. Version: 2016. – [Online; accessed 17-09-2019]

[Sze06] SZELISKI, Richard: *Image Alignment and Stitching: A Tutorial Technical Report MSR-TR-2004-92*. 2006

[uni] *Unity Remote 5*. `https://play.google.com/store/apps/details?id=com.unity3d.genericremote&hl=de`. – [Online; accessed 28-09-2019]

[uni19] *Unity Manual and Documentation*. `https://docs.unity3d.com/Manual/`. Version: 2019. – [Online; accessed 17-09-2019]

[vis] *Visual Studio Community*. `https://visualstudio.microsoft.com/de/`. – [Online; accessed 28-09-2019]

[VL01] VINCENT, Etienne ; LAGANIÈRE, Robert: *Detecting planar homographies in an image pair IEEE Conference Paper · February 2001 School of Information Technology and Engineering University of Ottawa Canada K1N 6N5*. 2001