Bachelorarbeit in Informatik

3D Image Stitching basierend auf Merkmalsextraktion

Alexander Paßberger
Universität Passau
Matrikel Nr. 81345
Betreuer: Prof. Dr. T. Sauer

15. September 2019

Zusammenfassung

Diese Bachelorarbeit legt ihren Schwerpunkt auf das Zusammensetzen von überlappenden 3D-Daten in Form von Point Clouds. Die hierzu nötigen Mechanismen werden von Grund auf in zwei Dimensionen dargestellt, in Form von Panorama Image Stitching, und anschließend auf das dreidimensionale Problem erweitert.Die Arbeit konzentriert sich hierbei auf die Verwendung von auf Merkmalsextraktion basierenden Algorithmen (engl. Feature Based Approaches).

Inhaltsverzeichnis

1 Einleitung **1**

2 Zweidimensionales Panorama Image Stitching **1**
2.1 Allgemeiner Ablauf 1
2.1.1 Bildaufnahme 1
2.1.2 Keypoint Erkennung und Deskriptoren Erzeugung 2
2.1.3 Deskriptoren Matching 2
2.1.4 Match Validierung und Homographie 3
2.1.5 Zusammensetzen 3
2.2 SIFT 3
2.2.1 Scale-space Extrema Detection 3
2.2.2 Keypoint localization 5
2.2.3 Orientation assignment 6
2.2.4 Keypoint descriptor 6
2.3 RANSAC 7
2.4 Implementierung 8
2.4.1 Das Struct ExtendedImage 8
2.4.2 Die Klasse Stitching 9
2.4.3 Die Abstrakte Methode FindFeatures 14
2.4.4 Das Hauptprogramm 15
2.5 Ergebnisse 18

3 Dreidimensionales Image Stitching **23**
3.1 Dreidimensionales SIFT 23
3.2 SHOT 25
3.3 Implementierung 27
3.3.1 Projektorganisation mit CMake 28
3.3.2 Das Struct ExtendedCloud 29
3.3.3 Das Hauptprogramm 29
3.3.4 Die Klasse Stitcher 34
3.4 Konfiguration 46
3.5 Ergebnisse 48
3.5.1 Visualisierung des Ablaufs 48
3.5.2 Überlapp Test 53
3.5.3 Weitere Ergebnisse 56

4 Fazit **58**

Literatur **59**

1 Einleitung

Das Zusammensetzen von zweidimensionalen Panoramas wird bereits lange erforscht. Schon 1975 findet sich im Artikel Computer methods for creating photomosaics von D. Milgram eine Möglichkeit zum Kombinieren von Fotos zu einem Fotomosaik [1]. Ein Merkmalsbasiertes Verfahren wird erstmals 1997 vorgestellt [2], mit der Vorstellung von SIFT 2004 [3] wird diese Verfahrensart zum hauptsächlich verwendeten. Der Algorithmus war erstmals stabil unter Rotation und Skalierung und ermöglichte somit ein zuverlässiges Ergebnis.
Mit der steigenden Rechenleistung von Computern gewannen dreidimensionale immer mehr an Bedeutung. So können mittlerweile mit der Bewegungssteuerungshardware Kinect von Microsoft dreidimensionale Scans von jedermann zu einem erschwinglichen Preis erstellt werden [4]. Auch neue Entwickelungen wie Augmented Reality [5], Virtual Reality [6] oder dreidimensionale Druckverfahren [7] erhöhen das Interesse an dreidimensionalen Daten. Zusätzlich bleiben klassische Anwendungen wie etwa CT-Scans, welche in dieser Arbeit verwendet werden.
Aus Konsequenz ergibt sich ein Interesse am Ausrichten und Zusammensetzen von dreidimensionalen Daten, ähnlich zu den bekannten Panorama Fotos. Diese Arbeit legt ihren Schwerpunkt auf die Implementierung eines solchen Verfahren und implementiert zum Verständnis der Abläufe auch einen zweidimensionalen Image Stitcher. Die Theorie hinter den verwendeten Algorithmen wird jeweils im Voraus dargelegt. Die Implementierungen werden schließlich mit Testdaten verifiziert.

2 Zweidimensionales Panorama Image Stitching

2.1 Allgemeiner Ablauf

Der Ablauf für das Stitching von Bildern basierend auf Merkmalsextraktion lässt sich im Allgemeinen in fünf essentielle Schritte unterteilen. Darüber hinaus gibt es noch weitere Schritte, welche das Ergebnis optimieren. Die in der literatur beschriebene Vorgehensweise und die Anzahl an Schritten schwankt daher. Bei den merkmalsbasierten Verfahren werden besondere Eigenschaften von jedem Bild miteinander verglichen (engl. Features). Hierfür werden zuerst besondere Schlüsselstellen ausfindig gemacht (engl. Keypoints), um anschließend an diesen Stellen einzigartige Deskriptoren zu berechnen [8].

2.1.1 Bildaufnahme

Für die Aufnahme der Ausgangsfotografien des Panoramas gibt es zwei Möglichkeiten: Mit mehreren Aufnahmen durch Bewegung der Kamera oder mit einer durchgängigen Aufnahme durch Rotation der Kamera. Darüber hinaus können die

beiden Varianten auch miteinander kombiniert werden.
In der ersten Möglichkeit werden die einzelnen Fotos aufgenommen, indem die Kamera parallel zur Aufnahmeszene bewegt wird. Dies kann in kleineren Dimensionen etwa durch eine Gleitplatte realisiert werden. Der Nachteil dieser Aufnahmetechnik ist die stets größer werdende Distanz, um welche die Kamera verschoben werden muss, je größer (also weiter weg) das Aufnahmemotiv ist. Außerdem generiert diese Aufnahmeart nicht das Gefühl eines dreidimensionalen Raumes, weil alle Bilder aus der selben Ebene aufgenommen wurden.
Die zweite Möglichkeit ist hingegen sehr einfach durchzuführen und benötigt kein zusätzliches Equipment. Um die gewünschte Szene zu erfassen, wird die Kamera selbst einfach rotiert. Für ein gleichmäßig aufgenommenes Panorama-Foto kann die Kamera auf einem Stativ rotiert werden. [9].

2.1.2 Keypoint Erkennung und Deskriptoren Erzeugung

Im nächsten Schritt werden in jedem Bild Keypoints ausfindig gemacht. Dies sind besondere Punkte in einem Foto, welche leicht in anderen Bildern wieder erkannt werden können. Charakteristische Punkte sind Linien, Ecken, Kanten und andere geometrische Formen. Diese Punkte werden anschließend möglichst einzigartig beschrieben. Mittlerweile gibt es sehr viele Algorithmen , die Schlüsselstellen auffinden und beschreiben, wie zum Beispiel SIFT (Scale Invariant Feature Transform), SURF (Speeded Up Robust Features), FAST (Features from Accelerated Segment Test), Harris Corner Detector oder ORB (Orientated FAST and Rotated BRIEF). Prinzipiell ist die Erkennung von Schlüsselstellen und das Erzeugen von einzigartigen Beschreibungen unabhängig voneinander. Die meisten Algorithmen im zweidimensionalen Bereich bieten jedoch ihren eigenen Deskriptor an. Teilweise handelt es sich auch um zusammengesetzte Verfahren. so ist ORB eine weiterentwickelte Kombination aus der Keypoint Erkennung von FAST und dem Deskriptor BRIEF (Binary Robust Independent Elementary Features). Auf den SIFT-Algorithmus wird im folgenden Kapitel 2.2 aufgrund der Verwendung in der Implementierung noch genauer eingegangen. Der SIFT-Algorithmus ist zwar langsamer als neuere Verfahren, dafür sehr gut erforscht und gilt allgemein als sehr robust aufgrund der hohen Anzahl an Schlüsselstellen [8].

2.1.3 Deskriptoren Matching

Nachdem die Schlüsselstellen der Bilder und deren Deskriptoren vorliegen müssen diese miteinander verglichen und die übereinstimmenden Punkte gefunden werden. Der einfachste Ansatz ist das Brute-Force Matching, bei welchem jeder Punkt mit jedem anderen Punkt verglichen wird. Anschließend wird jeweils der korrespondierende Punkt mit dem geringsten Abstand zurückgegeben. Dieser Ansatz führt

zu einer quadratischen Laufzeit, so dass als schnellere Alternative ein approximierender Algorithmus, basierend auf einen k-d-Baum, verwendet werden kann. Hierdurch verringert sich die Laufzeit auf $O(n \log n)$ [10].

2.1.4 Match Validierung und Homographie

Beim SIFT-Algorithmus wird direkt nach dem Berechnen der Matches ein Ratio-Test durchgeführt, welcher sich in der Originalliteratur auf Lowe zurückführen lässt. Hierfür berechnet man zu jedem Kernpunkt die zwei am besten korrespondierenden Schlüsselstellen im anderen Bild. Als Match genommen wird ein Punkt nur, wenn seine Distanz kleiner als 80% der Distanz des zweitbesten Matches ist [3]. Der Test kann auch für andere Verfahren verwendet werden.
Für die endgültige Validierung der gefundenen Matches wird Random Sample Consensus (RANSAC) verwendet. RANSAC berechnet auf Grundlage des gefundenen Modells direkt eine Homographie. Diese entspricht der nötigen Verzerrung des Bildes, damit es sich in die Szenerie des anderen einfügt [11]. Auf die genaue Funktionsweise von RANSAC wird im späteren Kapitel 2.3 noch eingegangen.

2.1.5 Zusammensetzen

Als letzten Schritt müssen die Bilder auf eine gemeinsame Ebene projiziert werden. Weil Homagraphien transitiv sind, kann dies inkrementell geschehen. Die Bilder können nach dem Verzerren einfach zusammen kopiert werden.

2.2 SIFT

SIFT (Scale Invariant Feature Transform) wurde 2004 von David G. Lowe in seinem Artikel „Distinctive Image Features from Scale-Invariant Keypoints“ vorgestellt [3]. Es handelt sich dabei um einen Algorithmus zur Extraktion und Beschreibung von Bildmerkmalen. Diese Bildpunkte sind sowohl unter Skalierung, Rotation und teilweise bei Veränderung des Standpunktes und der Beleuchtung invariant. Der SIFT-Algorithmus wird in der Implementierung des 2D-Stitchers sowohl zur Erkennung von Keypoints, als auch zur Beschreibung dieser als Deskriptor verwendet. Die Extraktion und Beschreibung erfolgt in vier Schritten, welche im Folgenden zusammengefasst dargestellt werden.

2.2.1 Scale-space Extrema Detection

Für die Erkennung der Schlüsselstellen definieren wir zuerst $L(\mathbb{R})$ als Menge aller reellwertigen Funktionen und darauf die Faltung entsprechend [12]:

$$f * g := \int_{\mathbb{R}} f(\cdot - t) g(t) dt, \quad * : L(\mathbb{R}) \times L(\mathbb{R}) \to L(\mathbb{R}) \tag{1}$$

Als ersten Schritt zur Erkennung von potentiellen Kandidaten werden Positionen und Skalen identifiziert, welche unter verschiedenen Betrachtungswinkel des selben Objektes wiederholt zugewiesen werden können. Dies wird erreicht, indem man nach stabilen Merkmalen in allen möglichen Skalierungen sucht. Hierzu wird die kontinuierliche Skalierungsfunktion Scale-Space verwendet. Das Scale-Space eines Bildes wird hierbei definiert als die Faltung des Bildes mit einem Skalierungsabhängigen Gaußfilter:

$$L(x, y, \sigma) = (G(\sigma) * I)(x, y), \tag{2}$$

mit

$$G(x, y, \sigma) = \frac{1}{2\Pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{3}$$

Wobei I(x,y) dem Eingabebild entspricht und * dem Faltungsoperator. Zum Extrahieren der Keypoints wird schließlich das Scale-Space Extrema der Differnz des Gaußfilters gefaltet mit dem Bild verwendet:

$$\begin{aligned} D(x, y, \sigma) &= (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) \\ &= L(x, y, k\sigma) - L(x, y, \sigma) \end{aligned} \tag{4}$$

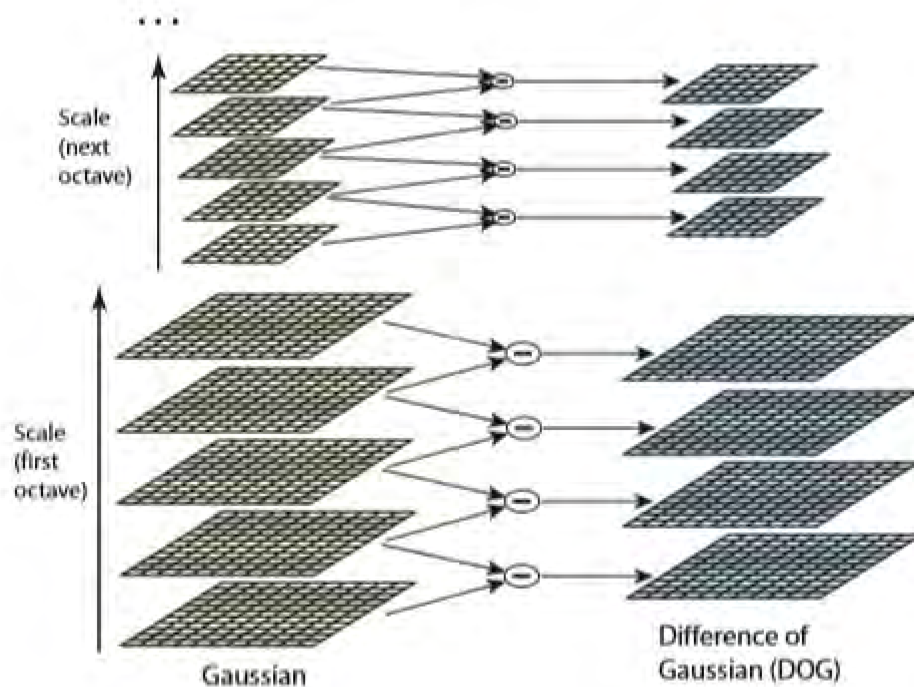


Abbildung 1: Veranschaulichung der oktavenweisen Berechnung der Differenz des Gaußfilters. Links sind die Orginalbilder jeweils um den Skalierungfaktor nach oben verändert. Rechts sind die Differenzbilder[3].

Dies lässt sich effizient umzusetzen, indem das Ursprungsbild inkrementell mit dem Gaußfilter gefaltet wird. Hierbei entstehen Bilder, welche sich um den Faktor k im Scale-Space unterscheiden. Der Scale-Space wird mit jeder Verdoppelung von σ in eine Oktave unterteilt, welche wiederum in s Intervallen mit $k = 2^{1/s}$ aufgeteilt wird. Um eine gesamte Oktave von Differenzen zu erhalten müssen somit s + 3 Bilder erstellt werden. Die benachbarten Bilder werden anschließend substrahiert, um die Bilder mit der Differenz des Gaußfilters zu erhalten. Nun wird von jeder Reihe und Spalte jedes zweite Pixel genommen, wodurch man die nächste Oktave erhält und der Vorgang wiederholt sich (siehe Abbildung 1).
Das lokale Maximum und Minimum erhält man schließlich, indem man jeden Punkt mit seinen acht direkten Nachbarn und seinen neun Nachbarm im Scale-Space darüber und darunter vergleicht (siehe Abbildung 2). Als potentieller Keypoint gewählt werden nur Punkte, welche maximal oder minimal unter allen verglichenen Punkten sind.

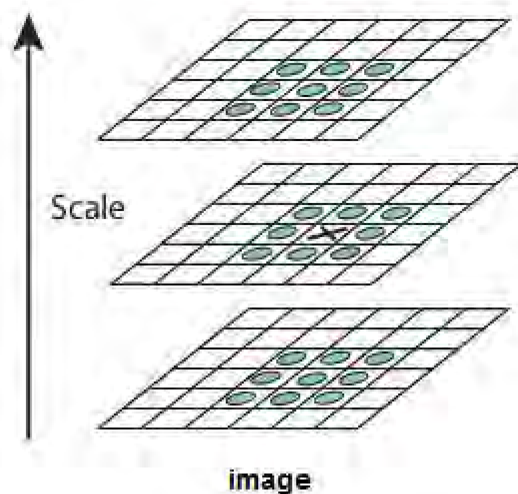


Abbildung 2: Darstellung der Erkennung eines lokalen Minimum beziehungsweise Maximum. Das x in der Mitte entspricht den aktuell vergleichenden Pixel [3].

2.2.2 Keypoint localization

Die gefundenen Schlüsselstellen werden anschließend weiter aussortiert. Als Erstes wird die Taylor-Expansion der Scale-Space-Funktion verwendet, welche zum Stichprobenpunkt verschoben wird:

$$D(x) = D + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x. \tag{5}$$

Wobei $x = (x, y, \sigma)^T$ der Offset vom Stichprobenpunkt ist. Ist der Betrag von D(x) am Extrema x̂ kleiner als 0.03, besitzt der Punkt einen geringen Kontrast und ist daher instabil. Weil die Differenz der Gaußfunktion besonders auf Eckpunkte reagiert, werden von diesen die ungenau bestimmten aussortiert. Diese besitzen eine große Hauptkrümmung an den Ecken, aber eine geringe an der Senkrechten.

2.2.3 Orientation assignment

Nachdem geeignete Punkte bestimmt sind, wird jedem Punkt eine Orientierung zugewiesen. Hierdurch erreicht SIFT die Robustheit bei Rotationen. Aus dem Bild, welches am nächsten zur Skalierung des Keypoints ist, wird für jeden Punkt $L(x, y)$ die Magnitude der Steigung $m(x, y)$ und die Orientierung $\theta(x, y)$ berechnet.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \qquad (6)$$

$$\theta(x,y) = \tan^{-1}(L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y)) \qquad (7)$$

Aus den Punkten innerhalb der Region der Schlüsselstelle wird ein 36-wertiges Histogram erstellt, welches den gesamten 360 Grad entspricht. Jeder Wert wird nach der Magnitude der Steigung und einem Gaußischen Kreisfenster mit σ gleich dem 1,5-fachen der Skalierung gewichtet. Der höchste Wert sowie jeder über 80% wird zur Berechnung der Orientierung verwendet. Hierdurch entstehen auch mehrfache Keypoints am gleichen Ort mit gleicher Skalierung, aber unterschiedlicher Orientierung. Diese tragen wesentlich zur Stabilität von SIFT bei.

2.2.4 Keypoint descriptor

Aus den Steigungen und Orientierungen wird schließlich der Deskriptor berechnet. Hierfür wird ein 16x16 Set mit der Steigung als Länge und der Orientierung als Richtung erstellt. Aus diesen ergibt sich, gewichtet mit einem Gaußischen Kreisfenster, ein 4x4 Deskriptor mit jeweils acht Orientierungen. Diese Werte werden zusätzlich interpoliert. Ebenso wird der Vektor auf Einheitslänge gebracht, um robuster gegen schwankende Lichtverhältnisse zu werden. Insgesamt ergibt sich daraus ein Vektor mit 128 Werten.

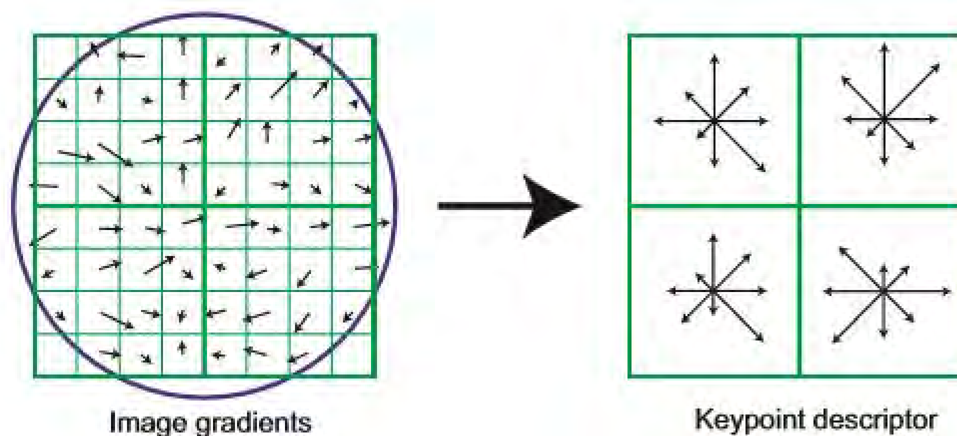


Abbildung 3: Darstellung der Berechnung eines SIFT Deskriptors. Die Abbildung zeigt einen vereinfachten 2x2 Deskriptor berechnet aus einem 8x8 Array. Jeder Quadrant wird zu einem Feld mit acht Orientierungen und Längen zusammengefasst [3].

2.3 RANSAC

Random Sample Consensus (RANSAC) wurde 1981 von Martin A. Fischler und Robert C. Bolles vorgestellt [13]. Es handelt sich um einen Algorithmus zur Modellerkennung anhand von empirischen Versuchsdaten. Der Algorithmus wurde speziell für die Bildverarbeitung mit Daten aus merkmalsbasierten Verfahren entwickelt. Im Gegensatz zu gewöhnlichen Glättungsverfahren werden zur Bestimmung des gesuchten Modells so wenig Punkte wie nötig verwendet. Der Ablauf von RANSAC ist wie folgt:

1. Aus einem Set P von Datenpunkten wird zufällig eine Teilmenge S1 mit n Datenpunkten genommen. Dabei entspricht n der minimalen Menge an Punkten, welche zur Lösung des gesuchten Modells nötig ist. Die Anzahl an Datenpunkten in P muss größer als n sein.

2. Aus der gewählten Teilmenge wird ein Modell M1 erstellt. Anschließend wird die Teilmenge S1* gebildet, indem zu S1 alle Punkte innerhalb einer definierten Fehlertoleranz hinzugefügt werden. S1* wird als Übereinstimmungsmenge bezeichnet.

3. Wenn die Anzahl an Datenpunkten in S1* größer als ein gewählter Grenzwert t ist, wird aus S1* ein neues Modell M1* berechnet. Wenn die Anzahl kleiner

als t ist, wird zufällig eine neue Teilmenge S2 ausgewählt und der Prozess wird wiederholt.

Wird nach einer bestimmten Anzahl an Wiederholungen keine Menge mit mehr als t Datenpunkten gefunden, wird entweder das größte gefundene Modell als Lösung genommen oder der Prozess in einem Fehlerzustand geendet.
Das Verfahren kann mit Verwendung eines deterministischen Selektionsverfahrens beim Auswählen der Teilmengen verbessert werden. Außerdem können die mit dem Modell berechneten konsistenten Punkte aus P erneut zur Übereinstimmungsmenge hinzugefügt werden. Aus der so vergrößerten Menge kann wieder eine neues Modell errechnet werden.

2.4 Implementierung

Zum Verständnis der Abläufe wird ein single-row Panorama Stitcher implementiert [9]. Dieser entspricht dem von Smartphones und Kameras bekannten Panorama-Modus. Zusammengesetzt wird ein von links nach rechts aufgenommenes Panorama, etwa durch Rotation.
Die Implementierung erfolgt in C++ unter Verwendung der OpenCV 4.1.1 Bibliothek inklusive Contribution Module. Das Programm ist als Visual Studio 2019 Projekt angelegt und besteht aus insgesamt vier Klassen beziehungsweise Structs und der Main.cpp Datei.

2.4.1 Das Struct ExtendedImage

```
1 struct ExtendedImage
2 {
3   Mat image;
4   vector<KeyPoint> keypoints;
5   Mat descriptors;
6   Mat homography;
7
8   ExtendedImage(Mat& image);
9 };
```

Listing 1: Die Struct-Definition von ExtendedImage

Das ExtendedImage dient der Speicherung von Bildern inklusive ihrer zugehörigen Schlüsselstellen, Deskriptoren und der inversen Homographie zum vorherigen Bild. Das ExtendedImage besitzt nur einen Konstruktor, bei welchem das Originalbild übergeben wird.

2.4.2 Die Klasse Stitching

```cpp
class Stitching
{
private:
  vector<DMatch> FindFeatureMatches(ExtendedImage& image1,
                                    ExtendedImage& image2);
  Mat FindHomography(ExtendedImage& image1,
                     ExtendedImage& image2,
                     vector<DMatch>& matches);
  Mat CombineImages();
  Mat CombineImage(const Mat& image1, const Mat& image2,
                   const Mat homography);
  Mat CropImage(const Mat& image);
protected:
  vector<ExtendedImage> images;
  virtual void FindFeatures(ExtendedImage& image) = 0;
public:
  Mat StitchImages(const vector<Mat> images);
  static void DrawKeyPoints(const Mat& img,
                            const vector<KeyPoint>& keypoints);
  static void DrawFeatureMatches(const Mat& img1,
                                 const vector<KeyPoint>& keypoints1,
                                 const Mat& img2,
                                 const vector<KeyPoint>& keypoints2,
                                 const vector<DMatch> matches);
};
```

Listing 2: Die Klassendeklaration von Stitching

In der Klasse Stitching geschieht die Hauptarbeit des Verfahrens: das Zusammensetzen der Bilder. Die Klasse ist abstrakt gestaltet, um einen leichten Wechsel des verwendeten Algorithmus zur Bestimmung von Keypoints und Deskriptoren zu ermöglichen. Als Schnittstelle dient die Methode StitchImages, welche einen Vektor von Bildern als Parameter entgegen nimmt und das fertige Panoramabild zurück gibt.

```cpp
Mat Stitching::StitchImages(vector<Mat> images)
{
  //calculate keypoints and descriptors for each Image
  cout << "Calculating Keypoints and Descriptors" << endl;
  for (int i = 0; i < images.size(); ++i)
  {
    this->images.push_back(ExtendedImage(images[i]));
    FindFeatures(this->images[i]);

    //Show Keypoints
    //DrawKeyPoints(images[i], this->images[i].keypoints);
  }

  cout << "Calculating Matches and Homographies" << endl;
  for (int i = 1; i < this->images.size(); ++i)
  {
    vector<DMatch> matches = FindFeatureMatches(
        this->images[i - 1], this->images[i]);
    Mat homography = FindHomography(
        this->images[i - 1], this -> images[i], matches);
        this->images[i].homography = homography.inv();
  }
  cout << "Combining Images" << endl;
  Mat StitchingResult = CombineImages();
  return CropImage(StitchingResult);
}
```

Listing 3: Die Methode StitchImages

Die Methode StitchImages stellt die Hauptschleife des Programms dar und durchläuft die aus dem allgemeinen Ablauf bekannten Schritte. In der ersten For-Schleife wird jedes Bild in ein ExtendedImage verkapselt. Anschließend werden für jedes Bild Schlüsselstellen und Deskriptoren in der Methode FindFeatures berechnet. Der auskommentierte Code dient lediglich zur Visualisieren der Keypoints und wird nur für den Abschnitt 2.5 Ergebnisse verwendet. Die zweite For-Schleife wird zur Berechnung der paarweisen Matches zwischen den Schlüsselstellen und der daraus resultierenden Homographie verwendet. Weil es einfacher ist die Bilder zum Schluss rückwärts zusammen zu setzen, da hierbei der linke Teil immer mit dem nächsten Bild ersetzt werden kann, wird jeweils die inverse der Homographie im zweiten Bild des Vergleichs gespeichert. Somit kann die im ExtendedImage gespeicherte Homographie direkt an der Verwendungsstelle des Bildes benutzt werden. Wenn alle Berechnungen fertig sind, werden die Ergebnisse zusammengesetzt. Zum Schluss wird der schwarze Rand aufgrund der Überlappung der Bilder entfernt und das fertige Panorama zurück gegeben.
Nachdem die Methode zur Berechnung der Schlüsselstellen und Deskriptoren aus-

gelagert ist, wird diese fürs Erste überspringen und es wird mit der Berechnung der Matches in der Methode FindFeatureMatches begonnen.

```
vector<DMatch> Stitching::FindFeatureMatches(ExtendedImage& image1,
                                              ExtendedImage& image2)
{
  Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create(
                                     DescriptorMatcher::FLANNBASED);
  int k = 2;
  vector<vector<DMatch> > matches;
  matcher->knnMatch(image1.descriptors, image2.descriptors,
                    matches, k);

  //Ratio test from Lowe
  vector<DMatch> good_matches;
  const double ratio = 0.8; //Lowe 2004
  for (int i = 0; i < matches.size(); ++i)
  {
    if (matches[i][0].distance < ratio * matches[i][1].distance)
        {
          good_matches.push_back(matches[i][0]);
    }
  }
  return good_matches;
}
```

Listing 4: Die Methode FindFeatureMatches ohne dem Code zur Visualisierung des Übereinstimmungen

Für das Bestimmen von Matches wird ein FLANNBASED Descriptor Matcher verwendet. FLANN steht für Fast Library for Apprroximate Nearest Neighbors [10]. Es handelt sich um einen approximierenden k-d-Baum-Algorithmus. Darüber hinaus wird die Anzahl an gesuchten Nachbarn für jeden Punkt auf k = 2 gesetzt. Dies ermöglicht die anschließende Durchführung des Ratio Tests von Lowe. Der Matcher liefert einen Vektor von Vektoren zurück, der jeweils die zwei nächsten Nachbarn im inneren Vektor beinhaltet. Dieser wird nun mit einer For-Schleife durchlaufen. Matches welche den Test bestehen werden gespeichert. Die übrigen Matches werden zum Schluss zurück gegeben. Der auskommentierte Code zum Schluss fungiert wieder nur zur Visualisierung für das Ergebnis.

```
1  Mat Stitching::FindHomography(ExtendedImage& image1,
2                                ExtendedImage& image2,
3                                vector<DMatch>& matches)
4  {
5    vector<Point2f> current;
6    vector<Point2f> other;
7    for (int y = 0; y < matches.size(); ++y)
8    {
9      current.push_back(image1.keypoints[matches[y].queryIdx].pt);
10         other.push_back(image2.keypoints[matches[y].trainIdx].pt);
11   }
12
13   Mat homography, mask;
14   homography = findHomography(current, other, RANSAC, 3.0, mask);
15   /*Inliners and Outliners for visualization
16   vector<DMatch> inliners, outliners;
17   for (int y = 0; y < matches.size(); ++y)
18   {
19     int d = (int)mask.at<uchar>(y, 0);
20         if ((int)mask.at<uchar>(y, 0) == 1)
21         {
22       inliners.push_back(matches[y]);
23         }
24         else
25         {
26           outliners.push_back(matches[y]);
27         }
28   }
29   */
30   return homography;
31 }
```

Listing 5: Die Methode FindHomography

Nachdem passende Matches gefunden sind, wird mittels RANSAC eine bestmögliche Homographie berechnet. Ein DMatch besteht aus einem queryIdx und einen trainIdx, welcher jeweils den Index im Vektor der Schlüsselstellen des gefundenen Matches angibt. Zur Verwendung von RANSAC werden mittels einer For-Schleife die an einem Match beteiligten Keypoints extrahiert. Die Maske wird nur zur Berechnung der Inliner und Outliner benötigt, also der korrekten und inkorrekten Matches nach RANSAC. Diese werden erneut lediglich zur Visualisierung benötigt. Die Berechnung ist daher auskommentiert. Die Homographie hingegen wird direkt von der in OpenCV enthaltenen findHomography-Methode zurückgegeben. RANSAC bezeichnet hierbei den zu verwendenen Algorithmus, wobei 3.0 den Threshold zur Wertung als Inliner entspricht. Der Wert ist der Standardwert der Methode und muss aufgrund der Verwendung einer Maske explizit angegeben werden.

```cpp
Mat Stitching::CombineImages()
{
  Mat lastImage = images[images.size() - 1].image;
  for (int i = (int) images.size() - 1; i > 0; --i)
  {
    Mat homography = images[i].homography;
    Mat beforeImage = images[i - 1].image;
    lastImage = CombineImage(beforeImage, lastImage, homography);
  }
  return lastImage;
}
```

Listing 6: Die Methode CombineImages

Die Methode CombineImages durchläuft die Bilder rückwärts und koordiniert die Reihenfolge der zu stitchenden Bildern. Das eigentliche Zusammensetzen erfolgt in der Methode CombineImage. Das resultierende Bild wird als Erstes mit dem letzten Bild initialisiert. Weil jeweils beim zweiten Bild die inverse Homographie gespeichert ist und die Homographien transitiv sind, kann jeweils die Homographie des aktuellen Bildes verwendet werden. Diese wird gemeinsam mit dem bisherigen Ergebnis und dem nächsten, beziehungsweise vorherigen Bild, an die CombineImage-Methode übergeben. Zum Schluss wird das fertige Ergebnis zurück gegeben.

```cpp
Mat Stitching::CombineImage(const Mat& image1,
                            const Mat& image2,
                            const Mat homography)
{
  Mat warpedImage;
  warpPerspective(image2, warpedImage, homography,
                  Size(image1.cols + image2.cols, image1.rows));
  Mat half(warpedImage, cv::Rect(0, 0, image1.cols, image1.rows));
  image1.copyTo(half);
  return warpedImage;
}
```

Listing 7: Die Methode CombineImage

In der Methode CombineImage geschieht das eigentliche Zusammensetzen der Bilder. Die Homographie wird mittels warpPerspective auf das zweite Bild(image2) angewendet und in warpedImage gespeichert. Um sicher zu sein, dass das Bild nicht abgeschnitten ist, wird es mit der Breite der Summe von beiden Bildern initialisiert. Nachdem sich die Bilder überlappen, bleibt auf der rechten Seite ein immer größer werdender schwarzer Rand übrig, welcher erst zum Schluss entfernt wird. Mit dem Mat half wird eine Referenz auf die linke Hälfte des Bildes erzeugt. Diese Hälfte wird durch das erste Bild (image1) ersetzt. Somit ergibt sich das zusammengesetzte Bild aus dem ersten Bild auf der linken Seite und dem verzerrten

zweiten Bild, dass am Rand des ersten Bildes anschließt. Das zusammengesetzte Bild wird dann zurück gegeben.

```
1  Mat Stitching::CropImage(const Mat& image)
2  {
3    Mat grayscale;
4    cvtColor(image, grayscale, COLOR_BGR2GRAY);
5    threshold(grayscale, grayscale, 0, 255, THRESH_BINARY);
6  
7    int columnToCrop = image.cols - 1;
8    bool hasBackground = true;
9    while (hasBackground)
10   {
11     hasBackground = false;
12     for (int y = 0; y < grayscale.rows; y++)
13     {
14       char x = grayscale.at<char>(y, columnToCrop);
15       if (grayscale.at<char>(y, columnToCrop) == 0)
16       {
17         hasBackground = true;
18         columnToCrop--;
19         break;
20       }
21     }
22   }
23   Rect resultingCrop(0, 0, columnToCrop, image.rows);
24   return image(resultingCrop);
25 }
```

Listing 8: Die Methode CropImage

Als letzen Schritt wird der schwarze Rand auf der rechten Seite abgeschnitten. Hierfür wird das Bild zuerst mit cvtColor in ein Bild mit Grauwerten konvertiert. Die Spalte, ab welcher das Bild abgeschnitten werden soll (columnToCrop), wird auf die letzte Spalte initialisiert. Anschließend läuft eine While-Schleife, bis sich die letzte Spalte nicht mehr im schwarzen Rand befindet. Hierzu wird für jeden y-Wert an der aktuellen x-Position (der Position columnToCrop) der Pixel-Wert angeschaut. Ist ein y-Wert null, also schwarz, läuft die while Schleife weiter. Wurde die entsprechende Stelle zum Abschneiden gefunden, wird der entsprechende Bereich aus dem Eingabebild gewählt und das fertige Panorama zurück gegeben.

2.4.3 Die Abstrakte Methode FindFeatures

Wie bereits beschrieben, ist die Methode FindFeatures in der Klasse Stitching nur abstrakt definiert. Das eigentliche Stitching erfolgt mit einer von Stitching abgeleiteten Klasse. Verwendet wird hierfür die Klasse Sift, welche nur diese eine Methode

überschreibt.

```
void Sift::FindFeatures(ExtendedImage& image)
{
  Ptr<SiftFeatureDetector> detector = SiftFeatureDetector::create();
  detector->detect(image.image, image.keypoints);
  detector->compute(image.image, image.keypoints,
                    image.descriptors);
}
```

Listing 9: Methodendefinition von FindFeatures in der Klasse Sift

Der Code hierfür ist trivial, da SIFT bereits in OpenCV implementiert ist. Es muss nur ein SiftFeatureDetector erstellt werden. Mit diesem sucht man anschließend mittels detect nach Schlüsselstellen und berechnet daraus mit compute die entsprechenden Deskriptoren.
OpenCV speichert Schlüsselstellen in einem Vektor von Typ Keypoint und Deskriptoren als Mat ab, unabhängig vom verwendeten Algorithmus. Damit ergibt sich auch der Nutzen die Methode abstrakt zu halten. Im Programm ist zusätzlich die Klasse Surf enthalten, welche ebenfalls verwendet werden kann. Für näheres zu SURF sei auf die Originalliteratur verwiesen [14].

```
void Surf::FindFeatures(ExtendedImage& image)
{
  Ptr<SURF> detector = SURF::create();
  detector->detect(image.image, image.keypoints);
  detector->compute(image.image, image.keypoints, image.descriptors);
}
```

Listing 10: Methodendefinition von FindFeatures in der Klasse Surf

Der Code unterscheidet sich in diesem Fall nur in der Art des verwendeten Detektors. Mit einer Länge von drei Zeilen ist der Code sehr kompakt. Das Auswechseln oder Hinzufügen eines anderen Algorithmus ist somit schnell und einfach umsetzbar.

2.4.4 Das Hauptprogramm

Das Hauptprogramm ist in der Datei Main.cpp definiert. Sie besteht aus einer Methode zum Einlesen eines Bildes und der main Methode.

```
Mat ReadFile(char* input)
{
  Mat img = imread(input, IMREAD_COLOR); // Read the file

  if (img.empty()) // Check for invalid input
  {
    cout << "Could not open or find the image" << std::endl;
    throw "Invalid image";
  }
  return img;
}
```

Listing 11: Methodendefinition von ReadFile

Die Methode ReadFile benutzt zum Einlesen die OpenCV-Funktion imread. Diese Funktion kann mit den meisten gängigen Bildformaten umgehen und liefert ein Objekt vom Typ Mat. Mat ist der Standardtyp in OpenCV und stellt ein Bild als Matrix dar. Hierdurch kann man mathematische Operation auf das Bild anwenden. Die Funktion kann ein Bild zusätzlich auf verschiedene Arten einlesen, dabei wird der Standard Farbenwert IMREAD_COLOR verwendet. Dies entspricht einem RGB Bild, wobei OpenCV die Werte in der Reihenfolge BGR speichert. Auch das Einlesen in Grauwerten, reduzierten Farben oder ähnlichem ist möglich. Anschließend wird überprüft, ob das erzeugte Objekt Werte enthält. Falls das Bild ungültig ist, wird eine Exception geworfen mit der Meldung „Invalid image“.

```
int main(int argc, char** argv)
{
  if (argc < 5)
  {
    cout << " Not enough arguments" << endl;
    return -1;
  }

  //read feature type
  string featureType = argv[1];
  for (int i = 0; i < featureType.length(); i++)
    featureType[i] = toupper(featureType[i]);

  //read images and stitch
  cout << "Reading Images..." << endl;
  vector<Mat> images;
  for (int i = 3; i < argc; i++)
  {
    images.push_back(ReadFile(argv[i]));
  }

  Mat result;
  if (featureType.compare(USE_SIFT) == 0)
  {
    cout << "Begin stitching images with SIFT" << endl;
    Sift stitcher;
    result = stitcher.StitchImages(images);
  } else if (featureType.compare(USE_SURF) == 0)
  {
    cout << "Begin stitching images with SURF" << endl;
    Surf stitcher;
    result = stitcher.StitchImages(images);
  }
  imwrite(argv[2], result);
  return 0;
}
```

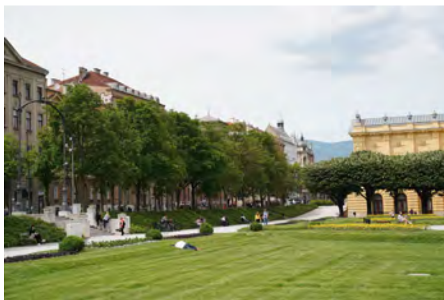
Listing 12: Die main Methode

Die Main-Methode erhält als Parameter zuerst den zu verwendenden Algorithmus (aktuell unterstützt sind SIFT und SURF). Als zweiten Parameter wird der Speicherort des fertigen Panoramas angegeben. Danach folgen eine beliebige Anzahl an Ausgangsbildern. Die Methode überprüft zu Beginn ob genügend Parameter angegeben wurden, um mindestens zwei Bilder zusammen zu fügen. Anschließend wird der zu verwendende Algorithmus ausgelesen und in Großbuchstaben umgewandelt. Die Schreibweise des zu verwendenden Verfahrens ist daher beliebig. Die übermittelten Bilder werden mit einer For-Schleife und der zuvor vorgestellten Methode

ReadFile eingelesen. Anschließend wird ein Objekt des richtigen Stitchers erstellt und das Ergebnis mittels StitchImages berechnet. Als Letztes wird das Resultat mit der OpenCV Funktion imwrite als JPEG-Datei im angegebenen Speicherort erzeugt.

2.5 Ergebnisse

Für die Betrachtung der Ergebnisse des implementierten Image Stitcher wird ein Testdurchlauf mit vier Bildern gemacht. Die Bilder zeigen den Kunstpavillon im König-Tomislav-Platz in Zagreb, Kroatien. Sie wurden am 25.05.2019 mit einer Sony Alpha 7 III aufgenommen. Die Fotos wurden frei Hand durch Rotation aufgenommen.



(a)

(b)

(c)

(d)

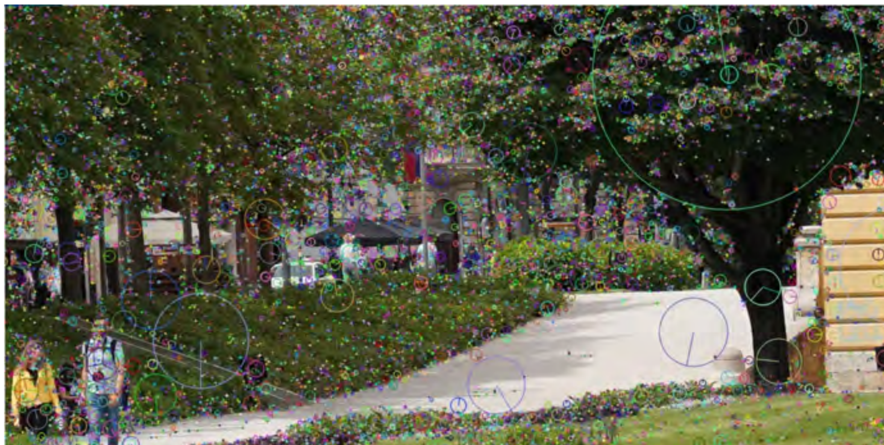
Abbildung 4: Die vier Ausgangsbilder vom König-Tomislav-Platz, aufgenommen am 25.05.2019 in Zagreb, Kroatien.

(a)

(b)

(c)

(d)



(e)

Abbildung 5: Die von SIFT gefundenen Schlüsselstellen in den Ausgangsbildern. Bild (e) zeigt einen Auschnitt des ersten Fotos mit 30% Zoom. Die Größe und Orientierung der Keypoints lassen sich nun deutlich erkennen.

Um den Ablauf des Stitchings zu betrachten, wurden in diesem Durchlauf zusätzlich Zwischenergebnisse mittels der OpenCV Funktionen DrawKeyPoints, DrawFeatureMatches und imwrite gespeichert. Die nächste Abbildung zeigt die von SIFT gefundenen Schlüsselstellen in den einzelnen Bildern. Die Keypoints werden

als Kreis mit der Größe des Keypoints als Durchmesser und der Orientierung gezeichnet [15]. Die Kreise mit Orientierung lassen sich im gesamten Bild großteils nur als Punkt erkennen. Erst wenn man in das Bild hineinzoomt, werden sie deutlich. Es lässt sich ebenfalls erkennen, dass die Punkte in allen Bildern gleichmäßig über die gesamte Größe verteilt sind. Zudem sind die Punkte sehr zahlreich.

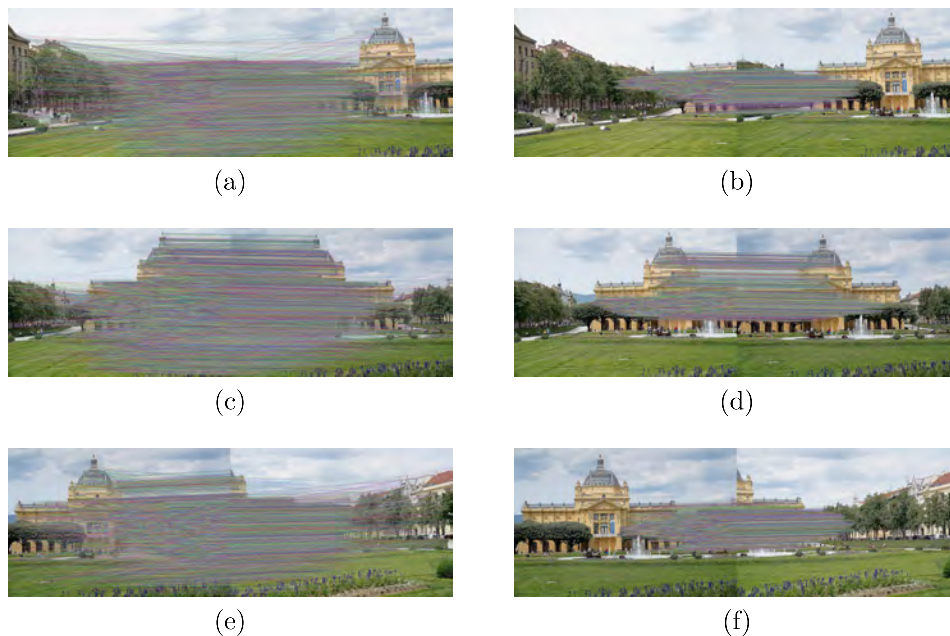
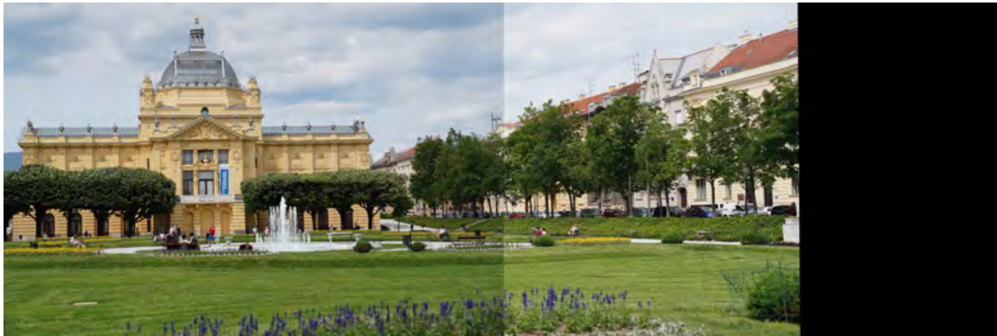


Abbildung 6: Die gefundenen Matches zwischen den Bildern jeweils nach dem Ratio-Test und nach der Verwendung von RANSAC.

Abbildung 6 zeigt jeweils die gefundenen Matches zwischen den Bildern vor und nach der Verwendung von RANSAC. Die Bilder (a), (c) und (e) zeigen die Matches nach dem Aussortieren mit dem Ratio-Test. Sie verteilen sich über die gesamte Größe beider Bilder und verlaufen in alle Richtungen. Sie beinhalten ebenso Matches in der nicht überlappenden Region. Die Bilder (b), (d) und f zeigen die von RANSAC berechneten Inliner Matches. Es lässt sich deutlich erkennen, dass diese beinahe ausschließlich in der überlappenden Szenerie liegen. Des Weiteren liegen fast nur noch sehr gerade Verbindungslinien vor.
Zuletzt wird das iterative Zusammensetzen der Bilder veranschaulicht. In diesem Fall bildet sich der schwarze Rand nur in den Zwischenschritten, das letzte Bild enthält keinen. Das Endergebnis entspricht somit auch dem letzten Zwischenergebnis und wird daher kein weiteres mal abgebildet. Das Ergebnis kann insgesamt als gut bezeichnet werden, da die Bilder richtig zusammengesetzt wurden und ein zusammenpassendes Panoramabild ist entstanden.

Die Unterschiede in der Helligkeit der einzelnen Bilder sind erkennbar. Dies war jedoch von Beginn an zu erwarten. Um die Differenz in der Helligkeit auszugleichen, müsste zum Schluss noch eine Variante des Blendings verwendet werden. Verschiedene Blending-Methoden finden sich etwa in [9], [11] oder in [16].
Das finale Panoramabild wirkt ebenfalls etwas schief, was jedoch auf die leicht schrägen Aufnahmen der Ausgangsbilder zurückzuführen ist. Eine automatische Lösung zum Ausrichten findet sich im Artikel Automatic Panoramic Image Stitching using Invariant Features [16]. Für ein genaueres Ergebnis an den Kanten der Bilder kann zusätzlich nach der Kamerarotation aufgelöst werden. Die Bilder werden dann mittels Bundle Adjustment zusammengesetzt. Genaueres hierzu findet sich ebenfalls im Artikel Automatic Panoramic Image Stitchingusing Invariant Features [16]. Diese Verfahren sind speziell für den zweidimensionalen Fall entworfen. Nachdem der Schwerpunkt dieser Arbeit im dreidimensionalen Teil liegt, wird hier nicht näher darauf eingegangen.



(a)

(b)

(c)

Abbildung 7: Die Zwischenergebnisse nach dem Zusammensetzen der Ausgangsaufnahmen. Das letzte Zwischenergebnis entspricht in diesem Fall dem Endresultat.

3 Dreidimensionales Image Stitching

Das Verfahren für den dreidimensionalen Raum bleibt im wesentlichen gleich. Zuerst werden Schlüsselstellen erkannt und ein Deskriptor für diese erstellt. Die Deskriptoren werden miteinander verglichen und die Matches anschließend verifiziert. Aus den verifizierten Matches wird schließlich die Transformationsmatrix berechnet, mit welchen die Bilder zusammengesetzt werden. Anstelle von Image Stitching wird der Prozess im Dreidimensionalen auch als Point Cloud Registrierung bezeichnet [17].

3.1 Dreidimensionales SIFT

Für die Schlüsselstellenerkennung im dreidimensionalen Raum wird ein adaptierter SIFT Detektor verwendet. Der Detektor wurde 2007 als Bestandteil des THRIFT Algorithmus vorgestellt [18]. Die Erkennung unterteilt sich in drei Schritte:

1. Als Erstes wird eine Dichtekarte der Daten erstellt. Hierfür wird eine Dichtefunktion $f(x, y, z)$ aproximiert. Wir bezeichnen mit $n(B)$ die Anzahl an Datenpunkten in der Region $B \subseteq \mathbb{R}^3$. Die Approximation $f(B)$ in jeder Region erfolgt schließlich durch:

$$\int_B f(x)dx = n(B) \tag{8}$$

Für die Konstruktion von f werden gleichgroße Boxen $B = \{B_{ijk}\}_{(i,j,k)\in I \subset \mathbb{Z}^3}$ erzeugt und gleichmäßig im Raum verteilt:

$$\begin{aligned} B_{ijk} = \{(x, y, z) \in \mathbb{R}^3 : \alpha i \leq x < \alpha(i + 1), \\ \beta j \leq y < \beta(j + 1), \\ \gamma k \leq z < \gamma(k + 1)\} \end{aligned} \tag{9}$$

Die Dichtefunktion f ergibt sich nun als Summe von delta Funktionen:

$$f(x, y, z) = \sum_{ijk\in I} D(i, j, k)\delta(x - C(i, j, k)_x, y - C(i, j, k)_y, z - C(i, j, k)_z) \tag{10}$$

mit $C(i, j, k) = \overline{B}_{ijk}$ als Zentrum der Box B_{ijk} und der normalisierten Dichtekarte:

$$D(i, j, k) = \frac{n(B_{ijk})}{\underset{(i,j,k)\in I}{\operatorname{argmax}}\{n(B_{ijk})\}} \tag{11}$$

2. Die erhaltene Dichtekarte wird wie das Bild im zweidimensionalen verwendet und mit einer Serie an dreidimensionalen Gaußfilter gefaltet. Die Skalierungen erhalten den Wert $\sigma = k\sigma_1$, mit σ_1 als Wert der darunter liegenden Schicht. Aus Effizienzgründen wird die Dichtekarte bei erreichen des Skalierungswertes zwei um den Faktor zwei herunterskaliert. Ebenfalls wird die Varianz σ des Gaußfilters um den Faktor zwei reduziert. Wir definieren nun die Scale-Space Funktion um z erweitert:

$$S(x, y, z, \sigma) = (D * g(\sigma))(x, y, z) \tag{12}$$

mit

$$g(x, y, z, \sigma) = \exp\left(\frac{-x^2 - y^2 - z^2}{2\sigma^2}\right). \tag{13}$$

3. Als Schlüsselpunkte gewählt werden Punkte, welche in allen drei Hauptkrümmungen hohe Werte aufweisen. Diese Punkte bilden signifikante Extrema in der Dichtekarte und können auch unter Rotation wiedergefunden werden. Für eine effiziente Berechnung wird hierfür eine 3x3 Hessematrix verwendet. Für einen Punkt x = (x, y, z) und der Skalierung σ gilt:

$$H(x, \sigma) = \begin{pmatrix} S_{xx}(x, \sigma) & S_{xy}(x, \sigma) & S_{xz}(x, \sigma) \\ S_{yx}(x, \sigma) & S_{yy}(x, \sigma) & S_{yz}(x, \sigma) \\ S_{zx}(x, \sigma) & S_{zy}(x, \sigma) & S_{zz}(x, \sigma) \end{pmatrix} \tag{14}$$

mit

$$S_{xx}(x, \sigma) = \frac{\partial^2}{\partial x^2} S(x, \sigma) \tag{15}$$

als zweite partielle Ableitung nach der x-Richtung der Scale-Space-Funktion. Die Terme in der Hessematrix werden berechnet, indem die zweite partielle Ableitung mit der Dichtekarte gefaltet werden:

$$S_{xx} = D * \frac{\partial^2}{\partial x^2} g(\sigma) \tag{16}$$

Anschließend wird $|Det(H)|$ für alle Punkte bestimmt und schlechte Ergebnisse durch einen Grenzwert eliminiert. Die lokalen Maxima werden schließlich als Schlüsselstellen gewählt.

Der THRIFT Detektor kann abschließend zusammengefasst werden als:

$$Interest(X) = \underset{(x,\sigma)}{arglocalmax} |Det(H(x, \sigma))|. \tag{17}$$

3.2 SHOT

SHOT (Signature of Histograms of OrienTations) ist ein speziell für den dreidimensionalen Raum entworfener Deskriptor [19]. Der Deskriptor verbindet Histogramme und Signaturen und ist gleichzeitig einzigartig und eindeutig.
Zu Beginn werden die totalen kleinsten Quadrate der Normalenrvektoren berechnet. Hierfür wird die Eigenwert Zerlegung der Kovarianz Matrix M, der k nächsten Nachbarn p_i des Punktes verwendet:

$$M = \frac{1}{k}\sum_{i=0}^{k}(p_i - \hat{p})(p_i - \hat{p})^T, \hat{p} = \frac{1}{k}\sum_{i=0}^{k} p_i \tag{18}$$

Der Gleichung werden schließlich gering gewichtete Distanzpunkte hinzugefügt, um robust gegenüber Störungen zu sein. Zusätzlich werden alle Punkte innerhalb der Sphäre mit Radius R, welcher zum Berechnen des Deskriptors verwendet wird, für die Berechnung von M verwendet. Aus Effizienzgründen wird die Mittelpunktberechnung durch die Schlüsselstelle p ersetzt. Mit $d_i = ||p_i - p||_2$ ergibt sich M nun als gewichtete lineare Kombination:

$$M = \frac{1}{\sum_{i:d_i \leq R}(R - d_i)} \sum_{i:d_i \leq R}(R - d_i)(p_i - p)(p_i - p)^T \tag{19}$$

Die erhaltenen Eigenvektoren müssen noch eindeutig gemacht werden. Hierzu wird das Vorzeichen der Vektoren so orientiert, dass es kohärent mit der Mehrheit ist. Die drei Eigenvektoren werden im folgenden, nach absteigenden Eigenwert sortiert, als x^+, y^+ und z^+ bezeichnet. Die entgegengesetzten Vektoren werden analog als x^-, y^- und z^- definiert. Die eindeutige x-Achse ergibt sich als:

$$S_x^+ \doteq \{i : d_i \leq R \wedge (p_i - p) \cdot x^+ \geq 0\} \tag{20}$$

$$S_x^- \doteq \{i : d_i \leq R \wedge (p_i - p) \cdot x^- > 0\} \tag{21}$$

$$x = \begin{cases} x^+, |S_x^+| \geq |S_x^-| \\ x^-, \text{sonst} \end{cases} \tag{22}$$

Die z-Achse wird auf gleicher weiße ermittelt, die y-Achse ergibt sich als $z \times x$.
Aus den vorliegenden Daten wird eine Menge an lokalen Histogrammen berechnet. Diese wird schlussendlich zum Deskriptor zusammengefasst. Für jedes lokale Histogramm werden die Berechnungspunkte anhand der Funktion $cos\theta_i$ unterteilt. Diese besitzt den Winkel θ zwischen den Oberflächenvektor an jedem Punkt innerhalb der Sphäre n_{vi} und der Oberflächen Normalen an der Schlüsselstelle n_u. Es gilt: $cos\theta_i = n_u \cdot n_{vi}$. Mithilfe eines isotropen sphärischen Gitters wird aus diesen Histogrammen eine Signatur erstellt. Das Gitter umfasst Partitionen entlang der radialen, der Azimut- und der Elevationsachse. Die einzelnen Volumen

der Signatur kodieren deskriptive Entitäten der lokalen Histogramme. Die grobe Unterteilung dieser wird als Deskriptor verwendet werden.
Um Grenzeffekte zu vermeiden werden die einzelnen Punkte der Histogramme mit ihren Nachbarn interpoliert. Hierfür wird jeder Punkt mit $1 - d$ multipliziert, wobei d die normalisierte Distanz der aktuellen Entität mit dem Mittelwert des Histogramms bezeichnet. Für zusätzliche Robustheit gegenüber unterschiedlichen Punktedichten wird der gesamte Deskriptor noch so normalisiert, dass er eins in der Summe entspricht.

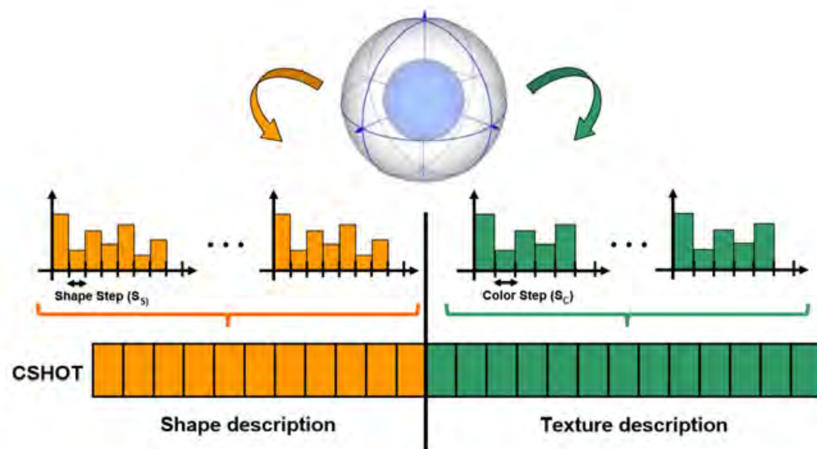


Abbildung 8: Die Abbildung zeigt eine Visualisierung des zusammengesetzten ColorSHOT Deskriptor. Die obere Sphäre entspricht dabei der Struktur der Signatur [20].

Den erhaltenen Konturendeskriptor kombinieren wir mit einer Texturbeschreibung zum Color SHOT Deskriptor [20]. Dafür wird die Signatur in der generischen Weise $SH_{G,f}(P)$ definiert. P entspricht der Schlüsselstelle und G entspricht einer Vektor-gewichteten Eigenschaft eines Scheitelpunkts. f bezeichnet die zum Vergleich verwendete Metrik. Im normalen SHOT entspricht f dem Produkt und G der Bestimmung der Oberflächennormalen.
Zum Erhalt des Deskriptor $D(P)$ der Schlüsselstelle P werden die m Signaturen von Histogrammen mit unterschiedlichen Eigenschafts- und Metrik-paaren vereint:

$$D(P) = \bigcup_{i=1}^{m} SH^{i}_{(G,f)}(P) \tag{23}$$

Für ColorShot wird der Fall $m = 2$ verwendet und der normale SHOT Deskriptor an die Erste Stelle gesetzt. Für den zweiten texturbasierten Teil wurde ein

Verfahren im RGB-Farbraum und im CIELab-Farbraum definiert. Im folgenden wird nur auf den Deskriptor im RGB-Raum eingegangen, welcher später in der Implementierung verwendet wird. Der Texturenbasierte Vektor G wird für jeden Scheitelpunkt auf das RGB-Tripel an Intensitäten gesetzt und im Folgenden mit R bezeichnet. Für die RGB-Werte lieferte eine Metrik basierend auf der Norm L_p in den Versuchen das beste Resultat. Es ist daher das in PCL implementierte Verfahren. Der Operator $l(\cdot)$ entspricht der Summe der absoluten Differenzen der Werte:

$$l(R_p, R_q) = \sum_{i=1}^{3} |R_p(i) - R_Q(i)| \tag{24}$$

3.3 Implementierung

Die Implementierung des dreidimensionalen Stitchers ähnelt im Aufbau dem zweidimensionalen Stitcher. Der Stitcher ist erneut in C++ und Visual Studio 2019 implementiert, benutzt statt der Bilderverarbeitungsbibliothek OpenCV nun aber die Point Cloud Library (PCL) in der Version 1.9.1. Weil PCL sehr viele Unterabhängigkeiten enthält, wird das Projekt selbst mit CMake erstellt. Die von PCL erzeugten Deskriptoren besitzen keinen einheitlichen Typ mehr. Daher wird die Funktion zur Kernpunktbestimmung und Deskriptoren-Erstellung nicht mehr abstrakt definiert. Das Programm besteht nur noch aus einem Struct zum Speichern der Daten zum jeweiligen Bild, der Klasse Stitcher und der main.cpp Datei. Zusätzlich ist eine Klasse CloudVisualizer enthalten, welche zur Visualisierung der (Zwischen-)Ergebnisse dient.

3.3.1 Projektorganisation mit CMake

```
1  cmake_minimum_required(VERSION 2.6 FATAL_ERROR)
2  project(3D-Image-Stitching)
3  find_package(PCL 1.9 REQUIRED)
4  include_directories(${PCL_INCLUDE_DIRS})
5  link_directories(${PCL_LIBRARY_DIRS})
6  add_definitions(${PCL_DEFINITIONS})
7  SET(TARGET_H
8       Stitcher.h
9       ExtendedCloud.h
10      CloudVisualizer.h
11 )
12 SET(TARGET_SRC
13      Stitcher.cpp
14      ExtendedCloud.cpp
15      CloudVisualizer.cpp
16 )
17 add_executable(3D-Image-Stitcher main.cpp
18                Stitcher.h Stitcher.cpp
19                ExtendedCloud.h ExtendedCloud.cpp
20                CloudVisualizer.h CloudVisualizer.cpp)
21 target_link_libraries(3D-Image-Stitcher ${PCL_LIBRARIES})
```

Listing 13: Das Makefile zur Erstellung des Projektes.

Als Erstes wird die minimale CMake Version zur Erstellung gesetzt. Anschließend wird der Projektname und das PCL Paket definiert. Über die Umgebungsvariablen PCL_INCLUDE_DIRS, PCL_LIBRARY_DIRS und PCL_DEFINITIONS werden die benötigten Abhängigkeiten inkludiert. Zum Schluss werden die Header und Source-Dateien gesetzt und die zu erzeugenden Dateien in add_executable angegeben. Als Ergebnis erhält man nach der Generierung mit CMake eine fertige Visual Studio Solution, welche alle Dateien und Abhängigkeiten enthält.

3.3.2 Das Struct ExtendedCloud

```
typedef PointCloud<PointXYZRGB> RGBCloud;
typedef PointCloud<SHOT1344> ColorShotCloud;

struct ExtendedCloud
{
public:
  RGBCloud::Ptr cloud;
  RGBCloud::Ptr downsampledCloud;
  RGBCloud::Ptr keypoints;
  PointCloud<Normal>::Ptr normals;
  ColorShotCloud::Ptr descriptors;
  Eigen::Matrix4f transformation;
  ExtendedCloud(RGBCloud::Ptr cloud);
};
```

Listing 14: Die Definition des Structes ExtendedCloud

Das Struct ähnelt dem ExtendedImage aus dem vorherigen Kapitel 2.4.1. Anstelle von Bildern werden Point Clouds vom Punkttyp PointXYZRGB gespeichert. Es handelt sich um Punkte mit einer x-, y- und z-Angabe und dem zugehörigen RGB Wert. Die Deskriptoren werden in einer PointCloud vom ColorSHOT spezifischen Typ SHOT1344 gespeichert, welcher aus 1344 float Werten besteht. Statt einer 3x3 Homographie wird im dreidimensionalen eine 4x4 Transformationsmatrix benötigt. Das Struct besitzt zu den äquivalenten Daten noch eine skalierte Cloud, weil die Berechnungszeiten auf den Orginaldaten zu lange dauern würden. Zusätzlich wird im Dreidimensionalen zur Deskriptoren-Bestimmung noch eine Normalen Point Cloud benötigt. Die Daten werden als shared Pointer aus der boost Bibliothek gespeichert, welche in PCL enthalten ist. Die Pointer geben ihren Speicher nach der Verwendung selbständig frei, wodurch kein Aufruf von delete mehr nötig ist [21].
Darüber hinaus enthält die Definitionsdatei des Structs die beiden Typ-Definitionen RGBCloud und ColorShotcloud, um den Code leserlicher zu gestalten.

3.3.3 Das Hauptprogramm

Die main.cpp Datei besteht aus einer Methode zum Einlesen der Daten, einer Methode zum Schreiben des Ergebnisses und der Main-Methode. Die verwendeten Daten liegen im Binärformat vor und müssen daher selbst in das PCL format Point Cloud gebracht werden.
Die ersten sechs Bytes der Daten geben die Breite (x-Richtung), Höhe (y-Richtung) und Tiefe (z-Richtung) des Bildes an. Anschließend folgen $w * h * d * 2$ Bytes mit

den Grauwerten des Bildes.

```
RGBCloud readFile(char* file)
{
  std::ifstream dataFile(file, std::ios_base::binary);
  dataFile.seekg(0, std::ios::end);
  size_t filesize = dataFile.tellg();
  dataFile.seekg(0, std::ios::beg);
  uint16_t* data = new uint16_t[filesize / 2];
  dataFile.read(reinterpret_cast<char*>(data), filesize);

  uint16_t width = data[0];
  uint16_t height = data[1];
  uint16_t depth = data[2];
  uint32_t totalPoints = (uint32_t)(width * height * depth);

  RGBCloud cloud;
  cloud.width = totalPoints;
  cloud.height = 1;
  cloud.is_dense = false;
  cloud.points.resize(totalPoints);

  int x = 0;
  int y = 0;
  int z = 0;
  for (size_t i = 0; i < totalPoints; i++)
  {
    cloud.points[i].x = x;
    cloud.points[i].y = y;
    cloud.points[i].z = z;

    uint16_t grayscale = data[i + 3];
    int rgb = grayscale * 0x00010101;
    cloud.points[i].rgb = rgb;

    x++;
    if (x % width == 0)
    {
      x = 0;
      y++;
      std::cout << "\n";
    }
    if ((y != 0) && (y % height == 0))
    {
      x = 0;
      y = 0;
      z++;
    }
  }
  return cloud;
}
```

Listing 15: Die Methode readFile zum Einlesen der Daten.

Als Erstes wird ein binärer Stream zum Einlesen der Datei erzeugt. Mittels seekg und tellg wird die Größe der Datei ermittelt. Der Zeiger wird mittels seekg auf das Ende und anschließend wieder auf Anfang gesetzt. Die Funktion tellg liefert die char Position am aktuellen Zeiger zurück. Weil ein char nur ein Byte entspricht, die Daten aber im zwei Byte Format vorliegen, wird das data Array mit der Hälfte des Ergebnisses initialisiert. Anschließend wird die gesamte Datei zwei Byte weise eingelesen.
Aus den Angaben zur Höhe, Breite und Tiefe wird die Gesamtzahl an Punkten ermittelt und eine entsprechende RGBCloud angelegt. Die Angabe height wird nur in sortierten Tiefenbildern verwendet, daher wird width gleich der Anzahl gesetzt. Die Point Cloud wird der Reihe nach mit einer For-Schleife und den umgerechneten RGB Wert befüllt. Die x-, y- und z- Werte werden dabei kontinuierlich erhöht. Die Umrechnung des RGB Wertes erfolgte dabei irrtümlich in die falsche Hexadezimale Darstellung 0x00RRGGBB. PCL verwendet für die interne Speicherung, im Gegensatz zu den eingelesenen Binärdaten, nur uint8_t für die einzelnen Werte [22]. Weil der gesamte RGB Ausdruck in einem float gespeichert wird, führt die Zuweisung aber zu keinem Überlauf. Das Ergebnis ist eine eindeutige Farbwert Abbildung, wodurch die Strukturen in den Bildern besser erkennbar wurden. Insbesondere mit der Verwendung des ColorSHOT Deskriptor wurde damit ein stabiles Stitching Verfahren erzeugt. Die Abbildung ist zum abschließenden Speichern der zusammengesetzten Daten umkehrbar, das Endresultat enthält damit wieder die Grauwerte der Ursprungsbilder. Die eigentlich falsche Konvertierung wurde daher beibehalten. Nach dem Einlesen der Daten wird der Speicher des Array data freigegeben und der Zeiger auf NULL gesetzt. Zum Schluss wird die fertige Point Cloud zurück gegeben.

```
RGBCloud readFile(char* file)
{
void SaveResult(RGBCloud::Ptr cloud, char* memoryLocation)
{
  PointXYZRGB min, max;
  getMinMax3D(*cloud, min, max);

  uint16_t width = max.x - min.x + 1;
  uint16_t height = max.y - min.y + 1;
  uint16_t depth = max.z - min.z + 1;

  int arraySize = (width * depth * height) + 3;
  uint16_t* data = new uint16_t[arraySize];
  data[0] = width;
  data[1] = height;
  data[2] = depth;

  for (int i = 0; i < cloud->points.size(); ++i)
  {
    PointXYZRGB current = cloud->points[i];
    int index = current.x + (current.y * width)
              + (current.z * width * height) + 3;
    //convert back to grayscale
    data[index] = current.rgb / 0x00010101;
  }

  std::ofstream dataFile(memoryLocation, std::ios::binary);
  dataFile.write(reinterpret_cast<char*>(data), arraySize * 2);
}
```

Listing 16: Die Methode SaveResult

Zum Speichern des Endergebnis werden als erstes die maximalen und minimalen x, y und z Werte extrahiert. Aus diesen wird die Breite, Höhe und Tiefe für das Ergebnis berechnet. Dies entspricht dem kleinsten Viereck, welches alle Punkte enthält. Daraus wird die Gesamtanzahl an Werten errechnet und das array data entsprechend erstellt. Die Breite, Höhe und Tiefe wird direkt eingefügt. Anschließend wird über die Point Cloud iteriert und für jeden Punkt der entsprechende Index im Array kalkuliert. Der zurückgerechnete Grauwert wird an der Indexstelle in das Array geschrieben. Zum Abschluss wird das Endergebnis binär an die vorgegebene Stelle gespeichert.

```
int main(int argc, char** argv)
{
  if (argc < 7)
  {
    std::cout << "Not enough parameters for Stitching given."
              << " Arguments are: (1) Leaf Size, (2) Sift Scale,"
              << " (3) RANSAC Threshold,"
              << " (4) Memory Location to Save Result,"
              << " (5+) Input Images" << std::endl;
    return -1;
  }

  float leafSize = atof(argv[1]);
  float scale = atof(argv[2]);
  float threshold = atof(argv[3]);

  //read images
  std::vector<RGBCloud::Ptr> clouds;
  for (int i = 5; i < argc; ++i)
  {
    clouds.push_back(readFile(argv[i]));
    //CloudVisualizer::visualizeCloud(*clouds[i - 5]);
  }

  //Stitch
  Stitcher stitcher(leafSize, scale, threshold);
  RGBCloud::Ptr result = stitcher.StitchPointClouds(clouds);

  //Save
  SaveResult(result, argv[4]);
  return 0;
}
```

Listing 17: Die Main-Methode

In der Main-Methode wird zuerst auf die korrekte Anzahl an Parametern überprüft. Stimmt diese nicht wird ein Bedienungshinweis ausgegeben und das Programm beendet. Wird das Programm korrekt aufgerufen werden anschließend die Leaf Size für das komprimieren, die zu verwendende Skalierung für SIFT und der Grenzwert für RANSAC eingelesen. Schließlich werden die Daten der Reihe nach eingelesen und der jeweilige Pointer in den Vektor clouds gespeichert. Ein Objekt vom Typ Stitcher wird erstellt und die eingelesenen Daten zusammengesetzt. Zum Schluss wird das Ergebnis gespeichert und das Programm beendet.

3.3.4 Die Klasse Stitcher

```
class Stitcher
{
private:
  Hinweis: Die Attributdefinitionen an dieser Stelle wurden fuer eine
           bessere Uebersicht hier ausgelassen. Die Definitionen
           finden sich im Abschnitt 3.4 Konfiguration.
  std::vector<ExtendedCloud> clouds;

  RGBCloud::Ptr DownsampleCloud(RGBCloud::Ptr ptrCloud);
  RGBCloud::Ptr ComputeKeypoints(ExtendedCloud& extendedCloud);
  PointCloud<Normal>::Ptr ComputeNormals(
      ExtendedCloud& extendedCloud);
  ColorShotCloud::Ptr ComputeColorShot(ExtendedCloud& extendedCloud);
  CorrespondencesPtr EstimateCorrespondances(
      ExtendedCloud& cloud1,
      ExtendedCloud& cloud2);
  CorrespondencesPtr RejectObliqueCorrespondences(
      ExtendedCloud& cloud1,
      ExtendedCloud& cloud2,
      CorrespondencesPtr correspondences);
  PointCloud<PointNormal>::Ptr ConvertNormalCloud(
      ExtendedCloud& cloud);
  CorrespondencesPtr RejectSurfaceNormalsCorrespondences(
      ExtendedCloud& cloud1,
      ExtendedCloud& cloud2,
      CorrespondencesPtr correspondences);
  CorrespondencesPtr ComputeGoodCorrespondences(
      ExtendedCloud& cloud1,
      ExtendedCloud& cloud2,
      CorrespondencesPtr correspondences);
  Eigen::Matrix4f EstimateTranslation(
      ExtendedCloud& cloud1,
      ExtendedCloud& cloud2,
      CorrespondencesPtr Correspondences);
  RGBCloud::Ptr CombinePointClouds();
public:
  Stitcher();
  RGBCloud::Ptr StitchPointClouds(std::vector<RGBCloud> clouds);
};
```

Listing 18: Die Klassendeklaration der Klasse Stitcher ohne Attribute

Die Klasse Stitcher ist ähnlich dem zweidimensionale Pendant aufgebaut. Es gibt eine zusätzliche Methode zum Herunterrechnen der Bilder und zum Berechnen der Oberflächennormalen. Die Methode zur Berechnung der Kernpunkte und zur Be-

rechnung der Deskriptoren ist in zwei einzelne Methoden aufgeteilt. Zur Keypoint-Bestimmung wird eine Adaption von SIFT verwendet. Zur Deskriptoren-Erzeugung wird der auf Farben erweiterte SHOT Deskriptor benutzt. Die Transformationsmatrix wird nicht direkt von RANSAC berechnet, sondern in einer eigenen Methode aus den erhaltenen Inliniern bestimmt. Darüber hinaus werden zusätzliche Methoden zur Match Verifizierung genutzt.
Die Klasse enthält zusätzlich die für die verwendeten Komponenten nötigen Konstanten und Konfigurationswerte im Header. Die Konfiguration muss beim Erzeugen des Stitcher Objektes im Konstruktor angegeben werden. Genaueres zu den Werten findet sich im Abschnitt 3.4 Konfiguration.

```cpp
RGBCloud::Ptr Stitcher::StitchPointClouds(std::vector<RGBCloud> clouds)
{
  for (size_t i = 0; i < clouds.size(); ++i)
  {
    //save to Extended Cloud
    RGBCloud::Ptr ptrCloud(new RGBCloud);
    *ptrCloud = clouds[i];
    ExtendedCloud extendedCloud(ptrCloud);

    //Downsample for faster calculations
    extendedCloud.downsampledCloud = DownsampleCloud(ptrCloud);

    //Calculate keypoints and descriptors for each Cloud
    extendedCloud.keypoints = ComputeKeypoints(extendedCloud);
    extendedCloud.normals = ComputeNormals(extendedCloud);
    extendedCloud.descriptors = ComputeColorShot(extendedCloud);
    this->clouds.push_back(extendedCloud);

    //Calculate Correspondences
    if (i > 0)
    {
      ExtendedCloud cloudBefore = this->clouds[i - 1];
      ExtendedCloud cloudCurrent = this->clouds[i];
      CorrespondencesPtr correspondences = EstimateCorrespondances(
            cloudBefore, cloudCurrent);
      correspondences = RejectSurfaceNormalsCorrespondences(
            cloudBefore, cloudCurrent, correspondences);
      correspondences = RejectObliqueCorrespondences(
            cloudBefore, cloudCurrent, correspondences);
      correspondences = ComputeGoodCorrespondences(
            cloudBefore, cloudCurrent, correspondences);
      this->clouds[i].transformation = EstimateTranslation(
            cloudBefore, cloudCurrent, correspondences).inverse();
    }
  }
  return CombinePointClouds();
}
```

Listing 19: Die Methode StitchPointClouds. Die im Code enthaltenen Zeilen zur Visualisierung und zur Ausgabe des Prozessfortschritts wurden für eine bessere Übersicht entfernt.

Die Methode StitchPointClouds stellt die Hauptschleife des Stitching Vorgangs dar. Im Gegensatz zum zweidimensionalen Pendant werden die Matches direkt in der ersten For-Schleife berechnet. Hierdurch kann auf einen falschen Match reagiert werden. Am Ablauf ändert sich sonst nicht viel. Für schnellere Berechnungszeiten wird das Bild zuerst komprimiert. Anschließend werden die Schlüsselstellen und

Oberflächennormalen bestimmt. Aus diesen werden die Deskriptoren erzeugt. Die Deskriptoren werden ab dem zweiten Bild mit den vorherigen verglichen und die erhaltenen Matches verifiziert. Schlussendlich wird die Transformationsmatrix berechnet und gespeichert. Sind alle Berechnungen abgeschlossen, werden die Bilder mit den erhaltenen Transformationen zusammengesetzt. Die auskommentierten Aufrufe an den CloudVisualizer werden für den Ergebnisteil verwendet.

```
1  RGBCloud::Ptr Stitcher::DownsampleCloud(RGBCloud::Ptr ptrCloud)
2  {
3    std::cout << "Total points:" << ptrCloud->points.size() << std::endl;
4    PointCloud<PointXYZRGB>::Ptr cloud_filtered(new RGBCloud);
5    VoxelGrid<PointXYZRGB> grid;
6    grid.setInputCloud(ptrCloud);
7    grid.setLeafSize(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE);
8    grid.filter(*cloud_filtered);
9    std::cout << "Points after downsampling:"
10             << cloud_filtered->points.size() << std::endl;
11   return cloud_filtered;
12 }
```

Listing 20: Die Methode DownsamlpeCloud

In der Methode DownSampleCloud wird die ursprüngliche Point Cloud für effizientere Berechnungszeiten herunter skaliert. Hierfür wird ein gleichmäßiges VoxelGrid verwendet. Die Leaf Size des Voxel Grids ist abhängig von der Größe der Point Cloud zu wählen und wird daher vom Benutzer beim Aufruf des Programms gesetzt. Das skalierte Bild wird anschließend zurück gegeben.

```cpp
RGBCloud::Ptr Stitcher::ComputeKeypoints(ExtendedCloud& extendedCloud)
{

  SIFTKeypoint<PointXYZRGB, PointWithScale> sift;
  PointCloud<PointWithScale>::Ptr keypoints(
        new PointCloud<PointWithScale>);

  sift.setInputCloud(extendedCloud.downsampledCloud);
  sift.setSearchMethod(search::KdTree<PointXYZRGB>::Ptr(
        new search::KdTree<PointXYZRGB>));
  sift.setScales(SCALE, OCTAVES, SCALES_PER_OCTAVE);
  sift.setMinimumContrast(CONTRAST);
  sift.compute(*keypoints);

  std::cout << "Number of keypoints:"
            << keypoints->points.size() << std::endl;
  RGBCloud::Ptr keypoints_rgb(new RGBCloud);
  copyPointCloud(*keypoints, *keypoints_rgb);
  //CloudVisualizer::visualizeKeypoints(
  //    extendedCloud.downsampledCloud, keypoints_rgb);
  return keypoints_rgb;
}
```

Listing 21: Die Methode ComputeKeypoints

In dieser Funktion werden die Schlüsselstellen des Bildes mittels SIFT berechnet. Für eine schnellere Suche wird als Suchmethode ein K-d-Baum verwendet. Die Skalierung ist abhängig von der Leaf Size und wird daher ebenfalls vom Benutzer gesetzt. Näheres zu Konfigurationswerten findet sich im Abschnitt 3.4 Konfiguration. Die gefundenen Schlüsselstellen werden zum Schluss von Punkten mit Skalierungen zurück in normale RGB-Punkte konvertiert. Dies ist zur späteren Berechnung der Deskriptoren notwendig.

```
PointCloud<Normal>::Ptr Stitcher::ComputeNormals(
    ExtendedCloud& extendedCloud)
{
  PointCloud<Normal>::Ptr ptrNormals(new PointCloud<Normal>);

  NormalEstimation<PointXYZRGB, Normal> normal;
  normal.setInputCloud(extendedCloud.downsampledCloud);
  normal.setSearchMethod(search::KdTree<PointXYZRGB>::Ptr(
      new search::KdTree<PointXYZRGB>));
  normal.setKSearch(NORMAL_K);
  normal.compute(*ptrNormals);
  //CloudVisualizer::visualizeNormals(
  //    extendedCloud.downsampledCloud, ptrNormals);
  std::cout << "Normals computed" << std::endl;
  return ptrNormals;
}
```

Listing 22: Die Methode ComputeNormals

Als nächstes werden die Oberflächennormalen berechnet. Diese sind ebenfalls zur Deskriptoren Erstellung notwendig. Sie werden aber auch in einem der Match Verifikationsverfahren verwendet. Für die Suche wird aus Effizienzgründen wieder ein K-d-Baum verwendet. Die NormalEstimation bietet neben der Suchfunktion mit einem Radius auch das Suchen mit einer Anzahl an k Nachbarn an. Dies erspart die Anpassung des Radius an die Leaf Size.

```
ColorShotCloud::Ptr Stitcher::ComputeColorShot(
    ExtendedCloud& extendedCloud)
{
  ColorShotCloud::Ptr descriptors(new ColorShotCloud());
  pcl::SHOTColorEstimation<PointXYZRGB, Normal, SHOT1344> shot;
  shot.setInputCloud(extendedCloud.keypoints);
  shot.setSearchSurface(extendedCloud.downsampledCloud);
  shot.setInputNormals(extendedCloud.normals);
  shot.setRadiusSearch(SHOT_RADIUS);
  shot.compute(*descriptors);
  std::cout << "Deskriptor computed" << std::endl;
  return descriptors;
}
```

Listing 23: Die Methode ComputeColorShot

Die Methode ComputeColorShot berechnet aus den vorhandenen Schlüsselstellen und Oberflächennormalen den Deskriptor. Hierzu wird eine um Farbwerte erweiterte Version von SHOT verwendet. Diese lieferte in den Tests als einziger Deskriptor ein gutes Matching Ergebnis. Der Deskriptor ist mit 1344 float Werten allerdings deutlich größer als andere Deskriptoren. Das Berechnen und der spätere Matching

Prozess dauert daher länger. Für die Berechnung werden die Schlüsselstellen als InputCloud verwendet, um an diesen einen Deskriptor zu erstellen. Für die Berechnung wird mit setSearchSurface das gesamte Bild verwendet. Der Radius wird allgemein auf das Doppelte der Leaf Size gesetzt. Hiermit wurden in den Tests gute Ergebnisse erzielt.

```
1  CorrespondencesPtr Stitcher::EstimateCorrespondances(
2      ExtendedCloud& cloud1, ExtendedCloud& cloud2)
3  {
4    registration::CorrespondenceEstimation<SHOT1344, SHOT1344> estimater;
5    CorrespondencesPtr correspondences(new Correspondences);
6    estimater.setInputSource(cloud1.descriptors);
7    estimater.setInputTarget(cloud2.descriptors);
8    estimater.determineReciprocalCorrespondences(
9        *correspondences, CORRESPONDENCES_THRESHOLD);
10   std::cout << "Number of Correspondences:"
11             << correspondences->size() << std::endl;
12   return correspondences;
13 }
```

Listing 24: Die Methode Estimate Correspondences

Für die Bestimmung der Matches werden paarweise die Deskriptoren der Point Clouds verglichen. Zusätzlich wird ein Grenzwert für die Distanz der Matches festgelegt, um die schlechtesten Ergebnisse direkt auszusortieren. Außerdem werden nur die Reciprocalen Matches verwendet, also die beidseitig übereinstimmenden Schlüsselstellen.

```
PointCloud<PointNormal>::Ptr Stitcher::convertNormalCloud(
    ExtendedCloud& cloud)
{
  //Extract Normals at Keypoints and convert to PointNormal
  PointCloud<PointNormal>::Ptr keypoints(new PointCloud<PointNormal>);
  for (size_t i = 0; i < cloud.keypoints->size(); ++i)
  {
    PointXYZRGB point = cloud.keypoints->at(i);
    std::vector<PointXYZRGB>::iterator it = std::find_if(
    cloud.downsampledCloud->begin(), cloud.downsampledCloud->end(),
      [point](const PointXYZRGB & other)
      {return (point.x == other.x && point.y == other.y
              && point.z == other.z);});
    int index = std::distance(cloud.downsampledCloud->begin(), it);
    PointNormal current;
    //Copy x, y, z values
    PointXYZRGB currentKeypoint = cloud.keypoints->at(i);
    current.x = currentKeypoint.x;
    current.y = currentKeypoint.y;
    current.z = currentKeypoint.z;
    //Copy normal values
    Normal currentNormal = cloud.normals->at(index);
    current.normal_x = currentNormal.normal_x;
    current.normal_y = currentNormal.normal_y;
    current.normal_z = currentNormal.normal_z;
    current.curvature = cloud.normals->at(index).curvature;
    keypoints->push_back(current);
  }
  return keypoints;
}
```

Listing 25: Die Methode ConvertNormalCloud

Zur Verwendung des ersten Verifikationsverfahren werden die Oberflächennormalen inklusive x-, y- und z-Werten an den Schlüsselstellen benötigt. Hierzu wird über die Schlüsselstellen iteriert. Mit find_if wird nach dem Index der Schlüsselstelle in der normalen Point Cloud gesucht. Ein Punkt wird dabei als gleich betrachtet, wenn die x, y und z Werte übereinstimmen. Anschließend werden die Werte aus der Normalen Point Cloud mit den x-, y- und z-Werten kombiniert zu einem PointNormal. Dieser wird zur Point Cloud vom Typ PointNormal hinzugefügt. Die fertige Point Cloud wird schließlich zurück gegeben.

```cpp
CorrespondencesPtr Stitcher::RejectSurfaceNormalsCorrespondences(
    ExtendedCloud& cloud1, ExtendedCloud& cloud2,
    CorrespondencesPtr correspondences)
{
  //Extract Normals at Keypoints and convert to PointNormal
  PointCloud<PointNormal>::Ptr keypoints1 = convertNormalCloud(cloud1);
  PointCloud<PointNormal>::Ptr keypoints2 = convertNormalCloud(cloud2);

  CorrespondencesPtr good_correspondences(new Correspondences);
  registration::CorrespondenceRejectorSurfaceNormal nRejector;
  nRejector.setThreshold(std::acos(DEG2RAD(45.0)));
  nRejector.initializeDataContainer<PointNormal, PointNormal>();
  nRejector.setInputSource<PointNormal>(keypoints1);
  nRejector.setInputNormals<PointNormal, PointNormal>(keypoints1);
  nRejector.setInputTarget<PointNormal>(keypoints2);
  nRejector.setTargetNormals<PointNormal, PointNormal>(keypoints2);
  nRejector.setInputCorrespondences(correspondences);
  nRejector.getCorrespondences(*good_correspondences);
  std::cout << "Number of Correspondences after Normal Rejection:"
            << good_correspondences->size()
            << std::endl;
        return good_correspondences;
 }
```

Listing 26: Die Methode RejectSurfaceNormalsCorrespondences

Mit den eben berechneten PointNormals können nun die ersten Matches aussortiert werden. Es werden alle Matches entfernt, welche mehr als 45 Grad von der Oberflächen Richtung abweichen. Damit werden zum Beispiel senkrechte Ebenen nicht mehr gegenseitig als Match betrachtet.

```
CorrespondencesPtr Stitcher::RejectObliqueCorrespondences(
    ExtendedCloud& cloud1, ExtendedCloud& cloud2,
    CorrespondencesPtr correspondences)
{
  CorrespondencesPtr good_correspondences(new Correspondences);
  for (size_t i = 0; i < correspondences->size(); ++i)
  {
    PointXYZRGB pointOne = cloud1.keypoints->at(
        correspondences->at(i).index_query);
    PointXYZRGB pointTwo = cloud2.keypoints->at(
        correspondences->at(i).index_match);
    float translationX = pointOne.x - pointTwo.x;
    float translationY = pointOne.y - pointTwo.y;
    float translationZ = pointOne.z - pointTwo.z;
    if ((translationY < OBLIQUE_THRESHOLD
         && translationZ < OBLIQUE_THRESHOLD)
        || (translationX < OBLIQUE_THRESHOLD
            && translationZ < OBLIQUE_THRESHOLD)
        || (translationX < OBLIQUE_THRESHOLD
            && translationY < OBLIQUE_THRESHOLD))
    {
      good_correspondences->push_back(correspondences->at(i));
    }
  }
  std::cout << "Number of Correspondences after Oblique Rejection:"
            << good_correspondences->size() << std::endl;
  return good_correspondences;
}
```

Listing 27: Die Methode RejectObliqueCorrespondences

Da es sich bei den späteren Daten um Zerstückelungen handelt, können wir zusätzlich schräge Matches aussortieren. Bei beliebigen Bildern ist diese Verifikationsverfahren nicht anwendbar. Es wird mit einer For-Schleife über sämtliche Matches iteriert. Von diesen werden alle ausortiert, welche eine Verschiebung in mehr als eine Himmelsrichtung verursachen würden. Um minimale Verschiebungen aufgrund der Skalierung nicht auszusortieren, müssen die anderen beiden Himmelsrichtungen lediglich unter einen Grenzwert liegen.

```
CorrespondencesPtr Stitcher::ComputeGoodCorrespondences(
    ExtendedCloud &cloud1, ExtendedCloud &cloud2,
    CorrespondencesPtr correspondences)
{
  CorrespondencesPtr good_correspondences(new Correspondences);
  registration::CorrespondenceRejectorSampleConsensus<PointXYZRGB>
      ransac;
  ransac.setInputSource(cloud1.keypoints);
  ransac.setInputTarget(cloud2.keypoints);
  ransac.setMaximumIterations(RANSAC_ITERATIONS);
  ransac.setInlierThreshold(RANSAC_THRESHOLD);
  ransac.setInputCorrespondences(correspondences);
  ransac.getCorrespondences(*good_correspondences);
  std::cout << "Number of Correspondences after Ransac:"
            << good_correspondences->size() << std::endl;
  return good_correspondences;
}
```

Listing 28: Die Methode ComputeGoodCorrespondences

Auf den aussortierten Matches wird schlussendlich RANSAC angewendet. In diesem Fall wird RANSAC aber nur zur Bestimmung von Inlinern eingesetzt, das endgültige Modell wird anschließend berechnet. Dies liegt daran, dass RANSAC versucht einer Translation und Transformation aufzulösen. Die so berechneten Rotationen führten häufig zu schlechten Ergebnissen.

```
1  Eigen::Matrix4f Stitcher::EstimateTranslation(
2      ExtendedCloud& cloud1, ExtendedCloud& cloud2,
3      CorrespondencesPtr correspondences)
4  {
5    Eigen::Matrix4f translation = Eigen::Matrix4f::Identity();
6    float translationX = 0.0f, translationY = 0.0f, translationZ = 0.0f;
7    for (int i = 0; i < correspondences->size(); ++i)
8    {
9      PointXYZRGB leftKeypoint = cloud1.keypoints->points[
10         (*correspondences)[i].index_query];
11     PointXYZRGB rightKeypoint = cloud2.keypoints->points[
12         (*correspondences)[i].index_match];
13     translationX += rightKeypoint.x - leftKeypoint.x;
14     translationY += rightKeypoint.y - leftKeypoint.y;
15     translationZ += rightKeypoint.z - leftKeypoint.z;
16   }
17   translationX /= correspondences->size();
18   translationY /= correspondences->size();
19   translationZ /= correspondences->size();
20   translation(0, 3) = translationX;
21   translation(1, 3) = translationY;
22   translation(2, 3) = translationZ;
23   std::cout << "Transformation Matrix:" << std::endl
24             << translation << std::endl;
25   return translation;
26 }
```

Listing 29: Die Methode EstimateTranslation

Neben der Verwendung von RANSAC zur Modellbestimmung enthält PCL auch einen alleinstehenden TransformationEstimater auf Basis der Singulärwertzerlegung (SVD). Diese liefert ebenfalls eine Rotation und somit schlechte Ergebnisse. Nachdem die Zerstückelungen keine Rotation aufweisen können, wird direkt die Translation aus den Differenzen im Raum berechnet.

```cpp
RGBCloud::Ptr Stitcher::CombinePointClouds()
{
  RGBCloud::Ptr resultingCloud = clouds[clouds.size() - 1].cloud;
  for (size_t i = clouds.size() - 1; i > 0; --i)
  {
    //Transform interim result
        RGBCloud::Ptr transformedCloud(new RGBCloud);
        Eigen::Matrix4f transformation = clouds[i].transformation;
        transformPointCloud(*resultingCloud, *transformedCloud,
                            transformation);

        //Add next cloud
        RGBCloud::Ptr nextCloud = clouds[i - 1].cloud;

        //CloudVisualizer::visualizeCloudWithCoords(*transformedCloud);
        //CloudVisualizer::visualizeCloudWithCoords(*nextCloud);

        RGBCloud::Ptr mergedCloud (new RGBCloud);
        *mergedCloud += *nextCloud;
        *mergedCloud += *transformedCloud;
        //CloudVisualizer::visualizeCloudWithCoords(*mergedCloud);
        resultingCloud = mergedCloud;
  }
  return resultingCloud;
}
```

Listing 30: Die Methode CombinePointClouds

Zuletzt werden die Bilder inkrementell vom Ende zum Anfang zusammengesetzt. Wie im Zweidimensionalen wird die Transformationsmatrix auf das bisherige Ergebnis angewendet. Dann wird das nächste Bild hinzugefügt. PCL implementiert das Zusammenfügen von Point Clouds mit den += Operator. Doppelte Punkte werden dabei überschrieben. Zum Schluss wird das fertige Endresultat zurück gegeben.

3.4 Konfiguration

Komprimierung

Für die Komprimierung der Originaldaten wird nur die Leaf Size für das verwendete VoxelGrid gebraucht. Diese ist abhängig von der Größe der Daten und wird daher als Parameter beim Programmaufruf gesetzt. Bei einer zu kleinen Leaf Size entsteht in der Implementierung des VoxelGrid ein Integer Überlauf. Dies stellt die untere Schranke des Wertes dar. Wird das Bild hingegen zu stark komprimiert, funktioniert der Matching Prozess nicht mehr. Da eine stärkere Komprimierung zu einer erheblichen Effizienzsteigerung führt, sollte der Wert im Allgemeinen so

groß wie möglich gewählt werden, ohne dass eine falsche Transformation berechnet wird. In den Tests war eine Komprimierung auf etwa 200.000 Bildpunkte in den meisten Fällen ausreichend für ein korrektes Matching Ergebnis.

SIFT

Die Anzahl an Oktaven und Skalierungen pro Oktave werden entsprechend der Literatur [3] auf drei und vier gesetzt. Der minimale Kontrast eines Punktes ist nach empirischen Daten auf den festen Wert 0.5 gesetzt. Der Wert jeder Skalierung ist abhängig von der Dichtheit und Größe des Bildes und wird als Programmparameter gesetzt. Der Wert ist vor allem von der verwendeten Leaf Size abhängig. Die Implementierung des SIFT Algorithmus verwendet intern ein VoxelGrid, welches bei einem zu niedrigen Wert wieder zu einem Integer Überlauf führt. Die Skalierung sollte aus Effizienzgründen so gering wie möglich gewählt werden, ohne einen Überlauf zu verursachen. Ein höherer Wert führte in den Versuchen nur zu marginal mehr Schlüsselstellen, welche zu vernachlässigen sind.

Oberflächennormalen

Für die Berechnung des Oberflächenvektors kann anstatt eines Radius eine Anzahl an Nachbarn gesetzt werden. Um nicht erneut von der Leaf Size abhängig zu sein, wird diese Möglichkeit verwendet. Für die Berechnung werden die 25 nächsten Nachbarn gewählt. Dies war in den Versuchen ausreichend für stabile Werte.

SHOT

Für die Berechnung des SHOT Deskriptors ist ein Radius notwendig. Um keinen weiteren Parameter zu benötigen, welcher jedes mal verändert werden muss, wird das k-nächste-Nachbarn Verfahren imitiert. Hierzu wird der Radius fest auf das Doppelte der LeafSize gesetzt.

Matching

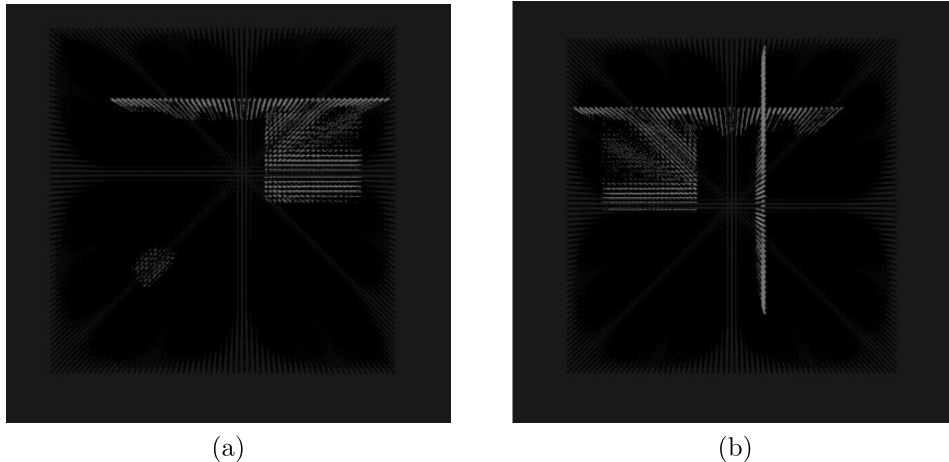
Für das Bestimmen der korrekten Übereinstimmungen wird ein Grenzwert von 1.0 mit angegeben. Dieser sortiert die schlechtesten Matches direkt aus. Der Wert ist bewusst hoch gewählt. Das aussortieren sollte eher den dafür spezialisierten Verifikationsverfahren überlassen werden. Ein niedrigerer Wert führte in den Versuchen zu einem schlechteren Ergebnis.

Verifikation

Für die Verifikation von schrägen Übereinstimmungen wird ein geringer Grenzwert von zwei Bildpunkten gewählt. Durch diesen sollen leicht verschobene Bildpunkte, etwa wegen der Komprimierung, trotzdem akzeptiert werden.
Für RANSAC werden zwei Konfigurationswerte verwendet. Die Implementierung des RANSAC Verfahrens in PCL gibt bei keinem gefundenen Modell nicht die größte Menge, sondern die Eingabemenge zurück. Dies führt unweigerlich zu einem Neustart des Programms mit anderen Werten. Die Anzahl an Iterationen wird daher mit 25.000 auf einen Wert gesetzt, bei welchen RANSAC einen Großteil der Möglichkeiten probiert. Dies soll sicherstellen, dass RANSAC in jedem Fall ein Modell findet. Der Grenzwert zur Akzeptanz wird beim Aufruf des Programms gesetzt. Somit lässt sich das Programm bei einer falschen oder nicht gefundenen Transformation noch etwas anpassen. In den Versuchen wurden Werte zwischen 0.01 und 0.1 verwendet, um ein korrektes Ergebnis zu erhalten.

3.5 Ergebnisse

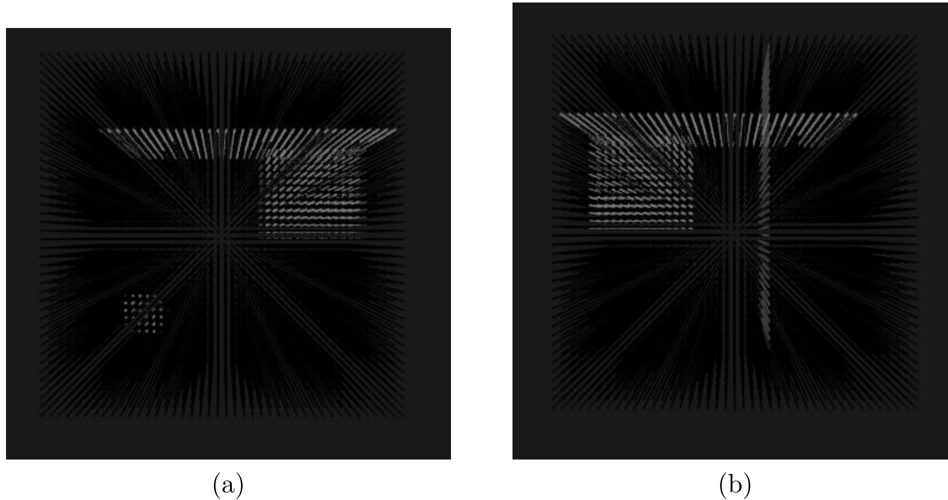
3.5.1 Visualisierung des Ablaufs



(a) (b)

Abbildung 9: Die zwei Ausgangsbilder des Würfels mit zwei Ebenen.

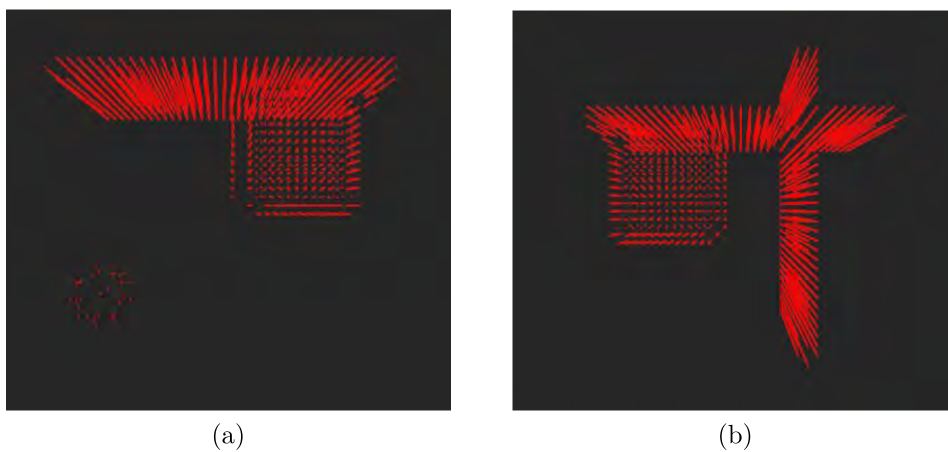
Für die Darstellung des Ablaufs werden Bilder eines Würfels mit zwei Ebenen verwendet. Der Würfel wurde in x- und y- Richtung in vier Einzelbilder zerschnitten. Die Bilder besitzen einen Überlapp von 50%. Verwendet werden die beiden Bilder unterhalb des y-Schnitts. Die Bilder werden mit den Werten Leaf Size = 1.5, Sift Scale = 0.1 und Ransac Treshold = 0.01 verarbeitet.



(a) (b)

Abbildung 10: Die komprimierten Bilder des Würfels mit zwei Ebenen.

Die Bilder werden als Erstes komprimiert. Die Form des Würfels und der Ebenen sind noch deutlich zu erkennen, die Pixeldichte ist merklich reduziert.



(a) (b)

Abbildung 11: Die Schlüsselstellen der Bilder.

Von den herunter-skalierten Bildern werden die Schlüsselstellen berechnet. Zur Verdeutlichung wurden die Schlüsselstellen rot eingefärbt und ohne die restlichen Datenpunkten dargestellt. Die Punkte verteilen sich gleichmäßig auf die Oberflächen des Würfels und der Ebenen.

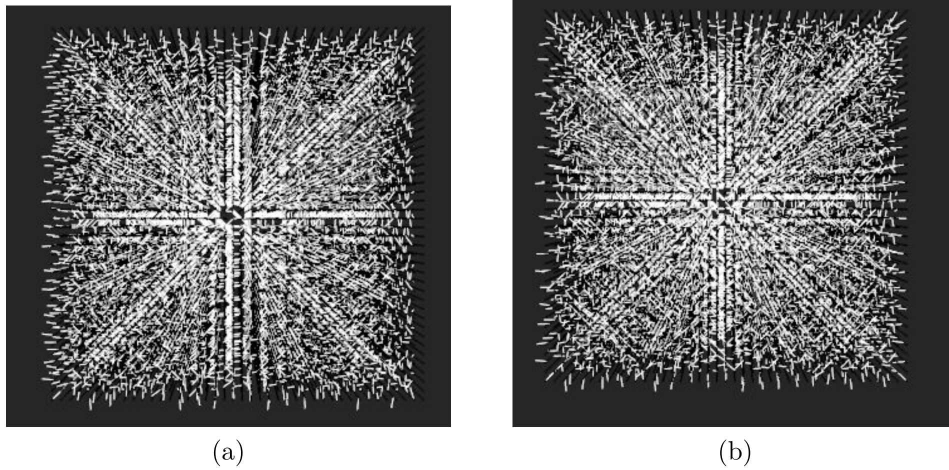

(a) (b)

Abbildung 12: Die Oberflächennormalen der Bilder.

Zur Deskriptoren Berechnung werden schließlich noch die Oberflächennormalen bestimmt. Die Richtung ist durch kleine Pfeile gekennzeichnet. Die Ausrichtung an den interessanten Stellen ist nur schwer erkennbar, weil die Normalen für jeden Bildpunkt berechnet und dargestellt werden.

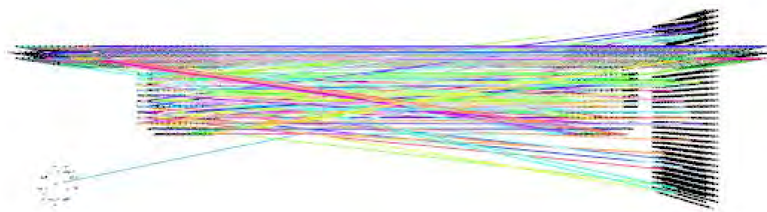


Abbildung 13: Die Matches zwischen den beiden Bildern direkt nach der Bestimmung.

Die Matches direkt nach Bestimmung sind quer verteilt. Es lassen sich deutlich falsche Übereinstimmungen zwischen der Oberfläche des Würfels und der vertikalen Ebene erkennen.

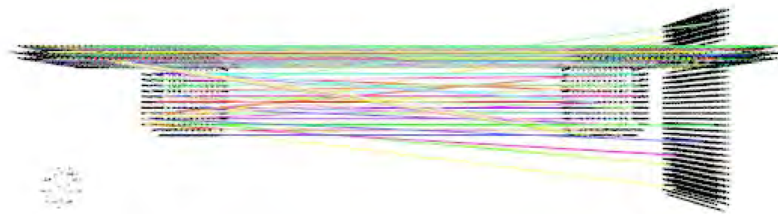


Abbildung 14: Die Matches zwischen den beiden Bildern nach Anwendung der Oberflächenverifikation.

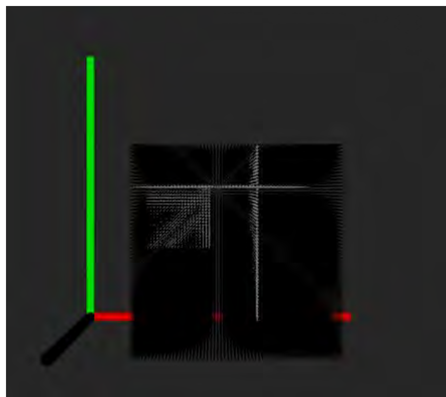
Nach der Verifikation anhand der Oberflächennormalen sind bereits die meisten falschen Übereinstimmungen aussortiert. Zwischen der linken Seite des Würfels und der linken Seite der vertikalen Ebene sind aber immer noch falsche Übereinstimmungen zu erkennen. Deren Ausrichtung zeigt jeweils nach links und wird somit als gleich angesehen.

Abbildung 15: Die Matches zwischen den beiden Bildern nach Anwendung der Geradenverifikation.

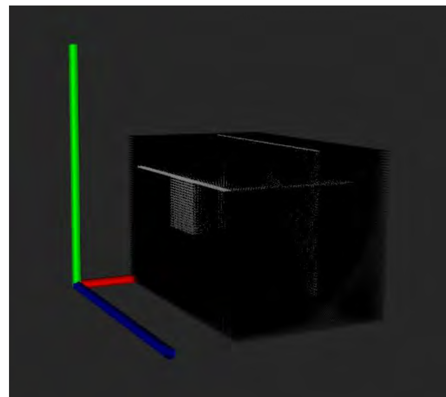
Nach dem Aussortieren von schrägen Übereinstimmungen ist auch ein Großteil dieser falschen Matches verschwunden. Es sind aber noch Verbindungen zwischen dem Würfel und der Oberfläche erkennbar.

Abbildung 16: Die Matches zwischen den beiden Bildern direkt nach der Verwendung von RANSAC.

Zum Schluss bestimmt RANSAC nur wenige korrekte Übereinstimmungen zwischen den Würfel in beiden Bildern. Diese sind zur Berechnung einer korrekten Transformationsmatrix ausreichend.



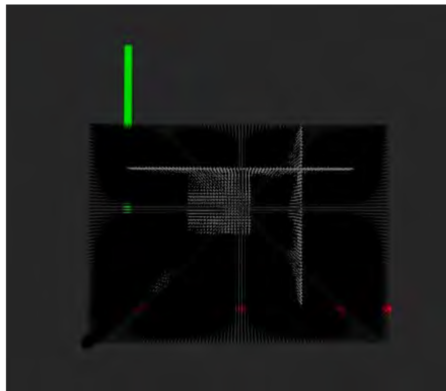
(a)



(b)

Abbildung 17: Das verschobene zweite Bild nach Anwendung der gefundenen Transformation.

Die inverse gefundene Transformationsmatrix wird auf das zweite Bild angewendet und die Datenpunkte werden entsprechend verschoben. Zur Deutlichkeit wurde dem Bild eine Koordinatenachse hinzugefügt.


(a)

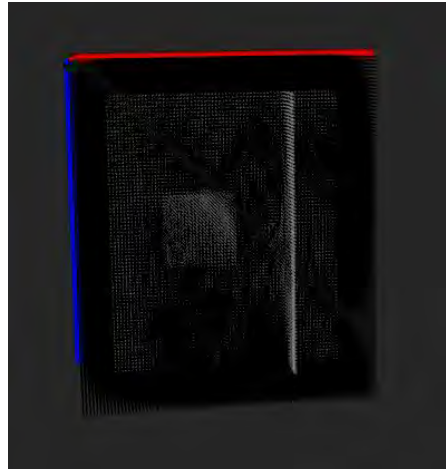
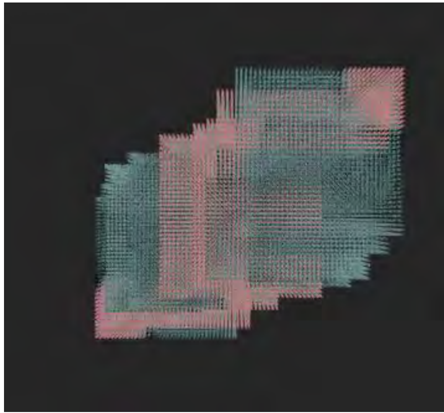

(b)

Abbildung 18: Das zusammengesetzte Ergebnis von vorne und von oben.

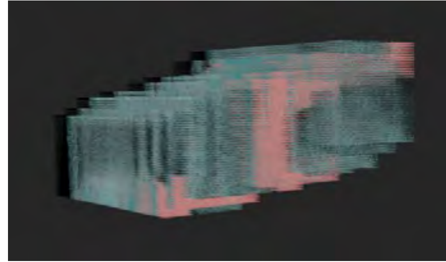
Das erste Bild und das transformierte Bild ergeben gemeinsam das Resultat. Der Würfel und die Ebene sind korrekt zusammengefügt worden. Es ist kein Übergang im Resultat ersichtlich.

3.5.2 Überlapp Test

Zur Bestimmung des nötigen Überlapps der Daten wird ein in x- und y-Richtung zerschnittenes Bild eines Kolben verwendet. Der Kolben besteht insgesamt aus neun Einzelbildern. Für die Visualisierung der Ergebnisse wurde der Code dahingehend geändert, dass zum Schluss nur die komprimierten Bilder zusammengefügt werden. Eine Visualisierung der zusammengesetzten Originaldaten verursachte einen Überlauf des Arbeitsspeichers. Das Programm wurde mit den Parametern Leaf Size = 4.5, Sift Scale = 1.5 und Ransac Threshold = 0.01 aufgerufen.



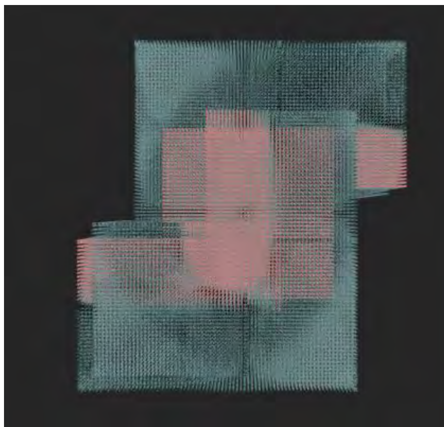
(a)



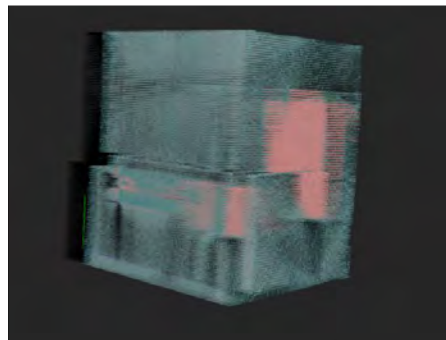
(b)

Abbildung 19: Die Endresultate des zusammengesetzten Kolbens von vorne und links mit 10% Überlapp.

Der erste Test wurde mit 10% Überlapp durchgeführt. Dabei konnte keine richtige Transformation berechnet werden. Das Ergebnis ist dementsprechend ein falsch zusammengesetztes Bild. Ein korrektes Ergebnis wurde mit dieser geringen Überschneidung allerdings nicht erwartet.



(a)

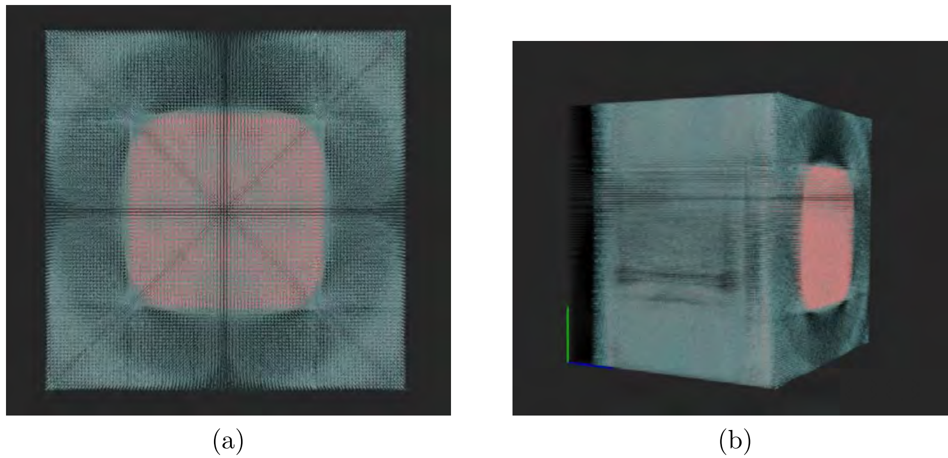


(b)

Abbildung 20: Die Endresultate des Zusammengesetzen Kolbens von vorne und links mit 35% Überlapp.

Der zweite Test wurde mit einer Überschneidung von 35% durchgeführt. Das Endresultat weist bereits Ähnlichkeiten zum gewünschten Ergebnis vor. Für ein zuverlässiges Stitching Ergebnis ist der Wert aber dennoch nicht ausreichend, da ein

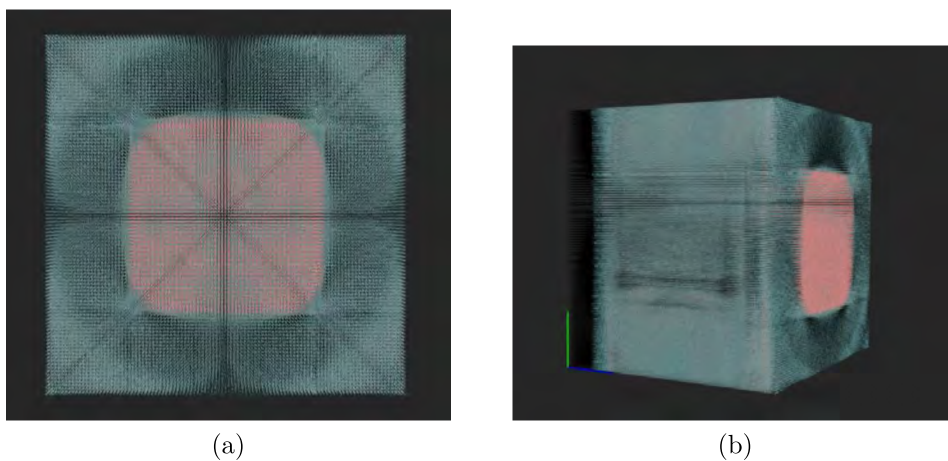
paar Transformationen noch falsch kalkuliert wurden.



(a) (b)

Abbildung 21: Die Endresultate des zusammengesetzten Kolbens von vorne und links mit 50% Überlapp.

Im dritten Test wurde der normale Fall mit 50% Überlapp getestet. Die Bilder sind nun korrekt zusammen gesetzt. Es wurden alle Transformationen korrekt ermittelt.



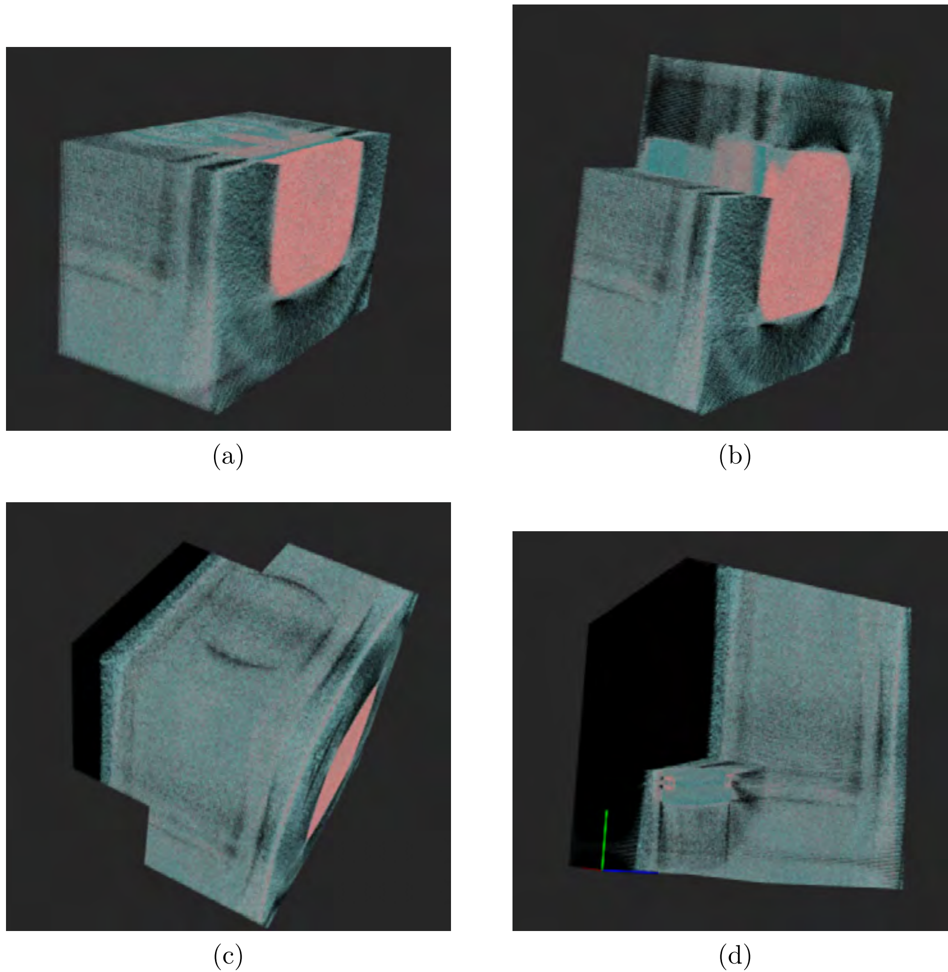
(a) (b)

Abbildung 22: Die Endresultate des zusammengesetzten Kolbens von vorne und links mit 75% Überlapp.

Der letzte Test wurde schließlich mit 75% durchgeführt. Wie zu erwarten, wurde der Kolben auch hier richtig zusammen gesetzt. Der Prozess ist somit ab einem Überlappwert von 50% stabil. Für eine genauere Bestimmung des benötigten Überllap wären zusätzliche Tests im Grenzbereich zwischen 35% und 50% nötig.

3.5.3 Weitere Ergebnisse

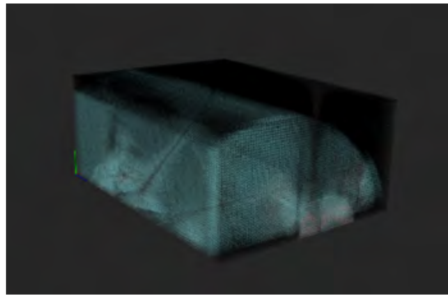
Zur Validierung des Programms wurden zwei weitere Bilder zusammengesetzt. Für die Visualisierung wurden erneut die herunter-skalierten Bilder zusammengefügt.

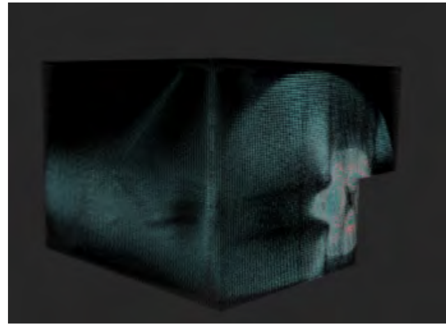


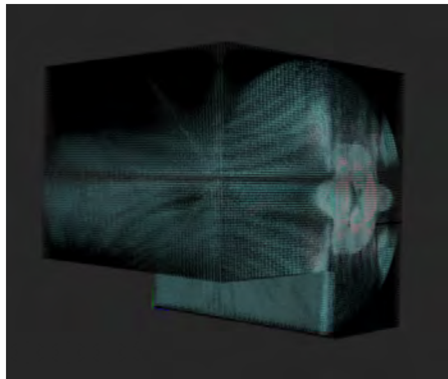
(a) (b) (c) (d)

Abbildung 23: Die Abbildung zeigt das Zusammensetzen des bekannten Kolben in allen Richtungen zerschnitten. Dargestellt ist das Zwischenergebnis nach drei, fünf, acht und zehn Bildern.

Das erste Bild ist der aus dem vorherigen Abschnitt bekannte Kolben. Der Kolben wurde in diesem Fall in allen drei Himmelsrichtungen zerschnitten und besteht aus zwölf Einzelbildern. Durch die zusätzliche Zerschneidung in z-Richtung mussten die Bilder weniger stark komprimiert werden, um ein korrektes Resultat zu erhalten. Das Programm wurde mit den Werten Leaf Size = 3.0, Sift Scale = 0.5 und Ransac Threshold = 0.1 aufgerufen.


(a)


(b)


(c)

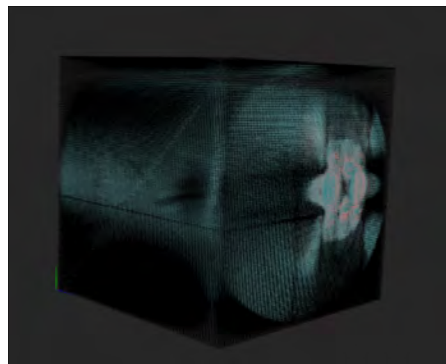

(d)

Abbildung 24: Die Abbildung zeigt das Zusammensetzen eines ausgestopften Kauzes. Dargestellt ist das Zwischenergebnis nach zwei, fünf und sieben Bildern sowie das Endergebnis.

Das zweite Bild ist ein ausgestopfter Kauz. Der Kauz ist aufgrund der starken Komprimierung (von etwa 200.000.000 Bildpunkte auf 200.000 Bildpunkte) nur schemenhaft zu erkennen. Das Bild wurde aus neun Einzelbildern, welche durch Zerschneidung in x- und y-Richtung gewonnen wurden, zusammengesetzt. Als Parameter für das Programm wurden die Werte Leaf Size = 9.5, Sift Scale = 1.5 und Ransac Threshold = 0.01 gewählt. Das Programm hat auch hier die korrekten Transformationen berechnet und den Kauz wie gewünscht zusammengefügt.

4 Fazit

Im Rahmen dieser Arbeit wurde ein Verfahren für das dreidimensionale Image Stitching vorgestellt und implementiert. Zum besseren Verständnis wurde zuerst ein grundlegender zweidimensionaler Stitcher implementiert. Für die Schlüsselstellen-Erkennung und deren Beschreibung mit einem Deskriptor wurden etablierte Algorithmen vorgestellt. Es wurde auf die Theorie hinter den verwendeten Verfahren eingegangen und die Matchverifikation und Modellerkennung dargestellt.
Die Implementierung wurde mit Beispieldaten aus zerschnittenen CT-Scans getestet und die Ergebnisse vorgestellt. Der Stitcher konnte die getesteten Daten korrekt zusammensetzen. Das Ziel der Implementierung eines Verfahren zum Stitchen von dreidimensionalen Daten ist damit erfüllt. Im Hinblick auf die durchgeführten Überlapp Tests wären weitere Tests im Grenzbereich zwischen 35% und 50% wünschenswert. Hierdurch könnte der exakte minimal benötigte Überlapp bestimmt werden. Dieser ist vermutlich ebenso von der gewählten Kompressionsrate abhängig. Auch hierzu wären weitere Tests wünschenswert.
Die Tests zeigten auch, dass das Verfahren sehr viel Rechenleistung und Arbeitspeicher benötigt. Trotzdem dauerten die Berechnungen lange. Eine Dauer von mehren Stunden pro Stitching Vorgang war hier die Regel. Gerade im Hinblick auf die Verwendung in Echtzeitsystemen wie etwa in Augmented Reality Brillen sind daher erhebliche Effizienzsteigerungen nötig. Vor allem der Matchingprozess selbst dauerte lange, bei nur etwa 35% Prozessorauslastung. Hier könnte eine Implementierung mit mehreren Threads eine erheblich Verkürzung der Gesamtberechnungszeit erwirken. Sollte der Stitcher für andere Daten verwendet werden, so wäre auch eine Adaption des im zweidimensionalen Bereich üblichen Blendings zum Ausgleich der Helligkeitsunterschiede als abschließenden Schritt denkbar.
Zusammengefasst lässt sich feststellen, dass für die Registrierung von dreidimensionalen Daten theoretisch belegte und zuverlässige Verfahren vorhanden sind. Mit der Point Cloud Library PCL ist auch eine breite Sammlung an Implementierungen der einzelnen Algorithmen zur Verwendung vorhanden. Der Forschungsschwerpunkt in Zukunft kann damit auf die Optimierung dieser, besonders im Hinblick auf die Effizienz, gesetzt werden.

Literatur

[1] D. Milgram. Computer methods for creating photomosaics. *IEEE Transactions on Computers C-24 (11)*, pages 1113–1119, 1975.

[2] I. Zoghlami, O. Faugeras, and R. Deriche. Using geometric corners to build a 2D mosaic form a set of images. *Proceedings of the International Conference on Computer Vision and Pattern Recognition, Puerto Rico*, pages 1113–1119, 1997.

[3] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. Technical report, Department of Computer Science, University of British Columbia, Vancouver, Canada, January 2004.

[4] 3D Scan mit Kinect, Windows Hardware Dev Center. URL `https://developer.microsoft.com/de-de/windows/hardware/3d-print/scanning-with-kinect`. [Online, besucht: 2019-09-012].

[5] AR auf dem Vormarsch: Warum viele Produkte künftig 3D-fähig sein müssen. URL `https://t3n.de/news/ar-vormarsch-3d-faehige-produkte-1024319/`. [Online, besucht: 2019-09-012].

[6] Virtual Reality auf dem Vormarsch. URL `https://messe-muenchen.de/de/magazin/magazin/menschen/virtual-reality-auf-dem-vormarsch/`. [Online, besucht: 2019-09-012].

[7] 3D-Druck-Technologien in Deutschland auf dem Vormarsch. URL `https://industrie.de/technik/3d-druck-technologien-in-deutschland-auf-dem-vormarsch/`. [Online, besucht: 2019-09-012].

[8] Adel Ebtsam, Elmogy Mohammed, and Elbakry Hazem. Image Stitching based on Feature Extraction Techniques: A Survey. Technical report, Mansoura University, Egypt, August 2014.

[9] Ms. Vrushali S. Sakharkar and Prof. Dr. S. R. Gupta. Image Stitching Techniques-An overview. Technical report, PRMIT & R, Badnera, Amravati, April 2013.

[10] Opencv: Feature matching, . URL `https://docs.opencv.org/trunk/dc/dc3/tutorial_py_matcher.html`. [Online, besucht: 2019-09-01].

[11] Adel Ebtsam, Elmogy Mohammed, and Elkbakry Hazem. Image Stitching System Based on ORB Feature-Based Technique and Compensation Blending. Technical report, Mansoura University, Egypt, 2015.

[12] Tomas Sauer. Skript zur Vorlesung Einführung in die Signal- und Bildverarbeitung. 2012. Universität Passau.

[13] Martin A. Fischler and Robert C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. Technical report, Artifical Intelligence Center, SRI International, Menlo Park, 1981.

[14] Anqi Xu and Gaurav Nami. SURF: Speeded-Up Robust Features. Technical report, McGill University, December 2008.

[15] OpenCV: cv::DrawMatchesFlags Struct Reference, . URL `https://docs.opencv.org/3.4/de/d30/structcv_1_1DrawMatchesFlags.html#a13278be2b5fb496f99e6cce707008bb6ac22e71af97a4741038f2fc47437371b8`. [Online, besucht: 2019-09-02].

[16] Matthew Brown and David G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. Technical report, Department of Computer Science, University of British Columbia, Vancouver, Canada, December 2006.

[17] Documentation - Point Cloud Library (PCL): The PCL Registration API, . URL `http://pointclouds.org/documentation/tutorials/registration_api.php`. [Online, besucht: 2019-09-08].

[18] Flint Alex, Dick Anthony, and van den Hengel Anton. Thrift: Local 3D Structure Recognition. Technical report, The University of Adelaide, North Terrace, 2007.

[19] S. Tomabard, L. Salti, and L. Di Stefano. Unique Signature of Histograms for Local Surface Description. Technical report, CVLab- DEIS, University of Bologna, Italy, September 2010.

[20] S. Tomabard, L. Salti, and L. Di Stefano. A Combined Texture-Shape Descriptor For Enhanced 3D Feature Matching. Technical report, University of Bologna, Italy, September 2011.

[21] Boost.smartptr: The smart pointer library. URL `https://www.boost.org/doc/libs/1_71_0/libs/smart_ptr/doc/html/smart_ptr.html#shared_ptr`. [Online, besucht: 2019-09-03].

[22] Point cloud library (pcl): pcl::pointxyzrgb struct reference, . URL http://docs.pointclouds.org/1.7.0/structpcl_1_1_point_x_y_z_r_g_b.html. [Online, besucht: 2019-09-13].

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig und unter ausschließlicher Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift