Abbildung 1: Visualisierung der markierten Features (links) und die berechnete Homography mit Matches (rechts).

Abbildung 2: Visualisierung der Rektifizierung (mitte) des rotierten Bildes (links). Die detektierten Unterschiede sind im Originalbild durch Rechtecke markiert (rechts).

CV/task2 — Spot the Difference

Ziel dieser Aufgabe ist es eine simple Feature Matching Pipeline zu implementieren, die Unterschiede in ähnlichen Bildern finden kann. Der erste Schritt besteht aus der Detektion von Feature Points und der Berechnung der dazugehörigen Feature Descriptors. Da wir auch leicht rotierte Bilder handeln möchten, rektifizieren wir unsere Bilder mit einer Homography. Danach können wir die Unterschiede zwischen den zwei aufeinander projizierten Bildern berechnen und markieren.

Für die Feature Detektion verwenden wir einen Harris Detector. Da Harris nicht skalierungsinvariant ist, verwenden wir den Laplacian Detector für die Schätzung des Scales (Harris-Laplacian Detector). Als Bonus Aufgabe implementieren wir den FAST Algorithmus für Feature Detektion.

Feature Points repräsentieren “interessante” Punkte in einem Bild. In unsere Aufgabe

haben wir zwei Bilder der gleichen Szene und wollen Feature Point Korrespondenzen zwischen den ersten und zweiten Bild finden (siehe Abbildung 1-links). Um solche Matches zu finden wird jeder unserer Feature Points mit einem sogenannten **Feature Descriptor** beschrieben. Nach dem Matchen wird eine sogenannte **Homography** (Abbildung 1-rechts) mittels der besten Matches geschätzt. Mittels dieser Transformation kann dann ein Bild auf das andere transformiert werden, sodass sich die beiden Bilder gegenseitig überlappen (Abbildung 2-links). Im letzten Schritt können dann mittels den überlappenden Bildern die Unterschiede berechnet werden (Abbildung 2-rechts).

1 Einleitung

Das Beispiel ist in mehrere Unteraufgaben gegliedert, wobei jede Unteraufgabe auf der Vorherigen aufbaut. Der prinzipielle Ablauf besteht aus den folgenden Schritten:

- Harris-Laplace Feature Detektion (5 Punkte)
  - Erstellen der Scale Space Pyramid - `buildOctavesPyramid` (2 Punkte)
  - Harris Corner Detektion - `calculateHarrisCornerMeasure` (1.0 Punkt)
  - Harris Non-Maxima Suppression - `nonMaximaSuppressionHarris` (0.5 Punkte)
  - Feature Detektion - `detectFeatures` (1.5 Punkte)
- BinBoost Feature Descriptors (6 Punkte)
  - Gradient Orientation und Magnitude des Grauwertbildes - `computeGradientOrientationsAndMagnitudes` (0.5 Punkte)
  - Oriented Gradients des Grauwertbildes - `computeGradients` (2 Punkte)
  - Integral Images - `computeIntegrals` (0.5 Punkte)
  - Orientation der Feature-Points - `computeOrientation` (2 Punkte)
  - Weak Learners Response - `computeWLResponse` (0.5 Punkte)
  - Patch Rectification - `rectifyPatch` (0.5 Punkte)
- Image Matching (2 Punkte)

  - Feature Matching - `matchFeatures` (1 Punkte)
  - Berechnen der Homography - `calculateHomography` (0.5 Punkte)
  - Transformation des Bildes - `transformImage` (0.5 Punkte)
- Unterschiede Berechnen (3 Punkte)
  - Berechnen der Unterschiede - `computeDiffs` (1 Punkt)
  - Markieren der Differenzen - `markDiffPoints` (2 Punkte)
- Bonus: FAST Feature Detektor (3 Punkte)

Diese Aufgabe ist mit Hilfe von OpenCV 2.4.9[1] zu implementieren. Nutzen Sie die Funktionen, die Ihnen OpenCV zur Verfügung stellt und achten Sie auf die unterschiedlichen Parameter und Bildtypen.

[1] `http://opencv.org/`

2 Harris-Laplace Feature Detektion (5 Punkte)

Für die Feature Detektion verwenden wir den Harris-Laplace [2] Detektor. Dieser Task besteht aus dem Erstellen einer Scale Space Pyramide (Funktion `buildOctavesPyramid`), berechnen der Harris Corner Response (Funktion `calculateHarrisCornerMeasure`), Non-Maxima Suppression (Funktion `nonMaximaSuppressionHarris`) und berechnen des Scales der die Difference of Gaussians (DoG) Response maximiert (Funktion `harrisLaplaceFeatureDetection`).

Die Idee hinter unserer Feature Detektion ist es den Harris Corner Detektor mit der Gaussian Scale Space Repräsentation zu kombinieren. Dadurch können Harris Corner unabhängig vom Scale des Bildes bzw. der Objekte in den Bildern detektiert werden. Die Scale Space Repräsentation eines Bildes ist eine Bildpyramide, die aus mehreren Oktaven besteht. Ein Oktav ist im wesentlichen ein skaliertes Bild (z.B. um den Faktor 2). Ein Oktav besteht aus mehreren so-genannten Layern, die Gauss-geblurrte Versionen des Eingabebildes sind. Layer mit höheren Scale sind stärker geblurred. Sie werden mit einem Gauss-Filter mit höherer Standardabweichung geblurred. Auf jeden dieser Layer (für alle Oktave) wird der Harris Corner Detektor angewendet um Feature Points zu finden.

2.1 Erstellen der Scale Space Pyramid - `buildOctavesPyramid` (2 Punkte)

In diesem Schritt erstellen wir eine Scale Space Pyramide (siehe Abbildung 3). Diese Pyramide besteht aus mehreren so-genannten “Oktaven”. Oktaven sind im wesentlichen skalierte Kopien des Bildes (z.B. Faktor zwei). Jede Oktave besteht aus mehreren Layern (Scales). Layer sind Gauss-gefilterte Bilder mit unterschiedlicher Standardabweichung. Für Scale Space Detektion brauchen wir die 2. Ableitung dieser Layer (Laplacian of Gaussians (LoG)). Da LoG allerdings rechenaufwändig ist, verwenden wir eine einfachere Approximation, die DoG genannt wird. DoG erreicht in der Praxis eine ähnliche Genauigkeit wie LoG. DoG wird berechnet indem man die absolute Differenz zwischen zwei benachbarten Layer berechnet.

Die Gauss-Pyramide startet mit einem Bild der gleichen Größe wie das Eingabebild I. Die Pyramide hat eine Tiefe von $numOctaves = 6$ Oktaven. Alle Bilder (Layer) in einer Oktave haben eine Größe von 0.5 Mal der Größe der Bilder in der vorherigen Oktave. Daher haben Bilder in der letzten Oktave eine Größe von $2^{-numOctaves+1} = 2^{-5}$ Mal der Größe des Eingabebildes. Jede Oktave besteht aus $numLayers + 3 = 7$ Bildern (Layer) B_l, die Gauss-geblurrte Bilder des skalierten Grauwertbildes sind. Die Standardabweichung σ_l (Scale) des Gauss-Kernels für Layer l außer dem ersten (!) ist definiert als:

$$\sigma_l = \sigma_0 \cdot k^l, \tag{1}$$

wobei $\sigma_0 = 1$ und $k = 2^{\frac{1}{numLayers}}$. Um den Rechenaufwand zu minimieren, blurren wir die Bilder inkrementell vom vorherigen Layer der Gauss Bild Pyramide. Um das korrekte

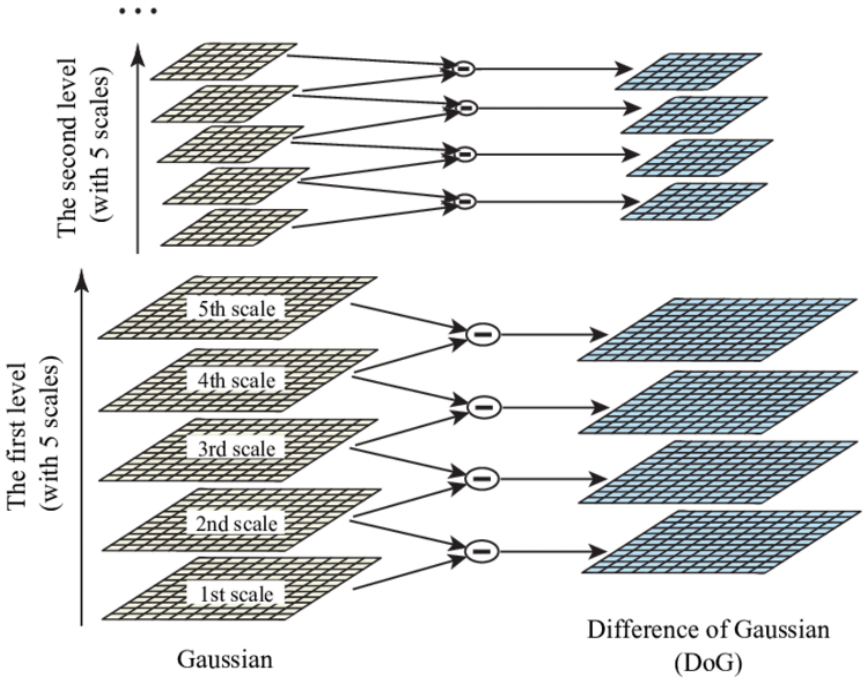

Abbildung 3: Schematische Illustration der Gauss- und DoG Pyramide [1].

inkrementelle $\widehat{\sigma_l}$ zu berechnen, wenden wir folgende Eigenschaft der Gauss-Faltung an. Wenn wir zwei Gauss Kernel mit σ_1 und σ_2 falten, hat der resultierende Gauss Kernel eine **Varianz** von $\sigma_{1+2}^2 = \sigma_1^2 + \sigma_2^2$. Daher ist unser inkrementelles $\widehat{\sigma_l}$ definiert als: $\widehat{\sigma_l} = \sqrt{\sigma_l^2 - \sigma_{l-1}^2}$.

Die Größe des Kernels für eine Faltung mit dem Gauss Kernel (die Gauss-Verteilung ist definiert auf ganz $\mathbb{R}$) ist definiert als $3 \times \sigma$ in allen Richtungen vom Zentrum des Kernels:

$$\widehat{size_l} = \lceil \widehat{\sigma_l} \cdot 3 \rceil \cdot 2 + 1 \tag{2}$$

Um den Response B_l des Layers l zu berechnen, blurren wir den Response des vorherigen Layers B_{l-1} auf der gleichen Oktave o mit dem Gauss-Kernel $G(\cdot)$ mittels Faltung: $B_l = B_{l-1} * G(\widehat{\sigma_l})$, $1 \leq l \leq numLayers + 2$. Um den ersten Layer ($l = 0$) einer Oktave zu berechnen, resizen wir das $numLayer$-tste Bild der vorherigen Oktaven (dieser Layer sollte ein σ von 2 haben, da $\sigma_{numLayers} = 2$). Um den ersten Layer der ersten Okaven zu berechnen, blurren wir das Eingabebild mit einem Gauss-Filter von σ_0, also $B_0 = I * G(\sigma_0)$. Mit diesen geblurrten Bildern berechnen wir die DoG Responses D_l mittels $D_l = |B_{l+1} - B_l|$.

Die DoG Pyramide hat die gleiche Anzahl der Oktaven, allerdings einen Layer weniger pro

Oktave (Abbildung 3). Der Grund dafür ist, dass wir immer 2 benachbarte Layer der Gauss Pyramide verwenden um einen Layer der DoG Pyramide mittels absoluter Differenz zu berechnen.

Die Implementierung der Erstellung der Pyramiden soll in der Funktion `buildOctavesPyramid` gemacht werden. Wir iterieren durch alle Oktaven. Zuerst berechnen wir uns den ersten Layer der Oktave in der Gauss Pyramide (Parameter `pyramid`) entweder mittels resizen des $numLayer$-tsten Layer der vorherigen Oktave oder mittels Blurren des Eingabebildes i mit dem entsprechenden Gauss-Filter (`l = 0`, `octave=0...numOctaves-1`). Zum Schluss fügen wir den berechneten Gauss Layer zu den `layers` vector hinzu.

Danach berechnen wir alle weiteren Layer der Gauss Pyramide der entsprechenden Oktaven ($1 \leq l \leq numLayers + 2$). Außerdem berechnen wir die absolute Differenz zum vorherigen Layer und fügen diese Differenz zu dem `DOG_layer` vector hinzu. Zum Schluss fügen wir den vector von Gauss und DoG layers in die entsprechenden pyramid vectoren ein.

Hilfreiche OpenCV Funktionen:

- `cv::GaussianBlur(...)`
- `cv::absdiff(...)`
- `cv::resize(...)` mit Interpolation $INTER_AREA$

2.2 Harris Corner Detektion - `calculateHarrisCornerMeasure` (1.0 Punkt)

Das Harris Corner Maß (auch Harris Response) definiert die “cornerness” eines Punktes in einem Bild. Die Nachbarschaft von einem Corner ändert sich stark in zwei unterschiedlichen Richtungen (im Vergleich dazu ändert sich eine Edge nur stark in eine Richtung). Die Harris Response ermöglicht es Corner zu detektieren. Je höher die Response, umso höher die “cornerness”. Für jede Bild Position (x, y) ist die so-genannte Autocorrelation Matrix definiert als

$$M(x,y) = \sum_{p,q} w(p,q) \cdot \begin{bmatrix} I_x^2(p,q) & I_x I_y(p,q) \\ I_x I_y(p,q) & I_y^2(p,q) \end{bmatrix}. \tag{3}$$

Um M zu berechnen, berechnen wir zuerst die Ableitungen I_x und I_y des Bilder I und multiplizieren diese mit einem Skalierungsfaktor σ_D. Um die Ableitungen zu berechnen verwenden wir einen Sobel Filter mit Kernel Größe 1. Daher, $I_x = (I * [-1, 0, 1]) \cdot \sigma_D$ und

$I_y = (I * [-1,0,1]^\top) \cdot \sigma_D$. Mittels I_x und I_y können wir einfach I_x^2, I_y^2 und $I_x I_y$ mittels elementweiser Multiplikation berechnen.

Die Indices (p,q) beschreiben eine Nachbarschaft um den Punkt (x,y) und

$$w(p,q) = \frac{1}{2\pi\sigma_I^2} \cdot \exp - \frac{p^2+q^2}{2\sigma_I^2} \tag{4}$$

ist eine Gewichtungsfunktion. Diese Funktion ist eine Gauss Funktion. Daher entspricht die Multiplikation mit der Gewichtungsfunktion einer Faltung der quadrierten Ableitungen (I_x^2, I_y^2, $I_x I_y$) mit einem Gauss Kernel mit Standardabweichung σ_I und einer Kernel Größe wie in Formel (2).

Die Harris Corner Response kann nun mittels der Matrix M folgendermaßen berechnen werden:

$$R(x,y) = \det M - 0.04 \cdot \mathrm{trace}^2(M), \tag{5}$$

wobei

$$\det M = M_{1,1}(x,y) \cdot M_{2,2}(x,y) - M_{1,2}(x,y) \cdot M_{2,1}(x,y) \tag{6}$$

$$\mathrm{trace}(M) = M_{1,1}(x,y) + M_{2,2}(x,y). \tag{7}$$

Ihre Aufgabe ist es nun die Harris Response auf jeder Bild Position in der Funktion `calculateHarrisCornerMeasure`) zu berechnen. Um die Ableitungen des Bildes zu berechnen, verwenden Sie die OpenCV Funktion `Sobel(...)` mit border type `BORDER_REPLICATE`, Kernel Größe 1 und Datentyp CV_32F. Um das Bild mittels Gauss Kernel zu blurren, verwenden Sie die Funktion `GaussianBlur(...)` mit $BORDER_REPLICATE$.

2.3 Harris Non-Maxima Suppression - `nonMaximaSuppressionHarris` (0.5 Punkte)

Da Corner unsere Feature Points sind, wollen wir ähnliche Detektionen (Near-Duplicates) entfernen. Um dies zu erreichen, wenden wir Non-Maxima Suppression auf die Harris Response an. Zuerst betrachten wir nur "gute" Corners Responses als Feature Points, indem wir nur Responses über einen Threshold t betrachten. t berechnen wir folgendermaßen:

$$t = t_c \cdot \max(R), \tag{8}$$

wobei $t_c = 0.01$ ein Threshold ist und $\max(R)$ das Maximum der Response Matrix R (`cornerness`) ist. Wir setzen alle Pixel die kleiner als der Threshold t ist auf 0. The resultierende gethresholdede Matrix nennen wir R_t.

Um nun Non-Maxima Suppression zu implementieren, dilaten wir die Matrix R_t mit einem 3×3 rechteckigen Strukturelement. Wenn man ein Grauwertbild dilated, wird jeder

Pixel auf das Maximum der Nachbarschaft, die durch das Strukturelement definiert wird, gesetzt. Diese dilatete Matrix nennen wir R_d.

Nach dieser Dilation können wir überprüfen ob ein Pixel in der gethresholdeden Matrix R_t dem Wert, der in der Dilateten Matrix R_d an der gleichen Stelle steht, entspricht. Nur diese Punkte sind lokale Maxima und werden daher als Feature Points verwendet. Wir setzen diese Punkte zu einem Wert > 0 in der `local_maxima` Matrix und alle anderen Punkte zu 0.

Helpful OpenCV functions:

- `cv::minMaxLoc(...)`
- `cv::threshold(...)`
- `cv::dilate(...)`
- `cv::compare(...)`
- `cv::bitwise_and(...)`

2.4 Feature Detektion - `detectFeatures` (1.5 Punkte)

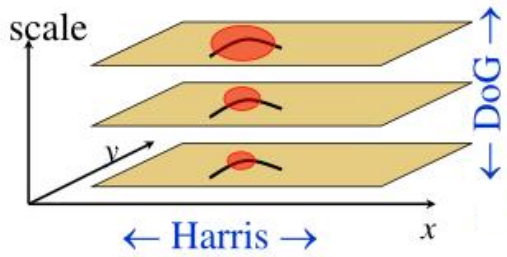

Abbildung 4: Überblick über unseren Harris-Laplace Detektor.

In diesen Schritt machen wir machen wir unseren Feature Detector skalierungsinvariant indem wir den Harris Detector mit unserer DoG Pyramide kombinieren und Features auf mehreren Scales detektieren (Harris Laplacian) (Abbildung 4). Um dies zu erreichen, bauen wir auf unseren vorher definierten Funktionen `buildOctavesPyramid`, `calculateHarrisCornerMeasure` und `nonMaximaSuppressionHarris` auf. Die Idee hinter dem Detector ist es nun Harris Corner in mehreren Layern der Gauss Scale Space Pyramide zu detektieren. Um Skalierungsinvarianz zu erreichen, prüfen wir außerdem ob

der Corner ein lokals Maximum in der DoG Pyramide ist, indem wir prüfen ob die DoG Response des Corners am aktuellen Scale größer ist als die Response auf dem vorherigen und nächsten Scale.

Genauer gesagt iterieren wir durch alle unsere Oktaven o und für jede Oktave iterieren wir über alle Layer l, die einen Vorgänger und Nachfolger in der DoG Pyramide haben ($1 \leq l \leq numLayers$). Für den aktuellen Oktaven o und Layer l müssen wir nun jenen Layer finden auf dem wir die Harris Corner berechnen. Per Definition ist dieser Layer zwei Scales unter dem aktuellen Layer l. Um diesen Layer zu finden, unterscheiden wir 3 Fälle:

- Wenn $l \geq 2$ können wir den Layer einfach mittels Zugriff auf die Gauss Pyramide `pyramid` in der aktuellen Oktave o und Layer $l-2$ bekommen.

- Wenn $l = 1$ und $o \geq 1$, können wir den Layer mittels Resizen eines Layers vom vorherigen Oktaven bekommen. Wir resizen den $numLayer - 1$-ten Layer des $o-1$-ten Oktaven um einen Faktor von 0.5 um ihn auf die gleiche Größe wie den aktuellen Layer zu verkleinern.

- Wenn $l = 1$ und $o = 0$, lassen wir den Fall aus.

Auf diesen Layer berechnen wir dann den Harris Response (`calculateHarrisCornerMeasure`) und machen anschließend Non-Maxima Suppression (`nonMaximaSuppressionHarris`). Für Corner Detektion setzen wir $\sigma_i = 2^{\frac{l}{numLayers}}$ und $\sigma_d = \sigma_i \cdot 0.7$.

Am Schluss prüfen wir ob unsere Corner auch lokale Maxima im Scale Space sind. Um dies zu erreichen, verwenden wir den vorherigen DoG Layer $l-1$, den aktuellen DoG Layer l und den nächsten DoG Layer $l+1$ des aktuellen Oktaven o. Wir iterieren durch alle Non-Maxima (x, y) unserer Harris Cornerness Matrix und prüfen ob diese Position auch ein lokales Maxima in der DoG Pyramide ist. Wir prüfen $D_l(x,y) > D_{l-1}(x,y) \wedge D_l(x,y) > D_{l+1}(x,y)$ und ob die DoG Response größer als ein Threshold ist, nämlich $D_l(x,y) \geq$`DOG_thresh`, wobei $D_l(x,y)$ die DoG Response auf Layer l auf Position (x,y) ist.

Um die Koordinaten des Keypoints zu finden, transformieren wir die Koordinaten des aktuellen Pixels der Cornerness Matrix auf die Größe des Ursprungsbildes (jede Oktave hat die halbe Größe der vorherigen Oktave!). Daher skalieren wir die Koordinaten x und y folgendermaßen:

$$x_i = x \cdot 2^o, y_i = y \cdot 2^o, \tag{9}$$

wobei x_i, y_i die Koordinaten im orignalen Bild sind, und x, y sind die Koordinaten im kleineren Bild im aktuellen Oktaven o. Die Größe s des Keypoints auf dieser Position berechnen wir wie folgt:

$$s = 3 \cdot 2^o \cdot \sigma_i \cdot 2. \tag{10}$$

Außerdem setzen wir den Winkel des Keypoints zu 0, die Response auf den Harris Cornerness Response und die Oktave zu $o \cdot (numLayers + 1) + l$.

Der aktuelle Punkt ist ein Keypoint nur wenn folgende Bedingungen erfüllt sind:

- der ganze Keypoint ($\pm \frac{s}{2}$ in allen Richtungen) ist innerhalb des Bildes
- der Wert im aktuellen DoG Layer ist größer als im vorherigen und nachfolgenden Layer (DoG Non-Maximum Suppression).
- die DoG Response ist größer als der DoG Threshold `DOG_thresh`, der als Konstante am Anfang der Funktion definiert ist.

Für alle Keypoints erstellen wir ein `cv::Keypoint` Objekt mit den entsprechenden Parametern (x_i, y_i, s, die Cornerness des Punktes und der aktuelle Oktave-/Layer-Index $o \cdot (numLayers + 1) + l$). Des Weiteren setzen wir die Orientierung des Keypoints zu 0. Wir fügen alle Keypoints zum features vector hinzu.

3 BinBoost Feature Descriptors (6 Punkte)

Nachdem wir unsere Keypoint Detektionen im Bild detektiert haben, ist unser Ziel nun diese zu beschreiben. Diese Beschreibung sollte invariant bezüglich Skalierung, Rotation und kleineren Distortions sein. Um solche Funktion zu finden, die Keypoint Detektionen beschreibt, können sogenannte gelernte Weak Learners verwendet werden. Dies ist die grundlegende Idee hinter dem BinBoost Descriptor [4]. Für eine vollständige Beschreibung wie BinBoost funktioniert, verweisen wir hier auf das Paper "Learning Image Descriptors with Boosting" [4].

Die Datei $binboost_128.bin$ enthält alle gelernten Parameter dieser Weak Learners. Diese Parameter werden vom Framework in der Funktion $readWLsBinForBinboost$ geladen (diese müssen Sie nicht ändern).

Wir unterteilen diesen Teil der Aufgabe in die folgenden Teilfunktionen:

- `computeGradientOrientationsAndMagnitudes`: 0.5 Punkte
- `computeGradients`: 2 Punkte
- `computeIntegrals`: 0.5 Punkte
- `computeOrientation`: 2 Punkte

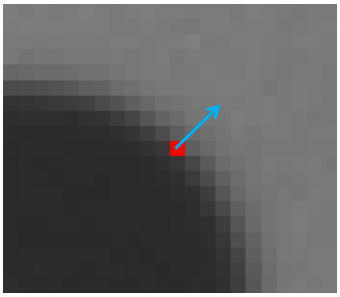

Abbildung 5: Gradienten eines Pixels sind Vektoren, die in die Richtung des stärksten Anstiegs zeigen.

- `rectifyPatch`: 0.5 Punkte
- `computeWLResponse`: 0.5 Punkte

3.1 Gradient Orientation und Magnitude des Grauwertbildes - `computeGradientOrientationsAndMagnitudes` (0.5 Punkte)

Im ersten Schritt unserer Feature Descriptor Pipeline berechnen wir die Gradient Magnitude und Gradient Orientation des Grauwertbildes. Wir können den Gradienten eines Pixels als Vektor interpretieren, der in die Richtung des stärksten Anstiegs zeigt (Abbildung 5).

Wir können nun die x und y Komponenten dieser Vektoren mittels Faltung mit `Sobel` Filtern in x und y Richtung berechnen:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}. \tag{11}$$

I_x und I_y werden mittels Faltung dieser Kernel mit dem Bild I berechnet:

$$I_x = I * S_x, \qquad I_y = I * S_y. \tag{12}$$

Um diese Faltung zu berechnen, verwenden wir die OpenCV Funktion `cv::Sobel(...)`. Weiters verwenden wir CV_32FC1 als Datentyp um die Gradienten als float Bild zu speichern.

Die Gradienten Vektoren auf jeder Pixel Position sind nun durch die I_x und I_y Matrizen repräsentiert. Um die Gradient Magnitude Matrix und Orientation Matrix zu berechnen, brauchen wir die Magnitude und Orientation für jeden Pixel.

Die Magnitude auf Position x, y ist definiert als

$$M(x,y) = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}. \tag{13}$$

Die Orientation θ auf Position x, y ist definiert als:

$$\theta(x,y) = \text{atan2}(I_y(x,y), I_x(x,y)). \tag{14}$$

Weiters soll der Winkel zwischen $(0, 2 \cdot \pi)$ normalisiert werden. Wenn der Winkle daher kleiner als 0 ist, addieren wir $2 \cdot \pi$ zum Winkel, sodass er sich im validen Bereich befindet.

Hilfreiche OpenCV Funktionen

- `std::atan2(...)`
- `std::sqrt(...)`
- `cv::Sobel(...)`

3.2 Oriented Gradients des Grauwertbildes - `computeGradients` (2 Punkte)

Ziel dieser Funktion ist es den Gradienten in 8 Gradient Orientation Bilder zu quantisieren (Abbildung 6). Um dies zu erreichen, verwenden wir die vorher implementierte Funktion `computeGradientOrientationsAndMagnitudes(...)`. Wir iterieren durch alle Pixel (x, y) des Bildes und betrachten nur jene mit hoher Gradient Magnitude für die weitere Verarbeitung. Genauer gesagt, nur wenn die Gradient Magnitude grösër als der Threshold `magnitudeThreshold` ist, quantisieren wir die Orientation. Andernfalls setzen wir alle Gradient Orientation Bilder auf dieser Position zu 0.

Die Grenzen der Gradient Orientation Bins sind definiert als $k \cdot \frac{2 \cdot \pi}{orientQuant}$ für $0 < k < orientQuant$. In unserem Fall ist orientQuant 8, also haben wir 8 Orientation Bins. Um den Gradient Bin für den aktuellen Pixel (x, y) zu berechnen dividieren wir die Orientation (in Bogenmaß) durch die Bin Größe:

$$index(x,y) = \lfloor \frac{\theta(x,y) \cdot orientQuant}{2 \cdot \pi} \rfloor \tag{15}$$

Beachten Sie, dass $index(x, y)$ ein integer ist. Wir casten also von float nach int nach dieser Operation. Darüber hinaus stellen wir sicher, dass $index(x, y)$ zwischen 0 und $orientQuant$ liegt. Wenn $index(x, y) = orientQuant$, setzen wir es zu 0.

Anstatt ein harte Zuweisung zum Gradienten Bin zu berechnen, verwenden wir eine softe Zuweisung zu mehreren Bins. Jeder Gradient beeinflusst 2 benachbarte Bins b

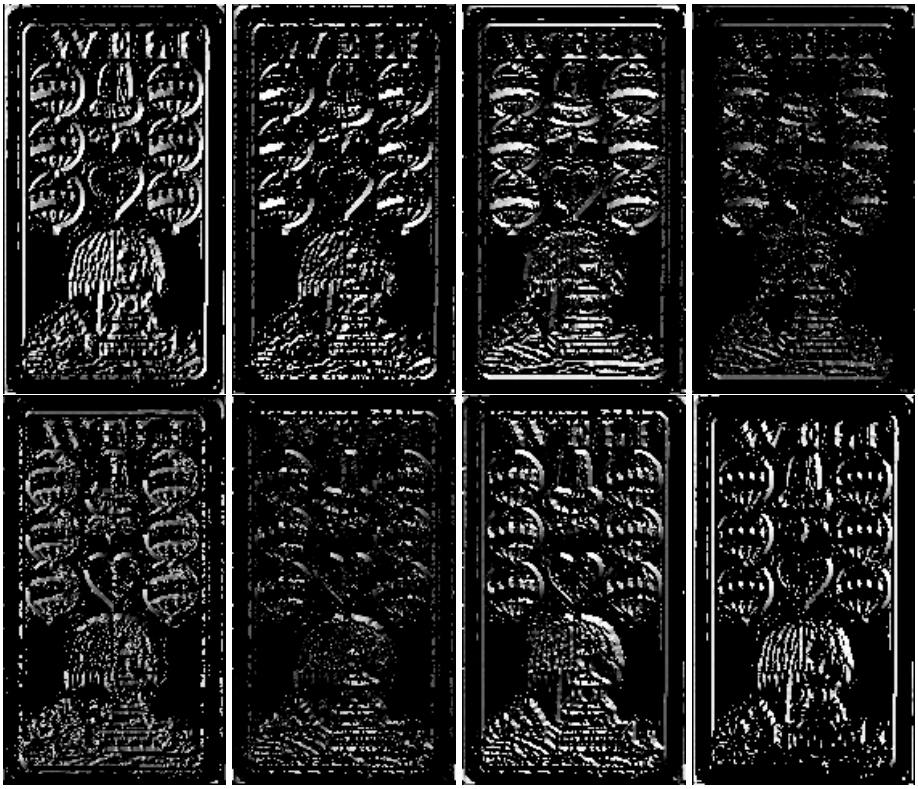

Abbildung 6: Gradienten in all 8 Richtungen.

in allen Richtungen, also $b \in \{index(x,y)-2,\ldots,index(x,y)+2\}$. Daher beeinflusst jeder Gradient 5 Bins in der Gradient Orientation Matrix. Um negative Indices zu vermeiden, normalisieren wir b zwischen 0 und $orientQuant$ mittels $\widehat{b} = \mod(b + index(x,y) + orientQuant, orientQuant)$.

Für alle diese 5 Bins berechnen wir ein Gewicht w, das der cosinus des Winkels zwischen Bin Zentrum und dem Winkel $\theta(x,y)$ ist. Der Winkel des Bin Zentrums $\widehat{b}$ ist definiert als $bc = (\widehat{b}+0.5)\cdot\frac{2\cdot\pi}{orientQuant}$. Wir vermeiden außerdem negative Werte von w da wir das Gewicht wie folgt definieren:

$$w = \max(0, \cos(\theta - bc)). \tag{16}$$

Zum Schluss weisen wir `round`$(w\cdot 255)$ der `gradients` Matrix auf der aktuellen Position (x,y) und dem aktuellen Nachbar Bin $\widehat{b}$ zu.

3.3 Integral Images - `computeIntegrals` (0.5 Punkte)

Integral Images sind nützlich für das Berechnen von Summen von rechteckige Regionen in einem Bild. Solche Summen können damit in konstanter Zeit mit nur 4 Additions-Operationen berechnet werden. Wir brauchen dies später für das Berechnen der Deskriptoren. Die Funktion `computeImages` bekommt als Eingabe einen Vektor von quantisierten Oriented Gradient Bildern des Grauwertbildes. Der Vektor besteht aus `orientQuant` Gradienten in verschiedenen Richtungen. Wir wollen nun ein Integral Image von diesen Bildern berechnen.

Die Definition des Integral Images ist:

$$I(x,y) = \sum_{i=0}^{i<x} \sum_{j=0}^{j<y} H(i,j), \tag{17}$$

wobei I das Integral Image bezeichnet, H das Grauwertbild bezeichnet und (x,y) die Position eines Pixels im Integral Image ist. Die erste Zeile und Spalte des Integral Images I ist 0. Das Integral Image hat eine Größe von $(rows+1)\times(cols+1)$, wobei $rows$ die Anzahl der Zeilen und $cols$ die Anzahl der Spalten in H ist. Eine naïve Berechnung des Integral Images mittels obiger Formel ist sehr rechenaufwändig. Allerdings können wir sehen, dass folgende Gleichung stimmt:

$$I(x,y) = I(x,y-1) + \sum_{i=0}^{i<x} H(i,y-1). \tag{18}$$

Wir können außerdem den Rechenaufwand verringern indem wir den zweiten Term iterativ berechnen. Sei $R(x,y)$ die kumulative Summe der Pixel in Zeile y:

$$R(x,y) = \begin{cases} R(x-1,y) + H(x-1,y-1) & wennx > 0, \\ 0 & andernfalls. \end{cases} \tag{19}$$

Danach können wir mittels $I(x,y) = I(x,y-1) + R(x,y)$ das Integral Image berechnen indem wir alle Pixel nur ein Mal durchgehen.

Mittels Integral Image I des Bildes H kann nun die Summe S der Pixel von Punkt (x_A, y_A) zum Punkt (x_D, y_D) im Bild H in konstanter Zeit unabhängig von der Größe des Rechtecks berechnet werden:

$$S = I(x_A, y_A) + I(x_D+1, y_D+1) - I(x_D+1, y_A) - I(x_A, y_D+1). \tag{20}$$

In der Funktion `computeIntegrals` iterieren wir durch alle Gradienten Bilder im `gradients` vector. Wir berechnen das Integral Image vom i-ten Gradienten Bild in der `Mat integrals[`i`]` Matrix, für $0 \leq i <$ `orientQuant`. Weiters berechnen wir das letzte Bild im `integrals` vector (Index `orientQuant`) als Summe aller vorherigen Integral Bilder.

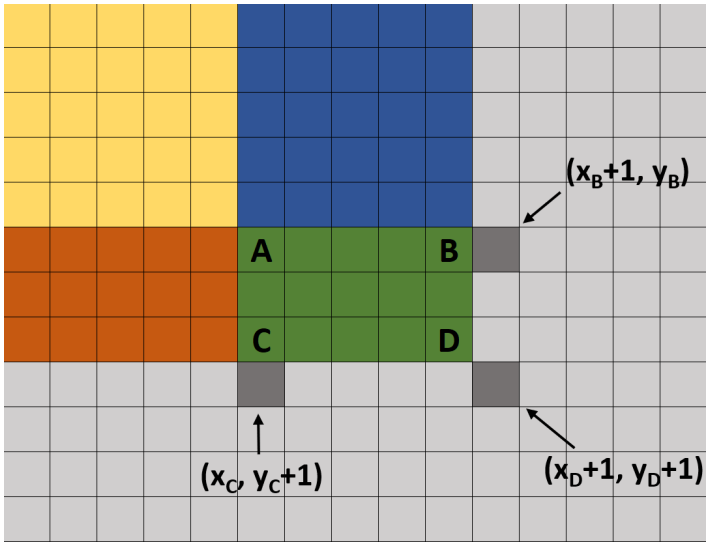

Abbildung 7: Illustration der Summen Berechnung mittels Integral Image.

`Hinweis`: Die Datentypen für unsere Zeilen-Summen Variablen und Integral Images sind integers. Die Gradienten Bilder H sind uint8.

3.4 Orientation der Feature-Points - `computeOrientation` (2 Punkte)

In diesen Schritt, um unsere Features Rotation invariant zu machen, normalisieren wir die Orientierung unserer Keypoints durch die dominante Gradienten Orientierung an diesem Punkt. Um dies zu erreichen, verwenden wir eine vereinfachte Form der SIFT [1] Orientierungs Schätzung (Abbildung 8). Wir berechnen die Gradient Magnitude und Orientierung für eine Region um den Keypoint. Danach erstellen wir ein Histogram von Gradienten Orientierungen. Jeder Pixel votet in einen Histogram Bin mit seiner Gradient Magnitude. Da Pixel im Zentrum stärkere Votes haben sollten, als Pixel am Rand des Patches, gewichten wir die Gradient Magnitude mit einem Gauss-Fenster. Die Keypoint Orientation ist die Orientierung mit den meisten Votes im Histogram.

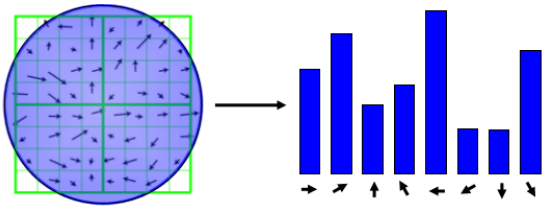

Abbildung 8: Orientierungs-Schätzung mittels berechnen eines Histograms von Gradienten Orientierungen.

Die Basis der Orientation Schätzung sind die Gradient Magnitude und Gradient Orientations, die wir als Eingabe zu unserer Funktion bekommen. Wir iterieren durch alle Keypoints kp und transformieren die Koordinaten in die Koordinaten des aktuellen Oktaven kp_o und Layer kp_l. Der Oktave und Layer kann vom $octave$ Feld des Keypoint Objekts mittels Division und Modulo von $numLayers + 1$ berechnet werden (das wurde bereits für Sie gemacht). Die Koordinaten müssen dann mittels $\frac{1}{2^{kp_o}}$ skaliert werden (das wurde ebenfalls bereits für Sie gemacht).

Danach betrachten wir eine lokale Nachbarschaft um die Keypoint Position x, y von $patchsize \times patchsize$, nämlich von $(x - patchsize/2, y - patchsize/2)$ nach $(x + patchsize/2, y + patchsize/2)$ (wir stellen auch sicher, dass wir nicht out-of-bounds gehen!). In diesem Rechteck wollen wir die dominante Gradienten Orientierung berechnen (Abbildung 8). Für jedes Pixel (i, j) in dieser Nachbarschaft berechnen wir nun den Histogram Eintrag. Der Histogram bin b ist definiert als

$$b = \text{round}\left(\theta_{kp_o,kp_l}(i,j) \cdot \frac{180}{\pi} \cdot \frac{nAngleBins}{360}\right), \tag{21}$$

wobei $\theta_{kp_o,kp_l}(i,j)$ den i,j-ten Eintrag im Orientation Bild auf Oktaven kp_o und Layer kp_l bezeichnet. Wir stellen außerdem sicher, dass b nicht out-of-bounds ist. Genauer gesagt, wenn $b < 0$, addieren wir $nAngleBins$. Wenn $b \geq nAngleBins$ subtrahieren wir $nAngleBins$.

Danach voten wir in das Histogram H mittels

$$H[b] := H[b] + M_{kp_o,kp_l}(i,j) \cdot s(i,j) \tag{22}$$

wobei M_{kp_o,kp_l} das Gradient Magnitude Bild auf Octave kp_o und Layer kp_l bezeichnet. $s(i,j)$ ist eine Gauss Skalierungsfaktor, der Punkte zum Keypoint Zentrum x, y präferiert:

$$s(i,j) = e^{-\frac{(i-x)^2 + (j-y)^2}{2 \cdot \sigma^2}}, \tag{23}$$

wobei $\sigma = 1.8$.

Nachdem wir durch die Nachbarschaft iteriert sind, müssen wir die dominante Gradienten Orientierung unserem Keypoint kp zuweisen. Um dies zu erreichen finden wir den Maximum Bin b_{max} des Histograms H und setzen den Winkel des Keypoints zu $(b_{max} + 0.5) \cdot \frac{360}{nAngleBins}$.

`Hinweis`: Wenn zwei Histogram Bins gleich groß sind, nehmen Sie den ersten Bin.

3.5 Patch Rectification - `rectifyPatch` (0.5 Punkte)

In diesem Abschnitt implementieren wir eine einfache Hilfs-Funktion, die Patches mittels vorher detektierten Keypoint Position (x, y), Scale und Winkel θ, rektifiziert. Wir

Abbildung 9: Zeigt gute Matches zwischen 2 ähnlichen Bildern mittels dem OpenCV Descriptor-Matcher. Jeder Match wird als gefärbte Linie zwischen den Feature-Punkten dargestellt.

implementieren diese Funktion in `rectifyPatch`.

Um einen Patch um den Keypoint kp zu rektifizieren, müssen wir diesen mittels Affiner Transformation M transformieren. Genauer gesagt, definieren wir diese Transformation als:

$$M = \begin{bmatrix} s \cdot \cos(\theta) & -s \cdot \sin(\theta) & (-s \cdot \cos(\theta) + s \cdot \sin(\theta)) \cdot \frac{patchSize}{2} + x \\ s \cdot \sin(\theta) & s \cdot \cos(\theta) & (-s \cdot \sin(\theta) - s \cdot \cos(\theta)) \cdot \frac{patchSize}{2} + y \end{bmatrix}, \tag{24}$$

wobei $s = \frac{kp.size}{patchSize}$ den Skalierungsfaktor der Transformation bezeichnet. Wir setzen s zu 1 wenn $kp.size \leq 0$. θ bezeichnet den Winkel des Keypoints. Wenn $\theta < 0$, setzen wir $\theta = 0$.

Die Ausgabe dieser Funktion ist ein rektifizierter Patch der Größe $patchSize \times patchSize$. `Hinweis`: Winkel sind in Grad gespeichert. std::sin, std::cos brauchen den Winkel in Bogenmaß.

Hilfreiche OpenCV Funktionen

- `warpAffine(...)`, verwenden Sie die Flags, die in der Funktion definiert sind.

Abbildung 10: Der Binboost Descriptor besteht aus rechteckigen Summen auf orientierten Gradienten Bildern. Wir zeigen hier exemplarisch 3 Features (unterschiedliche Rechtecke) auf 3 von 8 Oriented Gradient Bildern.

3.6 Weak Learners Response - `computeWLResponse` (0.5 Punkte)

Der Binboost [4] Descriptor besteht aus rechteckigen Summen auf unseren vorherigen berechneten Oriented Gradient Bildern (Abbildung 10). Der Descriptor lernt mehrere so-genannte "Weak Learners" auf einem Datenset von ähnlichen und nicht-ähnlichen Patches. Jeder "Weak Learner" definiert ein rechteckiges Feature auf einem Oriented Gradient Bild. In Abbildung 10 illustrieren wir 3 solche Weak Learners (in rot, grün und blau). In dieser Funktion implementieren wir die Feature Berechnung für einen einzigen Weak Learner, also die Summe eines Rechtecks.

Formal definieren wir G als den vector von Gradient Orientation Patches mit Größe $(rows \times cols)$, der $W = orientQuant$ unterschiedliche Orientations hat. Der Patch wurde vorher um einen detektierten Feature Point erstellt. d bezeichnet die Orientierung des aktuellen Weak Learners. Weiters sei x_{min}, x_{max}, y_{min}, y_{max} und Threshold t Attribut des Weak Learners. Die Response R dieses Features wird dann wie folgt berechnet:

$$R = \frac{\sum_{i=x_{min}}^{i<x_{max}} \sum_{j=y_{min}}^{j<y_{max}} G_d(i,j)}{\sum_{k=0}^{k<W} \sum_{i=x_{min}}^{i<x_{max}} \sum_{j=y_{min}}^{j<y_{max}} G_k(i,j)} - t \tag{25}$$

Diese Formel ist sehr rechenaufwändig. Glücklicherweise können wir unser Integral Image I verwenden um diese Berechnung mittels Formel (20) zu beschleunigen. I_W ist dabei die Summe aller Integral Images und das letzte Element des Oriented Gradient Patch vectors. Die Formel kann also simplifiziert werden zu:

$$R = \frac{I_d(x_{min}, y_{min}) + I_d(x_{max}+1, y_{max}+1) - I_d(x_{max}+1, y_{min}) - I_d(x_{min}, y_{max}+1)}{I_W(x_{min}, y_{min}) + I_W(x_{max}+1, y_{max}+1) - I_W(x_{max}+1, y_{min}) - I_W(x_{min}, y_{max}+1)} - t \tag{26}$$

In der Funktion $computeWLResponse$ bekommen wir den weak learner WL, die Anzahl an unterschiedlichen Richtungen $orientQuant$ und einen vector von Integral Images der Gradient Orientations des Patches (einschließlich des letzten Summe von allen Layers) als Eingabe. Mittels obiger Formel können wir nun die Response des Weak Learners berechnen. Wir verhindern außerdem Divisionen durch 0. Wenn der Nenner 0 ist, geben wir 0 zurück. Andernfalls geben wir das Ergebnis der Formel zurück. Die Attribute des Weak Learners WL sind $WL.x_min$, $WL.x_max$, $WL.y_min$, $WL.y_max$ und $WL.thresh$.

4 Image Matching (2 Punkte)

In diesem Abschnitt Matchen wir die beiden Bilder aufeinander mittels der vorher berechneten Keypoint Descriptors.

4.1 Feature Matching - `matchFeatures` (1 Punkte)

In diesen Schritt berechnen wir die Matches zwischen unseren Feature Descriptoren. Dieser Teil der Aufgabe soll in der Funktion `matchFeatures(...)` implementiert werden. Ein Match besteht aus zwei Feature Points. Einer dieser Punkte ist im so-genannten Source Image und der zweite Punkt im so-genannten Target Image. Das Matching wird von einem Descriptor-Matcher (BFMatcher) von OpenCV gemacht. Wir verwenden die `knnMatch` Methode um die $k = 2$ besten Matches für jeden Source Descriptor zu finden.

Der Matcher gibt eine Liste von Matches zurück und unsere Aufgabe ist es nun gute von schlechten Matches zu unterscheiden. In unserem Fall ist die Anzahl an Matches $k = 2$, das heißt dass der beste Match und zweitbeste Match für jeden Source Descriptor zurückgeliefert wird. Um die Matches nun robuster zu machen prüfen wir jeden Match ob die Distanz zwischen besten Match und zweitbesten Match hinreichend groß ist. Nur Matches für die folgendes gilt werden zum Output `matches` hinzugefügt:

$$\text{distance}(\text{query}, \text{match}_0) \leq \texttt{match_distance_factor} \cdot \text{distance}(\text{query}, \text{match}_1). \tag{27}$$

Des Weiteren fügen wir die dazugehörigen Source und Target Points zu den entsprechenden Listen für das Source Image und das Target Image. Das bedeutet, dass wir für jeden Match den `queryIdx` Punkt zum source-point-vector `good_matches_source` und `trainIdx` zum target-point-vector `good_matches_target` hinzufügen.

Das Ergebnis dieses Schrittes ist eine große Anzahl an Matches (Abbildung 11).

Abbildung 11: Illustration der guten Matches zwischen zwei ähnlichen Bildern. Die Matches wurden vom OpenCV Descriptor-Matcher erstellt. Jeder Match ist als gefärbte Linie zwischen den 2 matchenden Feature Points dargestellt.

Hilfreiche OpenCV Funktionen:

- `cv::DescriptorMatcher::knnMatch(...)`

4.2 Berechnen der Homography - `calculateHomography` (0.5 Punkte)

Um die Homography zwischen zwei Bildern zu berechnen verwenden wir die Matches des vorherigen Schritts. Wir implementieren diesen Schritt in der Funktion `calculateHomography(...)`. Für die Berechnung verwenden wir die Funktion `cv::findHomography` aus OpenCV. Wir verwenden die Methode `CV_RANSAC` mit einem maximalen Re-Projection Error von 3.

Hilfreiche OpenCV functions:

- `cv::findHomography(...)`

4.3 Transformation des Bildes - `transformImage` (0.5 Punkte)

Dieser Task wird in der Funktion `transformImage(...)` implementiert. Als Eingabe bekommen wir das rechte Bild, das linke Bild und eine Homography Matrix, die das linke

Bild zum rechten Bild transformiert. Das Ziel dieser Funktion ist es das rechte Bild auf das linke Bild zu warpen mittels der OpenCV Funktion `warpPerspective(...)` (Abbildung 12). Das Ergebnis sollte die gleiche Größe haben wie das linke Bild. Wir verwenden außerdem lineare Interpolation (`INTER_LINEAR`).

Abbildung 12: Vergleich zwischen den vergrößerten Bild (links) und dem Source Bild (rechts).

Hilfreiche OpenCV Funktionen:

- `cv::warpPerspective(...)`

5 Unterschiede Berechnen (3 Punkte)

In diesem Abschnitt finden wir die Unterschiede zwischen den 2 registrierten Bildern.

5.1 Berechnen der Unterschiede - `computeDiffs` (1 Punkt)

In diesem Schritt implementieren wir die Funktion `computeDiffs(...)`, die die Unterschiede zwischen den transformierten Bild und dem Source Bild berechnet (Abbildung 12).

Im ersten Schritt machen wir ein Pre-Processing auf beiden Bildern um Noise und Artifakte, die beim Transformieren entstehen, zu entfernen. Daher blurren wir beide Bilder mit einem Gauss Filter von σ. Wir setzen die Kernel Größe zu $\lceil 3 \cdot \sigma \rceil \cdot 2 + 1$.

Im nächsten Schritt berechnen wir die absoluten Differenzen zwischen den geblurrten Bildern B_l und B_r:

$$D = |B_l - B_r|, \tag{28}$$

wobei D die `diffImg` Variable ist. Um die "durchschnittliche" Differenz eines Pixels zu bekommen, der einfachheitshalber, konvertieren wir dieses Bild zu einem Grauwertbild. Von diesem Grauwert Differenz Bild D_g berechnen wir ein gethresholdetes Bild D_t mit dem `threshold` und der Funktion `cv::threshold(...)` mit Flag THRESH_BINARY. Daher ist $D_t = D_g >$ `threshold`. Als Ergebnis bekommen wir ein Binärbild indem jede Komponente einen Unterschied repräsentiert (Output: img_threshold).

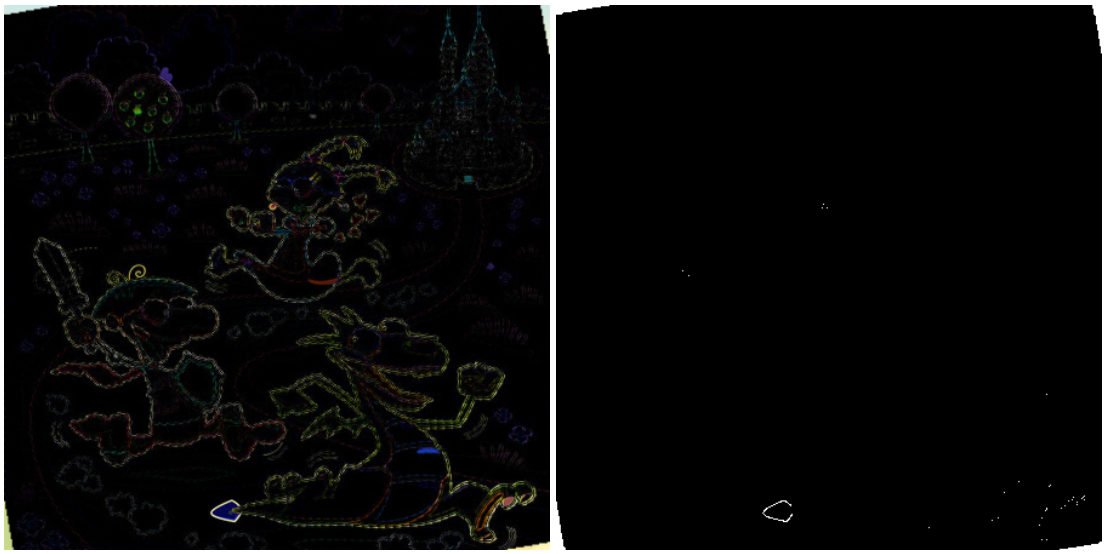

Abbildung 13: Absolute difference image (left). Thresholded image (right).

Um eine Beschreibung der "weißen" Pixel in unserem gethresholdeten Bild zu bekommen, wenden wir einen Canny Edge Detector (`cv::Canny(...)`) und die `cv::findContours(...)` Funktion an. Die resultierende Liste an Contours soll in den contours Ausgabe Parameter geschrieben werden. Für Canny verwenden wir einen `lowThreshold` von 0 und einen `highThreshold` von 255 und eine Kernel Größe von 3. Des Weiteren rufen wir `cv::findContours` mit den detektieren Edges auf und geben `RETR_TREE` als mode Flag und `CHAIN_APPROX_SIMPLE` als method Flag an.

Hilfreiche OpenCV Funktionen:

- `cv::absdiff(...)`
- `cv::Canny(...)`
- `cv::findContours(...)`

5.2 Markieren der Differenzen - `markDiffPoints` (2 Punkte)

Im letzten Schritt markieren wir die Differenzen zwischen den Bildern mittels Rechtecke und Kreisen. In der Funktion `markDiffPoints(...)` bekommen wir die vorher berechnete Contour Liste als Eingabe. Jede Contour repräsentiert eine Differenz zwischen Source und Target Bild. Für jede Contour wollen wir einen Kreis und ein Bounding Rectangle erzeugen, die diese Contour repräsentiert.

Um einen Kreis zu erstellen, brauchen wir das Zentrum der Contour. Wir definieren das Zentrum der Contour als durchschnittliche x- und y- Koordinate der Contour. Formal wird das Zentrum des Kreises $c_i = (x_i, y_i)$ der i-ten Contour als $x_i = \frac{1}{|contour_i|} \sum_{pt \in contour_i} pt_x$ und $y_i = \frac{1}{|contour_i|} \sum_{pt \in contour_i} pt_y$ definiert, wobei $|contour_i|$ die Anzahl der Elemente in der Contour i bezeichnet und pt_x/pt_y die x/y Koordinate des Punktes pt der $contour_i$ sind. Weiters definieren wir den Durchmesser des Kreises als das Maximum der Höhe und der Breite des Bounding Rectangles der $contour_i$ (`cv::boundingRect(...)`). Wie wir in Abbildung 14 sehen, detektieren wir eine große Anzahl an Kreisen für unsere Differenzen.

Da wir eine große Anzahl an ähnlichen (Near Duplicate) Differenz Punkten detektieren, erstellen wir auch ein Bounding Rectangle r für jede Contour, wenn der dazugehörige Kreis innerhalb unserer Größen Constraints ist, nämlich $diameter_i \leq maxDiameter$. Um diese Near Duplicates zu entfernen machen wir eine Form von Non-Maxima Suppression indem wir überlappende Rechtecke mergen. Genauer gesagt definieren wir das Bounding Rectangle r_i für Contour i als minimales einschließendes Rechteck der Contour i (erstellt mittels `cv::boundingRect`), das um 10 Pixel gepadded ist.

Um nun mehrere Near Duplicates zu entfernen, vergleichen wir das Bounding Rect r_i mit allen vorher eingefügten Bounding Rectangles r_j in der Rectangle Liste `diffRectangles`, und prüfen ob r_i mit r_j überlappt. Um dies zu prüfen verwenden wir die Hilfsfunktion `doRectOverlap(...)`. Wenn r_i und r_j überlappen, mergen wir die beiden Rectangles und wiederholen den Prozess. Genauer gesagt, updaten wir das Bounding Rectangle r_i folgendermaßen:

$$x = \min(r_i.x, r_j.x) \tag{29}$$

$$y = \min(r_i.y, r_j.y) \tag{30}$$

$$width = \max(r_i.x + r_i.width, r_j.x + r_j.width) - x \tag{31}$$

$$height = \max(r_i.y + r_i.height, r_j.y + r_j.height) - y. \tag{32}$$

Außerdem entfernen wir das alte Bounding Rectangle r_j von unserer Liste `diffRectangles` und wiederholen den Prozess für das nun vergrößerte Bounding Rectangle r_i. Zum Schluss, wenn keine Überlappungen mehr gefunden werden können, fügen wir r_i zu unserer Liste `diffRectangles` hinzu.

Abbildung 14: Markierte Unterschiede als Kreise (ohne Near Duplicate Detektion).

Wir visualisieren die Ergebnisse dieses Schritts in Abbildung 15.

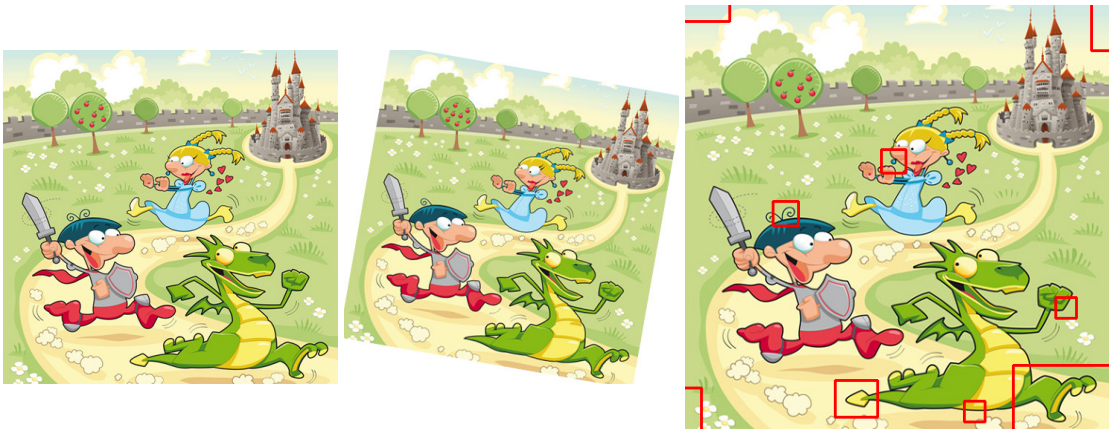

Abbildung 15: Markierte Unterschiede mittels Rechtecken (mit Near Duplicate Detektion).

Hilfreiche OpenCV Funktionen:

- `cv::boundingRect(...)`

5.3 Bonus: FAST Feature Detektor (3 Punkte)

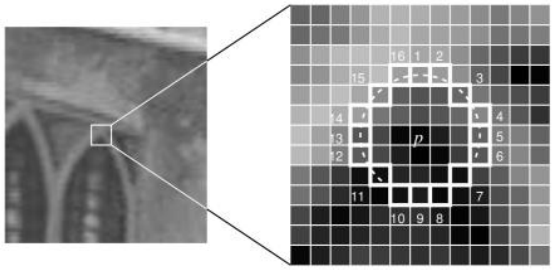

Abbildung 16: Vergrößerung eines Eckpunktes mit dem eingezeichneten Kreis, der für FAST verwendet wird um Pixel als Eckpunkte zu identifizieren. [3]

Dieser Teil der Aufgabe ist in der Funktion `Fast(...)` zu implementieren Die Idee des FAST Features (Features from Accelerated Segment Test) [3] ist, dass ein Eckpunkt über Pixel in seiner Nähe identifiziert werden kann. Dazu untersucht FAST alle Pixel, die auf einem Kreis um das betrachtet Pixel liegen und überprüft, ob diese eine höhere oder niedrigere Intensität als das zentrale Pixel haben. In unserem Falle handelt es sich um einen Kreis mit Radius $r = 3$. Die Koordinaten des Kreises sind im Parameter (`circle`) zu finden. Zu beachten ist, dass der gegebene Kreis um den Koordinaten Ursprung $(0,0)$ definiert ist. Weiters können Sie alle Features die näher als 9 Pixel am Rand des Bildes liegen verwerfen.

Um festzustellen ob ein Punkt $p = (x, y)$ ein guter Kandidat für einen Feature-Punkt ist, wird zuerst getestet, ob der Harris Corner Measure größer als ein gegebener Schwellwert ist. Der Schwellwert wird durch den Parameter `harrisThreshold` gegeben. Ist dies der Fall, testen Sie mit FAST ob es sich dabei um einen Feature-Punkt handelt. Zunächst werden ein unterer t_- und ein oberer t_+ Schwellwert berechnet.

$$\begin{aligned} t_+ &= I(x,y) + t \\ t_- &= I(x,y) - t \end{aligned} \tag{33}$$

t ist dabei durch den Funktionsparameter `thresh` gegeben.

Nachdem Sie die beiden Schwellwerte berechnete haben, durchlaufen Sie nun die Punkte des Kreises und überprüfen Sie, ob N aufeinander folgende Pixel einen Wert größer bzw. kleiner als die zuvor berechneten Schwellwerte besitzen. Hinweis: Um Randproblme am Beginn bzw. Ende des Kreises zu unterbinden, können Sie z.B. den Kreis zwei mal durchlaufen. Ist dies der Fall, handelt es sich um einen Feature-Punkt. Zeitgleich berechnen Sie die Summe der Absoluten Differenzen (SAD) zwischen dem zentralen Pixel des Kreises und den Pixel auf dem Kreis. Die SAD wird im Anschluss für die Non-Maxima Suppression

verwendet.

$$SAD(x,y) = \sum_{(x_c,y_c)\in C} |I(x,y) - I(x_c,y_c)| \tag{34}$$

Wenn nun alle Feature-Punkte gefunden wurden wird noch einen Non-Maxima Suppression durchgeführt. Dazu gehen Sie wie folgt vor: Überprüfen Sie der Reihe nach alle Feature-Punkte ob Sie eine Distanz ≤ 9 zu einem anderen Feature-Punkt aufweisen. Ist dies der Fall vergleichen Sie die SAD der beiden Punkte. Sollte der aktuelle Punkt, welcher im Zentrum liegt, eine SAD aufweisen die geringer ist als die SAD eines andere Punktes, welcher eine Distanz ≤ 9 hat, so ist der aktuelle Punkt kein lokales Maximum und kann deshalb aus der Liste der Feature-Punkte entfernt werden.

Abbildung 17: Beispiel einer Non-Maxima Suppression[2].

[2] https://github.com/tejaslodaya/car-detection-yolo/blob/master/README.md

6 Eingabeparameter

Die folgenden Parameter in der Konfigurationsdatei bestimmen das Verhalten des Programms (Anwendungsbeispiele finden Sie in den mit gelieferten JSON Dateien):

- Pfad der Bilder: `folder`
- Name des Testcases: `name`
- Name des linken Bildes: `input_left`
- Name des rechten Bildes: `input_right`
- Maximaler Distanz Faktor: `match_distance_factor`
- Name der Binboost Datei: `wl_bin_name`
- Threshold: `diff_threshold`
- Testcase ist nur für Bonus Aufgabe designed: `bonus_only: 1`
- Keine Unterschiede in diesem Testcase: `no_diff: 1`

7 Programmgerüst

Die folgende Funktionalität ist in dem vom ICG zur Verfügung gestellten Programmgerüst bereits implementiert und muss von Ihnen nic ht selbst programmiert werden:

- Die Konfigurationsdatei (JSON) wird vom Programmgerüst gelesen.
- Lesen der im Video enthaltenen Bilder
- Ausführen der einzelnen Funktionen
- Schreiben der Ausgabebilder

8 Abgabe

Die Aufgaben bestehen jeweils aus mehreren Schritten, die zum Teil aufeinander aufbauen, jedoch unabhängig voneinander beurteilt werden. Dadurch ist einerseits eine objektive Beurteilung sichergestellt und andererseits gewährleistet, dass auch bei unvollständiger Lösung der Aufgaben Punkte erzielt werden können.

Wir weisen ausdrücklich darauf hin, dass die Übungsaufgaben von jedem Teilnehmer eigenständig gelöst werden müssen. Wenn Quellcode anderen Teilnehmern zugänglich gemacht wird (bewusst oder durch Vernachlässigung eines gewissen Mindestmaßes an Datensicherheit), wird das betreffende Beispiel bei allen Beteiligten mit 0 Punkten bewertet, unabhängig davon, wer den Code ursprünglich erstellt hat. Ebenso ist es nicht zulässig, Code aus dem Internet, aus Büchern oder aus anderen Quellen zu verwenden. Es erfolgt sowohl eine automatische als auch eine manuelle Überprüfung auf Plagiate

Die Abgabe der Übungsbeispiele und die Termineinteilung für die Abgabegespräche erfolgt über ein Webportal. Die Abgabe erfolgt ausschließlich über das Abgabesystem. Eine Abgabe auf andere Art und Weise (z.B. per Email) wird nicht akzeptiert. Der genaue Abgabeprozess ist im TeachCenter beschrieben.

Die Tests werden automatisch ausgeführt. Das Testsystem ist zusätzlich mit einem Timeout von 7 Minuten versehen. Sollte Ihr Programm innerhalb dieser Zeit nicht beendet werden, wird es vom Testsystem abgebrochen. Überprüfen Sie deshalb bei Ihrer Abgabe unbedingt die Laufzeit Ihres Programms.

Da die abgegebenen Programme halbautomatisch getestet werden, muss die Übergabe der Parameter mit Hilfe von entsprechenden Konfigurationsdateien genauso erfolgen wie bei den einzelnen Beispielen spezifiziert. Insbesondere ist eine interaktive Eingabe von Parametern nicht zulässig. Sollte aufgrund von Änderungen am Konfigurationssystem die Ausführung der abgegebenen Dateien mit den Testdaten fehlschlagen, wird das Beispiel mit 0 Punkten bewertet. Die Konfigurationsdateien liegen im JSON-Format vor, zu deren Auswertung steht Ihnen `rapidjson` zur Verfügung. Die Verwendung ist aus dem Programmgerüst ersichtlich.

Jede Konfigurationsdatei enthält zumindest einen Testfall und dessen Konfiguration. Es ist auch möglich, dass eine Konfigurationsdatei mehrere Testfälle enthält, um gemeinsame Parameter nicht mehrfach in verschiedenen Dateien spezifizieren zu müssen. In manchen Konfigurationsdateien finden sich auch einstellbare Parameter, die in Form eines `select` Feldes vorliegen. Diese sollen die Handhabung der Konfigurationsdateien erleichtern und ein einfaches Umschalten der Modi gewährleisten.

Es steht Ihnen frei, z.B. zu Testzwecken eigene Erweiterungen zu implementieren. Stellen Sie jedoch sicher, dass solche Erweiterungen in Ihrem abgegebenen Code deaktiviert sind,

damit ein Vergleich der abgegebenen Arbeiten mit unserer Referenzimplementierung möglich ist.

Die Programmgerüste, die zur Verfügung gestellt werden, sind unmittelbar aus unserer Referenzimplementierung abgeleitet, indem nur jene Teile entfernt wurden, die dem Inhalt der Übung entsprechen. Die Verwendung dieser Gerüste ist nicht zwingend, aber Sie ersparen sich sehr viel Arbeit, wenn Sie davon Gebrauch machen.

9 Test Bilder

Die Test Bilder stammen von `www.puzzlesandriddles.com` und `www.tinyurl.com/y646hu99`.

Literatur

[1] David G Lowe. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision, 60(2):91–110, 2004.

[2] Krystian Mikolajczyk and Cordelia Schmid. Scale & affine invariant interest point detectors. International Journal of Computer Vision, 60(1):63–86, 2004.

[3] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In European Conference on Computer Vision, 2006.

[4] Tomasz Trzcinski, Mario Christoudias, and Vincent Lepetit. Learning image descriptors with boosting. IEEE Transactions on Pattern Analysis and Machine Intelligence, 37(3):597–610, 2015.