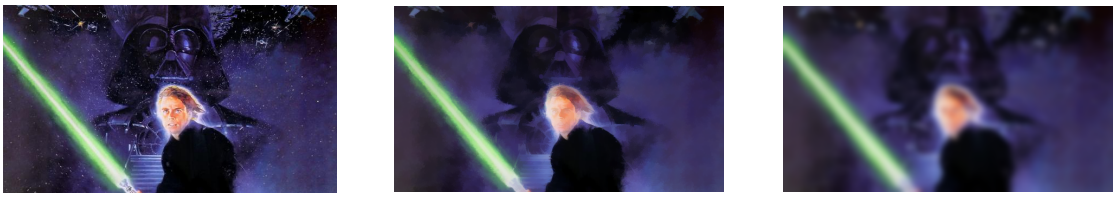
CV/task1 — Side Window Filtering

1 Überblick

Der Task 1 teilt sich in zwei Aufgabenstellungen. Zunächst sollte das Thema Punktoperationen behandelt werden. Dafür sollte eine Konvertierung vom BGR-Farbraum in das YUV-Farbemodell und eine entsprechende Rückkonvertierung durchgeführt werden. Zusätzlich sollen die einzelnen Kanäle Y, U und V visualisiert werden. Ziel der Aufgabe ist es, sich als Einstieg mit den verschiedenen Farbmodellen auseinanderzusetzen und Punktoperationen auf Bildern genauer zu behandeln.

Ziel der zweiten Aufgabenstellung ist es auf ein Bild den kanten-erhaltenden **S**ide-**W**indow-**F**ilter (SWF) [3] anzuwenden. Im Gegensatz zu anderen Filtern, wie dem einfachen Gauß-Filter, bleiben bei diesem Filter auch bei mehrfacher Anwendung die Kanten in den Bildern erhalten. Dadurch entsteht ein geglättetes Bild, in dem aber die Strukturen noch gut erhalten sind. Diese Filter-Technik kann eine Vielzahl gängiger Filter um kanten-erhaltende Eigenschaften erweitern (z.B. Gauß-, Box- oder Bilateral-Filter). Abbildung 1 zeigt ein Beispiel für die Anwendung.



(a) Eingabebild (b) Side-Window-Filter (c) Gauß-Filter

Abbildung 1: Im linken Bild ist das Eingabebild zu sehen. Das mittlere und das rechte Bild zeigen den Vergleich eines einfachen Gauß-Filter mit dem Side-Window-Filter. Die Filter wurden jeweils 30 mal angewendet. Es ist zu erkennen, dass mit dem Side-Window-Filter noch immer klare Strukturen zu erkennen sind, während mit dem Gauß-Filter kaum mehr etwas vom Eingabebild zu erkennen ist.

Die Eingabedaten werden aus JSON-Dateien bereits automatisch eingelesen und können sofort zur Berechnung herangezogen werden. Der Inhalt dieser Aufgabe beschränkt sich einzig und allein auf den SWF-Algorithmus und den Vergleich mit anderen Filtern, nicht auf etwaigen Aufwand, der sich durch die Programmiersprache ergibt. Das Framework ist so aufgebaut, dass Sie die relevanten Funktionen implementieren müssen um die gewünschte Ausgabe zu erreichen.

Der Algorithmus ist in mehrere Teile aufgeteilt und jeder Teil wird, so gut wie möglich, einzeln bewertet.

2 Aufgaben

Das Beispiel ist in mehrere Unteraufgaben gegliedert. Der Ablauf besteht aus den folgenden Schritten:

- Konvertierung BGR zu YUV
- Visualisierung der YUV Kanäle
- Konvertierung YUV zu BGR
- Initialisierung der Kernels
- Eigenhändige Berechnung der Faltung mit einem gegebenen Kernel
- Eigenhändige Berechnung der Faltung mit separierten Kernels
- Berechnung des Side-Window-Filters
- Bonus

Diese Aufgabe ist mit Hilfe von OpenCV[1] 3.2.x zu implementieren. Nutzen Sie die Funktionen, die Ihnen OpenCV zur Verfügung stellt und achten Sie auf die unterschiedlichen Parameter und Bildtypen.

2.1 BGR zu YUV Konvertierung (1.5 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `bgr_to_yuv(. . . )` zu implementieren. Es sollte das Eingabebild vom BGR-Farbraum in das YUV-Farbmodell[2] konvertiert werden. Die Konvertierung erfolgt mithilfe von Skalierungsfaktoren. Das Luma-Signale sowie die Chrominanzsignale werden durch Addition der gewichteten BGR Komponenten berechnet. Um dies zu erreichen gehen wir jeden Pixel im Eingangsbild `bgr_image` durch und berechnen uns anhand des Blau- (B), Rot- (R) und Grün-Werts (G) die Werte Y, U

[1] `http://opencv.org/`
[2] `https://de.wikipedia.org/wiki/YUV-Farbmodell`

und V. Diese werden dann in dieser Reihenfolge an der selben Position in das Ausgabebild `yuv_image` geschrieben.

Die Umrechnung erfolgt mit folgenden Formeln:

$$
\begin{aligned}
&\sout{Y := 0.299 \cdot R + 0.587 \cdot G + 0.144 \cdot B} \\
&Y := 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\
&\sout{U := (B - Y) \cdot 0.493} \\
&U := (B - Y) \cdot 0.492 \\
&V := (R - Y) \cdot 0.877
\end{aligned}
\tag{1}
$$

Es ergeben sich Wertebereiche bei Y von 0 bis 255, bei U von 0 bis ± 112 und bei V von 0 bis ± 157. Daher müssen zusätzlich die Komponenten U und V skaliert werden, um positive Werte zu erhalten, die in den Datentyp uchar passen. Bei Komponente U wird einfach der Wert 112 addiert. V wird auf einen Bereich von ± 128 skaliert und dann mit 128 addiert.

Da durch Floating-Point Rundungsfehler negative Werte oder Werte größer 255 entstehen könnten, werden Werte < 0 auf 0 und Werte > 255 auf 255 gesetzt. Um die Werte von Y, U und V dann als uchar abspeichern zu können werden diese zuvor noch gerundet.

Die Verwendung der Funktion `cv::cvtColor(...)` ist NICHT erlaubt!

`Hinweis`: OpenCV speichert Matrizen im BGR Format. Daher steht der Rot-Wert eines Pixels an dritter Stelle und der Blau-Wert eines Pixels an erster Stelle im Speicher.

Hilfreiche OpenCV-Methoden:

- `cvRound(...)`
- `cv::Mat::at<...>(...)`

2.2 Visualisierung der Y, U und V Kanäle (1 Punkt)

In der Funktion `visualize_yuv_channels(...)` sollen die einzelnen Kanäle Y, U und V des YUV-Eingabebildes visualisiert werden. Das heißt, dass es zuerst in die einzelnen Kanäle aufgeteilt und für jeden Kanal ein BGR-Bild erstellt werden muss. Der Kanal Y ist bereits Grayscale und kann direkt ohne weitere Bearbeitungen mittels `cv::cvtColor` in BGR konvertiert werden. Für die Kanäle U und V werden jeweils Colormaps zur Verfügung

gestellt. Diese Lookup-Tabellen enthalten die Zuweisung eines U- bzw. V-Pixelwerts zu einem BGR Wert der zur Visualisierung verwendet werden soll.

Formal berechnen wir also die Pixel-Werte der Ausgabe O_U für den Kanal U bzw. der Ausgabe O_V folgendermaßen:

$$\begin{aligned} &O_U(i,j)_c = u\_color\_map(c, U(i,j)) \\ &O_V(i,j)_c = v\_color\_map(c, V(i,j)) \\ &\forall c \in \{0,1,2\}. \end{aligned} \tag{2}$$

Wobei u_color_map bzw. v_color_map die Lookup Tabelle für den U bzw. V Kanal ist, $O_U(i,j)_c$ bzw. $O_V(i,j)_c$ das Pixel an der Position (i,j) mit den Farbkanal c der Ausgabe für den O bzw. V Kanal ist. $U(i,j)$ und $V(i,j)$ bezeichnen den Pixel an der Position (i,j) im U bzw. V Kanal. Zum Transformieren des Bildes muss diese Lookup-Operation auf alle Pixel des Bildes angewandt werden (also $0 \leq i < height$, $0 \leq j < width$, wobei $height$ die Höhe des Bildes bezeichnet und $width$ die Breite).

Hilfreiche OpenCV-Methoden:

- `cv::split(...)`
- `cv::cvtColor(...)`

(a) Kanal Y (b) Kanal U (c) Kanal V

Abbildung 2: Ergebnis der Visualisierung der Kanäle Y, U und V.

2.3 YUV zu BGR Konvertierung (1.5 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `yuv_to_bgr(...)` zu implementieren. Es sollte das Eingabebild vom YUV-Farbraum in das BGR-Farbmodell konvertiert werden. Dafür werden für jeden Pixel die Operationen der Funktion `bgr_to_yuv(...)` in umgekehrter Reihenfolge angewendet. Zuerst werden von den U-Werten wieder 112 abgezogen, um die Werte in den Bereich ± 112 zu bringen. Auch für den V-Wert wird

die umgekehrte Rechenoperation angewendet. Dafür wird zuerst 128 abgezogen und danach der Wert mit 157 multipliziert und durch 128 dividiert. Somit sind die Werte im Bereich ± 157. Für die Umrechnung von YUV zu BGR werden die Formeln

$$\begin{aligned} B &:= Y + \frac{U}{0.493} \\ R &:= Y + \frac{V}{0.877} \\ G &:= 1.704 \cdot Y - 0.509 \cdot R - 0.194 \cdot B; \end{aligned} \tag{3}$$

verwendet. Da durch Floating-Point Rundungsfehler negative Werte oder Werte größer 255 entstehen könnten, werden Werte < 0 auf 0 und Werte > 255 auf 255 gesetzt. Um die Werte als uchar abspeichern zu können, werden diese zuvor noch gerundet.

Die Verwendung der Funktion `cv::cvtColor(...)` ist NICHT erlaubt!

`Hinweis:` OpenCV speichert Bilder im BGR Format. Dies bedeutet, dass der Rot-Wert für jeden Pixel an dritter Stelle im Speicher steht und der Blau-Wert an erster Stelle steht.

Hilfreiche OpenCV-Methoden:

- `cvRound(...)`
- `cv::Mat::at<...>(...)`

2.4 Erstellung und Initialisierung der Kernel Matrix (1 Punkt)

Dieser Teil der Aufgabe ist in der Funktion `initializer_kernel_matrix(...)` zu implementieren. Hier werden Filter-Kernels initialisiert mit denen im Folgenden die Faltung und somit die Filterung des Eingabebildes durchgeführt werden soll.

Um die Berechnung von 2D Faltungen zu beschleunigen, werden diese in der Bildverarbeitung soweit möglich in zwei 1D Faltungen (mit zwei 1D Kernels) aufgeteilt. 2D Kernels bei denen dies möglich ist werden als *separierbar* bezeichnet. Dies hat den Vorteil, dass nur 1D Kernels initialisiert werden müssen und die Berechnung zweier Faltungen mit den zwei 1D Kernels schneller ist als die Faltung mit dem entsprechenden 2D Kernel. Im Folgenden initialisieren wir Kernels für einen Side-Window-Filter und einen einfachen Gauß-Filter. Letztere wird als 2D- und separierter 1D Kernel initialisiert und dient als Vergleich. Auf die *Separierbarkeit* wird später noch im Detail eingegangen.

Im Detail sind dies die Filter-Kernels `k_L, k_R, k, k_gauss_1d, k_gauss_2d`. Für alle Werte aus $\mathbb{R}$ sollen **float**-Werte verwendet werden. Bei mathematischen Funktionen (z.B. exp, sqrt, etc.) sollen jeweils die OpenCV Funktionen `cv::fktname` verwendet werden! Die Matrizen haben folgende Dimensionalität:

`{k_L, k_R, k, k_gauss_1d}` $\in \mathbb{R}^{1\times 2r+1}$,
`k_gauss_2d` $\in \mathbb{R}^{2r+1\times 2r+1}$,
r ... radius.

In dieser Übung implementieren wir die Side-Window-Version des sogenannten Box Filters. Alle dessen *Side-Windows* können - sofern richtig initialisiert - aus `k_L`, `k_R` und `k` berechnet werden. Bei `k_L` sollen die ersten $r+1$ Einträge jeweils $\frac{1.0}{r+1}$ sein und der Rest soll 0 sein. Bei `k_R` ist es genau umgekehrt. Hier sollen die letzte $r+1$ Einträge $\frac{1.0}{r+1}$ sein und der Rest soll 0 sein. Bei `k` werden alle $(2r+1)$ Werte mit $\frac{1.0}{2r+1}$ initialisiert.

`k_gauss_1d` ist der (separierte) 1D Gauß-Filter-Kernel. Da im Falle des Gauß-Filters beide 1D Kernels identisch sind, ist die Initialisierung eines einzelnen 1D Kernels ausreichend. Für jeden Eintrag wird die Gauß-Verteilung mit

$$G_{1D}(x) = \frac{1.0}{\sqrt{2.0\pi}\sigma} e^{-\frac{(x-r)^2}{2.0\sigma^2}} \tag{4}$$

berechnet. Wobei die Position x von 0 bis $2r+1$ iteriert wird.

`k_gauss_2d` soll dem 2D Gauß-Filter-Kernel entsprechen. Hier wird auch für jeden Eintrag eine Gauß-Verteilung in 2D berechnet:

$$G_{2D}(y,x) = \frac{1.0}{2.0\pi\sigma^2} e^{-\frac{(y-r)^2+(x-r)^2}{2.0\sigma^2}}. \tag{5}$$

Wir gehen auch hier beim Befüllen der Matrix die Positionen x und y von 0 bis $2r+1$ durch.

2.4.1 Beispiele

Wenn unsere Funktion mit $r = 3$ und $\sigma = 1$ aufgerufen wird, sollten folgende Filter-Kernel zurück gegeben werden:

$$\texttt{k\_L} = \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 & 0 & 0 & 0 \end{bmatrix},$$

$$\texttt{k\_R} = \begin{bmatrix} 0 & 0 & 0 & 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix},$$

$$\texttt{k} = \begin{bmatrix} 0.1429 & 0.1429 & 0.1429 & 0.1429 & 0.1429 & 0.1429 & 0.1429 \end{bmatrix},$$

$$\texttt{k\_gauss\_1d} = \begin{bmatrix} 0.0044 & 0.0540 & 0.2420 & 0.3989 & 0.2420 & 0.0540 & 0.0044 \end{bmatrix},$$

$$\texttt{k\_gauss\_2d} = \begin{bmatrix} 1.9641e-05 & 0.0002 & 0.0011 & 0.0018 & 0.0011 & 0.0002 & 1.9641e-05 \\ 0.0002 & 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 & 0.0002 \\ 0.0011 & 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 & 0.0011 \\ 0.0018 & 0.0215 & 0.0965 & 0.1592 & 0.0965 & 0.0215 & 0.0018 \\ 0.0011 & 0.0131 & 0.0585 & 0.0965 & 0.0585 & 0.0131 & 0.0011 \\ 0.0002 & 0.0029 & 0.0131 & 0.0215 & 0.0131 & 0.0029 & 0.0002 \\ 1.9641e-05 & 0.0002 & 0.0011 & 0.0018 & 0.0011 & 0.00024 & 1.9641e-05 \end{bmatrix}.$$

Hilfreiche OpenCV-Methoden:

- `Mat::ones(...)`
- `Mat::zeros(...)`
- `cv::hconcat(...)`
- `cv::exp(...)`
- `cv::sqrt(...)`

2.5 Manuelle Implementierung einer 2D Convolution (3 Punkt)

Ziel dieser Aufgabe ist die Implementierung einer 2-dimensionaler Faltung innerhalb der Funktion `manual_filter_2d(...)`. Hierbei wird ein 2D Kernel k auf die Eingabe I angewandt um das Ergebnis O zu erhalten.

$$O = I * k \tag{6}$$

Die Faltung wird mit dem Zeichen $*$ gekennzeichnet. Wir illustrieren eine solche Faltung in Abbildung 3. Um die Faltung durchzuführen wird die Nachbarschaft jeden Elements

$e \in I$ betrachtet. Hierbei stellt der Wert des Ergebnis O die Summe der Multiplikationen der einzelnen Gewichte $w \in k$ und der Nachbarschaft des Elements e dar. In unserem Fall muss die Eingabe `input` zuerst um den Radius des übergebenen `kernel` erweitert (*"Padding"*) werden, um sicher zu stellen, dass alle Elemente $e \in I$ eine Nachbarschaft besitzen, die der Größe des Kernels entspricht.

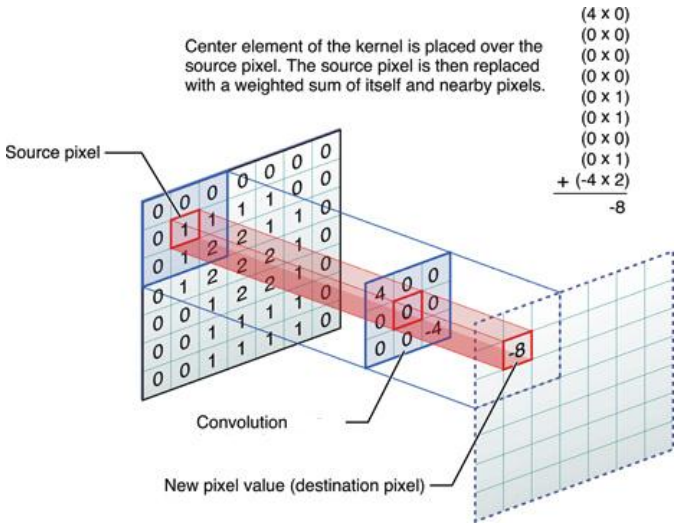


Abbildung 3: Illustration einer 2D Faltung mit einem Kernel der Größe 3×3. Quelle:[3].

Formal werden die Ausgaben $O(y,x)$ folgendermaßen berechnet:

$$O(y,x) = \sum_{k_y=0}^{k_{rows}} \sum_{k_x=0}^{k_{cols}} I\left(y + k_y - \frac{k_{rows}-1}{2}, x + k_x - \frac{k_{cols}-1}{2}\right) \cdot k(k_y,k_x), \tag{7}$$

wobei k_{rows} und k_{cols} die Anzahl der Zeilen bzw. Spalten des Filterkernels k sind. Da $O(\cdot,\cdot)$ die selbe Größe wie das Eingabebild $I(\cdot,\cdot)$ haben soll, gibt es am Rand von $O(\cdot,\cdot)$ out-of-bounds Zugriffe auf $I(\cdot,\cdot)$. So wird zum Beispiel für $O(0,0)$, bei einem Kernel der Größe 3×3, auf $I(-1,-1)$ zugegriffen. Um solche Zugriffe zu vermeiden, sollen die Pixelwerte von $I(\cdot,\cdot)$ am Rand gespiegelt werden. Am einfachsten ist dies mittels der Funktion `cv::copyMakeBorder(...)` zu erreichen. Es muss der Bordertyp `cv::BORDER_REPLICATE` verwendet werden.

Die Verwendung der Funktion `cv::filter2D(...)` ist NICHT erlaubt!

Hinweis: Das Eingabebild `input` im Code kann auf den selben Speicher zeigen, wie das Ausgabebild `output`. Es ist daher empfohlen eine temporäre Kopie des Eingabebildes zu erstellen bzw. die Ausgabe zunächst ein eine temporäre Matrix zu schreiben und diese dann am Ende der Funktion in `output` zu kopieren (z.B.: `output_tmp.copyTo(output)`).

[3] `https://miro.medium.com/max/464/0*e-SMFTzO8r7skkpc`

Hilfreiche OpenCV-Methoden:

- `cv::copyMakeBorder(. . . )`

2.6 Anwenden der separierbaren Kernels und Convolution (3 Punkte)

Dieser Teil der Aufgabe ist in `apply_separable_conv2d(. . . )` zu implementieren. Es wird dabei eine Eigenschaft der bereits angedeuteten *Separierbarkeit*[4] ausgenützt, die es ermöglicht einen 2D Kernel k in zwei 1D Kernel aufzuteilen (k_x und k_y). Einen 3×3 Box Filter könnte man zum Beispiel folgendermaßen aufteilen:

$$\underbrace{\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}}_{k} = \underbrace{\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix}}_{\underline{k_x} k_y} * \underbrace{\begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix}}_{\underline{k_y} k_x}. \tag{8}$$

Um den Filter k nun auf ein Eingabebild I anzuwenden um das Ergebnis O zu berechnen, kann zuerst der Filter ~~k_x~~ k_y auf das Bild angewandt werden. Auf das Ergebnis dieser Operation, wird anschließend der Filter ~~k_y~~ k_x angewandt. Formal kann man das Ergebnis also mittels ~~$O = I * k = (I * k_x) * k_y$~~ $O = I * k = (I * k_y) * k_x$ berechnen. Das Anwenden von separierten Filtern auf Bildern benötigt $2 \cdot M \cdot N \cdot k_{size}$ Multiplikationen, wobei M und N die Höhe bzw Breite des Bildes sind, und k_{size} die Länge des Kernels ist. Dies ist bei großen Kernel-Größen deutlich schneller als das Anwenden einer normalen Faltung, die $M \cdot N \cdot k_{size} \cdot k_{size}$ Multiplikationen benötigt.

In der Funktion `apply_separable_conv2d(. . . )` ist noch zu beachten dass der Parameter `y_kernel` noch der Form

$$\begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix} \tag{9}$$

entspricht und erst in die richtige Form

$$\begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} \tag{10}$$

transponiert werden muss. Für die Faltung der einzelnen separierten 1D Faltungen soll die zuvor implementierte Funktion `manual_filter_2d(. . . )` verwendet werden.

Die Verwendung der Funktion `cv::filter2D(. . . )` ist NICHT erlaubt!

[4] `https://de.wikipedia.org/wiki/Separierbarkeit`

Hilfreiche OpenCV-Methoden:

- `cv::transpose(. . . )`

2.7 Implementierung des Side Window Box Filters (3 Punkte)

Dieser Teil der Aufgabe ist in der Funktion `sw_filter(. . .)` zu implementieren. Hier sollen die Grundprinzipien eines Side Window Box Filters implementiert werden.

Der Vorteil eines Side Window Filters gegenüber normalen Filtern ist, dass beim Anwenden des Filters Kanten im Bild besser erhalten bleiben. Um dies zu erreichen, wird das Faltungs-Fenster in mehrere kleinere Fenster (nämlich Westen, Osten, Norden, Süden, Nord Westen, Nord Osten, Süd Westen, Süd Osten) unterteilt (siehe Abbildung 5). Nun wird für jeden Pixel das Filter Ergebnis für jedes dieser 8 Fenster separat berechnet. Das Ergebnis im Ausgabebild $O(y,x)$ ist jener dieser 8 Werte, der am ähnlichsten zum Pixel im Input Bild $I(y,x)$ ist (siehe auch Algorithmus 1).

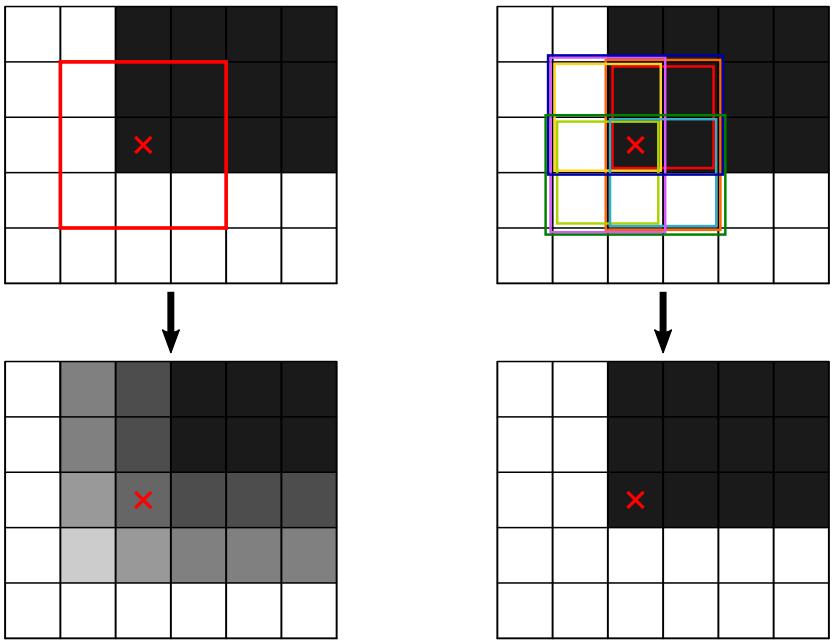


Abbildung 4: Einfacher Box Filter im Vergleich zu dessen SWF Erweiterung. Bei einem einfachem Filter wird pro Pixel üblicherweise nur ein Fenster ausgewertet (links). Hier wird bei einem Box Filter der Durchschnitt der Pixelwerte innerhalb des Fensters (hier rot) berechnet und dem Pixel im Zentrum des Kernels (rotes Kreuz) zugewiesen. Dadurch wird wie hier gezeigt die Kante geblurred. Die SWF Version (rechts) desselben Filters evaluiert mehrere Fenster (unterschiedliche Farben, aus Gründen der Visualisierung leicht versetzt) und weist dem Pixel im Kernelzentrum den Wert zu, der die geringste Differenz zum Wert des Pixel selbst hat. Hier bekommt das Pixel das Ergebnis des roten Fensters zugewiesen, da dessen Durchschnittswert denselben Wert besitzt wie das Pixel selbst. Dadurch bleibt die Kante erhalten.

Intuitiv erklärt, trifft das Faltungsfenster auf eine Kante im Bild, so verschmiert (blurred) ein normales Faltungsfenster die Kante. Beim Side Window Filter ist die Wahrscheinlichkeit hoch, dass ein Fenster vollständig in einer homogenen Fläche des Eingabebildes

fällt (siehe auch Abbildung 4). Dadurch wird die Kante dann im Ausgangsbild nicht mehr geblurred.

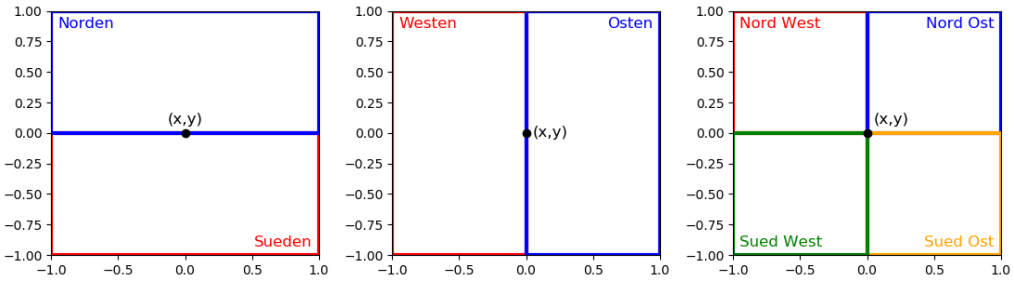


Abbildung 5: Illustration der 8 Side Window Fenster. Das große Faltungs-Fenter wird in ein Nord und Süd Fenster (links), Westen und Osten Fenster (mitte) und Nord West, Nord Ost, Süd Ost, Süd West Fenster (rechts) geteilt. Insgesamt werden daher die Ergebnisse von 8 kleineren Faltungs-Fenstern zum Filtern eines Pixels verwendet.

Algorithm 1 Formale Beschreibung des SWF Algorithmus.

Require: Sei $k(\cdot,\cdot)$ der Faltungs Filter,
$I(\cdot,\cdot)$ das Eingabebild,
$S = \{W, O, N, S, NW, NO, SW, SO\}$ die Menge der Side Windows,
$\mathscr{I}(n)$ Die Menge der Indices (i,j) die zum entsprechenden Side Window gehören.

1: Berechne alle Side Window Filter
$I_n(y,x) = \frac{1}{N_n} \sum_{(i,j)\in\mathscr{I}(n)} k(i,j) \cdot I(y+i, x+j)$, $N_n = \sum_{(i,j)\in\mathscr{I}(n)} k(i,j)$, $n \in S$
2: finde am besten passenden Side Window Filter zum Eingabebild
$m = \text{argmin}_{n\in S} |I(y,x) - I_n(y,x)|^2$
3: setze Output $O(y,x) := I_m(y,x)$

Um Algorithmus 1 in unserem Framework zu implementieren, teilen wir zuerst das Eingabebild in die drei Farbkanäle auf und führen die folgenden Schritte für jeden Kanal aus:

- Wir wenden alle Side Windows an (siehe Funktion `calc_side_windows` und Abschnitt 2.7.1)
- Wir berechnen für jedes Side Window die absolute Differenz zum jeweiligen Input-Kanal (siehe Funktion `calc_abs_difference` und Abschnitt 2.7.2)
- Wir finden für jedes Pixel im Input-Kanal das beste Side Window mit der geringsten Differenz (siehe Funktion `find_best_fitting_side_window` und Abschnitt 2.7.3)

Am Ende, wenn alle Farbkanäle prozessiert wurden, werden diese wieder zusammengefügt und in `result` gespeichert.

Hilfreiche OpenCV-Methoden:

- `cv::split(. . . )`
- `cv::merge(. . . )`

2.7.1 Speichern der Side Window Differenzen (1 Punkte)

Um SWF richtig umzusetzen, müssen die entsprechenden Faltungen umgesetzt werden. Das soll in der Funktion `calc_side_windows(. . . )` geschehen. Um die einzelnen Side Windows zu erzeugen müssen die einzelnen Parameter `kernel`, `kernel_left` und `kernel_right` mithilfe der Funktion `apply_seperable_conv2d(. . . )` kombiniert werden. Ein Beispiel für die erhaltenen Kernel wäre:

$$\begin{aligned} kernel &= \begin{bmatrix} 1/3 & 1/3 & 1/3 \end{bmatrix} \\ kernel\_left &= \begin{bmatrix} 1/2 & 1/2 & 0 \end{bmatrix} \\ kernel\_right &= \begin{bmatrix} 0 & 1/2 & 1/2 \end{bmatrix} \end{aligned}$$

Gesamt müssen folgende 8 Side Windows erstellt werden.

1. Nord Westen
2. Süd Westen
3. Nord Osten
4. Süd Osten
5. Westen
6. Osten
7. Norden
8. Süden

Um das Bild nun z.B. mit dem Side Window *Nord Westen* zu falten, müssen zwei der 1D Kernel (`kernel`, `kernel_left`, `kernel_right`) mittels separierter Faltung (Funktion `apply_seperable_conv2d(. . . )`) zu einem Filter kombiniert werden, der in der linken oberen Hälfte positiv ist, und überall sonst 0 ist. In diesem Fall, müsste also die Funktion mit den Kernel `kernel_left` und `kernel_left` aufgerufen werden, da:

$$\begin{bmatrix} 1/2 \\ 1/2 \\ 0 \end{bmatrix} * \begin{bmatrix} 1/2 & 1/2 & 0 \end{bmatrix} = \begin{bmatrix} 1/4 & 1/4 & 0 \\ 1/4 & 1/4 & 0 \\ 0 & 0 & 0 \end{bmatrix} = kernel\_nord\_westen.$$

Für die Referenzbilder wurde obige Reihenfolge der Side Windows verwendet. Eine andere Reihenfolge ändert bei korrekter Implementation nichts am Ergebnis des Side Window Filters.

2.7.2 Berechnung der absoluten Differenz (0.5 Punkte)

Abschließend wird in der Funktion `calc_abs_difference(...)` für jedes Side Window Bild die Differenz zur Eingabe berechnet und in den Parameter `side_window_differences` eingefügt. Für die einzelnen Differenzen wird die folgende Formel für die Berechnung der Absolutwerte verwendet:

$$D = |I - I_{SW}|. \tag{11}$$

Hilfreiche OpenCV-Methoden:

- `cv::abs(...)`

2.7.3 Finden des bestgeeignetsten Side Windows (0.5 Punkte)

In der Funktion `find_best_fitting_side_window(...)` soll für jeden Wert des Outputs `channel` der bestgeeignetste Wert der Side Windows in `filtered_imgs` gefunden werden. Hierzu müssen die Werte in allen Reihen und Spalten durchlaufen werden. Für jeden Wert sollen die absoluten Differenzen in `d_abs` an der gegebenen Stelle verglichen werden. Dabei stellt die geringste Differenz die geeignetste dar. Zuletzt soll dem `channel` der Wert an der gegebenen Stelle des dazugehörigen gefilterten Bildes aus `filtered_imgs` zugewiesen werden.

3 Bonusaufgaben (3 Punkte)

3.1 Bonus Aufgabe (3 Punkt)

Als Bonus Aufgabe soll die Side Window Technik auf einen bilateralen Filter [1] angewandt werden. Ein bilateral Filter ist ein nichtlinearer kanten-erhaltener Filter. Formal ist der bilateral Filter wie folgt definiert:

$$O(\boldsymbol{x}) = \frac{1}{W_p} \sum_{\boldsymbol{x}_i \in \omega} I(\boldsymbol{x}_i) \cdot f_r(||I(\boldsymbol{x}_i) - I(\boldsymbol{x})||) \cdot g_s(||\boldsymbol{x}_i - \boldsymbol{x}||).$$

Hierbei ist W_p ein Normierungsfaktor, $\boldsymbol{x} = (i, j)$ bezeichnet die aktuelle Pixel Position und $\boldsymbol{x}_i = (j, k)$ die Koordinaten einer lokalen Nachbarschaft um $\boldsymbol{x}$. Der bilateral Filter hat also eine ähnliche Form wie eine normale Faltung, mit dem Unterschied, dass die

Kernel-Gewichte abhängig vom Bildinhalt sind. Wir verwenden sowohl für f_r als auch g_s einen Gaußfilter, wodurch sich im diskreten Fall die Berechnung vereinfachen lässt.

Aufgrund der Vereinfachungen können wir $f_r(\cdot) \cdot g_s(\cdot)$ durch eine Gewichtungsfunktion wie folgt ersetzen:

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\cdot\sigma_d^2} - \frac{|I(i,j)-I(k,l)|^2}{2\cdot\sigma_r^2}\right),$$

wobei i, j die Koordinaten des gerade betrachteten Pixels sind und k, l die Koordinaten des betrachteten Nachbars, I stellt einen Farbkanal des Bilds dar. Diese Gewichtungsfunktion ersetzt sozusagen die Kernelfunktion $k(\cdot)$ in der normalen Faltung.

Die Formel des bilateralen Filters setzt sich somit aus

$$O(i,j) = \frac{\sum_{k,l} I(i,j) \cdot w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}$$

zusammen. Im Unterschied zu linearen Faltungen, hängt hier die Kernel Funktion $w(\cdot)$ auch vom Bildinhalt $I(\cdot)$ ab, da zum Berechnen der Gewichte Differenzen zwischen unterschiedlichen Elementen in $I(\cdot)$ verwendet werden.

Um nun diesen Filter zu verbessern, wollen wir die Side Window Technik darauf anwenden. Das bedeutet, dass der bilaterale Filter für jedes unserer 8 Side Windows separat aufgerufen wird. Danach wird wieder jenes gefilterte Pixel als Ergebnis genommen, dass die minimale Differenz zum Ursprungsbild hat. Für die Suche nach der minimalen Differenz können die bereits implementierten Funktionen `calc_abs_difference(...)` und `find_best_fitting_side_window(...)` verwendet werden.

Um nun die Box-Filter, die wir in Abschnitt 2.7 verwendet haben, mit einem solchen bilateral Filter ersetzen, müssen wir die Funktionen

- `create_bilateral_masks(...)`,
- `compute_bilateral_weight(...)`,
- `bilateral_filter(...)` und
- `bonus(...)`

implementieren.

3.1.1 Erstellen der Filter Masken

Dieser Teil der Aufgabe ist in `create_bilateral_masks(. . . )` zu implementieren. Hier sollen ähnlich zum Abschnitt 2.7.1 Masken für die Side Windows Nord Westen, Nord Osten, Süd Westen, Süd Osten, Norden, Süden, Westen und Osten erstellt werden. Wir verwenden dafür die Eingabemasken `kernel_left`, `kernel_right` und `kernel`. Ähnlich wie in Abschnitt 2.7.1 erstellen wir uns die Ausgabemasken hier durch Kombination dieser drei Eingabekernel. Dazu transponieren wir die Kernel entsprechend und Matrix-multiplizieren sie, um die gewünschte Ausgabemaske zu erhalten.

Die Maske für das Fenster "Nord Westen", kann zum Beispiel durch $kernel\_left^\top \cdot kernel\_left$ erstellt werden.

Wir speichern alle dieser 8 Masken im Ausgabevector `kernels`. Ähnlich wie im Abschnitt 2.7 spielt auch hier die Reihenfolge in der diese Filter erstellt werden bzw. im Anschluss angewendet werden bei korrekter Implementierung des bilateralen SWF keine Rolle.

Später verwenden wir diese berechneten Kernel nicht direkt für die Faltung. Diese Kernel stellen lediglich eine binäre Maske dar, die unsere Nachbarschaft definiert.

3.1.2 Berechnen der Gewichte des Bilateral Filters

Dieser Teil der Aufgabe ist in der Funktion `compute_bilateral_weight(. . . )` zu implementieren. Ziel dieser Funktion ist es die die Gewichte des bilateral Filters zu berechnen. Diese Funktion übernimmt das Eingabebild I, die Indizes i und j für das Input Bild, die Indizes k und l für die Filter Maske, sowie den Parameter σ_d. Die Funktion soll das Gewicht:

$$w(i,j,k,l) = \exp\left(-\frac{(i-k)^2+(j-l)^2}{2\cdot\sigma_d^2} - \frac{|I(i,j)-I(k,l)|^2}{2\cdot\sigma_r^2}\right)$$

zurückgeben. Der Parameter σ_r ist als Konstante im Code definiert.

3.1.3 Anwendung des Bilateral Filters

Dieser Teil der Aufgabe ist in der Funktion `bilateral_filter(. . . )` zu implementieren. Ziel dieser Funktion ist es den bilateral Filter auf das Side Window `kernel` anzuwenden.

Dazu gehen wir das Bild zeilenweise durch und gehen an jeder Position (i,j) die positiven Elemente der `kernel` Maske durch. Diese Elemente, mit den Koordinaten (k,l), definieren eine lokale Nachbarschaft um den Pixel (i,j).

Für jedes Pixel in der Nachbarschaft wird mittels `compute_bilateral_weight(...)` das entsprechende Filter-Gewicht berechnet. Das Gewicht wird, ähnlich wie bei einer normalen Faltung, mit dem entsprechenden Pixel-Wert im Bild multipliziert und im Fenster aufsummiert. Am Schluss wird diese Summe durch die Summe der Gewichte normalisiert und in das Ausgabebild O an der Stelle (i,j) geschrieben.

Formal berechnet man die Ausgabe dieser Funktion also mittels:

$$O(i,j) = \frac{\sum_{(k,l)\in\mathscr{I}(\text{kernel})} I(i+k-\frac{r-1}{2}, j+l-\frac{r-1}{2}) \cdot w(i,j,i+k-\frac{r-1}{2}, j+l-\frac{r-1}{2})}{\sum_{(k,l)\in\mathscr{I}(\text{kernel})} w(i,j,i+k-\frac{r-1}{2}, j+l-\frac{r-1}{2})}.$$

Wobei I das Eingabebild, O das Ausgabebild und $w(\ldots)$ das Gewicht ist. Die Indizes (i,j) gehen jeweils von 0 bis zur Bild-Höhe bzw. -Breite. Die Indizes (j,k) gehen über die positiven Elemente in der Filter Maske `filter`. Wie bei einer linearen Faltung muss das Kernel-Fenster um den aktuellen Pixel (i,j) zentriert werden. Daher wird beim Aufruf $w(\ldots)$ die Hälfte des Kernel-Radius (nämlich die Kernel-Höhe bzw. -Breite), $\frac{r-1}{2}$, vom Filter-Index abgezogen, sowie die aktuelle Position (i,j) im Bild addiert.

Um zu verhinden, dass beim Berechnen der Ergebnisse am Rand out-of-bounds Zugriffe entstehen, soll das Eingabebild an den Rändern gespiegelt werden. Dazu kann die Funktion `cv::copyMakeBorder(...)` mit dem Flag `cv::BORDER_REPLICATE` verwendet werden.

3.1.4 Integration der Bonus Aufgabe

Im letzten Schritt integrieren wir unsere implementierten Funktionen in der Funktion `bonus(...)`. Diese Funktion hat eine ähnliche Aufgabe, wie die Funktion `swf_filter(...)`.

Wir erstellen unsere 8 Filter Masken mittels der Funktion `create_bilateral_masks(...)`. Danach iterieren wir über alle 3 Farbkanäle. Für jeden Farbkanal und jeden unserer 8 erstellten Filter Masken wenden wir die Funktion `bilateral_filter(...)` an, um den Filter Response für das entsprechende Side Window zu erstellen. Anschließend berechnen wir mit `calc_abs_difference(...)` die absolute Differenz zwischen jedem Side Window Ergebnis und dem Eingabebild. Zum Schluss verwenden wir die Funktion `find_best_fitting_side_window(...)`, um mittels der absoluten Differenzen das beste Side Window in das Ergebnis zu schreiben.

4 Ein- und Ausgabeparameter

Folgende Parameter sind in den Konfigurationsdateien angegeben:

- Dateiname
- Radius der verwendeten Kernels
- Sigma für Gaußkernel
- Anzahl der Iterationen pro Filter

5 Programmgerüst

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

- Die Konfigurationsdatei (JSON) wird vom Programmgerüst gelesen.
- Lesen des Eingabebildes und der Eingabeparameter
- Iteratives Ausführen der einzelnen Funktionen
- Schreiben der Ausgabebilder in die dafür vorgesehenen Ordner

6 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.

Literatur

[1] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, Frédo Durand, et al. Bilateral Filtering: Theory and Applications. Foundations and Trends® in Computer Graphics and Vision, 4(1):1–73, 2009.

[2] Michal Podpora, Grzegorz Pawel Korbas, and Aleksandra Kawala-Janik. YUV vs RGB-Choosing a Color Space for Human-Machine Interaction. FedCSIS Position Papers, 18:29–34, 2014.

[3] Hui Yin, Yuanhao Gong, and Guoping Qiu. Side Window Filtering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2019.