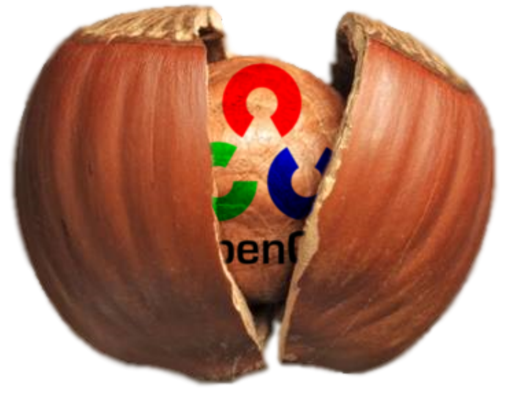
Fachhochschule Aachen
Fachbereich 9 — Medizintechnik und Technomathematik

Seminararbeit
im Studiengang Scientific Programming

OpenCV in a Nutshell



Autor:	Marius Politze
Mat. No.	995282
Erstprüfer:	Prof. Dr. Andreas Terstegge
Zweitprüfer:	Dr. Alexander Voß

Aachen, den 01. September 2010

Inhaltsverzeichnis

1 Einleitung 1
1.1 Ziel dieser Arbeit . . . 1
1.2 Was ist OpenCV? . . . 1
1.3 Aufbau der Bibliothek . . . 1

2 Erste Schritte 3
2.1 Benutzung von OpenCV . . . 3
2.1.1 Download und Installation . . . 3
2.1.2 Kompilieren von Programmen . . . 3

3 Datentypen 4
3.1 Einfache Datentypen . . . 4
3.1.1 CvPoint . . . 4
3.1.2 CvScalar . . . 4
3.1.3 CvSize . . . 4
3.1.4 CvRect . . . 4
3.1.5 CvMat . . . 5
3.2 IplImage . . . 5
3.2.1 Darstellung von (Farb-)Bilddaten . . . 5
3.3 CvSeq . . . 8
3.3.1 Speicherlayout von Konturen als CvSeq . . . 9

4 Funktionsumfang 11
4.1 Ein- und Ausgabe . . . 11
4.1.1 Bilder lesen und anzeigen . . . 11
4.1.2 Videos als Eingabedaten . . . 12
4.1.3 Zeichnen . . . 12
4.1.4 Daten speichern . . . 13
4.2 Bildmanipulationen . . . 14
4.2.1 Weichzeichnen . . . 14
4.2.2 Schwellwerte . . . 17
4.2.3 Morphologie . . . 19
4.2.4 Faltung . . . 23
4.2.5 Gradienten . . . 24
4.2.6 Flächen füllen . . . 27
4.3 Bildtransformationen . . . 28
4.3.1 Farbräume . . . 28
4.3.2 Resize . . . 28
4.3.3 Bildpyramiden . . . 29
4.3.4 Strecken, verzerren und rotieren . . . 29
4.4 Hough Transformationen . . . 32
4.5 Histogramme . . . 33
4.5.1 Histogrammausgleich . . . 33
4.5.2 Berechnen von Histogrammen . . . 34
4.5.3 Vergleichen von Histogrammen . . . 36
4.6 Vorlagenvergleich . . . 38

4.7 Segmentierung 39
4.7.1 Hintergrundsubtraktion 39
4.7.2 Pyramidensegmentierung 40
4.8 Konturen 42
4.8.1 Konturen finden und zeichnen 42
4.8.2 Konturen vergleichen 44

5 Fazit und Ausblick **48**
5.1 Mobile Augmented Reality: Recognizr 48
5.2 Robotik: Nao Roboter 48

Abkürzungsverzeichnis

BGR	Farbraum, in dem sich die Farbinformation aus Blau (B), Grün (G) und Rot (R) zusammensetzt
BSD-Lizenz	Berkeley Software Distribution Lizenz — Lizenztyp für Open-Source-Software
HSL	Farbraum, in dem sich die Farbinformation aus Farbton (H), Sättigung (S) und Helligkeit (L) zusammensetzt
HSV	Farbraum, in dem sich die Farbinformation aus Farbton (H), Sättigung (S) und Farbwert (V) zusammensetzt
MSVC++	Mircosoft Visual C++
OpenCV	Open Source Computer Vision
PGH	Paarweises geometrisches Histogrammm
RGB	Farbraum, in dem sich die Farbinformation aus Rot (R), Grün (G) und Blau (B) zusammensetzt
ROI	Region of Interest

1 Einleitung

1.1 Ziel dieser Arbeit

Die computerbasierte, automatisierte Bildverarbeitung ist ein aktuelles Forschungsgebiet und bietet daher neue Ansätze und Möglichkeiten für die Steuerung von Computersystemen. Eben diese Aktualität führt jedoch dazu, dass sich die Einarbeitung in das Thema als sehr komplex erweist.

Die Bibliothek OpenCV stellt eine Sammlung von ca. 500 Funktionen für den Einsatzbereich maschinelles Sehen bereit und versucht den Einsatz von maschinellem Sehen zu standardisieren. Zudem erfreut sich OpenCV großer Unterstützung durch Softwarehersteller und Forschungseinrichtungen.

Diese Arbeit soll für Programmierer mit C++ Kenntnissen den Umgang mit Bildverarbeitungsalgorithmen erleichtern, indem die wesentlichen Algorithmen anhand von Beispielquellcodes und Beispielbildern verdeutlicht werden. Die Quellcodebeispiele sind so abgebildet, dass sie sich 1:1 kompilieren und an eigenen Bilddaten testen lassen.

Durch die kurze Erklärung der Funktionen und eigenes Experimentieren soll dem Programmierer der Einstieg in die Bildverarbeitung so einfach wie möglich gestaltet werden.

1.2 Was ist OpenCV?

OpenCV ist eine Bibliothek von Funktionen, die hauptsächlich für den Einsatz in Anwendungen im Bereich maschinelles Sehen (Computer Vision) vorgesehen ist. Die Bibliothek ist in C geschrieben und kann unter Windows, Mac OS X und Linux kompiliert werden. Neben den in C geschriebenen Strukturen und Funktionen existiert ebenfalls eine Klassen- und Namespace-Orientierte Variante für C++. Zur Zeit befinden sich Anbindungen für die Sprachen Python, Ruby und Matlab in der Entwicklung, siehe [7, S. 1]. OpenCV ist unter der BSD-Lizenz veröffentlicht und somit sowohl für private als auch für kommerzielle Zwecke frei nutzbar.

Gestartet wurde OpenCV 1999 als Projekt der Firma Intel, um die bis dahin vorhandenen Bildverarbeitungs- und Erkennungs-Infrastrukturen zu ordnen. Heute zählt OpenCV mit einer Usergruppe mit 40.000 Mitgliedern und über zwei Millionen Downloads quasi zum Standard für Bildverarbeitung.

1.3 Aufbau der Bibliothek

Die Bibliothek besteht im Wesentlichen aus vier Teilen. Abbildung 1.1 zeigt die verschiedenen Bestandteile der Bibliothek sowie eine schematische Darstellung der Zusammenhänge zwischen diesen. Die Teile der Bibliothek können in Programmen weitgehend unabhängig verwendet werden. Neben den Funktionen zur Verarbeitung von Bilddaten (*CV* und *CXCORE*) bietet OpenCV weiterhin die Möglichkeit zur Bildein- und Ausgabe (*HighGUI*) und zur statistischen Analyse und Maschine Learning (*MLL*). Die grundlegenden Datentypen liefert *CXTYPES*.

Ein weiterer Bestandteil der Bibliothek sind experimentelle Funktionen wie dreidimensionale Objektverfolgung oder Gesichtserkennung (*CvAux*). Diese werden in Zukunft möglicherweise in den Hauptzweig *CV* übernommen.

Die vorliegende Ausarbeitung wird Funktionen aus *HighGUI*, *CV* und *CXCORE* und den Strukturen aus *CXTYPES* vorstellen. Als Programmiersprache für die Quellcodebeispiele wird C verwendet, da sich diese besser auf eingebettete Systeme übertragen lassen.

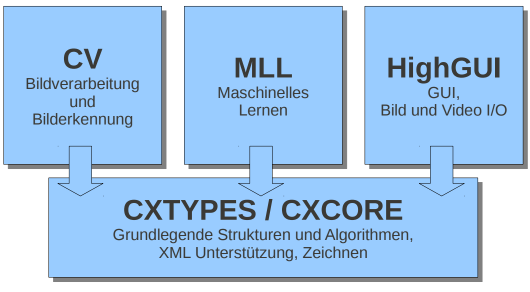


Abbildung 1.1: Die grundlegende Struktur von OpenCV nach [7]

2 Erste Schritte

2.1 Benutzung von OpenCV

Zum Start der Entwicklung von Programmen, die OpenCV nutzen, gibt es zwei Webseiten, die beim Einstieg helfen. Zum einen ist dies die offizielle Homepage [5] und zum anderen die Sourceforge Seite des Projekts [6], auf der die Quellcodes der aktuell stabilen Version zu finden sind. Im Repository befinden sich neben den Quellen für die Bibliothek auch eine Vielzahl von Beispielen.

2.1.1 Download und Installation

Auf der Sourceforge Seite kann das neueste Release als kompilierte Bibliothek für Windows heruntergeladen werden. Für Unix ähnliche Betriebssysteme gibt es ein Paket mit allen Abhängigkeiten. Natürlich kann OpenCV auch direkt aus den Quelldateien compiliert werden, siehe dazu [4]. Laut den Angaben der Entwickler ist OpenCV so geschrieben, dass es sich mit Borland C++, MSVC++, Intel und GNU Kompilern erstellen lässt.

Unter Linux besteht zudem die Möglichkeit, das aktuellste Release mit dem Paketmanager zu installieren.

2.1.2 Kompilieren von Programmen

In dieser Ausarbeitung werden verschiedene Quellcodebeispiele gezeigt. Diese sollten sich mit jedem der oben genannten Kompiler erstellen lassen. Für den GNU Kompiler unter Linux wird an dieser Stelle exemplarisch der benötigte Aufruf gezeigt:

```
1 g++ foo.cpp -I/usr/include/opencv -lcv -lhighgui -o foo
```

Listing 2.1: Kompileraufruf für Beispielcodes

Mit diesem Aufruf können sowohl die meisten Beispiele aus dem OpenCV Repository als auch die im Folgenden gezeigten Codebeispiele kompiliert werden.

3 Datentypen

OpenCV stellt nicht nur Funktionen zur Bildverarbeitung, sondern liefert auch die benötigten Datenstrukturen, um Bilder und Ergebnisse der verschiedenen Bildverarbeitungsoperationen zu speichern. Neben einfachen Datentypen, wie 2D und 3D Punkt mit Integer oder Fließkommagenauigkeit (*CvPoint* und Varianten) bietet OpenCV auch eine Repräsentation für Bilddaten das *IplImage* (siehe dazu IplImage) und *CvSeq*. *CvSeq* wird zum Beispiel zur Speicherung von azyklischen Graphen verwendet (siehe dazu CvSeq).

3.1 Einfache Datentypen

OpenCV bietet für einige grundlegende Daten einfache Strukturen zu deren Speicherung. Die meisten der Funktionen, die OpenCV bietet, nutzen, neben dem Eingabebild, mindestens einen dieser Datentypen als Parameter oder Rückgabewert.

3.1.1 CvPoint

Die Struktur *CvPoint* bietet OpenCV in verschiedenen Varianten an. Eines haben alle diese Varianten gemeinsam: sie stellen einen Punkt in einem Raum dar. Es existieren Implementierungen für zweidimensionale Räume mit Integer und Fließkommagenauigkeit und für dreidimensionale Räume mit einfacher und doppelter Fließkommagenauigkeit. Die einfachste Variante, *CvPoint*, ist zweidimensional und ganzzahlig.

3.1.2 CvScalar

Die Struktur *CvScalar* nutzt OpenCV hauptsächlich zur Darstellung von Farbinformationen. Nähere Informationen zu Farbinformationen sind unter 3.2.1 Darstellung von (Farb-)Bilddaten zu finden. Hier sei zunächst nur angemerkt, dass die Struktur *CvScalar* vier Werte mit doppelter Fließkommagenauigkeit beinhaltet. Zur Initialisierung eines *CvScalar* bietet OpenCV drei Formen:

cvScalar Initialisiert ein *CvScalar*, sodass alle vier Attribute unterschiedliche Werte erhalten können.

cvScalarAll Initialisiert ein *CvScalar*, sodass alle vier Attribute den gleichen Wert haben.

CV_RGB Initialisiert ein *CvScalar*, sodass die Attribute den Wert der übergebenen Farbe im RGB-Farbraum annehmen. Diese Form ist ein Makro, das *cvScalar* aufruft, sodass der erste und dritte Parameter vertauscht werden, da OpenCV intern BGR und nicht RGB verwendet.

3.1.3 CvSize

CvSize dient zur Beschreibung einer rechteckigen Fläche mit Höhe und Breite. Die Struktur existiert ebenfalls mit Integer- und Fließkommagenauigkeit.

3.1.4 CvRect

Ähnlich, wie *cvSize* beschreibt auch *cvRect* eine rechteckige Fläche. *cvRect* besitzt jedoch zusätzlich Koordinaten x und y, die die Position des Rechtecks beschreiben.

3.1.5 CvMat

OpenCV bringt eine eigene Klasse zur Darstellung von Matrizen, *CvMat*. Mit der Funktion *cvCreateMat* kann eine Matrix beliebiger Größe angelegt werden, zudem kann der Datentyp der Matrixelemente ausgewählt werden. *cvCreateMat* allokiert Speicher, der nach der Verwendung mittels der Funktion *cvReleaseMat* wieder freigegeben werden muss.

Der Aufruf der Funktionen *cvCreate[Typ]* und *cvRelease[Typ]* ist typisch für viele der in OpenCV verwendeten Datenstrukturen. Dieses Schema findet beispielsweise auch bei *CvSeq*, siehe 3.3.1 Speicherlayout von Konturen als CvSeq, oder Bilddaten, siehe 3.2.1 Benutzen des IplImage, Anwendung.

3.2 IplImage

Das *IplImage* ist die grundlegende Datenstruktur, die OpenCV verwendet, und wird genutzt, um Bilddaten und Metainformationen, wie Farbraum und Größe, zu speichern. Im Weiteren werden einige Felder dieser Datenstruktur genauer erklärt.

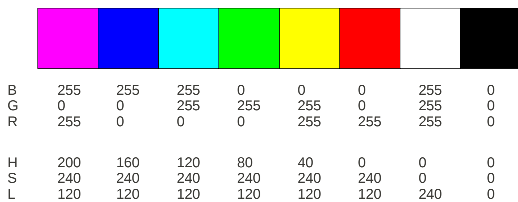
```
1  typedef struct _IplImage
2  {
3      int   nSize;              /* sizeof(IplImage) */
4      int   ID;                 /* version (=0)*/
5      int   nChannels;          /* Most of OpenCV functions support 1,2,3 or 4 channels */
6      int   alphaChannel;       /* Ignored by OpenCV */
7      int   depth;              /* Pixel depth in bits: IPL_DEPTH_8U, IPL_DEPTH_8S,
8                                   IPL_DEPTH_16S, IPL_DEPTH_32S, IPL_DEPTH_32F and
9                                   IPL_DEPTH_64F are supported.  */
10     char colorModel[4];       /* Ignored by OpenCV */
11     char channelSeq[4];       /* ditto */
12     int   dataOrder;          /* 0 - interleaved color channels, 1 - separate color channels.
13                                  cvCreateImage can only create interleaved images */
14     int   origin;             /* 0 - top-left origin,
15                                  1 - bottom-left origin (Windows bitmaps style).  */
16     int   align;              /* Alignment of image rows (4 or 8).
17                                  OpenCV ignores it and uses widthStep instead.    */
18     int   width;              /* Image width in pixels.                           */
19     int   height;             /* Image height in pixels.                          */
20     struct _IplROI *roi;      /* Image ROI. If NULL, the whole image is selected. */
21     struct _IplImage *maskROI;      /* Must be NULL. */
22     void  *imageId;                 /* "           " */
23     struct _IplTileInfo *tileInfo;  /* "           " */
24     int   imageSize;          /* Image data size in bytes
25                                  (==image->height*image->widthStep
26                                  in case of interleaved data)*/
27     char *imageData;          /* Pointer to aligned image data.         */
28     int   widthStep;          /* Size of aligned image row in bytes.    */
29     int   BorderMode[4];      /* Ignored by OpenCV.                     */
30     int   BorderConst[4];     /* Ditto.                                 */
31     char *imageDataOrigin;    /* Pointer to very origin of image data
32                                  (not necessarily aligned) -
33                                  needed for correct deallocation */
34 }
35 IplImage;
```

Listing 3.1: IplImage Struktur

3.2.1 Darstellung von (Farb-)Bilddaten

Bilddaten werden zur Speicherung i.d.R. in Pixel unterteilt. Diese Pixel werden in einer rechteckigen, matrixartigen (zweidimensionalen) Struktur angeordnet. Jeder dieser Pixel hat eine eindeutige Position innerhalb dieser Struktur und enthält die Farbinformation an dieser Stelle des Bildes. Sind die Pixel nur klein genug, so entsteht für den Betrachter der Eindruck eines kontinuierlichen Bildes. Wie die Farben in jedem Pixel codiert werden, hängt vom sogenannten Farbraum ab. Ein Beispiel für Farben und deren Repräsentation in verschiedenen Farbräumen liefert Abbildung 3.1.

Im Folgenden wird der Aufbau der Farbräume RGB und HSL (bzw. HSV) näher erklärt, da diese später Verwendung finden.



B	255	255	255	0	0	0	255	0
G	0	0	255	255	255	0	255	0
R	255	0	0	0	255	255	255	0
H	200	160	120	80	40	0	0	0
S	240	240	240	240	240	240	0	0
L	120	120	120	120	120	120	240	0

Abbildung 3.1: Beispiele für Farben mit den Repräsentationen in BGR und HSV Werten

RGB / BGR

Der wohl am häufigsten verwendete Farbraum ist RGB. Für diesen Farbraum wird die Farbinformation in Anteile von Rot, Grün und Blau aufgeteilt und dann so abgespeichert. Der klare Vorteil dieses Farbraumes ist, dass die Farbinformationen dann direkt dargestellt werden können, indem Licht in den Farben Rot, Grün und Blau in den gespeicherten Intensitäten gemischt wird. Ein Beispiel für die Verwendung dieses Farbraums zur Anzeige von Bilddaten liefert jeder Computerbildschirm. Weitere Informationen zum RGB Farbraum finden sich unter [3].

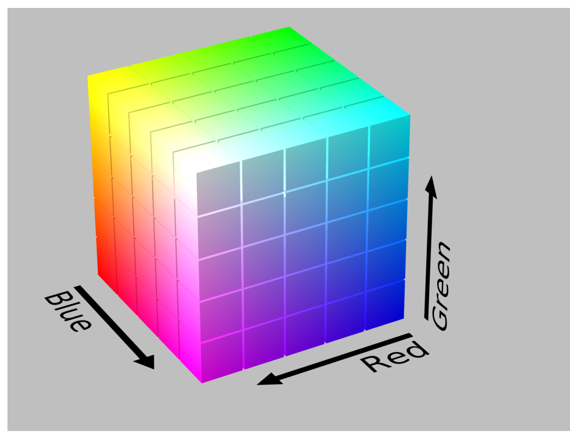


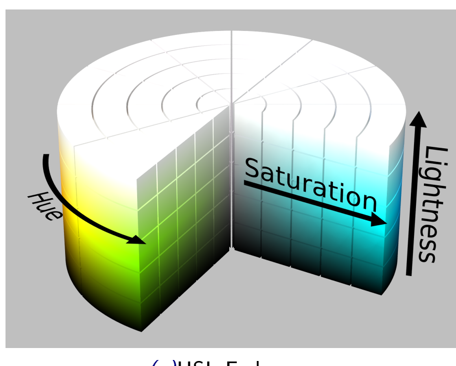
Abbildung 3.2: Graphische Darstellung des RGB Farbraums aus [2]

Die Farbintensität wird i.d.R. pro Farbe als ein ganzzahliger Wert zwischen 0 und 255 gespeichert. Dies entspricht einer Größe von 3 Byte (24 Bit) pro Pixel. Weit verbreitet sind auch Darstellungen mit Werten zwischen 0 und 1. OpenCV nutzt zur Darstellung der Bilddaten nicht RGB, sondern das sehr änliche Format BGR, bei dem nur die Reihenfolge der Werte vertauscht ist. BGR hat den Vorteil, dass der Farbwert äquivalent zur Wellenlänge des Lichts steigt.

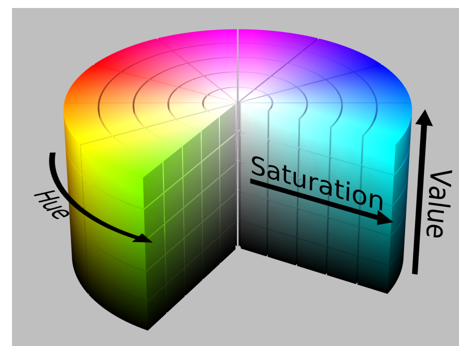
HSL / HSV

Ähnlich wie RGB teilen auch die Farbräume HSL und HSV die Farbinformationen in drei Dimensionen auf. Die in den Werten enthaltenen Informationen sind jedoch grundlegend anders. Abbildung 3.3 zeigt, dass es sich bei HSV und HSL nicht, wie bei RGB, um einen würfelförmigen, sondern einen zylindrischen Farbraum handelt. Der Name HSV wird durch die gespeicherten Komponenten Farbton (Hue), Sättigung (Saturation) und Helligkeit (Lightness) bzw. Wert (Value) induziert. Als Grundlage für beide Zylinder dient ein Farbkreis.

Die Bedeutung des Farbtons ist in beiden Farbräumen die gleiche. Sie beschreibt den Winkel, auf dem die zu speichernde Farbe im Kreis liegt. Rot liegt hier auf 0°, Grün auf 120° und Blau auf 240°. Je nach Implementierung wird der Winkel in Radianten, Grad, zwischen 0 und 1 oder als ganzzahliger Wert zwischen 0 und 180 abgespeichert.



*(a)*HSL Farbraum



*(b)*HSV Farbraum

Abbildung 3.3: HSL und HSV Farbräume aus [2]

Auch die Bedeutung der Sättigung ist in beiden Farbräumen gleich. Sie beschreibt die Entfernung der Farbe von der Kreismitte. Die Sättigung wird häufig mit ganzzahligen Werten zwischen 0 und 255 oder mit Werten zwischen 0 und 1 abgespeichert.

Im Farbwert bzw. der Helligkeit unterscheiden sich die beiden Formate. Während bei HSL die maximale Helligkeit bedeutet, dass die Farbe weiß wird, bedeutet bei HSV ein maximaler Farbwert, dass die Farbe ihre maximale Intensität erreicht. In beiden Fällen bedeutet minimaler Farbwert bzw. Helligkeit jedoch, dass die Farbe schwarz wird. Aus diesem Grund ist die Handhabung des HSL Formats in manchen Fällen intuitiver. Weitere Informationen zum HSV und HSL Farbraum finden sich unter [2].

Ein Pixel im HSL bzw. HSV Farbraum benötigt somit, wie auch im RGB Farbraum, 3 Byte (24 Bit).

Graustufen

Häufig ist es nicht nötig Bildinformationen farbig zu speichern. Zum einen reduziert dies den benötigten Speicherplatz drastisch, zum anderen gibt es eine große Zahl von Algorithmen, für die die Farbinformationen irrelevant sind. Betrachtet man den HSL oder HSV Farbraum, wird klar, dass zur Darstellung eines Graustufenbildes die Komponente der Sättigung komplett wegfallen kann. Fällt die Sättigung weg, kann ebenso der Farbton wegfallen. Der Farbraum reduziert sich so von einem drei auf einen eindimensionalen Raum.

Für die Speicherung eines Graustufen-Pixels reicht in den meisten Fällen eine Unterteilung in 256 Graustufen. Ein Graustufen-Pixel benötigt damit nur 1 Byte (8 Bit).

Speicherlayout des IplImage

Aus den Eckdaten der verschiedenen Farbräume lässt sich schnell schließen, dass eine universelle Datenstruktur zur Speicherung von Bilddaten eine gewisse Flexibilität benötigt. Im Folgenden werden die wichtigsten Felder der IplImage Struktur kurz erklärt.

Die Attribute *width* und *height* stehen, fast selbsterklärend, für die Breite und die Höhe des gespeicherten Bildes in Pixeln. Das Attribut *nChannels* gibt die Dimension des Farbraums, die sogenannten Kanäle, an. Im Fall eines RGB, HSV oder HSL Bildes ist diese beispielsweise drei, bei Graustufenbildern eins. Das letzte wichtige Attribut ist *depth*. Dieses Attribut gibt die Anzahl der Bits an, die pro Pixel verwendet werden, und wie diese interpretiert werden.

Für die Wahl von *depth* gibt es folgende Möglichkeiten:

IPL_DEPTH_8U 8 Bit ohne Vorzeichen (Wertebereich wie *unsigned char*)

IPL_DEPTH_8S 8 Bit mit Vorzeichen (Wertebereich wie *char*)

IPL_DEPTH_16S 16 Bit mit Vorzeichen (Wertebereich wie *short*)

IPL_DEPTH_32S 32 Bit mit Vorzeichen (Wertebereich wie *int*)

IPL_DEPTH_32F 32 Bit Fließkomma (Wertebereich wie *float*)

IPL_DEPTH_64F 64 Bit Fließkomma (Wertebereich wie *double*)

Beim Erstellen eines *IplImage* allokiert OpenCV dann einen Speicherbereich der Größe $width * height * nChannels * depth$. Auf diesen Speicherbereich kann dann direkt über den Zeiger *imageData* zugegriffen werden. So kann einfach ein pixelweiser Durchlauf durch das Bild realisiert werden:

```
void pixelwise(IplImage* img){
    for(int y=0; y<img->height; y++){
        unsigned char* ptr = (unsigned char*) (img->imageData + y * img->widthStep);
        for(int x=0; x<img->width; x++){
            /* Bei drei Kanaelen enthalten
               ptr[3*x], ptr[3*x+1] und ptr[3*x+2]
               die Farbinformationen aus dem 1., 2. und 3. Kanal. */
        }
    }
}
```

Listing 3.2: Pixelweiser Durchlauf durch ein IplImage vom Typ IPL_DEPTH_8U mit drei Kanälen

Im Listing 3.2 fällt noch ein Attribut der Struktur *IplImage* auf, welches noch keine Erwähnung fand. *widthStep* gibt die Länge einer Zeile in Bytes an. Diese Angabe ist im Prinzip eine Kurzform von $width * nChannels * depth$. Um einen fehlerfreien Durchlauf zu garantieren sollte jedoch immer *widthStep* verwendet werden. Die Adressierung der einzelnen Farbwerte erfolgt indem der Zeiger *ptr* (Zeilenanfang) für jeden Pixel um die Anzahl der Kanäle, z.B. drei bei RGB und HSV oder eins bei Grauwerten, verschoben wird.

Benutzen des IplImage

Eine IplImage Struktur kann mit der Funktion *cvCreateImage* unter Angabe der Breite, der Höhe, der Farbkanäle und des Datentyps erstellt werden. Die Datentypen, die für das IplImage zur Verfügung stehen, werden unter 3.2.1 Speicherlayout des IplImage aufgeführt. Neben dem Erstellen eines leeren Bildes können die Bilddaten direkt aus einer Bilddatei gelesen werden. Dieser Vorgang wird unter 4.1.1 Bilder lesen und anzeigen beschrieben. Ist die Bearbeitung der Bilddaten beendet, so muss der allokierte Speicher mit der Funktion *cvReleaseImage* wieder freigegeben werden.

Maskierung

Neben der Bearbeitung des kompletten Bildes bietet OpenCV die Möglichkeit Regionen des Bildes zu maskieren und somit die Bearbeitung auf einzelne Bildteile zu beschränken. Diese Vorgehensweise kann die Verarbeitungszeit reduzieren und erspart unnötiges Umkopieren der Bilddaten.

Mit den Funktionen *cvSetImageROI* und *cvResetImageROI* können rechteckige Masken direkt auf einem Bild gesetzt bzw. entfernt werden. Weiterhin bieten viele Funktionen einen *mask* Parameter an, der durch ein Binärbild (Schwarz-Weiß) dargestellt wird, das die selbe Größe hat wie die Bilddaten. Die Verarbeitung der Bilddaten erfolgt dann nur dort, wo das Binärbild einen Wert größer null hat.

3.3 CvSeq

Eine Sequenz wird in OpenCV genutzt, um eine Menge von Daten strukturiert zu speichern. Der Datentyp kann quasi frei* ausgewählt werden. Da die verschiedenen Datentypen nur mehr oder weniger Verwendung in OpenCV finden, soll hier im Detail nur auf einen wichtigen, die Konturen, eingegangen werden.

*OpenCV liefert für *CvSeq* 9 Datentypen. Weitere Informationen liefert [7, S. 224]

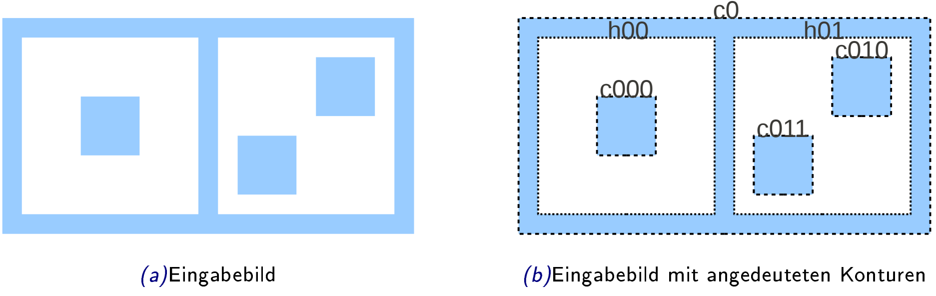
```
typedef struct CvSeq {
    int flags;                 /* Miscellaneous flags.                */
    int header_size;           /* Size of sequence header.            */
    struct CvSeq* h_prev;      /* Previous sequence.                  */
    struct CvSeq* h_next;      /* Next sequence.                      */
    struct CvSeq* v_prev;      /* 2nd previous sequence.              */
    struct CvSeq* v_next       /* 2nd next sequence.                  */
    int total;                 /* Total number of elements.           */
    int elem_size;             /* Size of sequence element in bytes.  */
    char* block_max;           /* Maximal bound of the last block.    */
    char* ptr;                 /* Current write pointer.              */
    int delta_elems;           /* Grow seq this many at a time.       */
    CvMemStorage* storage;     /* Where the seq is stored.            */
    CvSeqBlock* free_blocks;   /* Free blocks list.                   */
    CvSeqBlock* first;         /* Pointer to the first sequence block. */
} CvSeq;
```

Listing 3.3: CvSeq Struktur

Eine Kontur besteht aus Punkten (bzw. Knoten oder Vertices), deren Verbindung eine geschlossene Fläche bildet. Eine solche Reihe von Punkten wird auch Polygon genannt. OpenCV kann solche Konturen selbstständig aus präparierten Bildern erstellen.

Konturen sind ein häufig verwendetes Werkzeug in der Bildverarbeitung. Sie werden genutzt, um Teile des Bildes abzugrenzen, und sind somit das Ergebnis einer Vielzahl von Bildverarbeitungsalgorithmen. Genauere Informationen zur Extraktion von Konturen können unter 4.8 gefunden werden.

OpenCV ordnet Konturen untereinander zudem in hierarchischen Strukturen an. Werden Konturen aus einem Bild extrahiert, so liefert OpenCV eine *CvSeq* von Konturen. Ein Beispiel dafür ist in Abbildung 3.4 zu sehen*.



*(a)*Eingabebild *(b)*Eingabebild mit angedeuteten Konturen

Abbildung 3.4: Beispielbild und gefundene Konturen

3.3.1 Speicherlayout von Konturen als CvSeq

Die gefundenen Konturen werden in “Konturen” (in Abbildung 3.4b beschriftet mit c) und Löcher (in Abbildung 3.4b beschriftet mit h) unterteilt und hierarchisch Angeordnet. OpenCV kennt vier verschiedene Anordnungen von Konturen:

CV_RETR_EXTERNAL Nur die äußerste Kontur wird gesucht.

CV_RETR_LIST Die gefundenen Konturen und Löcher werden als Liste zurückgegeben.

CV_RETR_CCOMP Die gefundenen Konturen werden als Liste zurückgegeben, die Löcher werden als Kinder der Konturen behandelt, die sie enthalten.

CV_RETR_TREE Eine komplette Hierarchie wird zurückgegeben. Löcher, die Konturen enthalten, werden als deren Eltern behandelt.

*In Abbildung 3.4b wurden die Konturen aufgrund der Übersichtlichkeit benannt.

Die verschiedenen Anordnungen werden am Beispiel aus Abbildung 3.4 in Abbildung 3.5 demonstriert.

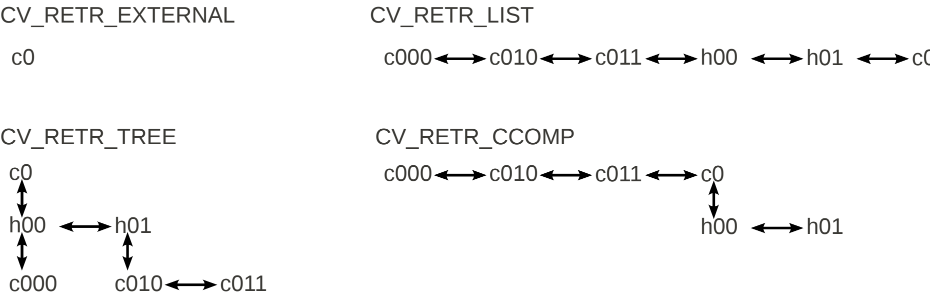


Abbildung 3.5: Verschiedene Anordnungen von Konturen

Über die Attribute *v_next* und *v_prev* kann der Graph vertikal durchlaufen werden, mit *h_next* und *h_prev* horizontal.

CvMemStorage

Da die Graphen, die mittels *CvSeq* beschrieben werden, häufig dynamisch wachsen, wird eine Struktur benötigt, die die nachträgliche Allokation von Speicher vornimmt. Diese Aufgabe übernimmt die Struktur *CvMemStorage*. Bevor mit *CvSeq* gearbeitet werden kann, muss immer ein solcher Speicherbereich mit der Funktion *cvCreateMemStorage* erstellt werden. Freigegeben wird dieser dynamische Speicherbereich mit der Funktion *cvReleaseMemStorage*.

4 Funktionsumfang

Eine häufige Aufgabe von Programmen, die Bilddaten verarbeiten, ist die Erkennung und Verfolgung von Objekten. Zu diesem Zweck gibt es mehrere Strategien, wie Objekte aus Bilddaten erkannt werden können. Diese sind:

- Konturenerkennung
- Segmentierung
- Vorlagenvergleich

Die Strategien können für sich alleine benutzt werden. Es bietet sich jedoch an sie zu kombinieren, um die Genauigkeit des Algorithmus zu erhöhen.

Der Funktionsumfang von OpenCV bietet eine Vielzahl von Bildverarbeitungsalgorithmen, die Teile dieser Aufgaben erfüllen. Einige besonders anschauliche oder solche, die häufige Verwendung finden, werden im Folgenden erklärt.

4.1 Ein- und Ausgabe

Das Lesen, Speichern und Anzeigen von Bildern ist in OpenCV möglichst einfach gehalten. So bietet die Bibliothek einfache Lese- und Speicherfunktionen für Bilder und Videos aus bzw. in Dateien und eine rudimentäre Implementierung von Fenstern zur Anzeige von Bilddaten.

Neben diesen Funktionen zur Ein- und Ausgabe von Daten bietet OpenCV Funktionen zum Zeichnen einfacher Formen.

4.1.1 Bilder lesen und anzeigen

Das Beispiel aus Listing 4.1 zeigt ein Grundgerüst vieler Programme, die OpenCV nutzen. Zunächst wird ein Bild aus einer Datei geladen, dann angezeigt und zum Schluss der von den Daten belegte Speicher wieder freigegeben.

```
1  #include <highgui.h>

3  int main(int argc, char** argv){
4      IplImage* img = cvLoadImage(argv[1]);
5      cvNamedWindow("Hello OpenCV",CV_WINDOW_AUTOSIZE);
6      cvShowImage("Hello OpenCV",img);
7      cvWaitKey(0);
8      cvSaveImage("test.png",img);
9      cvReleaseImage(&img);
10     cvDestroyWindow("Hello OpenCV");
11 }
```

Listing 4.1: Öffnen und Anzeigen eines Bildes

Die in Listing 4.1 verwendeten Funktionen werden im Folgenden kurz erklärt. Die Funktion *cvLoadImage* lädt das Bild aus einer Datei in den Programmspeicher und gibt einen Zeiger auf eine *IplImage* Struktur zurück. Der so belegte Speicher muss, wenn die Bilddaten nicht mehr benötigt werden, mit der Funktion *cvReleaseImage* wieder freigegeben werden.

Die restlichen Funktionen dienen der Anzeige des geladenen Bildes auf dem Bildschirm. *cvNamedWindow* erstellt ein neues Fenster. Die Fenster werden von OpenCV intern verwaltet und

können über den Namen (bzw. Titel), referenziert werden. Das Anzeigen des Bildes wird mit *cvShowImage* erreicht. Da keine der Funktionen die Ausführung blockiert, würde das Fenster nach der Ausführung sofort verschwinden. *cvWaitKey* blockiert, bis der Benutzer eine Taste drückt oder ein Zeitintervall in Millisekunden abgelaufen ist. Das angelegte Fenster wird dann mit *cvDestroyWindow* freigegeben.

4.1.2 Videos als Eingabedaten

Neben einzelnen Bildern können auch Bilddaten aus Videos und von Kameras gelesen werden. Das Listing 4.2 zeigt einen Quelltext, der Bilder aus einer Videodatei liest und auf dem Bildschirm anzeigt.

```
1  #include <highgui.h>

3  int main(int argc, char** argv){
4      CvCapture *cap = cvCreateFileCapture(argv[1]);
5      cvNamedWindow("Hello OpenCV",CV_WINDOW_AUTOSIZE);
6      IplImage* img = cvQueryFrame(cap);
7      while(img != 0){
8          cvShowImage("Hello OpenCV",img);
9          cvWaitKey(100);
10         img = cvQueryFrame(cap);
11     }
12     cvDestroyWindow("Hello OpenCV");
13     cvReleaseCapture(&cap);
14 }
```

Listing 4.2: Öffnen und Anzeigen eines Videos

Für das Lesen von Videodateien muss die Datei zunächst geöffnet werden. Die Funktion *cvCreateFileCapture* initialisiert eine *CvCapture* Struktur, aus der ein Video bildweise gelesen werden kann. *cvQueryFrame* gibt dann das nächste Bild zurück.

Neben Videodateien können auch Webcams angesprochen werden, indem die *CvCapture* mit der Funktion *cvCreateCameraCapture* initialisiert wird.

4.1.3 Zeichnen

Um die Ergebnisse von Bildanalysen zu zeigen, ist es üblich z.B. gefundene Objekte oder markante Punkte direkt in die Bilder zu zeichnen. OpenCV bietet dafür Zeichenfunktionen unter anderem für verschiedene geometrische Formen, wie zum Beispiel:

cvCircle zum zeichnen von Kreisen.

cvEllipse zum zeichnen von Ellipsen.

cvLine zum zeichnen von Linien.

cvRectangle zum zeichnen von Rechtecken.

cvPolyline zum zeichnen von Polygonen.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
6      cvNamedWindow("Hello OpenCV",CV_WINDOW_AUTOSIZE);

8      CvScalar color = CV_RGB(255,0,0);
9      cvCircle(img, cvPoint(100,100), 50, color);
10     cvEllipse(img, cvPoint(100,250), cvSize(50,100),75,45,270,color);
11     cvLine(img, cvPoint(300,100), cvPoint(400,400), color);
```

```
13      cvShowImage("Hello OpenCV",img);
14      cvWaitKey(0);

16      cvSaveImage("drawing.png",img);

18      cvReleaseImage(&img);
19      cvDestroyWindow("Hello OpenCV");
20  }
```

Listing 4.3: Erstellen eines Bildes, zeichnen einiger Formen und speichern des Bildes

Listing 4.3 zeigt die Benutzung einiger Zeichenfunktionen. Weitere Zeichenfunktionen, die OpenCV bietet, benötigen Ergebnisse von Bildverarbeitungsalgorithmen und werden an entsprechender Stelle vorgestellt, siehe dazu auch 4.8 Konturen.

Text zeichnen

Neben geometrischen Formen bietet OpenCV die Möglichkeit Text auf Bilder zu schreiben. Hierfür muss zunächst eine Struktur vom Typ *CvFont* initialisiert werden, die Schriftgröße und Schriftart bestimmt. Listing 4.4 zeigt die Anwendung der Funktion *cvPutText* zum Zeichnen von Text in ein Bild.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* img = cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
6       cvNamedWindow("Hello OpenCV",CV_WINDOW_AUTOSIZE);

8       CvScalar color = CV_RGB(255,0,0);
9       CvFont font;
10      cvInitFont(&font, CV_FONT_HERSHEY_PLAIN, 1,1);

12      cvPutText(img, "Hello OpenCV", cvPoint(100,50), &font, color);

14      cvShowImage("Hello OpenCV",img);
15      cvWaitKey(0);

17      cvSaveImage("drawing.png",img);

19      cvReleaseImage(&img);
20      cvDestroyWindow("Hello OpenCV");
21  }
```

Listing 4.4: Platzieren von Text auf einem Bild

4.1.4 Daten speichern

Das in Listing 4.3 gezeigte Programm erzeugt zunächst ein leeres Bild. Dies geschieht mit der Funktion *cvCreateImage*. Nachdem die Formen in 4.1.3 Zeichnen auf das Bild gezeichnet wurden, wird das Bild mit der Funktion *cvSaveImage* als Datei gespeichert. Ähnlich wie ein einzelnes Bild lässt sich eine Reihe von Bildern als Videodatei speichern.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* img = cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
6       cvNamedWindow("Hello OpenCV",CV_WINDOW_AUTOSIZE);
7       CvScalar color = CV_RGB(255,0,0);
8       CvVideoWriter* writer =
            cvCreateVideoWriter("save_video.avi",CV_FOURCC('M','J','P','G'),10,cvSize(640,480));

10      for(int i=0; i<100; i++){
11          cvZero(img);
12          cvCircle(img, cvPoint(100+i,100), 50, color);
13          cvWriteFrame(writer, img);
14          cvShowImage("Hello OpenCV",img);
```

```
        cvWaitKey(100);
    }

    cvSaveImage("drawing.png",img);
    cvReleaseVideoWriter(&writer);
    cvReleaseImage(&img);
    cvDestroyWindow("Hello OpenCV");
}
```

Listing 4.5: Anzeigen und Speichern einer Abfolge von Bildern als Videodatei

Listing 4.5 zeigt die zum Speichern von Videos nötigen Aufrufe. Mit der Funktion *cvCreateVideoWriter* wird eine Struktur initialisiert, mit der über die Funktion *cvWriteFrame* bildweise in eine Videodatei geschrieben werden kann. Abgeschlossen wird die Datei dann beim Freigeben der Struktur durch *cvReleaseVideoWriter*.

4.2 Bildmanipulationen

Mit den Funktionen aus dem vorherigen Kapitel ist es nun möglich Eingabedaten zu lesen und Ergebnisse auf dem Bildschirm oder als Datei auszugeben. Dieser Abschnitt zeigt nun grundlegende Bildmanipulationen, die in Bildverarbeitungsalgorithmen Verwendung finden. Viele dieser Algorithmen sind zudem bekannt aus Bildbearbeitungsprogrammen.

Solche Bildmanipulationen werden häufig als Filter bezeichnet. Filter werden benutzt um die Charakteristiken von Bilddaten zu verbessern und die Auswertung zu erleichtern (oder überhaupt möglich zu machen). Dabei greifen die Filter auf mathematische Abbildungen zurück, die lokal auf einzelne Pixel bzw. deren Umgebung angewendet werden. Für die Abbildung verwenden fast alle Filter sogenannte Kernels. Ein Kernel beschreibt die Größe und Form der zu betrachtenden Umgebung und die Lage eines sogenannten Ankers in der Umgebung. Die Position des Ankers markiert den aktuellen Pixel.

In den folgenden Abschnitten werden die verschiedenen Algorithmen anhand von Beispielbildern erklärt. Die Originalbilder sind in Abbildung 4.1a bis 4.1c dargestellt*.

(a) Foto einer Replik von K.I.T.T. *(b)* Bild von einer Gummiente, aufgenommen mit der Kamera eines NAO Roboters *(c)* Testbild

Abbildung 4.1: Verwendete Beispielbilder

4.2.1 Weichzeichnen

Bei der Aufnahme von realen Bilddaten kommt es häufig zu Rauschen. Diese Bildstörung hat massiven Einfluss auf die Auswertung von Bilddaten. Eine Möglichkeit, die Auswirkungen zu verringern ist das Weichzeichnen der Bilddaten. Dabei werden die Werte der nebeneinander liegenden Pixel miteinander verrechnet. Wie genau die Pixel miteinander verrechnet werden, hängt

*zu Abbildung 4.1a siehe http://de.wikipedia.org/wiki/Knight_Rider

von der Weichzeichnungsmethode ab. OpenCV unterstützt verschiedene Typen von Weichzeichnern: *einfacher Weichzeichner*, *medialer Weichzeichner*, *gaußscher Weichzeichner* und *bilateraler Weichzeichner*.

Listing 4.6 zeigt einen Quellcode, der alle erwähnten Weichzeichner auf ein Bild anwendet. Die Ergebnisse aus diesem Code sind in Abbildung 4.2, 4.3, 4.4 und 4.5 zu sehen.

```
#include <highgui.h>
#include <cv.h>

int main(int argc, char** argv){
    IplImage* img = cvLoadImage(argv[1]);
    IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
    cvNamedWindow("Blurring",CV_WINDOW_AUTOSIZE);

    cvSmooth(img, dst, CV_BLUR, 9, 9);
    cvShowImage("Blurring",dst);
    cvWaitKey(1000);
    cvSaveImage("smoothing_blur.png", dst);

    cvSmooth(img, dst, CV_MEDIAN, 9, 9);
    cvShowImage("Blurring",dst);
    cvWaitKey(1000);
    cvSaveImage("smoothing_median.png", dst);

    cvSmooth(img, dst, CV_GAUSSIAN, 9, 9);
    cvShowImage("Blurring",dst);
    cvWaitKey(1000);
    cvSaveImage("smoothing_gaussian.png", dst);

    cvSmooth(img, dst, CV_BILATERAL, 15, 7);
    cvShowImage("Blurring",dst);
    cvWaitKey(1000);
    cvSaveImage("smoothing_bilateral.png", dst);

    cvReleaseImage(&img);
    cvReleaseImage(&dst);
    cvDestroyWindow("Blurring");
}
```

Listing 4.6: Verschiedene Weichzeichner im Einsatz

Einfaches Weichzeichnen

Die in Abbildung 4.2 gezeigten Bilder entstehen durch einfaches Weichzeichnen. Dabei wird in einer Umgebung* um jeden Pixel der Durchschnitt der Pixel berechnet. Dieser Wert wird dann dem mittleren Pixel zugewiesen. Besonders an diesem Filter ist, dass nur lineare Operationen auf den Pixeln durchgeführt werden, sodass dieser Filter am wenigsten Rechenzeit beansprucht. Lineare Filter bezeichnet man auch als Faltung, eine genauere Beschreibung dieses Ablaufes findet sich in 4.2.4 Faltung.

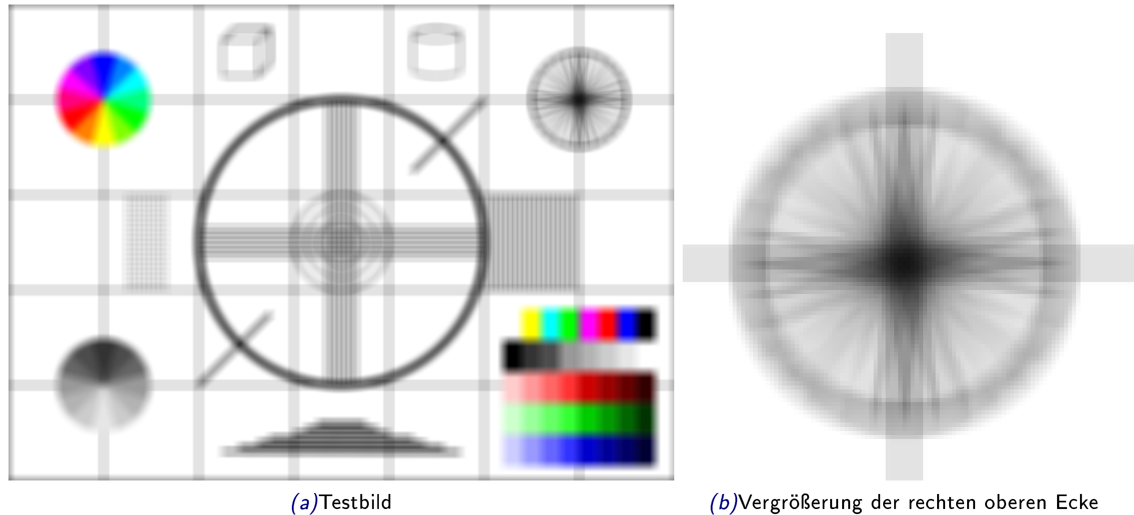
Der Nachteil dieses Filters wird in Abbildung 4.2b deutlich. Die Deutlichkeit wichtiger Merkmale, wie Kanten, wird genau so reduziert wie das Bildrauschen.

Mediales Weichzeichnen

Ähnlich wie der einfache Weichzeichner geht auch der mediale Weichzeichner über jeden Pixel seiner Umgebung. Dabei wird jedoch nicht der Durchschnitt, sondern der Median der Pixelwerte verwendet. Abbildung 4.3 zeigt die Ergebnisse. Durch diesen Weichzeichner wird das Bild in große Flächen gleicher Farbe unterteilt. Dies kann für Segmentierungsalgorithmen genutzt werden.

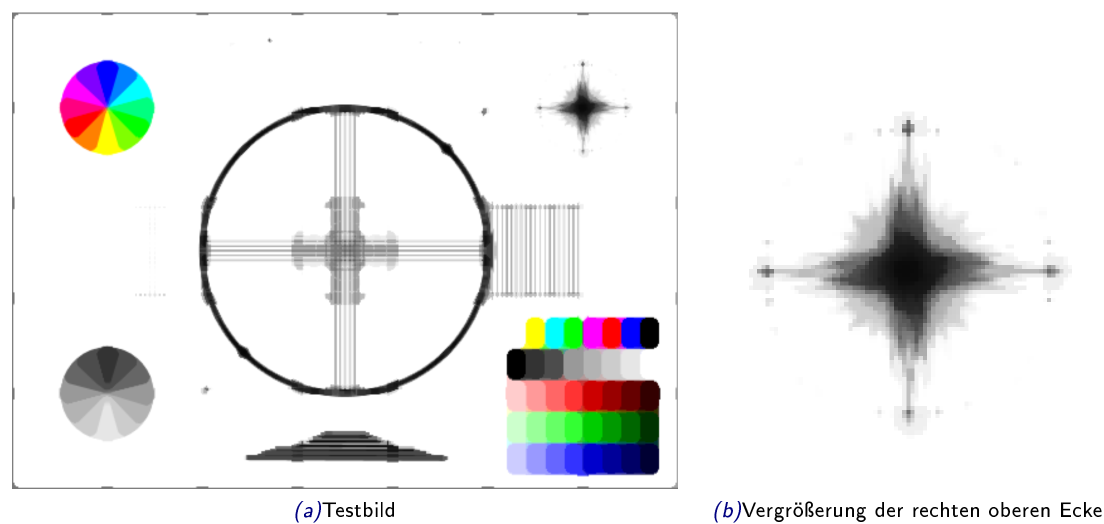
Dennoch bietet der mediale Weichzeichner einen großen Nachteil. Feine Strukturen gehen, wie beim einfachen Weichzeichner, komplett verloren. Mediales Weichzeichnen zur Aufbereitung eines Bildes, auf dem feine Kanten, wie etwa Barcodes, gefunden werden sollen, ist demnach ungünstig.

*In diesem Fall ist die Umgebung ein Rechteck. Die Kantenlänge lässt sich über die Parameter der Funktion *cvSmooth* angeben.



*(a)*Testbild *(b)*Vergrößerung der rechten oberen Ecke

Abbildung 4.2: Ergebnisse des einfachen Weichzeichners



*(a)*Testbild *(b)*Vergrößerung der rechten oberen Ecke

Abbildung 4.3: Ergebnisse des medialen Weichzeichners

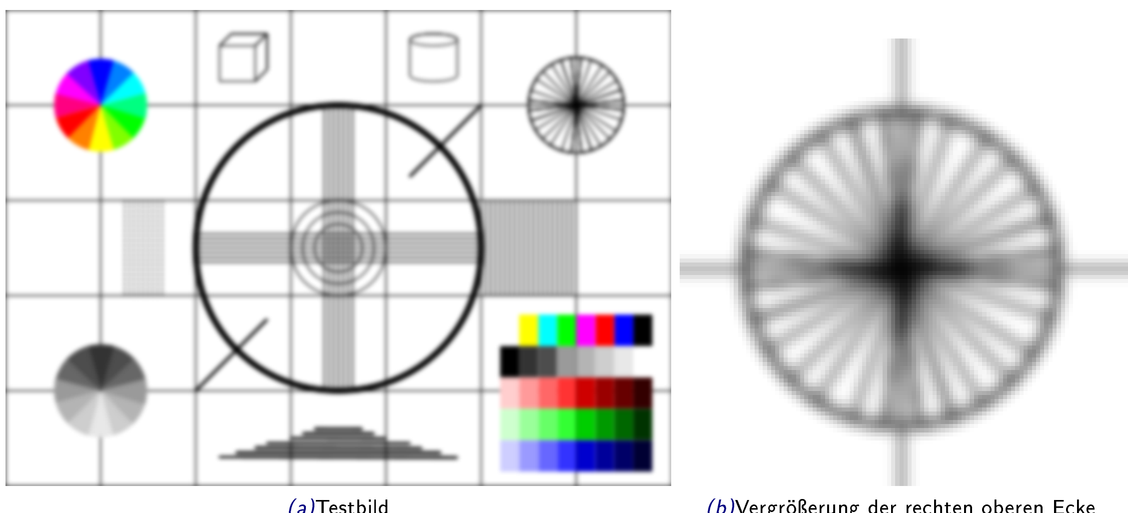
Gaußsches Weichzeichnen

Der gaußsche Weichzeichner bildet wie der einfache Weichzeichner einen Mittelwert, jedoch gewichtet er die Pixel mit der Entfernung zum Mittelpunkt der Umgebung. Die Gewichtung bewirkt, dass Kanten, die im Gegensatz zu Rauschen eine räumliche Kontinuität heben, nicht so stark weichgezeichnet werden. Abbildung 4.4 zeigt ein Beispiel, auf das der gaußsche Weichzeichner angewendet wurde.

Bilaterales Weichzeichnen

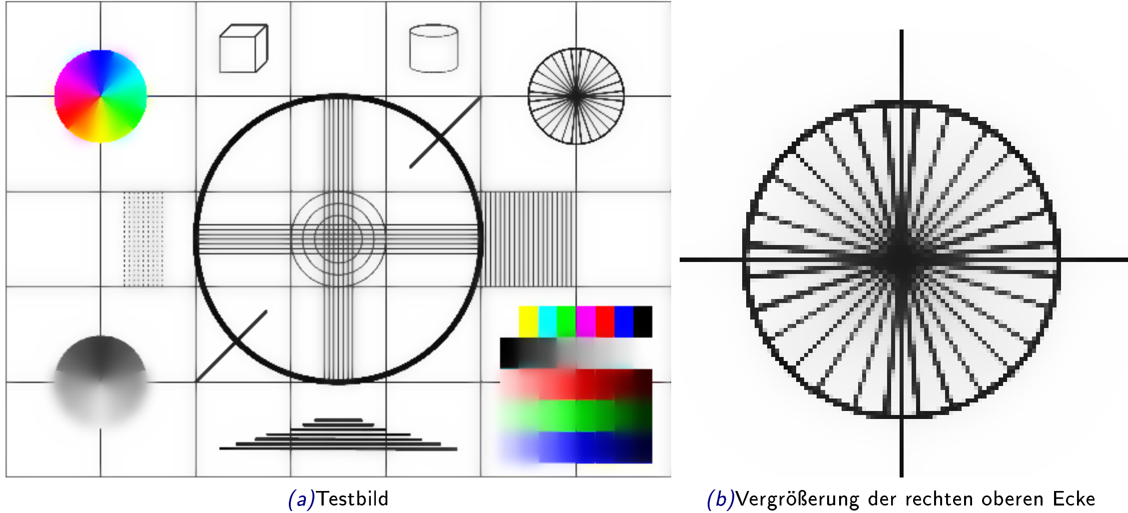
Der bilaterale Weichzeichner benutzt einen adaptiven Algorithmus. Kleine Störungen auf großen Flächen werden eliminiert. Große Störungen, die markante Kanten in den Bilddaten darstellen könnten, bleiben bestehen Die in Abbildung 4.5 dargestellten Bilder zeigen die Ergebnisse der bilateralen Filterung*.

*Zum Zeitpunkt des Verfassens dieses Textes gibt es scheinbar einen Fehler in der Implementierung des bilateralen Weichzeichners, sodass die erzeugten Bilder komplett schwarz sind. Die gezeigten Bilder wurden mit dem



*(a)*Testbild *(b)*Vergrößerung der rechten oberen Ecke

Abbildung 4.4: Ergebnisse des gaußschen Weichzeichners



*(a)*Testbild *(b)*Vergrößerung der rechten oberen Ecke

Abbildung 4.5: Ergebnisse des bilateralen Weichzeichners

Der bilaterale Filter lässt sich zudem sinnvoll iterativ auf die Bilddaten anwenden und erzeugt eine immer bessere Weichzeichnung mit Erhaltung feiner Strukturen. Die guten Eigenschaften der bilateralen Filterung gehen jedoch sehr zulasten der Rechenzeit.

4.2.2 Schwellwerte

Schwellwerte haben zwei Anwendungsbereiche in der Bildverarbeitung. Zum Einen können sie eingesetzt werden, um Bilddaten in Binärbilder (schwarz oder weiß) umzuwandeln. Diese Umwandlung ist besonders dann nützlich, wenn nur einfache Formen, Text oder Zahlen erkannt werden sollen. Zum Anderen kann ein Schwellwert genutzt werden, um Areale eines Bildes zu klassifizieren, die bereits von einem Bildverarbeitungsalgorithmus verarbeitet wurden. Ein solcher Algorithmus ist zum Beispiel das Template Matching, das unter 4.6 Vorlagenvergleich beschrieben wird.

Grafikprogramm GIMP nachgestellt.

```
#include <highgui.h>
#include <cv.h>

int main(int argc, char** argv){
    IplImage* img = cvLoadImage(argv[1]);
    IplImage* tmp = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
    IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U,
        tmp->nChannels);
    cvNamedWindow("Threshold",CV_WINDOW_AUTOSIZE);

    cvCvtColor(img, tmp, CV_BGR2GRAY);
    cvSaveImage("threshold_in.png", tmp);

    cvThreshold(tmp, dst, 128, 255, CV_THRESH_BINARY);
    cvShowImage("Threshold", dst);
    cvWaitKey(1000);
    cvSaveImage("threshold_binary.png", dst);

    cvThreshold(tmp, dst, 100, 255, CV_THRESH_TRUNC);
    cvShowImage("Threshold", dst);
    cvWaitKey(1000);
    cvSaveImage("threshold_trunc.png", dst);

    cvThreshold(tmp, dst, 128, 255, CV_THRESH_TOZERO);
    cvShowImage("Threshold", dst);
    cvWaitKey(1000);
    cvSaveImage("threshold_tozero.png", dst);

    cvAdaptiveThreshold(tmp, dst, 255, CV_ADAPTIVE_THRESH_MEAN_C, CV_THRESH_BINARY, 71, 15);
    cvShowImage("Threshold", dst);
    cvWaitKey(1000);
    cvSaveImage("threshold_adaptive.png", dst);

    cvReleaseImage(&img);
    cvReleaseImage(&tmp);
    cvReleaseImage(&dst);
    cvDestroyWindow("Threshold");
}
```

Listing 4.7: Quellcode, der verschiedene Schwellwerte auf ein Bild anwendet

*(a)*Quellbild *(b)*Nach binärem Schwellwert

Abbildung 4.6: Quellbild und Ergebnis nach binärem Schwellwert

Listing 4.7 zeigt die Verwendung der Funktion *cvThreshold*, die die Schwellwertanalyse durchführt. Der gezeigte Algorithmus konvertiert das Bild zunächst in Graustufen, damit das Ergebnis anschaulicher ist. Wird ein Bild mit Farbkanälen mit der Funktion *cvThreshold* verarbeitet, so wird jeder Kanal einzeln betrachtet.

Binärer Schwellwert

OpenCV bietet als einfachste Schwellwert Operation den binären Schwellwert, dabei wird jeder Wert unter dem Schwellwert auf null (schwarz) und jeder Wert über dem Schwellwert auf einen Maximalwert (weiß) gesetzt. Der Maximalwert hängt vom verwendeten Farbraum ab.

*(a)*Nach abgeschnittenem Schwellwert

*(b)*Nach "Null" Schwellwert

Abbildung 4.7: Weitere Schwellwerte: abgeschnitten und "Null"

Andere Schwellwerte

Neben dem binären Schwellwert bietet OpenCV noch weitere Möglichkeiten, wie die Werte ober- und unterhalb des Schwellwertes zu behandeln sind.

CV_THRESH_BINARY Setzt alle Werte über dem Schwellwert auf einen Maximalwert und alle Werte unter dem Schwellwert auf null. Siehe 4.2.2 Binärer Schwellwert.

CV_THRESH_BINARY_INV Setzt alle Werte über dem Schwellwert auf null und alle Werte unter dem Schwellwert auf einen Maximalwert.

CV_THRESH_TRUNC Setzt alle Werte über dem Schwellwert auf den Schwellwert, die Werte unter dem Schwellwert bleiben gleich.

CV_THRESH_TOZERO Setzt alle Werte unter dem Schwellwert auf null, die Werte über dem Schwellwert bleiben gleich.

CV_THRESH_TOZERO_INV Setzt alle Werte über dem Schwellwert auf null, die Werte unter dem Schwellwert bleiben gleich.

Adaptiver Schwellwert

OpenCV bietet auch die Möglichkeit den binären Schwellwert adaptiv zu berechnen. Es wird dann kein absoluter Schwellwert angegeben, sondern für einzelne Bildbereiche automatisch ein Schwellwert bestimmt. Die Größe des Bildbereichs wird der Funktion als Parameter übergeben.

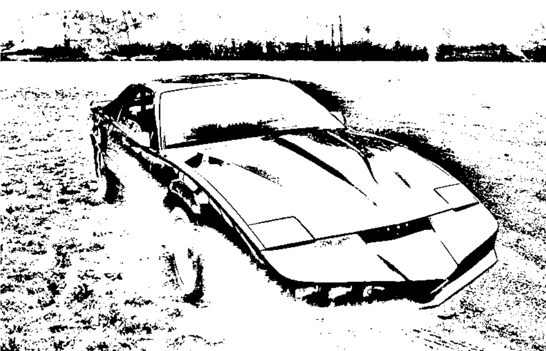
4.2.3 Morphologie

Hinter dem Stichwort Morphologie stehen zwei weniger bekannte Filter, die Dilatation und die Erosion. Beide Verfahren können, wie auch die Weichzeichner, dazu genutzt werden Elemente in Bilddaten, die durch Rauschen verfälscht wurden, zu verdeutlichen. Die Vorgehensweise der morphologischen Filter ist jedoch etwas anders.

Dilataion und Erosion

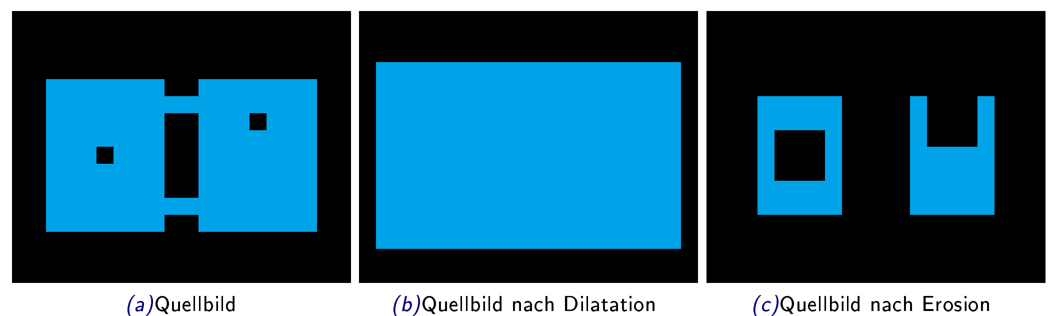
Für die Dilatation wird zuvor bei den Weichzeichnern jeder Pixel und die Pixel in einer Umgebung in den Bilddaten betrachtet und aus diesen ein neuer Wert berechnet, indem das Maximum in der Umgebung bestimmt wird. Die Erosion verfährt gleich, jedoch wird anstatt des Maximums das Minimum berechnet.

*(a)*Nach binärem Schwellwert



*(b)*Nach adaptivem, binärem Schwellwert in einer Umgebung von 71×71 Pixeln

Abbildung 4.8: Ergebnisse von *cvThreshold* bei globalem und adaptivem Schwellwert



*(a)*Quellbild *(b)*Quellbild nach Dilatation *(c)*Quellbild nach Erosion

Abbildung 4.9: Quellbild mit 20x20 Pixeln und die Ergebnisse nach Dilatation und Erosion mit einem quadratischen 3x3 Kernel

Aus den Grundoperationen Dilatation und Erosion lassen sich weitere morphologische Operationen ableiten, das Opening, das Closing, der morphologische Gradient, Top Hat und Black Hat. Diese fünf Operationen sind Kombinationen aus Dilatation und Erosion.

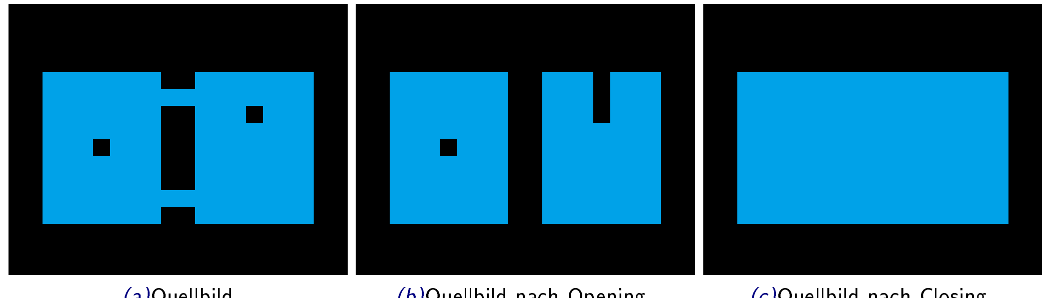
Opening und Closing

Die einfachste Kombination aus Dilatation und Erosion sind das Opening und Closing. Die beiden Grundoperationen werden dabei hintereinander angewendet: Für das Opening erst die Erosion, dann Dilatation — das Closing verfährt genau andersherum und führt erst eine Dilatation und dann eine Erosion durch. Diese Operationen werden häufig zur Verbesserung von Binärbildern verwendet, da mittels dieser Operationen Rauschen und Bildfehler sehr effektiv reduziert werden können.

Das Opening löst kleine Verbindungen zwischen größeren Flächen, das Closing füllt Löcher in Flächen. Der Effekt kann in Abbildung 4.10 nachvollzogen werden.

Top Hat und Black Hat

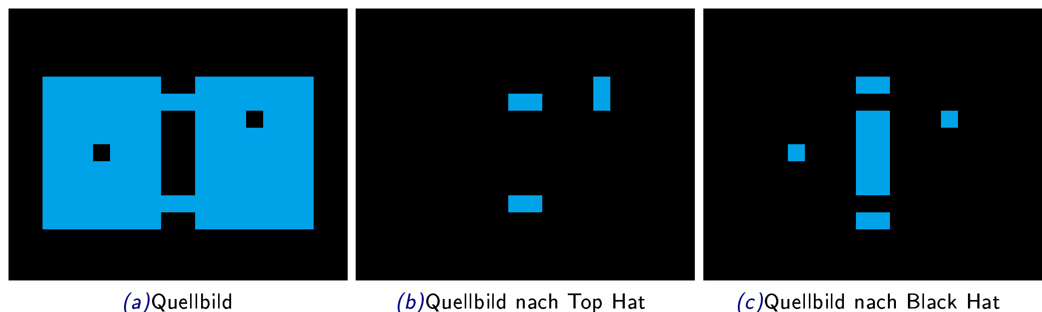
Top Hat und Black Hat isolieren Teilflächen die, im Gegensatz zu ihrer Umgebung, besonders hell (Top Hat) oder besonders dunkel (Black Hat) sind. Die Top Hat Operation auf einem Bild I wird dazu definiert als $TopHead(I) = src - Open(I)$. Top Head zeigt demnach die Pixel, die vom Opening verändert wurden. Dies sind genau die Verbindungen zwischen Flächen, die normalerweise durch das Opening reduziert würden.



(a)Quellbild (b)Quellbild nach Opening (c)Quellbild nach Closing

Abbildung 4.10: Quellbild mit 20x20 Pixeln und die Ergebnisse nach Opening und Closing mit einem quadratischen 3x3 Kernel

Im Gegensatz dazu wird die Black Hat Operation als $BlackHat(I) = Close(I) - src$ definiert. Ähnlich wie Top Hat zeigt Black Hat also die Flächen, die vom Closing verändert wurden. Black Hat zeigt also (dunkle) Löcher in hellen Flächen.



(a)Quellbild (b)Quellbild nach Top Hat (c)Quellbild nach Black Hat

Abbildung 4.11: Quellbild mit 20x20 Pixeln und die Ergebnisse nach Top Hat und Black Hat mit einem quadratischen 3x3 Kernel

Morphologischer Gradient

Eine weitere Kombination der Morphologischen Operationen ist der morphologische Gradient. Wie auch die anderen Gradienten verdeutlicht diese Operation die Kanten im Bild. Genauere Informationen zu Kanten und den anderen Gradienten finden sich unter 4.2.5 Gradienten. Der morphologische Gradient auf einem Quellbild I wird definiert als $Dilate(I) - Erode(I)$.

Morphologische Operationen in OpenCV

Für die grundlegenden Operationen Dilatation und Erosion stellt OpenCV die Funktionen *cvDelate* und *cvErode* zur Verfügung. Die zusammengesetzten morphologischen Operationen können über einen Parameter der Funktion *cvMorphologyEx* ausgeführt werden. Listing 4.8 zeigt den Aufruf der Funktionen.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U,
           img->nChannels);
7      int flags = 4 | (255 << 8);
```

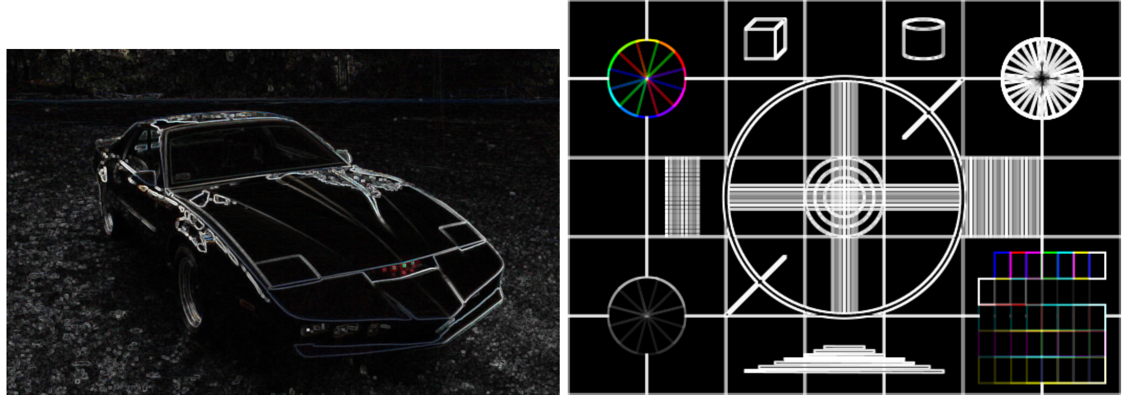


Abbildung 4.12: Anwendung des morphologischen Gradienten auf zwei Beispielbilder

```
9       cvDilate(img,dst);
10      cvSaveImage("morphology_delate.png", dst);

12      cvErode(img,dst);
13      cvSaveImage("morphology_erode.png", dst);

15      IplConvKernel* kernel = cvCreateStructuringElementEx(3,3,1,1,CV_SHAPE_RECT);

17      cvMorphologyEx(img,dst,NULL,kernel,CV_MOP_OPEN);
18      cvSaveImage("morphology_open.png", dst);

20      cvMorphologyEx(img,dst,NULL,kernel,CV_MOP_CLOSE);
21      cvSaveImage("morphology_close.png", dst);

23      cvMorphologyEx(img,dst,NULL,kernel,CV_MOP_GRADIENT);
24      cvSaveImage("morphology_grad.png", dst);

26      cvMorphologyEx(img,dst,NULL,kernel,CV_MOP_TOPHAT);
27      cvSaveImage("morphology_tophat.png", dst);

29      cvMorphologyEx(img,dst,NULL,kernel,CV_MOP_BLACKHAT);
30      cvSaveImage("morphology_blackhat.png", dst);

32      cvReleaseStructuringElement(&kernel);

34      cvReleaseImage(&img);
35      cvReleaseImage(&dst);
36  }
```

Listing 4.8: Führt die Morphologischen Operationen an einem Bild durch

Besonders zu beachten ist bei dem Aufruf von *cvMorphologyEx*, dass ein Kernel definiert werden muss, mit dem die Funktion arbeitet. Dieser Kernel wird mit der Funktion *cvCreateStructuringElementEx* erstellt und dann beim Aufruf an *cvMorphologyEx* übergeben. Eine Besonderheit von diesen selbst erstellten Kernels ist, dass sie nicht zwingend rechteckig sind und somit mehr Flexibilität erlauben. Im gezeigten Beispiel in Listing 4.8 wird dennoch ein quadratischer Kernel mit der Größe 3×3 erstellt. Die von OpenCV angebotenen Formen für Kernels sind:

CV_SHAPE_RECT für rechteckige Kernels

CV_SHAPE_CROSS für kreuzförmige Kernels

CV_SHAPE_ELLIPSE für elliptische Kernels

CV_SHAPE_CUSTOM für Kernels deren Form durch eine 2-Dimensionale *cvMat* Struktur beschrieben wird.

4.2.4 Faltung

Fast alle der bis hier vorgestellten Filter hatten eine Gemeinsamkeit — sie durchlaufen das Bild pixelweise und betrachten zu jedem Pixel im Ergebnisbild E einen Pixel im Urbild I mitsamt seiner Umgebung. Ist dieser Zusammenhang linear, so lässt er sich durch folgende Formel beschreiben:

$$E(x,y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j)G(i,j)$$

Dabei ist G eine $M_i \times M_j$ Matrix und (a_i, a_j) die Position des Urpixels (Ankers) in der Matrix. Je nach Belegung der Matrix entstehen dann unterschiedliche Effekte. E wird dann Faltung von I genannt.

$$G_{smooth} = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & (\frac{1}{9}) & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \qquad G_{sobel} = \begin{bmatrix} 1 & -2 & 1 \\ 2 & (-4) & 2 \\ 1 & -2 & 1 \end{bmatrix}$$

Abbildung 4.13: Verschiedene Faltungs-Matrizen, G_{smooth}: einfacher Weichzeichner, G_{sobel}: Sobel Gradient. Die Werte in Klammern stellen jeweils die Position des Ankers dar.

Abbildung 4.13 zeigt zwei wichtige Faltungs-Matrizen. Solche Matrizen stellen die Grundlage für eine Vielzahl von Algorithmen, insbesondere für die in Abschnitt 4.2.5 diskutierten Gradienten, dar.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U,
           img->nChannels);
7      CvMat* kernel = cvCreateMat(9,9, CV_32FC1);
8      cvNamedWindow("Convolution",CV_WINDOW_AUTOSIZE);

10     cvSet(kernel,cvScalar(1.0f/(9*9)));

12     cvFilter2D(img, dst, kernel, cvPoint(1,1));
13     cvShowImage("Convolution", dst);
14     cvWaitKey(1000);
15     cvSaveImage("convolution.png", dst);

17     cvReleaseImage(&img);
18     cvReleaseImage(&dst);
19     cvDestroyWindow("Convolution");
20 }
```

Listing 4.9: Erstellt eine Faltung, die sich verhält wie der einfache Weichzeichner aus 4.2.1 Einfaches Weichzeichnen

Listing 4.9 zeigt die Verwendung der Faltung in OpenCV. Zunächst wird eine Matrix erstellt, um die Informationen des Kernels darzustellen. Dieser wird dann mit der Funktion *cvFilter2D* auf den Bilddaten angewendet. Das Ergebnis ist identisch mit dem aus 4.2.1 Einfaches Weichzeichnen*.

*Bei genauem Hinsehen fällt auf, dass die Ränder der Bilder unterschiedlich sind. Dies liegt am Verhalten der Faltung auf Pixeln, die außerhalb des Bildes liegen. Für genauere Informationen sei auf [7, S. 146] verwiesen.

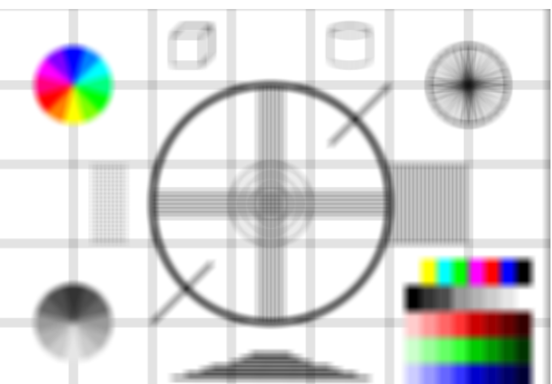


Abbildung 4.14: Ergebnis der in Listing 4.9 gezeigten Faltung.

4.2.5 Gradienten

Eine prominente Anwendung der Faltung aus Abschnitt 4.2.4 sind die Gradienten. Gradienten von Bildern sind Näherungen der Ableitungen und können zur Erkennung von Kanten in Bilddaten eingesetzt werden, siehe dazu auch 4.2.3 Morphologischer Gradient.

Die Sobel Ableitung

Die grundlegende Operation ist die Sobel Ableitung. Mit dieser Operation können aus Bilddaten Ableitungen in x- und y-Richtung erstellt werden. OpenCV bietet hierfür die Funktion *cvSobel*.

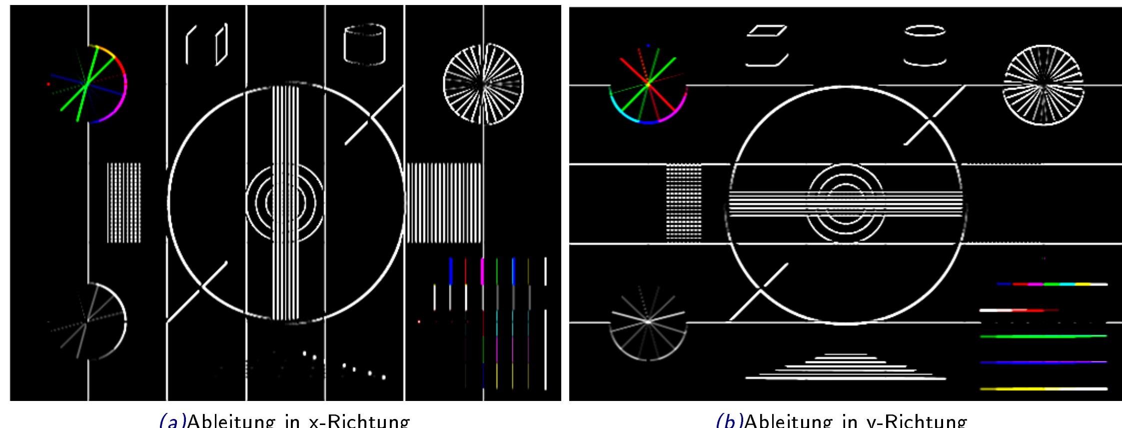
```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_16S,
           img->nChannels);
7      cvNamedWindow("Sobel",CV_WINDOW_AUTOSIZE);

9      cvSobel(img, dst, 1, 0);
10     cvShowImage("Sobel", dst);
11     cvWaitKey(1000);
12     cvSaveImage("sobel_x.png", dst);

14     cvSobel(img, dst, 0, 1);
15     cvShowImage("Sobel", dst);
16     cvWaitKey(1000);
17     cvSaveImage("sobel_y.png", dst);

19     cvReleaseImage(&img);
20     cvReleaseImage(&dst);
21     cvDestroyWindow("Sobel");
22 }
```

Listing 4.10: Erstellt die erste Ableitung eines Quellbildes in x- und y-Richtung

Über die Parameter *xorder* und *yorder* kann die Ordnung der Ableitung bestimmt werden. Zu beachten ist, dass Sobel nur die Ableitung in eine der beiden Richtungen erstellen kann, sodass mindestens einer der beiden Parameter null sein muss.

Der Laplace Operator

Eine Anwendung der Sobel Ableitung ist der Laplace Operator. Für den Laplace Operator gibt es die Grundannahme, dass eine Kante genau dort ist, wo der Unterschied zwischen benachbarten



(a) Ableitung in x-Richtung

(b) Ableitung in y-Richtung

Abbildung 4.15: Ergebnisse des in 4.10 gezeigten Quellcodes.

Pixeln maximal ist. Der Laplace Operator auf einem Quellbild I ist demnach definiert als:

$$Laplace(I) = \frac{\partial I^2}{\partial x^2} + \frac{\partial I^2}{\partial y^2} \equiv Sobel(I,2,0) + Sobel(I,0,2)$$

OpenCV setzt den Laplace Operator wie definiert um, sodass der Aufruf dem des Sobel Operators sehr ähnelt. Listing 4.11 zeigt die Anwendung des Laplace Operators.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_16S,
           img->nChannels);
7      cvNamedWindow("Laplace",CV_WINDOW_AUTOSIZE);

9      cvLaplace(img, dst);
10     cvShowImage("Laplace", dst);
11     cvWaitKey(1000);
12     cvSaveImage("laplace.png", dst);

14     cvReleaseImage(&img);
15     cvReleaseImage(&dst);
16     cvDestroyWindow("Sobel");
17 }
```

Listing 4.11: Anwendung des Laplace Operators auf ein Quellbild

Der Laplace Operator eignet sich sehr gut zur Kantenerkennung. In den Ergebnisbildern in Abbildung 4.16 sieht man eine starke Ähnlichkeit zu den Ergebnissen des morphologischen Gradienten in Abbildung 4.12.

Von Kanten zu Konturen

Neben Kanten spielen Konturen eine wichtige Rolle in Bildverarbeitungsalgorithmen. Konturen haben dabei gegenüber kanten eine entscheidende Eigenschaft: sie sind binär. Ein Pixel gehört entweder zu einer Kontur oder nicht. OpenCV berechnet Konturen mit dem Algorithmus nach Canny. Dieser ordnet die Pixel auf den gefundenen Kanten einer Kontur zu, wenn sie zwischen bestimmten Grenzwerten liegen.

```
1  #include <highgui.h>
2  #include <cv.h>
```

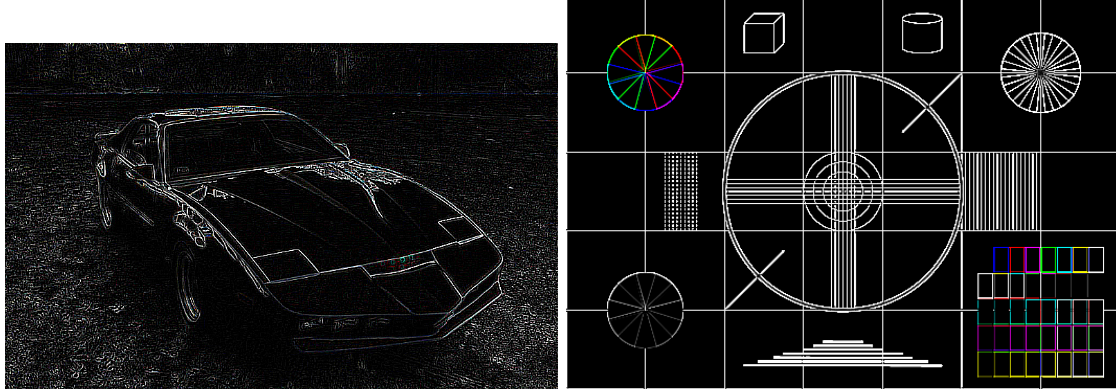


Abbildung 4.16: Zwei Ergebnisse des in 4.11 gezeigten Quellcodes.

```
4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* tmp = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
7      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
8      cvNamedWindow("Canny",CV_WINDOW_AUTOSIZE);

10     cvCvtColor(img, tmp, CV_BGR2GRAY);

12     cvCanny(tmp, dst,150,100);
13     cvShowImage("Canny", dst);
14     cvWaitKey(1000);
15     cvSaveImage("canny.png", dst);

17     cvReleaseImage(&img);
18     cvReleaseImage(&dst);
19     cvReleaseImage(&tmp);
20     cvDestroyWindow("Canny");
21 }
```

Listing 4.12: Anwendung des Canny Operators auf ein Quellbild

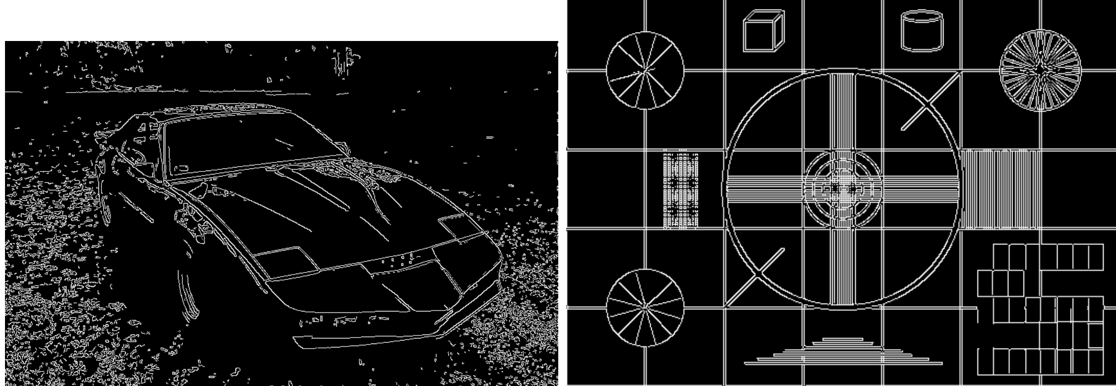


Abbildung 4.17: Zwei Ergebnisse des in 4.12 gezeigten Quellcodes.

Bei der Verwendung von *cvCanny* ist zu beachten, dass die Funktion nur grauwertige Eingabebilder akzeptiert. Das Verhältnis zwischen dem unteren und dem oberen Schwellwert soll nach [7, S. 152] zwischen 2:1 und 3:1 liegen, andere Werte sind aber auch möglich.

4.2.6 Flächen füllen

Diese Methode der Bildtransformation ist einer Funktion von üblichen Computer Grafikprogrammen nachempfunden*. Nach der Auswahl eines Startpixels werden alle angrenzenden Pixel, die eine ähnliche Farbe haben wie der Startpixel, auf eine angegebene Farbe gesetzt.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* mask = cvCreateImage(cvSize(img->width+2, img->height+2), IPL_DEPTH_8U, 1);
7      cvZero(mask);
8      cvNamedWindow("Flood",CV_WINDOW_AUTOSIZE);
9      int flags = 4 | (255 << 8);

11     cvFloodFill(img, cvPoint(450,250), CV_RGB(255,0,0), cvScalarAll(7), cvScalarAll(7), NULL,
           flags,mask);
12     cvShowImage("Flood",img);
13     cvWaitKey(1000);
14     cvSaveImage("floodfill_image.png", img);
15     cvShowImage("Flood",mask);
16     cvWaitKey(1000);
17     cvSaveImage("floodfill_mask.png", mask);

19     cvReleaseImage(&img);
20     cvReleaseImage(&mask);
21     cvDestroyWindow("Flood");
22 }
```

Listing 4.13: Quellcode, der eine Fläche rot füllt, die von Pixel (450,250) ausgeht

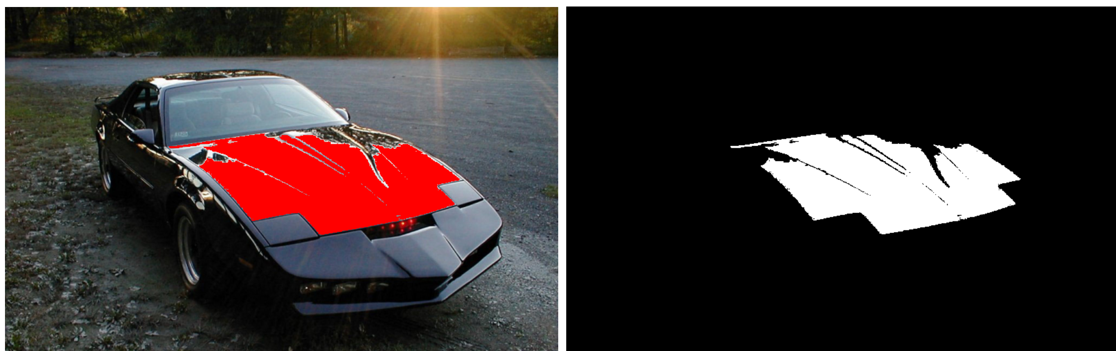


Abbildung 4.18: Ergebnisse des in 4.13 gezeigten Quellcodes.

Der Quellcode aus 4.13 erzeugt zwei Bilder. Einmal wird die Füllung auf das Eingabebild angewendet und zusätzlich wird die Füllung im Bild *mask* aufgezeichnet. Die so erstellte Maske kann verwendet werden, um bestimmte Regionen im Bild zu isolieren und die Bearbeitung auf diese Bildregionen zu reduzieren. Zu beachten ist an dieser Stelle, dass die Maske zwei Pixel breiter und zwei Pixel höher als das Quellbild sein muss.

Optionen von cvFloodFill

cvFloodFill bietet zur Veränderung des Verhaltens lediglich einen *flags* Parameter. Die Funktion kennt für diesen Parameter folgende Werte:

4 *cvFloodFill* sucht in einer kreuzförmigen Umgebung nach Pixeln mit ähnlicher Farbe.

8 *cvFloodFill* sucht in einer quadratischen Umgebung nach Pixeln mit ähnlicher Farbe.

*Diese setzten diese Funktion meist durch ein Farbeimer-Werkzeug um.

CV_FLOODFILL_FIXED_RANGE Die Farbe der Pixel wird nur mit der Farbe des Startpixels verglichen. Ist diese Option nicht gesetzt, werden die Pixel in der Umgebung miteinander verglichen.

CV_FLOODFILL_MASK_ONLY Die Funktion füllt nur die Maske und nicht das Bild.

Die Optionen können mit einem bitweisen Oder verknüpft werden. Zudem kann mit dem *flags* Parameter auch der Farbwert bestimmt werden, mit dem die Maske gefüllt wird. Dieser Farbwert darf acht Bit groß sein und muss um acht Bit verschoben werden, sodass die oben beschriebenen Optionswerte 4 bzw. 8 nicht überschrieben werden.

4.3 Bildtransformationen

Neben Bildmanipulationen bzw. Filtern, die das Bild lokal verarbeiten und Merkmale verbessern, gibt es noch eine weitere Klasse von Operationen, die auf Bilddaten ausgeführt werden kann. Die Bildtransformationen werden global auf das ganze Bild angewendet und ändern die Eigenschaften des gesamten Bildes wie Farbraum und Größe.

4.3.1 Farbräume

Wie in 3.2.1 Darstellung von (Farb-)Bilddaten beschrieben, gibt es verschiedene Arten die Informationen in einem Bild darzustellen. Die verschiedenen Verfahren haben Vor- und Nachteile, die es nötig machen, dass ein Bild von der einen Darstellungsform in eine andere konvertiert wird.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* hsv = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
8      cvNamedWindow("RGB to HSV",CV_WINDOW_AUTOSIZE);

10     cvCvtColor(img, hsv, CV_BGR2HSV);
11     cvAddS(hsv,cvScalar(50,0,0),hsv);
12     cvCvtColor(hsv, dst, CV_HSV2BGR);

14     cvShowImage("RGB to HSV", dst);
15     cvWaitKey(1000);
16     cvSaveImage("cvtcolor.png", dst);

18     cvReleaseImage(&img);
19     cvReleaseImage(&hsv);
20     cvReleaseImage(&dst);
21     cvDestroyWindow("RGB to HSV");
22 }
```

Listing 4.14: Ändert den Farbraum eines Quellbildes, addiert dann 50 zum Farbwert und konvertiert dann das Bild zur Ausgabe wieder nach RGB

OpenCV unterstützt diverse Konvertierungen. Eine genaue Liste der Formate findet sich in [7, S. 58f] oder [1, S. 265ff]

4.3.2 Resize

Häufig liegen Bilddaten in einer Größe (Auflösung) vor, die für die benötigte Anwendung nicht geeignet ist. Man muss das Bild dann vergrößern oder verkleinern. OpenCV bietet dafür die Funktion *cvResize*. Diese Funktion ermöglicht es Bilddaten aus einem Bild an die Auflösung eines anderen Bildes anzupassen. Da es dabei häufig keine 1:1 Übereinstimmung zwischen Pixeln im Urbild und dem erzeugten Bild gibt, müssen die Pixel interpoliert werden. OpenCV unterstützt vier verschiedene Interpolationsoperationen:

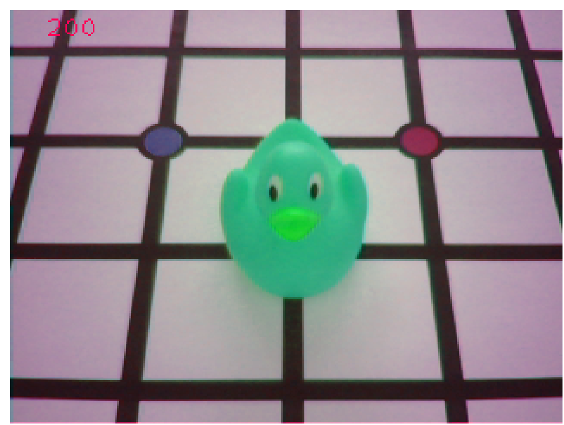


Abbildung 4.19: Ergebnis des in Listing 4.14 gezeigten Programms.

CV_INTER_NN Nächster Nachbar — OpenCV nutzt den Wert des nächstgelegen Pixels

CV_INTER_LINEAR (Bi-)Linear — Zweidimensionale, lineare Interpolation der nahe gelegenen Pixel (2x2-Umgebung)

CV_INTER_AREA Flächenbasiert — Zweidimensionale, lineare Interpolation, die die Lage des Pixels innerhalb der Umgebung beachtet

CV_INTER_CUBIC (Bi-)Kubisch — Interpolation über zweidimensionale, kubische Polynome (4x4-Umgebung)

4.3.3 Bildpyramiden

Bildpyramiden sind eine weit verbreitete Technik in Bildverarbeitungsalgorithmen. Eine Bildpyramide ist eine Sammlung von Bildern, die alle aus einem Originalbild berechnet werden. Im Wesentlichen unterscheidet man zwischen zwei verschiedenen Typen von Bildpyramiden: Gaußsche (G) und Laplacesche (L). Die Pyramiden sind in Ebenen unterteilt, diese Ebenen werden mit einem Index versehen, sodass G_0 das Urbild darstellt.

Zur Erstellung der Gaußschen Pyramide bietet OpenCV die Funktion *cvPyrDown*. Diese Funktion führt zunächst einen Gaußschen Weichzeichner auf dem Bild durch und reduziert das Bild dann um jede zweite Zeile und jede zweite Spalte — das Bild hat nach der Operation also nur noch ein Viertel der Fläche. Die Ebenen der Gaußschen Pyramide sind definiert als:

$$G_{i+1} = PyrDown(G_i)$$

Eine ähnliche Operation stellt *cvPyrUp* dar. *cvPyrUp* vervierfacht die Größe des Bildes — fügt also Zeilen und Spalten ein. Die neu eingefügten Zeilen und Spalten werden abermals über einen Gaußschen Weichzeichner bestimmt. Offensichtlich ist *cvPyrUp* dennoch nicht die inverse Funktion zu *cvPyrDown*. Die Laplacesche Pyramide definiert sich genau über diesen Unterschied:

$$L_i = G_i - PyrUp(G_{i+1})$$

Es gibt mehrere Algorithmen, die Nutzen aus den beiden Pyramiden ziehen. Insbesondere die Pyramidensegmentierung, die in 4.7.2 Pyramidensegmentierung beschrieben wird.

4.3.4 Strecken, verzerren und rotieren

OpenCV bietet eine Reihe von Bildtransformationen, die sich auf die Geometrie des Bildes beziehen. Diese Bildtransformationen werden in zwei Kategorien unterteilt: affine Transformationen und perspektivische Transformationen.

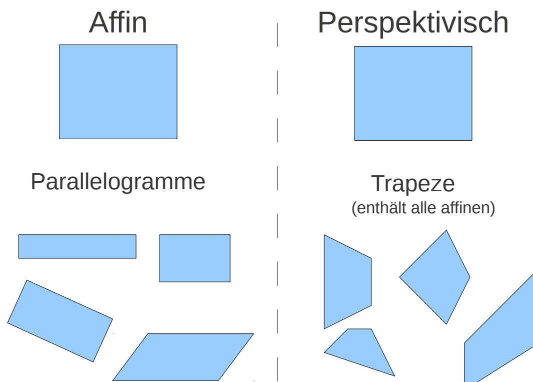


Abbildung 4.20: Affine und perspektivische Transformationen

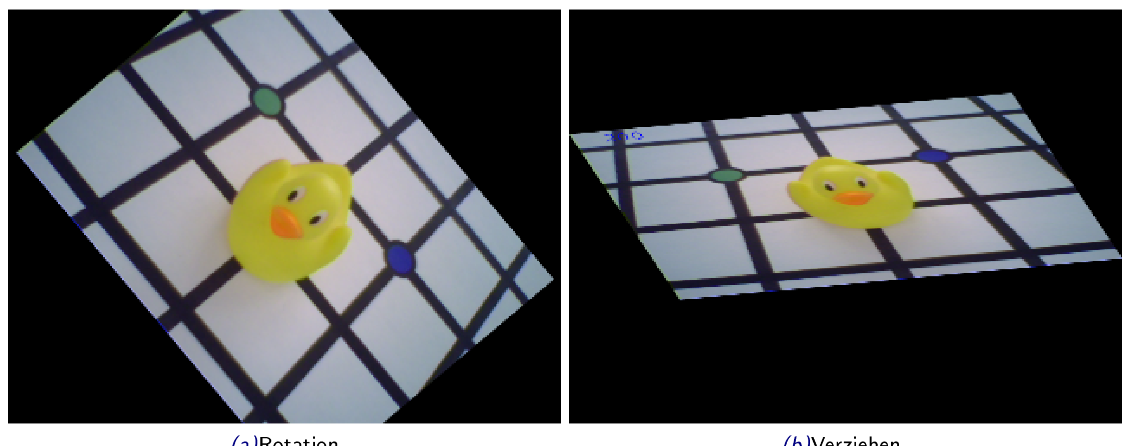
Affine Transformationen

Affine Transformationen erstellen aus dem Bild immer ein Parallelogramm, sodass folgende Effekte erreicht werden können: Rotation, Vergrößern, Verkleinern und paralleles Verziehen der Gegenüberliegenden Seiten des Bildes.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7      cvNamedWindow("Warp",CV_WINDOW_AUTOSIZE);

9      CvMat* rotation = cvCreateMat(2,3,CV_32FC1);
10     CvMat* warp = cvCreateMat(2,3,CV_32FC1);

12     CvPoint2D32f center = cvPoint2D32f(img->width/2, img->height/2);
13     double angle = -50.0;
14     double scale = 0.8;
15     cv2DRotationMatrix(center, angle, scale, rotation);
16     cvWarpAffine(img, dst, rotation);

18     cvShowImage("Warp", dst);
19     cvWaitKey(1000);
20     cvSaveImage("warpaffine_rotate.png", dst);

22     CvPoint2D32f srcPts[3], dstPts[3];
23     srcPts[0].x = 0;             srcPts[0].y = 0;
24     srcPts[1].x = img->width;    srcPts[1].y = 0;
25     srcPts[2].x = 0;             srcPts[2].y = img->height;
26     dstPts[0].x = img->width*0.0; dstPts[0].y = img->height*0.3;
27     dstPts[1].x = img->width*0.8; dstPts[1].y = img->height*0.2;
28     dstPts[2].x = img->width*0.2; dstPts[2].y = img->height*0.7;
29     cvGetAffineTransform(srcPts, dstPts, warp);
30     cvWarpAffine(img, dst, warp);

32     cvShowImage("Warp", dst);
33     cvWaitKey(1000);
34     cvSaveImage("warpaffine_warp.png", dst);

36     cvReleaseImage(&img);
37     cvReleaseImage(&dst);
38     cvReleaseMat(&rotate);
39     cvReleaseMat(&warp);
40     cvDestroyWindow("Warp");
41 }
```

Listing 4.15: Führt zwei affine Transformationen durch: Rotation und Verziehen

Für die Transformation muss im Programm wie in Listing 4.15 zunächst eine Transformationsmatrix aufgestellt werden. Für die affinen Transformationen ist dies eine 2×3 Matrix. OpenCV



*(a)*Rotation *(b)*Verziehen

Abbildung 4.21: Ergebnisse des in 4.15 gezeigten Quellcodes.

bietet zwei Möglichkeiten diese aufzustellen. Zum Einen wird durch die Funktion *cvRotationMatrix* eine Transformationsmatrix, die eine Rotation und Skalierung der Bilddaten vornehmen kann, initialisiert. Die andere Funktion, *cvGetAffineTransform*, bietet zudem die Möglichkeit das Bild zu verzerren. Aus drei Punkten im Urbild zusammen mit ihrer Abbildung berechnet *cvGetAffineTransform* die Transformationsmatrix. Die Transformation wird dann mit der Funktion *cvWarpAffine* durchgeführt.

Perspektivische Transformationen

Eine höhere Flexibilität als mit der affinen Transformation erreicht man mit der perspektivischen Transformation. Die Benutzung der perspektivischen Transformation in OpenCV ist dennoch sehr ähnlich zu den affinen Transformationen.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7      cvNamedWindow("Warp",CV_WINDOW_AUTOSIZE);

9      CvMat* warp = cvCreateMat(3,3,CV_32FC1);

11     CvPoint2D32f srcPts[4], dstPts[4];
12     srcPts[0].x = 0;               srcPts[0].y = 0;
13     srcPts[1].x = img->width;      srcPts[1].y = 0;
14     srcPts[2].x = 0;               srcPts[2].y = img->height;
15     srcPts[3].x = img->width;      srcPts[3].y = img->height;
16     dstPts[0].x = img->width*0.1;  dstPts[0].y = img->height*0.3;
17     dstPts[1].x = img->width*0.9;  dstPts[1].y = img->height*0.2;
18     dstPts[2].x = img->width*0.2;  dstPts[2].y = img->height*0.7;
19     dstPts[3].x = img->width*0.8;  dstPts[3].y = img->height*0.9;

21     cvGetPerspectiveTransform(srcPts, dstPts, warp);
22     cvWarpPerspective(img, dst, warp);

24     cvShowImage("Warp", dst);
25     cvWaitKey(1000);
26     cvSaveImage("warpperspective.png", dst);

28     cvReleaseImage(&img);
29     cvReleaseImage(&dst);
30     cvDestroyWindow("Warp");
31 }
```

Listing 4.16: Führt zwei affine Transformationen durch: Rotation und Verziehen

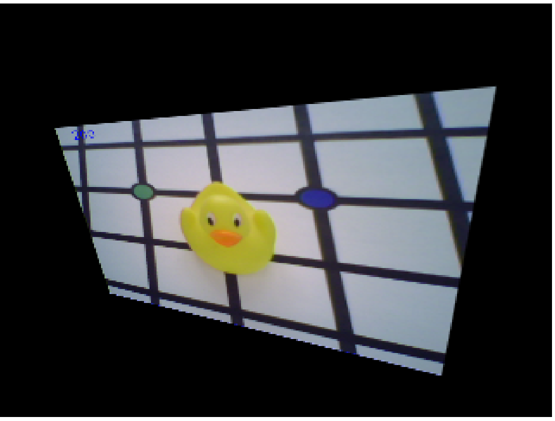


Abbildung 4.22: Ergebnis des in 4.16 gezeigten Quellcodes.

Wie die affine Transformation benötigt die perspektivische Transformation eine Transformatoinsmatrix. Diese wird mit der Funktion *cvGetPerspectiveTransform* erzeugt. Im Gegensatz zur affinen Transformation benötigt die perspektivische Transformation vier Punkte aus dem Urbild und die entsprechenden Abbildungen, um die Transformationsmatrix zu berechnen. Die Transformation wird dann mittels *cvWarpPerspective* durchgeführt.

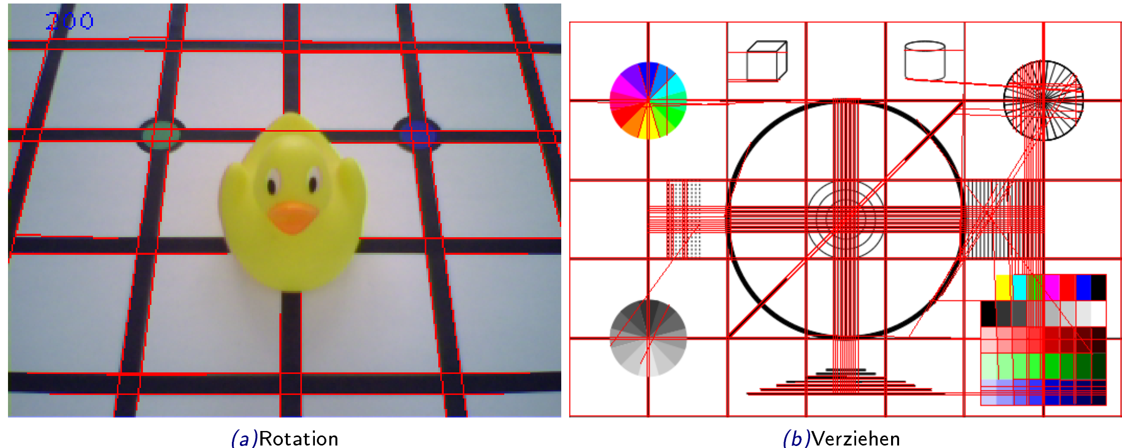
4.4 Hough Transformationen

Die sogenannten Hough Transformationen suchen nach geometrischen Objeken, genauer Kreisen oder Linien, in Bilddaten. OpenCV bietet genau für diesen Zweck die Funktionen *cvHoughLines2* und *cvHoughCircles*.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
6      IplImage* dst = cvLoadImage(argv[1]);
7      cvNamedWindow("Hough",CV_WINDOW_AUTOSIZE);

9      CvMemStorage* storage = cvCreateMemStorage(0);

11     cvCanny(img,img,50,200);

13     CvSeq* results = cvHoughLines2(img, storage, CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 40, 40,
           40);

15     for(int i=0; i<results->total; i++){
16         CvPoint* pt = (CvPoint*) cvGetSeqElem(results, i);
17         cvLine(dst, pt[0], pt[1], CV_RGB(255,0,0),1);
18     }

20     cvShowImage("Hough", dst);
21     cvWaitKey(1000);
22     cvSaveImage("hough_lines.png", dst);

24     cvReleaseMemStorage(&storage);
25     cvReleaseImage(&img);
26     cvReleaseImage(&dst);
27     cvDestroyWindow("Hough");
28 }
```

Listing 4.17: Liniensuche auf Bilddaten und Zeichnen der Linien in das Bild

Die Anwendung beider Funktionen ist sehr ähnlich, sodass diese nur anhand von *cvHoughLines2* gezeigt wird. Zunächst muss ein Speicherbereich für die gefundenen Linien angelegt werden, die als *CvSeq* zurückgegeben werden, siehe dazu auch 3.3 CvSeq. Die zurückgegebene Sequenz besteht



(a)Rotation (b)Verziehen

Abbildung 4.23: Ergebnisse des in 4.17 gezeigten Quellcodes. Die gefundenen Linien wurden rot markiert.

aus *cvPoint* Strukturen, die nacheinander durchlaufen und dann weiter verarbeitet werden können. Bevor *cvHoughLines2* auf die Bilddaten angewendet wird, wird zunächst eine Kantenerkennung mittels *cvCanny* durchgeführt. Dies verbessert die Ergebnisse der Linienerkennung enorm.

4.5 Histogramme

Histogramme sind eine Darstellung für die Häufigkeitsverteilung bestimmter Merkmale in einer Menge von Daten. In der Bildverarbeitung werden Histogramme genutzt, um die Häufigkeitsverteilung der Farbwerte in einem Bild zu repräsentieren. Histogramme können dabei ein-, zwei- und auch dreidimensional auftreten, in OpenCV auch für beliebige Dimensionen. Die häufigsten Histogrammtypen sind eindimensionale für einen Farbkanal im RGB Farbraum (Rot, Grün oder Blau) oder zweidimensionale für die Kanäle Farbwert und Sättigung des HSL oder HSV Farbraums.

4.5.1 Histogrammausgleich

Bei der Aufnahme von Bilddaten kommt es häufig zu Problemen, wie Über- oder Unterbelichtung. Aus diesen Problemen resultiert, dass nur ein geringer Bereich der möglichen Farbwerte eines Farbraumes genutzt wird. Der Kontrast des Bildes und somit der Unterschied zwischen Kanten ist dann schlechter, als er sein könnte. Durch das Ausgleichen des Histogramms, das Aufspreizen der vorhandenen Werte auf den kompletten Farbraum, kann der Kontrast des Bildes erhöht werden.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1]);
6      IplImage* red = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
7      IplImage* green = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
8      IplImage* blue = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
9      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
10     cvNamedWindow("Histogram",CV_WINDOW_AUTOSIZE);

12     cvSplit(img,blue,green,red,NULL);

14     cvEqualizeHist(blue,blue);
15     cvEqualizeHist(green,green);
16     cvEqualizeHist(red,red);

18     cvMerge(blue,green,red,NULL,dst);
```

```
20      cvShowImage("Histogram", dst);
21      cvWaitKey(1000);
22      cvSaveImage("histogram_equal.png", dst);

24      cvReleaseImage(&img);
25      cvReleaseImage(&red);
26      cvReleaseImage(&green);
27      cvReleaseImage(&blue);
28      cvReleaseImage(&dst);
29      cvDestroyWindow("Histogram");
30  }
```

Listing 4.18: Führt einen Histogrammausgleich auf einem Farbbild durch. Die Farbkanäle müssen dafür einzeln bearbeitet werden.

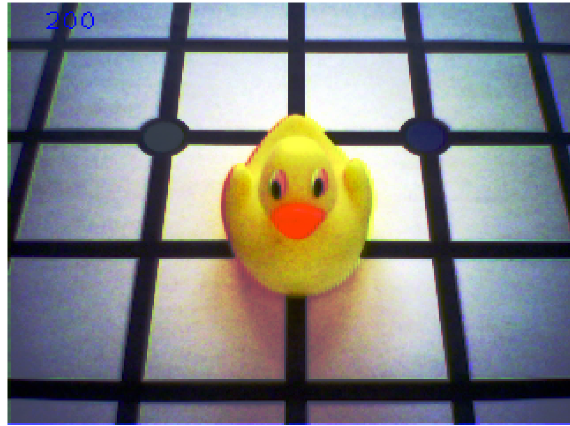


Abbildung 4.24: Ergebnis des Histogramm Ausgleichs anhand von einem Kamerabild mit relativ schlechtem Kontrast.

Listing 4.18 zeigt die Verwendung des Histogrammausgleichs in OpenCV. Dafür wird die Funktion *cvEqualizeHist* genutzt. Da OpenCV diesen automatisierten Ausgleich nur auf Grauwertbildern durchführen kann, wird das Eingabebild mit der Funktion *cvSplit* in die drei Kanäle aufgeteilt und nach der Verarbeitung wieder mittels *cvMerge* zu einem Gesamtbild zusammengefügt.

Dieses Vorgehen ist zwar sehr einfach, birgt jedoch die Gefahr, dass es bei sehr ungleichmäßiger Abdeckung der Kanäle des Farbraums zu Farbverschiebungen kommt — i.d.R. führt ein Histogrammausgleich im HSV Farbraum auf Sättigung und Wert bzw. im HSL Farbraum auf Sättigung und Helligkeit zu besseren Ergebnissen.

4.5.2 Berechnen von Histogrammen

Neben dem automatisierten Ausgleich des Histogramms kann OpenCV auch Histogramme von Bildern berechnen. Für das Speichern der Histogramme nutzt OpenCV die Struktur *CvHistogram*.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* img = cvLoadImage(argv[1]);
6       cvCvtColor(img, img, CV_BGR2HSV);

8       IplImage* h_plane = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
9       IplImage* s_plane = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
10      IplImage* v_plane = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
11      IplImage* planes[] = {h_plane, s_plane};
12      cvSplit(img,h_plane,s_plane,v_plane,NULL);

14      CvHistogram* hist;
15      int sizes[] = {15,16};
16      float h_range[] = {0, 180};
```

```
17      float s_range[] = {0, 255};
18      float* ranges[] = {h_range, s_range};
19      hist = cvCreateHist(2, sizes, CV_HIST_ARRAY, ranges, 1);

21      cvCalcHist(planes, hist, 0, NULL);

23      int scale = 10;
24      IplImage* dst = cvCreateImage(cvSize(scale*sizes[0], scale*sizes[1]), IPL_DEPTH_8U, 3);
25      cvZero(dst);

27      float maxVal = .0f;
28      cvGetMinMaxHistValue(hist, NULL, &maxVal, NULL, NULL);

30      for(int i=0; i<sizes[0]; i++){
31          for(int j=0; j<sizes[1]; j++){
32              float value = cvQueryHistValue_2D(hist, i, j);
33              int intensity = cvRound(value * 255 / maxVal);
34              cvRectangle(dst,
35                          cvPoint(i*scale, j*scale),
36                          cvPoint((i+1)*scale, (j+1)*scale),
37                          cvScalar(intensity, intensity, intensity),
38                          CV_FILLED);
39          }
40      }

42      cvNamedWindow("Histogram",CV_WINDOW_AUTOSIZE);
43      cvShowImage("Histogram", dst);
44      cvWaitKey(1000);
45      cvSaveImage("histogram_calc.png", dst);

47      cvReleaseImage(&img);
48      cvReleaseImage(&h_plane);
49      cvReleaseImage(&s_plane);
50      cvReleaseImage(&v_plane);
51      cvReleaseImage(&dst);
52      cvReleaseHist(&hist);
53      cvDestroyWindow("Histogram");
54  }
```

Listing 4.19: Berechnung und Zeichnen eines zweidimensionalen H-S Histogramms eines Eingabebildes.

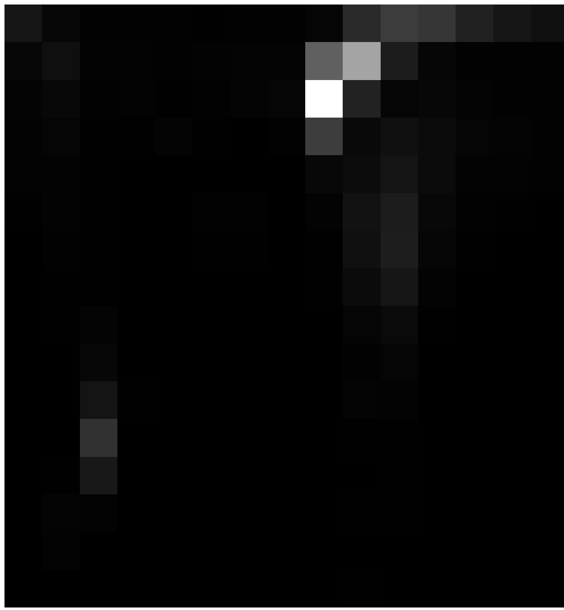


Abbildung 4.25: Gezeichnetes H-S Histogramm des in Abbildung 4.1b gezeigten Bildes. Der Farbwert ist auf der x- und die Sättigung auf der y-Achse. Die Helligkeit gibt die Anzahl des Auftretens an.

Für die Berechnung des Histogramms wird zunächst eine Datenstruktur vom Typ *CvHistoramm* initialisiert. Dies geschieht mit der Funktion *cvCreateHist* aus der Dimension und der Granularität

des Histogramms. Das Histogramm wird dann durch die Funktion *cvCalcHist* mit den Daten aus den Kanälen gefüllt. Die weiteren Befehle in Listing 4.19 dienen dem Zeichnen des Histogramms.

Dafür wird das Histogramm mit zwei Schleifen durchlaufen und feldweise ausgelesen. Die ausgelesenen Werte werden dann als Grauwert in ein Bild gezeichnet, das dann ausgegeben wird.

4.5.3 Vergleichen von Histogrammen

Eine Möglichkeit Objekte in einem Bild zu erkennen ist ihre Histogramme miteinander zu vergleichen. Im Gegensatz zu anderen Verfahren, wie Vorlagenvergleich, ist der Vergleich von Histogrammen kaum anfällig für Rotation des Objekts. Für den Vergleich von Histogrammen bietet OpenCV die Funktion *cvCompareHist* . Genau genommen bietet *cvCompareHist* sogar vier unterschiedliche Vergleichsmethoden. Als Grundlage für den Vergleich dienen stochastische Methoden, die die Histogramme als diskrete Stichproben betrachten. Für solche Stichproben lassen sich dann Wahrscheinlichkeiten berechnen, mit der zwei Histogramme oder ein Histogramm und ein einzelner Wert aus der selben Grundmenge stammen.

```
1  #include <highgui.h>
2  #include <cv.h>
3  #include <stdio.h>

5  int main(int argc, char** argv){
6      IplImage* img1 = cvLoadImage(argv[1]);
7      cvCvtColor(img1, img1, CV_BGR2HSV);

9      IplImage* img2 = cvLoadImage(argv[2]);
10     cvCvtColor(img2, img2, CV_BGR2HSV);

12     int sizes[] = {15,16};
13     float h_range[] = {0, 180};
14     float s_range[] = {0, 255};
15     float* ranges[] = {h_range, s_range};

17     CvHistogram* hist_img1;
18     CvHistogram* hist_img2;
19     hist_img1 = cvCreateHist(2, sizes, CV_HIST_ARRAY, ranges, 1);
20     hist_img2 = cvCreateHist(2, sizes, CV_HIST_ARRAY, ranges, 1);

22     IplImage* h_plane = cvCreateImage(cvSize(img1->width, img1->height), IPL_DEPTH_8U, 1);
23     IplImage* s_plane = cvCreateImage(cvSize(img1->width, img1->height), IPL_DEPTH_8U, 1);
24     IplImage* v_plane = cvCreateImage(cvSize(img1->width, img1->height), IPL_DEPTH_8U, 1);
25     IplImage* planes[] = {h_plane, s_plane};
26     cvSplit(img1,h_plane,s_plane,v_plane,NULL);

28     cvCalcHist(planes, hist_img1, 0, NULL);

30     cvReleaseImage(&h_plane);
31     cvReleaseImage(&s_plane);
32     cvReleaseImage(&v_plane);

34     h_plane = cvCreateImage(cvSize(img2->width, img2->height), IPL_DEPTH_8U, 1);
35     s_plane = cvCreateImage(cvSize(img2->width, img2->height), IPL_DEPTH_8U, 1);
36     v_plane = cvCreateImage(cvSize(img2->width, img2->height), IPL_DEPTH_8U, 1);
37     planes[0] = h_plane; planes[1] = s_plane;
38     cvSplit(img2,h_plane,s_plane,v_plane,NULL);

40     cvCalcHist(planes, hist_img2, 0, NULL);

42     cvNormalizeHist(hist_img1, 1.0);
43     cvNormalizeHist(hist_img2, 1.0);

45     double correl = cvCompareHist(hist_img1, hist_img2, CV_COMP_CORREL);
46     double chisqr = cvCompareHist(hist_img1, hist_img2, CV_COMP_CHISQR);
47     double intersec = cvCompareHist(hist_img1, hist_img2, CV_COMP_INTERSECT);
48     double bhatta = cvCompareHist(hist_img1, hist_img2, CV_COMP_BHATTACHARYYA);

50     printf("Correl: %f \t ChiSqr: %f \t Intersec: %f \t Bhattacharyya: %f \n",correl, chisqr,
           intersec, bhatta);

52     cvReleaseImage(&img1);
53     cvReleaseImage(&img2);
54     cvReleaseImage(&h_plane);
55     cvReleaseImage(&s_plane);
```

```
56      cvReleaseImage(&v_plane);
57      cvReleaseHist(&hist_img1);
58      cvReleaseHist(&hist_img2);
59  }
```

Listing 4.20: Berechnung von H-S Histogrammen von zwei Bildern und anschließender Vergleich mit unterschiedlichen Methoden.

Listing 4.20 zeigt die Verwendung von verschiedenen Verfahren für den Histogrammvergleich, die durch den letzten Parameter der Funktion *cvCompareHist* bestimmt werden können. Anhand der Ausgabe des Programms lässt sich leicht erkennen, dass die verschiedenen Verfahren unterschiedliche Maßzahlen für den Vergleich verwenden. Eine Erläuterung der für den Vergleich verwendeten Verfahren, die *cvCompareHist* nutzt, und die resultierenden Wertebereiche für sehr ähnliche oder sehr unterschiedliche Histogramme können [1, S.286] entnommen werden.

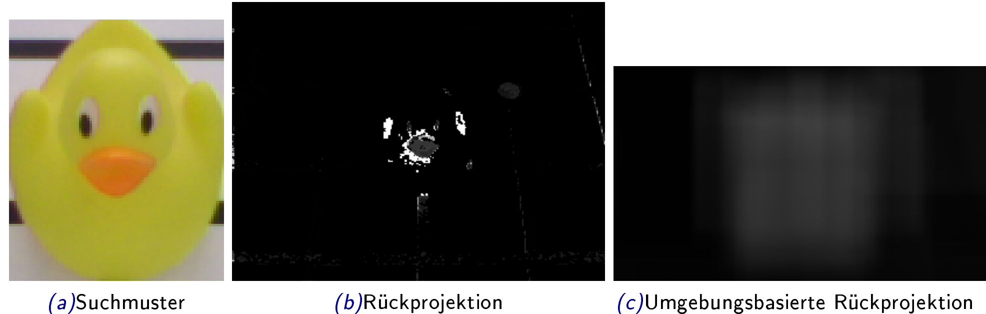
Rückprojektion

Sollen nicht nur zwei Histogramme miteinander verglichen werden, sondern die Lage eines Objekts mit einem bestimmten Histogramm in einem Bild gefunden werden, ist die Funktion *cvCompareHist* weniger von Nutzen. OpenCV liefert für diesen Anwendungsfall zwei Methoden die lokal, ähnlich wie ein Filter, für jeden Pixel des Bildes (und seine Umgebung) einen solchen Histogrammvergleich durchführen: *cvCalcBackProject* und *cvCalcBackProjectPatch*.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* needle = cvLoadImage(argv[1]);
6       cvCvtColor(needle, needle, CV_BGR2HSV);

8       CvSize patch = cvSize(20,20);

10      IplImage* hay = cvLoadImage(argv[2]);
11      IplImage* dst1 = cvCreateImage(cvSize(hay->width, hay->height), IPL_DEPTH_8U, 1);
12      IplImage* dst2 = cvCreateImage(cvSize(hay->width-patch.width+1,
            hay->height-patch.height+1), IPL_DEPTH_32F, 1);

14      IplImage* h_hay = cvCreateImage(cvSize(hay->width, hay->height), IPL_DEPTH_8U, 1);
15      IplImage* s_hay = cvCreateImage(cvSize(hay->width, hay->height), IPL_DEPTH_8U, 1);
16      IplImage* v_hay = cvCreateImage(cvSize(hay->width, hay->height), IPL_DEPTH_8U, 1);
17      IplImage* planes_hay[] = {h_hay, s_hay};
18      cvSplit(hay,h_hay,s_hay,v_hay,NULL);


21      IplImage* h_needle = cvCreateImage(cvSize(needle->width, needle->height), IPL_DEPTH_8U, 1);
22      IplImage* s_needle = cvCreateImage(cvSize(needle->width, needle->height), IPL_DEPTH_8U, 1);
23      IplImage* v_needle = cvCreateImage(cvSize(needle->width, needle->height), IPL_DEPTH_8U, 1);
24      IplImage* planes_needle[] = {h_needle, s_needle};
25      cvSplit(needle,h_needle,s_needle,v_needle,NULL);

27      CvHistogram* hist;
28      int sizes[] = {15,16};
29      float h_range[] = {0, 180};
30      float s_range[] = {0, 255};
31      float* ranges[] = {h_range, s_range};
32      hist = cvCreateHist(2, sizes, CV_HIST_ARRAY, ranges, 1);

34      cvCalcHist(planes_needle, hist, 0, NULL);

36      cvCalcBackProject(planes_hay, dst1, hist);

38      cvCalcBackProjectPatch(planes_hay, dst2, patch, hist, CV_COMP_INTERSECT, 500);

40      cvNamedWindow("BackProject");
41      cvShowImage("BackProject", dst1);
42      cvWaitKey(0);
43      cvShowImage("BackProject", dst2);
44      cvWaitKey(0);

46      cvSaveImage("histogram_backproject.png", dst1);
47      cvSaveImage("histogram_backproject_patch.png", dst2);
```

```
49      cvReleaseImage(&needle);
50      cvReleaseImage(&hay);
51      cvReleaseImage(&dst1);
52      cvReleaseImage(&dst2);
53      cvReleaseImage(&h_needle);
54      cvReleaseImage(&s_needle);
55      cvReleaseImage(&v_needle);
56      cvReleaseImage(&h_hay);
57      cvReleaseImage(&s_hay);
58      cvReleaseImage(&v_hay);
59      cvReleaseHist(&hist);
60      cvDestroyWindow("BackProject");
61  }
```

Listing 4.21: Erzeugen von Rückprojektionen mittels der Funktionen *cvCalcBackProject* und *cvCalcBackProjectPatch*.



*(a)*Suchmuster *(b)*Rückprojektion *(c)*Umgebungsbasierte Rückprojektion

Abbildung 4.26: Ergebnisse der pixelweisen und umgebungsbasierten Rückprojektion. Als Eingabebilder wurden 4.26a und 4.1b verwendet.

Die Funktionen *cvCalcBackProject* und *cvCalcBackProjectPatch* erzeugen ein Bild, in dem für jeden Pixel die Wahrscheinlichkeit berechnet wird, ob das Histogramm aus diesem Pixel (und seiner Umgebung) entstanden sein könnte. *cvCalcBackProject* liefert dabei die Wahrscheinlichkeiten für einzelne Pixel und *cvCalcBackProjectPatch* für eine Umgebung. Auf einem solchen Bild lässt sich dann durch Bestimmung des Maximums eine mögliche Position des Objekts finden.

4.6 Vorlagenvergleich

Ähnlich wie die Rückprojektion arbeitet auch der Vorlagenvergleich. Jedoch nicht mit Histogrammen, sondern direkt mit den Bilddaten. OpenCV bietet für diesen Zweck die Funktion *cvMatchTemplate*, die genutzt werden kann, um Objekte in einem Bild zu finden. Der Vorlagenvergleich positioniert die Vorlage nacheinander über jedem Pixel des Originalbilds und berechnet den Unterschied zu den darunter liegenden Pixeln im Originalbild. Um so kleiner dieser Unterschied ist, um so höher ist die Wahrscheinlichkeit, dass das Bild an dieser Stelle im Bild liegt.

Im Gegensatz zu dem unter 4.5.3 Vergleichen von Histogrammen gezeigten Verfahren ist der Vorlagenvergleich äußerst anfällig für Änderungen der Größe und der Rotation des Objekts.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* tpl = cvLoadImage(argv[1]);
6       IplImage* img = cvLoadImage(argv[2]);
7       IplImage* dst = cvCreateImage(cvSize(img->width-tpl->width+1, img->height-tpl->height+1),
            IPL_DEPTH_32F, 1);
8       cvNamedWindow("MatchTemplate",CV_WINDOW_AUTOSIZE);
```

```
10      cvMatchTemplate(img,tpl,dst,CV_TM_CCORR_NORMED);
11      cvNormalize(dst,dst,255,0,CV_MINMAX);

13      cvShowImage("MatchTemplate", dst);
14      cvWaitKey(0);
15      cvSaveImage("matchtemplate.png", dst);

17      cvReleaseImage(&img);
18      cvReleaseImage(&dst);
19      cvReleaseImage(&tpl);
20      cvDestroyWindow("MatchTemplate");
21  }
```

Listing 4.22: Durchführen eines Vorlagenvergleichs mittels *cvMatchTemplate*

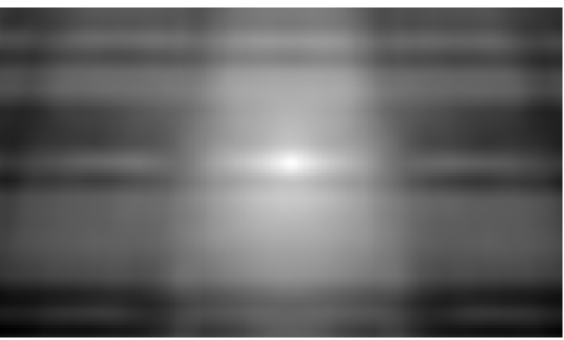


Abbildung 4.27: Eine Wahrscheinlichkeitskarte — das Ergebnis eines Vorlagenvergleichs von 4.26a auf 4.1b. Helle Stellen im Bild zeigen höhere Wahrscheinlichkeit.

Auch *cvMatchTemplate* bietet verschiedene Möglichkeiten für die Berechnung des Unterschieds zwischen der Vorlage und dem Originalbild. Wie auch die Histogrammvergleiche haben diese unterschiedliche Wertebereiche für gute und schlechte Treffer — diese Wertebereiche wie auch die exakten Formeln können [1, S. 367] entnommen werden.

4.7 Segmentierung

Im Gegensatz zum Vorlagenvergleich bietet die Segmentierung die Möglichkeit Objekte in Bildern zu erkennen, die nicht vorher bekannt sind. Dafür teilt die Segmentierung das Bild in Teile mit gleichen Eigenschaften, wie Farbe oder Kontrast, auf. Als Grundlage für diese Aufteilung wird entweder die Ähnlichkeit von verschiedenen Bildbereichen genutzt (4.7.2 Pyramidensegmentierung) oder die Unterschiede zwischen mehreren Bildern, die zu suchende Merkmale enthalten oder nicht enthalten (4.7.1 Hintergrundsubtraktion).

4.7.1 Hintergrundsubtraktion

Eine Möglichkeit der Segmentierung ist die Hintergrundsubtraktion. Dabei wird zunächst ein (oder mehrere) Bild(er) ohne die zu suchenden Merkmale aufgenommen und erst dann ein Bild mit dem Merkmal. Die Daten der Bilder ohne Merkmal werden dann von denen mit Merkmal abgezogen, sodass Bereiche, die sich nicht verändert haben, eliminiert werden.

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* back = cvLoadImage(argv[1]);
6       IplImage* img = cvLoadImage(argv[2]);
7       IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
8       cvNamedWindow("Backgroundsubtraction",CV_WINDOW_AUTOSIZE);
```

```
10      cvAbsDiff(back,img,dst);

12      cvShowImage("Backgroundsubtraction", dst);
13      cvWaitKey(0);
14      cvSaveImage("backgroundsub.png", dst);

16      cvReleaseImage(&img);
17      cvReleaseImage(&back);
18      cvReleaseImage(&dst);
19      cvDestroyWindow("Backgroundsubtraction");
20  }
```

Listing 4.23: Durchführen einer Hintergrundsubtraktion mit einem Hintergrundbild

*(a)*Ausgangsbild ohne Merkmal *(b)*Ausgangsbild mit Merkmal *(c)*Ergebnis der Hintergrundsubtraktion

Abbildung 4.28: Ergebnis der Hintergrundsubtraktion mit den Ausgangsbildern mit und ohne Merkmal.

Abbildung 4.28c zeigt ein Ergebnis der Hintergrundsubtraktion. Sehr gut zu sehen ist, dass Teile, die sich in den Bildern unterscheiden, im Ergebnisbild klar heraus stechen. Durch Rauschen, wackeln der Kamera oder kleinere Schatten entstandene Segmente können nachträglich mit Weichzeichnen und Grenzwertanalyse ausgebessert werden. Über morphologische Operationen kann das Ergebnis ebenfalls weiter verbessert werden, sodass am Ende des Prozesses eine Maske entsteht, die auf das Bild angewandt werden kann. Dies reduziert die weitere Bearbeitung der Bilddaten auf die durch Hintergrundsubtraktion isolierten Bereiche.

Zur Verbesserung des Resultats der Hintergrundsubtraktion bietet es sich an nicht nur ein Bild ohne Merkmal zu nutzen, sondern den Mittelwert über mehrere solcher Bilder. Dieses Vorgehen reduziert die Einflüsse durch Rauschen und periodische Bewegungen, wie das Bewegen von Blättern an einem Baum, die bei einzelnen Bildern für falsche Merkmale sorgen können.

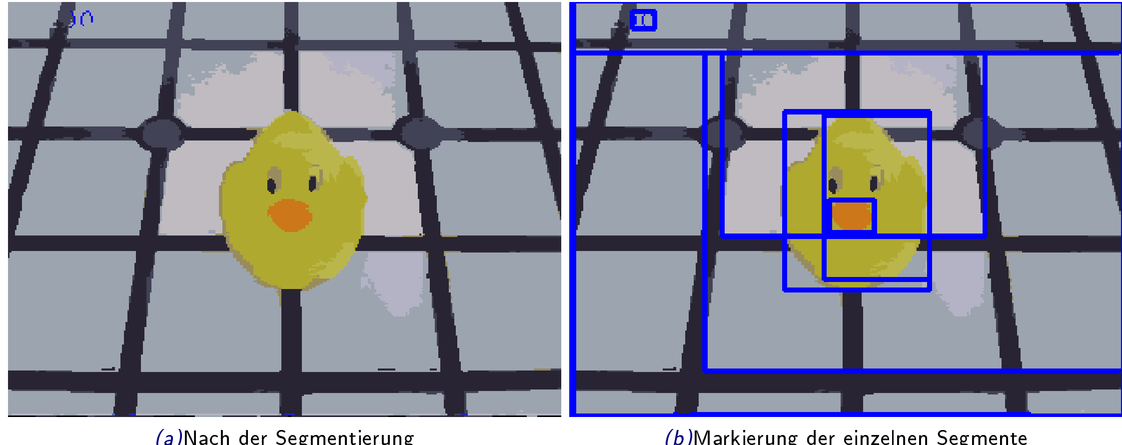
4.7.2 Pyramidensegmentierung

Eine weitere Methode der Segmentierung ist die Pyramiden Segmentierung. Diese Methode bildet die Segmente in den Bilddaten nicht über den Vergleich von Bilddaten mit und ohne Merkmal, sondern über die Ähnlichkeit der (Farb-) Werte in bestimmten Arealen. OpenCV bietet hierfür die Funktion *cvPyrSegmentation*, die für die Segmentierung Gauß- und Laplace-Pyramiden nutzt, wie sie in 4.3.3 Bildpyramiden beschrieben werden. Der durchgeführte Algorithmus wird in der Literatur auch Pyramid Linking genannt. Eine genauere Beschreibung der Funktionsweise findet sich in [8, S. 456].

```
1   #include <highgui.h>
2   #include <cv.h>

4   int main(int argc, char** argv){
5       IplImage* img = cvLoadImage(argv[1]);
6       IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7       cvNamedWindow("PyrSegmentation",CV_WINDOW_AUTOSIZE);

9       CvMemStorage* storage = cvCreateMemStorage(0);
10      CvSeq* comp = NULL;
```

```
12      cvPyrSegmentation(img, dst, storage, &comp, 4, 200, 50);

14      cvSaveImage("pyrsegmentation_raw.png", dst);

16      for(int i=0; i<comp->total; i++){
17          CvConnectedComp* cc = (CvConnectedComp*) cvGetSeqElem(comp,i);
18          cvDrawContours(dst, cc->contour, CV_RGB(255,0,0), CV_RGB(0,255,0), 2, 2);
19          cvRectangle(dst,
20                      cvPoint(cc->rect.x, cc->rect.y),
21                      cvPoint(cc->rect.x + cc->rect.width, cc->rect.y + cc->rect.height),
22                      CV_RGB(0,0,255), 2, 2);
23      }

25      cvShowImage("PyrSegmentation", dst);
26      cvWaitKey(0);
27      cvSaveImage("pyrsegmentation_squares.png", dst);

29      cvReleaseImage(&img);
30      cvReleaseImage(&dst);
31      cvDestroyWindow("PyrSegmentation");
32  }
```

Listing 4.24: Durchführen einer Hintergrundsubtraktion mit einem Hintergrundbild



*(a)*Nach der Segmentierung *(b)*Markierung der einzelnen Segmente

Abbildung 4.29: Die Pyramidensegmentierung liefert zum einen ein Bild, in dem die als zusammengehörig erkannten Flächen einfarbig dargestellt werden, und zum anderen Rechtecke, die jede als Segment erkannte Fläche komplett beinhalten.

Die Pyramidensegmentierung liefert zwei Ergebnisse. Das eine ist ein Bild, in dem jedes erkannte Segment als eine einfarbige, zusammenhängende Fläche dargestellt wird. Die Farbe ergibt sich aus dem Mittelwert der in der Fläche liegenden Farben. Das Zweite Ergebnis der Funktion *cvPyrSegementation* ist eine *CvSeq* mit Rechtecken (Bounding Box), die je ein gefundenes Segment genau beinhalten.

Ein besonderes Augenmerk sollte auf den Parameter *level* der Funktion *cvPyrSegementation* gelegt werden. Da für jedes Level die Höhe und die Breite des Bildes halbiert wird, müssen sowohl die Höhe als auch die Breite des Eingabebildes durch den Faktor 2^{length} teilbar sein. Ist dies nicht der Fall, wirft OpenCV eine Ausnahme mit relativ nichtssagender Beschreibung. Ein Bild mit der Kantenlänge 640×480 lässt sich demnach mit fünf Ebenen bearbeiten ($480 = 2^5 \times 5 \times 3$), wobei sich ein Bild mit 800×600 nur 3 Ebenen ermöglicht ($600 = 2^3 \times 5^2 \times 3$).

4.8 Konturen

Eine weitere Methode zur Erkennung von Merkmalen in Bildern sind die Konturen. Neben der Lage eines Objekts liefern diese auch eine Aussage über dessen Form. OpenCV kann Konturen nur aus Binärbildern (Schwarz-Weiß) extrahieren. Diese werden i.d.R. durch Grenzwertanalyse von vorverarbeiteten Bildern erstellt. Zur Verdeutlichung der Ergebnisse wird im Folgenden auf diese Vorverarbeitung verzichtet, sodass die Quelltextlistings in diesem Abschnitt nur die Verarbeitung des Binärbildes zeigen. Abbildung 4.30 zeigt die verwendeten Bilder.

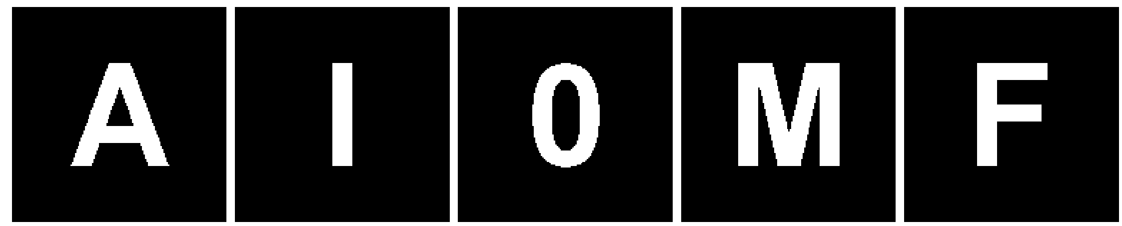


Abbildung 4.30: Binäre Beispielbilder, die in diesem Abschnitt verwendet werden, um die Benutzung der Konturenfunktionen in OpenCV zu zeigen.

4.8.1 Konturen finden und zeichnen

Die grundlegende Operation für das Arbeiten mit Konturen ist das Finden dieser in den Bilddaten. OpenCV bietet zu diesem Zweck die Funktion *cvFindContours*. Konturen werden, wie in 3.3 CvSeq beschrieben als CvSeq zurückgegeben und können dann mit der Funktion *cvDrawContours* gezeichnet werden. Ein Beispiel für gezeichnete Konturen liefert Abbildung 4.31.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7      cvCvtColor(img, dst, CV_GRAY2BGR);

9      cvNamedWindow("Contours",CV_WINDOW_AUTOSIZE);
10     CvMemStorage* storage = cvCreateMemStorage(0);
11     CvSeq* firstContour;

13     int numOfContours = cvFindContours(img, storage, &firstContour,sizeof(CvContour),
           CV_RETR_LIST);

15     cvDrawContours(dst,  firstContour, CV_RGB(255,0,0), CV_RGB(0,0,255),5,2);

17     cvShowImage("Contours", dst);
18     cvWaitKey(0);
19     cvSaveImage("contours.png", dst);

21     cvReleaseMemStorage(&storage);
22     cvReleaseImage(&img);
23     cvReleaseImage(&dst);
24     cvDestroyWindow("Contours");
25 }
```

Listing 4.25: Finden und zeichnen von Konturen in einem Binärbild.

Listing 4.25 zeigt die Anwendung der Funktionen *cvFindContours* und *cvDrawContours*. Besonders zu beachten ist, dass *cvFindContours* das gewünschte Speicherlayout der Konturen übergeben werden kann — die möglichen Ausprägungen wurden unter 3.3.1 Speicherlayout von Konturen als CvSeq bereits vorgestellt.

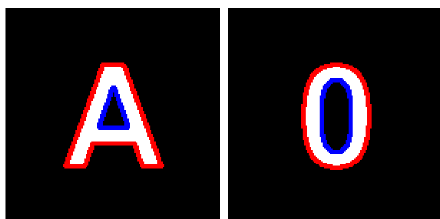
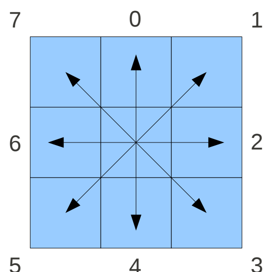


Abbildung 4.31: Binäre Beispielbilder, die in diesem Abschnitt verwendet werden, um die Benutzung der Konturenfunktionen in OpenCV zu zeigen.

Freeman Kettencode

Eine Alternative zur Darstellung der Konturen als Polygone ist die Darstellung als Kettencode. Der Kettencode speichert nicht die einzelnen Punkte, sondern Schritte von einem Punkt der Kontur zum nächsten. Die Schritte können dabei in 8 Richtungen gemacht werden. Abbildung 4.32 zeigt die möglichen Richtungen und ihre Codierung als Zahlen.



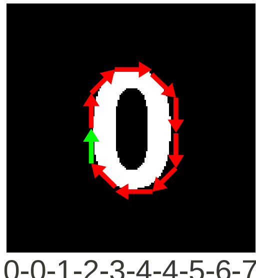


Abbildung 4.32: Richtungen und deren Codierung im Freeman Kettencode und Beispielbild mit Codierung im Freeman Kettencode beginnend ab dem grünen Pfeil.

```
1  #include <highgui.h>
2  #include <cv.h>

4  int main(int argc, char** argv){
5      IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
6      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 3);
7      cvCvtColor(img, dst, CV_GRAY2BGR);

9      cvNamedWindow("Freeman Contours",CV_WINDOW_AUTOSIZE);
10     CvMemStorage* storage = cvCreateMemStorage(0);
11     CvSeq* firstContour;

13     int numOfContours = cvFindContours(img, storage, &firstContour,sizeof(CvChain),
           CV_RETR_LIST, CV_CHAIN_CODE);

15     cvDrawContours(dst,  firstContour, CV_RGB(255,0,0), CV_RGB(0,0,255),5,2);

17     cvShowImage("Freeman Contours", dst);
18     cvWaitKey(0);
19     cvSaveImage("contoursfreeman.png", dst);

21     cvReleaseMemStorage(&storage);
22     cvReleaseImage(&img);
23     cvReleaseImage(&dst);
24     cvDestroyWindow("Freeman Contours");
25 }
```

Listing 4.26: Finden und zeichnen von Konturen als Freeman Kettencode.

4.8.2 Konturen vergleichen

Über die gefundenen Konturen lassen sich Aussagen über die Form des gefundenen Objekts machen. Diese sind unter anderem Momente und geometrische Histogramme.

Momente von Konturen

Der einfachste Weg Aussagen über eine Kontur zu treffen sind Momente und Zentralmomente. Mit OpenCV lassen sich diese mit der Funktion *cvContoursMoments* berechnen. Die berechneten Momente werden in einer eigenen Struktur, *CvMoments*, gespeichert. Die Momente m und Zentralmomente μ sind Summen über die Konturen und wie folgt definiert:

$$m_{p,q} = \sum_{i=1}^{n} I(x,y)x^p y^q$$

$$\mu_{p,q} = \sum_{i=1}^{n} I(x,y)(x-\overline{x})^p(y-\overline{y})^q$$

Diese berechneten Momente sind jedoch rotations-, größen- und auch lageabhängig, sodass sie keinen guten Vergleich von Konturen erlauben. Abhilfe schaffen die normalisierten Momente, die sich aus den Momenten und Zentralmomenten berechnen lassen, und weder größen- noch lageabhängig sind. Durch Kombination von normalisierten Momenten lässt sich dann eine rotationsunabhängige Variante der Momente berechnen: die Hu Momente. Das genaue Verfahren der Berechnung wird in [7, S. 254] erklärt.

OpenCV berechnet mit der Funktion *cvContoursMoments* sowohl Momente als auch Zentralmomente. Die Hu Momente lassen sich dann mit der Funktion *cvGetHuMoments* berechnen. Neben den Momenten aus Konturen können Momente auch von Bildern berechnet werden. Dies übernimmt die Funktion *cvMoments*.

```
1  #include <highgui.h>
2  #include <cv.h>
3  #include <stdio.h>

5  int main(int argc, char** argv){
6      IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
7      IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
8      cvZero(dst);

10     cvNamedWindow("Contours",CV_WINDOW_AUTOSIZE);
11     CvMemStorage* storage = cvCreateMemStorage(0);
12     CvSeq* firstContour;

14     int numOfContours = cvFindContours(img, storage, &firstContour,sizeof(CvContour),
           CV_RETR_LIST);

16     cvDrawContours(dst,  firstContour, cvScalarAll(255), cvScalarAll(255), 1, -1, 1);

18     CvMoments moments;
19     cvMoments(dst,&moments,1);

21     printf("Moments:\n%e\t%e\t%e\t%e\t%e\t%e\t%e\t%e\t%e\t%e\t\n",moments.m00, moments.m10,
           moments.m01, moments.m20, moments.m11, moments.m02,moments.m30, moments.m21,
           moments.m12, moments.m03);

23     printf("Central Moments:\n%e\t%e\t%e\t%e\t%e\t%e\t%e\t\n",moments.mu20, moments.mu11,
           moments.mu02, moments.mu30, moments.mu21, moments.mu12, moments.mu03);

25     CvHuMoments hu;
26     cvGetHuMoments(&moments, &hu);

28     printf("Hu Moments:\n%e\t%e\t%e\t%e\t%e\t%e\t%e\t\n",hu.hu1, hu.hu2, hu.hu3, hu.hu4,
           hu.hu5, hu.hu6, hu.hu7);

30     cvShowImage("Contours", dst);
31     cvWaitKey(0);
32     cvSaveImage("contours.png", dst);
```

```
34      cvReleaseMemStorage(&storage);
35      cvReleaseImage(&img);
36      cvReleaseImage(&dst);
37      cvDestroyWindow("Contours");
38  }
```

Listing 4.27: Berechnen der verschiedenen Momente aus einer Kontur.

Vor allem die Hu Momente eignen sich zum Vergleich von Konturen. Ein Beispiel in Listing 4.28 zeigt wie nacheinander für die selbe Kontur in verschiedenen Rotationen und Skalierungen die Hu Momente berechnet werden.

```
1   #include <highgui.h>
2   #include <cv.h>
3   #include <stdio.h>

5   int main(int argc, char** argv){
6       IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
7       IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);
8       cvCopy(img, dst);

10      cvNamedWindow("Contours",CV_WINDOW_AUTOSIZE);

12      CvMat* rotate = cvCreateMat(2,3,CV_32FC1);
13      CvPoint2D32f center = cvPoint2D32f(img->width/2, img->height/2);

15      CvMoments moments;
16      CvHuMoments hu;

18      for(int i=0; i<=12; i++){
19          cv2DRotationMatrix(center, i*30.0, 1.0, rotate);
20          cvWarpAffine(img, dst, rotate);

22          cvMoments(dst,&moments,1);
23          cvGetHuMoments(&moments, &hu);

25          printf("%3d: %e\t%e\t%e\t%e\t%e\t%e\t%e\t\n",i*30,hu.hu1, hu.hu2, hu.hu3, hu.hu4,
                hu.hu5, hu.hu6, hu.hu7);

27          cvShowImage("Contours", dst);
28          cvWaitKey(0);
29      }

31      for(int i=1; i<=8; i++){
32          cv2DRotationMatrix(center, .0, 2.0/i, rotate);
33          cvWarpAffine(img, dst, rotate);

35          cvMoments(dst,&moments,1);
36          cvGetHuMoments(&moments, &hu);

38          printf("%3d: %e\t%e\t%e\t%e\t%e\t%e\t%e\t\n",(int)(2.0/i*100),hu.hu1, hu.hu2, hu.hu3,
                hu.hu4, hu.hu5, hu.hu6, hu.hu7);

40          cvShowImage("Contours", dst);
41          cvWaitKey(0);
42      }

44      cvReleaseImage(&img);
45      cvReleaseImage(&dst);
46      cvReleaseMat(&rotate);
47      cvDestroyWindow("Contours");
48  }
```

Listing 4.28: Berechnen der Hu Momente eines Bildes nach Rotation und Skalierung.

Geometrische Histogramme

Die Grundlage für geometrische Histogramme bilden Konturen auf Basis von Kettencodes. Im einfachsten Fall zählen diese Histogramme das Vorkommen der einzelnen Richtungen der Kettenglieder. Geometrische Histogramme sind lageunabhängig, aber größen- und rotationsabhängig,

jedoch äußern sich Rotationen um vielfache von 45° lediglich in einer Verschiebung. Ebenso verhält sich die Größenabhängigkeit sehr gutartig, sodass normalisierte Histogramme quasi nicht größenabhängig sind.

OpenCV nimmt die Idee des geometrischen Histogrammes auf und bietet mit der Funktion *cvCalcPGH* die Möglichkeit ein ähnliches Histogramm auf Basis von Polygonen zu erstellen. Als Merkmale für dieses zweidimensionale Histogramm dienen die Winkel und die minimalen bzw. maximalen Abstände zwischen den Segmenten der Polygone.

```
1  #include <highgui.h>
2  #include <cv.h>
3  #include <stdio.h>

5  void drawHist(CvHistogram* hist){
6      int sizes[] = {15, 15};
7      int scale = 10;
8      IplImage* dst = cvCreateImage(cvSize(scale*sizes[0], scale*sizes[1]), IPL_DEPTH_8U, 3);
9      cvZero(dst);

11     float maxVal = .0f;
12     cvGetMinMaxHistValue(hist, NULL, &maxVal, NULL, NULL);

14     for(int i=0; i<sizes[0]; i++){
15         for(int j=0; j<sizes[1]; j++){
16             float value = cvQueryHistValue_2D(hist, i, j);
17             int intensity = cvRound(value * 255 / maxVal);
18             cvRectangle(dst,
19                         cvPoint(i*scale, j*scale),
20                         cvPoint((i+1)*scale, (j+1)*scale),
21                         cvScalar(intensity, intensity, intensity),
22                         CV_FILLED);
23         }
24     }
25     cvShowImage("Histogram", dst);
26     cvReleaseImage(&dst);
27 }

29 int main(int argc, char** argv){
30     IplImage* img = cvLoadImage(argv[1], CV_LOAD_IMAGE_GRAYSCALE);
31     IplImage* dst = cvCreateImage(cvSize(img->width, img->height), IPL_DEPTH_8U, 1);

33     cvNamedWindow("Histogram",CV_WINDOW_AUTOSIZE);
34     cvNamedWindow("Contours",CV_WINDOW_AUTOSIZE);
35     CvMemStorage* storage = cvCreateMemStorage(0);
36     CvSeq* firstContour;

38     CvHistogram* hist;
39     int sizes[] = {15, 15};
40     float range1[] = {0, 180};
41     float range2[] = {0, 180};
42     float* ranges[] = {range1, range2};
43     hist = cvCreateHist(2, sizes, CV_HIST_ARRAY, ranges, 1);

45     CvMat* rotate = cvCreateMat(2,3,CV_32FC1);
46     CvPoint2D32f center = cvPoint2D32f(img->width/2, img->height/2);

48     for(int i=0; i<=8; i++){
49         cv2DRotationMatrix(center, i*45.0, 1.0, rotate);
50         cvWarpAffine(img, dst, rotate);

52         cvClearHist(hist);
53         int numOfContours = cvFindContours(dst, storage, &firstContour, sizeof(CvContour),
               CV_RETR_LIST);
54         cvCalcPGH(firstContour, hist);
55         cvShowImage("Contours", dst);
56         drawHist(hist);
57         cvWaitKey(0);
58     }

60     for(int i=1; i<=8; i++){
61         cv2DRotationMatrix(center, .0, 2.0/i, rotate);
62         cvWarpAffine(img, dst, rotate);

64         cvClearHist(hist);
65         int numOfContours = cvFindContours(dst, storage, &firstContour, sizeof(CvContour),
               CV_RETR_LIST);
66         cvCalcPGH(firstContour, hist);
```

```
67          cvShowImage("Contours", dst);
68          drawHist(hist);
69          cvWaitKey(0);
70      }

72      cvReleaseMemStorage(&storage);
73      cvReleaseImage(&img);
74      cvReleaseImage(&dst);
75      cvReleaseMat(&rotate);
76      cvDestroyWindow("Contours");
77      cvDestroyWindow("Histogram");
78  }
```

Listing 4.29: Berechnen der Histogramme einer Kontur nach Rotation und Skalierung.

5 Fazit und Ausblick

OpenCV bietet viele Möglichkeiten im Bereich der Bildverarbeitung. Neben den vorgestellten Funktionen zur Bildverarbeitung bietet die Bibliothek aber zudem Funktionen für maschinelles Lernen. Eine Disziplin der Informatik, die, wie die Bildverarbeitung, zunehmend wichtiger wird. Insbesondere, aber nicht ausschließlich, finden diese Teilgebiete in der automatisierten Steuerung von Robotern eine Anwendung.

Nicht nur die jüngste Entwicklung dieser beiden Disziplinen mit der Entdeckung immer besserer und/oder schnellerer Algorithmen, sondern auch der Fortschritt von tragbaren Geräten, wie Smartphones mit immer leistungsfähigeren Prozessoren, lassen die Möglichkeiten von bildbasierten Anwendungen für den Endverbraucher wachsen.

Zwei Anwendungsgebiete sollen an dieser Stelle einen Eindruck vermitteln, wie leistungsfähig blildbasierte Programme heute sind.

5.1 Mobile Augmented Reality: Recognizr

Recognizr ist eine Anwendung für (Android) Mobiltelefone, die Kameradaten mit Daten aus sozialen Netzwerken kombiniert. Die Anwendung durchsucht dafür die, über die Accounts des Nutzers in verschiedenen Sozialen Netzwerken, wie Facebook oder MySpace, verknüpften Kontakte. Zudem kann der Benutzer zu einem erkannten Gesicht weitere Informationen, wie Visitenkarten oder Notizen, speichern.

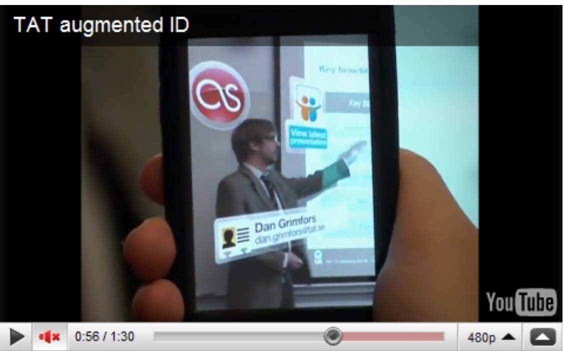


Abbildung 5.1: Screenshot eines Promotionvideos des Herstellers, der die Anwendung Recognizr auf einem Handybildschirm zeigt.

5.2 Robotik: Nao Roboter

Der humanoide Nao Roboter wurde als Standardplattform für den RoboCup entworfen und hat als solcher 2007 den vierbeinigen Aibo ersetzt. Der Nao ist von haus aus mit den Laufzeitbibliotheken von OpenCV ausgestattet, sodass sich Programme direkt auf den Roboter übertragen lassen. Auf dem mit 500 MHz getakteten Prozessor lassen sich, je nach Algorithmus, circa zehn Bilder pro Sekunde in VGA Auflösung verarbeiten.

ABBILDUNGSVERZEICHNIS

1.1 Die grundlegende Struktur von OpenCV nach [7] 2

3.1 Beispiele für Farben mit den Repräsentationen in BGR und HSV Werten 6
3.2 Graphische Darstellung des RGB Farbraums aus [2] 6
3.3 HSL und HSV Farbräume aus [2] 7
3.4 Beispielbild und gefundene Konturen 9
3.5 Verschiedene Anordnungen von Konturen 10

4.1 Verwendete Beispielbilder 14
4.2 Ergebnisse des einfachen Weichzeichners 16
4.3 Ergebnisse des medialen Weichzeichners 16
4.4 Ergebnisse des gaußschen Weichzeichners 17
4.5 Ergebnisse des bilateralen Weichzeichners 17
4.6 Quellbild und Ergebnis nach binärem Schwellwert 18
4.7 Weitere Schwellwerte: abgeschnitten und "Null" 19
4.8 Ergebnisse von *cvThreshold* bei globalem und adaptivem Schwellwert 20
4.9 Quellbild mit 20x20 Pixeln und die Ergebnisse nach Dilatation und Erosion mit einem quadratischen 3x3 Kernel 20
4.10 Quellbild mit 20x20 Pixeln und die Ergebnisse nach Opening und Closing mit einem quadratischen 3x3 Kernel 21
4.11 Quellbild mit 20x20 Pixeln und die Ergebnisse nach Top Hat und Black Hat mit einem quadratischen 3x3 Kernel 21
4.12 Anwendung des morphologischen Gradienten auf zwei Beispielbilder 22
4.13 Verschiedene Faltungs-Matrizen, G_{smooth}: einfacher Weichzeichner, G_{sobel}: Sobel Gradient. Die Werte in Klammern stellen jeweils die Position des Ankers dar. . . . 23
4.14 Ergebnis der in Listing 4.9 gezeigten Faltung. 24
4.15 Ergebnisse des in 4.10 gezeigten Quellcodes. 25
4.16 Zwei Ergebnisse des in 4.11 gezeigten Quellcodes. 26
4.17 Zwei Ergebnisse des in 4.12 gezeigten Quellcodes. 26
4.18 Ergebnisse des in 4.13 gezeigten Quellcodes. 27
4.19 Ergebnis des in Listing 4.14 gezeigten Programms. 29
4.20 Affine und perspektivische Transformationen 30
4.21 Ergebnisse des in 4.15 gezeigten Quellcodes. 31
4.22 Ergebnis des in 4.16 gezeigten Quellcodes. 32
4.23 Ergebnisse des in 4.17 gezeigten Quellcodes. Die gefundenen Linien wurden rot markiert. 33
4.24 Ergebnis des Histogramm Ausgleichs anhand von einem Kamerabild mit relativ schlechtem Kontrast. 34
4.25 Gezeichnetes H-S Histogramm des in Abbildung 4.1b gezeigten Bildes. Der Farbwert ist auf der x- und die Sättigung auf der y-Achse. Die Helligkeit gibt die Anzahl des Auftretens an. 35
4.26 Ergebnisse der pixelweisen und umgebungsbasierten Rückprojektion. Als Eingabebilder wurden 4.26a und 4.1b verwendet. 38
4.27 Eine Wahrscheinlichkeitskarte — das Ergebnis eines Vorlagenvergleichs von 4.26a auf 4.1b. Helle Stellen im Bild zeigen höhere Wahrscheinlichkeit. 39

4.28 Ergebnis der Hintergrundsubtraktion mit den Ausgangsbildern mit und ohne Merkmal. 40
4.29 Die Pyramidensegmentierung liefert zum einen ein Bild, in dem die als zusammengehörig erkannten Flächen einfarbig dargestellt werden, und zum anderen Rechtecke, die jede als Segment erkannte Fläche komplett beinhalten. 41
4.30 Binäre Beispielbilder, die in diesem Abschnitt verwendet werden, um die Benutzung der Konturenfunktionen in OpenCV zu zeigen. 42
4.31 Binäre Beispielbilder, die in diesem Abschnitt verwendet werden, um die Benutzung der Konturenfunktionen in OpenCV zu zeigen. 43
4.32 Richtungen und deren Codierung im Freeman Kettencode und Beispielbild mit Codierung im Freeman Kettencode beginnend ab dem grünen Pfeil. 43

5.1 Screenshot eines Promotionvideos des Herstellers, der die Anwendung Recognizr auf einem Handybildschirm zeigt. 48

LITERATURVERZEICHNIS

[1] WILLOW GARAGE INC. (Hrsg.): *OpenCV Reference Manual v2.1.* März 2010

[2] WIKIPEDIA FOUNDATION INC. (Hrsg.): *HSL and HSV.* http://en.wikipedia.org/wiki/HSL_and_HSV. Version: 2010-07-21

[3] WIKIPEDIA FOUNDATION INC. (Hrsg.): *RGB color model.* http://en.wikipedia.org/wiki/RGB_color_model. Version: 2010-07-21

[4] WILLOW GARAGE INC. (Hrsg.): *OpenCV Developer Version SVN Server.* https://code.ros.org/svn/opencv/trunk/. Version: 2010-07-29

[5] BRADSKI, Gary u. a. ; WILLOW GARAGE INC. (Hrsg.): *OpenCV Wiki.* http://opencv.willowgarage.com/wiki/. Version: 2010-07-20

[6] BRADSKI, Gary ; DAVIES, Bob u. a.: *Open Computer Vision Library at Source Forge.* http://sourceforge.net/projects/opencvlibrary/. Version: 2010-07-29

[7] BRADSKI, Gary ; KAEHLER, Adrian: *Learning OpenCV.* First. Sebastopool, CA : O'Reilly Media Inc., 2008 http://oreilly.com/catalog/9780596516130. – ISBN 978–0–596–51613–0

[8] JÄHNE, Prof. Dr. B.: *Digitale Bildverarbeitung.* 5. Springer-Verlag Berlin, 2002. – ISBN 3–540–51260–3