OCR-Bibliothek

Einführung

Im Zuge der Automatisierung von technischen Prozessen (Fertigung, Qualitätskontrollen) besteht ein immer größer werdender Bedarf an Systemen für die Erkennung von Schriftzeichen (im weiteren OCR = Optical Character Recognition genannt).
Die OCR-Bibliothek stellt für den Anwender Funktionen zur Verfügung, die ihm mit Hilfe eines Framegrabbers und einer CCD-Kamera in die Lage versetzen, Schrifterkennung auf einem IBM-AT oder kompatiblen Rechner durchzuführen.
Ein konkretes Einsatzgebiet für die OCR-Bibliothek ist die Erkennung von Seriennummern auf Silizium-Wafern im Halbleiterfertigungsprozess. Darüberhinaus sind jedoch auch weitere Anwendungsmöglichkeiten der OCR-Bibliothek denkbar, insbesondere bei Stückgutprozessen beispielsweise in der Lagerhaltung, wo bislang haupsächlich auf Strichcodes basierende Systeme zum Einsatz kommen.

Verarbeitungsstufen der OCR-Bibliothek

Die Schrifterkennung mit Hilfe der OCR-Bibliothek gliedert sich in drei Verarbeitungsstufen, welche in Bild 1 schematisch dargestellt werden.

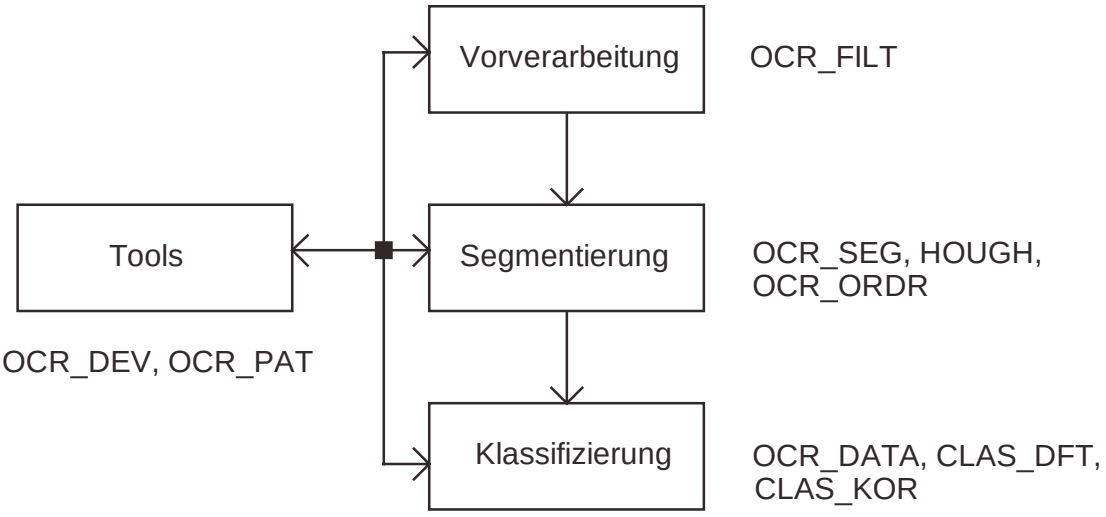


Bild 1: Verarbeitungsstufen der OCR-Bibliothek

Vorverarbeitung

In Ihr wird das Bild durch ein geeignetes Verfahren biniarisiert, d.h. es wird entschieden, ob ein Bildpunkt für die Weiterverarbeitung relevant ist oder nicht. In der OCR-Bibliothek ist dies durch einen Kantenfilter realisiert, die auf zwei Sobel-Matritzen für X- und Y-Richtung basiert. Die Matrizen weisen folgende Belegung auf:

$$\text{Sobel X} = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

$$\text{Sobel Y} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

Bild 2: Aufbau der Sobel-Matriten

Der Wert eines Pixels für den Sobel-X bzw. Sobel-Y Anteil berechnet sich daher wie folgt:

$$\begin{aligned} \text{Sobel X}(x,y) &= i(x-1,y-1) + 2i(x,y-1) + i(x+1,y-1) - \\ &\quad i\ (x-1,y+1) - 2i(x,y+1) - i(x+1,y+1) \\ \text{Sobel Y}(x,y) &= i(x-1,y-1) - i(x+1,y-1) + 2i(x-1,y) - \\ &\quad 2i(x+1,y) + i(x-1,y+1) - i(x+1,y+1) \end{aligned}$$

Bild 3 Berechnung des Sobel-X und Sobel-Y Anteils

Der Sobel-X und Sobel-Y Anteil eines Pixels wird nun nach folgender Formel zu einem Wert zusammengefaßt, der letztendlich das Filterergebnis darstellt.

$$\text{Sobel}(x,y) = \max\left(|\text{Sobel X}(x,y)|, |\text{Sobel Y}(x,y)|\right)$$

Bild 4: Berechnung des Sobel-Werts eines Pixels

Bild 5 und Bild 6 zeigen anhand einer Beispielvorlage die Wirkungsweise des Sobel-Filters.

Bild 5: Beispielvorlage

Bild 6: Beispielvorlage nach Anwendung des Sobel-Filters

Segmentierung

Die Segmentierung versucht, zusammengehörige Pixel im Bild zu lokalisieren (hier Zeichen, es sind aber auch beliebige andere Objekte denkbar). Die Segmentierung in der OCR-Bibliothek basiert auf einem Konturpunktverfahren, das mit Hilfe von Startpunkt und Richtungsketten das segmentierte Objekt beschreibt. Es existieren insgesammt 8 Richtungs-Codes:

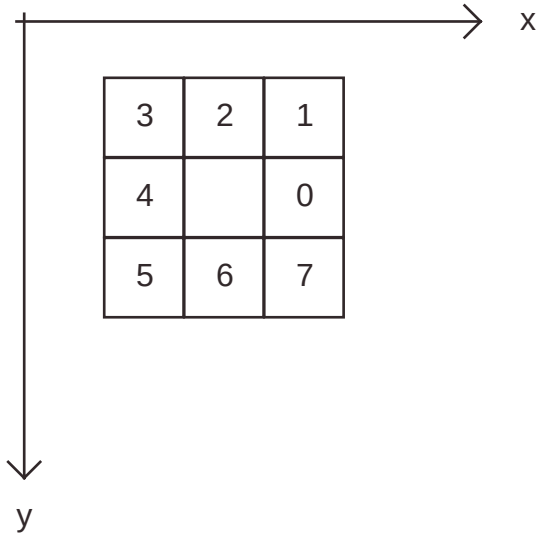


Bild 7: Richtungscodes

Der Aufbau einer Richtungskette wird in Bild 8 anhand eines Beispielobjekts gezeigt.

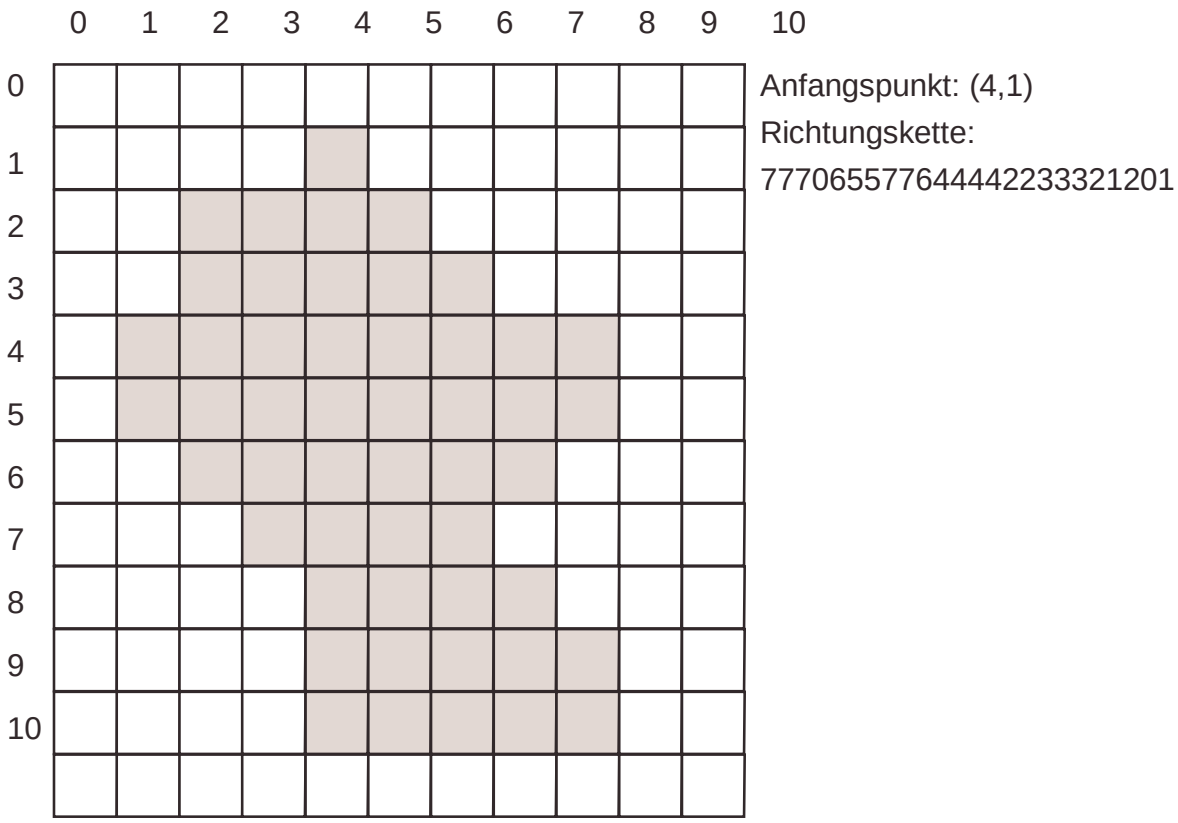


Bild 8: Aufbau einer Richtungskette

Da für die korrekte Lesereihenfolge der Rotationswinkel der Vorlage eine entscheidende Rolle spielt (Von links nach rechts, von oben nach unten), werden nach der Segmentierung die Startkoordinaten der segmentierten Objekte einer Hough-Transformation unterzogen.

$$p_k = x\cos\theta(k) + y\sin\theta(k)$$
$$H(p_k,k) = H(p_k,k) + 1$$
$$\theta(k) = 0\ldots\pi - \frac{\pi}{k}$$

Bild 10: Houghtransformation

Was ist darunter zu verstehen ? Nun, Für jede Startkoordinate wird die Transformation k mal durchgeführt und das Ergebnis in einem sogenannten Akkumulator-Array festgehalten. Die Häufungspunkte in diesem Akkumulator-Array stellen erkannte Linienzüge dar. Im konkreten Anwendungsfall (Bestimmung der Rotation der Vorlage) wird nur das Häufungsmaximum gesucht, aus der k-Koordinate kann dann leicht der Roationswinkel bestimmt werden. Bild 11 zeigt zum besseren Verständnis Abbildungen der Transformation von zwei Linien. Schön zu erkennen sind die Häufungsmaxima, aus denen sich wie schon gesagt, die Winkel der Linien bestimmen lassen.

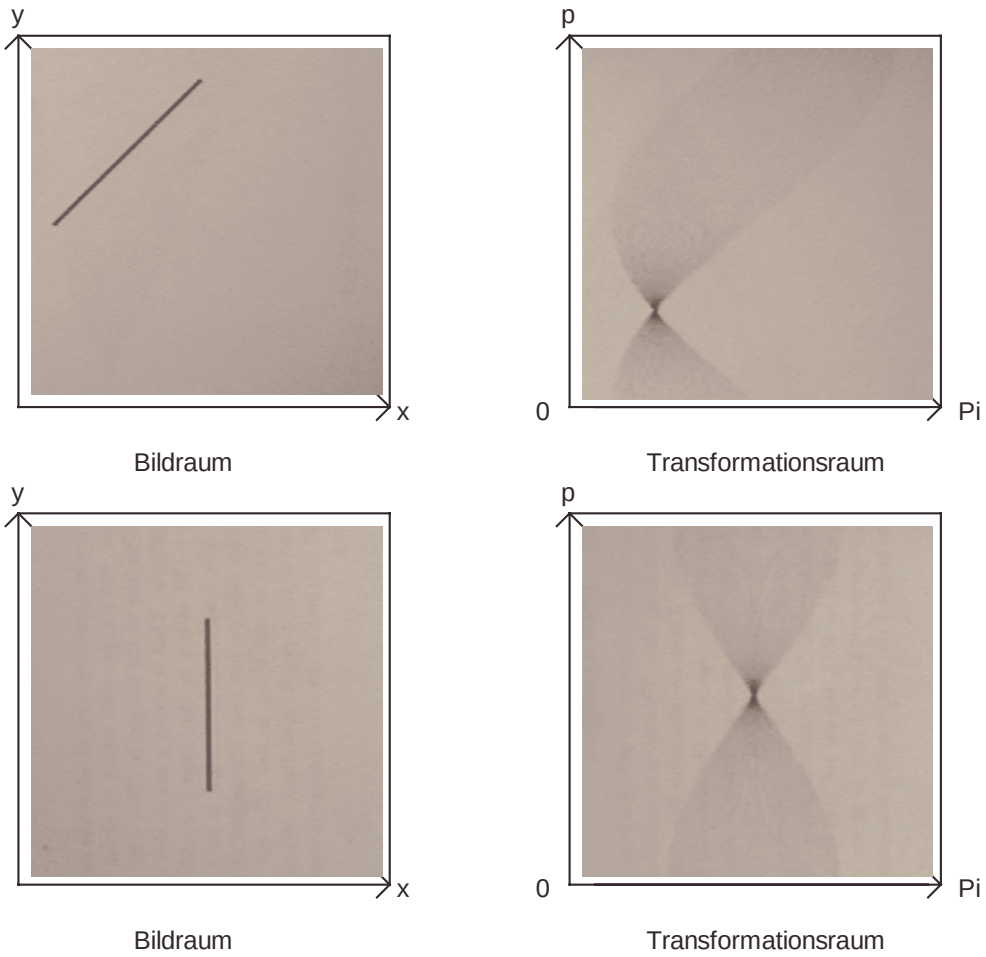


Bild 11: Ergebnisse einer Houghtransformation

Klassifizierung

Die Klassifizierung interpretiert die segmentierten Objekte anhand einer Referenzbibliothek und ordnet ihnen den zugehörigen ASCII-Code zu. In der OCR-Bibliothek existieren dazu zwei Verfahren:

Pattern Matching

Beim ersten Verfahren handelt es sich um eine modifizierte Form des Patternmatchings, wobei das zu klassifizierende Zeichen pixelweise mit allen in der Referenzbibliothek vorhandenenen Zeichen verglichen wird. Dazu ist es notwendig, mit einem festen Zeichenraster (beispielsweise 12x16) zu arbeiten, was eine Normierung des zu erkennenden Zeichens eben auf dieses Maß notwendig macht. Der eigentliche Pixelvergleich wird durch folgende Formel beschrieben

$$r(I,M) = \frac{\left[N\sum_i I_i M_i - (\sum_i I_i)(\sum_i M_i \right]}{\sqrt{\left[N\sum_i I_i^2 - (\sum_i I_i)^2 \right]\left[N\sum_i M_i^2 - (\sum_i M_i)^2 \right]}}$$

dabei:

N = Anzahl Pixel

I_i = Pixelintensitä t desi - ten Pixels des Zeichens

M_i = Pixelintensitä t desi - ten Pixels des Vergleichszeichens

$r(I,M)$ = Ergebnis des Pixelsvergleichs

Bild 12: Klassifizierungsverfahren ´Patternmatching´

Das Ergebnis des Pixelsvergleich bewegt sich zwischen -1.0 und +1.0. Dabei stellt -1.0 keine Übereinstimmung, 1.0 perfekte Übereinstimmung dar. Nach dem das Zeichen mit allen Zeichen der Zeichenbibliothek verglichen worden ist, stellt der Maximalwert des Pixelsvergleichs das erkannte Zeichen dar, dessen ACII-Code zurückgeliefert wird.

Fourierdeskriptoren

Das zweite Klassifizierungverfahren basiert auf sog. Fourier-Deskriptoren. Dazu wird die aus der Segmentierung gewonnene Richtungskette in relativ zum Startpunkt bezogene X und Y-Koordinateten umgewandelt (xn , yn), auf eine durch die Zeichenbibliothek vorgegebene Länge (N) normiert, und eine diskrete Fouriertransformation mit k Koeffizienten nach folgenden Formeln durchgeführt:

$$F(k) = \sum_{n=0}^{N-1}\left[x_n \cos\frac{2\pi nk}{N} + y_n \sin\frac{2\pi nk}{N} \right] + j\sum_{n=0}^{N-1}\left[y_n \cos\frac{2\pi nk}{N} + x_n \sin\frac{2\pi nk}{N} \right]$$

$$\left|F(k)\right| = \sqrt{\Re\left\{F(k)\right\}^2 + \Im\left\{F(k)\right\}^2}$$

Für $k = 1 \ldots N-1$

Bild 13: Klassifizierungsverfahren ´Fourierdeskriptoren´

Beim Klassifizierungsprozess wird nun das gewonnene Spektrum des zu erkennenden Zeichens mit allen Spektren der Zeichenbibliothek verglichen, der ASCII-Code des Zeichens mit dem ähnlichsten Spektrum stellt dann das erkannte Zeichen dar. Dieses Verfahren hat den Vorteil, größen- sowie rotationsinvariant zu sein. Aus dem Spektrum kann überdies hinaus der Rotationswinkel des Zeichens mit Hilfe eines Fourierkoeffizienten nach folgender Formel bestimmt werden:

$$\phi(k) = \arctan\left(\frac{\Im\{F(k)\}}{\Re\{F(k)\}}\right)$$

Rotationswinkel = $\phi(k) - \phi'(k)$

dabei:

$\phi(k)$ = Phase Fourierdeskriptor des zu erkennenden Zeichens

$\phi'(k)$ = Phase Fourierdeskriptor des erkannten Zeichens in der Zeichenbibliothek

Bild 14: Rotationswinkel eines Zeichens

Dazu ist der betragsmäßig größte Fourierdeskriptor zu wählen, um eine möglichst hohe Winkelgenauigkeit zu erzielen. Allerdings muß nach der Bestimmung des Rotationswinkels eine Korrektur des Startpunktes auf der Konturlinie erfolgen:

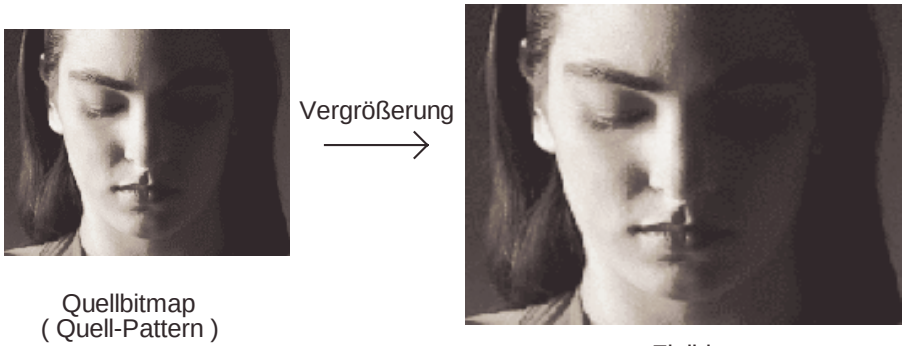
$$\tilde{\phi} = \phi(k+1) - \phi(k) + \phi'(k+1) - \phi'(k)$$

Bild 15: Phasenkorrektor eines Zeichens

Dieser Korrekturwinkel muß vom ermittelten Rotationswinkel subtrahiert werden, um den tatsächlichen Rotationswinkel zu erhalten.

Tools

Tools enthält Routinen, deren Funktionalitäten von anderen Modulen der OCR-Bibliothek verwendet werden. Hierzu zählen neben Schnittstellenfunktionen zur verwendeten Framegrabber-Hardware insbesondere Funktionen für die Bearbeitung von rechteckigen Bildausschnitten (Pattern). Eine Funktion, Normierung von Pattern, soll nun hier näher beschrieben werden, kann sie doch, ohne Übertreibung, als *State of the Art* bezeichnet werden. Bild 15 zeigt zum besseren Verständnis der Problemstellung, was unter Normierung (Vergrößerung/Verkleinerung auf definiertes Maß) von Pattern zu verstehen ist.



Quellbitmap
(Quell-Pattern)

Zielbitmap
(Ziel-Pattern)

Bild 15: Vergrößerung ein Bitmap (256 Graustufen)

Das eigentliche Problem bei der Normierung besteht darin, daß die Abmessungen der Zielbitmap (Höhe, Breite) notwendigerweise nicht ein ganzzahliges Vielfaches

(bzw. Quotient) der Abmessungen der Quellbitmap sein müssen. Wenn dieser Fall vorliegt, sind zwei Vorgehensweisen denkbar:

- Normierung ohne Interpolation von Zwischenwerten
- Normierung mit Interpolation von Zwischenwerten

Die erste Vorgehensweise birgt die Gefahr, daß bei einer Verkleinerung der Quellbitmap relevante Informationen verlorengehen (feine Linienzüge o.ä.), bei einer Vergrößerung häßliche Treppeneffekte entstehen können.
In der gängigen Literatur wird zur Interpolation von Zwischenwerten hauptsächlich ein Verfahren beschrieben, welches auf dem sogenannten Rothstein-Code basiert. Hierbei handelt es sich um den Weiman-Algorithmus, welcher von Feibush, Levoy und Cook entwickelt worden ist.[1]
Der von mir entwickelte Algorithmus hat hingegen drei entscheidende Vorteile:

- Besseres Laufzeitverhalten
- Verkleinerung bzw. Vergrößerung mit dem selben Algorithmus
- Verwendung von Integerarithmetik

Wie arbeitet nun dieser Algorithmus ?

Bild 16 zeigt anhand eines Beispiels schematisch die Verkleinerung einer 5*3 auf eine 3*2 Bitmap.

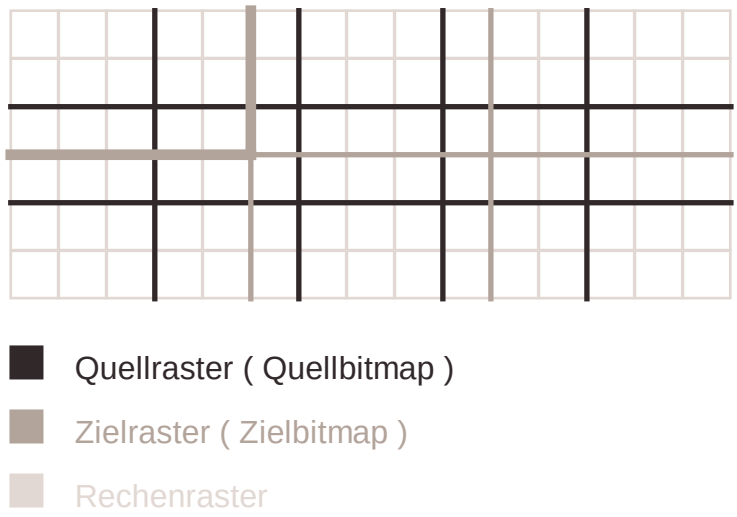


Bild 16: Verkleinerung einer 5*3 auf eine 3*2 Bitmap

Daß Bild 16 mit einem 15*6 Raster unterlegt ist, ist kein Druckfehler, sondern eine Tatsache, die zum Verständnis des Algorithmus unbedingt erforderlich ist. Dieses stellt das interne Rechenraster dar, Aktionen (holen eines neuen Farbwerts aus der Quellbitmap, speichern eines Farbwertes in die Zielbitmap) finden immer an diesen Rastergrenzen dar. Die Farbwerte der Zielbitmap

[1]Computer Graphics, p 822 ff, Addison-Wesley,1992

setzen sich immer aus mehreren gewichteten Rechtechteckflächen zusammen (Bei der Verkleinerung aus Teilen davon, siehe unten)
Beispielsweise berechnet sich der Wert des Pixels 0,0 (im Bild hervorgehoben) der Zielbitmap:

$$dst(0,0) = \frac{6}{15} src(0,0) + \frac{4}{15} src(0,1) + \frac{3}{15} src(1,0) + \frac{2}{15} src(1,1)$$

Die Größe des Rechenrasters berechnet sich wie folgt:

Rasterbreite = Quellbitmapbreite*Zielbitmapbreite
Rasterhöhe =Quellbitmaphöhe*Zielbitmaphöhe

Die Vergrößerung einer Bitmap findet völlig analog zur Verkleinerung statt. Bild 18 zeigt die Vergrößerung einer 3*2 auf eine 5*7 Bitmap

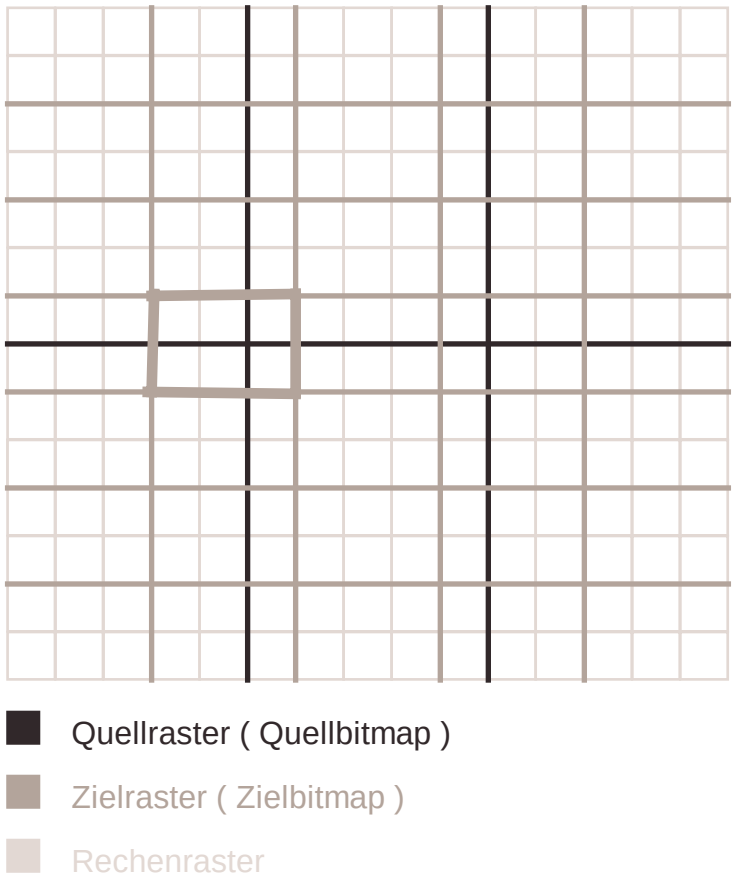


Bild 18 Vergrößerung einer Bitmap

Hier berechnet sich beispielsweise der Wert des Pixels 1,4 (im Bild hervorgehoben) folgendermaßen:

$$dst(1,4) = \frac{2}{6} src(0,0) + \frac{1}{6} src(1,0) + \frac{2}{6} src(0,1) + \frac{1}{6} src(1,1)$$

Durch die Einführung des Rechenrasters in der oben beschrieben Form ergibt sich nun folgender, von mir entwickelter Algorithmus, den ich nun für den interessierten Leser in C-Notation vorstellen möchte:

```
static int norm_bitmap(int src_w,int src_h,unsigned char *src_data, int dst_w,int dst_h,unsigned char *dst_data)
{
        unsigned long   *line_accu;
        unsigned long   col;
        unsigned short  x_lim,y_lim;
        unsigned short  x,y;
        unsigned short  s_xoff,s_yoff,d_xoff,d_yoff;
        unsigned short  s_xinc,s_yinc,d_xinc,d_yinc;
        unsigned short  x_step,y_step;

        x_lim = src_w*dst_w;                                    // X-raster size
        y_lim = src_h*dst_h;                                    // Y-raster size

        if( (line_accu = malloc(dst_w*sizeof(unsigned long)))== NULL)   // Allocate line-accumulator
                return(0);
        memset(line_accu,0,dst_w*sizeof(unsigned long));        // Clear line-accumulator

        s_yoff = 0;
        d_yoff = 0;
        s_yinc = dst_h;
        d_yinc = src_h;
        y=0;

        while(y < y_lim)
        {
                s_xoff = 0;
                d_xoff = 0;
                s_xinc = 0;
                d_xinc = src_w;
                y_step = min(s_yinc,d_yinc);                    // Calc current Y-step size
                x = 0;

                while(x < x_lim)
                {
                        if(s_xinc == 0)                         // Source X-raster hit ?
                        {
                                col    = src_data[s_yoff+s_xoff];   // Take source pixel
                                s_xinc = dst_w;
                                s_xoff++;
                        }
                        if(d_xinc == 0)                         // Destination X-raster hit ?
                        {
                                d_xoff++;
                                d_xinc = src_w;
                        }
                        x_step = min(s_xinc,d_xinc);            // Calculate X-step size
                        line_accu[d_xoff]+=col*x_step*y_step;   // Update line-accumulator
                        s_xinc-=x_step;
                        d_xinc-=x_step;
                        x+=x_step;
                }
                s_yinc-=y_step;
```

```
                d_yinc-=y_step;

                if(s_yinc == 0)                                         // Source Y-raster hit ?
                {
                        s_yoff+=src_w;
                        s_yinc=dst_h;
                }
                if(d_yinc == 0)                                         // Destination Y-raster hit?
                {
                        for(x=0;x<dst_w;x++)                            // Write destination line
                                dst_data[d_yoff+x] = line_accu[x]/(src_w*src_h);
                        memset(line_accu,0,dst_w*sizeof(unsigned long)); // Clear line-accumulator
                        d_yoff+=dst_w;
                        d_yinc=src_h;
                }
                y+=y_step;
        }
        free(line_accu);
        return(1);
}
```

Abschließend möchte ich sagen, daß mir kein anderer Algorithmus mit derselben Funktionalität bekannt ist, der schneller arbeitet. Implementierte Algorithmen in kommerziellen Produkten(Aldus Photostyler[2]) bestätigen wohl meine Aussage.

[2]Ansonsten ein hervorragendes Programm, Anm. des Autors

Modulbeschreibung

Modul OCR_FILT.C

Das Modul OCR_FILT.C führt in einem rechteckigen Bildausschnitt ein Kantenfilterung mit Hilfe zweier Sobelmatritzen durch, und ersetzt den Bildausschnitt durch das Filterergebnis nach der Formel:

Pixel(i,j) = abs(sobelx(i,j))+abs(sobely(i,j)).

Funktion OCR_FILT_SOBELXY

Prototyp:

int ocr_filt_sobelxy(int x1,int y1,int w,int h);

Parameter:

x1: X-Koordinate des zu segmentierenden Bildausschnitts
y1: Y-Koordinate des zu segmentierenden Bildausschnitts
w: Breite des zu segmentierenden Bildausschnitts
h: Höhe des zu segmentierenden Bildausschnitts

Beschreibung:

Die Funktion führt eine Kantenfilterung wie oben näher beschrieben durch. Die Ausführungszeit für die volle Bildgröße (768*512 Pixel) beträgt dank Assembleroptimierungen nur 0.8 Sekundem auf einem 486DX33 Rechner.

Modul OCR_SEG.C

Das Modul OCR_SEG.C segmentiert einen rechteckigen Bildausschnitt im Speicher der PC-Grab und generiert eine zur Weiterverarbeitung geeignete Datenstruktur.

Datentypen

```
typedef struct ocr_seg_char
{
        int                 x1;
        int                 y1;
        int                 w;
        int                 h;
        int                 x_s;
        int                 y_s;
        int                 dir_cnt;
        double              chain_len;
        unsigned char       *dir_list;
        struct ocr_seg_char *next;
}OCR_SEG_CHAR;
```

Funktion OCR_SEG_IMAGE

Prototyp:

OCR_SEG_CHAR * ocr_seg_image(int x1, int y1, int w, int h);

Parameter:

x1: X-Koordinate des zu segmentierenden Bildausschnitts
y1: Y-Koordinate des zu segmentierenden Bildausschnitts
Breite des zu segmentierenden Bildauschnitts
Höhe des zu segmentierenden Bildausschnitts

Beschreibung:

Die Funktion segmentiert den spezifizierten Bereich mit Hilfe eines Konturpunktverfolgungsverfahren. Dazu muß der zu segmentierende Bereich bereits in biniarisierter Form vorliegen (Schwelle Grauwert = 128). Zurückgeliefert wird ein Zeiger auf eine OCR_SEG_CHAR Struktur, welcher die Wurzel der einfach verketteten Liste der segmentierten Zeichen darstellt. Im Fehlerfall wird ein NULL-Zeiger zurückgeliefert.

Funktion OCR_SEG_FREE

Prototyp:

OCR_SEG_FREE(OCR_SEG_CHAR *root);

Parameter:

root: Zeiger auf die Wurzel der einfach verketteten Strukturen OCR SEG_CHAR.

Beschreibung:

Die Funktion gibt den von OCR_SEG_IMAGE belegten Speicher wieder frei.

Modul OCR_PAT.C

Das Modul OCR_PAT.C stellt Funktionen zur Verfügung, um rechteckige Bildausschnitte (Pattern) aus dem Bildspeicher der PC-Grab in eine geeignete interne Struktur umzuwandeln. Da für einige Klassifizierungsverfahren Pattern einheitlicher Größe benötigt werden, existiert zusätzlich eine Funktion zur Normierung der Pattern. Bei der Normierung der Pattern wird ein Interpolationsverfahren angewendet, um Quantiesierungseffekte möglichst zu vermeiden.

Datentypen

```
typedef struct pat
{
        int             w;
        int             h;
        int             size;
        unsigned char   *data;
}PAT;
```

Funktion OCR_PAT_GET

Prototyp:

PAT *ocr_pat_get(int x1, int y1, int w, int h);

Parameter:

x1: X-Koordinate des Patterns
y1: Y-Koordinate des Patterns
w: Breite des Patterns
h: Höhe des Patterns

Beschreibung:

Die Funktion erzeugt eine PAT-Struktur aus dem spezifizierten rechteckigen Bildausschnitt und liefert einen Zeiger auf diese zurück.

Funktion OCR_PAT_CPY

Prototyp:

PAT *ocr_pat_cpy(PAT *pat);

Parameter:

pat: Zeiger auf eine PAT-Struktur

Beschreibung:

Die Funktion erzeugt eine Kopie einer PAT-Struktur und liefert einen Zeiger auf diese zurück.

Funktion OCR_PAT_NRM

Prototyp:

PAT *ocr_pat_nrm(int norm_w, int norm_h, PAT *pat);

Parameter:

norm_w: Breite des normierten Pattern
norm_h: Höhe des normierten Pattern
pat: Zeiger auf das zu normierende Pattern

Beschreibung:

Die Funktion erzeugt eine PAT-Struktur mit durch norm_w und norm_h angegebener Breite und Höhe und liefert einen Zeiger auf sie zurück. Dabei wird ein Interpolationsverfahren angewendet, um Quantisierungseffekte bei der Normierung möglichst auszuschließen.

Funktion OCR_PAT_FREE

Prototyp:

void ocr_pat_free(PAT *pat);

Parameter:

pat: Zeiger auf eine PAT-Struktur

Beschreibung:

Die Funktion gibt den durch die PAT-Struktur belegten Speicher frei.

Funktion OCR_PAT_SAVE

Prototyp:

void ocr_pat_save(FILE *f, PAT *pat);

Parameter:

f: Zeiger auf eine FILE-Struktur
pat: Zeiger auf eine PAT-Struktur

Beschreibung:

Die Funktion speichert eine PAT-Struktur in den durch f spezifizierten Stream.

Funktion OCR_PAT_LOAD

Prototyp:

PAT *ocr_pat_load(FILE *f);

Parameter:

f: Zeiger auf eine FILE-Struktur

Beschreibung:

Die Funktion erzeugt eine PAT-Struktur aus dem durch f spezifizierten Stream und liefert einen Zeiger auf sie zurück.

Modul OCR_DATA.C

Das Modul OCR_DATA.C stellt Funktionen für die Verwaltung von Pattern-Bibliotheken zur Verfügung. Die Pattern in der Bibliothek müssen notwendigerweise nicht auf einheitliche Größe normiert sein, dies hat den Vorteil, dieselbe Bibliothek für unterschiedliche Klassifikationsverfahren verwenden zu können.

Datentypen

```
typedef struct ocr_data_elem
{
        PAT             *pat;
        int             x_s;
        int             y_s;
        int             dir_cnt;
        double          chain_len;
        unsigned char   *dir_list;
        int             ascii_code;
}OCR_DATA_ELEM;

typedef struct ocr_data
{
        int             cnt;
        OCR_DATA_ELEM **elem;
}OCR_DATA;
```

Funktion OCR_DATA_ADD

Prototyp:

```
int ocr_data_add(OCR_DATA *ocr_data,
OCR_SEG_CHAR *ch, int ascii_code);
```

Parameter:

ocr_data:	Zeiger auf eine OCR_DATA-Struktur
ch:	Zeiger auf eine OCR_SEG_CHAR Struktur
ascii_code:	Mit dem Objekt zu assoziierender ASCII-CODE

Beschreibung:

Die Funktion erweitert die Zeichen-Bibliothek ocr_data um ein Zeichen mit dem Wert ascii_code.

Funktion OCR_DATA_FREE

Prototyp:

```
void ocr_data_free(OCR_DATA *ocr_data);
```

Parameter:

ocr_data:	Zeiger auf eine OCR_DATA-Struktur

Beschreibung:

Die Funktion erweitert die Zeichen-Bibliothek ocr_data
um ein Zeichen mit dem Wert ascii_code.

Funktion OCR_DATA_SAVE

Prototyp:

void ocr_data_save(OCR_DATA *ocr_data, char *fname);

Parameter:

ocr_data: Zeiger auf eine OCR_DATA-Struktur
fname: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion sichert die Zeichen-Bibliothek ocr_data in
einer Datei mit dem Namen fname.

Funktion OCR_DATA_LOAD

Prototyp:

OCR_DATA *ocr_data_load(char *fname);

Parameter:

fname: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion lädt eine Zeichen-Bibliothek aus einer Datei mit dem Namen fname und erzeugt eine OCR_DATA Struktur.

Funktion OCR_DATA_NEW

Prototyp:

OCR_DATA *ocr_data_new(void);

Parameter:

Beschreibung:

Die Funktion erzeugt eine neue Zeichen-Bibliothek , indem eine Struktur OCR_DATA angelegt wird. Der Zeiger auf diese Struktur wird bei allen weiteren Zugriffen auf die Zeichen-Bibliothek benötigt

Modul CLAS_KOR.C

Das Modul CLAS_KOR.C stellt Funktionen für ein Klassifizierungsverfahren zur Verfügung, welches auf einem Pattern-Matching Verfahren beruht.

Datentypen

```
typedef struct kor_pre
{

        int     n;              // Anzahl Bildpunkte
        float   si;             // Summe i
        float   si2;            // Summe i^2
        float   n_si2_si;       // n*si2-si*si
        float   *data;
}KOR_PAT;

typedef struct ocr_clas_kor_data
{
        int             cnt;
        int             norm_w;
        int             norm_h;
        int             size;
        KOR_PAT         **kor_pat;
        int             *ascii_tab;
}OCR_CLAS_KOR_DATA;
```

Funktion OCR_CLAS_KOR_INIT

Prototyp:

OCR_CLAS_KOR_INIT *ocr_clas_kor_init(int norm_w, int norm_h, OCR_DATA *ocr_data);

Parameter:

norm_w: Normierungsbreite der Zeichen
norm_h: Normierungshöhe der Zeichen
ocr_data: Zeiger auf eine OCR_DATA-Struktur

Beschreibung:

Die Funktion initialisiert das Klassifizierungsverfahren, in dem sie aus einer Zeichen-Bibliothek ocr_data eine geignete Zwischenstruktur OCR_CLAS_COR_DATA erzeugt. Die Zeichen-Bibliothek wird anschließend nicht weiter benötigt und kann freigegeben werden.

Funktion OCR_CLAS_KOR_FREE

Prototyp:

```
void ocr_clas_kor_free(OCR_CLAS_KOR_DATA
*kor_data);
```

Parameter:

kor_data: Zeiger auf eine OCR_CLAS_KOR_DATA-Struktur

Beschreibung:

Die Funktion gibt den durch das Klassifizierungsverfahren intern benötigten Speicher frei.

Funktion OCR_CLAS_KOR_LOAD

Prototyp:

```
OCR_CLAS_KOR_DATA *ocr_clas_kor_load(char
*fname);
```

Parameter:

f_name: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion lädt eine Zwischenstruktur OCR_CLAS_KOR_DATA aus einer Datei mit dem Namen fname.

Funktion OCR_CLAS_KOR_SAVE

Prototyp:

```
int ocr_clas_kor_save(OCR_CLAS_KOR_DATA
*kor_data, char *fname);
```

Parameter:

kor_data: Zeiger auf eine OCR_CLAS_KOR_DATA-Struktur
f_name: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion sichert die Zwischenstruktur kor_data in einer Datei mit dem Namen fname.

Funktion OCR_CLAS_KOR_TEST

Prototyp:

```
int ocr_clas_kor_test(OCR_CLAS_KOR_DATA
*kor_data, OCR_SEG_CHAR *ch, int *ascii_code, float
*kor);
```

Parameter:

kor_data: Zeiger auf eine OCR_CLAS_KOR_DATA-Struktur
ch: Zeiger auf eine OCR_SEG_CHAR-Struktur

Beschreibung:

Die Funktion führt eine Klassifizierung des Zeichens ch durch, in dem sie in der kor_data-Struktur ein Pattern sucht, welches die größte Übereinstimmung mit dem zu untersuchenden Pattern besitzt. Dazu wird das Zeichen auf das in der KOR_DATA-Struktur festgelegte Maß normiert, und anschließend mit allen gelernten Zeichen verglichen. Zurückgeliefert wird der ASCII-Code des Zeichens und ein Korrelationsergebnis, das bei perfekter Übereinstimmung den Wert 1.0, bei keiner Übereinstimmung den Wert 0.0 erhält.

Modul CLAS_DFT.C

Das Modul CLAS_DFT.C stellt Funktionen für ein Klassifizierungsverfahren zur Verfügung, welches auf einem Fourierdeskriptor Verfahren beruht.

Datentypen

```
typedef struct COMPLEX_POLAR
{
double betrag;
double phase;
}COMPLEX_PLOAR;

typedef struct ocr_clas_dft_data
{
        int                     cnt;
        int                     ld_n;
        COMPLEX_POLAR           **cpl;
        int                     *ascii_tab;
}OCR_CLAS_DFT_DATA;
```

Funktion OCR_CLAS_DFT_INIT

Prototyp:

```
OCR_CLAS_DFT_INIT *ocr_clas_dft_init(int ld_n,
OCR_DATA *ocr_data);
```

Parameter:

ld_n: Zweierlogarithmus der Breite des Fourierspektrums
ocr_data: Zeiger auf eine OCR_DATA-Struktur

Beschreibung:

Die Funktion initialisiert das Klassifizierungsverfahren, in dem sie aus einer Zeichen-Bibliothek ocr_data eine geeignete Zwischenstruktur OCR_CLAS_DFT_DATA erzeugt. Die Zeichen-Bibliothek wird anschließend nicht weiter benötigt und kann freigegeben werden.

Funktion OCR_CLAS_DFT_FREE

Prototyp:

void ocr_clas_dft_free(OCR_CLAS_DFT_DATA *kor_data);

Parameter:

kor_data: Zeiger auf eine OCR_CLAS_DFT_DATA-Struktur

Beschreibung:

Die Funktion gibt den durch das Klassifizierungsverfahren intern benötigten Speicher frei.

Funktion OCR_CLAS_DFT_LOAD

Prototyp:

OCR_CLAS_DFT_DATA *ocr_clas_dft_load(char *fname);

Parameter:

f_name: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion lädt eine Zwischenstruktur OCR_CLAS_DFT_DATA aus einer Datei mit dem Namen fname.

Funktion OCR_CLAS_DFT_SAVE

Prototyp:

int ocr_clas_dft_save(OCR_CLAS_DFT_DATA *dft_data, char *fname);

Parameter:

kor_data: Zeiger auf eine OCR_CLAS_DFT_DATA-Struktur
f_name: Zeiger auf einen Dateinamen

Beschreibung:

Die Funktion sichert die Zwischenstruktur dft_data in einer Datei mit dem Namen fname.

Funktion OCR_CLAS_DFT_TEST

Prototyp:

int ocr_clas_dft_test(OCR_CLAS_DFT_DATA *kor_data, OCR_SEG:CHAR *ch, int *ascii_code, float *kor, float *rot);

Parameter:

dft_data: Zeiger auf eine OCR_CLAS_DFT_DATA-Struktur
ch: Zeiger auf eine OCR_SEG_CHAR-Struktur

Beschreibung:

Die Funktion vergleicht das Fourierspektrum des Zeichens ch mit den Spektren, die in der dft_data Struktur gespeichert sind. Dazu werden nur die ersten N/2 Koeffizienten zum Vergleich herangezogen, da die zweite Hälfte des Spektrums aufgrund von Spiegelfrequenzen für die Klassifizierung nicht relevant ist. Zurückgeliefert wird der ASCII-Code des Zeichens und ein Korrelationsergebnis, das bei perfekter Übereinstimmung den Wert 1.0, bei keiner Übereinstimmung den Wert 0.0 erhält. In rot wird die Verdrehung des Zeichens zum erkannten Zeichen in der Bibliothek in Rad zurückgeliefert.

Modul OCR_DEV.C

Dieses Modul stellt eine Schnittstelle zur eingesetzten Framegrabber-Hardware dar, d.h. alle anderer Module benutzen, wenn sie den Framegrabber ansprechen wollen, folgende Funktionen:

Funktion OCR_DEV_LOCK_ROW

Prototyp:

void far *ocr_dev_lock_row(int y);

Parameter:

y: Zeilenkoordinate

Beschreibung:

Die Funktion liefert einen Zeiger auf die Framegrabberzeile y und verriegelt diese.

Funktion OCR_DEV_UNLOCK_ROW

Prototyp:

void *ocr_dev_unlock_row(int y, void far *ptr);

Parameter:

y: Zeilenkoordinate

Beschreibung:

Die Funktion entriegelt die Zeile y.

Funktion OCR_DEV_GET_ROW

Prototyp:

void *ocr_dev_get_row(int x1, int y1, int x2, unsigned char *data);

Parameter:

x1: Spaltenkoordinate Start
y1: Zeilenkoordinate
x2: Spaltenkoordinate Ende

Beschreibung:

Die Funktion kopiert die angegebene Zeile in das Array data.

Funktion OCR_DEV_SET_ROW

Prototyp:

void *ocr_dev_set_row(int x1, int y1, int x2, unsigned char *data);

Parameter:

x1: Spaltenkoordinate Start
y1: Zeilenkoordinate
x2: Spaltenkoordinate Ende
data: Zeiger auf ein Array

Beschreibung:

Die Funktion kopiert das angegebene Array in die Zeile y1.

Funktion OCR_DEV_GET_COL

Prototyp:

void *ocr_dev_get_col(int x1, int y1, int y2, unsigned char *data);

Parameter:

x1: Spaltenkoordinate
y1: Zeilenkoordinate Start
x2: Zeilenkoordinate Ende

Beschreibung:

Die Funktion Kopiert die angegebene Spalte in das Array data.

Funktion OCR_DEV_SET_COL

Prototyp:

void *ocr_dev_set_col(int x1, int y1, int y2, unsigned char *data);

Parameter:

x1: Spaltenkoordinate
y1: Zeilenkoordinate Start
y2: Zeilenkoordinate Ende
data: Zeiger auf ein Array

Beschreibung:

Die Funktion kopiert das angegebene Array in die Spalte x1.

Modul HOUGH.C

Dieses Modul wird benötigt, um eine Aussage zu treffen, ob die Vorlage rotiert abgetastet wurde und wenn ja, welche Größe der Rotationswinkel besitzt. Hierzu wird die sogenannte Hough-Transformation eingesetzt.
Formel:
p(phi) = x*cos(phi)+y*sin(phi) und
H(p,phi) = H(p,phi)+1

Funktion HOUGH_CALC_ANGLE

Prototyp:

double hough_calc_angle(int x1,int y1,int w,int h,OCR_SEG_CHAR *ch)

Parameter:

x1: X-Koordinate des zu transformierenden Bildausschnitts
y1: Y-Koordinate des zu transformierenden Bildausschnitts
w: Breite des zu transformierenden Bildausschnitts
h: Höhe des zu transformierenden Bildausschnitts
ch: Zeiger auf eine OCR_SEG_CHAR Struktur

Beschreibung:

Die Funktion transformiert die Koordinaten der segmentierten Objekte mittels der Hough-Transformation. Anschließend wird im Transformationsraum ein Häufungsmaximum bestimmt, dessen Winkelkoordinate den Rotationswinkel der Vorlage darstellt. zurückgelieftert wird der Rotationswinkel in Rad (-PI/2 <= angle <= PI/2).

Modul OCR_ORDR.C

Diese Modul ordnetet die segmentierten Objekte neu an, um eine korrekte Lesereihenfolge zu gewährleisten (Von Links nach Rechts, von oben nach unten). Die alte Reihenfolge der segmentierten Objekte geht dadruch allerdings verloren.

Funktion OCR_ORDR_SORTXY

Prototyp:
OCR_SEG_CHAR *ocr_ordr_sortxy(int x1,int y1,int w,int h,double rot_angle,OCR_SEG_CHAR *root)

Parameter:

x1: X-Koordinate des zu transformierenden Bildausschnitts
y1: Y-Koordinate des zu transformierenden Bildausschnitts
w: Breite des zu transformierenden Bildausschnitts
h: Höhe des zu transformierenden Bildausschnitts
rot_angle: Rotationswinkel der Vorlage
root: Zeiger auf eine OCR_SEG_CHAR Struktur

Beschreibung:

Die Funktion ordnet die segmentierten Objekte bezüglich ihrer X- und Y-Koordinaten neu an, so daß sich einen korrekte Lesereihenfolge ergibt. Zurückgeliefert wird ein Zeiger auf eine OCR_SEG_CHAR Struktur, welcher die Wurzel der geordenten Objekte darstellt.