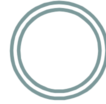
Bildverarbeitung



Inhalt:

- RGB-Farbmodell
- Fraktal: Apfelmännchen
- Komplexe Zahlen
- BufferedImage
- Invertieren
- Farbraumkonvertierungen
- Binarisieren

Block M.: "*Java-Intensivkurs - In 14 Tagen lernen Projekte erfolgreich zu realisieren*" , Springer-Verlag 2007
Burger W. ,Burge M.J.: "*Digitale Bildverarbeitung*", 2.Auflage, Springer-Verlag 2006

RGB-Farbmodell I

Das menschliche Auge kann hauptsächlich Licht von drei verschiedenen Wellenlängen wahrnehmen. Diese Wellenlängen werden vom Auge als die Farben Rot, Grün und Blau wahrgenommen. Die Mischung der Intensitäten dieser drei Wellenlängen ermöglicht es, viele verschiedene Farben zu sehen.

Der Bau von Monitoren wurde dadurch relativ einfach, da man, um eine bestimmte Farbe zu erzeugen, nur Licht mit drei verschiedenen Wellenlängen zu erzeugen braucht. Die exakte Mischung macht dann eine spezifische Farbe aus.

Dieses Farbmodell wird RGB-Modell genannt, da es auf den drei Grundfarben Rot, Grün und Blau basiert. **Schwarz** erhält man, wenn keine der drei Farbkanäle Licht liefert. **Weiß** durch die gleichzeitige, maximale Aktivierung aller drei Grundfarben. **Gelb** zum Beispiel wird durch eine Mischung aus Rot und Grün erzeugt. Man spricht hierbei von additiver Farbmischung, da die wahrgenommene Farbe immer heller wird, je mehr Grundfarben man hinzunimmt. Die Farbmischung eines Tuschkastens dagegen wird als subtraktiv bezeichnet. Vermischt man - wie Kinder es gerne machen - viele Farben miteinander, erhält man etwas sehr dunkles, meist ein hässliches Braun.

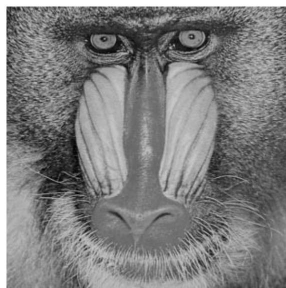
Bildverarbeitung

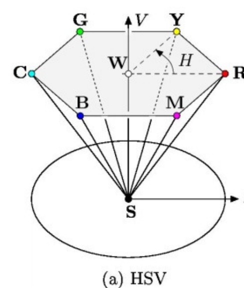
Freie Universität Berlin

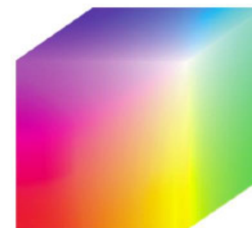
Farbräume

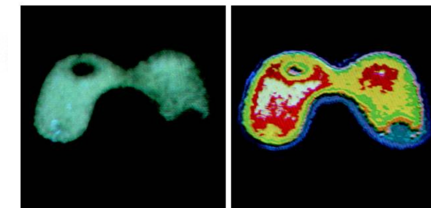
Es gibt weitere wichtige Farbräume, wie z.B.:

- Binäre Bilder
- Grauwertbilder
- HSI/HSV (Hue, Saturation, Value/Brightness/Intensity)
- CMY (cyan, magenta, yellow)
- CMYK
- Colorimetrische Farbräume (kalibriertes Farbsystem)
- Pseudofarben
- ...









a b

FIGURE 6.20 (a) Monochrome image of the Picker Thyroid Phantom. (b) Result of density slicing into eight colors. (Courtesy of Dr. J. L. Blankenship, Instrumentation and Controls Division, Oak Ridge National Laboratory.)

RGB-Farbmodell II

Die Funktion *setColor* des **Graphics**-Objekts setzt die Farbe, die zum Zeichnen durch zukünftige Befehle benutzt werden soll. Der folgende Code erzeugt ein Fenster mit 5 farbigen Rechtecken.

```
import java.awt.*;

public class Bunt extends FensterSchliesstSchickKurz{
   public Bunt(String title1, int w, int h){
       super(title1, w, h);
   }
   public void paint(Graphics g){
       g.setColor(Color.RED);
       g.fillRect(1,1,95,100);

       g.setColor(new Color(255,0,0)); // Rot
       g.fillRect(101,1,95,100);

       g.setColor(new Color(0,255,0)); // Grün
       g.fillRect(201,1,95,100);

       g.setColor(new Color(0,0,255)); // Blau
       g.fillRect(301,1,95,100);

       g.setColor(new Color(255,255,0)); // Rot + Grün = ?
       g.fillRect(401,1,95,100);

       g.setColor(new Color(100,100,100)); // Rot + Grün + Blau = ?
       g.fillRect(501,1,95,100);
   }

   public static void main(String[] args){
       Bunt b = new Bunt ("Farbige Rechtecke", 500, 100);
   }
}
```

RGB-Farbmodell III

Jetzt sind wir in der Lage einen kontinuierlichen Farbverlauf zu erzeugen:

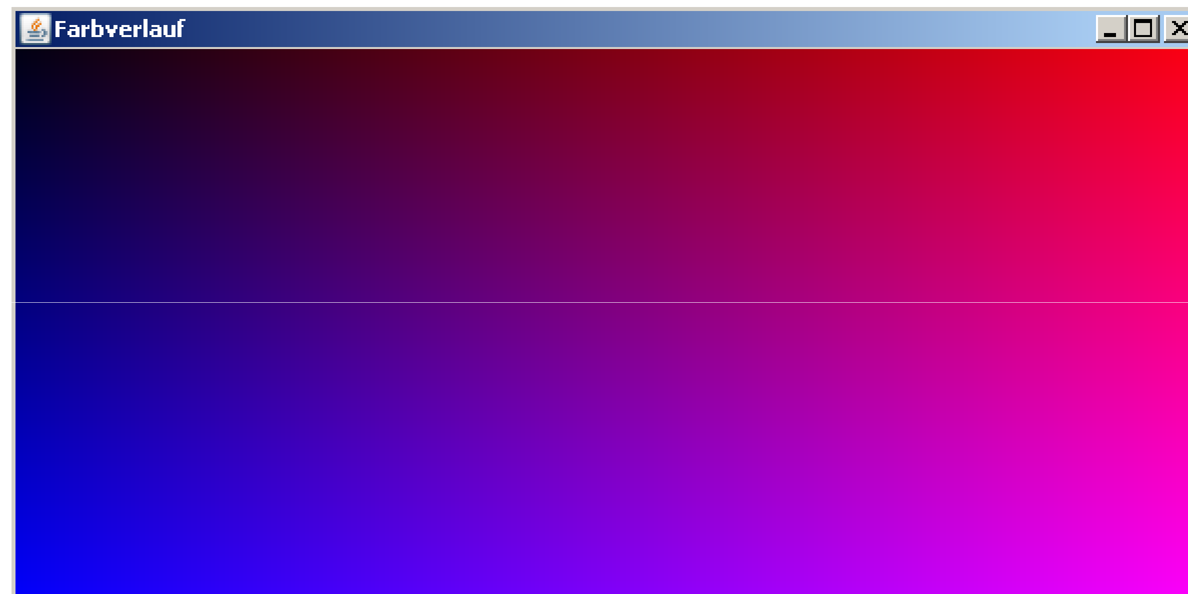
```
import java.awt.*;

public class Farbverlauf extends FensterSchliesstSchickKurz{
   static int fensterBreite = 600;
   static int fensterHoehe  = 300;

   public Farbverlauf(String title1, int w, int h){
       super(title1, w, h);
   }

   public void paint(Graphics g){
      for(double x=1; x<fensterBreite ; x++){
         for(double y=1; y<fensterHoehe  ; y++){
            int rot   = (int) Math.floor( 255*x/fensterBreite );
            int blau  = (int) Math.floor( 255*y/fensterHoehe  );
            g.setColor(new Color(rot,0,blau));
            g.fillRect(x,y,1,1);
         }
      }
   }
   public static void main(String[] args){
      Farbverlauf b = new Farbverlauf ("Farbverlauf", fensterBreite ,
                                       fensterHoehe);
      b.setVisible(true);
   }
}
```

Bildverarbeitung

Freie Universität Berlin

RGB-Farbmodell IV

Das führt zu folgender Ausgabe:



Apfelmännchen:

Machen wir das Ganze noch etwas interessanter und erzeugen geometrische Muster, die eine hohe Selbstähnlichkeit aufweisen. Das bedeutet, dass ein Muster aus mehreren verkleinerten Kopien seiner selbst besteht.

Dazu ist allerdings ein kleiner Ausflug in die Mathematik notwendig. Im Speziellen wird uns eine kleine Einführung in die Komplexen Zahlen helfen, Fraktale zu verstehen und realisieren zu können.

Komplexe Zahlen I:

Eine Komplexe Zahl besteht also aus einem Paar reeller Zahlen. Der Teil einer Komplexen Zahl der auf der x-Achse abgetragen wird, ist der Realteil, der Wert auf der y-Achse wird Imaginärteil genannt. Dieser wird durch ein 'i' markiert. Ein Beispiel für eine Komplexe Zahl ist: $1+1i$.

Mit Komplexen Zahlen können wir auch rechnen. Nehmen wir zum Beispiel die Addition

$$(-2+3i)+(1-2i),$$

so ist das Ergebnis mit einem Zwischenschritt einfach

$$(-2+3i)+(1-2i)=-2+1+(3-2)i=-1+1i.$$

Komplexe Zahlen II:

Etwas komplizierter ist die Multiplikation. Das Produkt $A*B$ der beiden Komplexen Zahlen $A=a_1+a_2 i$ und $B=b_1+b_2 i$ ist definiert als

$$a_1*b_1-a_2*b_2+(a_1*b_2+a_2*b_1)i.$$

Eine Besonderheit sei noch angemerkt, denn es gilt:

$$(0+1i)*(0+1i)=1i*1i=i^2=-1.$$

Komplexe Zahlen III:

Die Objektorientierung von Java bietet eine sehr einfache Möglichkeit das Rechnen mittels Komplexer Zahlen zu implementieren. Sehen wir uns ein Beispiel an:

```
class KomplexeZahl{
   double re;      // Speichert den Realteil
   double im;      // Speichert den Imaginärteil

   public KomplexeZahl(double r, double i){  // Konstruktor
      re = r;
      im = i;
   }

   public KomplexeZahl plus(KomplexeZahl k){
      return new KomplexeZahl(re + k.re,im + k.im);
   }

   public KomplexeZahl mal(KomplexeZahl k){
      return new KomplexeZahl(re*k.re - im*k.im, re*k.im + im*k.re);
   }

   public double norm(){  // Berechnet den Abstand der Zahl vom Ursprung
      return Math.sqrt(re*re + im*im);
   }

   public String text(){  // Gibt die Zahl als Text aus
      return re+" + "+im+" i";
   }
}
```

Komplexe Zahlen IV:

Die Rechnung mit Komplexen Zahlen erweist sich jetzt als genauso einfach, wie die Rechnung mit natürlichen Zahlen:

```
KomplexeZahl zahl1 = new KomplexeZahl(1, 1);
KomplexeZahl zahl2 = new KomplexeZahl(2, -2);
System.out.println("Ergebnis: "+(zahl1.plus(zahl2)).text());
```

Liefert:

```
C:\Java>java KomplexeZahl
Ergebnis: 3.0 + -1.0 i
```

Apfelmännchen I:

Wir können jetzt mit Komplexen Zahlen rechnen, aber was hat das mit Bildern zu tun? Unsere Motivation ist der Einstieg in die Fraktale.

Komplexe Zahlen eignen sich zur Berechnung von Bildern, da jede Zahl direkt einem Punkt auf einer 2-dimensionalen Fläche zugeordnet werden kann. Berechnen wir also für eine Fläche von Komplexen Zahlen jeweils für jede dieser Zahlen einen Funktionswert, so füllen wir Stück für Stück eine besagte Fläche mit Werten. Jetzt müssen wir jedem Funktionswert nur noch eine Farbe zuordnen und fertig ist das Meisterwerk.

Betrachten wir zunächst folgendes Programm und seine Ausgabe.

```
import java.awt.*;

public class Fraktal extends FensterSchliesstSchickKurz{
   static int resolution = 500;

   public Fraktal (String title1, int w, int h){
      super(title1, w, h);
   }

   public int berechnePunkt(double x, double y){
      KomplexeZahl c = new KomplexeZahl(x,y);
      KomplexeZahl z = new KomplexeZahl(0,0);
      for (double iter = 0;iter < 40; iter ++){
         z = (z.mal(z)).plus(c);
         if(z.norm() > 4) break;
      }
      return (int)Math.floor((255)*iter/40);
   }
   ...
```

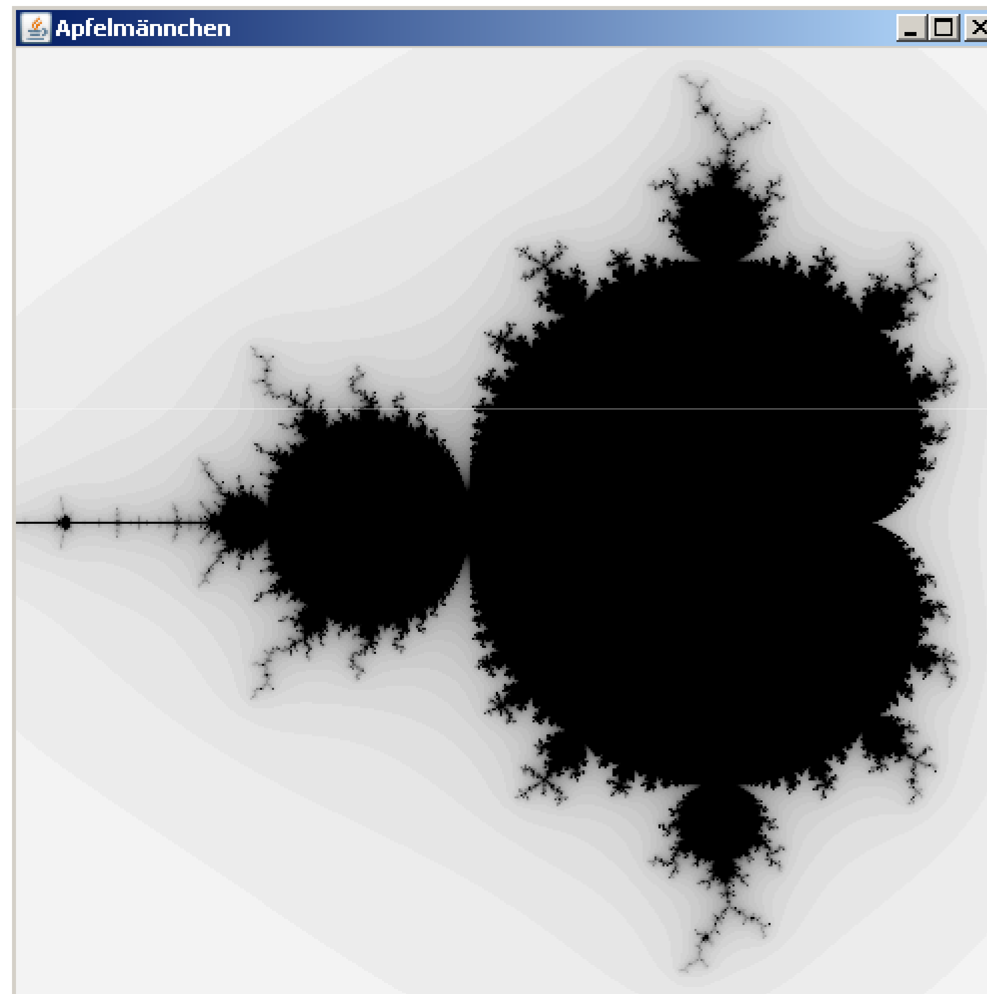
Apfelmännchen II:

weiter geht's:

```
   ...
   public void berechneBild(Graphics g){
      for (double x=0; x<aufloesung;x++){
         for (double y=0; y<aufloesung; y++){
            int Fxy = 255 - berechnePunkt(2.5*x/aufloesung - 1.9, 2.5*y/aufloesung - 1.3);
            g.setColor(new Color(Fxy, Fxy, Fxy));
            g.fillRect((int)x, (int)y, 1, 1);
         }
      }
   }

   public void paint(Graphics g){
      berechneBild(g);
   }

   public static void main(String[] args){
      Fraktal f = new Fraktal ("Apfelmännchen", aufloesung, aufloesung);
      f.setVisible(true);
   }
}
```

Apfelmännchen III:

Unser Programm erzeugt ein Fraktal mit dem berühmten Namen Apfelmännchen und das sieht wie folgt aus:



Apfelmännchen IV:

Unter Verwendung der Klasse BufferedImage wollen wir mehr Farbe ins Spiel bringen:

```
public class FraktalBuntFinal extends FensterSchliesstSchickKurz{
   static int aufloesung     = 100;
   static int fensterRandLRU = 5;   // Fensterrand links, rechts, unten
   static int fensterRandO   = 31;  // Fensterrand oben
   static int itermax        = 2000;// Maximale Anzahl von Iterationen
   static int schwellenwert  = 35;  // bis zum Erreichen des Schwellwerts.
   static int[][] farben     = {
              {   1,  255,255,255}, // Hohe Iterationszahlen sollen hell,
              { 300,   10, 10, 40}, // die etwas niedrigeren dunkel,
              { 500,  205, 60, 40}, // die "Spiralen" rot
              { 850,  120,140,255}, // und die "Arme" hellblau werden.
              {1000,   50, 30,255}, // Innen kommt ein dunkleres Blau,
              {1100,    0,255,  0}, // dann grelles Grün
              {1997,   20, 70, 20}, // und ein dunkleres Grün.
              {itermax, 0,  0,  0}};// Der Apfelmann wird schwarz.

   static double bildBreite  = 0.000003628;
   // Der Ausschnitt wird auf 3:4 verzerrt
   static double bildHoehe   = bildBreite*3.f/4.f;
   // Die Position in der Komplexen-Zahlen-Ebene
   static double [] bildPos  = {-0.743643135-(2*bildBreite/2),
                                 0.131825963-(2*bildBreite*3.f/8.f)};

   BufferedImage bild;
   ...
```

Bildverarbeitung

Freie Universität Berlin

Apfelmännchen V:

weiter geht's:

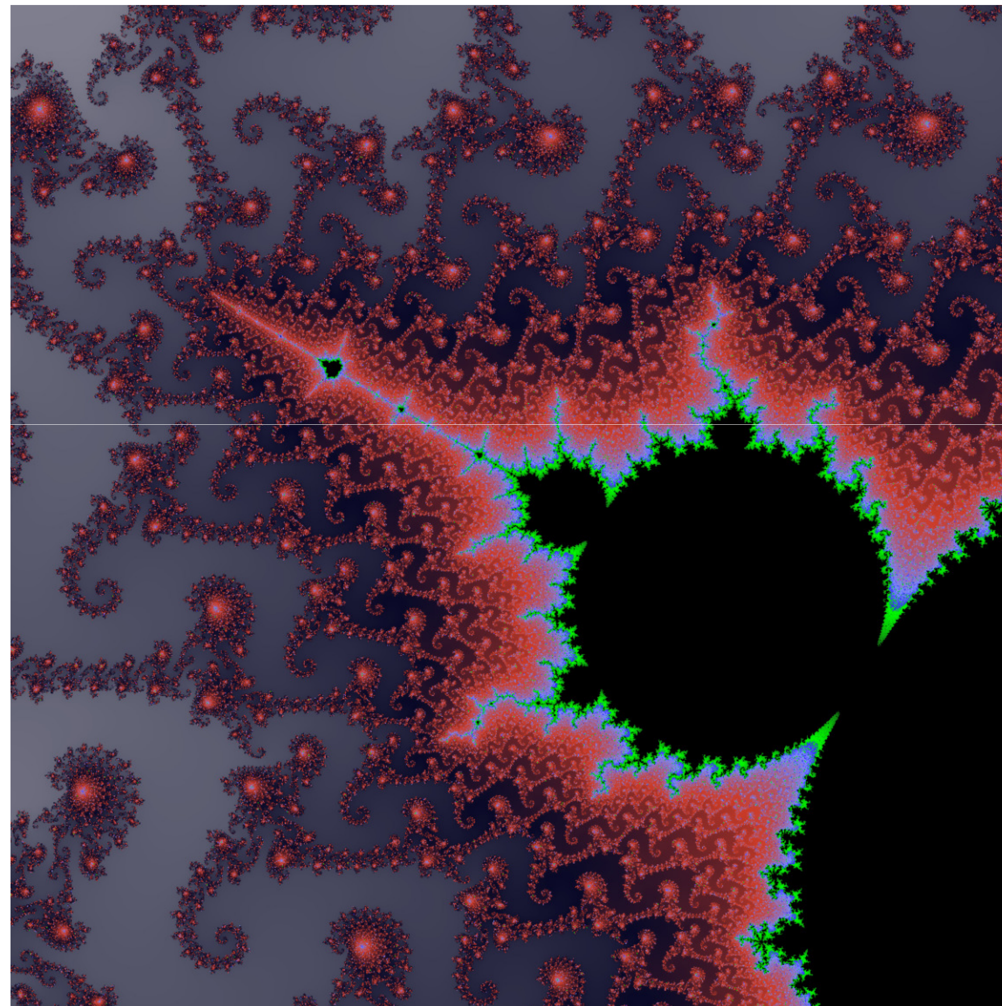
```
...
public FraktalBuntFinal (String title1, int w, int h){
   super(title1, w, h);
   bild = new BufferedImage(aufloesung, aufloesung,BufferedImage.TYPE_INT_RGB);
   Graphics gimg = bild.createGraphics();

   berechneBild(gimg);
      try {
         ImageIO.write(bild, "BMP", new File("ApfelmannBunt.bmp"));
      } catch(IOException e){
         System.out.println("Fehler beim Speichern!");
   }
}

public Color berechneFarbe(int iter){
   int F[] = new int[3];
   for (int i=1; i<farben.length-1; i++){
      if (iter < farben[i][0]){
         int iterationsInterval = farben[i-1][0]-farben[i][0];
         double gewichtetesMittel = (iter-farben[i][0])/(double)iterationsInterval;
         for (int f=0; f<3; f++){
            int farbInterval = farben[i-1][f+1]-farben[i][f+1];
            F[f] = (int)(gewichtetesMittel*farbInterval)
                         +farben[i][f+1];
         }
         return new Color(F[0], F[1], F[2]);
      }
   }
   return Color.BLACK;
}
...
```

Apfelmännchen VI:

```
   public int berechnePunkt(KomplexeZahl c){
      KomplexeZahl z = new KomplexeZahl(0,0);
      int iter = 0;
      for (; (iter <= itermax) && (z.norm() < schwellenwert); iter ++)
         z = (z.mal(z)).plus(c);
      return iter;
   }

   public void berechneBild(Graphics g){
      for (int x=0; x<aufloesung;x++){
         for (int y=0; y<aufloesung; y++){
            KomplexeZahl c = new KomplexeZahl(bildBreite*(double)(x)/aufloesung + bildPos[0],
                                              bildBreite*(double)(y)/aufloesung + bildPos[1]);
            g.setColor(berechneFarbe(berechnePunkt(c)));
            g.fillRect(x, y, 1, 1);
         }
      }
   }

   public void paint(Graphics g){
      g.drawImage(bild, fensterRandLRU, fensterRandO, this);
   }

   public void update(Graphics g){
      paint(g);
   }

   public static void main(String[] args){
      FraktalBuntFinal f = new FraktalBuntFinal("Apfelmännchen - Bunt", aufloesung+2*fensterRandLRU,
                              aufloesung+fensterRandLRU+fensterRandO);
      f.setVisible(true);
   }
}
```

Apfelmännchen VII:

Wir erhalten folgende Ausgabe:



Bilder invertieren I:

Als erstes wollen wir ein Bild invertieren. Dies bedeutet jeden Bildpixel mit seinem farblichen Gegenspieler einzufärben. Benutzen wir die gewohnte RGB-Darstellung, kann eine Invertierung leicht durchgeführt werden. Ist der Helligkeitswert einer der Grundfarben x, so ist sein invertierter Helligkeitswert 255-x. Aus 0 wird also 255, aus 255 wird 0. 127 wird zu 128, ein mittelhelles Grau bleibt demnach fast unverändert.

Die Sache wird ein wenig komplizierter, da die drei Farbkanäle gemeinsam in einem Integer gespeichert werden. Dabei geht man wie folgt vor:

Sind r, g und b die Helligkeitswerte der drei Grundfarben, jeweils im Bereich von 0 bis 255, dann ergibt sich der gemeinsame RGB-Wert aus:

rgb=256*256*r+256*g+b.

Das Schöne an dieser Art der Darstellung ist, dass alle drei Farben speicherplatzgünstig in einem einzigen int gespeichert und aus diesem wieder eindeutig rekonstruiert werden können. Ein int in Java verfügt über 8 Byte. Hier sind die ersten beiden Bytes für den Alpha-Wert (Transparenz) der Zahl reserviert, der in der reinen RGB-Darstellung keine Rolle spielt. Die restlichen 3 Byte kodieren jeweils einen Farbkanal.

Freie Universität Berlin

Bilder invertieren II:

Um an den konkreten Wert nur eines Bytes aus dem int zu kommen, setzten wir die anderen auf 0 und verschieben das entsprechende Byte an die niederwertigste Stelle. Dazu benutzen wir das bitweise UND, dass die UND-Funktion elementweise auf zwei Bitworte anwendet. Die einzelnen Farbkanäle ließen sich dann durch

```
int rot   = (farbe & 256*256*255)/(256*256);
int gruen = (farbe & 256*255)/256;
int blau  = (farbe & 255);
```

extrahieren.

Bilder invertieren III:

Jetzt können wir eine Funktion zum Invertieren eines RGB-Bildes implementieren:

```
public BufferedImage invertiere(BufferedImage b){
   int x = b.getWidth();
   int y = b.getHeight();
   BufferedImage ib = new BufferedImage(x,y,BufferedImage.TYPE_INT_RGB);
   for (int i=0; i<x; i++){
      for (int k=0; k<y; k++){
         // 255 + 256*255 + 256*256*255
         int neu = 255 + 256*255 + 256*256*255 - b.getRGB(i,k);
         ib.setRGB(i,k,neu);
      }
   }
   return ib;
}
```

Bilder invertieren III:

Das könnte folgende Ausgabe liefern:

Farbbilder zu Grauwertbilder konvertieren I:

Das gleiche Prinzip können wir nutzen, um aus einem Farbbild ein Graubild zu erstellen. Laufen wir wieder über jeden Bildpunkt, berechnen einen durchschnittlichen Helligkeitswert aus den drei Kanälen und schreiben diesen in die drei Kanäle des Pixels im neuen Bild. Dazu ist es allerdings nötig, zuerst den Farbwert jedes einzelnen Kanals zu bestimmen.

Das lässt sich realisieren, indem wir die beiden Zeilen innerhalb der for-Schleife von *invertiere* durch diese ersetzen:

```
int alt      = b.getRGB(i,k);
int rot      = (alt & 256*256*255)/(256*256);
int gruen    = (alt & 256*255)/256;
int blau     = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
int neu      = 256*256*grauwert + 256*grauwert + grauwert;
ib.setRGB(i,k,neu);
```

Eine Besonderheit ist hierbei, dass wir anstelle des einfachen arithmetischen Mittelwerts der Helligkeiten ein gewichteten Mittelwert berechnen. Dies ist durch die Biologie des menschlichen Auges motiviert. Grüne Farbanteile im Licht werden stärker wahrgenommen als rote und diese wiederum stärker als blaue.

Farbbilder zu Grauwertbilder konvertieren II:

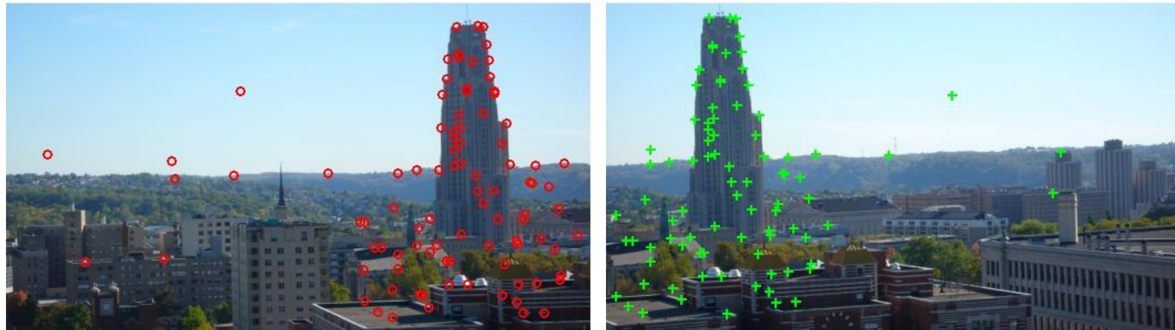
Liefert für das vorherige Beispiel folgende Ausgabe:

Farbbilder zu Grauwertbilder konvertieren III:

Eine Anwendung ist beispielsweise die Objektwiedererkennung. Die meisten Merkmalsdetektoren arbeiten auf Grauwertbildern:

Farbbilder zu Grauwertbilder konvertieren IV:

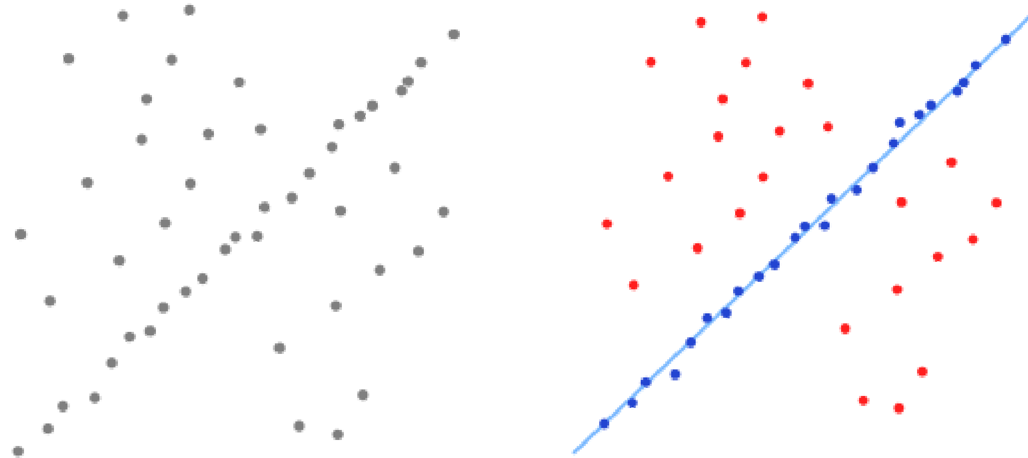
Verschiedene Aufnahmen



zusammenstitchen (Mosaiking):

Mosaiking I:

Ransac-Algorithmus:



1) Wähle zufällig so viele Punkte aus den Datenpunkten wie nötig sind, um die Parameter des Modells zu berechnen. Das geschieht in Erwartung, dass diese Menge frei von Ausreißern ist.

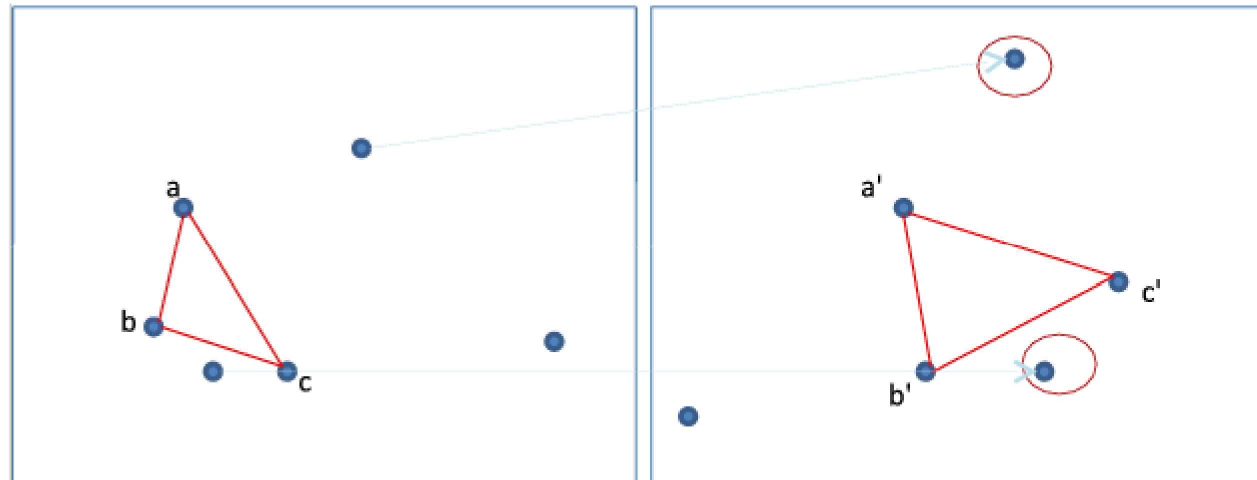
2) Ermittle mit den gewählten Punkten die Modellparameter.

3) Bestimme die Teilmenge der Messwerte, deren Abstand zur Modellkurve kleiner als ein bestimmter Grenzwert ist (diese Teilmenge wird Consensus set genannt). Alle Punkte, die einen größeren Abstand haben, werden als grobe Fehler angesehen. Enthält die Teilmenge eine gewisse Mindestanzahl an Werten, wurde vermutlich ein gutes Modell gefunden und der Consensus set wird gespeichert.

4) Wiederhole die Schritte 1–3 mehrmals.

Mosaiking II:

Beispiel:



Least Squares mit Heuristik versus RANSAC I:

Least Squares (Methode der kleinsten Quadrate) mit der Heuristik, dass der schlechteste Punkt aus der Menge entfernt wird, ist ein Standardverfahren, um derartige Probleme zu lösen . Für ein gegebenes Modell wird die Summe der quadratischen Abweichungen minimiert. Zunächst wird angenommen, dass alle Punkte zum vorliegenden Modell gehören und für jeden Punkt die quadratische Abweichung berechnet.

Ziel ist eine Gerade in der Form: f(x)=a*x+b. Für n Datenpunkte {(x1, y1),(x2, y2),...,(xn, yn)} kann die Gerade mit Hilfe des Verschiebungssatzes angegeben werden

$$a = \frac{\sum_{i=1}^{n}(y_i-\mu_y)(x_i-\mu_x)}{\sum_{i=1}^{n}(x_i-\mu_x)^2} \text{ und } b = \mu_y - a \cdot \mu_x$$

mit

$$\mu_x = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ und } \mu_y = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

Least Squares mit Heuristik versus RANSAC II:

Zwei Punkte, die diese Gerade beschreiben sind p1=(0,b) und p2=(1,a*b). Jetzt lässt sich die Distanz für alle Punkte zu dieser Geraden berechnen, z.B. für einen Punkt p0=(x0, y0) mit

$$dist = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}.$$

Sollten Punkte ausserhalb der erlaubten Toleranz t liegen, so wird der am entferntest liegende gelöscht und für die verbleibenden Punkte ein neues Geradenmodell berechnet. Das wird solange wiederholt, bis alle Punkte innerhalb einer Toleranz liegen.

Beispielcode findet sich hier:

http://www.java-uni.de/forum/viewtopic.php?f=9&t=227

Binarisierung I:

Den Grauwert kann man auch verwenden, um das Bild zu binarisieren. Dies bedeutet, man entscheidet für jeden Pixel, ob er entweder schwarz oder weiß sein soll, abhängig von der Intensität des Grauwerts.

```
int alt = b.getRGB(i,k);
int rot   = (alt & 256*256*255)/(256*256);
int gruen = (alt & 256*255)/256;
int blau  = (alt & 255);
int grauwert = (int) Math.floor(0.299*rot + 0.587*gruen + 0.114*blau);
if (grauwert > 125)
   ib.setRGB(i,k,256*256*255 + 256*255 + 255);
else
   ib.setRGB(i,k,0);
```

Unsere Binarisierungsmethode setzt Pixel mit einem Grauwert *grau* größer als 125 auf Weiss und alle anderen auf Schwarz. Da alle Bildpunkte mit dem gleichen Schwellwert binarisiert werden, nennt man dieses Verfahren auch globale Binarisierung.

Binarisierung II:

Unser Beispiel binarisiert:

Binarisierung III:

Bessere Verfahren ermitteln die Schwellwerte für kleinere Flächen, diese bezeichnet man als lokale Binarisierung.

mein aktuelles Verfahren ☺

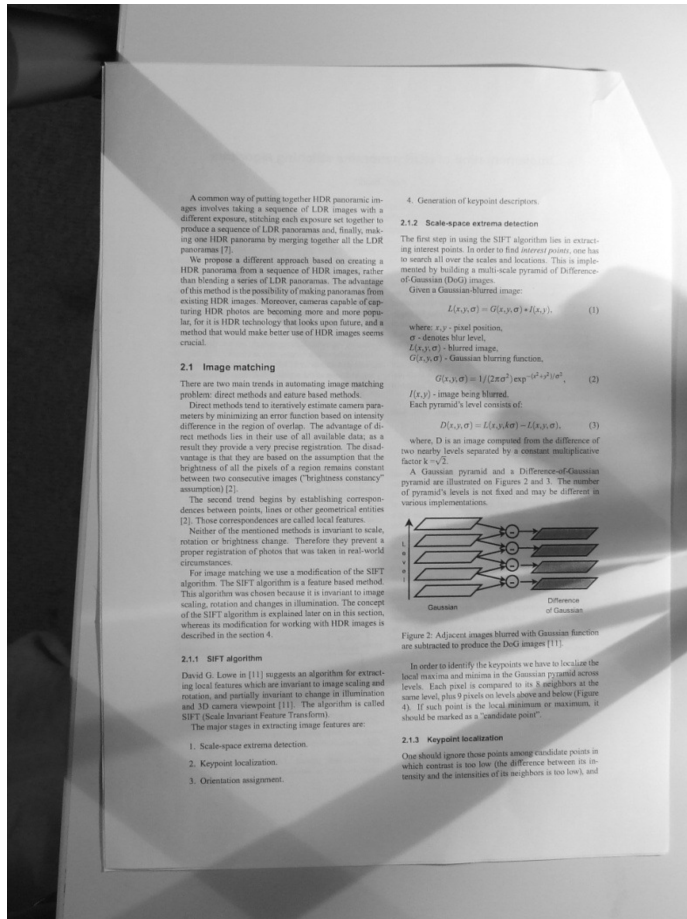
Drei Standardverfahren (von links nach rechts: Kavallieratou, Niblack, YunLi)

Bildverarbeitung

Freie Universität Berlin

Binarisierung IV:

Binarisierung von Text zur Eliminierung von Schatten und Identifikation von Text:



A common way of putting together HDR panoramic images involves taking a sequence of LDR images with a different exposure, stitching each exposure set together to produce a sequence of LDR panoramas and, finally, making one HDR panorama by merging together all the LDR panoramas [7].

We propose a different approach based on creating a HDR panorama from a sequence of HDR images, rather than blending a series of LDR panoramas. The advantage of this method is the possibility of making panoramas from existing HDR images. Moreover, cameras capable of capturing HDR photos are becoming more and more popular, for it is HDR technology that looks upon future, and a method that would make better use of HDR images seems crucial.

2.1 Image matching

There are two main trends in automating image matching problem: direct methods and eature based methods.

Direct methods tend to iteratively estimate camera parameters by minimizing an error function based on intensity difference in the region of overlap. The advantage of direct methods lies in their use of all available data; as a result they provide a very precise registration. The disadvantage is that they are based on the assumption that the brightness of all the pixels of a region remains constant between two consecutive images ("brightness constancy" assumption) [2].

The second trend begins by establishing correspondences between points, lines or other geometrical entities [2]. Those correspondences are called local features.

Neither of the mentioned methods is invariant to scale, rotation or brightness change. Therefore they prevent a proper registration of photos that was taken in real-world circumstances.

For image matching we use a modification of the SIFT algorithm. The SIFT algorithm is a feature based method. This algorithm was chosen because it is invariant to image scaling, rotation and changes in illumination. The concept of the SIFT algorithm is explained later on in this section, whereas its modification for working with HDR images is described in the section 4.

2.1.1 SIFT algorithm

David G. Lowe in [11] suggests an algorithm for extracting local features which are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint [11]. The algorithm is called SIFT (Scale Invariant Feature Transform).

The major stages in extracting image features are:

1. Scale-space extrema detection.
2. Keypoint localization.
3. Orientation assignment.
4. Generation of keypoint descriptors.

2.1.2 Scale-space extrema detection

The first step in using the SIFT algorithm lies in extracting interest points. In order to find *interest points*, one has to search all over the scales and locations. This is implemented by building a multi-scale pyramid of Difference-of-Gaussian (DoG) images.

Given a Gaussian-blurred image:

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y), \qquad (1)$$

where: x, y - pixel position,
σ - denotes blur level,
$L(x,y,\sigma)$ - blurred image,
$G(x,y,\sigma)$ - Gaussian blurring function,

$$G(x,y,\sigma) = 1/(2\pi\sigma^2)\exp^{-(x^2+y^2)/\sigma^2}, \qquad (2)$$

$I(x,y)$ - image being blurred.
Each pyramid's level consists of:

$$D(x,y,\sigma) = L(x,y,k\sigma) - L(x,y,\sigma), \qquad (3)$$

where, D is an image computed from the difference of two nearby levels separated by a constant multiplicative factor $k = \sqrt{2}$.

A Gaussian pyramid and a Difference-of-Gaussian pyramid are illustrated on Figures 2 and 3. The number of pyramid's levels is not fixed and may be different in various implementations.

Level

Gaussian

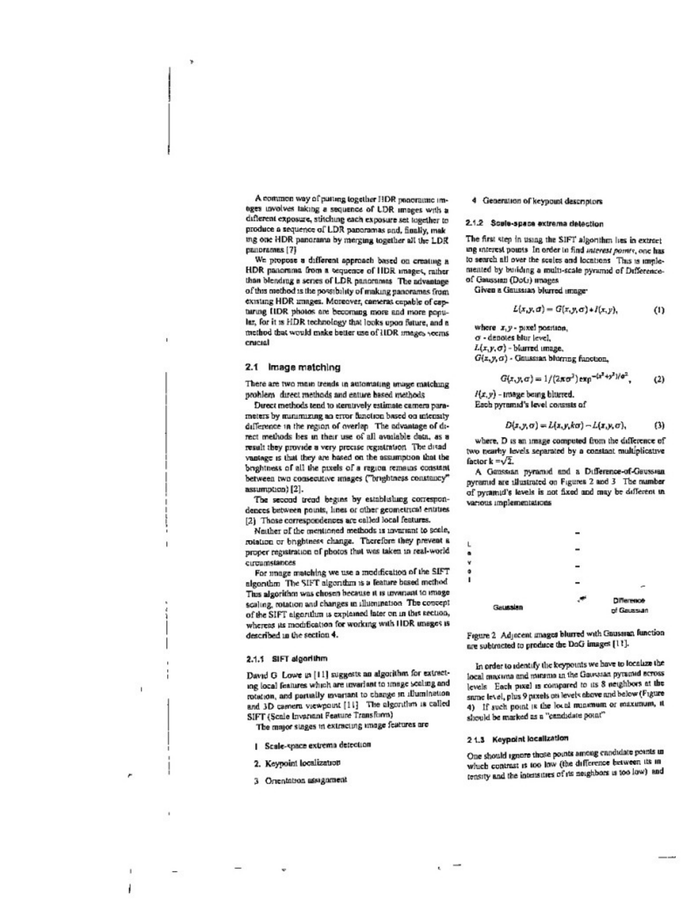
Difference of Gaussian

Figure 2: Adjacent images blurred with Gaussian function are subtracted to produce the DoG images [11].

In order to identify the keypoints we have to localize the local maxima and minima in the Gaussian pyramid across levels. Each pixel is compared to its 8 neighbors at the same level, plus 9 pixels on levels above and below (Figure 4). If such point is the local minimum or maximum, it should be marked as a "candidate point".

2.1.3 Keypoint localization

One should ignore those points among candidate points in which contrast is too low (the difference between its intensity and the intensities of its neighbors is too low), and



A common way of putting together HDR panoramic images involves taking a sequence of LDR images with a different exposure, stitching each exposure set together to produce a sequence of LDR panoramas and, finally, making one HDR panorama by merging together all the LDR panoramas [7].

We propose a different approach based on creating a HDR panorama from a sequence of HDR images, rather than blending a series of LDR panoramas. The advantage of this method is the possibility of making panoramas from existing HDR images. Moreover, cameras capable of capturing HDR photos are becoming more and more popular, for it is HDR technology that looks upon future, and a method that would make better use of HDR images seems crucial.

2.1 Image matching

There are two main trends in automating image matching problem: direct methods and eature based methods.

Direct methods tend to iteratively estimate camera parameters by minimizing an error function based on intensity difference in the region of overlap. The advantage of direct methods lies in their use of all available data; as a result they provide a very precise registration. The disadvantage is that they are based on the assumption that the brightness of all the pixels of a region remains constant between two consecutive images ("brightness constancy" assumption) [2].

The second trend begins by establishing correspondences between points, lines or other geometrical entities [2]. Those correspondences are called local features.

Neither of the mentioned methods is invariant to scale, rotation or brightness change. Therefore they prevent a proper registration of photos that was taken in real-world circumstances.

For image matching we use a modification of the SIFT algorithm. The SIFT algorithm is a feature based method. This algorithm was chosen because it is invariant to image scaling, rotation and changes in illumination. The concept of the SIFT algorithm is explained later on in this section, whereas its modification for working with HDR images is described in the section 4.

2.1.1 SIFT algorithm

David G. Lowe in [11] suggests an algorithm for extracting local features which are invariant to image scaling and rotation, and partially invariant to change in illumination and 3D camera viewpoint [11]. The algorithm is called SIFT (Scale Invariant Feature Transform).

The major stages in extracting image features are:

1. Scale-space extrema detection.
2. Keypoint localization.
3. Orientation assignment.
4. Generation of keypoint descriptors.

2.1.2 Scale-space extrema detection

The first step in using the SIFT algorithm lies in extracting interest points. In order to find *interest points*, one has to search all over the scales and locations. This is implemented by building a multi-scale pyramid of Difference-of-Gaussian (DoG) images.

Given a Gaussian-blurred image:

$$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y), \qquad (1)$$

where: x, y - pixel position,
σ - denotes blur level,
$L(x,y,\sigma)$ - blurred image,
$G(x,y,\sigma)$ - Gaussian blurring function,

$$G(x,y,\sigma) = 1/(2\pi\sigma^2)\exp^{-(x^2+y^2)/\sigma^2}, \qquad (2)$$

$I(x,y)$ - image being blurred.
Each pyramid's level consists of:

$$D(x,y,\sigma) = L(x,y,k\sigma) - L(x,y,\sigma), \qquad (3)$$

where, D is an image computed from the difference of two nearby levels separated by a constant multiplicative factor $k = \sqrt{2}$.

A Gaussian pyramid and a Difference-of-Gaussian pyramid are illustrated on Figures 2 and 3. The number of pyramid's levels is not fixed and may be different in various implementations.

Gaussian

Difference of Gaussian

Figure 2: Adjacent images blurred with Gaussian function are subtracted to produce the DoG images [11].

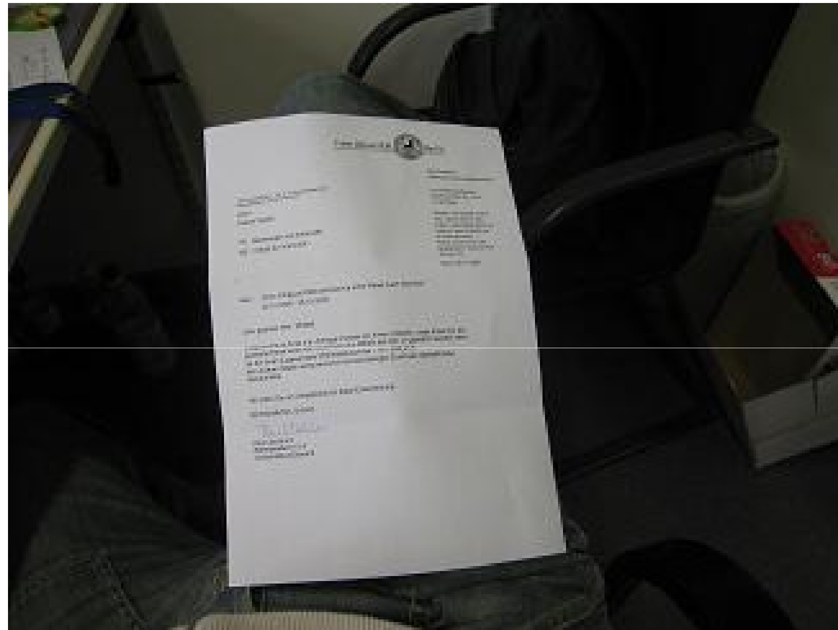
In order to identify the keypoints we have to localize the local maxima and minima in the Gaussian pyramid across levels. Each pixel is compared to its 8 neighbors at the same level, plus 9 pixels on levels above and below (Figure 4). If such point is the local minimum or maximum, it should be marked as a "candidate point".

2.1.3 Keypoint localization

One should ignore those points among candidate points in which contrast is too low (the difference between its intensity and the intensities of its neighbors is too low), and

Binarisierung V:

Binarisierung von Text zur Eliminierung von Schatten und Identifikation von Text:



Freie Universität Berlin