KOMPETENZZENTRUM FÜR MATHEMATISCHE MODELLIERUNG IN MINT-PROJEKTEN IN DER SCHULE

KOMMS Reports Nr. 7 (2017)

Reports zur Mathematischen Modellierung
in MINT-Projekten in der Schule

Einfaches Motion Capturing in MATLAB

Dr. Andreas Roth

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Zusammenfassung:
Der vorliegende Artikel befasst sich mit der Realisierung eines einfachen Motion Capturing Verfahrens in MATLAB als Vorschlag für eine Umsetzung in der Schule. Die zugrunde liegende Mathematik kann ab der Mittelstufe leicht vermittelt werden. Je nach technischer Ausstattung können mit einfachen Mitteln farbige Marker in Videos oder Webcam-Streams verfolgt werden. Notwendige Konzepte und Algorithmen werden im Artikel beleuchtet.

1 Einleitung

Motion Capturing (Bewegungserfassung) ist eine Bezeichnung für ein Verfahren, bei dem Bewegungsabläufe aus Bild- oder Videomaterial von einem Computer erkannt werden. Offensichtliche Anwendungen liegen zum einen in der Film- oder Videospielbranche, wo die Bewegungen menschlicher Darsteller gerne auf virtuelle Charaktere übertragen werden, welche dann ihrerseits in Videospielen, Animations- oder Spielfilmen vorkommen. In jüngerer Zeit findet die Bewegungserkennung auch Anwendung in Interfaces zur interaktiven Steuerung von Geräten wie Computern oder Spielekonsolen. Das Ziel hierbei ist, auf lange Sicht eine natürliche Bedienbarkeit ohne Tastaturen oder Zeigewerkzeuge wie Mäuse zu ermöglichen. Alle in diesen unterschiedlichen Bereichen angewandten Verfahren entstammen der Bildverarbeitung als Unterbereich der Mathematik und Informatik und unterscheiden sich je nach Anwendung erheblich in ihrer Komplexität. Einen natürlichen Bewegungsablauf dreidimensional zu erfassen und zu verarbeiten erfordert hohen technischen und finanziellen Aufwand, der von Privatpersonen nicht ohne weiteres erbracht werden kann, da synchronisierte Staffeln aus Kameras und rechenintensive Algorithmen benutzt werden müssen. Da man aber ohne größere Probleme mit verschiedenen Softwarelösungen die Bilder einer günstigen Webcam oder eines aufgezeichneten Videos auslesen kann, ist der technische Aufwand für die 2D Bilderkennung nicht so hoch. Wenn man sich des weiteren auf die Erkennung von Markern beschränkt, welche in einem Bild erkannt und in einem Video verfolgt werden sollen, dann ist auch der mathematische Aufwand gering und es sind auch keine umfassenden Programmierkenntnisse erforderlich. Das Ziel dieses Artikels ist also die Konzeption und Realisierung eines einfachen, Marker-basierten Erkennungsverfahrens. Hierbei werden an einer Person oder an Gegenständen farblich hervorstechende Punkte angebracht (grüne Tischtennisbälle). In Fotos oder Videos sollen diese dann nach Angabe der zu suchenden Farbe automatisch vom Computer gefunden werden. Dazu müssen wir klären, wie Farben oder Bilder im Computer repräsentiert sind, um darauf basierend die Position der gesuchten Punkte im Bild zu errechnen. Dabei werden nur mathematische Hilfsmittel benutzt, die mit Mittelstufenwissen schon nachvollziehbar sein sollten. Zusätzlich ist eine gewisse Affinität zum Programmieren hilfreich, da der Lernerfolg um so größer ist, wenn die theoretischen Konzepte direkt in eine funktionierende Software umgesetzt werden können.

2 Farbmodelle und Bilddatenformat

Betrachtet man das Bildpanel eines Computermonitors in angemessener Vergrößerung, so wird man feststellen, dass das Bild eigentlich aus einzelnen Leuchtpunkten, den *Pixeln*, besteht. Jedes Pixel nimmt eine bestimmte Farbe an. Alle Pixel zusammen betrachtet ergeben

Abbildung 2.1: Visualisierung des RGB (links) und des HSV (rechts) Farbraumes.

dann das Bild. Ein Pixel besteht nun zusätzlich aus drei Subpixeln mit den Farben Rot, Grün und Blau, deren Intensität einzeln gesteuert werden kann. Aus größerer Entfernung entsteht aus diesen drei Farben ein gemischter Farbwert, mit dem viele verschiedene natürliche Farbeindrücke dargestellt werden können. Die Bilddarstellung auf Monitoren basiert also auf dem *RGB-Farbmodell* [1]. Eine Farbe ist dabei durch die separaten Intensitätswerte der Farben Rot, Grün und Blau als Zahlentripel repräsentiert. Wie in Rechnersystemen üblich, sind Zahlen aufgrund der Speicherarchitektur im *Binärsystem* dargestellt, die kleinste Speichereinheit ist das *Bit*, welche die Ziffern 0 (aus) oder 1 (an) darstellen kann. Typischerweise stehen dabei zur Speicherung des Intensitätswertes eines Farbkanals 8 Bit an Speicher zur Verfügung. Das bedeutet, dass insgesamt $2^8 = 256$ Intensitätswerte aufgelöst werden können (0-255), und das eine komplette Farbe $3 \cdot 8 = 24$ Bit an Speicher belegt. Man spricht in einem solchen Falle von 24 Bit Farbtiefe. Insgesamt können so etwa 16 Millionen Farben dargestellt werden. Oft werden die Intensitätswerte auch stattdessen auf das Intervall $[0, 1]$ skaliert, was eine einfachere Verarbeitung mit Fließkommazahlen ermöglicht. Man kann sich nun die darstellbaren Farben mit ihren drei Intensitäten als Koordinaten in einem dreidimensionalen Farbwürfel vorstellen, was die gängige Visualisierung des RGB-Farbmodells ist (s.Abbildung 2.1). Der Koordinatenursprung $(0, 0, 0)$ entspricht Schwarz, der gegenüberliegende Punkt $(1, 1, 1)$ repräsentiert Weiß.

Alternativ können die Farben durch eine Koordinatentransformation auch mit anderen Farbmodellen dargestellt werden. Eines davon ist das *HSV-Farbmodell* [1], wo eine Farbe

Dateikopf	DiesIstEinRGBBild	01010101		
Informationsblock	FarbtiefeHöheBreite			
	24,2,2	00011000	00000010	00000010
Bilddaten	RGB			
	0,0,0	00000000	00000000	00000000
	255,120,0	11111111	01111000	00000000
	0,255,190	00000000	11111111	10111110
	25,110,255	00011001	01101110	11111111

Abbildung 2.2: *Schematische Darstellung eines Bildformates. Daneben befindet sich die Darstellung der Binärzahlen, die hintereinander im Speicher abgelegt würden.*

ebenfalls durch drei Zahlen repräsentiert wird. Diese entsprechen jetzt aber den Kanälen Farbton (*Hue*), Sättigung (*Saturation*) und Helligkeit (*Value*). Das Arbeiten mit dem HSV-Farbmodell fällt meistens etwas leichter, da die Veränderung des Farbwertes bei Änderung eines Kanalwertes intuitiver ist: Der Farbeindruck wird dunkler bei Änderung des Helligkeitswerts, der Farbton ändert sich entsprechend der Farbreihenfolge bei der Lichtbrechung und wird intensiver oder kontrastärmer bei Änderung des Sättigungswertes. Die Farben kann man sich für dieses Farbmodell in einem Farbzylinder gemäß Abbildung 2.1 angeordnet vorstellen. In Bildverarbeitungsprogrammen sind in Farbauswahldialogen die Farben oft in diesem HSV-Schema angeordnet, wobei man mit einem Regler den Farbton ändert und in einem Schnitt durch den Zylinder dann die entsprechenden Werte für Helligkeit und Sättigung anklicken kann.

Da wir nun wissen, wie Farben repräsentiert werden, können wir einzelne Farbpunkte zu Bildern zusammenfassen. Für die Bilder wiederum gibt es zahlreiche Datenformate wie PNG [2], die in ihren Standards festlegen, in welcher Struktur die Farbdaten im Speicher abgelegt sein müssen. Darauf basierend wird dann durch Software- und Hardwareentwickler sichergestellt, dass z.B. ein digitales Kamerabild gespeichert, verarbeitet, ausgedruckt oder auf einem Monitor dargestellt werden kann. Im folgenden wird an einem fiktiven Beispiel ein mögliches RGB-basiertes Bildformat entwickelt. Da im Computerspeicher einfach nur Nullen und Einsen hintereinander gereiht sind, ist es immens wichtig, dass in einem Standard wie [2] festgelegt ist, in welcher Reihenfolge und Form Bilddaten im Speicher abgelegt sind, damit diese auch richtig interpretiert werden können. Gemäß unserem erfundenen Standard besteht eine Bilddatei aus einem *Dateikopf*, einem *Informationsblock* und einem Block *Bilddaten* (s.Abbildung 2.2). Der Dateikopf besteht aus 8 Bit, die die Information kodieren, dass es sich bei den folgenden Blöcken um ein Bildformat handelt. Der Code in Abbildung 2.2 ist dabei frei erfunden. Damit weiß der Computer gemäß Standard, dass auf diesen Block nun ein Informationsblock mit drei 8-Bit Zahlen folgt, von denen die erste die Farbtiefe angibt, die beiden anderen die Auflösung in Zeilen und Spalten. Unser fiktiver Standard gibt an, dass die Farbwerte der Pixel zeilenweise im Speicher abgelegt

sind. Der Informationsblock in Abbildung 2.2 sagt uns:

- Die Farbtiefe beträgt 24 Bit, damit stehen für jeden Farbkanal 8 Bit an Speicher zur Verfügung
- Die Auflösung beträgt 2×2 Pixel
- Nach den 24 Bit des Informationsblocks beginnt der Datenblock und nach $24 * 2 * 2 = 96$ gelesenen Bits ist er zu Ende.
- Jeweils ein 8-Bit-Tripel bildet den Farbwert eines Pixels, beim seriellen Lesen ergeben zwei aufeinanderfolgende Pixel eine Bildzeile. Es gibt 2 Bildzeilen.

Diese genauen Angaben sind notwendig, da der Datenblock unseres Bildes nicht gegliedert dargestellt wird, sondern einfach alle zugehörigen Bits hintereinander im Speicher liegen. Nur der Standard und das Betriebssystem legen fest, dass auf die vorliegende Art und Weise angeordnete Daten ein Bild repräsentieren. Enthält der Informationsblock fehlerhafte Daten, so kann nicht errechnet werden, wo der Datenblock endet und es werden zu wenig oder zu viele Bits als Pixeldaten gelesen. Beides resultiert in einer fehlerhaften Bilddarstellung. Wird jedoch alles richtig gelesen, so kann der Computer das Bitmuster in die korrekte grafische Darstellung (s.Abbildung 2.3) umsetzen.

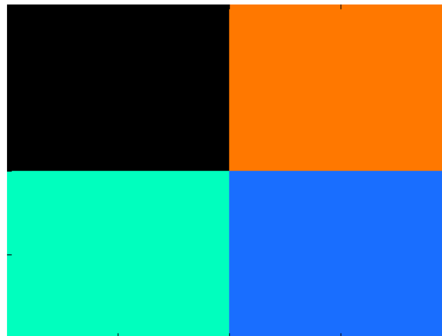

Abbildung 2.3: Grafische Repräsentation des fiktiven Formats, wenn alle Daten aus Abbildung 2.2 richtig interpretiert werden.

3 Farbmetriken und Umsetzung in MATLAB

Unser Ziel besteht darin, den Computer leuchtend grüne Punkte in einem Bild wie in Abbildung 3.1 finden zu lassen. Wir wollen mit den Bilddaten zur Bewegungserkennung arbeiten und uns nicht mit der Realisierung ihrer grafischen Darstellung befassen. Diese sollte aber vorhanden sein, damit wir unsere Ergebnisse kontrollieren können. Es sollte also eine Sprache mit einem vorhandenen, einfach zu nutzenden Grafiksystem zur Umsetzung der hier diskutieren Methoden benutzt werden. Dazu nutzen wir die kommerzielle Software MATLAB, die uns ein solches bietet, eine eigene Scriptsprache zum Programmieren und Funktionen zur Bildverarbeitung mitbringt. Es können aber auch entsprechende kostenlose Lösungen wie GNU Octave, Scilab, FreeMat oder Python mit entsprechenden Paketen benutzt werden, wo es ebenfalls schon vorgefertigte Funktionen zum Einlesen und Darstellen von Bilddaten gibt. Die angebenen Codeausschnitte und Funktionen beziehen sich auf MATLAB,sollten aber mit leichten Anpassungen auch mit den anderen vorgeschlagenen Programmen laufen. Es kann aber auch einfach alles als Pseudocode gelesen werden und in der Programmiersprache der Wahl umgesetzt werden.

Abbildung 3.1: Beispielbild, in dem grüne Tischtennisbälle gefunden werden sollen.

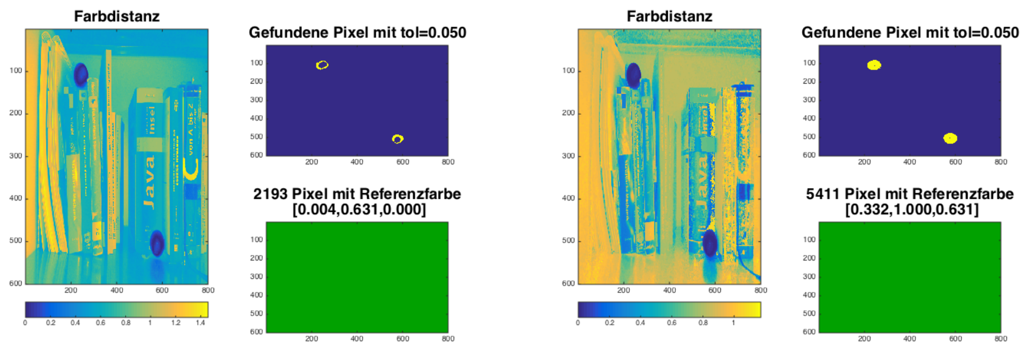

Abbildung 3.2: Links: Farberkennung für RGB. Rechts: Farberkennung für HSV. Bei gleicher Toleranz und Gewichten des Helligkeitskanals bei HSV mit $c_V = 0.1$ werden mit der HSV-Metrik mehr zu den gesuchten Gegenständen gehörende Pixel gefunden.

Wenn wir mit einem Befehl wie `X=imread('dateiname.png')` ein Bild einlesen, so erhalten wir mit `X` ein Array, welches des Dimensionen $n_x \times n_y \times 3$ hat, wobei n_x, n_y die Bildauflösung angeben. Die Farbdaten für Rot, Grün und Blau sind als Werte zwischen 0 und 255 in der dritten Array-Dimension angeordnet. So erhalten wir die Rot-Werte für alle Pixel mit `X(:,:,1)`, alle Grün-Werte mit `X(:,:,2)` und alle Blau-Werte mit `X(:,:,3)`.
Die zu findende Referenzfarbe hat ebenfalls einen RGB-Vektor. Nun, da wir Zugriff auf die Farbdaten haben, müssen wir irgendwie die Ähnlichkeit der Pixelfarben zu unserer Referenzfarbe quantifizieren. Dabei können wir uns der Interpretation des RGB-Farbmodells als Würfel bedienen: Sind die einzelnen Farben Punkte in dessen dreidimensionalen Koordinatensystem, so können wir ohne weiteres Abstände zwischen den Farben berechnen, z.B. mittels der *euklidischen Metrik*. Sei unsere Referenzfarbe $R = (r_R, g_R, b_R)$ gegeben, dann erhalten wir die Distanz $d(R, C_{ij})$ zur Pixelfarbe i, j, $C_{ij} = (r_{ij}, g_{ij}, b_{ij})$, durch

$$d(R, C_{ij}) = \sqrt{(r_R - r_{ij})^2 + (g_R - g_{ij})^2 + (b_R - b_{ij})^2}$$

Der Farbabstand kann nun für jedes Pixel berechnet werden. Zwei Farben sind ähnlich, wenn ihre Abweichung in dieser Metrik klein ist. Dazu kann man einen Toleranzwert tol definieren, sodass alle Pixel mit $d(R, C_{ij}) < tol$ als Pixel mit der Referenzfarbe gezählt werden. Wählt man tol zu groß, werden zu viele Pixel gefunden, wählt man ihn zu klein, so findet man vielleicht gar keine Pixel mehr. Auch bei einer Änderung der Beleuchtungssituation kann es sein, dass ein eben noch sinnvoller Toleranzwert keine sinnvollen Ergebnisse mehr liefert. Die korrekte Wahl von tol ist bei RGB-Farben deshalb recht schwer, da das Farbmodell weniger intuitiv ist und für das menschliche Auge klar unterschiedliche Farben im Farbwürfel doch relativ nahe zueinander liegen können. Hier bietet sich der Wechsel auf den HSV-Farbraum und eine gewichtete Metrik an. Für die Transformation der RGB- in HSV-Farben findet man nach kurzer Suche im Internet zahlreiche fertige Umsetzungen in verschiedenen Programmiersprachen. In MATLAB steht bereits die Funktion `rgb2hsv` zur Verfügung. Sei wieder $R = (h_R, s_R, v_R)$ die Referenzfarbe in HSV, und $C_{ij} = (h_{ij}, s_{ij}, v_{ij})$ die Farbe des i, j-ten Pixels, so ist eine Farbmetrik gegeben durch

$$d_c(R, C_{ij}) = \sqrt{c_H(h_R - h_{ij})^2 + c_S(s_R - s_{ij})^2 + c_V(v_R - v_{ij})^2}$$

mit Gewichten $0 \leq c_H, c_S, c_V \leq 1$. Damit kann nun z.B. der Farbtonabstand gegenüber der Helligkeit des Farbeindrucks stärker gewichtet werden, was die Sensitivität des Farbabstands bezüglich Belichtungsänderung verringert. Damit werden auch bei fester Beleuchtungssituation größere relevante Farbbereiche gefunden, da in erster Linie die Nähe der Farbtöne zählt, und nicht die Helligkeit. Bei RGB ändert sich die Helligkeit mit allen 3 Farbkanälen, sodass eine gleichwertige Gewichtung nicht so einfach möglich ist.

In Abbildung 3.2 sind für einen Toleranzwert von 0.05 und die angezeigte Referenzfarbe die gefundenen Pixel aus Abbildung 3.1 markiert. Hierbei wurde bei der HSV-Metrik der Helligkeitskanal mit 0.1 gewichtet, sodass Helligkeitsunterschiede eine geringere Rolle spielen als Farbton- oder Sättigungsunterschiede. Im Vergleich zur RGB-Metrik geben die gefundenen Pixel die tatsächlichen Ausdehnungen der Tischtennisbälle besser wieder. Die Nutzung der HSV-Metrik birgt jedoch den Nachteil, dass das Bild zuerst in HSV konvertiert werden muss, was je nach Auflösung signifikant viel Rechenzeit kostet, insbesondere, wenn jedes Bild eines Videos ausgewertet wird. Die Berechnung der Farbdistanzen kann in MATLAB

mittels komponentenweiser Matrixoperationen durchgeführt werden, wodurch man die in interpretierten Scriptsprachen normalerweise langsamen Schleifen vermeiden kann. Der erforderliche Code zum Berechnen der Farbdistanzen wird dadurch auch erheblich kürzer:

Listing 3.1: Metrik auswerten

```
% Bild einlesen, B enthaelt Farbdaten
B=imread(name);
B=imresize(B,[600,800]);
B=double(B)/255;

% Referenzfarbe speichern
refCRGB=[0.0039,0.6314,0];

% Bildmatrizen erzeugen, die nur die Referenzfarbe
% in jedem Pixel enthalten
nn=size(B);
refCMatRGB=repmat(permute(refCRGB,[1,3,2]),nn(1),nn(2));
% Konvertierung nach HSV
refCMatHSV=rgb2hsv(refCMatRGB);
refCHSV=permute(refCMatHSV(1,1,:),[1,3,2]);

% Farbdistanzen berechnen, komponentenweise
abst1=B-refCMatRGB;
abst2=rgb2hsv(B)-refCMatHSV;

% RGB Metrik, euklidische Distanz
dist1=sqrt(sum(abst1.^2,3));
% HSV Metrik, Kanaele gewichtet
dist2=sqrt(1*abst2(:,:,1).^2+1*abst2(:,:,2).^2+0.1*abst2(:,:,3).^2);

% Ipix1 und 2 enthalten 1 an gefundenen Pixeln, 0 sonst
Ipix1=dist1<tol1;
Ipix2=dist2<tol2;
```

Die Arrays `dist1` und `dist2` sind in Abbildung 3.2 als *Farbdistanz* für RGB bzw. HSV dargestellt. Die logischen Arrays `Ipix1` und `IPix2` sind dort unter *Gefundene Pixel...* visualisiert.

4 Koordinaten finden mit k-Means

Nachdem wir nun erfolgreich Pixel mit Referenzfarbe im Bild identifizieren können, müssen wir diese Informationen noch in Koordinaten im Bild-Koordinatensystem umrechnen. Dabei liegt der Ursprung in der linken oberen Ecke, die x-Richtung ist die vertikale, die y-Richtung die horizontale Achse. Grundsätzlich hat jedes Pixel in diesem Koordinatensystem eine Adresse, wenn man die Pixel gemäß Abbildung 4.1 nummeriert. Hat man nur einen Pixelhaufen gefunden, so könnte man einfach den Schwerpunkt der gefundenen Pixelkoordinaten bestimmen, welcher dann die Position des markierten Gegenstandes wiedergeben würde. Das funktioniert allerdings schon nicht mehr, wenn mehrere Marker wie

in Abbildung 3.1 zu finden sind. Der Schwerpunkt aller gefundenen Pixelkoordinaten läge dann in der Mitte zwischen den beiden Markern, was keine von beiden Positionen korrekt repräsentiert.

Es wird also ein Verfahren benötigt, dass in einer vorliegenden Punktewolke mit mehreren Häufungen (*Clustern*) diese identifizieren und ihren jeweiligen Schwerpunkt berechnen kann, wie es in Abbildung 4.1 dargestellt ist. So etwas ist mit dem *k-Means* Algorithmus [3] bereits realisiert worden. Dabei werden vorliegende Punktkoordinaten in eine bestimmte Anzahl Cluster eingeteilt. Nach jedem Einteilungsschritt werden die Schwerpunkte jedes Clusters berechnet und die Gesamtsumme der Abstände der Punkte zu ihren Clusterzentren gebildet. Diese Schritte werden so lange wiederholt, wie sich die Gesamtsumme der Abstände noch verringert. Wenn dies nicht mehr geschieht, so kann davon ausgegangen werden, dass die berechneten Clusterzentren eine sinnvolle Aufteilung der Punktewolke darstellen.

Den Algorithmus gibt es in verschiedenen Implementierungen. Die hier verwendete Version funktioniert folgendermaßen:

Als Parameter bekommt der Algorithmus: Punkte $P_i = (x_i, y_i), i = 1, 2, ..., n$, die in Cluster geteilt werden sollen, die Anzahl der zu findenden Cluster k und Startpositionen für die Clusterschwerpunkte $C_j, j = 1, 2, ..., k$. Letztere können auch zufällig gesetzt werden.

- Wiederhole, so lange A sich noch verringert:
  - Für alle i=1,...,n
    - Bestimme für P_i Clusterzentrum C_j mit geringstem Abstand
    - Füge P_i zu Cluster J hinzu
  - für alle j=1,...,k
    - Wenn Cluster J nicht leer, bestimme neues Zentrum C_j mit (n_j ist Anzahl der Punkte in J)

$$C_j = \sum_{i \in J} P_i / n_j$$

    - Summiere alle Abstände der Punkte in J zu C_j:

$$S_j = \sum_{i \in J} d(P_i, C_j)$$

    - Wenn Cluster J leer, platziere C_j zufällig und fahre fort.
  - Berechne die Gesamtsumme neu:

$$A = \sum_j S_j$$

In Listing 4.1 ist der vorgestellte Algorithmus in der Funktion `myKmeans` in MATLAB realisiert. Mit Hilfe der Befehle `[I,J]=find(Ipix1==1)` (s.Listing 3.1) können wir uns in den Arrays `I` und `J` die gefundenen Pixelkoordinaten ausgeben lassen, welche wir dann als Eingang für das Argument `X` in Listing 4.1 verwenden können (`X=[I';J']`). In Abbildung

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)

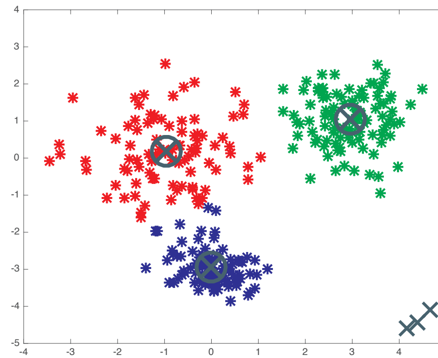

Abbildung 4.1: Links: Pixelkoordinaten im Bild-Koordinatensystem. Wie reduziert man die gefundenen roten Pixel zu einer Koordinate, die die Position des gesuchten Gegenstandes korrekt repräsentiert? Rechts: Punktewolke mit durch k-Means gefundenen Schwerpunkten.

Abbildung 4.2: Mit k-Means gefundene Koordinaten, basierend auf den in Abbildung 3.2 gefundenen Pixeln.

4.2 kann man sehen, dass die korrekten Schwerpunkte der beiden relevanten Pixelkoordinaten gefunden wurden und dass die Positionen der Tischtennisbälle korrekt wiedergegeben werden.

Listing 4.1: Beispielimplementierung k-Means

```
function [center]=myKmeans(X,anzClust,centerIni)
% X         : 2xn Matrix, jede Spalte ist ein Punkt der Punktewolke
% anzClust  : anzahl der zu findenen Zentren
% centerIni : Startpositionen der Zentren

% Initialisierungen
center=centerIni;
clustIndices=1:anzClust;
anzPkt=size(X,2);
abstSum=zeros(1,anzClust);
gesamtSum=1e100;
gesamtSumAlt=inf;
anzIt=0;
% Mittelwert und Standardabweichung Punktewolke
M=sum(X,2)/anzPkt;
var=sqrt(sum(sum((X-repmat(M,[1,anzPkt])).^2,1),2)/anzPkt);
% Hauptschleife
while (gesamtSum<gesamtSumAlt)&&(anzIt<100)
    gesamtSumAlt=gesamtSum;
    % Finde zu jedem Punkt das naechstliegende Clusterzentrum
    dist=repmat(X,[1,1,anzClust])...
         -repmat(permute(center,[1,3,2]),[1,anzPkt,1]);
    norm=dist(1,:,:).^2+dist(2,:,:).^2;
    [~,minI]=min(norm,[],3);
    % Finde heraus, ob und welche Clusterzentren nicht vorkommen
    [c]=unique(minI);
    c2=setdiff(clustIndices,c);
    % c2 enthaelt Indices der nicht zugewiesenen Clusterzentren, diese
    % werden spaeter neu positioniert
    % Basierend auf der aktuellen Clusterzuweisung, errechne neue
    % Schwerpunkte
    for k=1:anzClust
        % Berechne die neuen Zentren
        center(:,k)=sum(X(:,minI==k),2)/sum(minI==k);
        % Berechne die Abstandssumme fuer Cluster k
        abstSum(k)=sum(sum((X(:,minI==k)...
            -repmat(center(:,k),[1,sum(minI==k)])).^2,1),2);
    end
    % Clusterzentren, die nicht zugewiesen wurden, werden zufaellig um
    % das erste Clusterzentrum neu positioniert
    center(:,c2)=var*rand(2,length(c2))...
        +repmat(center(:,c(1)),[1,length(c2)]);
    % Gesamtsumme der Abstaende der Punkte zu den zugeordneten
```

```
    % Clusterzentren
    gesamtSum=sum(abstSum,2);
    anzIt=anzIt+1;
end
```

5 Zusammenfassung

Es wurde gezeigt, wie in einem Kamerabild Positionen farblich markierter Gegenstände berechnet werden können. Das Verfahren dafür ist recht einfach zu realisieren und bildet den Grundbaustein für komplexere Anwendungen, welche darauf aufbauen könnten. Allerdings unterscheiden sich die konkreten Realisierungen je nach Plattform stärker. In MATLAB ist es kein Problem, das Erkennungsverfahren auf die Einzelbilder eines Videos oder eines Kamerastreams anzuwenden und damit auch in Echtzeit Punkte zu verfolgen. Diese Daten können dann weiter verarbeitet werden. Denkbar wäre z.B. eine Anwendung im Sport, wo vollautomatisch Bewegungsabläufe verfolgt, analysiert und optimiert werden können. Hat man Zugriff auf den Bilderstream einer Webcam, so kann man in Echtzeit virtuelle Objekte an den markierten Punkten einblenden, die diesen dann durch das Bild folgen.

Literatur

[1] M. K. Agoston. *ComputerGraphics and Geometric Modeling: Implementation and Algorithms*. Number ISBN 1-85233-818-0. 2005.

[2] ISO/IEC 15948:2004, Information technology - Computer graphics and image processing - Portable Network Graphics (PNG): Functional specification. http://www.iso.org/iso/catalogue_detail.htm?csnumber=29581.

[3] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. *University of California Press*, pages S. 281–297, 1967.