Maximalpunktzahl

Körper vermessen mit Python und OpenCV

Der japanische Modehersteller Zozo verschickt für nur drei Euro Anzüge, mit denen Kunden ihre Körper vermessen sollen. Mit dem Zozo-Suit kann man aber nicht nur seine Konfektionsgröße bestimmen, sondern auch Daten fürs Motion Tracking sammeln. Wir zeigen, wie Sie die Grundlagen dafür mit Python und OpenCV schaffen.

Von Pina Merkert

Der von Zozo verschickte Messanzug ist raffiniert: Man zieht ihn an, lässt die App von allen Seiten Fotos von einem schießen und bekommt nach einigen magischen Berechnungen einen umfangreichen Satz akkurater Körpermaße, ohne dafür ein Maßband gebraucht zu haben [1]. Die App wertet dafür die Positionen von Dutzenden zwei Zentimeter durchmessenden weißen Punkten aus, die Zozo auf den faltenfreien Elasthan-Anzug gedruckt hat.

Zozo verschenkt den Anzug. Man muss sich lediglich registrieren und die Versandkosten von drei Euro zahlen, um einen zu bekommen. Ob man danach Kleider bei Zozo bestellt, bleibt einem selbst überlassen. Natürlich haben wir uns sofort gefragt, ob man denn mit dem Anzug nicht noch mehr machen kann. Und tatsächlich haben wir ein Gist des japanischen Forschers Kazuhiro Sasao mit Python-Code gefunden, der einige der Marker auf dem Anzug auch ohne die Zozo-App identifiziert (siehe ct.de/yx6j). Der Code erkennt frontal aufgenommene Punkte recht gut, scheitert aber an Punkten, die nur schräg zu sehen sind.

Das Programm basiert auf dem Open-Computer-Vision-Framework (OpenCV), das man über ein vollständiges Binding auch von Python aus verwenden kann. OpenCV ist hoch optimiert, sodass die rechenintensiven Operationen beim Analysieren der Fotos vom Zozo-Suit auf einem Mittelklasse-PC in weniger als fünf Sekunden fertig werden. Außerdem ist es ein umfangreicher Werkzeugkasten, der uns zum Experimentieren animiert hat. Mit einem neuen Ansatz beim Auswerten der Punkte gelang es uns auch, viele der schräg aufgenommenen Punkte korrekt zu erkennen. Unser als Open Source auf GitHub veröffentlichtes Programm (ct.de/yx6j) liefert für Bilder vom Zozo-Suit Positionen der Punkte, zum Wiedererkennen eine ID, eine grob geschätzte Entfernung und

einen Wert, der aussagt, wie sicher das Programm ist, einen Punkt gefunden zu haben. Mit diesen Daten können Sie eigene Anwendungen für den Zozo-Suit entwickeln.

Bildanalyse

Wie er nach Punkten suchen soll, erklärt man dem Rechner mit OpenCV. Es nutzt Pillow, den Nachfolger der Python-Imaging-Library (PIL), sowie Numpy, das Berechnungen mit Arrays und Matrizen erleichtert. Alle drei Pakete installieren Sie mit `pip` in ein Virtualenv:

```
python3 -m venv env
source env/bin/activate
pip install pillow numpy opencv-python
```

Windows-Nutzer, die Python nicht zum `PATH` hinzugefügt haben, müssen beim ersten Befehl den vollen Pfad zur `python.exe` angeben. Der zweite Befehl besteht unter Windows aus dem Aufruf von `Scrips\activate` (Details erklärt die Dokumentation ct.de/yx6j).

Danach importiert man nur noch OpenCV ins Python-Programm und lädt ein Bild:

```
from cv2 import imread
im = imread("Dateiname.jpg")
```

Die Suche nach den Punkten übernimmt dann unsere Funktion `detect_points()` aus `detect_points.py` im Repository auf GitHub (ct.de/yx6j).

Konturensuche

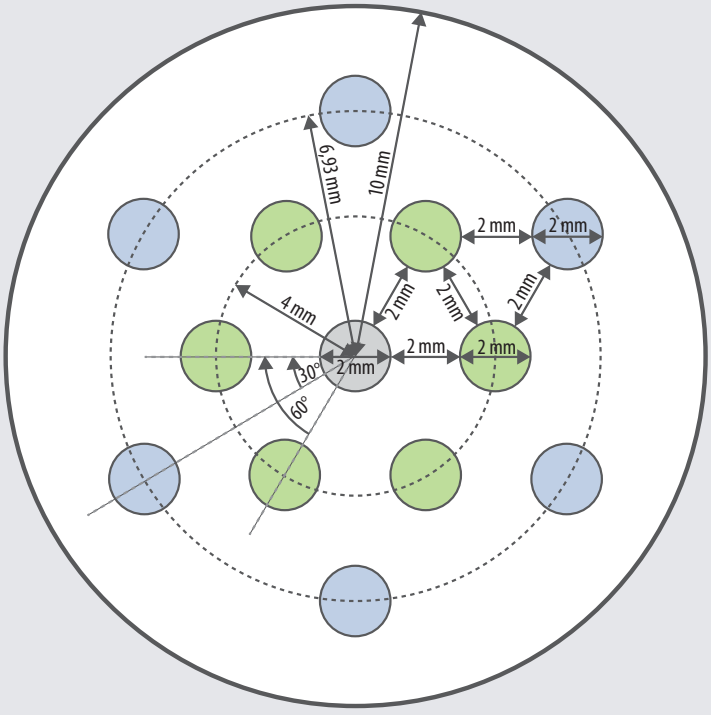
`detect_points()` fahndet zunächst nach kontrastreichen Formen in einer Größe, die zur Größe der Marker auf dem Anzug passt. Formen sucht man mit OpenCV recht leicht als Konturen mit der Funktion `findContours()`. Die erwartet aber kein Farbbild, sondern ein binäres Bild, also ein Bild, in dem nur die Farben Weiß und Schwarz vorkommen.

Ein solches erzeugt `threshold()` aus einem Graustufenbild. Die Funktion unterstützt mehrere Verfahren, den Schwellwert zu berechnen. Einfache Varianten verwenden lediglich einen fixen Schwellwert. Um besser mit unterschiedlichen Belichtungen umzugehen, verwendet unser Programm das Verfahren von Otsu. Das passt den Schwellwert so an, dass die Varianzen der Farbwerte in schwarzen und weißen Bereichen jeweils klein, die Varianz zwischen den Klassen jedoch gleichzeitig möglichst groß ist (siehe ct.de/yx6j). Der Vorteil davon: Man muss nicht für jedes Bild per Hand nach einem geeigneten Schwellwert suchen.

Die OpenCV-Funktion `cvtColor(im, COLOR_BGR2GRAY)` konvertiert das Farbbild zuvor in ein Graustufenbild. Im Prinzip reicht das zum Berechnen der Umrisse. Bei verrauschten Bildern kann das Rauschen aber zu ausgefransten Umrissen führen, was die Erkennung verschlechtert.

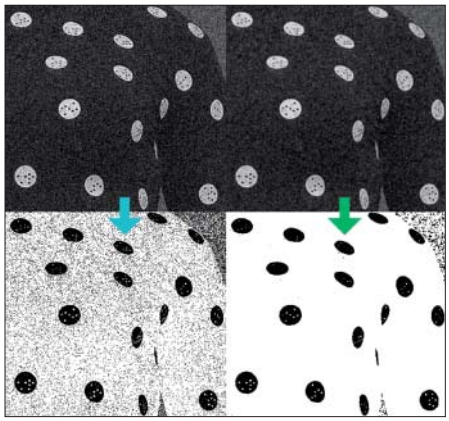
Punktekunde

Es gibt auf dem Zozo-Suit Punkte in zwei Größen. Um den Kragen, die Handgelenke und Knöchel sind kleine Punkte mit einem Zentimeter Durchmesser angeordnet, die sich nicht unterscheiden lassen. Über den Rest des Anzugs sind circa 300 Punkte mit zwei Zentimetern Durchmesser verteilt, die im Inneren ein einzigartiges Muster aus zwei Millimeter großen schwarzen Pünktchen haben, wodurch sie sich eindeutig voneinander unterscheiden lassen. In der Mitte gibt es immer einen schwarzen Mittelpunkt, selbst bei den kleinen Punkten am Kragen, Handgelenk und Knöchel. Bei den großen Punkten können sich auf einem Kreis um diesen Mittelpunkt weitere sechs Pünktchen im Abstand von vier Millimeter befinden. Beim Zozo-Suit sind allerdings nie alle sechs Pünktchen auf diesem Kreis vorhanden, aber mindestens zwei.



Den inneren Kreis umschließt ein äußerer Kreis, auf dem ebenfalls zwei bis fünf schwarze Pünktchen liegen können. Da diese Pünktchen einen gleichmäßigen Abstand von vier Millimetern zu Pünktchen auf dem inneren Kreis haben sollen, hat der Kreis einen Radius von etwa 6,93 Millimeter und die Positionen der Pünktchen sind um 30° verdreht. Die Zeichnung zeigt alle möglichen Positionen für schwarze Pünktchen, die aber nie alle besetzt sind.

Ein schwarzes Pünktchen kodiert ein Bit an Information. Mit zwei Kreisen mit je sechs Pünktchen lassen sich also theoretisch 12 Bit darstellen. Da die Pünktchen in der Praxis aber verdreht sein könnten, zählt der Algorithmus nur das der sechs möglichen Muster (jeweils um 60° verdreht), dessen 12 Bit die höchste Zahl ergeben. Beachtet man zusätzlich die Einschränkung, dass nur zwei bis fünf Pünktchenpositionen auf jedem Kreis besetzt sein können, bleiben 608 mögliche Muster und damit auch nur 608 unterschiedliche IDs für die zwei Zentimeter großen Punkte.



Bei stark verrauschten Bildern sorgt ein Gaußscher Weichzeichner (Größe 3 × 3, rechte Spalte) für weniger ausgefranste Konturen (untere Zeile).

Ein Gaußscher Weichzeichner glättet das Bild, damit das nicht passiert. Alles zusammen sieht dann so aus:

```
im_gray = cvtColor(im, COLOR_BGR2GRAY)
im_blur = GaussianBlur(
      im_gray, (3, 3), 0)
ret, th = threshold(
      im_blur, 0, 255,
      THRESH_BINARY_INV + THRESH_OTSU)
imgEdge, cont, hier = findContours(th,
      RETR_TREE, CHAIN_APPROX_NONE)
```

In der Variable `cont` steht danach eine Liste aller im Bild gefundenen Konturen. Die Liste `hier` definiert die Hierarchie dieser Konturen, also welche innerhalb von anderen Konturen liegen.

Plausibilitätsprüfung

OpenCV bringt einige praktische Funktionen mit, um zu prüfen, ob eine Kontur der Umriss eines Markers sein könnte. `boundingRect()` bestimmt Höhe, Breite und Position einer Kontur. `contourArea()` berechnet den Flächeninhalt, `arcLength()` die Umrisslänge. `fitEllipse()` findet die Ellipse, die möglichst genau der Kontur entspricht.

Mit diesem Handwerkszeug prüft der Code zunächst die grobe Größe. Die Annahme ist, dass eine Person zwischen 1,3 und 2,2 Meter groß ist. Außerdem sollte sie so fotografiert sein, dass sie mindestens die Hälfte der Bildhöhe einnimmt, maximal aber genau ins Bild passt. Mit diesen Annahmen lassen sich ein minimaler und ein maximaler Faktor berechnen, mit dem man Höhe und Breite der Konturen in Pixeln multipliziert und mit der Punktgröße in Metern (0,02 m) vergleicht. Zu kleine und zu große Konturen kommen nicht infrage.

Danach prüft der Code, ob der Flächeninhalt der Kontur nicht größer als der größtmögliche Kreis ist, was beispielsweise quadratische Konturen aussortiert. Die Umrisslänge einer kleinstmöglichen Ellipse ist aber gleichzeitig nicht kleiner als der doppelte Punktdurchmesser. Außerdem sollte der Umriss nicht viel länger sein als der Umriss einer Ellipse mit der gleichen Höhe und Breite. Das schließt Konturen aus, die stark von einer elliptischen Form abweichen.

Nur wenn die einfachen Prüfungen erfolgreich waren, fittet das Programm eine Ellipse in die Kontur. Danach scheiden extrem flache Ellipsen aus (kleiner Durchmesser weniger als 10 Prozent des großen Durchmessers), da es wegen zu geringer Auflösung für sie keine Hoffnung gibt, dass der Algorithmus eine korrekte ID findet. Danach prüft der Code, wie stark die Kontur von der Ellipse abweicht.

Das ist mathematisch kein einfaches Problem, da das Programm zu jedem Punkt der Kontur den Punkt auf der Ellipse mit dem kleinsten Abstand finden muss. Versucht man das Problem analytisch zu lösen, bekommt man eine quadratische Gleichung, die sich nicht auflösen lässt. Wir verwenden deswegen den von Carl Chatfield beschriebenen iterativen Algorithmus, der in nur drei Iterationen einen Punkt findet, der innerhalb der Genauigkeit von 32-Bit-Gleitkommazahlen dem korrekten Punkt entspricht. Carl Chatfield erklärt den Algorithmus in seinem englischen Blog sehr anschaulich (siehe ct.de/yx6j). Die von uns verwendete Implementierung in `ellipse_helpers.py` entspricht weitgehend der im Blog beschriebenen Idee, verwendet aber ein paar zusätzliche Optimierungen, die Chatfield auf StackOverflow verraten hat.

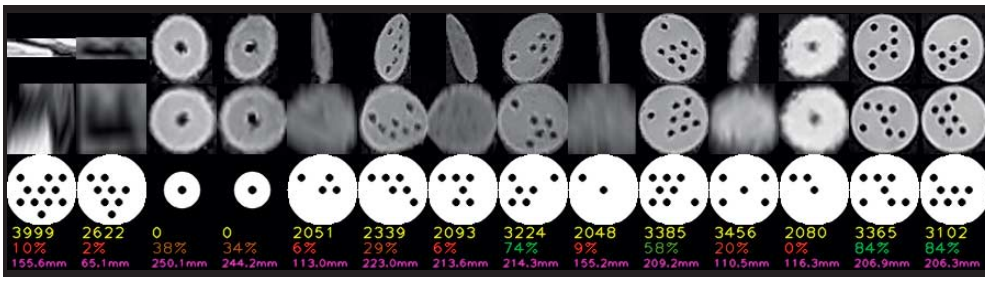
Die Prüfungen sortieren alle Konturen aus, die als Marker nicht infrage kommen. Übrig bleibt eine Liste mit Konturen und gefitteten Ellipsen, die vielleicht Marker sein könnten, bei denen aber je nach Hintergrund noch Formen dabei sein könnten, die nicht zu Markern gehören.

Verkreisung

Handelt es sich bei den Formen um Marker, sind diese kreisförmig, wenn sie direkt von vorne fotografiert sind, sonst elliptisch. Außerdem kann die Ellipse noch um ihren Mittelpunkt gedreht sein.

Die Punkte lassen sich am leichtesten auswerten, wenn die Kontur kreisförmig ist. Dafür transformiert der Algorithmus alle Formen in quadratische Ausschnitte des Bilds, bei denen Breite und Länge dem längeren Durchmesser der Ellipse entsprechen, um keine Auflösung zu verlieren.

Um aus der Ellipse einen Kreis zu machen, muss man sie um den Winkel, um den sie verdreht ist, zurückdrehen und den kleinen Durchmesser so strecken, dass er so groß wie der größere Durchmesser wird. Beides sind affine Transformationen, die man mit einer geeigneten Transformationsmatrix zu einer Operation zusammenfassen kann. OpenCV bringt die praktische Funktion `getAffineTransform()` mit, die die nötige Transformationsmatrix berechnet. Man muss lediglich drei Punkte im Quellbild und die gewünschten Positionen der Punkte im Zielbild angeben. Unser Algorithmus legt dafür die Mittelpunkte von Ellipse und Kreis übereinander und ordnet die Extrempunkte der Ellipse (einer an der breitesten Stelle und einer an der schmalsten



Die erste Reihe zeigt Formen, die als Marker infrage kommen. Die zweite Zeile streckt sie auf eine quadratische Fläche. In der dritten Zeile hat das Programm die erkannte ID als Punkt dargestellt (ohne Verdrehung der Vorlage), in Gelb darunter steht die ID als Zahl. Bei falsch erkannten Punkten hat das Programm wenig Zutrauen in seine Erkennung (rote Zahlen ganz unten).

Stelle) zwei Kreispunkten (oben und links) zu. Die eigentliche Berechnung übernimmt dann `warpAffine()`.

Diese Berechnungen führt die Funktion `unskew_point()` in `markers.py` aus. Sie gibt entzerrte quadratische Bilder zurück, die aber unterschiedlich groß sein können.

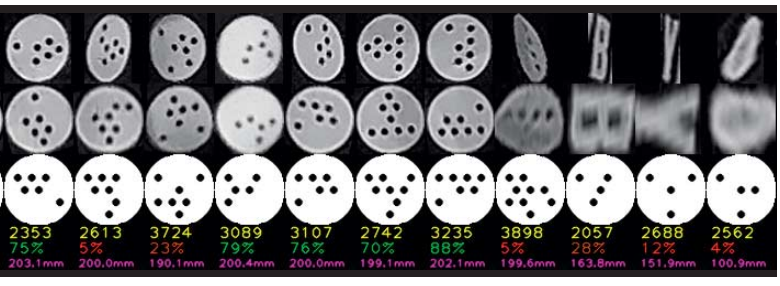
Erkennungsdienst

Um die Erkennung der schwarzen Pünktchen, die die Marker identifizieren, kümmert sich `get_point_id()`. Kazuhiro Sasaos Code nutzte dafür Konturen, die er aber bei schräg aufgenommenen Punkten nicht mehr verlässlich erkennen konnte. Unser Code vergleicht stattdessen die Helligkeit von Pixeln, was robuster arbeitet.

Die Funktion steht zunächst dem Problem gegenüber, dass das Muster im Bild beliebig verdreht sein könnte. Sie muss also zunächst herausfinden, um welchen Winkel das Muster gedreht ist. Da die schwarzen Punkte auf den beiden Ringen im Abstand von 60° vorkommen, reicht es, nur die Verdrehungen bis zu einem Sechstelkreis zu prüfen. Dafür legt die Funktion eine Maske über das Bild, die alle zwölf möglichen Pünktchen freistellt. Ist diese Maske so gedreht wie die tatsächlich vorhandenen Pünktchen im Bild, haben die maskierten Pixel eine kleinere durchschnittliche Helligkeit als alle Masken mit abweichendem Winkel. Sind Maske und Punkt gegeneinander verdreht, ist durch die Maske nämlich mehr Weiß vom Punkt zu erkennen. Für diese Prüfung berechnet das Programm zunächst 60 verschieden gedrehte Masken (Schritte von 1° erwiesen sich in unseren Experimenten als ausreichend).

Die Funktion `generate_mask()` berechnet invertierte Bilder (Farbwerte von 0 bis 1) von Punkten. Dafür zeichnet sie keinen Umriss, sondern nur die sonst schwarzen Pünktchen als weiße Pünktchen (Wert 1) auf schwarzem Grund (Wert 0). Eine Bitmaske gibt an, welche der zwölf möglichen Pünktchen dabei vorkommen. `generate_mask()` verwendet dafür die in OpenCV integrierte Zeichenfunktion `circle()`, die beim Linientyp `LINE_AA` auch Kreise mit Kantenglättung zeichnen kann. Übergibt man der Funktion die Bitmaske `4095`, was binär zwölf Einsen entspricht, zeichnet sie einen Marker mit allen zwölf möglichen Pünktchen. Die Funktion akzeptiert als zusätzlichen Parameter auch einen Winkel, um den sie das ganze Muster dreht.

`get_point_id()` berechnet mit `generate_mask()` nun 60 verschiedene Masken für die Winkel von 0° bis 59°. Diese Masken multipliziert sie Pixel für Pixel mit dem Bild des Markers und summiert die Farbwerte aller Pixel. Geteilt durch die Summe aller (weißen) Pixel in der Maske ergibt das eine durchschnittliche Helligkeit für das maskierte Bild. Liegen die weißen Punkte in der Maske genau über schwarzen Punkten im Bild des Markers, ist die durchschnittliche Helligkeit kleiner als in Bildern, in denen die Maske helle Pixel ausstanzt.



Diese Ansicht produziert das Programm als Vorschau: Neben einigermaßen sicher erkannten Markern steht in Gelb die ID und in Pink die geschätzte Entfernung.

Mit diesem Winkel kann die Funktion anschließend prüfen, ob im Bild des Markers an der Position eines Pünktchens hauptsächlich dunkle oder mehr helle Pixel sind. Sie berechnet dafür wieder mit `generate_mask()` eine Maske, übergibt dieser aber ein Bitmuster mit nur einer 1. Die zwölf Bitmasken von `000000000001` bis `100000000000` lassen sich ganz leicht mit bitweisem Shift << berechnen:

```
for i in range(12):
  mask = generate_mask(imc.shape,
     ellipse[1][1] / 2, pattern_angle,
     bit_mask=1 << i)
```

Ums Maskieren und Addieren der Pixelwerte kümmert sich wie bereits bei der Winkelbestimmung Numpy:

```
  masked_sum = np.sum(imc * mask)
  sum_of_mask = np.sum(mask)
  sums[i] = masked_sum / sum_of_mask
```

Im `sums`-Array sammeln sich die Werte, sodass sich anschließend leicht ein Minimum und ein Maximum berechnen lässt:

```
min_sum = np.min(sums)
max_sum = np.max(sums)
```

Die Funktion geht nun davon aus, dass ein schwarzes Pünktchen vorhanden ist, wenn es näher am Minimum als am Maximum liegt. Auf dieser Entscheidungsgrundlage berechnet die Funktion eine Bitmaske. Da der Code aber keine Verdrehungen über 60° berücksichtigt hat, könnte es fünf weitere Bitmuster geben, die eine größere ID ergeben.

Musterverdreher

Damit um 60°, 120° oder 180° verdreht aufgenommene Muster die gleiche ID ergeben, gibt `get_point_id()` nur die größtmögliche ID aus. Um das sicherzustellen, verdreht `find_max_id_in_pattern()` das Bitmuster in alle sechs möglichen Positionen und gibt nur die größte ID zurück.

Dafür trennt die Funktion zunächst die unteren und oberen 6 Bit:

```
low = pattern & 63
high = (pattern & (63 << 6)) >> 6
```

Das bitweise Und mit der Maske 63 setzt alle Bits über dem sechsten auf 0.

Für die fünf weiteren Bitmuster müsste Python die Bits rollen, wofür es aber keinen Befehl gibt. Schiebt man die Bits, die man links herausschiebt, aber rechts wieder hinein, sorgt das für die gleiche Operation. Daneben muss man nur noch dafür sorgen, dass die so erzeugte Zahl nicht mehr als 6 Bit hat:

```
id_candidates=np.zeros(6,dtype=np.int)
id_candidates[0] = pattern
for i in range(1, 6):
  l = (low<<i)%(1<<6)+((low<<i)>>6)
  h = (high<<i)%(1<<6)+((high<<i)>>6)
  id_candidates[i] = l+(h<<6)
```

Zuletzt gibt sie nur noch die maximale ID zurück:

```
return np.max(id_candidates)
```

Unsicherheit

Mit dem beschriebenen Vorgehen berechnet `get_point_id()` für jedes quadratische Bild eine ID, auch wenn im Bild gar kein Zozo-Marker zu sehen ist. Um gegen solche Falsch-Positiven vorzugehen, gibt die Funktion neben der ID einen `confidence`-Wert zurück, der aussagt, wie sicher die Funktion ist, dass es sich bei dem Punkt um einen Marker handelt (1: sehr sicher, 0,5: eher unsicher, 0: fast sicher, dass es kein Marker ist).

```
confidence = 1
sc = (max_sum-min_sum)**2 / (
      800+(max_sum-min_sum)**2)
for i in range(12):
  dc = 1-(min(sums[i]-min_sum,
              max_sum - sums[i])**2)/(
         (max_sum-thresh)**2)
  confidence = min(confidence, dc*sc)
```

Die Idee hinter der Berechnung dieses Werts ist, dass ein ideal fotografierter Marker an allen Stellen mit schwarzem Pünktchen gleichmäßig dunkel und an allen Stellen ohne Pünktchen gleichmäßig hell ist. In diesem Fall ist `dc=1`. Je öfter die Helligkeit an den Stellen, an denen die Funktion schwarze Pünktchen erwartet, weder ganz hellen noch ganz dunklen Pünktchen entspricht, desto kleiner wird `dc`.

Damit weitgehend einfarbige Ellipsen keine hohe `confidence` erreichen können, gibt es zusätzlich noch `sc`. Dieser Wert ist 0, wenn `min_sum` und `max_sum` gleich groß sind. Er steigt aber recht schnell (bei `max_sum - min_sum = 28` ist er schon etwa 0,5) und nähert sich für kontrastreiche Bilder an 1 an.

Da schon einzelne Ausreißer zu einer falschen ID führen, zählt nur der kleinste `confidence`-Wert (berechnet als `dc * sc`) aller zwölf möglichen Pünktchenpositionen.

Kleine Punkte

Am Kragen, den Handgelenken und den Knöcheln hat der Zozo-Suit kleine Punkte mit einem Durchmesser von einem Zentimeter. Sie haben auch einen zwei Millimeter großen Mittelpunkt, sind aber nicht voneinander zu unterscheiden. Das Programm versucht, sie mit der Funktion `is_small_point()` zu erkennen. Sie erzeugt dafür ähnlich wie beim Berechnen der ID der großen Punkte eine Maske und vergleicht die Helligkeit des Mittelpunkts mit dem Ring darum. Sie gibt ähnlich wie die eben beschriebene `confidence` einen Wert zwischen 0 und 1 zurück, der beschreibt, wie sicher sie ist, dass eine Form ein solcher Punkt ist.

Ist sich `is_small_point()` zu mehr als 30 Prozent sicher, dass eine Form ein kleiner Punkt ist, ordnet das Programm ihm die ID 0 zu. Andernfalls versucht es, die ID eines großen Punkts zu bestimmen, was bei fehlerhaft erkannten Formen zu niedrigen `confidence`-Werten führt.

Entfernungsschätzung

Um den Abstand der Punkte zur Kamera zu schätzen, fehlt dem Programm ein absolutes Maß. Das ist aber nicht schlimm,

da die Entfernungswerte ohnehin sehr stark rauschen. Für die Schätzung sucht es sich zuerst die zehn großen Punkte mit der größten `confidence`. Diese Punkte wurden vermutlich ohne große Verzerrung aufgenommen. Der größere Durchmesser der Ellipse gibt daher recht genau die Größe des Punkts in Pixeln wieder. Die zehn Durchmesser mittelt das Programm und postuliert, dass ein Punkt mit dieser gemittelten Größe genau zwei Meter von der Kamera entfernt ist.

Aus dem Verhältnis der Größe jedes einzelnen Punkts zum eben berechneten Durchschnitt, multipliziert mit 200 (oder mit 400 für die kleinen Punkte), ergibt sich eine geschätzte Distanz des Punkts in Metern. Der berechnete Wert ist auf circa fünf Zentimeter genau, wobei es gerade bei schlecht erkannten Punkten Ausreißer gibt, die entfernter liegen.

Einsatz

Um unser Programm auszuprobieren, checken Sie einfach das Repository bei GitHub aus (siehe ct.de/yx6j), installieren die `requirements.txt` und führen `python detect_points.py IMG_20180911_163153_c.jpg` aus. Das mitgelieferte Foto können Sie durch ein eigenes ersetzen. Nach kurzer Rechenzeit erscheint ein Fenster mit einem Vorschaubild, das Sie mit der Esc-Taste schließen können. Neben der Vorschau im Fenster legt das Testprogramm die Bilddateien `point_positions.png` und `collected_points.png` an. Letzteres ist ein sehr breites Bild mit den verzerrten und unverzerrten Punkten und einer Zeichnung des zur erkannten ID passenden Punkts. Darunter steht die ID, und die `confidence`. Die Datei `point_positions.png` zeigt wie die Vorschau die erkannten Punkte im Kontext des Ursprungsbilds, bei Punkten mit einer `confidence` über 30 Prozent steht die ID und die geschätzte Entfernung daneben. Die Farbe der gezeichneten Ellipse gibt Auskunft über die `confidence`: Rote Umrandungen haben sehr kleine, grüne sehr große Zutrauenswerte.

Um unsere Marker-Erkennung in eigenen Projekten zu nutzen, rufen Sie einfach `detect_points()` auf und übergeben der Funktion das Farbbild. Die Koordinaten der Punkte gibt das Programm in Pixeln relativ zur linken oberen Ecke aus, die Entfernungen sind in Metern. Beispielsweise können Sie mit den erkannten Positionen ein parametrisiertes 3D-Modell fitten und mit dem die Magie der Zozo-App nachbauen. Oder Sie nehmen sich gleichzeitig aus mehreren Perspektiven auf und nutzen den Zozo-Suit für kostengünstiges Motion-Tracking. Lassen Sie es uns wissen, wenn Sie eine dieser Ideen umsetzen. Wir können es kaum erwarten, kreative Anwendungen für den Zozo-Suit kennenzulernen.

(pmk@ct.de) c't

Literatur

[1] Michael Link, Maßenfertigung, Immer gut sitzende Kleidung per Smartphone-App bestellen, c't 19/2018, S. 106

Quellcode: ***ct.de/yx6j***