Devlog - Computer Vision mit openCV.js

Mattis Thieme

November 2024

Inhaltsverzeichnis

1 Einleitung und Motivation

(Einleitung und Motivation)

In diesem Devlog möchte ich einen Überblick über die geläufigen Methoden der Kreiserkennung und Kreisklassifizierung aufzeigen, von einfachen und leicht verständlichen Verfahren wie der Geometrie- oder Farberkennung, bis hin zu den mächtigen Werkzeugen der Musteranalyse und Objekterkennung.

Alle Ansätze werde ich dabei stets an einem Beispiel demonstrieren. Damit Du leicht alles auf deinem eigenen Rechner selber ausprobieren kannst, testen wir alle Methoden an Hand von Euromünzen. Warum ausgerechnet Euromünzen? Sie eignen sich perfekt: sie sind alle rund, haben klar festgelegte Farben und Größen und fast jeder sollte welche zur Hand haben. Im Laufe des Devlogs werden wir somit eine Webanwendung entwickeln, welche auf eine angeschlossene Webkamera zugreift, die Bilddaten nach Münzen durchsucht, und jede gefundene Münze ihren 'Wert' zuteilt. Am Ende wird dann anschließend der Gesamtbetrag aller Münzen im Bild angezeigt.

2 Wahl der Technologien

Wenn Probleme im Bereich der Bildverarbeitung gelöst werden sollen, fällt die Wahl häufig auf OpenCV. Und das nicht ohne Grund: OpenCV bietet ein rießiges Spektrum an Funktionen und Algorithmen, von einfachen Bildoperationen hin zu ausgereiften Algorithmen der Gesichtserkennung, Bildsegmentierung und Objekterkennung. Auch Maschinelles Lernen und Deep Learning sind in OpenCV integriert.

Die Wahl der Bibliothek wäre somit schnell getroffen, wenn wir nicht noch ein weiteres Kriterium hätten: die Webanwendung. Da OpenCV jedoch ursprünglich in C++ geschrieben ist, ist das primäre Interface, mit welchem auf die Funktionaltiätenzugegriffen wird, auch in C++ verfasst. Es gibt zwar mit Java und Python auch noch weitere alternative Schnittstellen, jedoch soll unsere Webanwendung, wie bereits oben erwähnt, nicht auf einem Server laufen, sondern direkt im Browser des Clients. Die Lösung: openCV.js.

Als relativ neuer Bestandteil des openCV-Projektes, ist openCV.js eine JavaScript-Portierung der OpenCV-Bibliothek. Sie ermöglicht es, OpenCV-Funktionen direkt im Browser auszuführen, ohne dass der Nutzer eine zusätzliche Software installieren muss. Somit können wir die volle Bandbreite der OpenCV-Funktionen nutzen, ohne auf die Vorteile einer Webanwendung verzichten zu müssen.

3 Integration in Webanwendung

3.1 openCV.js

Als aller erstes brauchen wir natürlich eine aktuelle Version von openCV.js. Diese können wir direkt von der offiziellen openCV-Webseite herunterladen:

https://docs.opencv.org/4.10.0/opencv.js

Speichere die Datei in deinem Projektverzeichnis und binde sie in deiner HTML-Datei wie eine normale JavaScript-Datei ein.

3.2 Grundaufbau der Webseite

Für unsere Webanwendung benötigen wir zunächst eine simple HTML-Struktur mit mindestens zwei Elementen: einem Video-Element für den Kamerastream und einem Canvas-Element, auf welchem wir die Bildverarbeitungsergebnisse anzeigen können. Beide Elemente sollten idealerweise übereinander liegen und die selbe Größe haben.

Die html-Datei könnte also wie folgt aussehen:

```html
<!DOCTYPE html>
<html lang="de">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Video Player mit Canvas Overlay</title>
    <link rel="stylesheet" href="style.css">
    <script src="lib/opencv_4.10.0.js"></script>
</head>
<body>
<div id="mainContainer">
    <div id="textContainer">
        <h1>CoinFinder</h1>
        <p>Aktueller Wert: <span id="value">0</span>€</p>
    </div>
    <div class="container" id="videoContainer">
        <video id="video" width="720" height="540" autoplay muted loop></video>
        <canvas id="canvas" width="720" height="540"></canvas>
    </div>
</div>
</body>
</html>
```

Und die dazugehörige CSS-Datei:

```css
*{
    box-sizing: border-box;
}

html, body {
    background-color: #18204d;
    color: #ffffff;
    font-family: Arial, Helvetica, sans-serif;
}

#mainContainer{
    position: relative;
    width: 97vw;
    height: 97dvh;
    display: flex;
    flex-direction: column;
}

#textContainer{
    display: flex;
    align-items: center;
    margin: 0 1em 0 1em;
}

#videoContainer{
    position: relative;
    height: 100%;
    width: 100%;
    flex-grow: 1;
```

```
30      align-items: center;
31      justify-content: center;
32      display: flex;
33
34  }
35
36  #video, #canvas {
37      height: 100%;
38      width: 100%;
39      left: 0;
40      top: 0;
41      position: absolute;
42      aspect-ratio: inherit;
43      object-fit: contain;
44  }
45
46  #mainContainer{
47      border: 0.5em solid #ffa300;
48  }
49
50  #videoContainer{
51      border: 0.5em solid blue;
52  }
```

3.3 Grundlegende openCV Methoden

Zunächst benötigen wir eine Methode, um auf die Kamera des Nutzers zuzugreifen und den Videostream auf dem Video-Element anzuzeigen. Dafür können wir die getUserMedia-API verwenden. Zur Sicherheit führen wir Funktionen mit openCV-Funktionalitäten erst nach dem window.onload-Event aus, um sicherzustellen, dass alle Elemente, insbesondere die 10MB große openCV.js Bibliothek, vollständig geladen sind:

```
1       window.onload = () => {
2           video = document.getElementById('video');
3
4           // Webcam stream erhalten
5           navigator.mediaDevices.getUserMedia({
6               video: true,
7               audio: false
8           }).then(stream => {
9               video.srcObject = stream;
10              video.onloadedmetadata = () => {
11                  video.play();
12
13                  //start the main loop
14                  requestAnimationFrame(mainLoop);
15              };
16          }).catch(error => {
17              console.error('Error accessing the camera: ', error);
18          });
19      };
20  }
```

Um mit openCV arbeiten zu können. brauchen wir eine Möglichkeit die Bilddaten aus dem Video-Element zu extrahieren und in eine openCV-Matrix zu konvertieren. Dafür können wir die cv.imread()-Methode verwenden, jedoch braucht diese ein HTML-Canvas-Element als Argument. Wir können also nicht direkt das Video-Element übergeben, sondern müssen zuerst die Bilddaten auf ein Canvas-Element zeichnen. Dafür erstellen wir uns einen neuen Canvas, welcher nur im JavaScript-Code existiert und nicht Teil des DOM ist. Nachdem wir den aktuellen Frame auf das Canvas gezeichnet haben, können wir die

Bilddaten mit cv.imread() in eine openCV-Matrix konvertieren:

```
let tempCanvas = document.createElement('canvas');
let tempCtx= tempCanvas.getContext('2d', { willReadFrequently: true });
function GetFrame(){
    //Größe des temp. Canvas auf die Größe des output Canvas setzen
    tempCanvas.width = outputCanvas.width;
    tempCanvas.height = outputCanvas.height;

    //Bilddaten des Videos auf das temp. Canvas zeichnen
    tempCtx.drawImage(video, 0, 0, tempCanvas.width, tempCanvas.height);

    //Bilddaten in openCV-Matrix konvertieren und zurückgeben
    return cv.imread(tempCanvas);
}
```

Bevor wir nun mit der Kreiserkennung beginnen, sollten wir uns noch eine Methode schreiben, um die openCV-Matrix auf das Canvas-Element zu zeichnen. Dafür können wir die cv.imshow()-Methode verwenden. Auch diese benötigt wieder ein Canvas-Element als Argument, in unserem Fall nehmen wir unser bereits definierten outputCanvas, welcher über unserem Video-Element liegt. In der Regel benutzen wir eine Matrix nicht weiter, nachdem wir sie auf das Canvas gezeichnet haben, deshalb können wir sie in der Methode direkt wieder freigeben, um Speicherplatz zu sparen und Speicherlecks zu vermeiden:

```
function ShowFrame(inputMat){
    cv.imshow('canvas', inputMat); //Matrix auf Canvas zeichnen
    inputMat.delete(); //free memory
}
```

Hinweis zur Speicherfreigabe: openCV.js verwaltet den Speicher nicht automatisch, wie es bei JavaScript üblich ist. Das bedeutet, dass wir selbst dafür verantwortlich sind, den Speicher freizugeben, sobald wir ihn nicht mehr benötigen. Dies betrifft hauptsächlich Objekte vom Typ cv.Mat, welche wir mit der delete()-Methode freigeben können. Tun wir dies nicht, verbraucht der Browser mit jedem neuen Aufruf von new cv.Mat() oder cv.imread() mehr Speicher, bis irgendwann der Browsertab abstürzt. Sollte dein Programm nach einigen Sekunden oder Minuten aufhören zu funktionieren, könnte dies ein Hinweis auf ein Speicherleck sein. Schaue in diesem Fall in die Konsole nach einer entsprechenden Fehlermeldung.

Zu guter Letzt benötigen wir noch einen Hauptloop, welcher die Bilddaten aus dem Video-Element extrahiert, die Kreiserkennung durchführt und das Ergebnis auf dem Canvas-Element anzeigt. Dafür können wir die requestAnimationFrame()-Methode verwenden, welche uns eine optimale Bildwiederholrate garantiert. Rufe die Methode am Besten nach dem Laden der Kamera auf, um sicherzustellen, dass alle Elemente vollständig geladen sind:

```
function mainLoop() {
    let currentFrame = GetFrame();

    //Unsere Bildverarbeitungsmethoden kommen hier rein

    ShowFrame(currentFrame);
    requestAnimationFrame(mainLoop); //Funktion wiederholen
}
```

Nun sind wir bereit, mit der Kreiserkennung zu beginnen. Im nächsten Abschnitt werden wir uns die Circle Hough Transform genauer ansehen und sie auf unser Beispiel anwenden.

4 Kreiserkennung

4.1 Circle Hough Transform

4.2 Parameter und Einstellungen

5 Kreis-Klassifizierung

5.1 Geometrie

5.2 Farbanalyse

5.2.1 Durchschnitt und Helligkeit

5.2.2 Histogrammvergleich

5.2.3 Farbverlauf-Analyse

5.3 Musteranalyse

5.4 Objekterkennung

5.5 Machinelles Lernen

6 Limitationen

7 Fazit

Die Möglichkeit von openCV.js, OpenCV-Funktionen direkt im Browser des Clients auszuführen, eröffnet eine Vielzahl von neuen Anwendungsmöglichkeiten. Rechenintensive Bildverarbeitungsalgorithmen müssen nun nicht mehr auf einem Server laufen, wodurch Kapazitäten frei werden und für andere Aufgaben genutzt werden können. Immer mehr Onlinebesuche finden auf mobilen Geräten statt - die Verwendung der eingebauten Kamera für Bildverarbeitungsaufgaben könnte somit in Zukunft eine wichtige Rolle spielen. (Beispiele einfügen?)