Devlog - Computer Vision mit openCV.js

Mattis Thieme

November 2024

Inhaltsverzeichnis

1 Einleitung

Probleme der Bilderkennung und Bildverarbeitung gewinnen immer mehr an Relevanz. Kein Wunder: Mit der stetigen Verbreitung von mobilen Endgeräten haben immer mehr Systeme Zugriff auf eine (ziemlich hochwertige) Kamera. Die Möglichkeiten, die sich dadurch ergeben, sind vielfältig: über das Scannen von Dokumenten, hin zur Klassifizierung von Tieren und Pflanzen bis hin zum industriellen Einsatz in der Qualitätskontrolle.

Jedoch geht der Trend auch in die Richtung von Onlineservices und Webanwendungen. Programme die früher noch installiert werden mussten, laufen nun direkt im Browser des Clients. Die Gründe sind vielfältig: volle Platformunabhängigkeit, leichtes Einspielen von Updates und keine Installation von zusätzlicher Software. Aber wie sieht es mit Bildverarbeitung und Bilderkennung im Webkontext aus? Die bekannteste Bibliothek für Bildverarbeitung, OpenCV, ist ursprünglich in C++ geschrieben und muss somit auf Serverseite laufen. Aber nicht immer ist die Option sinnvoll oder vorhanden. Nicht immer steht die benötigte Rechenleistung zur Verfügung, auch die Latenz kann letztendlich ein Problem darstellen. Ist es möglich diese Probleme auch im Browser des Clients zu lösen? Die Antwort: Ja! Mit openCV.js.

Ich starte diesen DevBlog um mich genau mit genau dieser Bibliothek auseinanderzusetzen und meine Erfahrungen und Erkenntnisse hier zu teilen. Ich werde genau das gerade erwähnte Problem lösen: Bildverarbeitung direkt im Browser des Clients. Aber was genau möchte ich nun erkennen oder verarbeiten?

Mein Ziel ist eine Webanwendung, welche Münzen erkennen kann und diese anschließend ihren Wert zuordnet. Warum ausgerechnet Münzen? Zum einen ist es ein recht komplexes Problem, welches gleich viele verschiedene Aspekte der Bildverarbeitung auf Einaml abdeckt, zum anderen ist es eine Problemstellung, welches sich leicht auf einem anderen System reproduzieren lässt. Alles was du brauchst, ist lediglich eine Webkamera und ein paar Münzen.

Ich habe viele verschieden Ansätze und Ideen, wie ich dieses Problem lösen könnte. Von einfachen Bildoperationen, über die Kreiserkennung bis hin zur Objekterkennung und Musteranalyse. Ob sie alle funktionieren und erfolgreich sind? Keine Ahnung! Jedoch werde ich meine Fortschritte und Erkenntnisse in diesem Blog festhalten und stets meinen Programmcode teilen. Und vielleicht kann ich dir sogar helfen, wenn du vor einem ähnlichen Problem stehen solltest!

Interessiert? Dann lass uns anfangen!

2 Motivation

Aber warum das ganze? Einst musste ich (wie du vielleicht auch) ein Problem der Bildverarbeitung lösen, welches zwingend im Webkontext stattfinden sollte. Die Anforderungen waren klar: die Bildverarbeitung sollte direkt im Browser des Clients stattfinden, ohne dass der Nutzer eine zusätzliche Software installieren muss.

Die Bibliothek openCV war natürlich die erste Wahl, jedoch stand ich nun genau vor diesem Problem: wie bekomme ich openCV, eine Bibliothek, die ursprünglich in C++ geschrieben ist, in meiner Webanwendung zum Laufen? Meine Lösung war natürlich openCV.js.

Obwohl es sich zunächst als die beste Lösung angehört hat, war die Nutzung von openCV.js jedoch kein Selbstläufer. Wie du wahrscheinlich bereits mit Schrecken festgestellt hast, ist die offizielle Dokumentation nur für die C++ Schnittstelle geschrieben, und Tutorials für openCV.js sind leider rar gesät. Die JavaScript-SChnittstelle von openCV ist nun mal die neuste Änderung des mittlerweise 20 Jahre

alten Projektes und somit noch nicht so ausgereift und erprobt wie die anderen Schnittstellen.

Genau aus diesen Gründen habe ich mich für das Schreiben dieses Devlogs entschieden. Ich möchte meine Erfahrungen und Erkenntnisse teilen, um anderen Entwicklern zu helfen, die vor dem gleichen Problem stehen. Ich möchte zeigen, dass es gar nicht so kompliziert ist, openCV.js in einer Webanwendung zu nutzen und wie mächtig die Bibliothek tatsächlich ist. Ja es gibt ein paar Eigenheiten und Macken, aber genau diese werde ich erläutern, damit du nicht die gleichen Fehler machst wie sie ich einst gemacht habe.

3 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.

4 Integration in Webanwendung

4.1 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.

4.2 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:

```
1 <!DOCTYPE html>
```

```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;
    align-items: center;
    justify-content: center;
    display: flex;

}

#video, #canvas {
    height: 100%;
    width: 100%;
    left: 0;
    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  }
```

4.3 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:

```
1   let tempCanvas = document.createElement('canvas');
2   let tempCtx= tempCanvas.getContext('2d', { willReadFrequently: true });
3   function GetFrame(){
4       //Größe des temp. Canvas auf die Größe des output Canvas setzen
5       tempCanvas.width = outputCanvas.width;
6       tempCanvas.height = outputCanvas.height;
7
8       //Bilddaten des Videos auf das temp. Canvas zeichnen
9       tempCtx.drawImage(video, 0, 0, tempCanvas.width, tempCanvas.height);
```

```
10
11      //Bilddaten in openCV-Matrix konvertieren und zurückgeben
12      return cv.imread(tempCanvas);
13  }
```

Bevor wir nun mit der Kreiserkennung beginnen, sollten wir uns noch eine Methode schreiben, um die Änderungen die wir machen auch in unserer Webanwendung anzuzeigen. Dafür können wir die cv.imshow()-Methode verwenden. Auch diese benötigt wieder ein Canvas-Element als Argument, in unserem Fall nehmen wir unseren bereits definierten outputCanvas, welcher direkt 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:

```
1  function ShowFrame(inputMat){
2      cv.imshow('canvas', inputMat); //Matrix auf Canvas zeichnen
3      inputMat.delete(); //free memory
4  }
```

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:

```
1  function mainLoop() {
2      let currentFrame = GetFrame();
3
4      //Unsere Bildverarbeitungsmethoden kommen hier rein
5
6      ShowFrame(currentFrame);
7      requestAnimationFrame(mainLoop); //Funktion wiederholen
8  }
```

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.

5 Kreiserkennung

5.1 Circle Hough Transform

5.2 Parameter und Einstellungen

6 Kreis-Klassifizierung

6.1 Geometrie

6.2 Farbanalyse

6.2.1 Durchschnitt und Helligkeit

6.2.2 Histogrammvergleich

6.2.3 Farbverlauf-Analyse

6.3 Musteranalyse

6.4 Objekterkennung

6.5 Machinelles Lernen

7 Limitationen

8 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?)