

Devlog - Computer Vision mit openCV.js

Mattis Thieme

November 2024

Inhaltsverzeichnis

1 Einleitung	4
1.1 Motivation	4
2 Erstellen der Webseite	6
2.1 openCV.js	6
2.2 Grundaufbau der Webseite	6
2.3 Grundlegende openCV Methoden	8
3 Eingriffe zur Laufzeit	10
3.1 Die Bedeutung von einfachem Testing	10
3.2 Variablen-Slider	10
3.3 Cookies	12
4 Kreiserkennung	15
4.1 Funktionsweise der HoughCircles-Methode	15
4.2 Buffering der Kreise	17
5 Template Matching	18
5.1 Überlegungen	18
5.2 Die Vorlagen	18
5.3 Funktionsweise	20
5.4 Exkurs: Mehrere Matrizen in einem Canvas anzeigen	21
5.5 Erste Ergebnisse	22
6 Anpassungen	24
6.1 Kantenerkennung	24
6.1.1 Funktionsweise der Canny-Methode	24
6.1.2 Die korrekten Parameter finden	24
6.2 Matrizen rotieren und transformieren	24
6.3 Neues Problem: Mat-Types	24
6.4 Asynchronität und Ladebalken	24
7 Ergebnisse	25
7.1 Erste Ergebnisse	25

7.2	Zuschneiden der Bilder	25
7.3	Weitere Anpassungen	25
7.4	Finale Ergebnisse	25
8	Fazit	26
8.1	Zusammenfassung	26
8.2	Die größten Probleme	26
8.3	Ausblick	26

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 Einmal abdeckt, zum anderen ist es eine Problemstellung, welches sich leicht auf einem anderen System reproduzieren lässt. Alles was du brauchst, ist lediglich eine Webcam und ein paar Münzen.

Ich habe viele verschiedene 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!

1.1 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.

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 Funktionalitäten zugegriffen 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.

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 mittlerweile 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.

2 Erstellen der Webseite

2.1 openCV.js

Als aller erstes braucht es natürlich eine aktuelle Version von openCV.js. Diese kann direkt von der offiziellen openCV-Webseite herunterladen:

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

Es handelt sich hierbei um eine mittels Emscripten kompilierte Version von OpenCV, welche in JavaScript ausgeführt werden kann. Da der Programmcode nur in Bytecode zur Verfügung steht, ist es nicht möglich, den Code direkt zu lesen oder zu verändern. Dies führt zu dem Problem, dass IDE-Funktionalitäten wie Autovervollständigung oder Syntax-Highlighting leider nicht verfügbar sind.

2.2 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.

Nach einigen Tests habe ich mich für folgende Struktur entschieden:

```
1 <!DOCTYPE html>
2 <html lang="de">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Video Player mit Canvas Overlay</title>
7   <link rel="stylesheet" href="style.css">
8   <script src="lib/opencv_4.10.0.js"></script>
9 </head>
10 <body>
11 <div id="mainContainer">
12   <div id="textContainer">
13     <h1>CoinFinder</h1>
14     <h3>Aktueller Wert: <span id="value">0</span></h3>
15   </div>
16   <div class="container" id="videoContainer">
17     <video id="video" width="720" height="540" autoplay muted loop></video>
18     <canvas id="outputCanvas" width="720" height="540"></canvas>
19   </div>
20 </div>
21 </body>
22 </html>
```

Und die dazugehörige CSS-Datei:

```
1 *{
2   box-sizing: border-box;
3 }
4
5 html, body {
6   background-color: #18204d;
7   color: #ffffff;
8   font-family: Arial, Helvetica, sans-serif;
9 }
10
11 h3{
```

```

12         margin: 0 0 0 1em;
13     }
14
15     #mainContainer{
16         position: relative;
17         width: 97vw;
18         height: 97dvh;
19         display: flex;
20         flex-direction: column;
21     }
22
23     #textContainer{
24         display: flex;
25         align-items: center;
26     }
27
28     #textContainer > *{
29         margin: 0 1em 0 1em;
30     }
31
32     button{
33         background-color: #ffa300;
34         color: #18204d;
35         border: none;
36         padding: 0.5em 1em;
37         font-size: 1em;
38         cursor: pointer;
39     }
40
41     #videoContainer{
42         position: relative;
43         height: 100%;
44         width: 100%;
45         flex-grow: 1;
46         align-items: center;
47         justify-content: center;
48         display: flex;
49     }
50
51
52     #video, #outputCanvas {
53         height: 100%;
54         width: 100%;
55         left: 0;
56         top: 0;
57         position: absolute;
58         aspect-ratio: inherit;
59         object-fit: contain;
60     }
61
62     #outputCanvas{
63         image-rendering: pixelated;
64     }
65
66     #mainContainer{
67         border: 0.5em solid #ffa300;
68     }
69
70     #videoContainer{
71         border: 0.5em solid blue;
72     }

```

Nun haben wir ein Canvas-Element, welches über dem Video-Element liegt und exakt die selbe Größe

hat. Dies ermöglicht es mir, die Bildverarbeitungsergebnisse direkt auf dem dem Canvas zu zeichnen und sie dem Nutzer anzuzeigen. Der Canvas hat zudem die Eigenschaft `image-rendering: pixelated` um auf kleinere Bilder scharf zu zeichnen.

Als nächstes braucht es nun grundlegende Methoden, um auf die Kamera des Nutzers zuzugreifen und den Videostream auf dem Video-Element anzuzeigen.

2.3 Grundlegende openCV Methoden

Um auf die Kamera des Nutzers zuzugreifen, kann man unter anderem ein openCV-VideoCapture-Objekt verwenden. Dieses Objekt kann entweder direkt auf ein Video-Element zugreifen oder auf eine Videodatei. Da wir in unserem Fall den Kamerastream verwenden wollen, greifen wir direkt auf das Video-Element zu. Für den Zugriff auf die Webkamera benötigen wir die `getUserMedia()`-Methode des MediaDevices-Interfaces. Um sicherzustellen, dass openCV vollständig geladen ist, warten wir auf das "loadEvent" des Fensters und initialisieren erst dann die Kamera:

```

1  window.addEventListener("load", function () {
2      video = document.getElementById('video');
3      outputCanvas = document.getElementById('outputCanvas');
4      videoContainer = document.getElementById('videoContainer');
5
6      // get the camera
7      navigator.mediaDevices.getUserMedia({
8          video: true,
9          audio: false
10     }).then(stream => {
11         video.srcObject = stream;
12         video.onloadedmetadata = () => {
13             video.play()
14
15             //print camera stats
16             console.log("Camera resolution: " + video.videoWidth + "x" + video.
17                         videoHeight);
18             console.log("Camera frame rate: " + stream.getVideoTracks()[0].
19                         getSettings().frameRate+ " fps");
20
21             //initialize the inputMat-Matrix
22             inputMat = new cv.Mat(video.height, video.width, cv.CV_8UC4);
23             guiMat = new cv.Mat(video.height, video.width, cv.CV_8UC4);
24             videoCapture = new cv.VideoCapture(video);
25
26             //check if everything is loaded
27             cameraLoaded = true;
28             console.log("Camera loaded");
29             CheckIfLoadingFinished();
30     });
31 }).catch(error => {
32     console.error('Error accessing the camera: ', error);
33 });
34 }
```

Nachdem die Kamera initialisiert ist, kann die `read()`-Methode des VideoCapture-Objekts verwendet werden, um den aktuellen Frame des Videostreams in eine openCV-Matrix zu konvertieren.

Nun brauchen wir jedoch noch eine Methode, um eine openCV-Matrix auf einem Canvas ausgeben zu können. Dafür können wir die `cv.imshow()`-Methode verwenden, welche eine Matrix auf ein Canvas-Element zeichnet. In unserem Fall verwenden wir das `outputCanvas`-Element, welches über dem Video-Element liegt. Die Funktion sieht wie folgt aus:

```

1 function ShowMatrix(src, canvas){
2     cv.imshow(canvas, src);
3 }

```

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 Browertab 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. Ich habe zusätzlich einen Bool "loopActive" hinzugefügt, um den Loop bei Bedarf auch nur einmalig ausführen zu können. Der Loop sieht wie folgt aus:

```

1 let waitingForAnimationFrame = false;
2 let angle = 0;
3 function mainLoop() {
4     if(!loadingFinished){
5         console.warn("Can't start the loop because something is not loaded yet");
6         return;
7     }
8
9     waitingForAnimationFrame = false;
10
11    console.log("--- loop started");
12
13    videoCapture.read(inputMat);
14    videoCapture.read(guiMat);
15
16    //do something with the matrix
17
18    ShowMatrix(guiMat, outputCanvas);
19
20    if(loopActive){
21        waitingForAnimationFrame = true;
22        requestAnimationFrame(mainLoop);
23    }
24
25    console.log("--- loop ended");
26 }

```

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.

3 Eingriffe zur Laufzeit

3.1 Die Bedeutung von einfachem Testing

Meine nächste Aufgabe, die Implementierung der Kreiserkennung, erfordert eine Menge an Feintuning und Anpassungen. Jeder Parameter muss sorgfältig gewählt werden, um ein optimales Ergebnis zu erzielen. Zudem kann ich vorhinein nicht ermittelt werden, welche Parameter die besten sind. Es ist also ein iterativer Prozess, bei dem ich die Parameter anpasse, das Ergebnis betrachte und dann erneut die Parameter anpasse.

Nun stelle man sich vor, man müsste für jede kleine Änderung die Webseite neu laden, die Kamera neu ausrichten und die Münzen neu platzieren. Um diesen Prozess deutlich zu beschleunigen, habe ich mich dazu entschieden, einige Eingriffe zur Laufzeit zu implementieren. Diese Eingriffe sollen es mir ermöglichen, die Parameter der Kreiserkennung und weiterer openCV Funktionen direkt zur Laufzeit zu verändern, ohne die Webseite neu laden zu müssen, und somit schnell die besten Parameter zu finden.

3.2 Variablen-Slider

Meine Idee war es, für jede Variable im JavaScript-Code einen Slider zu erstellen zu können, mit dem ich schnell und einfach den Wert der Variable verändern kann. Dieser Slider soll dann den Wert der Variable anzeigen und bei Veränderung den Wert der Variable ändern.

Für die Sliders habe ich die Bibliothek [noUISlider](#) verwendet. Über die data-Attribute von den HTML-Elementen kann unkompliziert der Wertebereich und die Schrittweite des Sliders festgelegt werden.

Nun muss aber standardmäßig ein Event-Listener für jeden Slider erstellt werden, der bei Veränderung den Wert einer Variable ändert. Da ich jedoch nicht für jede Variable einen eigenen Event-Listener erstellen möchte, habe ich mich für einen anderen Ansatz entschieden. Stattdessen soll man direkt im HTML-Element des Sliders angeben können, welche Variable durch diesen Slider verändert werden soll.

Hierfür benötigen wir zum Ersten ein eigenes data-Attribute von HTML, in welchem der Name der zu verändernden Variable angegeben wird. Zum Zweiten benötigen wir eine Funktion, die alle Slider-Elemente durchgeht und für jedes Element den Event-Listener erstellt. Anschließend braucht es noch eine Möglichkeit den String-Namen der Variable in eine Referenz auf die Variable umzuwandeln.

Die Event-Listener Funktion sieht dann wie folgt aus:

```
1 function InitSliders(){
2     // Alle Slider-Container selektieren
3     const sliderContainers = document.querySelectorAll('.sliderContainer');
4
5     sliderContainers.forEach(container => {
6         const sliderElement = container.querySelector('.slider');
7         const valueElement = container.querySelector('.sliderValue');
8
9         const min = parseFloat(container.dataset.min);
10        const max = parseFloat(container.dataset.max);
11        const step = parseFloat(container.dataset.step);
12        const var1 = container.dataset.var1;
13        const var2 = container.dataset.var2;
14        const isRange = container.dataset.range === "true"; // Überprüft, ob
15        // Bereichsmodus aktiv ist
16        //console.log("Data for slider: min: " + min + " max: " + max + " step: " +
17        //           step + " var1: " + var1 + " var2: " + var2 + " isRange: " + isRange);
18
19        // Startwerte auslesen
```

```

18     let startValues = [];
19     if (isRange) {
20         // Bereichsmodus: Startwerte aus den <span>-Elementen lesen
21         const spanValues = valueElement.querySelectorAll('span');
22         startValues = Array.from(spanValues).map(span => parseFloat(span.
23            textContent));
24         if (startValues.length !== 2) {
25             // Fallback: Standardwerte in der Mitte des Bereichs
26             startValues = [min + (max - min) / 3, max - (max - min) / 3];
27         }
28     } else {
29         // Einzelregler: Einzelwert auslesen
30         startValues = [parseFloat(valueElement.textContent) || (min + max) /
31             2];
32     }
33
34     // Slider erstellen
35     noUiSlider.create(sliderElement, {
36         start: startValues,
37         range: {
38             'min': min,
39             'max': max
40         },
41         step: step,
42         connect: isRange ? true : [true, false] // Verbindet die Regler bei
43             Range-Modus
44     });
45
46     SetSliderValueFromCookie(container.id);
47
48     // Update-Event
49     sliderElement.noUiSlider.on('update', (values, handle) => {
50         if (isRange) {
51             // Bereichsmodus: Werte in <span>-Elementen aktualisieren
52             if(step >= 1){
53                 valueElement.textContent = values.map(value => Math.round(value
54                     )).join(' - ');
55             }else{
56                 valueElement.textContent = values.map(value => value).join(' -
57                     ');
58             }
59
60             //TODO: Beide Variablen aktualisieren
61             if(var1 !== undefined && var2 !== undefined){
62                 UpdateVariable(var1, values, 0);
63                 UpdateVariable(var2, values, 1);
64             }else{
65                 console.log("var1 or var2 is undefined. Slider will not change
66                     any variables");
67             }
68
69         } else {
70             // Einzelregler: Textinhalt aktualisieren
71             if(step >= 1){
72                 valueElement.textContent = Math.round(values[handle]);
73             }else{
74                 valueElement.textContent = values[handle];
75             }
76
77             if(var1 !== undefined){
78                 // Variable aktualisieren
79                 UpdateVariable(var1, values, handle);
80             }else{

```

```

76         console.log("var1 is undefined. Slider will not change any
77             variables");
78     }
79 }
80
81     // Cookie setzen
82     SetSliderCookie(container.id, values);
83 );
84 );
85
86 console.log("Sliders initialized");
87 }

```

Die Funktion `UpdateVariable()` wandelt schließlich den String-Namen der Variable in eine Referenz um und ändert den Wert der Variable. Hierfür wird die `eval()`-Funktion verwendet, die den String als JavaScript-Code interpretiert.

```

1 function UpdateVariable(varName, values, handle){
2     //check if variable exists
3     if (eval('typeof ' + varName) === 'undefined') {
4         console.warn('Variable '+varName+' does not exist');
5         return;
6     }
7
8     //update variable
9     if(eval('typeof ' + varName) === 'number'){
10         eval(varName + ' = ' + parseFloat(values[handle]));
11     }else if(eval('typeof ' + varName) === 'string'){
12         eval(varName + ' = ' + values[handle]);
13     }
14 }

```

Wie du vielleicht gesehen hast, gibt es auch noch die Funktionen `SetSliderValueFromCookie()` und `SetSliderCookie()`. Diese Funktionen dienen dazu, die Werte der Slider in Cookies zu speichern und beim Laden der Webseite wiederherzustellen. So behalten die Slider ihre eingestellten Werte auch nach einem Neuladen der Webseite.

3.3 Cookies

Cookies sind kleine Textdateien, die auf dem Computer des Benutzers gespeichert werden. Sie dienen dazu, Informationen über den Benutzer zu speichern, um die Benutzererfahrung zu verbessern. In meinem Fall möchte ich die Werte der Slider speichern, um sie beim Neuladen der Webseite wiederherzustellen.

Jedoch ist das Speichern von Werten in Cookies nicht ganz trivial: Pro Cookie kann nur ein einziger String mit einer maximalen Länge von 4096 Bytes gespeichert werden. Da die Werte der Slider jedoch Arrays von Zahlen sind, müssen diese erst in einen String umgewandelt werden. Hierfür habe ich mich für die `JSON.stringify()`-Funktion entschieden, die ein JavaScript-Objekt in einen JSON-String umwandelt. Die Funktion zum schreiben des Cookies sieht dann wie folgt aus:

```

1 function SetSliderCookie(sliderId, value) {
2     // Holen des bestehenden Cookie-Werts
3     let sliderValues = GetSliderCookie();
4
5     // Setze den Wert des Sliders im Cookie
6     sliderValues[sliderId] = value;
7
8     // Setze den Cookie mit den neuen Werten
9     const expires = new Date();

```

```

10     expires.setDate(expires.getDate() + 7); // Cookie läuft in 7 Tagen ab
11     document.cookie = `sliderValues=${JSON.stringify(sliderValues)}; expires=${
12         expires.toUTCString()}; path=/;`;

```

Zunächst erstelle ich ein Objekt `SliderValues` aus dem Cookie, falls es bereits existiert. Anschließend füge ich den neuen Wert des Sliders hinzu und speichere das Objekt als JSON-String im Cookie. Hierfür muss zudem ein Ablaufdatum für den Cookie festgelegt werden, in meinem Fall läuft der Cookie nach 7 Tagen ab. Der Cookie sieht dann wie folgt aus:

```

1 {"slider1": ["5100.00"], "slider2": ["3900.00"], "slider3": ["7.00"], "slider4": ["3.00"],
  "sliderValue3": ["230.00"], "sliderValue4": ["129.00"]}

```

Um die Daten aus dem Cookie wiederherzustellen, muss in der Funktion `GetSliderCookie()` der Cookie ausgelesen und in ein JavaScript-Objekt umgewandelt werden. Eine saubere Lösung, welche selbst beim Vorhandensein von weiteren Cookies funktioniert, könnte wie folgt aussehen:

```

1 function GetSliderCookie(){
2     const value = `; ${document.cookie}`;
3     const parts = value.split(`; sliderValues=`);
4     if (parts.length === 2) {
5         const cookieValue = parts.pop().split(',').shift();
6         return JSON.parse(cookieValue); // Parsen des JSON-Strings
7     }
8     return {}; // Falls der Cookie nicht existiert, ein leeres Objekt zurückgeben
9 }

```

Lass mich diese Funktion kurz erklären: Zunächst wird der Cookie-Wert aus dem `document.cookie`-String extrahiert. Die „cookie“ Eigenschaft des Objekts „`document`“ enthält alle Cookies, die auf der aktuellen Seite gesetzt wurden. Dem String wird ein Semikolon und ein Leerzeichen vorangestellt, um sicherzustellen, dass der Cookie-Wert korrekt extrahiert wird, selbst wenn noch andere Cookies vorhanden sind. Nun wird über `value.split('; sliderValues=')` der String zweiseitig geteilt, der erste Teil, welcher alles vor dem Cookie-Wert enthält, wird per `pop()` verworfen. Der zweite Teil, welcher den Cookie-Wert inklusive möglicher weiterer Cookies enthält, muss noch weiter verarbeitet werden. Hierfür wird der String per `split(',')` wieder in ein Array geteilt und mit `shift()` das erste Element, welches den Cookie-Wert enthält, extrahiert. Nun erst können wir sicher sein, dass mögliche Cookies vor und nach unserem Cookie-Wert entfernt wurden. Der Cookie-Wert wird schließlich per `JSON.parse()` in ein JavaScript-Objekt umgewandelt und zurückgegeben.

Zu guter Letzt braucht es noch eine Funktion, welche beim Start der Webseite die Werte der Slider aus dem Cookie wiederherstellt. Diese Funktion sieht wie folgt aus:

```

1 function SetSliderValueFromCookie(containerID){
2     const sliderValues = GetSliderCookie();
3     const sliderContainer = document.getElementById(containerID);
4
5     if (!sliderContainer) {
6         console.error('Slider container not found');
7         return;
8     }
9
10    const slider = sliderContainer.querySelector('.slider');
11    const sliderValueElement = sliderContainer.querySelector('.sliderValue');
12
13    if (!slider || !sliderValueElement) {
14        console.error('Slider or value element not found');
15        return;
16    }
17
18    const savedValue = sliderValues[containerID];

```

```

19
20     if (savedValue !== undefined) {
21         // Den Slider auf den gespeicherten Wert setzen
22         slider.noUiSlider.set(savedValue);
23
24         // Anzeige des aktuellen Werts aktualisieren
25         sliderValueElement.textContent = Array.isArray(savedValue) ? savedValue.
26             join(' - ') : savedValue;
27     } else{
28         console.log("No saved value for " + containerID + " found in cookie");
29     }

```

Nun habe ich endlich eine solide Möglichkeit geschaffen, schnell und unkompliziert Variablen zur Laufzeit zu verändern und diese Änderungen sogar über einen Neustart der Webseite hinweg zu speichern. Dies wird mir hoffentlich eine Menge Zeit und Nerven sparen, wenn ich nun als nächstes mit der Implementierung der Kreiserkennung beginne.

4 Kreiserkennung

4.1 Funktionsweise der HoughCircles-Methode

Das Implementieren der Kreiserkennung in openCV.js war tatsächlich ziemlich einfach. Alles was es grundsätzlich dafür braucht ist diese Methode:

```
1 cv.HoughCircles(image, circles, method, dp, minDist, param1, param2, minRadius,  
    maxRadius);
```

Gehen wir erstmal die einzelnen Parameter durch:

image - Dies ist das Bild, in dem die Kreise gefunden werden sollen. Es muss eine openCV-Matrix sein und sollte als Graustufenbild vorliegen.

circles - In diese Matrix werden die gefundenen Kreise gespeichert. Ja - die Methode schreibt die Kreise direkt in eine openCV-Matrix. In OpenCV können Matrizen nicht nur für die Speicherung von Bildern verwendet werden, sondern auch für die Speicherung von Vektoren. In diesem Fall wird in der Matrix „circles“ in jeder Zeile ein Kreis gespeichert. Ein Kreis ist dabei ein Vektor der Form $[x, y, r]$, wobei (x, y) die Koordinaten des Mittelpunkts des Kreises und r der Radius des Kreises sind.

Weiter unten zeige ich eine Möglichkeit, wie die einzelnen Kreise in dieser Matrix in Objekte umgewandelt werden können. Dies macht die weitere Verarbeitung der Kreise deutlich einfacher.

method - Dies ist eine Konstante, die angibt, welche Methode zur Kreiserkennung verwendet werden soll. In openCV.js gibt es aktuell nur eine Methode, die verwendet werden kann, nämlich `cv.HOUGH_GRADIENT`.

dp - Hier wird es leider etwas komplizierter. Der Parameter `dp` ist ein inverser Skalierungsfaktor, der die Genauigkeit der Kreiserkennung beeinflusst. Ein Wert von 1 bedeutet, dass die Auflösung des Eingabebildes verwendet wird. Ein Wert von 2 bedeutet, dass das Eingabebild halb so groß wie das Originalbild ist. Je kleiner der Wert, desto genauer ist die Kreiserkennung, aber auch desto langsamer.

minDist - Dieser Parameter gibt den minimalen Abstand zwischen den Mittelpunkten der gefundenen Kreise an. Dies ist sehr nützlich um die Anzahl an falschen Positiven zu reduzieren.

param1 - Dieser Parameter ist spezifisch für die Methode `cv.HOUGH_GRADIENT`. Es handelt sich hier um den oberen Schwellwert für die Kantenerkennung, welche intern durchgeführt wird. Der untere Schwellwert ist stets die Hälfte dieses Wertes.

param2 - Dieser Parameter ist ebenfalls spezifisch für die Methode `cv.HOUGH_GRADIENT`. Er ist der Schwellwert für die Kreiserkennung. Je kleiner der Wert, desto empfindlicher ist die Kreiserkennung, d.h. ein Kreis muss weniger stark ausgeprägt sein, um trotzdem als solcher erkannt zu werden. Dies bedeutet jedoch auch, dass das mit kleinerem Wert das Risiko für falsche Kreise steigt.

Der Einfachheit halber habe ich eine eigene Methode nur für das Finden von Kreisen geschrieben, die die Methode `cv.HoughCircles` verwendet. Diese Methode gibt dann ein Array von Kreis-Objekten zurück, die ich dann weiterverarbeiten kann. Die Methode sieht so aus:

```
1 function FindCircles(inputMat, guiMat){  
2  
3     //Eingabe-Matrix in Graustufen umwandeln  
4     cv.cvtColor(inputMat, grayMat, cv.COLOR_RGBA2GRAY);
```

```

5
6     //reset circlesMat
7     circlesMat.delete();
8     circlesMat = new cv.Mat();
9
10    //Hough-Transformation
11    cv.HoughCircles(grayMat, circlesMat, cv.HOUGH_GRADIENT, dp, minRadius, param1,
12                      param2, minRadius, maxRadius);
13
14    //create circle objects
15    let foundCircles = [];
16    for (let i = 0; i < circlesMat.cols; ++i) {
17        let x = circlesMat.data32F[i * 3];
18        let y = circlesMat.data32F[i * 3 + 1];
19        let radius = circlesMat.data32F[i * 3 + 2];
20
21        foundCircles.push(new Circle(x, y, radius));
22    }
23
24    //draw circles
25    for(let i = 0; i < foundCircles.length; i++){
26        DrawCircle(foundCircles[i], guiMat, [255,255,255,255]);
27    }
28
29    return foundCircles;
}

```

Auch für das Anzeigen der Kreise habe ich eine ausgelagerte Methode geschrieben. Dies wäre eigentlich nicht nötig gewesen, da für das Zeichnen von Kreisen nur eine einziger Methodenaufruf benötigt wird. Ich habe die Methode dennoch geschrieben, um die Lesbarkeit des Codes zu erhöhen. Die Methode sieht so aus:

```

1 function DrawCircle(circle, guiMat, color){
2     //return if parameter is not set
3     if(circle === undefined){
4         console.error("Circle is undefined");
5         return;
6     }
7
8     //draw circle
9     cv.circle(guiMat, new cv.Point(circle.x, circle.y), circle.radius, color, 2);
10 }

```

Bei den Parametern `dp`, `param1` und `param2` stellte sich mir nun die Frage, welche Werte ich hier verwenden soll. Eine pauschale Antwort darauf gibt es nicht, da die optimalen Werte stark vom Bild, vom Verwendungszweck und in meinem Beispiel auch von der Kamera abhängen.

Meine nächste Aufgabe war es nun, für meinen Anwendungsfall die optimalen Werte für diese Parameter zu finden. Hierfür habe ich den Mainloop so angepasst, dass in einer Schleife die Kreiserkennung auf dem aktuellen Kamera-Frame durchgeführt wird. Die gefundenen Kreise werden dann auf eine separate "GUI-Matrix" gezeichnet, die dann auf dem Bildschirm angezeigt wird. Durch das Verändern der Parameter per Slider konnte ich so direkt die Auswirkungen auf die Kreiserkennung sehen.

Letztendlich habe ich bei diesen Werten sehr gute Ergebnisse erzielt:

- `dp = 2`
- `minDist = 18`
- `param1 = 230`

- `param2 = 129`
- `minRadius = 18`
- `maxRadius = 67`

Dies bedeutet nicht, dass diese Werte auch für andere Anwendungsfälle optimal sind. Je nach Lichtverhältnissen, Kamerawinkel und Bildqualität können diese Werte stark variieren. Es ist also sehr wichtig, die Parameter für die Kreiserkennung an jeden Anwendungsfall individuell anzupassen.

4.2 Buffering der Kreise

5 Template Matching

5.1 Überlegungen

Jetzt wo ich die Kreiserkennung implementiert habe, ist es mir möglich Münzen im Bild zu erkennen. Der nächste Schritt wäre es nun, die Münzen zu klassifizieren und zu unterscheiden. Doch wie klassifiziert man in openCV Bilder? Eine Möglichkeit ist das Template Matching. Dabei wird ein Template, also eine Vorlage, über ein Bild geschoben und die Ähnlichkeit an jedem Punkt beziehungsweise Pixel berechnet. Das Ergebnis ist eine Heatmap, die die Übereinstimmung an jedem Punkt im Bild anzeigt.

5.2 Die Vorlagen

Damit ich die Münzen anhand ihrer Bildinformastionen klassifizieren kann, benötige ich zuerst Vorlagen, mit denen ich die Münzen vergleichen kann. Ich habe also Bilder von den Münzen gemacht und diese als Vorlagen gespeichert. Diese Vorlagenbilder sind die Referenzbilder, anhand derer ich die Münzen im Kamerabild auf Ähnlichkeit überprüfen möchte.

Hier hat sich vor allem die spiegelnde Oberfläche der Münzen als Problem herausgestellt. Ich musste viel mit dem Lichteinfallswinkel und der Beleuchtung spielen, um von jeder Münze ein gutes Bild zu bekommen. Auch der Abnutzungsgrad hat das Erstellen der Vorlagen erschwert. So musste ich vor allem bei den 50-20-10 Cent Münzen und bei den 5-2-1 Cent Münzen darauf achten, dass die Münzen stets eine ähnliche Optik ausweisen. Andernfalls könnte es passieren, dass die Spiegelung oder die Abnutzung als Unterschied interpretiert wird und nicht mehr die Ziffern auf den Münzen.

Meine Vorlagen sehen wie folgt aus:



Bevor ich nun mit dem Vergleich der Vorlagen beginne, brauche ich zunächst eine Möglichkeit um sowohl die Vorlagen als auch die daraus resultierenden Ergebnisse zu speichern. Dafür habe ich ein global verfügbares Objekt `COINS` erstellt, welches die Münzen als Schlüssel enthält und als Wert ein Objekt mit den Eigenschaften `diameter` und `value` hat. Weitere Eigenschaften können dann zur Laufzeit den einzelnen Münzen hinzugefügt werden.

```
1 let COINS = {  
2     Euro2 : {  
3         diameter: 25.75,  
4         value: 2  
5     },
```

```

6   Euro1 : {
7     diameter: 23.25,
8     value: 1
9   },
10  Cent50 : {
11    diameter: 24.25,
12    value: 0.5
13  },
14  Cent20 : {
15    diameter: 22.25,
16    value: 0.2
17  },
18  Cent10 : {
19    diameter: 19.75,
20    value: 0.1
21  },
22  Cent5 : {
23    diameter: 21.25,
24    value: 0.05
25  },
26  Cent2 : {
27    diameter: 18.75,
28    value: 0.02
29  },
30  Cent1 : {
31    diameter: 16.25,
32    value: 0.01
33  }
34 }

```

Diese Datenstruktur kann ich nun verwenden, um die Vorlagen zu speichern und ihnen eine Münze zuzuordnen. In der folgenden Methode iteriere ich über jede Münze und lese das Vorlagenbild ein:

```

1 const templatesLoaded = new Event('templatesLoaded');
2 let path = "../Templates/"
3
4 function InitTemplates() {
5   let coinLength = Object.keys(COINS).length;
6   let loadedCoins = 0;
7   Object.entries(COINS).forEach(([key, value]) => {
8     //is in the template folder a picture with the same name as the key?
9     let img = new Image();
10
11     img.onload = () => {
12       //save image as matrix in the coin object
13       COINS[key].template = cv.imread(img);
14
15       loadedCoins++;
16       if(loadedCoins === coinLength){
17         document.dispatchEvent(templatesLoaded);
18       }
19     }
20
21     img.onerror = () => {
22       console.log("error: " + key);
23     }
24
25     img.src = path + key + ".png";
26
27   });
28 }

```

Das Event *templatesLoaded* wird ausgelöst, sobald alle Vorlagen geladen wurden. Somit kann ich

sicherstellen, dass Programmschritt welche die Vorlagen benötigen erst ausgeführt werden, wenn die Vorlagen auch wirklich geladen wurden.

5.3 Funktionsweise

Template Matching funktioniert in openCV.js mit der folgenden Funktion:

```
1 cv.matchTemplate(image, templ, result, method, mask);
```

Lasst uns die Parameter genauer betrachten:

- **image** - Das Bild, in dem das Template gesucht werden soll. Es muss als openCV-Matrix vorliegen.
- **templ** - Das Template, das gesucht werden soll. Es muss auch als openCV-Matrix vorliegen.
- **result** - Die Heatmap, in der die Übereinstimmung an jedem Punkt im Bild gespeichert wird. Auch diese muss als openCV-Matrix vorliegen.
- **method** - Die Methode, die zur Berechnung der Übereinstimmung verwendet werden soll. Aktuell gibt es folgende Methoden in openCV.js:
 - cv.TM_SQDIFF** - Summe der quadrierten Differenzen (kleinere Werte bedeuten bessere Übereinstimmung)
 - cv.TM_SQDIFF_NORMED** - Normalisierte Summe der quadrierten Differenzen (kleinere Werte bedeuten bessere Übereinstimmung)
 - cv.TM_CCORR** - Kreuzkorrelation (größere Werte bedeuten bessere Übereinstimmung)
 - cv.TM_CCORR_NORMED** - Normalisierte Kreuzkorrelation (größere Werte bedeuten bessere Übereinstimmung)
 - cv.TM_CCOEFF** - Kreuzkorrelationskoeffizient (größere Werte bedeuten bessere Übereinstimmung)
 - cv.TM_CCOEFF_NORMED** - Normalisierter Kreuzkorrelationskoeffizient (größere Werte bedeuten bessere Übereinstimmung)
- **mask** - (optional) Eine Maske, die angibt, welche Bereiche des Bildes berücksichtigt werden sollen. In meinem Fall benutze ich keine Maske, da ich bereits vor Aufruf der Funktion ungewünschte Bereiche des Bildes ausschneide.

Meine Funktion für das Template Matching sieht nun so aus:

```
1 function MatchTemplates(src, circle){  
2     //create result string  
3     let resultsString = [];  
4     Object.entries(COINS).forEach(([key, value]) => {  
5         let resultMat = new cv.Mat();  
6  
7         //resize template to the size of src  
8         let templateResized = new cv.Mat();  
9         cv.resize(COINS[key].template, templateResized, COINS[key].template.size(),  
10            0, 0, cv.INTER_AREA);  
11  
12         cv.matchTemplate(src, templateResized, resultMat, cv.TM_SQDIFF_NORMED);  
13  
14         //get highest value  
15         let minMax = cv.minMaxLoc(resultMat);  
16         let min = minMax.minVal;
```

```

17     resultsString.push(new Result(key, min));
18
19     templateResized.delete();
20 );
21
22 //sort results from highest to lowest
23 resultsString.sort((a, b) => a.value - b.value);
24
25 //console.dir(resultsString);
26
27 //save bestmatch in circle
28 circle.bestMatch = COINS[resultsString[0].name];
29 circle.matchValue = resultsString[0].value;
30
31 //set matchValue to 2 decimal places
32 circle.matchValue = Math.round(circle.matchValue * 100) / 100;
33
34 src.delete();
35 }

```

Innerhalb des main-loops wird diese Funktion nun für jeden gefundenen Kreis aufgerufen. Der Parameter `src` ist dabei eine ausgeschnittene Region des Bildes, in der sich der Kreis befindet. Der Parameter "circle" ist das Circle-Objekt, in dem die Informationen zum Kreis gespeichert sind. Die Funktion iteriert anschließend über alle Vorlagen und berechnet die Übereinstimmung für jede Vorlage. Hierbei muss beachtet werden, dass Vorlage und das zu prüfende Bild vorher auf die selbe Größe gebracht werden müssen. Die Ergebnisse werden in einem Array gespeichert und anschließend sortiert. Das beste Ergebnis wird dann im Circle-Objekt gespeichert.

5.4 Exkurs: Mehrere Matrizen in einem Canvas anzeigen

Bei wachsender Komplexität der Webanwendung ob ich mir immer wieder gewünscht eine Möglichkeit zu haben, mehrere Matrizen in einem Canvas anzuzeigen. Dies ist standardmäßig in openCV.js nicht vorgesehen, da die Funktion `cv.imshow()` immer nur eine Matrix anzeigen kann. Also habe ich mir eine eigene Funktion geschrieben, die mehrere Matrizen in einem Canvas anzeigen kann. Diese Funktion sieht so aus:

```

1 function ShowMatrices(src, canvas) {
2     if (src.length === 0) {
3         console.error("No src to show");
4         return;
5     }
6
7     if (src.length === 1) {
8         console.warn("Only one matrix to show. Using ShowMatrix instead");
9         ShowMatrix(src[0], canvas);
10    return;
11 }
12
13 let numberofMatrices = src.length;
14
15 // Berechnung der Gittergröße
16 let gridCols = Math.ceil(Math.sqrt(numberofMatrices));
17 let gridRows = Math.ceil(numberofMatrices / gridCols);
18
19 // Maximalbreite und -höhe der Matrizen basierend auf der Größe des Canvas
20 let cellWidth = canvas.width / gridCols;
21 let cellHeight = canvas.height / gridRows;
22
23 // Canvas leeren

```

```

24 let ctx = canvas.getContext('2d');
25 ctx.clearRect(0, 0, canvas.width, canvas.height);
26
27 // Jede Matrix im Gitter zeichnen
28 src.forEach((mat, index) => {
29     // Spalte und Zeile bestimmen
30     let col = index % gridCols;
31     let row = Math.floor(index / gridCols);
32
33     // Zielbereich für diese Matrix
34     let x = col * cellWidth;
35     let y = row * cellHeight;
36     let targetSize = new cv.Size(cellWidth, cellHeight);
37
38     // Matrix auf passende Größe skalieren
39     let resizedMat = new cv.Mat();
40     cv.resize(mat, resizedMat, targetSize, 0, 0, cv.INTER_AREA);
41
42     // Matrix in RGBA umwandeln wenn es sich um eine Graustufenmatrix handelt
43     if (resizedMat.channels() === 1) {
44         let rgbaMat = new cv.Mat();
45         cv.cvtColor(resizedMat, rgbaMat, cv.COLOR_GRAY2RGB);
46         resizedMat.delete();
47         resizedMat = rgbaMat;
48     }
49
50     // Erstelle ein temporäres Canvas für diese Matrix
51     let tempCanvas = document.createElement('canvas');
52     let tempCtx = tempCanvas.getContext('2d');
53     tempCanvas.width = resizedMat.cols;
54     tempCanvas.height = resizedMat.rows;
55
56     // In ein ImageData konvertieren und in das temporäre Canvas zeichnen
57     let imageData = new ImageData(new Uint8ClampedArray(resizedMat.data),
58                                   resizedMat.cols, resizedMat.rows);
59     tempCtx.putImageData(imageData, 0, 0);
60
61     // Zeichne das temporäre Canvas auf das Haupt-Canvas
62     ctx.drawImage(tempCanvas, 0, 0, resizedMat.cols, resizedMat.rows, x, y,
63                   cellWidth, cellHeight);
64
65     // Bereinige die temporären Matrizen
66     resizedMat.delete();
67 });
68 }

```

Um mehrere Bilder in einen Canvas zu zeichnen, musste ich die Größe der Bilder anpassen, damit sie in ein Grid passen. Dafür habe ich die Funktion `cv.resize()` verwendet. Anschließend müssen die Matrizen zuerst in ein temporäres Canvas gezeichnet werden, bevor sie in das Haupt-Canvas gezeichnet werden können. Die Funktion `drawImage()` von Canvas kann nur Bilder zeichnen, keine Matrizen. Daher musste ich die Matrizen in ein `ImageData`-Objekt umwandeln, bevor ich sie in das Canvas zeichnen konnte.

5.5 Erste Ergebnisse

Es hat sich gezeigt, dass der Vergleich der Vorlagen prinzipiell funktionieren kann. Jedoch ist die Genauigkeit noch nicht zufriedenstellend. Münzen können zwar grob in die richtige Kategorie eingeordnet werden (sprich 1-2-5 Cent Münzen und 10-20-50 Cent Münzen), jedoch ist die Unterscheidung innerhalb dieser Kategorien nicht funktional. So werden aktuell nur maximal die Hälfte der Münzen korrekt zugeordnet.

Es scheint, als wären die Informationen der Ziffern auf den Münzen nicht ausreichend, um eine genaue Klassifizierung zu ermöglichen. Die Farbe einer Münze hat im Gegensatz dazu einen relativ großen Einfluss auf das Ergebnis. So werden Münzen meist der korrekten Farbkategorie zugeordnet, jedoch nicht dem korrekten Wert. Dies ist ein Hinweis darauf, dass die Farbinformationen der Münzen stärker gewichtet werden als die Forminformationen. Da sich Münzen innerhalb einer Kategorie (z.B. 1-2-5 Cent Münzen) jedoch nur in der Größe und ihrer Ziffer optisch unterscheiden, ist es schwierig eine weitere Methode der Klassifizierung zu finden. Ein Prüfen der Größe würde in meinem Anwendungsfall nicht funktionieren, da sowohl der Abstand zur Kamera als auch die Position der Münzen im Bild variabel sind.

Eine Möglichkeit die Genauigkeit zu erhöhen wäre es, den störenden Faktor der Farbinformationen zu eliminieren, und stattdessen nur Vorlagen zu verwenden, welche nur die notwendigen Informationen enthalten. Dies könnte z.B. durch das Erstellen von Kantenbildern der Münzen erreicht werden. Solche Bilder wie sie vom Canny-Algorithmus erstellt werden, enthalten nur die Kanten des Ausgangsbildes und keine Farbinformationen, da sie rein als Schwarz-Weiß-Bilder vorliegen. Diese Kantenbilder könnten dann an Stelle der aktuellen Farbbilder als Vorlagen verwendet werden.

Jedoch gibt es noch ein weiteres Problem: die Rotation. Bereits eine kleine Drehung des Ausgangsbildes verursacht große Unterschiede in der Übereinstimmung. Das Template Matching in openCV berechnet standardmäßig nur die Übereinstimmung für das Template in der gleichen Rotation wie das Bild. Dies ist ein Problem, da die Münzen im Kamerabild in unterschiedlichen Rotationen vorkommen können. Eine Möglichkeit dieses Problem zu lösen wäre es, das Template in verschiedenen Rotationen zu speichern und für jede Rotation die Übereinstimmung zu berechnen. Dies würde jedoch die Anzahl der Vorlagen vervielfachen und einen unverhältnismäßig hohen Aufwand bedeuten. Eine andere Möglichkeit wäre es, das Ausgangsbild mehrfach zu rotieren und für jede Rotation die Übereinstimmung zu den Vorlagen berechnen. So wäre bei ausreichend kleingranularer Rotation sichergestellt, dass jede Münze mindestens einmal eine ähnliche Ausrichtung wie das zutreffende Template hat. Dies würde jedoch die Rechenzeit signifikant erhöhen, da bedeutend mehr Berechnungen pro Münze durchgeführt werden müssten. Zudem braucht es dafür eine Möglichkeit, alle Zwischenergebnisse der Rotationen zu speichern, um anschließend das beste Ergebnis zu wählen.

6 Anpassungen

6.1 Kantenerkennung

6.1.1 Funktionsweise der Canny-Methode

6.1.2 Die korrekten Parameter finden

6.2 Matrizen rotieren und transformieren

6.3 Neues Problem: Mat-Types

6.4 Asynchronität und Ladebalken

7 Ergebnisse

7.1 Erste Ergebnisse

7.2 Zuschneiden der Bilder

7.3 Weitere Anpassungen

7.4 Finale Ergebnisse

8 Fazit

8.1 Zusammenfassung

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.

8.2 Die größten Probleme

8.3 Ausblick