Recognition and Analysis of a Smart Home User Interface

Julian Lingnau
In kooperation with Liebherr-Hausgeräte Ochsenhausen GmbH and RWU University of Applied Sciences Ravensburg-Weingarten, 2025 Germany, Repo: [1]

Abstract — Unlike modern industrial systems, which often include comprehensive bidirectional data monitoring, the current generation of smart refrigerator interfaces within the company lacks a feedback mechanism to verify whether displayed information matches the intended output. This discrepancy can lead to operational inefficiencies and undetected UI malfunctions, as data is transmitted to the display but not read back for validation. To address this issue, a computer vision-based system has been developed to automatically recognize and interpret the refrigerator's UI display. The system utilizes image processing techniques to detect the interface, analyze displayed values and compare them with the expected output. Core functionalities include text recognition using optical character recognition (OCR), color and symbol analysis for detecting visual errors. This approach enables automated error detection, allowing engineers to identify inconsistencies without manual inspection. Furthermore, the system can be deployed for automated UI testing during the manufacturing process, ensuring higher reliability and reducing the quality control effort. By integrating computer vision for UI validation, this project enhances the reliability and operational transparency of smart refrigeration systems, bridging the gap between data transmission and visual confirmation.

March 31, 2025

Keywords: Smart refrigerator, computer vision, User Interface, UI recognition, OCR, image processing, automated error detection, Multi-scale Template Matching

1 Introduction

Graphical User Interfaces (GUIs) are critical for human-machine interaction in smart home appliances. In the current company, the smart refrigerator display is designed as a one-way communication channel, where data is sent to the screen without any hardware mechanism for reading back the output.

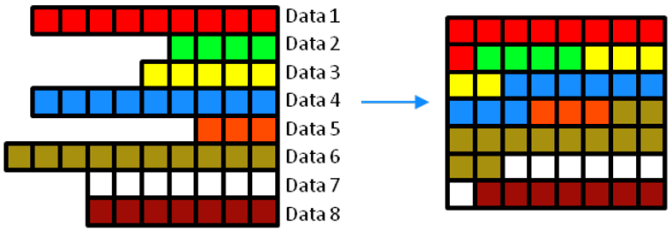


Figure 1: One-way communication Array to UI

This unidirectional design-driven by cost minimization and manufacturing constraints raises a significant **problem:** How is it possible to ensure that the displayed information is accurate and matches the intended commands?

To address this challenge, a software-based solution was developed that uses a standard, low-cost camera to capture display snapshots during automated Jenkins tests. The images are analyzed using traditional computer vision techniques. Although this approach may appear naive compared to bidirectional systems, it provides a cost-effective method for early detection and diagnosis of display errors. The method approximates human visual perception and ensures that automated test results closely match those observed visually.

In this paper, **three** fundamental questions will be addressed:

1. **System Capabilities:** How good does the system identify user interface elements, extract textual data, classify color states to ensure accurate validation of display outputs?

2. **Methodological Approach:** What specific methodologies were employed to develop a reproducible approach using open-source tools and how were these critically evaluated to determine their essential components?

3. **Method Criteria:** Why was a hybrid approach, combining classical methods with deep learning models, selected over solely traditional or modern techniques, considering the static user interface and strict cost constraints, particularly regarding transparency, interpretability and efficiency?

The remainder of this paper is structured as follows: Section 2 discusses related work and relevant prior approaches, providing the rationale for selecting classical image processing techniques. Section 3 presents detailed explanations of the algorithms, implementation-specific techniques, ensuring that the methodology can be replicated. Section 4 evaluates the output and show the results.

1.1 Project Requirements

To ensure clarity, the functional and non-functional requirements of the implemented solution are outlined below.

Non-Functional Requirements:

- Executable on standard company notebooks.
- Input images provided in Full HD resolution.
- Single-image processing time must not exceed 10 seconds.
- Compatible with Python and integrable into automation frameworks (specifically Jenkins).
- Adaptable framework suitable for varying display sizes and resolutions.
- Modular and maintainable code structure.

Functional Requirements:

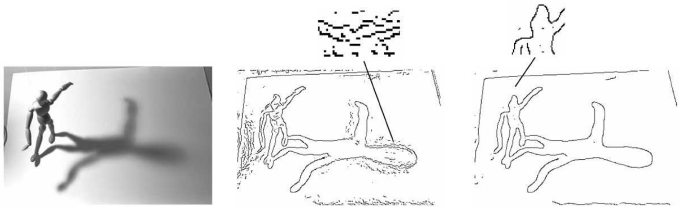
- Accurate detection of display position and dimensions within captured images.
- Output analysis including pixel counts per color and detailed color histograms.
- Reliable text recognition using Optical Character Recognition (OCR).
- Verification of the presence and correct positioning of specific UI elements.
- Detection and analysis of dynamic UI elements.

2 Related Work

Validation and testing of graphical user interfaces (GUIs) have traditionally involved image processing, object detection and optical character recognition (OCR). This section critically reviews state-of-the-art methods reported in the literature for addressing similar challenges and highlights their relevance to the project.

2.1 UI Element Detection and Extraction

The problem of UI recognition has been addressed through various techniques. Early systems in 2009, such as Sikuli, leveraged template matching on screenshots for automation tasks. It was a simple tool automate tests by just giving the whole screen of a GUI and check if the user presses buttons, or reproducing steps for example: Access Website with Username=admin and click login will result in TestLogin **True/False** more examples will be found under [2]. Edge detection techniques, like Canny edge detection, remain prevalent in many applications because of their ability to identify semantically meaningful image features such as object boundaries. As Szeliski notes, edge points 'carry important semantic associations' such as defining 'the boundaries of objects'. However, a well-known limitation of these techniques is their sensitivity to noise. The process of taking image derivatives, which is central to many edge detectors, 'accentuates high frequencies and hence amplifies noise' ([3] 7.2 Edges and contours). Therefore, in situations with significant noise, reflections, or uneven illumination, the basic edge detection approach may be insufficient without additional pre-processing or more sophisticated techniques.



(a) (b) (c)

Figure 2: Scale selection for edge detection (a) original image; (b–c) Canny/Deriche edge detector tuned to the finer (mannequin) and coarser (shadow) scales ([3] 7.2 Edges and contours)

Recent advances in machine learning have significantly enhanced object detection capabilities, particularly in the context of GUI element detection. Deep convolutional neural networks (CNNs), such as Faster R-CNN and YOLO, are widely used for this purpose. These models are excellent in element detection because they are explicitly trained to recognize hierarchical features (e.g., edges, textures, shapes) through supervised learning on annotated datasets, as demonstrated in studies by Ren [4]. Their ability to learn complex visual patterns makes them robust against variations in illumination and background complexity. However, these methods require extensive training datasets and substantial computational resources, which can be impractical in embedded or low-resource environments.

In many organizations, the availability of sufficient training data is a significant challenge. For instance, relying on a limited set of images or templates may not be adequate for achieving reliable detection. Furthermore, deploying such sophisticated models for simple tasks can be overly complex and inefficient. Studies like those by Chen highlight the challenges of GUI element detection, including high in-class variance and cross-class similarity, which complicate accurate region detection and classification [5].

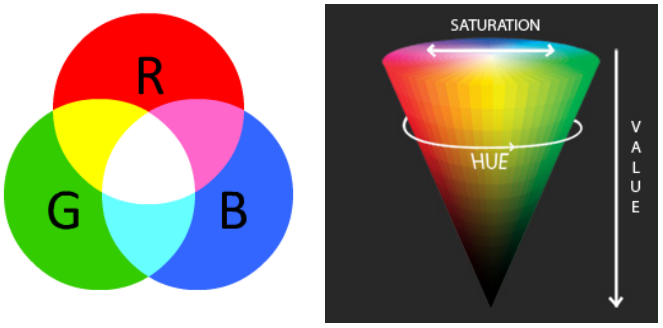
Illumination Variations and Autofocus:
Capturing clear images of UI elements is challenging due to varying lighting conditions. Common issues include reflections, glare, and uneven exposure, which can lead to inaccuracies in image analysis. Researchers address these challenges with hardware-based solutions like fixed lighting setups and software methods such as histogram equalization [6] and image normalization [7]. Histogram equalization is a technique used to adjust the contrast of images by transforming the histogram to a more uniform distribution, enhancing the visibility of details in both bright and dark areas. This process helps to improve the overall image quality by redistributing the pixel values to make the most of the available dynamic range. Image normalization is a process that adjusts the range of pixel values to bring the image into a standard range, often used to ensure consistent brightness and contrast levels across different images.

Autofocus, particularly in low-end cameras, poses significant challenges. The autofocus process can be time-consuming, often requiring manual calibration or predefined delays to ensure proper focus. This time constraint is problematic in high-throughput testing scenarios where speed is crucial. In contrast, higher-end cameras offer faster and more reliable autofocus systems [8]. However, these cameras come with a higher price tag (often between €4,000 and €6,000) and are larger than simple webcams.

For low-end cameras, manual focus adjustments can be used to bypass autofocus limitations, although this may require additional setup time.

Color Classification and Color Spaces:
The choice of color space significantly impacts robustness in UI testing. RGB and HSV are two most common

color spaces, each with its own approach to representing color. RGB directly encodes pixel intensity values using red, green and blue components. However, this makes it highly sensitive to lighting changes. As explained by Mr. Glover [9], specifying a color like red as (255, 0, 0) in RGB is very specific; slight variations in lighting can significantly alter these values. In real-world conditions, the color red is often a mix of all three primary colors, making reliable detection challenging.



(a) (b)

Figure 3: RGB (a); HSV (b) [9]

In contrast and also justified by Mr. Glover, was that the HSV color model separates chromaticity (Hue and Saturation) from luminance (Value). This characteristic makes it inherently more robust to illumination variations. By decoupling color information from intensity, HSV allows for more reliable classification of UI element colors, particularly under varying lighting conditions. For example, when detecting a "red-like" color, HSV only needs to adjust the Hue to capture a specific range, reducing noise and simplifying the detection process. This makes HSV a favored choice in computer vision and UI testing applications where lighting conditions may not be consistent.

2.2 OCR vs DeepOCR

Text extraction from images plays a critical role in numerous applications, including UI analysis and document processing. Traditional Optical Character Recognition techniques, such as those implemented in Tesseract, rely heavily on manually crafted feature extraction and rule based methods. These systems typically perform well under ideal conditions where text is clean, well aligned and printed in standard fonts; but their performance degrades significantly in the presence of noise, low contrast, or unconventional layouts.

In contrast, the advent of deep learning has given rise to DeepOCR systems (for example, EasyOCR and PaddleOCR) that employ end to end neural network architectures. By leveraging Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), these approaches automatically learn hierarchical features directly from large scale annotated datasets. This data driven methodology enables DeepOCR systems to achieve substantially higher robustness and accuracy. As the benchmark in Fig.7 BLUDELTA shows, is the best deep learning OCR 20% better than the best without [10].

Its important to notice that in this test 89 numbers consisting of 570 characters in the benchmark were used as ground truth (66% good quality 33% bad quality). Why numbers? "Numbers were used because they cannot or only rarely be corrected in the subsequent process of an iOCR. If one or the other letter tilts during recognition, you can use "similarities" to draw conclusions about the correct word (e.g. also using NLP models), which is not possible with numbers" [10].

Figure 4: Random cash register receipts (Testdata) [10]

- **Exact Match:** The number must match the ground truth exactly.
- **Levenshtein Distance:** How similar are the recognized values (measure of quality)
- DeepOCR blue
- ClassicOCR orange

Tool	Exakt	Levenshtein
Google OCR	92%	95,70%
Paddle OCR V2.5	64%	92,26%
AbbyyFineReader15	71%	86,47%
Omnipage Ultimate V19.2	62%	82%
Tesseract 5 OCR	57%	73%
OCR.space	50%	74%
Onlineocr.net	42%	69%

Figure 5: Benchmark Results [10]

"It is noticeable that both (Google and Paddle) use DeepLearning (DeepOCR) and were apparently much better than the competition with poor image quality. Both Google and Paddle are likely to have achieved a lot here in a very short time through deep learning and data. PaddleOCR can also be trained with your own data and improved accordingly. It can be assumed that with training PaddleOCR can achieve a similar performance as Google." [10]

2.3 Template vs Feature-Based Matching

As outlined in Brunelli ([11] 1.1. Template Matching and Computer Vision), Template matching is a foundational method in computer vision, where a predefined template or pattern serves as a reference to locate similar regions in a larger image. In this approach, both the template and the image region are transformed into vector representations and compared using distance measures (for example, the L2 norm) to quantify similarity. Even small variations in scale, rotation, or illumination can significantly impact the matching process. Consequently, template matching is most effective in controlled environments where target objects maintain a consistent size and orientation.

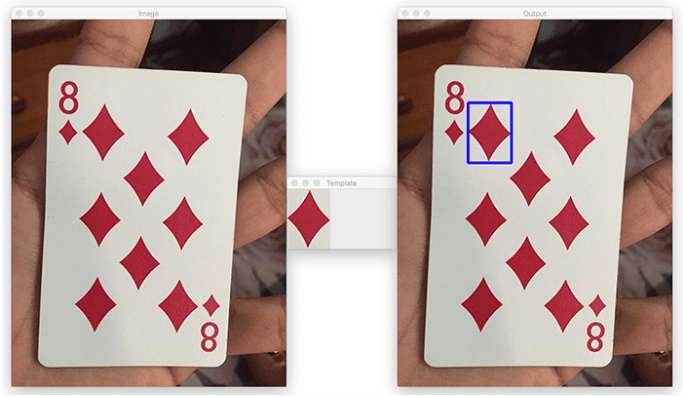


Figure 6: **Left:** Input image containing a template of a "diamond." **Middle:** The template of a "diamond" that we want to detect in the left image. **Right:** Output of applying basic template matching [12]

In contrast, feature-based methods which can be found under ([3] 7.1.1 Feature detectors) rely on extracting distinctive features such as edges, corners, or textures from images. Algorithms like BRISK (Binary Robust Invariant Scalable Keypoints) detect these features and facilitate robust against scaling, rotation and illumination. This robustness makes feature-based approaches particularly suitable for scenarios where objects may appear under varying conditions.

Algorithm	Feature Detection	Feature Description	invariant under scaling	invariant under rotation	invariant under illumination	Free/no patent
SIFT	✓	✓	✓	✓	✓	✗
Harris	✓	✗	✗	✓	✓	✓
FAST	✓	✗	✗	✓	✓	✓
BRIEF	✗	✓	✗	✗	✓	✓
ORB	✓	✓	✗	✓	✓	✓
BRISK	✓	✓	✓	✓	✓	✓

Figure 7: Overview: Feature detectors

Figure 8: **Left:** Example of a successful Feature-based Matching **Right:** The template of a head that we want to detect in the left image.[13]

To understand the rotation problem of template matching more deeply, below is the result that would have been obtained if we had applied template matching with the same criteria.

Figure 9: **Left:** Example of a failed Template Matching **Right:** The template of a head that we want to detect in the left image.[13]

However, in cases where images lack sufficient distinctive features; such as small or simple UI icons feature-based methods may struggle to perform effectively. In these situations, template matching can serve as a viable alternative, as it does not depend on the presence of prominent features and can effectively detect predefined patterns in standardized UI elements.

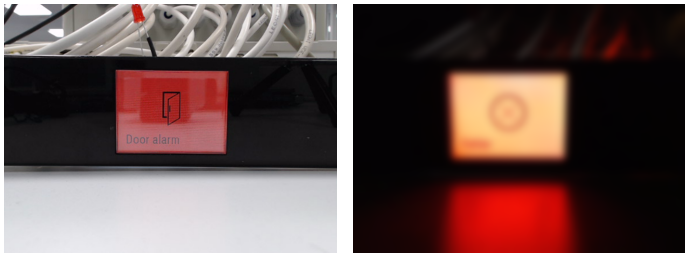
In conclusion, selecting from among image processing techniques, deep learning approaches, OCR methodologies and matching strategies requires careful consideration of specific application requirements – particularly the constraints of the production environment, available computational resources and the expected variability in UI elements. Striking the right balance between the advantages and limitations of these methods is essential for effective and efficient GUI validation and testing.

3 Methodology

3.1 Step-by-Step Process

The automated UI validation system pipeline comprises several computer vision stages that analyze captured UI images to verify that they meet expected properties:

1. **Image Acquisition:** The process begins by capturing the UI screen. A simple solution was selected to address the lighting issues described in the Related Work section. Due we had the same problem of the light scenarios:



(a) (b)

Figure 10: Daylight UI (a) good quality but reflections; (b) Nighttime UI blurry

To prevent any effortful pre-normalization, a setup consisting of a darkened, styrofoam enclosure with a lid is utilized, into which the UI is smoothly inserted using a custom-carved sliding mechanism that ensures a perfect fit. The camera, which is a Logitech C922 Pro, is mounted centrally at a fixed distance, with neatly arranged cables and the entire enclosure darkened.

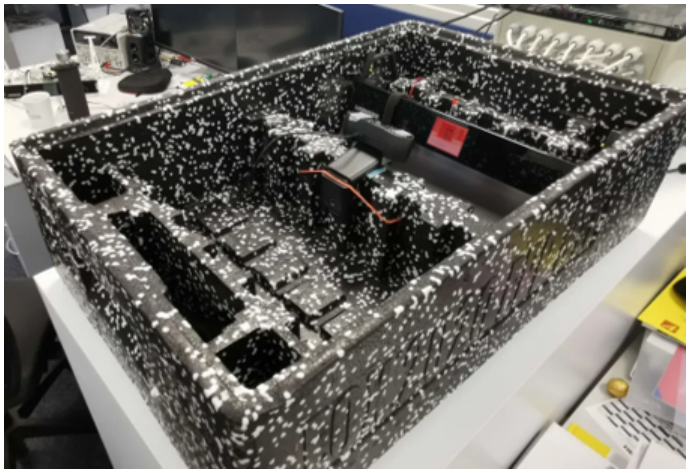


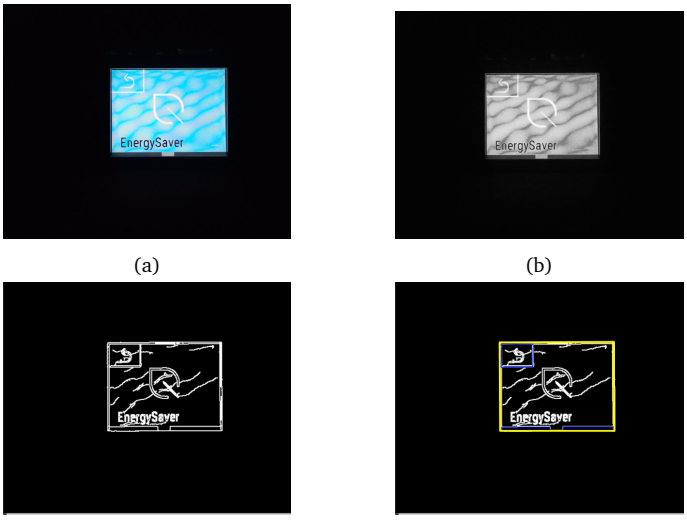
Figure 11: Setup including UI and the camera

Due to the fixed position, it is possible to disable autofocus and program the camera to capture the UI at an exact location, thereby reducing delay times. This configuration ensures highly consistent lighting conditions, eliminating the need for additional snapshot normalization. A high-resolution image of the interface is then captured using CommandCam [14], ensuring that the entire UI is visible. For example, the following command is executed:

```
CommandCam.exe /devname "c922 Pro Stream Webcam" /delay 500
```

This command captures a snapshot after a 500 ms delay to allow the camera to stabilize. The raw image is then used directly for further analysis.

2. **Display Region Detection (Calibration):** To interpret the information on the display, it is important to detect and recognize the display itself. For this some kind of a calibration by detecting the four cornor points is needed. Ensure that the provided snapshot is bright enough (not equal to black) to make the edges and corners of the UI easily visible. Since Canny edge detection relies solely on brightness gradients, the image's color is irrelevant. In practice, the image is first converted to grayscale — discarding all color information; as long as the image is bright enough, strong edges and corners will be reliably identified. For this Edge detection is performed using the Canny algorithm [15] to highlight given features.



(a) (b) (c) (d)

Figure 12: (a) Input UI Image; (b) Image converted to grayscale; (c) Detected edges by canny; (d) quadrilateral correct display region quadrilateral uncorrect / to small

The detected edges are then filtered using geometric heuristics to isolate a clear quadrilateral representing the display region as seen in Fig.12(d). A Hough line transform is utilized to identify straight lines and confirm perpendicular pairs that mark the corners. Once the corners are determined, they are saved using a JSON-based calibration routine, ensuring that subsequent images captured during the current cycle are automatically rectified. Finally, a perspective transformation (homography) is applied to warp and crop the UI to a flat, fronto-parallel view, ensuring that even if later snapshots vary in brightness, the location of the display remains known.

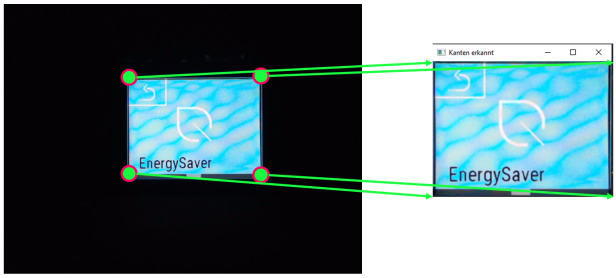


Figure 13: UI corners transformed to dynamic UI size (320x240)

Once the corner points are identified/saved, extracting the UI is straightforward — even in complete darkness as seen below.

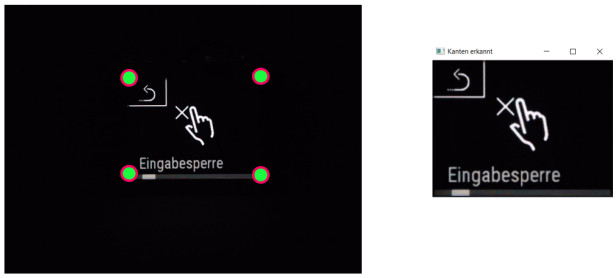


Figure 14: UI corners transformed to dynamic UI size (320x240)

3. **Color Classification of UI Components:** With the UI region extracted, the colors of key UI elements are analyzed using the HSV color space. HSV was selected over RGB because RGB-based color testing is both complicated and imprecise. In RGB (actually BGR, as OpenCV loads images in this order), comparisons such as "if $R > B$ and $G > B$" require complex conditional logic and still may not reliably indicate whether a pixel is truly red. In contrast, HSV offers a clear and mathematically defined representation of color. The hue component is measured on a circular scale from 0° to 360°, with red appearing at both 0° and 360°. This circular nature allows us to set unambiguous thresholds.

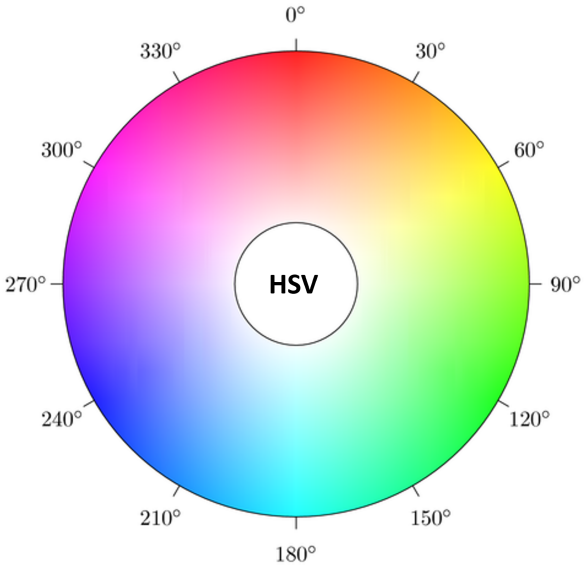


Figure 15: HSV Colorcircle

For each pixel, the RGB values are converted to HSV using the following equation:

$$
\begin{aligned}
V &\leftarrow \max(R,G,B),\\
S &\leftarrow \begin{cases} \dfrac{V-\min(R,G,B)}{V}, & \text{if } V \neq 0,\\ 0, & \text{otherwise,} \end{cases}\\
H &\leftarrow \begin{cases} 60^\circ \times \dfrac{(G-B)}{V-\min(R,G,B)}, & \text{if } V = R,\\ 120^\circ + 60^\circ \times \dfrac{(B-R)}{V-\min(R,G,B)}, & \text{if } V = G,\\ 240^\circ + 60^\circ \times \dfrac{(R-G)}{V-\min(R,G,B)}, & \text{if } V = B,\\ 0, & \text{if } R=G=B, \end{cases}
\end{aligned}
\tag{1}
$$

[16] If $H < 0$, then $H \leftarrow H + 360$. After these steps, the values satisfy

$$0 \le V \le 1, \quad 0 \le S \le 1, \quad 0^\circ \le H \le 360^\circ.$$

Scaling for Different Data Types

- **8-bit images:**

  $$V \leftarrow 255\,V, \quad S \leftarrow 255\,S, \quad H \leftarrow \frac{H}{2}$$

  This maps V and S into the range $[0,255]$ and compresses H into $[0,180]$.
- **16-bit images:** Currently not supported for HSV in OpenCV 3.4. (If it were, similar scaling would apply, but using a wider integer range.)
- **32-bit images:** The floating-point (H,S,V) values are left as is.

Meaning of Variables

- R,G,B: Normalized intensities of the red, green and blue channels in $[0,1]$.
- V: Brightness, defined as $\max(R,G,B)$.
- $\min(R,G,B)$: The smallest channel value, used to calculate $\Delta = V - \min(R,G,B)$.
- S: Saturation, which indicates color purity; $S = 0$ if $V = 0$, else $S = \frac{\Delta}{V}$.
- H: Hue, the color angle in degrees $[0,360]$. If $\Delta = 0$ (i.e., $R = G = B$), then $H = 0$.

This separation of brightness (V) from chromatic components (H and S) makes HSV more intuitive for many color-based operations. It also allows simpler thresholding when searching for a specific hue range (for example, detecting red at around $H = 0^\circ$ or 360°) under varying lighting conditions.

Instead of manually implementing the conversion formulas, OpenCV's `cvtColor` function is used to automatically convert the image data. Following Python code snippet illustrates how the conversion is performed:

```
def convert_rgb_to_hsv(image_path):
    # Read the image (OpenCV loads images
        in BGR format by default)
    image = cv2.imread(image_path)

    # Convert the image from BGR to HSV
    hsv_image = cv2.cvtColor(image,
        cv2.COLOR_BGR2HSV)
    return hsv_image
```

Figure 16: Python code: BGR to HSV example

It is important to note that OpenCV represents hue values on a scale from 0 to 179 ([0,179] i.e., 180 discrete levels), effectively halving the conventional 360° scale. Thus, a hue range of 0°–10° corresponds approximately to 0–5 in OpenCV, while 350°–360° maps to roughly 175–179.

This clear definition of hue boundaries significantly simplifies the analysis. For example, if a region corresponding to a red error screen contains more than 60,000 pixels, it is classified as significant. Due to its robustness against variations in lighting conditions, the HSV

color space is ideal for UI testing. Although this level of color analysis is sufficient for the current application, it would be inadequate for safety-critical systems such as those used in traffic sign recognition or aviation displays, where the precision and reliability are essential.

For a visual representation of the HSV color model — including its circular hue structure, interested readers can refer to [17].

4. **Text Recognition (OCR):** The system extracts and validates textual content in the UI via Optical Character Recognition. The EasyOCR library (Jaided AI, 2020 [18]) is employed as the OCR engine because it leverages a deep learning model pre-trained on a wide variety of scene texts. Internally, EasyOCR combines a robust text detection module built on advanced algorithms such as EAST/CRAFT with a neural network-based text recognizer — typically implemented as a CRNN. This integration enables accurate processing of a wide variety of fonts, detection of low-contrast text and reliable recognition of pixelated UI elements. Based on these advantages, the ease of implementation as well as the insights from related work — this approach for the UI text recognition was adopted.

Figure 17: EasyOCR traffic sign recognition example [18]

One key advantage of EasyOCR is its inherent support for multiple languages. The system recognizes over 80 languages while also allowing easy expansion to additional languages via a catalog of language models. This makes the solution dynamically adaptable for future localization efforts without significant re-engineering. In practice, even when UI text is rendered with modern, anti-aliased fonts or appears on complex backgrounds, EasyOCR reliably extracts text along with corresponding bounding boxes and confidence scores. These outputs are then compared against expected UI strings (e.g., verifying that the correct error message is displayed or that a button label matches the specification).

5. **Template Detection:** In the final stage, the system verifies the presence and correct placement of specific UI elements (e.g., icons, logos, widgets), which are defined as templates. Achieved was this by employing two complementary techniques: feature-based matching and template matching.

Initially, state-of-the-art feature-based matching was applied using the BRISK algorithm [19] to extract keypoints and compute binary descriptors that, as we know are freely available and invariant under scaling, rotation and illumination. These descriptors were matched between the reference template and the UI screenshot using Hamming distance. Matches were further refined using the RANSAC algorithm, which estimates a robust homography H aligning the template with the detected region. With that the results are only the best matches.

(a) (b)

Figure 18: Grayscaled Image (a); Same Image after BRISK has detected the most prominent corners (b)

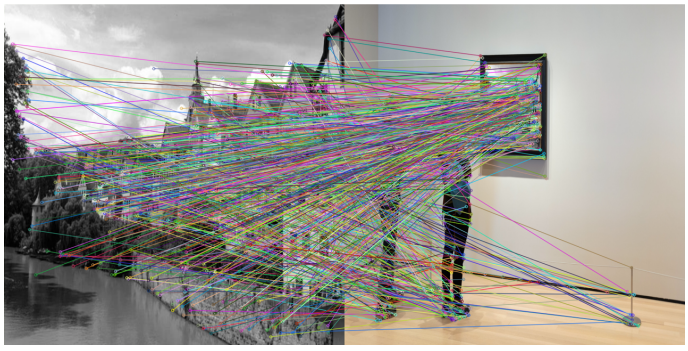


Figure 19: BRISK Feature Matching (747 matches but a lot false-positives)

Figure 20: BRISK + RANSAC (117 best matches little less but almost none false-positives)

$$p_i' \approx H\, p_i \quad \text{for all selected keypoints } p_i.$$

RANSAC (**RAN**dom **SA**mple **C**onsensus) iteratively selects four random feature pairs and evaluates (compute) each candidate H exact by counting the inliers (those satisfying $\|p_j' - H\, p_j\| < \epsilon$). The homography with the largest inlier count is kept then refined via least-squares estimation. This approach is effective when the UI image contains abundant and clear features [20].

However, extensive testing revealed that for small or pixelated icons (e.g., 50×50 or 100×100 pixels) with sparse feature information, the BRISK and RANSAC approach produced only a few matches — often around five even with wrong results as seen in Fig.21. Moreover, feature matching was sensitive to sparse features; it only performed reliably when tons of features are given. So again a alternative solution has been found.

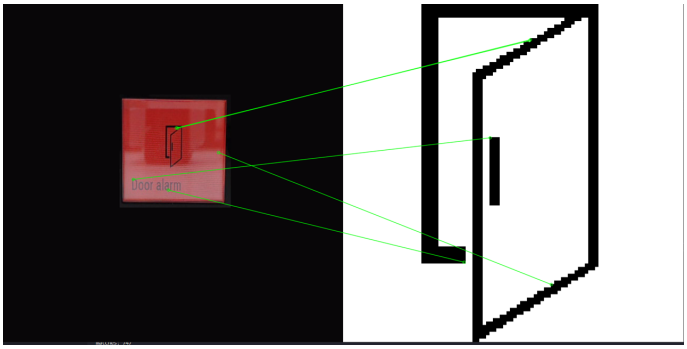
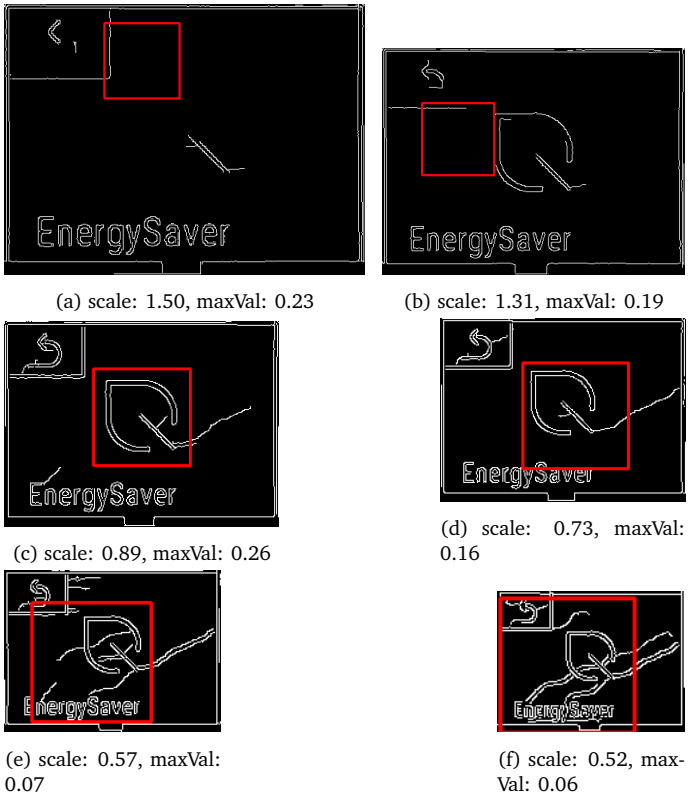


Figure 21: BRISK + RANSAC (4 best matches but totally wrong due sparse features)

Consequently, multi-scale template matching was integrated as the final detection strategy. In contrast to standard template matching — which slides a fixed-size template over the multi-scale template matching iteratively resizes the UI image over a range of scales and computes the normalized cross-correlation (NCC) at each scale. The NCC, denoted as γ (Gamma), is defined as:

$$\gamma = \frac{\sum_{x,y}(f(x,y) - \bar{f}_{u,v})(t(x-u,y-v) - \bar{t})}{\sqrt{\sum_{x,y}(f(x,y) - \bar{f}_{u,v})^2 \sum_{x,y}(t(x-u,y-v) - \bar{t})^2}} \tag{2}$$

[21] where $f(x,y)$ is the image value at position (x,y), $\bar{f}_{u,v}$ is the mean of the image values within the template area shifted by (u,v), $t(x-u,y-v)$ is the template value at position $(x-u,y-v)$, and $\bar{t}$ is the mean of the template values. The summation is performed over all pixels in the relevant region. The overall best match is selected if the maximum correlation exceeds a predetermined threshold (e.g., $C > 0.45$).



(a) scale: 1.50, maxVal: 0.23

(b) scale: 1.31, maxVal: 0.19

(c) scale: 0.89, maxVal: 0.26

(d) scale: 0.73, maxVal: 0.16

(e) scale: 0.57, maxVal: 0.07

(f) scale: 0.52, maxVal: 0.06

Figure 22: 6 picked and scaled images from 1.5 to 0.2 (41 total steps)

(a) Template (b) scale: 1.02, maxVal: 0.50

Figure 23: Eco Template (a); Bestmatch of the Multi-scale template matching (b) is > 0.45 ? YES!

This multi-scale approach, as detailed in [22], is particularly effective for detecting icons across varying sizes, as it adapts to differences between the stored template and the actual UI element. Nonetheless, the method is not flawless: "It's important to keep in mind that template matching does not do a good job of telling us if an object does not appear in an image. Although thresholds on the correlation coefficient can be set, in practice this approach proves neither reliable nor robust. If you are looking for a more robust approach, you'll have to explore keypoint matching."[22]

Also another problem which the author came cross was, when several similar icons are present (such as different modes represented by slight variations of a waterdrop icon), the algorithm might detect the common template shape without distinguishing context-specific modifications.

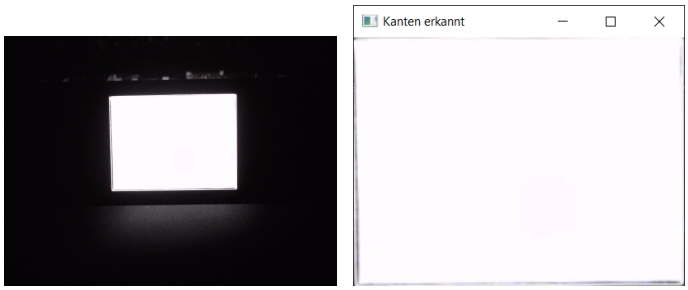
Figure 24: Almost similar UI templates

While no method guarantees absolute detection accuracy, a well-tuned threshold significantly increases the probability of a correct match. A solution is better than none.

4 Evaluation and Results

In the following, a full run-through of the entire cascade — from image recognition and calibration to analysis will be shown. This intentionally include both; high-quality images and some blurry, low-quality ones to test whether the technique holds up.

4.1 Calibration



(a) (b)

Figure 25: Input calibration Image (a); Found UI (b)

The UI was found. Saved corners: [[204.0, 115.0], [447.0, 112.0], [448.0, 295.0], [206.0, 297.0]] (ordered as top-left, top-right, bottom-right, bottom-left) which is correct due we as human would recognize this UI also.

(a) (b)

Figure 26: Input calibration Image (Transformed for a test) (a); Found UI (b)

The UI was found even when an unnaturally transformed import is given. Saved corners: [[112.0, 112.0], [446.0, 115.0], [531.0, 334.0], [204.0, 300.0]]

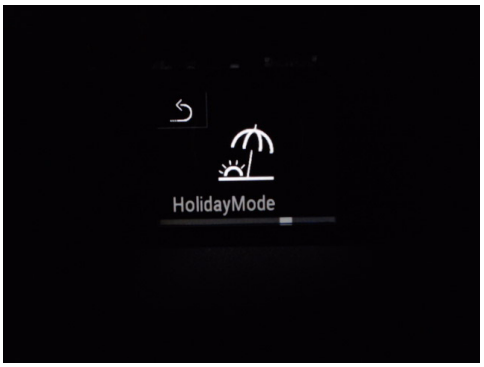


Figure 27: Input calibration Image

The UI was not found. Saved corners: [] which is also correct due we as human can't see any display edges.

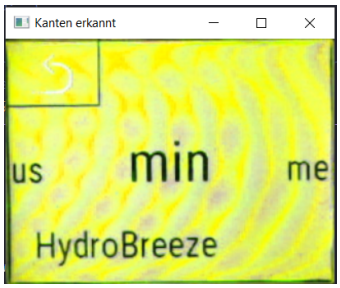
That example is good for calibration, due it shows how important a non black UI is for detecting corners.

The following steps assume that we had saved the correct detected corners from the white image at the beginning from Fig.25.

4.2 Analyze Colors

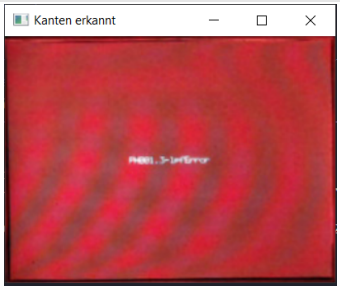
(a)

black:	602
white:	837
red:	27
yellow:	6
green:	12
cyan:	71833
blue:	3389
magenta:	94



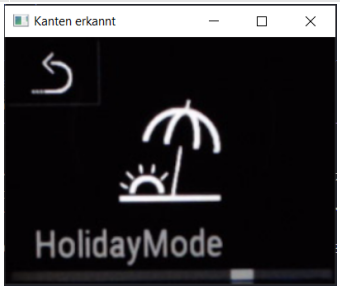
(b)

black:	65
white:	39
red:	6
yellow:	75147
green:	1538
cyan:	2
blue:	3
magenta:	0



(c)

black:	73
white:	38
red:	76655
yellow:	0
green:	1
cyan:	2
blue:	3
magenta:	28



(d)

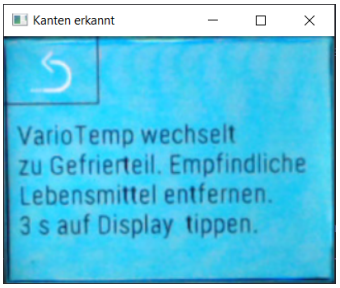
black:	66861
white:	1318
red:	1927
yellow:	74
green:	31
cyan:	337
blue:	3708
magenta:	2544

Figure 28: HSV Color Analysis Results

Even when fractures appear on the display (due to imperfect camera focus), the colors still work effectively for unit tests, as it is sufficient to verify whether we are on an error screen, in warning mode, or in eco mode.

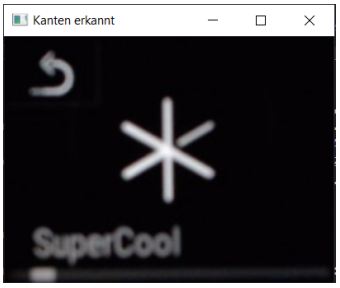
4.3 OCR

(a)
Result: "VarioTemp" **probability:** 0.999



(b)
Result: "VarioTemp wechselt zu Gefrierteil . Empfindliche Lebensmittel entfernen. 3 s auf Display tippen."

(c)
Result: "Kuhlteil 'C BioFresh 21-2 c Gefrienteil -21 'c"



(d)
Result: "SoeCool" **probability:** 0.377

Figure 29: OCR Results

In image (a), the system achieved near-perfect accuracy in recognizing "VarioTemp" owing to excellent image quality. Image (b) demonstrates that the complete sentence was generally well recognized, although one period was rendered with excessive spacing; the remainder of the text was accurately captured. In image (c), recognition issues emerged with smaller characters: the umlaut "ü" and its diacritical marks were not properly detected, the degree Celsius symbol was misinterpreted, and the vertical bar ("|") was mistakenly read as the digit "1." One of the three minus symbols was omitted, while the sequence "rt" was erroneously rendered as "n" due to the proximity of UI elements. Image (d), featuring a deliberately pixelated input, was only partially recognized, yielding a confidence level of 37%.

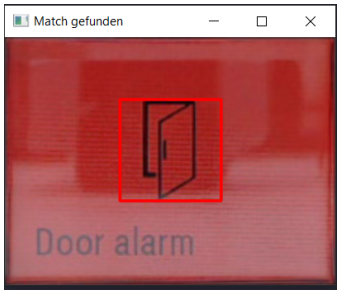
4.4 Multi-Scale Template Matching

In the following, the template is displayed on the left and its corresponding result on the right.

(a) Template

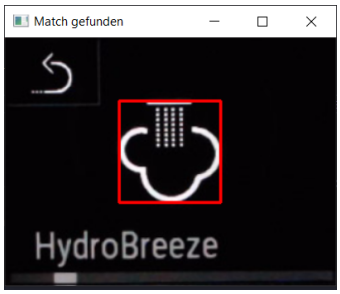
(b) scale: 1.02, maxVal: 0.4986

(c) Template



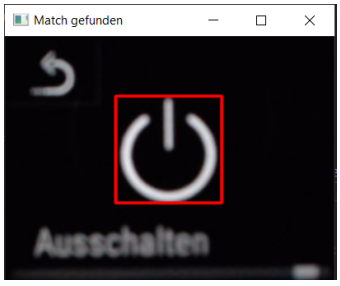
(d) scale: 1.02, maxVal: 0.8010

(e) Template



(f) scale: 1.02, maxVal: 0.5631

(g) Template



(h) scale: 0.97, maxVal: 0.3299

(i) Template

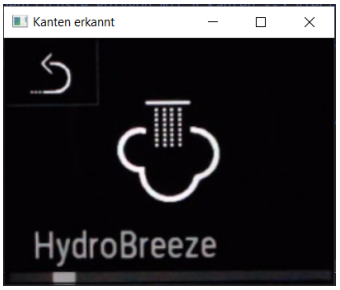
(j) scale: 1.02, maxVal: 0.3919

Figure 30: Multi-Scale Template Matches

Generally, a higher threshold helps minimize false positives; however, to allow even low-quality images to pass, its seen that a minimum threshold of > 0.30 is needed.

Tests were also conducted using an incorrect template as input. None of the resulting maxVal values exceeded 0.22, a positive outcome that stands for the multi-scale template matching approach — a threshold of 0.3 perfectly fits for the application. No instance was found where the program mistakenly matched an incorrect template. As seen below, the images further support these findings.

(a) Template

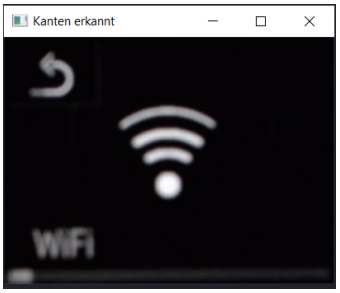


(b) scale: 0.73, maxVal: 0.1037

(c) Template

(d) scale: 1.39, maxVal: 0.2199

(e) Template



(f) scale: 1.45, maxVal: 0.1650

Figure 31: Multi-Scale Template NO Matches

In conclusion, runtime tests were conducted to evaluate the performance of the system. A single run without OCR processing takes approximately 750ms. When incorporating OCR using a deep learning model on the CPU, the total runtime increases to around 3500ms due to the computational demands. However, utilizing a GPU for OCR processing significantly reduces the total runtime back to approximately 750ms.

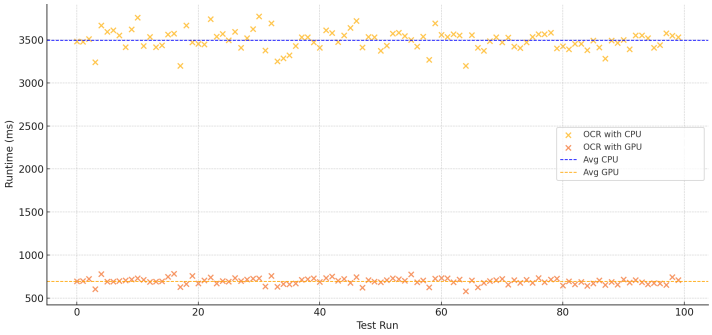


Figure 32: Runtime across 100 tests using CPU and GPU

References

[1] Julian Lingnau. The project - ui detection, 2025. https://github.com/infinite0007/UIDetection Accessed: 2025-03-31.

[2] Tom Yeh and Others. Sikuli: Testing gui screenshots for automation, 2009. https://kleuker.iui.hs-osnabrueck.de/CSI/Werkzeuge/Sikuli/sikuli.html Accessed: 2025-02-12.

[3] Richard Szeliski. Computer Vision: Algorithms and Applications. Springer, 2 edition, 2010-2022. isbn 978-3-642-15703-2 http://szeliski.org/Book.

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. arXiv preprint arXiv:1506.01497, 2015. https://arxiv.org/abs/1506.01497 Accessed: 2025-02-02.

[5] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: Old fashioned or deep learning or a combination? arXiv preprint arXiv:2008.05132, 2020. https://arxiv.org/abs/2008.05132 Accessed: 2025-02-04.

[6] Wikipedia. Histogram equalization. https://en.wikipedia.org/wiki/Histogram_equalization Accessed: 2025-03-15.

[7] Wikipedia. Image normalization. https://en.wikipedia.org/wiki/Normalization_(image_processing) Accessed: 2025-03-16.

[8] Hillary: ThePhoblographer Grigonis. Best cameras for low light autofocus, 2024. https://www.thephoblographer.com/2024/08/09/best-cameras-for-low-light-autofocus/ Accessed: 2025-03-17.

[9] Nathan: Curtin University Glover. Hsv vs. rgb, 2016. https://handmap.github.io/hsv-vs-rgb/ Accessed: 2025-03-24.

[10] Christian: BLUDELTA Weiler. Ocr and deepocr in comparison. https://www.bludelta.de/en/ocr-and-deepocr-in-comparison/ Accessed: 2025-03-28.

[11] Roberto Brunelli. Template Matching Techniques in Computer Vision: Theory and Practice. Wiley, 2009. isbn 978-0-470-51706-2 https://onlinelibrary.wiley.com/doi/book/10.1002/9780470744055.

[12] PyImageSearch. Opencv template matching | cv2.matchtemplate, 2021. https://pyimagesearch.com/2021/03/22/opencv-template-matching-cv2-matchtemplate/ Accessed: 2025-03-25.

[13] Rean Neil Luces: Medium. Template-based versus feature-based template matching, 2019. https://medium.datadriveninvestor.com/template-based-versus-feature-based-template Accessed: 2025-03-29.

[14] Ted Burke. Commandcam, 2011. https://github.com/tedburke/CommandCam Accessed: 2025-02-24.

[15] OpenCV. Opencv documentation – canny edge detection tutorial. https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html Accessed: 2025-03-27.

[16] OpenCV. Opencv documentation – color conversions (rgb hsv). https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html Accessed: 2025-03-18.

[17] Wisotop. Hsv- und hsl-farbmodell. https://wisotop.de/hsv-und-hsl-farbmodell.php. Accessed: 2025-03-18.

[18] EasyOCR. Easyocr: Ready-to-use ocr with 80+ languages, 2021. https://github.com/JaidedAI/EasyOCR Accessed: 2025-03-12.

[19] P. Schwinda and P. d'Angelo. Evaluating the applicability of brisk for the geometric registration of remote sensing images, 2015. https://elib.dlr.de/97889/1/brisk_rm.pdf Accessed: 2025-03-27.

[20] Department of Computer Science, University of Maryland. Homography estimation, 2019. https://www.cs.umd.edu/class/fall2019/cmsc426-0201/files/17_Homography-estimation.pdf Accessed: 2025-03-28.

[21] Kai Briechle and Uwe D. Hanebeck. Template matching using fast normalized cross correlation. https://isas.iar.kit.edu/pdf/SPIE01_BriechleHanebeck_CrossCorr.pdf Accessed: 2025-03-28.

[22] Adrian Rosebrock. Multi-scale template matching using python and opencv. https://pyimagesearch.com/2015/01/26/multi-scale-template-matching-using-python-opencv/, 2015. Accessed: 2025-02-06.