Computer Vision - Project 1: Qwirkle Score Calculator

Vrînceanu Radu-Tudor
Faculty of Mathematics and Computer Science
University of Bucharest

April 2025

1 Background and Game Description

Qwirkle is a tile-laying board game renowned for its blend of strategic depth and accessibility, making it suitable for family play. The core mechanic involves players placing tiles onto a playing surface to form lines based on shared attributes, either color or shape. The game utilizes a set of 108 tiles, each representing a unique combination of one of six distinct shapes (circle, square, diamond, four-point star, clover, and eight-point star) and one of six distinct colors (red, orange, yellow, green, blue, and white). This results in 36 unique shape-color tile types, with each type appearing exactly three times in the full set.

Two primary versions of the game exist: Qwirkle Classic and Qwirkle Connect. In Qwirkle Classic, gameplay unfolds directly on a flat surface, such as a table, without any predefined grid or boundaries. Players place tiles adjacent to existing tiles, expanding the play area organically. The objective remains consistent: score points by forming lines and completing *Qwirkles* – lines containing six tiles that share a common attribute (either color or shape) but are unique in the other attribute.

Qwirkle Connect, the primary focus of this project, introduces a structured playing environment through a dedicated game board. This board is typically modular, composed of four identical quadrants that can be rotated or rearranged before each game, offering layout variability. Each quadrant contains specific marked squares: six black squares used for initial tile placement at the gameś start, ten squares offering a +1 bonus point when a tile is placed on them, and two squares offering a +2 bonus point. When assembled, these quadrants form a continuous playing grid. The presence of the board and bonus squares adds a layer of tactical consideration compared to the classic version.

Regardless of the version, the fundamental placement rules apply. Tiles placed during a playerś turn must form a single continuous line (either horizontally or vertically) and must connect to at least one tile already on the playing surface. All tiles within a newly formed or extended line must share a single attribute (either all the same color or all the same shape) and must not contain duplicate shape-color combinations within that line. A line must consist of at least two tiles; a single tile does not constitute a line for scoring purposes. Invalid placements include exceeding the six-tile limit per line, duplicating a tile within a line, placing a tile unconnected to existing ones, or placing a tile that does not match the attribute of the line it joins.

Scoring is based on the lines created or extended during a playerś turn. Each tile in a valid line contributes one point to that lineś score. If a player places tiles that contribute to multiple lines simultaneously (e.g., placing a tile at the intersection of a row and column), they score points for all affected lines. Completing a line of six unique tiles (a Qwirkle) grants a bonus of six additional points on top of the standard line score. In Qwirkle Connect, placing tiles on bonus squares adds the indicated bonus points (+1 or +2) to the score calculation for that turn. These bonus points are awarded only once per tile placement, even if the tile contributes to multiple lines. The project described herein focuses on automating the process of identifying placed tiles and calculating the resultant score based on these rules, primarily for the Qwirkle Connect variant, with Qwirkle Classic considered as a bonus challenge.

2 Methodology

The automated Qwirkle score calculation system employs a series of computer vision techniques to process game images, identify tile placements, and compute scores according to the rules. The methodology

differs slightly between the primary target, Qwirkle Connect (with a board), and the bonus challenge, Qwirkle Classic (without a board).

2.1 Qwirkle Connect Methodology

For the Qwirkle Connect variant, the presence of a defined board provides a structured reference frame. The process begins with **Board Detection and Warping**. Input images, often captured from various viewpoints (regular, rotated, perspective), first undergo preprocessing, typically involving median blurring to reduce noise and texture interference, as seen in the `preprocessing` utility function. To isolate the board, techniques like HSV color thresholding are used to create a mask focusing on the boardś color range. Edge detection, specifically the Canny algorithm (`cv.Canny`), is applied to the masked or preprocessed image. Morphological operations like closing (`cv.morphologyEx` with `MORPH_CLOSE`) may be used to enhance edge continuity. Contours are then extracted from the edge map using `cv.findContours`. The largest contour, assumed to be the boardś boundary, is selected. The `cv.approxPolyDP` function is used to approximate the contour shape with a polygon, ideally identifying the four corners of the board. These detected corners are ordered consistently (e.g., top-left, top-right, bottom-left, bottom-right) using a helper function like `order_points`. A perspective transformation matrix is calculated using `cv.getPerspectiveTransform`, mapping the detected corners to the corners of a target rectangular image (e.g., 960x960 pixels). The `cv.warpPerspective` function then applies this transformation to rectify the board image, correcting for perspective distortion and providing a top-down, standardized view, as implemented in the `extract_boardgame_from_image` function.

Once the board is warped, **Bonus Square Detection** is performed. The Qwirkle Connect board features squares offering +1 or +2 bonus points. The `extract_points_boardgame_from_image` function demonstrates a method to identify the configuration of these bonus squares. Since the board consists of four identical quadrants that can be rotated, the system needs to determine the orientation of each quadrant. This is achieved by analyzing specific reference points or patterns within the quadrants, potentially using template matching on known unique features within each quadrant type against predefined templates (like the `'2'` symbol mentioned in the codeś template paths). Based on the detected orientation, a 16x16 matrix (`game_points`) representing the boardś bonus values is populated using predefined quadrant layouts (`game_matrix`).

To track changes between game states (i.e., identify newly placed tiles), **Image Alignment** is crucial. Subsequent game images are warped similarly to the initial image. Then, feature-based alignment techniques like SIFT (Scale-Invariant Feature Transform) or ORB (Oriented FAST and Rotated BRIEF) are employed, as seen in `warp_image_to_starting_image_SIFT`. Keypoints and descriptors are computed for both the current warped image and the reference (previous or initial) warped image using `sift.detectAndCompute` or `orb.detectAndCompute`. Descriptors are matched using algorithms like Brute-Force matching (`cv.BFMatcher`, `cv.DescriptorMatcher_create`) often combined with Loweś ratio test (for SIFT) or distance thresholding to find reliable correspondences. From these matches, a homography matrix (`cv.findHomography` with RANSAC) is estimated, representing the geometric transformation (primarily rotation and minor translation/scaling) between the two images. This homography is used with `cv.warpPerspective` to align the current image precisely with the reference image, facilitating difference detection.

Tile Detection and Recognition involves identifying individual tiles on the aligned, warped board. A common approach, evident in the provided code (`extract_pieces_from_board`), is template matching. A library of template images, representing each of the 36 unique shape-color tile types, is prepared. Each template is slid across the warped board image, and a similarity score is computed at each location using `cv.matchTemplate` with a method like `TM_CCOEFF_NORMED`. Locations where the score exceeds a predefined threshold (`THRESHOLD_DETECTION_PIECE`) are considered potential detections. Since a single tile might yield multiple high-score detections around its true location, Non-Maximum Suppression (NMS), often via `cv.dnn.NMSBoxes`, is applied to filter redundant detections and retain only the most confident match for each tile. The recognized tileś identity (shape and color) is determined by the template that produced the best match, converted into a coded format (e.g., `'1R'` for circle-red) using functions like `shape_name_to_code`.

Position Determination assigns grid coordinates to each detected tile. Since the board is warped to a fixed size (e.g., 960x960), the board can be divided into a 16x16 grid. The centroid or top-left corner coordinates of a detected tileś bounding box are mapped to the corresponding grid cell (row 1-16, column A-P). The `extract_code_cell` function exemplifies converting numerical row/column indices into the required alphanumeric format.

Finally, **Score Calculation** integrates the information about newly placed tiles. By comparing the current board state (represented as a 16x16 matrix of tile codes or empty cells) with the previous state, the newly placed tiles are identified. The `compute_score` function implements the scoring logic. For each new tile, it checks horizontally and vertically adjacent tiles to identify the lines formed or extended. The length of each such line is determined. If a line reaches six tiles, a Qwirkle bonus (6 points) is added. The base score for each line is its length. Additionally, the bonus points associated with the grid positions of the newly placed tiles (obtained from the `game_points` matrix) are added, ensuring bonuses are counted only once per tile placement. The total score for the move is the sum of all line scores, Qwirkle bonuses, and position-based bonuses.

2.2 Qwirkle Classic (Bonus) Methodology

The Qwirkle Classic version presents the challenge of operating without a fixed board or coordinate system.

Relative Positioning is adopted. The position of the very first tile placed (specifically, the top-leftmost tile of the initial move) is defined as the origin (0, 0) of a relative Cartesian coordinate system. The positions of all subsequent tiles are calculated relative to this origin. Tile dimensions are estimated or assumed, and the relative grid coordinates (which can be negative) are determined based on the spatial relationship between detected tiles, as implemented in `extract_position` and `set_position_to_matrix`.

Image Alignment remains essential. Similar to Qwirkle Connect, SIFT features are used (`convert_next_image_usin` to detect keypoints and compute descriptors in consecutive images. Matching and homography estimation allow subsequent images to be warped and aligned relative to the first image containing the origin tile. This ensures that relative positions can be consistently tracked across different moves and potential camera shifts.

Tile Detection and Recognition follows a similar template matching approach (`extract_pieces_from_image`) as in Qwirkle Connect, using the same set of 36 shape-color templates and NMS to identify tiles within the aligned image frame.

Score Calculation for Qwirkle Classic is simpler as there are no bonus squares. The logic focuses on identifying newly placed tiles by comparing the current relative grid state with the previous one. Scores are calculated based on the lengths of newly formed or extended lines (horizontal and vertical) and the 6-point bonus for completing a Qwirkle. The scoring logic is embedded within the main processing loop in `qwirkle_classic_code.py`.

2.3 Data Structures

Across both versions, a primary data structure is a 2D array or matrix (e.g., `game_state` in the classic code, implicitly used in connect code). In Qwirkle Connect, this is typically a 16x16 matrix storing the code of the tile in each cell or a marker for an empty cell. In Qwirkle Classic, a larger dynamic grid (e.g., 216x216 centered around the origin) is used to accommodate the potentially sprawling layout, storing tile codes and potentially associated detection confidence or bounding box information.

3 Implementation Details

The project was implemented in Python, leveraging several key libraries for computer vision and numerical operations. The core functionalities are encapsulated within functions designed to handle specific stages of the image processing and game logic pipeline.

3.1 Libraries and Setup

The implementation relies heavily on **OpenCV (cv2)** for image processing tasks such as reading/writing images, color space conversions (BGR to HSV, BGR to Gray), filtering (median blur), morphological operations (opening, closing), edge detection (Canny), contour analysis (`findContours`, `contourArea`, `arcLength`, `approxPolyDP`), feature detection and matching (SIFT, ORB, BFMatcher), geometric transformations (`getPerspectiveTransform`, `warpPerspective`, `findHomography`), and template matching (`matchTemplate`, `dnn.NMSBoxes`). **NumPy** is used extensively for numerical operations, array manipulation, and handling image data as matrices. **Matplotlib** is utilized for visualizing intermediate results and debugging. The **dotenv** library is employed to manage configuration parameters (like dataset paths

and detection thresholds) stored in a `.env` file, promoting separation of configuration from code. Standard Python libraries like **os** and **glob** are used for file system interactions, such as constructing paths and finding template files.

3.2 Qwirkle Connect Implementation

Key functions drive the Qwirkle Connect pipeline:

- `extract_boardgame_from_image`: This function takes an image path, reads the image, resizes it, and applies preprocessing (`preprocessing` utility, likely involving median blur). It uses HSV color thresholding (`cv.inRange`) to isolate the board, followed by Canny edge detection (`cv.Canny`) and morphological closing (`cv.morphologyEx`) to find robust edges. It then finds contours (`cv.findContours`), selects the largest one, approximates its corners (`cv.approxPolyDP`), orders them (`order_points`), and computes a perspective transform (`cv.getPerspectiveTransform`) to warp the board (`cv.warpPerspective`) into a standardized top-down view (e.g., 960x960 pixels). It returns both the warped board and a version with a margin (`warp_with_gap`).
- `extract_points_boardgame_from_image`: This function determines the bonus point layout on the warped board. It iterates through predefined reference locations corresponding to potential quadrant orientations, performs template matching (`cv.matchTemplate`) using specific templates (e.g., `2_*.png`) within small patches around these locations to identify the rotation of each quadrant. Based on the best matches, it populates a 16x16 NumPy array (`game_points`) representing the bonus values (+1, +2, or 0) for each square, using pre-defined quadrant layouts (`game_matrix`).
- `warp_image_to_starting_image_SIFT`: Aligns a subsequent game image to the initial reference image. It extracts SIFT features (`cv.SIFT_create`, `detectAndCompute`) from both images, matches them using a Brute-Force matcher (`cv.DescriptorMatcher_create`), and computes a homography matrix (`cv.findHomography` with RANSAC). This matrix is then used to warp (`cv.warpPerspective`) the current game image to align with the reference frame. An ORB-based version (`warp_image_to_starting_image_O`) exists but SIFT appears preferred in the final connect pipeline.
- `extract_pieces_from_board`: This function performs tile detection on the aligned, warped board. It iterates through color categories and corresponding template files (`templates_config`). For each template, it resizes it, applies blurring/morphology, and uses `cv.matchTemplate` (with `TM_CCOEFF_NORMED`) across the board image. Detections exceeding `THRESHOLD_DETECTION_PIECE` are collected. Non-Maximum Suppression (`cv.dnn.NMSBoxes`) is applied to eliminate redundant detections for the same tile. The function returns a list of detected tiles, including their code (derived using `shape_name_to_code`), bounding box, and confidence score.
- `shape_name_to_code`: Converts template filenames and color categories into the standard tile code (e.g., `1R`, `6B`) based on shape names extracted via regex (`re.search`).
- `extract_code_cell`: Converts numerical 0-based row and column indices into the required 1-based alphanumeric grid position (e.g., `1A`, `16P`).
- `compute_score`: Calculates the score for a move. It takes the current board state (16x16 matrix), the previous state, and the bonus points matrix. It identifies newly placed tiles by comparing the states. For each new tile, it traverses horizontally and vertically to find connected lines, calculates line lengths, checks for Qwirkles (length 6), adds line scores and Qwirkle bonuses (6 points), and incorporates bonus points from the `game_points` matrix, ensuring each bonus square is counted only once per move. It returns the total score and the list of newly placed tiles with positions and codes.

The overall workflow involves processing the initial image (`g_00.jpg`) to get the reference warped board and bonus layout. Then, for each subsequent image (`g_01.jpg` to `g_20.jpg`), it is warped, aligned to the reference using SIFT, tiles are detected via template matching and NMS, the board state matrix is updated, and finally, the `compute_score` function determines the newly placed tiles and the score for that move. Results are written to text files (`OUTPUT_PATH`) in the specified format.

3.3 Qwirkle Classic (Bonus) Implementation

The bonus implementation adapts the methodology for the boardless scenario:

- `convert_next_image_using_SIFT`: Similar to the Connect version, this aligns subsequent images to the first image (`1_01.jpg`) which establishes the origin. It uses SIFT features, matching, and homography estimation (`cv.findHomography` with RANSAC) to warp (`cv.warpPerspective`) images into a common reference frame based on the initial tile placement.
- `extract_pieces_from_image`: This function is largely the same as in the Connect version, using template matching (`cv.matchTemplate`, `TM_CCOEFF_NORMED`) with the same set of templates and NMS (`cv.dnn.NMSBoxes`) to detect tiles in the (potentially warped) image. It returns detected tile codes, bounding boxes, and scores.
- `extract_position`: Calculates the relative Cartesian coordinates (x, y) for each detected tile based on its pixel coordinates (`x`, `y` from the bounding box) relative to the origin tile 's coordinates (`origin_x`, `origin_y`), using an estimated `tile_size`.
- `set_position_to_matrix`: Converts the relative Cartesian coordinates (`coord_x`, `coord_y`) into indices for a large 2D array (`game_state`, e.g., 216x216) centered around the origin.
- `filter_pieces`: Appears to refine detections, possibly by selecting the highest-scoring detection if multiple templates match at nearly the same location.

The classic workflow establishes the origin using the top-leftmost tile detected in the first move image (`1_01.jpg`). Subsequent images are aligned to this first image using `convert_next_image_using_SIFT`. Tiles are detected using `extract_pieces_from_image`. Their relative positions are calculated (`extract_position`) and stored in the `game_state` matrix (`set_position_to_matrix`). The code iterates through images, detecting tiles and updating the `game_state`. The scoring logic seems less explicitly modularized compared to the Connect version but involves identifying new tiles by comparing states and calculating scores based on line lengths and Qwirkle bonuses (implicitly, as the provided snippet writes "0" for the score). Output files are generated in the specified format, listing relative coordinates and tile codes.

4 Results and Evaluation

The project 's success is evaluated based on its ability to correctly process test images and output accurate information regarding tile placements and scores, adhering strictly to the format specified for automated evaluation. The evaluation process, as detailed in the project description PDF, focuses on the system 's performance across a set of 100 test images (excluding the initial configuration images for each of the 5 games).

For each test image corresponding to a player ś move, the system is required to generate a text file containing:

1. The grid positions (e.g., `1A`, `16P` for Connect; relative coordinates for Classic) of all tiles newly placed during that move.
2. The code representing the shape and color (e.g., `1R`, `6W`) for each newly placed tile at its corresponding position.
3. The total score achieved by the player for that specific move, calculated according to the game rules (including line scores, Qwirkle bonuses, and Connect bonus square points).

The grading scheme assigns points based on the accuracy of these outputs for each test image. Specifically, correctly identifying the positions of newly added tiles contributes 0.02 points per image, correctly identifying the tile types (shape and color) contributes 0.015 points per image, and correctly calculating the score contributes 0.005 points per image, summing to a potential 0.04 points per correctly processed move image. An additional 0.5 points are awarded ex officio if the submitted results strictly follow the specified format, allowing the provided evaluation script (`code` directory in the evaluation data) to run without issues. A mandatory oral presentation accounts for another 0.5 points.

The provided code implementations for both Qwirkle Connect and Qwirkle Classic are structured to process input images from the specified dataset paths (`TRAIN_DATASET_CONNECT_PATH`, `TRAIN_DATASET_CLASSIC_PATH`)

and generate output text files in the designated `OUTPUT_PATH` directory, matching the required format. The code includes steps for image alignment, tile detection, position determination, and score calculation, demonstrating the capability to address the evaluation criteria. While the code snippets primarily show processing loops for the training data (e.g., iterating through images `1_01.jpg` to `1_20.jpg`), the structure is designed to be applied similarly to the test dataset upon its release. The accuracy of the results produced by this implementation on the final test set would determine the final score according to the outlined criteria. Explicit performance metrics (e.g., percentage of correctly identified tiles or scores) are not available within the provided code context, as they depend on execution against the unseen test data and ground truth.

5 Conclusion and Future Work

This project successfully outlined and implemented a computer vision system designed to automate the score calculation for the game Qwirkle, with a primary focus on the Qwirkle Connect variant and a bonus consideration for Qwirkle Classic. By leveraging techniques such as image preprocessing, board detection via contour analysis and perspective warping, feature-based image alignment (SIFT), template matching with Non-Maximum Suppression for tile recognition, and rule-based score computation, the system demonstrates a viable approach to interpreting game states from images and determining player scores.

The methodology addresses key challenges, including variations in image perspective and rotation, the need to identify specific bonus squares in Qwirkle Connect, and the requirement for relative positioning in the boardless Qwirkle Classic scenario. The implementation utilizes standard Python libraries like OpenCV and NumPy, providing a foundation for processing the provided image datasets and generating outputs in the specified format for evaluation.

Future work could explore several avenues for improvement and extension. Robustness to varying lighting conditions could be enhanced, perhaps through adaptive thresholding or more advanced color constancy algorithms. The template matching approach for tile recognition, while effective, could potentially be replaced or augmented by machine learning models (e.g., Convolutional Neural Networks) trained specifically on tile images, which might offer better generalization to different tile appearances or partial occlusions. The alignment process using SIFT/ORB could be further optimized for speed and accuracy, especially for significant perspective changes. For Qwirkle Classic, developing more sophisticated methods for estimating the grid layout and handling tile drift or slight movements between turns could improve positional accuracy. Additionally, integrating optical character recognition (OCR) for potential future game variants or score pads could be considered. Finally, refining the score calculation logic to handle edge cases or ambiguities even more robustly would contribute to the system
's overall reliability.

A Code Snippets

This appendix includes key code snippets from the implementation for both Qwirkle Connect and Qwirkle Classic.

A.1 Qwirkle Connect: Board Extraction and Warping

```
def extract_boardgame_from_image(image_name, path_to_dataset=TRAIN_DATASET_CONNECT_PATH):
    first_image_path = os.path.join(path_to_dataset, f"{image_name}.jpg")
    first_image = cv.imread(first_image_path, cv.IMREAD_COLOR_BGR)
    first_image = cv.resize(first_image, None, fx=0.8, fy=0.8)

    first_image_preprocessing = preprocessing(first_image) # get rid of textures by using median blur filters (texture can bring up noise in corner detection)

    hsv_threshold_low, hsv_threshold_high = (15, 0, 0), (255, 255, 255)
    hsv_first_image_preprocessing = cv.cvtColor(first_image_preprocessing, cv.COLOR_BGR2HSV)
    board_mask = cv.inRange(hsv_first_image_preprocessing, hsv_threshold_low, hsv_threshold_high)

    first_board_image = cv.bitwise_and(first_image, first_image, mask=board_mask)
```

```
    median_pixel_value = np.median(first_board_image)
    lower_threshold = int(max(0, (1.0 - 0.33) * median_pixel_value))
    upper_threshold = int(min(255, (1.0 + 0.33) * median_pixel_value))

    edges = cv.Canny(first_board_image, lower_threshold, upper_threshold)
    kernel = np.ones((3, 3), np.uint8)
    dilated_edges = cv.morphologyEx(edges, cv.MORPH_CLOSE, kernel, iterations=3)

    contours, _ = cv.findContours(dilated_edges, cv.RETR_EXTERNAL, cv.
    CHAIN_APPROX_SIMPLE)

    maximum_contour_area, best_contour_idx = 0.0, 0
    for (idx, contour) in enumerate(contours):
        area = cv.contourArea(contour)
        if area > maximum_contour_area:
            best_contour_idx = idx
            maximum_contour_area = area

    contour = contours[best_contour_idx]
    perimeter = cv.arcLength(contour, True)
    corners = cv.approxPolyDP(contour, 0.02 * perimeter, True)
    corners = order_points(corners)

    margin = 16
    height_with_gap, width_with_gap = 960 + margin*2, 960 + margin*2
    warp_to_with_gap = np.array([
        [margin, margin],
        [width_with_gap - margin, margin],
        [margin, height_with_gap - margin],
        [width_with_gap - margin, height_with_gap - margin]
    ]).astype(np.float32)
    perspective_matrix_with_gap = cv.getPerspectiveTransform(corners.astype(np.float32),
     warp_to_with_gap)

    height, width = 960, 960
    warp_to = np.array([[0, 0], [width, 0], [0, height], [width, height]]).astype(np.
    float32)
    perspective_matrix = cv.getPerspectiveTransform(corners.astype(np.float32), warp_to)

    return (cv.warpPerspective(first_image, perspective_matrix, dsize=(width, height)),
            cv.warpPerspective(first_image, perspective_matrix_with_gap, dsize=(
    width_with_gap, height_with_gap)))
```

Listing 1: Function to detect the board and warp perspective (from qwirkle_connect_code.py)

A.2 Qwirkle Connect: Tile Detection

```
def extract_pieces_from_board(image, starting_image):
    detections = defaultdict(list)
    copy_image = image.copy()

    height, width = image.shape[:2]
    patch_height, patch_width = height // 16, width // 16

    for color_name, template_paths in templates_config.items():
        for template_path in template_paths:
            template = cv.imread(template_path, cv.IMREAD_COLOR_BGR|cv.
    IMREAD_IGNORE_ORIENTATION)
            template = cv.resize(template, None, fx=1.2, fy=1.2)
            template = cv.medianBlur(template, 3)
            template = cv.morphologyEx(template, cv.MORPH_CLOSE, np.ones((3,3), np.uint8
    ), iterations=1)
            h, w, _ = template.shape

            res = cv.matchTemplate(image, template, cv.TM_CCOEFF_NORMED)
            loc = np.where(res >= THRESHOLD_DETECTION_PIECE)

            boxes = []
            scores = []

            for pt in zip(*loc[::-1]):
```

```
                box = [pt[0], pt[1], w, h]  # x, y, width, height
                boxes.append(box)
                scores.append(res[pt[1], pt[0]])

        indices = cv.dnn.NMSBoxes(boxes, scores, score_threshold=
    THRESHOLD_DETECTION_PIECE, nms_threshold=0.3)
        for i in indices:
            i = i[0] if isinstance(i, (tuple, list, np.ndarray)) else i
            x, y, bw, bh = boxes[i]

            center_x, center_y = x + bw // 2, y + bh // 2
            grid_x, grid_y = center_y // patch_height, center_x // patch_width

            detections[(grid_x, grid_y)].append((shape_name_to_code(color_name,
    template_path), boxes[i], scores[i]))
            # cv.rectangle(copy_image, (x, y), (x + bw, y + bh), (255, 255, 0), 2)
            # cv.putText(copy_image, f"{shape_name_to_code(color_name, template_path
    )} {scores[i]:.2f}", (x, y - 10), cv.FONT_HERSHEY_SIMPLEX, 0.9, (36,255,12), 2)

    # plt.figure(figsize=[40, 10])
    # plt.axis("off"); plt.imshow(cv.cvtColor(copy_image, cv.COLOR_BGR2RGB)); plt.title
    ("Detections")

    final_detections = {}
    for key, value in detections.items():
        value.sort(key=lambda x: x[2], reverse=True)
        final_detections[key] = value[0]

    return final_detections
```

Listing 2: Function for detecting tiles using template matching (from qwirkle_connect_code.py)

A.3 Qwirkle Classic: Image Alignment

```
def convert_next_image_using_SIFT(first_image, next_image):
    # preprocess both
    gray1 = preprocess(first_image)
    gray2 = preprocess(next_image)

    # SIFT detect & describe
    sift = cv.SIFT_create()
    kps1, desc1 = sift.detectAndCompute(gray1, None)
    kps2, desc2 = sift.detectAndCompute(gray2, None)

    # k-NN match + Lowe\\'s ratio test
    bf = cv.BFMatcher()
    knn_matches = bf.knnMatch(desc1, desc2, k=2)
    good_matches = [m for m,n in knn_matches if m.distance < 0.7 * n.distance]

    if len(good_matches) < 8:
        raise RuntimeError(f"Too few matches ({len(good_matches)})")

    # extract points
    pts1 = np.float32([kps1[m.queryIdx].pt for m in good_matches]).reshape(-1,1,2)
    pts2 = np.float32([kps2[m.trainIdx].pt for m in good_matches]).reshape(-1,1,2)

    # robust homography
    H, mask = cv.findHomography(
        pts2, pts1,
        method=cv.RANSAC,
        ransacReprojThreshold=3.0,
        maxIters=2000,
        confidence=0.995
    )

    # warp
    h, w = first_image.shape[:2]
    warped = cv.warpPerspective(
        next_image, H,
        dsize=(w,h),
        flags=cv.INTER_LINEAR,
        borderMode=cv.BORDER_CONSTANT
```

```
    )
    return warped, (kps1, kps2, good_matches, mask)
```

Listing 3: Function to align subsequent images using SIFT (from qwirkle_classic_code.py)

A.4 Qwirkle Classic: Relative Position Calculation

```
def extract_position(detections, origin_x, origin_y):
    tile_size = 240; rel_positions = [] # Assuming tile_size, might need calibration
    for code, (x, y, w, h), score in detections:
        gx = round((x - origin_x) / tile_size)
        gy = round((y - origin_y) / tile_size)
        rel_positions.append((code, gx, -gy, score)) # Inverting y for typical Cartesian
    return rel_positions

def set_position_to_matrix(n, coord_x, coord_y):
    origin = n // 2
    position_x = origin - coord_y # Map Cartesian y to matrix row (inverted)
    position_y = origin + coord_x # Map Cartesian x to matrix column
    return position_x, position_y
```

Listing 4: Functions for calculating relative positions (from qwirkle_classic_code.py)

A.5 Qwirkle Connect: Extraction of game board

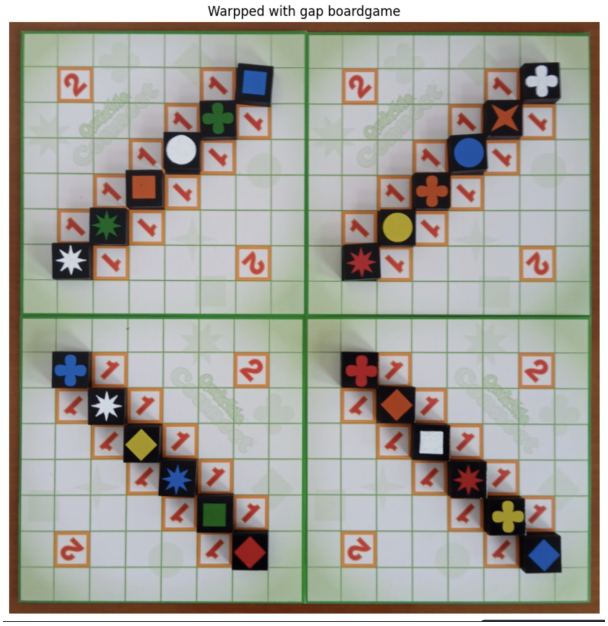

Figure 1: Game board of Qwirkle Connect.

A.6 Qwirkle Classic: SIFT

Figure 2: SIFT features to get the transformation of the next game image.