University of Bayreuth

Bachelor's Thesis

Scoring Board Games with Computer Vision

Lukas Waymann

supervised by

Dr. Oleg Lobachev

and

Prof. Dr. Michael Guthe

2018-04-23

End-of-game scoring is involved and error-prone for some board games. This thesis evaluates the feasibility of automating it for two, *Take it Easy!* and *Kingdom Builder*, using photos taken with camera phones after playing. Recognition schemes that extract the needed information from a photo using traditional computer vision techniques, such as feature detection and template matching, are presented for both games.

The results achieved for *Take it Easy!* are 100 % accurate across 33 test photos, but *Kingdom Builder* may call for other methods to approach human-level performance.

Brettspielendpunktestandbestimmung mit maschinellem Sehen

Lukas Waymann

2018-04-23

Die Bestimmung des Endpunktestands mancher Brettspiele ist langwierig oder fehleranfällig. Diese Arbeit beschäftigt sich mit der Machbarkeit einer automatisierten Auswertung für zwei Spiele: *Take it Easy!* und *Kingdom Builder*. Für beide Spiele werden Vorgehensweisen zur Erkennung der dafür notwendigen Informationen aus mit Smartphones erstellten Photos vorgestellt. Dabei finden traditionelle Methoden des maschinellen Sehens, wie Bildmerkmaldetektion und Template-Matching, Anwendung.

Die Erkennung für *Take it Easy!* ist über 33 Testbilder 100 % fehlerfrei, aber für *Kingdom Builder* könnten andere Ansätze notwendig sein, um ausreichende Robustheit zu erreichen.

Acknowledgments

For introducing the game of *Take it Easy!* to me, which led to the idea for this thesis, and for lending me his copy for much longer than initially planned, I want to thank Florens Winkler. I am very grateful to my advisor, Dr. Oleg Lobachev, for readily accepting my self-chosen subject, for his guidance, and for his many insightful comments and suggestions.

I appreciate all the help I got at collecting photographs for testing my attempts to build robust recognition pipelines—from Jingyi Li, from my family, and from Karina Hinz. Furthermore, I thank everyone that proofread parts of this thesis, especially Jialiang Pan and my sister.

I want to thank my parents for their patience and constant support.

Contents

List of Figures

Chapter 1

Introduction

What one fool can do, another can.

Ancient Simian Proverb

For some board games, figuring out how many points all players have after the game ended can be error-prone, time-consuming, or both. Taking a photo with a smartphone that then automates scoring and displays the number of points reached by each player would be convenient. Robust recognition of a game's various playing pieces would enable this and sidestep the human tendency to overlook something or lose count. If that recognition takes no more than a few seconds, the time-consuming aspect of scoring would also be removed. Current computer vision techniques appear to be well past the point where such scoring programs are feasible for many board games.

To test this hypothesis, I developed and implemented methods for recognition of the game elements relevant to scoring for two games—*Take it Easy!* (Chapter 2), and *Kingdom Builder* (Chapter 3). A single photo is the input each time. My implementations[1] use version 3.6 of the Python programming language [16] and a number of third-party libraries; most importantly NumPy [12], OpenCV [2], and scikit-learn [14]. I have not integrated either recognition scheme into a smartphone app, but did observe the performance constraints of running on modest hardware and producing results at least as fast as a human could.

This thesis is structured as follows. Chapters 2 and 3 both start by introducing the board game that is their respective subject and explain the scoring rules. With the concrete goal of locating *Take it Easy!* boards, Section 2.1 also provides a general overview of feature-based image alignment, a standard technique that is likewise reused in Section 3.1 for *Kingdom Builder*. Section 2.2 explains template matching and how it forms the core of the *Take it Easy!* recognition, and Section 2.3 makes robustness improvements using a color classifier built upon k-nearest neighbors (k-NN). Section 3.2 describes the settlement token recognition for *Kingdom Builder*. Discussions of the accomplished results conclude both chapters. Lastly, Chapter 4 provides a summary and an outlook.

[1] The full source code is available at `https://github.com/meribold/hummerhorn`.

Chapter 2

Take it Easy!

> *We do these things not because they are easy, but because we thought they were going to be easy.*
>
> The Programmers' Credo

Take it Easy! is an abstract board game created by Peter Burley and first published in 1983 [18, p. 230]. Each player starts with an empty board like the one shown in Figure 2.1 and a set of 27 (3^3) unique hexagonal tiles of which they will place 19 on their board. The result may look like Figure 2.2.

Before the game starts, one player is designated the *caller*. This player draws one of their remaining tiles at random each turn, telling everyone else which one it is. All players then place their copy of this tile on any still empty hexagon of their board. After 19 turns, all boards are completely filled and the game ends.

The objective is to complete unbroken lines reaching from one edge of the playable area to the other. These lines can have any of three possible directions (vertical and two diagonal ones) and consist of 3 to 5 tiles with same-colored stripes in that direction. For each unbroken line, a player is awarded points equal to the product of the number corresponding to the line's color and the line's length. For example, a vertical yellow line at the middle of the board like in Figure 2.2 is worth $5 \cdot 9 = 45$ points.

In my experience, people use aids to calculate their score: my dad used pen and paper, my granddad fetched his pocket calculator, but everyone else I played with used their phone's calculator app. This makes *Take it Easy!* an ideal candidate for automated scoring using a photo from a smartphone's built-in camera as input.

2.1 Locating the board

The first step toward identifying all of the tiles on a photographed game board such as the one in Figure 2.3 is to locate the board in the photograph. The perspective can then be corrected so that opposite edges of the photographed board are parallel and corners form right angles. Any in-plane rotation can be removed and areas outside of the board cropped. Figure 2.10 on page 26 illustrates the goal of this section.

Locating the board is robustly accomplished using *feature detection* and *description* algorithms like SIFT [9] or SURF [1] on an idealized query image of the board and the

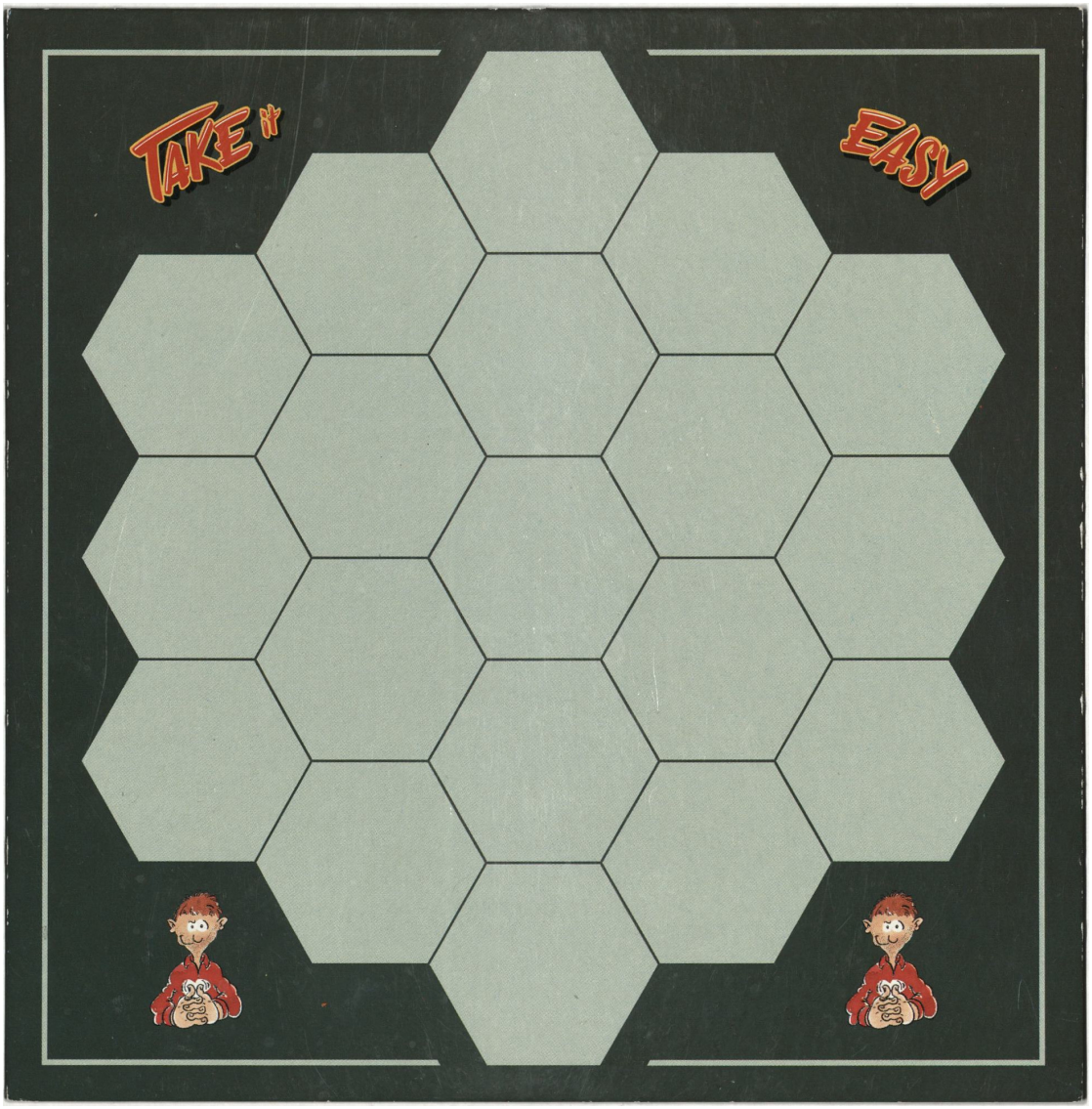


Figure 2.1: An empty *Take it Easy!* board. In the older version of the game that I'm using here, every player's board has a different background color. In a more recent release all the boards are identical. (Also, the box the game comes in got much, much bigger, with lots of padding inside.)

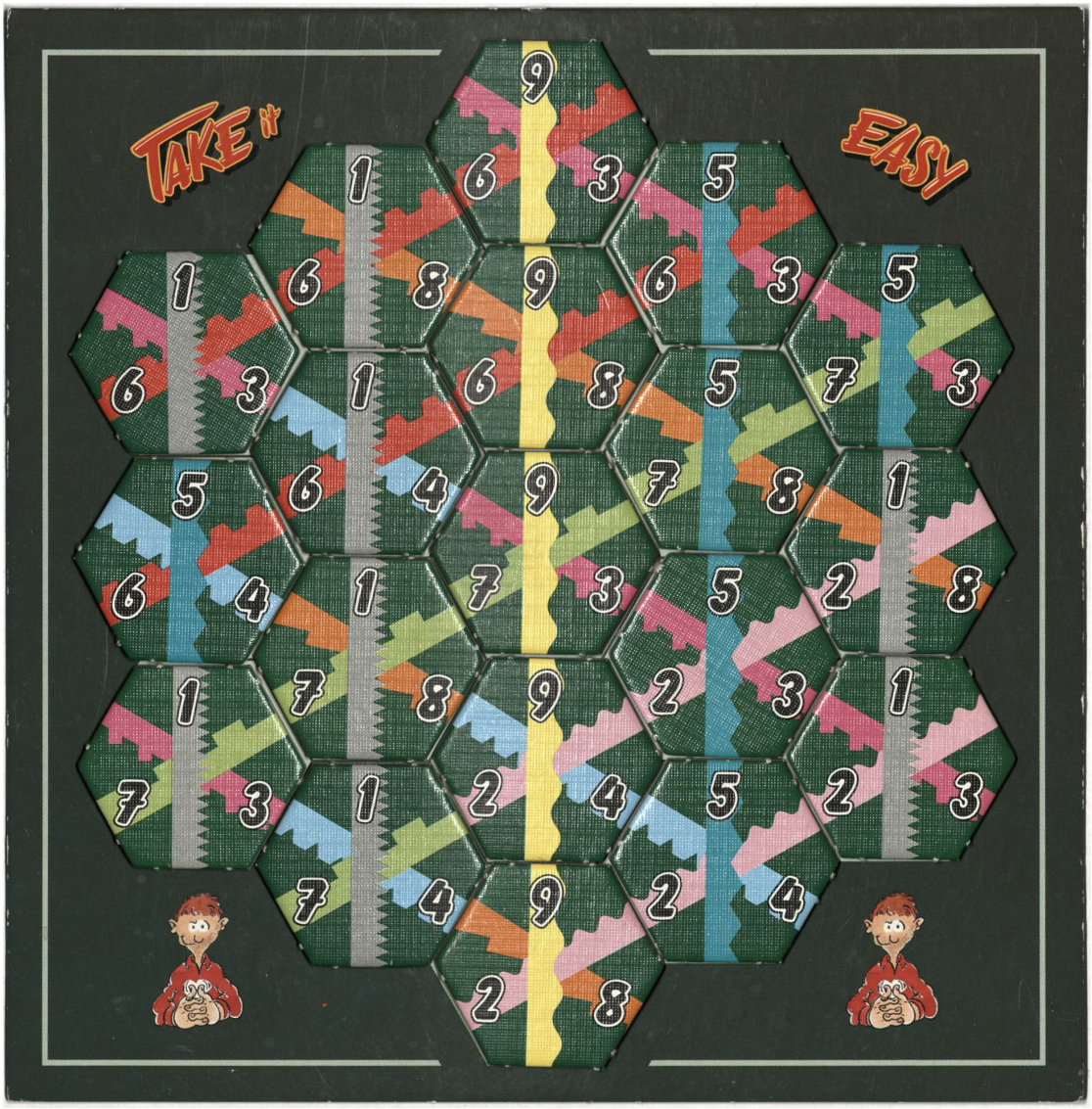


Figure 2.2: A filled *Take it Easy!* board. The numbers have to be upright; tiles may not be turned. You can convince yourself that the score is 193... or maybe that it's easy to lose count when not using any aids such as a calculator or a piece of paper. Random trivia: the maximum possible score is 307.

Figure 2.3: Photograph of a filled *Take it Easy!* board. This is one of 33 photos I used to evaluate the performance of my code.

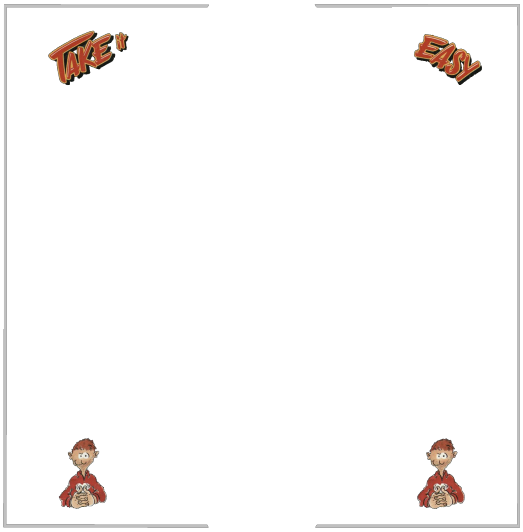


Figure 2.4: Query image for locating *Take it Easy!* boards created from a scan using the GIMP. The image is mostly transparent since the boards have different background colors. The writing at the top and the two cartoon characters are always the same, though.

photo, followed by *feature matching*. The results are pairs of corresponding points, one in the query image and one in the photo each, from which a projective transformation can be established.

A feature of an image is a distinctive point or region of interest that we hope to be able to reidentify in another photo with the same object—even in case of differing viewpoints, lighting conditions, scaling, rotation, etc. After being detected, these features must still be *described*, that is, the information from the region's pixels has to be transformed into a *feature vector* which should be largely unaffected by changes to viewing conditions such as those enumerated above.

Finally, the two sets of feature vectors from a pair of images can be matched. The first image may be a query image of an object (and only that object) under near-ideal conditions and the second a test image for which we wish to establish if and where it contains that object. The query image I have created to find *Take it Easy!* boards in photographs is shown in Figure 2.4 and I am using SURF to extract features.

2.1.1 SURF

SURF is a feature detector and descriptor proposed by Bay, Tuytelaars, and Van Gool in 2006 [1]. It is a "popular variant of SIFT" [15, p. 223]: Bay et al. describe it as "based on similar properties, with a complexity stripped down even further" [1, p. 409].

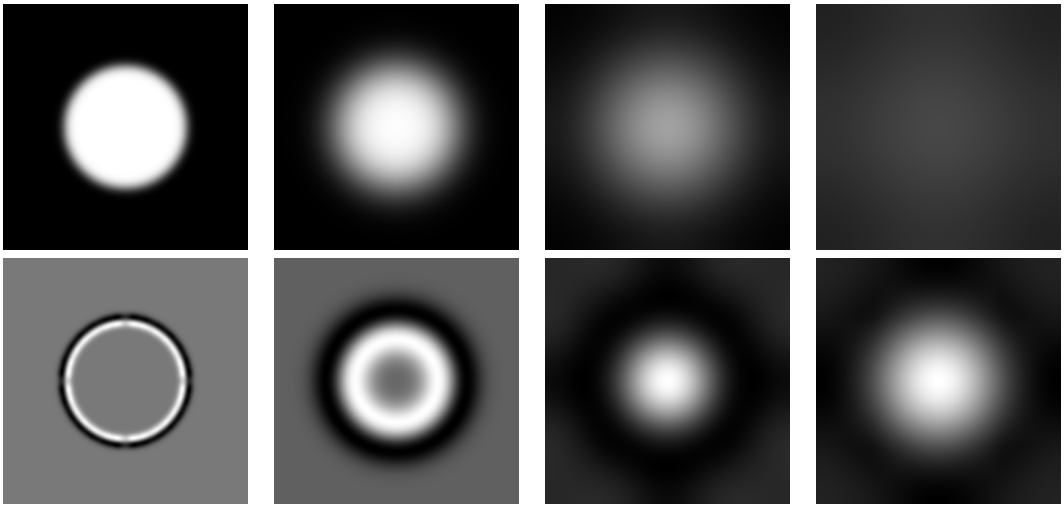
Interest point detection The kind of features SURF and SIFT find are *blob-like* structures. Roughly, these can be thought of as relatively homogeneous regions that differ in brightness[1] compared to their surroundings. The detection stage of SURF approximates finding local maxima of the *scale-normalized determinant* of the *Hessian matrix* in *Gaussian scale space*.

What does the determinant of the Hessian matrix (DoH) have to do with the above intuition of blobs? The eigenvalues λ_1 and λ_2 of the Hessian at any coordinate pair (x, y) of an image are the principal[2] curvatures at that location. Its determinant is the product $\lambda_1 \cdot \lambda_2$ of these eigenvalues (this is known as the Gaussian curvature). Thus, maxima of the DoH are locations with significant curvature in all directions.

This is not what we wanted though: local maxima of the DoH will resemble point-like features, like a single bright pixel on dark background. The center of a monochrome bright circle on dark background, on the other hand, has a Gaussian curvature of 0. Yet, if the circle were blurred with a sufficiently strong Gaussian kernel, its center would become a local curvature peak (see Figure 2.5), which is one reason maxima of the DoH are determined in *scale space*.

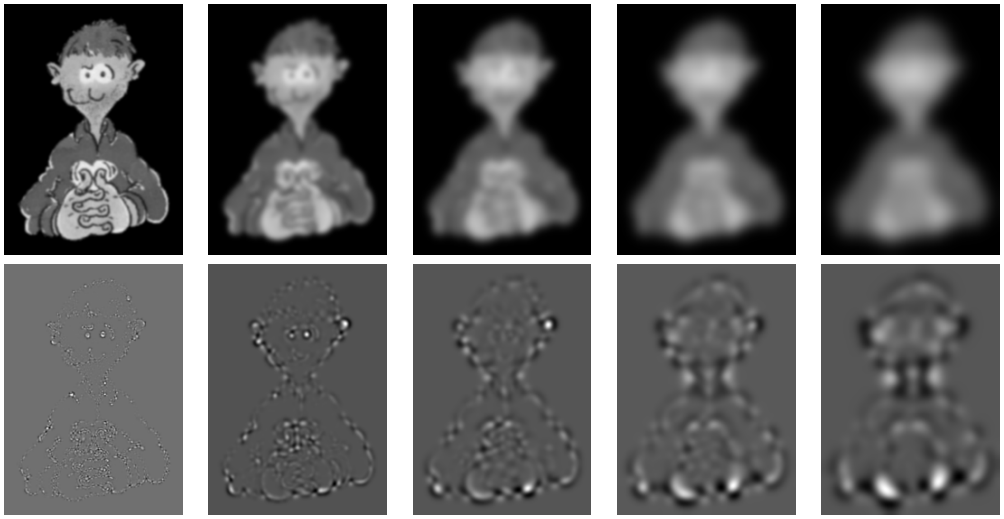
The scale space of an image is a family of derived images obtained by convolution with a Gaussian function (see the top row of Figure 2.6). The variance $t = \sigma^2$ of the Gaussian is the third dimension of the scale space. Because Gaussian smoothing naturally reduces the values of derivatives, a *scale-normalized* derivative operator is used to compute

[1] SIFT and SURF don't use color [9, p. 8, 1, p. 405].
[2] minimum and maximum



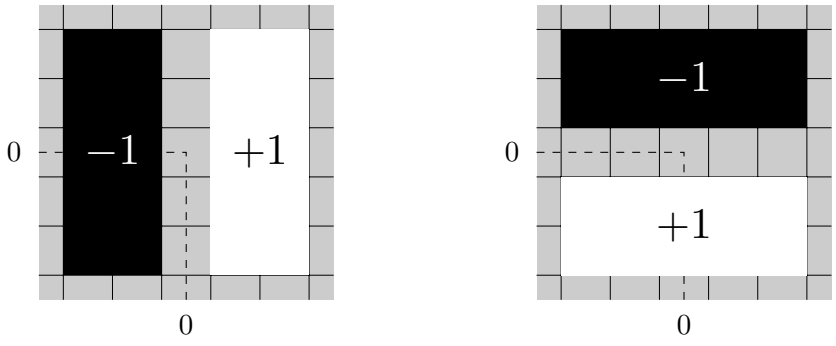
$\sigma = 8$ $\sigma = 32$ $\sigma \approx 71$ $\sigma = 128$

Figure 2.5: A white circle with a radius of 100 pixels on black background at different scales (blurred with successively stronger Gaussian kernels) and the corresponding DoH responses below. In the DoH images, pixels with a value of zero are gray and black pixels have negative values. Also note that the response images are normalized: equal gray levels don't correspond to equal values across images. The maximum scale-normalized response is observed when $\sigma \approx 71$, from which the circle's radius can be calculated as $\sqrt{2} \cdot 71 \approx 100$.



$\sigma = 1$ $\sigma = 3$ $\sigma = 5$ $\sigma = 7$ $\sigma = 9$

Figure 2.6: The cartoon character that is printed on *Take it Easy!* boards at different scales and the corresponding DoH responses below. Dark as well as bright blobs result in positive responses: both principal curvatures are negative for bright blobs, so the product is positive.



Kernel for the x-coordinate Kernel for the y-coordinate

Figure 2.7: Convolution kernels used by SURF to obtain orientation vectors for pixels inside a disk around a feature's location. The 5×5 kernels shown here correspond to the smallest scale at which SURF detects features.

the entries of the Hessian matrix. This operator was introduced by Lindeberg, who formulated a *principle for automatic scale selection* [7, p. 83]:

> In the absence of other evidence, assume that a scale level, at which some (possibly non-linear) combination of normalized derivatives assumes a local maximum over scales, can be treated as reflecting a characteristic length of a corresponding structure in the data.

Lindeberg provides theoretical and empirical justification for this principle.

In summary, finding DoH maxima with respect to scale in addition to image coordinates allows detecting blobs of arbitrary size and simultaneously provides estimates of blob sizes. SURF heavily approximates the interest point detection scheme outlined above using box filters, but the general idea is retained.

Orientation assignment Before feature vectors are constructed, an orientation is assigned to each detected interest point. This orientation is used when extracting information to describe the feature later and ensures that rotating the image has no effect on a feature's descriptor, i.e., feature vectors will be invariant to rotation. The specific orientation of a feature doesn't have any merit in itself, other than being reproducible.

How are orientations determined? For each interest point, every pixel in a disk around this point is first assigned an orientation vector individually. The size of the disk is proportional to the feature's scale. Then, for each angle from a sample of the interval $[0, 2\pi)$,[3] the entries of orientation vectors with agreeing angle ($\pm\pi/6$) are accumulated into weighted sums,[4] yielding a new vector. From these result vectors—one for each angle

[3] Bay, Tuytelaars, and Van Gool [1] don't go into detail on how angles should be sampled, but Oyallon and Rabin [13, p. 202] use 40 uniformly distributed angles in their implementation.

[4] Vectors of close pixels are weighted more strongly than those of far-away pixels. The weights are outputs of a Gaussian function centered at the interest point.

from the considered sample of $[0, 2\pi)$—the longest is selected and provides the feature's orientation.

But how are the individual orientation vectors determined? Two kernels like those shown in Figure 2.7 are convolved with the part of the image that falls into the disk.[5] The x-coordinates of orientation vectors are results from the first convolution, and the y-coordinates results from the second. The side length of these kernels is equal to $2/3$ of the disk's radius and therefore also proportional to the feature's scale.

Feature description SURF descriptors are constructed from square regions centered on interest points. Each square is rotated to align with its feature's orientation and, once again, the size is proportional to the feature's scale. Squares then are partitioned into 16 identical subsquares, like this: ▦. Finally, 4 numbers are computed per subsquare. (SURF therefore describes features with $4 \cdot 16 = 64$ numbers apiece.)

Here's how the 4 numbers come about. Within a subsquare, convolutions with the same kind of kernels that were already used to obtain orientation vectors[6] are computed at 25 evenly distributed points: ⁘ (this box represents *one* of the 16 tiny squares in the previous graphic). Then, the convolution results are weighted based on the distance to the interest point.[7] The 4 numbers all are sums of these weighted results. The respective summands are: 1) the 25 results from the first kernel, 2) the 25 results from the second kernel, 3) the absolute values of the results from the first kernel, and 4) the absolute values of the results from the second kernel.

If all of this seems oddly specific, take comfort knowing that the SURF descriptors are the product of much experimentation [1, p. 411].

2.1.2 Feature matching

Having obtained a bunch of feature locations and descriptors for both the idealized *Take it Easy!* board (Figure 2.4) as well as a photo of a to-be-evaluated board (e.g. Figure 2.3), we are left with the task of feature matching: for each descriptor from the idealized board, the best match from the photo should be determined. As in the SURF paper [1, p. 416], the metric employed is the Euclidean distance. To achieve good performance, the matching is carried out using FLANN, a library for fast approximate nearest neighbor searches by Muja and Lowe [11].

It is likely that a considerable number of the best matches will be false positives; for example because some features may only have been detected in the query image but not in the photo [8, p. 104]. A popular remedy, which is also used by Bay et al. [1, p. 416], is Lowe's ratio test. Lowe suggests "comparing the distance of the closest neighbor to that of the second-closest neighbor" because false matches tend to come with additional close neighbors of similar distance as the closest [8, p. 104]. Typically, matches with a

[5] This type of kernels is known as Haar wavelets.

[6] See Figure 2.7. The kernel sizes are, you guessed it, proportional to the feature's scale.

[7] As employed in the process of assigning orientations to features, a Gaussian centered at the interest point supplies the weights.

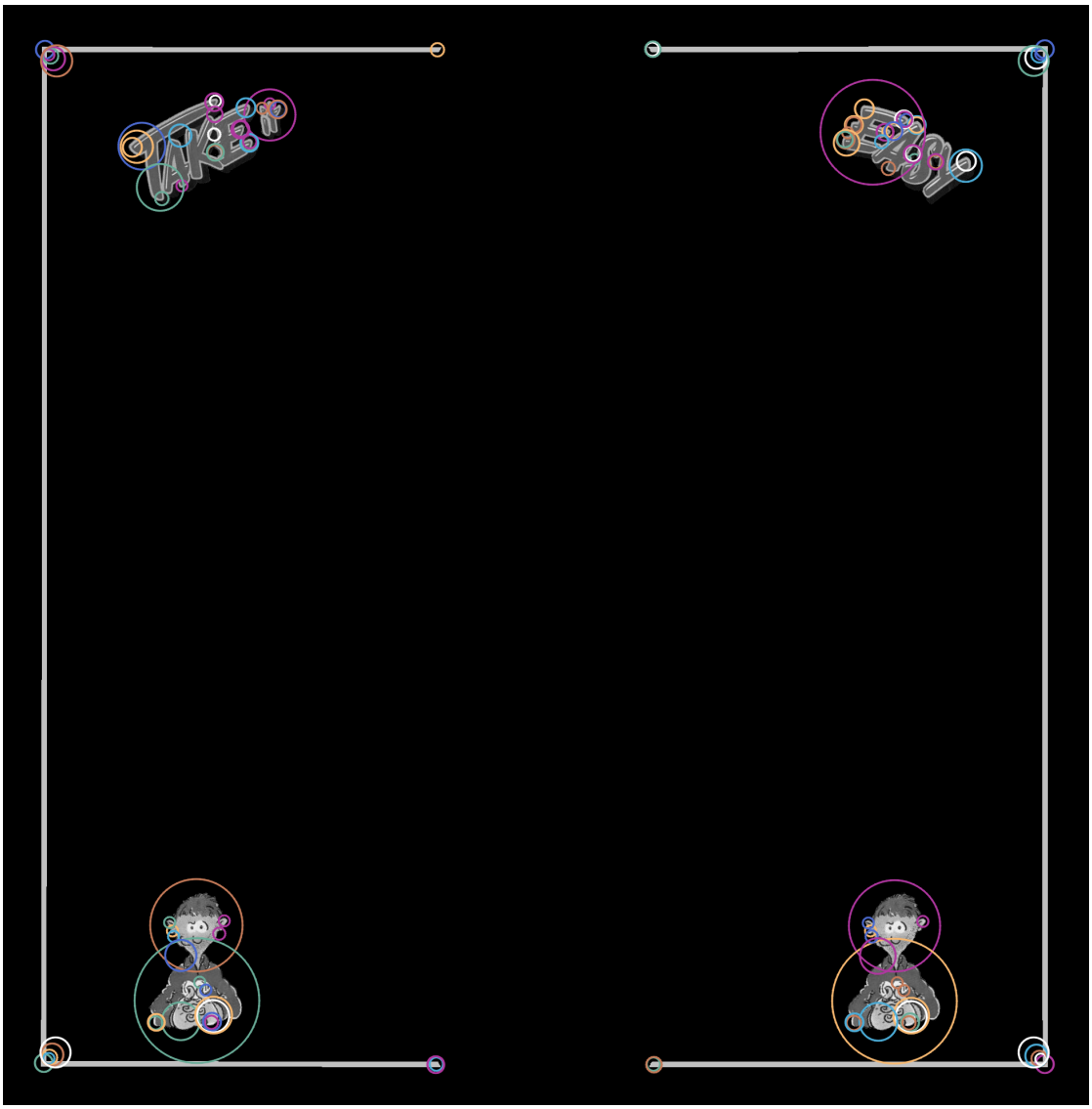


Figure 2.8: Visualization of a small subset of the interest points the SURF implementation from OpenCV [2] finds in the query image for locating *Take it Easy!* boards (Figure 2.4).

second-nearest neighbor within about 130 % of the smallest distance are rejected. Take a look at any of the figures with a *Take it Easy!* board, though.

The same cartoon character appears twice. We can therefore expect two almost equally good matches for any features created from either of these. For this reason, I am using the third-closest neighbor's distance in place of the second. Additionally, the second-closest neighbor is also kept if the distance is only barely greater than that of the closest neighbor. Concretely, matches are filtered with the following code, where `matches` is a list of triples of `DMatch` objects.[8]

```
good_matches = []
for fst, snd, thd in matches:
    if fst.distance < 0.70 * thd.distance:
        good_matches.append(fst)
    if snd.distance < 0.70 * thd.distance and \
       snd.distance < 1.05 * fst.distance:
        good_matches.append(snd)
```

It's not uncommon for this to remove 90 % of the matches. See Figure 2.9.

2.1.3 Homography and perspective correction

The final step toward achieving the goal of this section—warping a photo with a game board into a centered and orthogonal top-down view with nothing but the board—is fitting a homography to the kept feature matches.

A homography or projective transformation can map points projected onto the image from a planar surface in space to points projected from another planar surface. Straight lines are preserved [15, p. 37]; parallelism, angles between lines, and lengths are not. Our first surface is implied by the board in a photo and the second one by the query image shown in Figure 2.4. The board in the photo is warped onto, or aligned with, the query image. Figure 2.10 shows some photos and their transforms. The transformed images resemble photos taken from directly above the center of the board, at an angle that is essentially orthogonal to the board, and with no in-plane rotation—like the query image.

Four pairs of corresponding points, with no three-point subset of either four points being collinear, imply a homography. We would generally hope to have matched many more than four features, but there's no guarantee all, or even the majority, were matched correctly. Those that were matched correctly may not be absolutely accurate. Therefore, we need to estimate a homography. Up to a maximum permitted error, the estimated homography should be in agreement with as many feature matches as possible.

A widely used method for homography estimation [15, p. 318, 4, p. 21] is based on the RANSAC [5] algorithm. The acronym's expanded form, random sample consensus, is quite descriptive of the approach. Four feature matches are selected at random and the implied homography is computed. The other feature matches are then classified as inliers

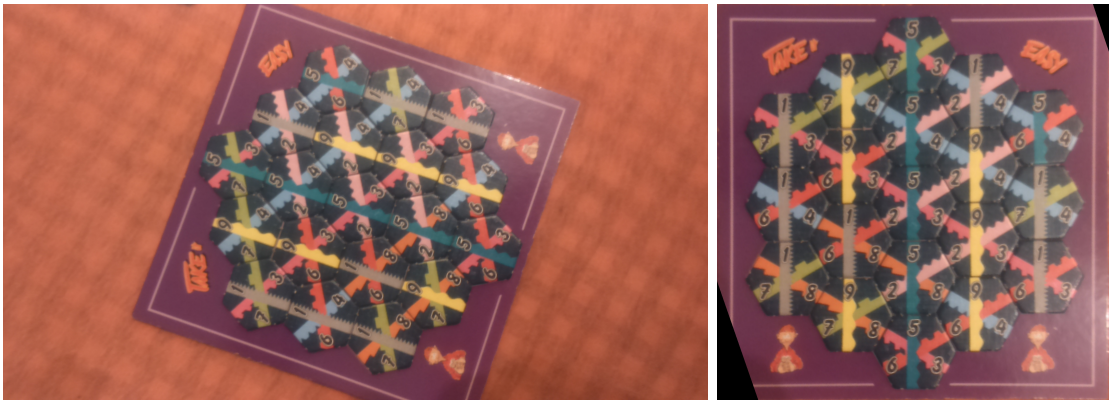
[8] `DMatch` is a class from OpenCV that represents matches of feature descriptors. The important thing is that there is a `distance` member.

(a) Locations of features that are the best match for any of the query image's features.

(b) Locations of features that passed the trial of ratio.

Figure 2.9: Locations of matched features (a) before and (b) after filtering with the variation of Lowe's ratio test explained in the main text. It's super effective. Note that a circle in either image doesn't imply that the feature was matched *correctly*. Still, the vast majority of obviously incorrect matches has been pruned—i.e., those in the middle or outside of the board.

(a) The photo from Figure 2.3.



(b) A somewhat blurry photo with high in-plane rotation.

(c) A photo taken from an angle. Some lens distortion is visible.

Figure 2.10: Some photos of *Take it Easy!* boards before and after correcting the perspective to match that of the query image (Figure 2.4) using the homography estimated from feature matches.

or outliers, depending on whether they concur with the homography well enough.[9] This is repeated. New homographies derived from random samples are evaluated until a specified maximum number of iterations has been exhausted, after which the homography with the most inliers is returned, or the process may be cut short early if a good homography is found quickly.[10]

2.1.4 Remarks

Being quite old and quite successful,[11] *Take it Easy!* has seen a number of releases with several redesigns of the game boards. A smartphone app would ideally support as many of these as possible. I expect that the same approach for locating the board would lend itself to most of them without major changes, but this is a potential difficulty.

Tweaking the parameters of SURF and those for homography estimation along with adapting Lowe's ratio test as detailed in Section 2.1.2 sufficed to locate the boards in all of my 33 test photographs correctly. If necessary, the robustness could be further improved though:

- Additional geometric constraints could be enforced when estimating a homography. We know that a photographed board is still approximately square and that it should take up a majority of the space in the photo, so homographies resulting in tiny or heavily distorted quadrilaterals could be rejected.
- It may be viable to combine another feature detector and descriptor with SURF. Szeliski notes that "[e]dges and lines provide information that is complementary to both keypoint and region-based descriptors" [15, p. 207].

2.2 Identifying numerals

Being able to transform photographed boards into a top-down view with known size, we now can compute the coordinates of tiles with planar trigonometry. At least up to an error caused by sloppy placement of individual tiles, the center of a tile is just a function of its index. The column-major indexing scheme that I'll be using to refer to tiles is illustrated in Figure 2.11.

Three characteristics visually distinguish tiles: numerals, stripe colors, and stripe shapes. I am using numerals as the primary classification method. Each numeral is assigned a region of interest (ROI) purely based on knowing the layout of the tiles and the board. Sufficiently large ROI dimensions were experimentally chosen so that numerals of casually placed tiles still fall into their ROI, as long as the displacement isn't excessive.

Since tiles may not be turned and each type of stripe has a definite direction (e.g., yellow is always vertical), the number of possible digits for any ROI is three. The candidate

[9] The euclidean distance between the destination point of the match and the point obtained by transforming the source point with the homography is compared to a threshold.

[10] The OpenCV implementation adaptively lowers the number of remaining iterations whenever having updated the best-so-far homography with the goal of evaluating just enough four-match samples so that at least one was free from outliers with a given probability.

[11] Burley puts the number of copies sold at above 1 000 000 [3].

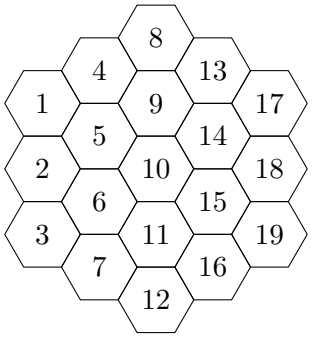
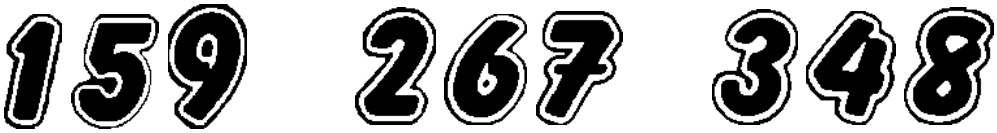


Figure 2.11: Abstract depiction of a *Take it Easy!* board. Each tile is labeled with an index.



(a) top (b) bottom-left (c) bottom-right

Figure 2.12: Idealized numeral templates created from scanned tiles using the GIMP. The arrangement into triples signifies where on tiles the numerals can appear: (a) at the top, (b) at the bottom-left, or (c) at the bottom-right. Note that we never need to distinguish numerals across these groups.

triples for the top, left, and right location are $(1, 5, 9)$, $(2, 6, 7)$, and $(3, 4, 8)$, respectively. Which of the three possible numerals is present within each ROI is determined with *template matching*.

2.2.1 Template matching

Given an image and another, smaller image (called template or patch), template matching compares the small image to every possible location of the first image and quantifies their similarity. The result is a 2D array where each entry signifies how well the template matched the image at that location.[12] The similarity values derive from comparing each pixel of the template to the corresponding pixel of the currently considered image area (which has the same dimensions as the template). Several comparison methods are possible; for example the sum of absolute differences, the sum of squared differences, and cross-correlation.

Figure 2.12 shows the template images I created to locate and identify numerals. Within each of the 57 ROIs (three per tile), template matching is performed three times, since three numerals are possible. The digit of the template that yielded the best match and the associated location are saved. Furthermore, the ratio of the best comparison

[12]Imagine sliding the template across the image pixel by pixel.

result[13] and the best result obtained with one of the two other templates is a natural indicator of how reliably a numeral was identified.

For a color image, matching can be done separately for each channel and the results combined, or the image can be converted to grayscale. I tried both. I also experimented with different comparison methods and with preprocessing board images in various ways. Taking the maximum of all channels at each pixel and then binarizing the resulting grayscale image with adaptive thresholding gave the best results. See Figure 2.13.[14]

The comparison method I'm using is the sum of squared differences (SSD). The best match for one template is therefore the minimum SSD. To render the minima from all three considered templates comparable, each is divided by the number of non-transparent pixels of the template—otherwise smaller templates, such as , would be systematically favored. The template with the smallest of these minimum mean squared differences (MMSDs) confers the digit it represents to the ROI.

2.2.2 Scores

As said before and shown in Figure 2.13, numeral identifications are assigned scores that assess the confidence into their correctness. These scores are ratios of the best similarity value (MMSD) from the runner-up and the winning numeral template. For example, a score of 2.0 means that the winning template's MMSD was half as big as that of the second-best: it matched twice as well. Identifications with a score of 1.3 or higher are considered safe, because I haven't observed any misidentification with a score above that threshold.

2.2.3 ROI refinement and template rotation

In case a numeral was not identified safely (with a score above 1.3), another attempt is made using a refined ROI. The expected location of the numeral is recalculated using the locations of safely identified numerals on the same tile. As this affords us higher confidence in the location's accuracy, the ROI size can be decreased.

When two numerals were available to recalculate the location, the ROI size is decreased more aggressively. Moreover, an angle describing the tile's rotation is calculated. Numeral templates are rotated by this angle during the ensuing reruns of template matching.

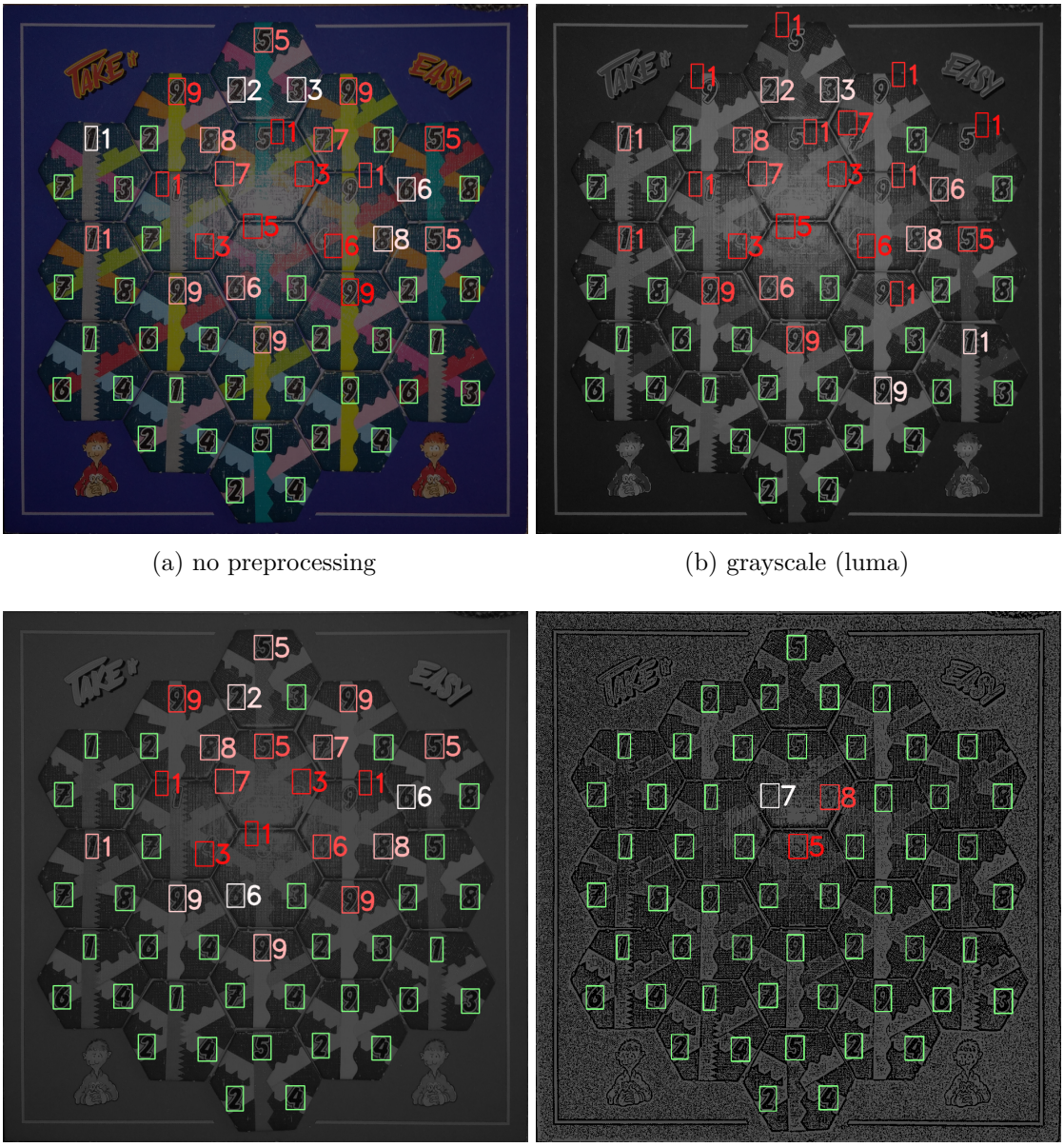
All of this is repeated until an iteration doesn't lift any numeral's score into the $[1.3, \infty)$ range.

2.2.4 Remarks

In most of my photos, all numerals are identified correctly: only 7 out of 1881 (3 numerals per tile, 19 tiles per board, 33 photos) are misidentified. 37 numerals have scores below 1.3. The common theme with these is the presence of strong reflections. See Figure 2.14 and Figure 2.15.

[13] Depending on the comparison method used, this can be a minimum or a maximum.

[14] After this is done, I'm also throwing in an erosion and adding some noise to areas that are very colorful in the original image. Neither of these steps has a particularly large effect, but they help a little.



(a) no preprocessing

(b) grayscale (luma)

(c) pixelwise maximum

(d) pixelwise maximum binarized with adaptive thresholding

Figure 2.13: Evaluation of different kinds of preprocessing for numeral identification based on template matching. The comparison method used is the sum of squared differences for all images. Green rectangles indicate safe identifications (scores above a threshold of 1.3; this is explained in Section 2.2.2) and the predicted digit for these is not shown here. Colors ranging from white to red indicate progressively lower confidence.

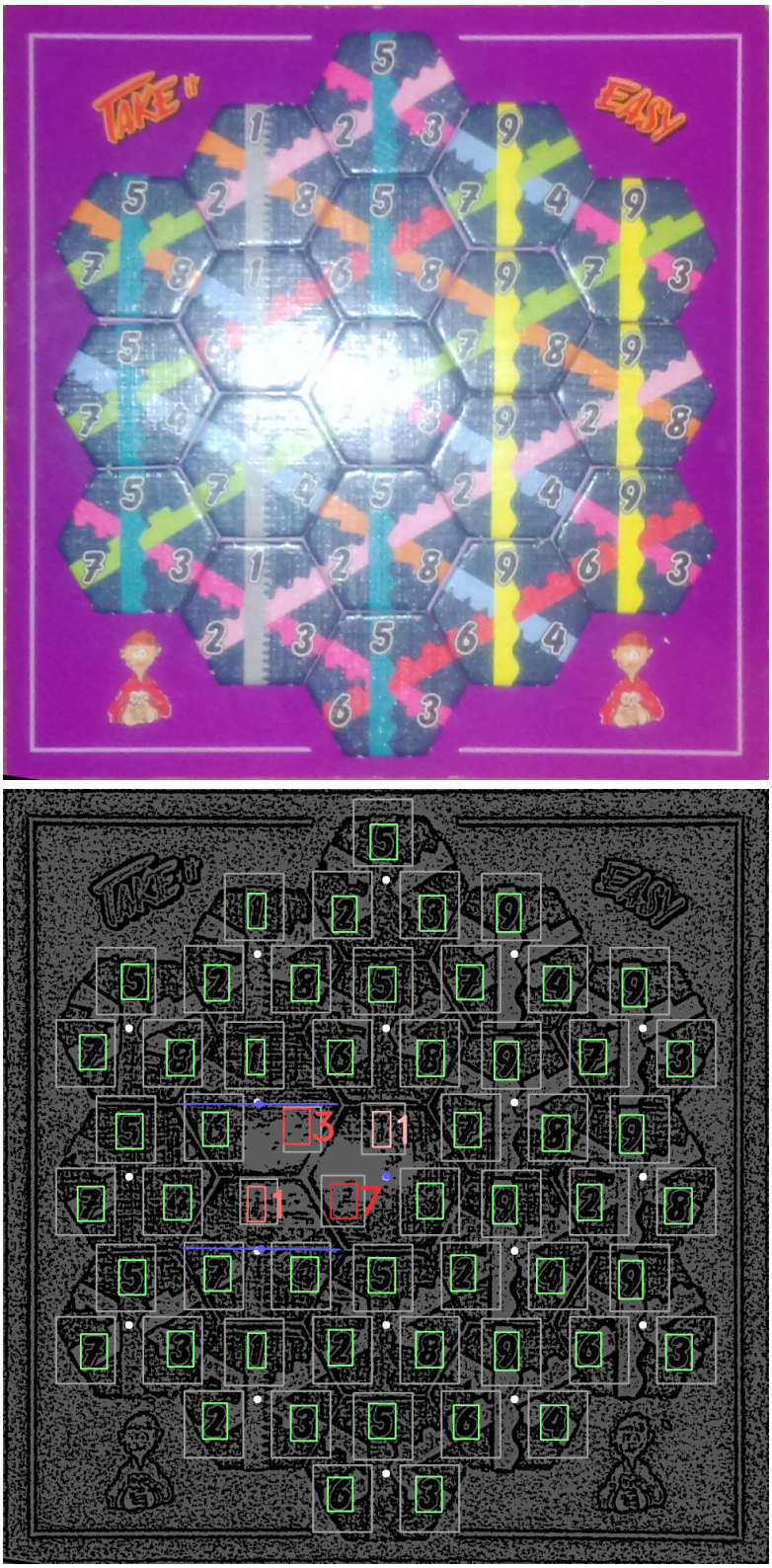


Figure 2.14: Results of the complete numeral identification process for a photo with strong reflections. The predictions for safely identified numerals are intentionally not shown to declutter the image. The gray rectangles are the ROIs used. Note the three different sizes: the base size, the reduced size used for two numerals of tile 10 (the center tile), and the more aggressively reduced size used in tiles for which an angle was determined (5 and 6). Amended tile centers and estimated tile rotations are shown in blue. Only the numeral at the bottom-right of tile 5 was misidentified.

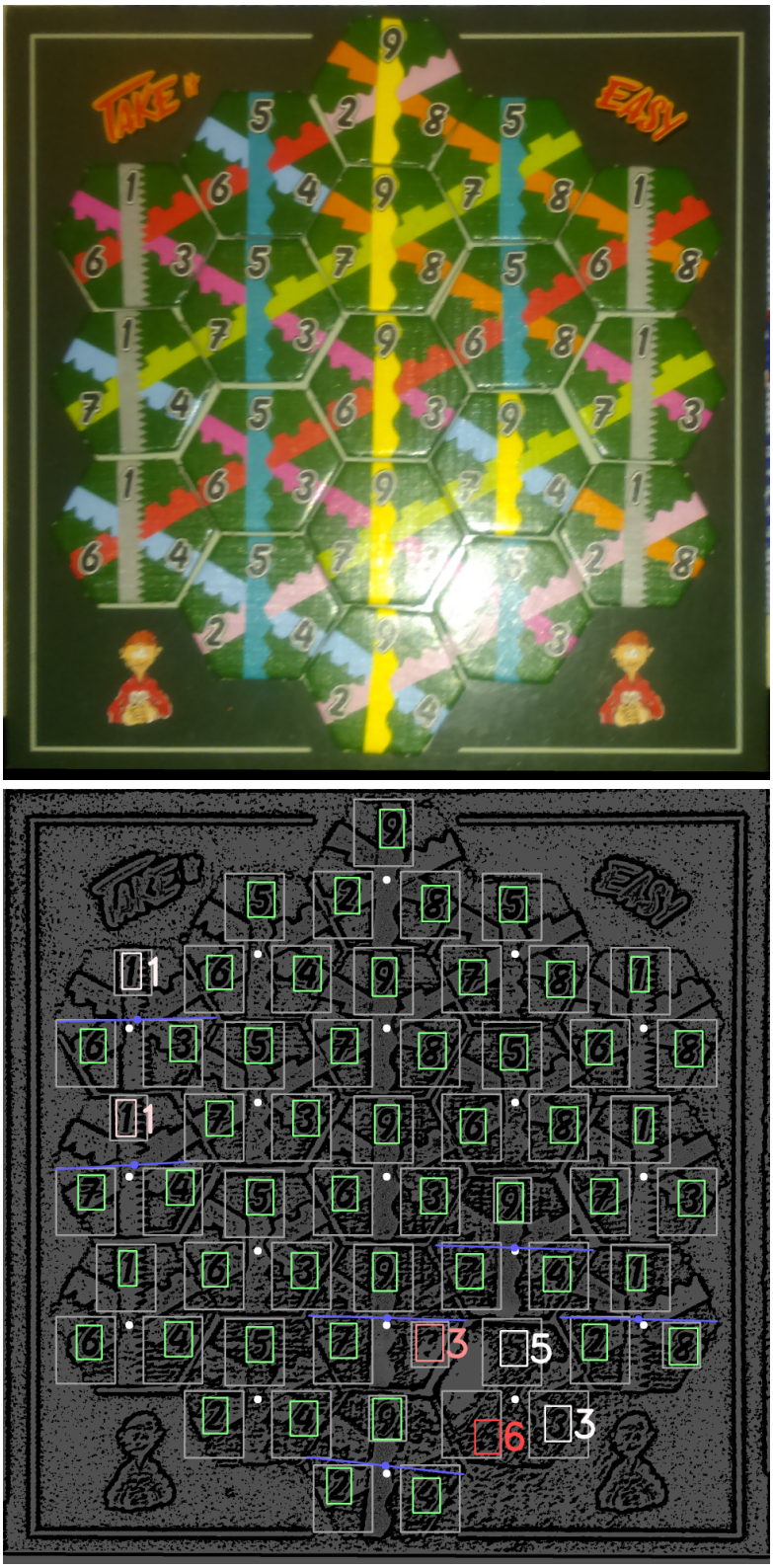


Figure 2.15: Results of the complete numeral identification process for another photo with strong reflections. Tile 12 (the tile closest to the bottom edge) is rotated to the right quite a bit. Also note that no ROI of tile 16 was amended because not a single numeral is safely identified. Making sure that the three detected numeral locations of any tile always at least are self-consistent is a possible improvement: the relative locations on tile 16 don't make sense now.

2.3 Classifying colors

Not much can be done about the numeral misidentifications in Figure 2.14 and Figure 2.15, since the numerals essentially are completely obscured by reflections. Unfortunately, these reflections are characteristic for a range of photos. While, at their most intense, caused by the phone's flash or flashlight—in which case the solution would just be to turn these off—other light sources can have similar effects.[15]

Because the reflections are localized, an area that is as far as possible from a numeral that couldn't be conclusively identified[16] has the highest chance of being unaffected by them. I'm therefore sampling stripe colors opposite of such numerals. Note that color classification is purely a fallback strategy used for stripes whose numeral wasn't safely identified. It is completely skipped for photos that didn't cause problems for numeral identification.

2.3.1 Ground truth

Because lighting conditions and cheap phone cameras affect color reproduction considerably, I have not hard-coded the nine stripe colors in any way. Instead, areas that can safely be assumed to make up part of a stripe of a specific color are sampled. Only tiles with at least two safely identified numerals are used for this. Otherwise, the tile's rotation is unknown and defining a region for sampling a color is problematic: background pixels may be included. Figure 2.16 shows two examples of the regions used and the resulting training data.

Pixels that are both below a saturation and above a value threshold (in the HSV model) are removed from the training data; they largely stem from strong reflections and won't be useful. Figure 2.16 also shows the training data after this cleanup is performed.

2.3.2 Classifying pixels with *k*-NN

For the classification of colors, the k-nearest neighbors (k-NN) algorithm is used; specifically, the implementation from scikit-learn [14]. Pixels from the to-be-classified stripe are sampled in the same way as for the generation of training data before, although slightly bigger regions are used now.[17] Each pixel is then classified individually with k-NN.

Perhaps unsurprisingly, k-nearest neighbors classifies a test sample (an RGB triple in this case) by finding the specified number (k) of nearest neighbors in the training data. One way a prediction can be obtained is by a simple majority vote: the predicted class is the class that contributed the most neighbors. I am using probabilistic classification instead: a triple of probabilities adding up to 100 % that signifies how many of the k neighbors are from the training set for each of the three considered colors is computed rather than a single class label. A value of 200 for k has worked well.

[15] An app could probably make sure photos are taken without flash.

[16] When I write "safely" or "conclusively" identified, I'm referring to the score threshold of 1.3 explained in Section 2.2.2.

[17] Including some background pixels is less of an issue here. The important thing is that we get at least a few good pixels for which the stripe color isn't hidden by reflections.

Figure 2.16: Two boards with the regions used to extract training data for color classification highlighted (left), and the extracted pixels arranged in grids (right). The grids are made up of the highlighted regions' pixels. Each grid row shows the colors for one stripe direction. From top to bottom and left to right, the colors' corresponding digits are $(1, 5, 9)$, $(2, 6, 7)$, and $(3, 4, 8)$. In the lower grid for each board, pixels that most certainly resulted from reflections have been removed. Note that the colors aren't actually sampled uniformly: pixels are repeated in 8 out of 9 squares of the grids. Also note that the colors from the second board don't match those from the first particularly well.

To correct for non-uniform sampling of training data, each probability is weighted by the inverse of the number of training samples taken for that color. For example, more yellow than gray stripes were sampled for both boards shown in Figure 2.16, and therefore, if a pixel had the same number of yellow and gray neighbors, a higher probability should be assigned to gray.

The metric I'm using to determine the nearest neighbors is the Chebyshev distance (also known as maximum or L_∞ metric), i.e., the maximum absolute difference across all three channels. The idea is that, for example, $(96, 96, 96)$ probably shouldn't be considered further from $(32, 32, 32)$ than $(32, 96, 32)$ is. This rules out a lot of metrics like the Manhattan or Euclidean distance.[18]

2.3.3 Classifying stripes

Stripes are classified by combining the color predictions made for their pixels. A key aspect is that pixels that didn't have k neighbors within a distance of 30 in the training data are ignored. This way, the values of most pixels from the sampled region of a to-be-classified stripe can be dominated by reflections, and a good prediction can still be made based on the few that aren't. The predicted color for a stripe is the maximum of the three sums obtained by element-wise accumulation of the probability triples from pixels that did have k close-enough neighbors. The triple obtained by dividing those sums by the total number of classified pixels is used to quantify the prediction's certainty. Here are some examples of how to interpret it:

- $(1, 0, 0)$: All pixels were classified as the first color. The probability estimates were 100 % without exception.
- $(0.2, 0, 0)$: 20 % of the pixels were classified as the first color with a probability of 100 % each. The remaining pixels were ignored.
- $(0.3, 0.4, 0)$: The first color got votes equivalent to 30 % of the pixels predicting it with probability estimates of 100 %. Likewise, the second color got votes equivalent to 40 %. All pixels assigned a probability of zero to the third color and 30 % of the pixels were ignored.

The entries always add up to values in the range $[0, 1]$.

2.3.4 Remarks

It generally isn't obvious whether a stripe's predicted color or the result of numeral identification should be prioritized when they disagree. I am using the former in almost all cases: the color-based prediction is only rejected when the maximum of the probability triple explained above is smaller than 0.02. This is crude and could surely be improved, but more test photos are needed because no stripe was misclassified in my 33 photos.[19]

[18] A possible improvement over the Chebyshev distance of RGB values may be using HSV or a similar color space and a weighted metric.

[19] Two times, no classification was made at all though, because not a single pixel met the requirement of having k close-enough neighbors. No errors resulted from this in the end as both numeral identifications were already correct.

Many other aspects of the outlined color classification scheme could also be improved further:

- At least for collecting training data, pixels can be sampled on both sides of diagonal stripes (i.e., also on the side with the numeral). For the classification itself, the chance that no useful information can be extracted on the numeral side is high: numeral identification doesn't fail for no reason.
- Using a weighted metric in HSV or a similar color space could probably assess the similarity of colors better than the Chebyshev distance of RGB values. This is complicated by the fact that the hue values of gray are all over the place, though, so measuring hue differences without taking saturation into account isn't viable.
- When classifying pixels, neighbors could be weighted based on distance. When aggregating the pixel classifications into a prediction for the stripe, pixels could be weighted based on the mean distance of the k neighbors.
- Grouping the training data for each color, for example with k-means clustering, may drastically decrease the number of neighbors needed for k-NN classification without compromising the accuracy. This should make things significantly faster. The overall processing speed of my Python code is discussed in Section 2.4.

As said before, no stripe was misclassified in my test photos. Without taking more, most of these changes would therefore just be stabs in the dark.

2.4 Discussion

2.4.1 Performance

In my test photos, 37 out of 1881 numerals were identified with scores below 1.3, and 7 of these numerals were misidentified. Color classification is performed for the stripes corresponding to the 37 numerals that weren't safely identified, and for all but 2, the correct color was predicted. For the remaining 2, no prediction was made at all and the numerals were already identified correctly. Ultimately, the recognition is 100 % accurate for my test photos. See Figure 2.17 for some examples.

One avenue to further increased robustness that hasn't been discussed so far isn't about computer vision, but a property of the game itself. The same tile can't appear more than once, because there is exactly one of each possible tile. Recalling that 19 out of 27 tiles are placed on the board, we can put the chance an isolated recognition error (for example) has to result in one of the 18 tiles that were already predicted elsewhere being repeated at

$$\frac{18}{27} \approx 0.67$$

An improved guess for the lowest-confidence location that transforms the predicted configuration for the whole board into one that is self-consistent could be made in that case.

2.4.2 Processing speed

The time needed to score one photo varies depending on whether, and to what extend, color classification has to be employed. It's skipped for about half of my photos. The *wall time* for scoring one of those with my ThinkPad X121e from 2011 is typically close to 10 seconds.[20,21] This time is quite evenly divided between three chunks of work:

1. Starting the CPython interpreter and importing modules. This already takes more than 2 seconds on my laptop. The obvious remedy is to use a compiled language.
2. Locating the board by detecting, describing, and matching features, and estimating a homography. This takes 3 to 7 seconds. Most of that time is taken up by feature detection and especially description. I doubt that much speedup is attainable here. The OpenCV implementation of SURF manages to utilize both of my CPU cores nearly 100 % of the time and I already implemented two basic optimizations: photos are downscaled considerably before detecting features and the interest points and descriptors of the board template are cached to disk.
3. Preprocessing the image for template matching and template matching itself. This takes 3 to 4 seconds. The warped photo and the numeral templates could probably be smaller to speed it up.[22] Another feasible optimization is template-matching in multiple ROIs in parallel.

I need about half a minute to score one board myself with the help of a calculator app and when trying to be quick.[23] Sometimes I make mistakes. 10 seconds wall time thus is a good starting point for automated scoring.

Color classification, however, is slow. It can take more than a second for one stripe.[24] Usually, this isn't too much of a problem because only 4 or 5 numerals are identified with poor confidence. Sometimes scoring a photo takes half a minute, though. Consolidating pixels from the training data and reducing the number of neighbors used by k-NN can most likely speed this up drastically. No work is done in parallel either, which would naturally be possible by classifying multiple stripes at the same time.

Supplementary to most already mentioned points, GPU computing may also be worth looking into. Since porting the Python code to a mobile OS would most probably entail a complete rewrite in a different language,[25] I didn't spend much time optimizing it.

[20] The CPU (AMD E-450) has two cores with 1650 MHz. Phones can probably achieve similar performance by now. The mean CPU usage is around 135 %. Some potential for parallelization is untapped.

[21] The main factor causing wall times to vary is how many features SURF detects and subsequently computes descriptors for.

[22] Photos are always warped to the size of the board template now, which is slightly below 2000×2000 pixels.

[23] Yeah, $n = 1$. Maybe try it yourself with Figure 2.3 on page 18.

[24] All measurements were done on my ThinkPad X121e.

[25] An alternative is sending photos to a remote server for processing. No reimplementation would be necessary this way.

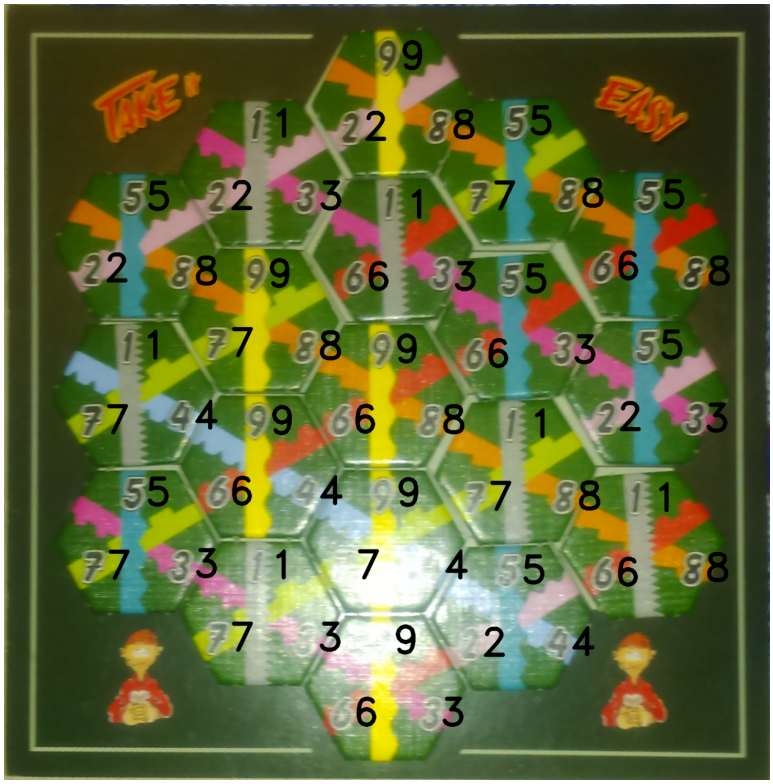


Figure 2.17: Two *Take it Easy!* boards with the final predictions made by the recognition program superimposed.

Figure 2.17 cont.: Two more *Take it Easy!* boards with the final predictions made by the recognition program superimposed.

Chapter 3

Kingdom Builder

So, what are you? Farmers.
Fishermen. Web designers.

The 12th Doctor,
Doctor Who

Kingdom Builder is a strategy game by Donald X. Vaccarino for two to four players and the 2012 winner of the German *Spiel des Jahres*[1] award [6]. Figure 3.1 shows a photo taken after completing a game. Players take turns placing settlement tokens on terrain hexes until one player runs out of their 40 settlements. I won't go into the placement rules here, but only focus on the scoring.

During the game's setup, three out of ten *Kingdom Builder cards* are randomly drawn and revealed. These cards specify scoring rules and are evaluated for each player separately after the game ends. I have reproduced the text of all *Kingdom Builder cards* verbatim in Table 3.1 to give an idea of how long scoring takes.

3.1 Locating the board sections

If you look closely at Figure 3.1, you'll notice that there isn't one contiguous game board, but 4 separate sections. In total, *Kingdom Builder* includes 8 such board sections, and setting up a game involves selecting a random subset of 4 from them.[2] This complicates locating the board compared to *Take it Easy!*, but not by much. The approach taken here is basically the same as in Section 2.1. I scanned all 8 board sections and then cut them out and downscaled the images with the GIMP (see Figure 3.2). Now we can:

1. Compute SURF descriptors for each scan.
2. Compute SURF descriptors for the input photograph.
3. Match the descriptors.
4. Filter the matches with Lowe's ratio test.[3]
5. Finally, estimate 8 homographies.

[1] Game of the Year

[2] $1680 = 8 \cdot 7 \cdot 6 \cdot 5$ different game boards are consequently possible (order matters). Another factor of $2^3 = 8$ results when some sections are rotated by 180°. The instructions are unclear about whether this should be done.

[3] Unlike for *Take it Easy!*, no modifications are necessary now.

Figure 3.1: Photograph of a *Kingdom Builder* board after the game ended. Four people played, so all colors are used: orange, blue, black, and beige. Only the mountain hexes and the hexes with ornate frames can never have settlements.

Table 3.1: The text on each of the ten *Kingdom Builder cards* reproduced verbatim. *Location* and *castle* hexes are the hexes with ornate gold and silver frames, respectively (see Figure 3.1). Three of these cards are randomly drawn before the game starts and visibly placed next to the board. The scoring functions (right column) of those are evaluated for all players after the game ends and 3 additional points (i.e., gold) are always awarded for every settlement adjacent to a castle hex. Clearly, a lot of counting is involved. The player with the most points wins.

Name	**Objective**	**Description**
Farmers	Build settlements in all sectors	3 gold for each of your own settlements in that sector with the fewest of your own settlements.
Fishermen	Build settlements on the waterfront	1 gold for each of your own settlements built adjacent to one or more water hexes.
Workers	Build settlements next to location or castle hexes	1 gold for each of your own settlements built adjacent to a location or castle hex.
Citizens	Create a large settlement area	1 gold for every 2 of your settlements in your largest own settlement area.
Lords	Build the most settlements in each sector	Each sector: 12 gold for the maximum number of settlements there; 6 gold for the next highest number of settlements.
Hermits	Create many settlement areas	1 gold for each of your own separate settlement and for each separate settlement area.
Knights	Build many settlements on one horizontal line	2 gold for each of your own settlements built on that horizontal line with the most of your own settlements.
Miners	Build settlements next to a mountain	1 gold for each of your own settlements built adjacent to one or more mountain hexes.
Merchants	Connect location and castle hexes	4 gold for each location and/or castle hex linked contiguously by your own settlements to other location and/or castle hexes.
Discoverers	Build settlements on many horizontal lines	1 gold for each horizontal line on which you have built at least one of your own settlements.

Figure 3.2: Two of my eight scans of *Kingdom Builder* game board sections. They all have a size of 1920×1629 pixels. The hexes with ornate gold and silver frames are called *location* and *castle* hexes, respectively.

Figure 3.3: Two of the board sections from Figure 3.1 aligned with the scans shown in Figure 3.2 using the homographies estimated from feature matches.

To select the homographies from the 4 board sections that are actually in the photo, the 8 absolute numbers of feature matches that were inliers are consulted. The 4 board sections whose homographies yielded the most inliers are assumed to be the ones that are in the photo. Homography estimation with SURF features is robust enough not to be thrown off by the presence of settlement tokens in the photographs. Additional geometric constraints could be enforced on top of that if necessary.[4]

3.2 Settlement token recognition

With the exception of mountain, location, and castle hexes, one settlement token may be on every hex. Water is usually also exempt from settlement placement, but not in general. I experimented with several approaches to settlement detection. These include:

- Detecting blobs in structural similarity (SSIM) [17] images computed from a board section aligned with the corresponding scan and the scan itself.
- Detecting blobs in absolute difference images of a board section aligned with the corresponding scan and the scan itself.
- Computing SURF (and also SIFT) descriptors for a board section image and the corresponding scan at identical locations, and identifying hexes in which descriptor pairs match unusually poorly.
- Computing a grayscale image for each possible settlement color in which each pixel's value denotes its likelihood of belonging to a settlement of that color, and detecting blobs. The grayscale images are based on expected hue, saturation, and value distributions for that color.

No single detection method yielded particularly good results. I therefore tuned the parameters of the last three to err on the side of false positives—but rarely miss settlements—and combined them: only hexes in which all three detectors predicted a settlement are classified as containing one.

The next three subsections explain the individual detectors in greater detail. All of them operate on one board section at a time and return a 10×10 matrix in which non-zero entries signify that a settlement was detected in that hex.

3.2.1 Blob detection in absolute difference images

The warped board section and the corresponding scan are normalized, blurred, and the pixelwise absolute difference is computed for each channel. The blurring is done for two reasons:

- Differing areas that are much smaller than settlement tokens are irrelevant.
- The warped board section and the scan aren't perfectly aligned everywhere; for example because a homography can't correct lens distortion.

[4] All board sections should have roughly the same size and mostly rectangular shape in the photo and none of them may overlap.

Figure 3.4 shows two examples of the resulting difference images with circles drawn around all detected blobs in the color of the channel in which the blob was detected. Not only bright but also dark blobs are considered: sometimes a settlement actually matches the scan *better* than the surrounding terrain does. Additionally, hex edges are slightly darkened with an erosion to disconnect any touching settlements and ensure these are registered as separate blobs.

The blob extraction itself is carried out by OpenCV's `SimpleBlobDetector`, which applies successively higher thresholds and extracts contours from the resulting binary images. These contours are filtered based on their size (number of pixels), and various attributes of the shape, such as circularity and convexity. Hexes with at least one blob are predicted to contain a settlement.

3.2.2 SURF descriptor distances

The second detector also compares the given warped board section to its corresponding scan. However, instead of the pixelwise absolute difference, SURF descriptors are used. The Euclidean distances of descriptors computed at identical locations of the two aligned images are compared to a threshold. Any hex with a pair of descriptors that don't match each other well enough for their distance to stay below the threshold is considered to have a settlement.

See Figure 3.5 for an example of where settlements are detected. An advantage of this detector is that SURF descriptors are unaffected by locally uniform brightness offsets (illumination bias) and also invariant to contrast changes [1, p. 410].

3.2.3 Blob detection in probability images

The warped board sections extracted from a photo are used directly by the third settlement detector: no comparisons to the scans are made. For each of the four settlement colors—orange, blue, black, and beige—the board section is transformed into a grayscale image where the value of every pixel denotes the degree of belief that the pixel could be part of a settlement of the considered color. OpenCV's `SimpleBlobDetector` is used to extract blobs again.

The transformations to grayscale images are based on expected hue, saturation, and value ranges for the considered settlement color. Probability density functions of normal distributions are evaluated for the three numbers of each pixel and a grayscale image is constructed from the products. As a concrete example, the transformation for detecting orange settlements is

$$\textbf{orangeness}(h, s, v) = f(h \mid 15, 5) \cdot f(s \mid 255, 75) \cdot f(v \mid 255, 100)$$

where $f(x \mid \mu, \sigma)$ is the probability density of the normal distribution with expected value μ and standard deviation σ, and h, s, and v are the hue, saturation, and value of a pixel. The domain is $[0, 180] \times [0, 255] \times [0, 255] \subset \mathbb{N}^3$: the maximum hue is 180, so 15 is exactly between pure red (0) and pure yellow (30). See Figure 3.6 for an example of the normalized results of applying this function to all pixels of a board section image with some orange settlements.

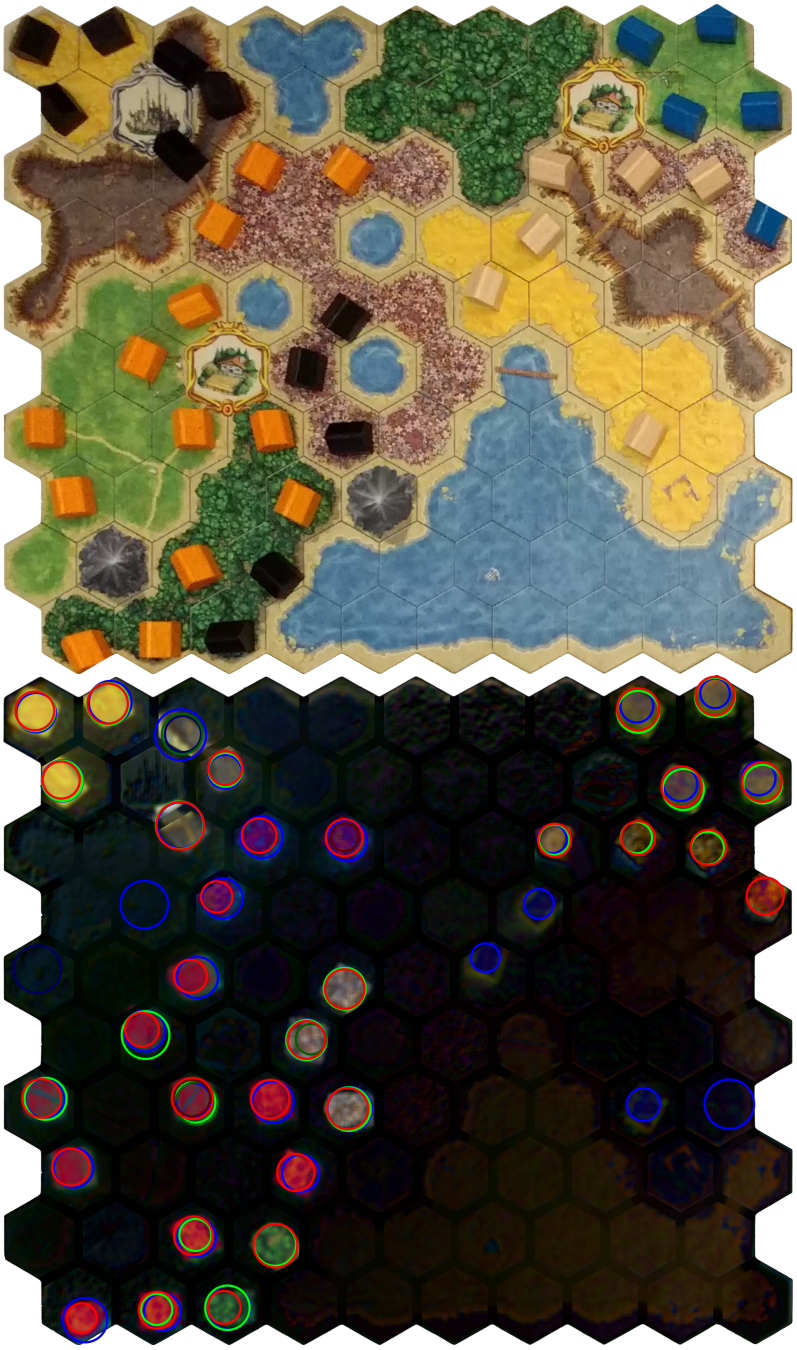


Figure 3.4: Blobs detected with OpenCV's `SimpleBlobDetector` in an absolute difference image of a warped board section and the corresponding scan. The top image shows the board section again for comparison. Blob detection is separately performed for all channels (RGB), as only one is meaningfully different in some cases. Also note that two touching black settlements near the top-left corner have been successfully separated.

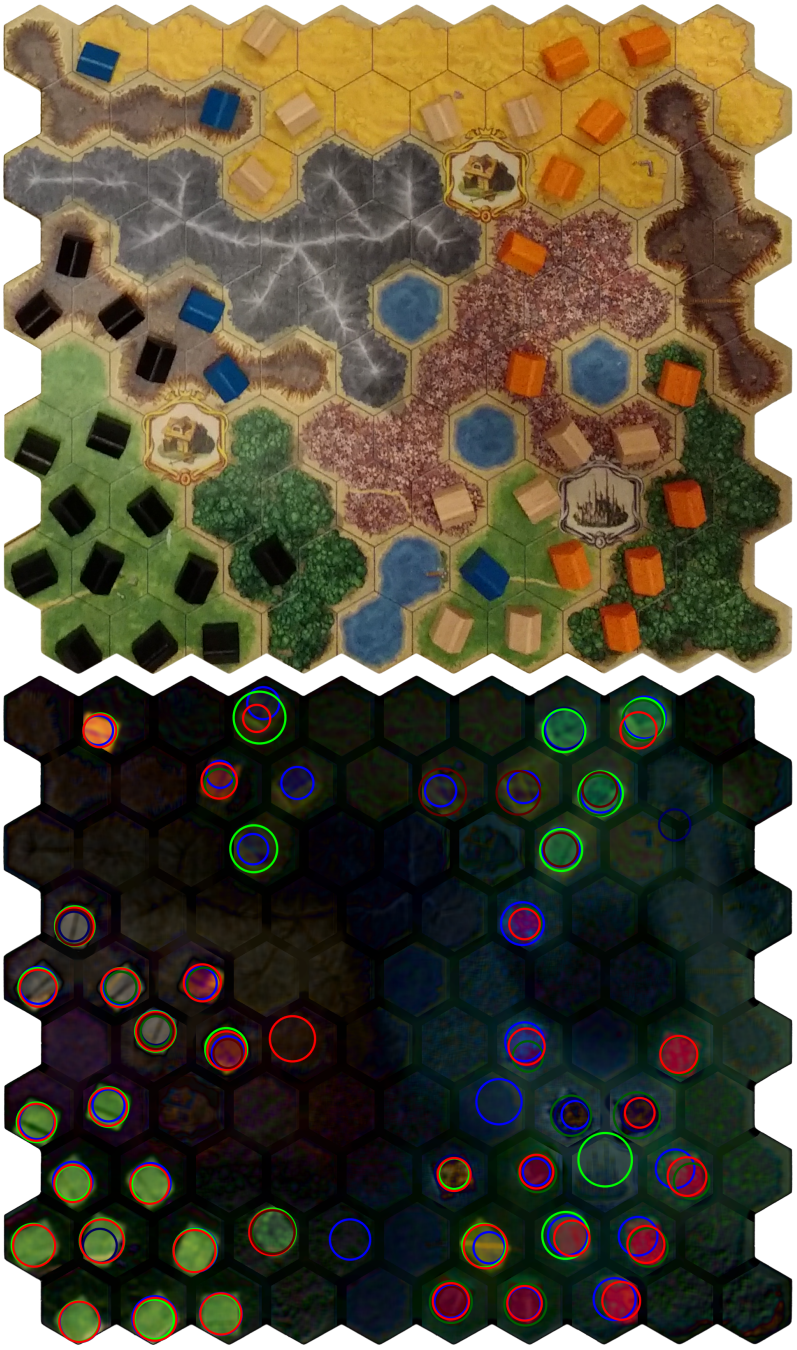


Figure 3.4 cont.: Another demonstration of blob detection in an absolute difference image computed from a warped board section (top) and the corresponding scan (not shown). Beige settlements are particularly hard to detect.

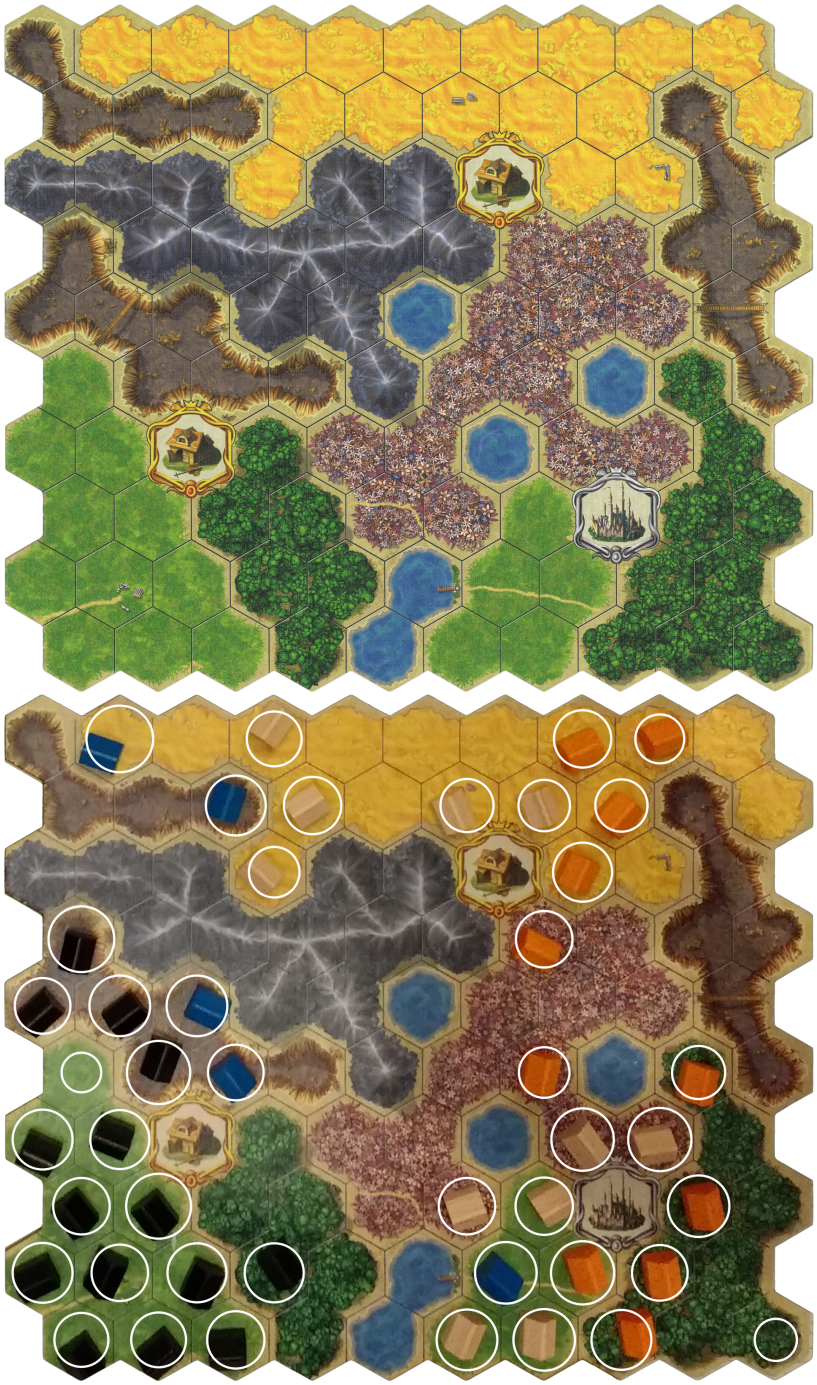


Figure 3.5: A scan of a board section (top) and an aligned board section from a photo (bottom) with hexes, in which a pair of SURF descriptors computed at identical locations in both images was found to significantly differ, labeled. The circle radii are proportional to the largest descriptor distance that occurred within a hex.

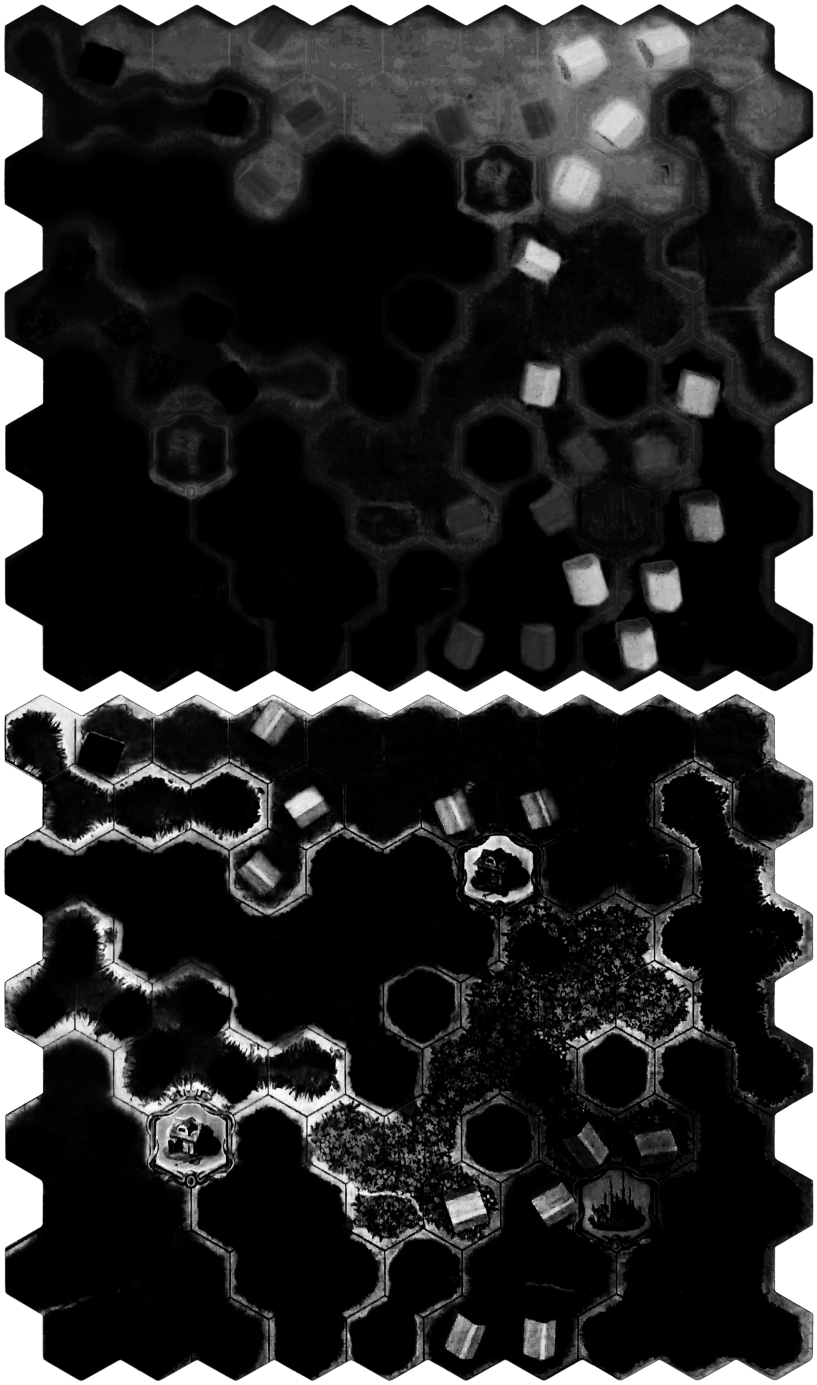


Figure 3.6: Two of the four transforms of the board section shown at the bottom of Figure 3.5 that are used for direct settlement detection. In the top image, pixels that match the expected hue, saturation, and value of orange settlements well are bright. In the bottom image, the same is true for beige settlements. Detecting beige settlements is complicated by beige hex edges.

Direct settlement detection based on color is the most inaccurate method of all three; especially because the number of false positives for beige tokens is high. However, the primary function is predicting the color of settlement tokens when the other two detectors already marked a hex as containing one.

Whenever blobs of different colors are detected in one hex, orange is prioritized over blue, blue over black, and black over beige. The motivation for this primitive resolution method is that the tendency toward false positives increases in the same order: orange blobs are rarely detected where no orange settlement is, while beige blobs frequently are.

3.3 Discussion

The settlement predictions obtained by combining my three detectors for the photo from Figure 3.1 are shown in Figure 3.7. These results are 100 % correct, but this is often not the case. The predictions for two more photos are shown in Figure 3.8. Despite the advantage of making comparisons to empty reference board sections, human-level performance is a long way off.

It is possible that my recognition scheme could be made more robust by adjusting some parameters. I have already experimented for a while, though, and the number of knobs that can be tweaked is quite large—at least when trying to exhaustively explore this space by hand. I expect that an approach based on machine learning techniques such as artificial neural networks is more suitable for recognizing *Kingdom Builder* settlement tokens.[5] To quote a blog post by Bartosz Milewski [10] wildly out of context:

> Image recognition is one of these areas where the analytic approach fails miserably.

Granted, Milewski was referring to much more difficult recognition problems; this one may not be out of reach for handcrafted solutions. Tangentially, things would be simplified a little had the designer or publisher of *Kingdom Builder* not chosen beige as one of the settlement colors. More generally, considering automated scoring during the design of board games has the potential to make it much easier.

Given a reliable settlement recognition method, an app would either have to let the user select which *Kingdom Builder cards* were drawn, or, if they are in the photo, recognize them to actually perform scoring. Both approaches are viable: the former wouldn't take the user much time and the latter could be done with feature matching because the cards contain distinctive artwork.

I tested my code for *Kingdom Builder* with two more photos that aren't shown here,[6] and for which only a single misprediction each resulted. However, the lighting conditions under which the two photos were taken also resemble those leading to the error-free predictions shown in Figure 3.7. In summary, detecting *Kingdom Builder* board sections is unproblematic, but more work—and perhaps a different approach—is needed for sufficiently robust settlement recognition.

[5] A classifier that takes an image of a single hex as input may work well.

[6] They can be found in the Git repository.

Figure 3.7: Predicted settlement locations and colors for the photo shown in Figure 3.1. No mistakes were made for this photo.

Figure 3.8: Predicted settlement locations and colors for a photo where most light comes from the side. Many settlements were missed, incorrect colors are predicted frequently, and settlements are predicted on some hexes where none are.

Figure 3.8 cont.: Predicted settlement locations and colors for a photo of the same board, but with an additional light source. The predictions are different but not better.

Chapter 4

Conclusion

Consequences! My old nemesis.

Master Po Ping

We have seen methods for recognition of the elements required to perform end-of-game scoring for two board games—*Take it Easy!* (Chapter 2) and *Kingdom Builder* (Chapter 3). For both games, traditional computer vision techniques were employed. The performance of the *Take it Easy!* recognition is good: no mistakes were made across 33 photographs. *Kingdom Builder*, on the other hand, may call for machine learning approaches such as artificial neural networks that recently led to many advances in computer vision: the strategy outlined in Chapter 3 is not accurate enough to be of practical use.

My main motivation was evaluating the feasibility of using smartphones for automated scoring. An app implementing the recognition scheme for *Take it Easy!* from Chapter 2 is viable and some people would likely find it preferable to mental arithmetic or use of a calculator; both in terms of speed and accuracy. More work is needed to reach a conclusion for *Kingdom Builder*.

Among other methods, color classifiers were used for both *Take it Easy!* and *Kingdom Builder*. Despite varying viewing conditions and strong reflections, the former classifier was highly accurate. The latter was not. Why is that? A key difference is that we were able to gather prior knowledge about the colors' actual appearance in each particular *Take it Easy!* photo. Numeral identification provided a set of regions with known color that were sampled before classifying unknown regions. No such thing was possible for *Kingdom Builder*, because there is no redundant source of information that complements settlement colors.

In general, modern board games are diverse and recognition methods that proved useful for one can't be expected to transfer to another. An exception may be detecting the game board; the standard method of feature-based alignment worked well (with minor adaptations) for both *Take it Easy!* and *Kingdom Builder*. It isn't hard to imagine that this could also be the case for more games.

Board game creators and publishers could facilitate the development of robust and fast scoring programs for their games by already considering automated scoring during the design process. The use of highly contrasting colors or distinctive markers for elements that must be recognized are two ways to do this. Furthermore, the option of providing a scoring app could afford game designers greater freedom with respect to possible game

mechanics. Design ideas that would otherwise make scoring too complicated or tedious can be reconsidered if a scoring app is to be bundled with the game.

In the introduction, I hypothesized that current computer vision techniques enable smartphone apps that automate end-of-game scoring based on a photograph for many board games. This was confirmed for *Take it Easy!*. Error-free recognition was achieved across 33 test photographs. The results for *Kingdom Builder* are inconclusive, but several promising avenues toward improved recognition remain unexplored.

Glossary

Terms

hex A hexagonal space on a game board. 41–44, 46, 47, 50–52, 54

Acronyms

FLANN fast library for approximate nearest neighbors 22

GIMP GNU Image Manipulation Program 18, 28, 41
GNU GNU's Not Unix 59

RANSAC random sample consensus 24

SIFT scale-invariant feature transform 15, 19, 46
SURF speeded up robust features 15, 19, 21–23, 27, 37, 41, 46, 47, 50

Abbreviations

DoH determinant of the Hessian matrix 19–21

HSV hue, saturation, value 33, 35, 36

***k*-NN** *k*-nearest neighbors 13, 33, 36, 37

MMSD minimum mean squared difference 29

OS operating system 37

RGB red, green, blue 33, 35, 36, 48
ROI region of interest 27–29, 31, 32, 37

SSD sum of squared differences 28–30
SSIM structural similarity 46

Bibliography

[1] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006: 9th European Conference on Computer Vision. Proceedings, Part I*. Springer, 2006, pp. 404–417. ISBN: 978-3-540-33833-8. DOI: 10.1007/11744023_32.

[2] Gary Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[3] Peter Burley. *Take it Easy! A Puzzle! – A Maze! – A Game!* Burley Games. 2016. URL: http://burleygames.com/board-games/take-it-easy/ (visited on 2018-03-13).

[4] Elan Dubrofsky. "Homography Estimation". MA thesis. University of British Columbia, 2009-03. URL: https://www.cs.ubc.ca/grads/resources/thesis/May09/Dubrofsky_Elan.pdf (visited on 2018-03-12).

[5] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Communications of the ACM* 24.6 (1981-06), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692.

[6] *Kingdom Builder*. German. Spiel des Jahres e. V. 2012. URL: http://spiel-des-jahres.com/kingdom-builder (visited on 2018-04-05).

[7] Tony Lindeberg. "Feature Detection with Automatic Scale Selection". In: *International Journal of Computer Vision* 30.2 (1998), pp. 79–116. ISSN: 1573-1405. DOI: 10.1023/A:1008045108935.

[8] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2 (2004-11-04), pp. 91–110. ISSN: 1573-1405. DOI: 10.1023/B:VISI.0000029664.99615.94.

[9] David G. Lowe. "Object Recognition from Local Scale-Invariant Features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, pp. 1150–1157. DOI: 10.1109/ICCV.1999.790410.

[10] Bartosz Milewski. *The Earth is Flat*. 2018-01-11. URL: https://bartoszmilewski.com/2018/01/11/the-earth-is-flat/ (visited on 2018-04-14).

[11] Marius Muja and David Lowe. *FLANN – Fast Library for Approximate Nearest Neighbors. User Manual*. Version 1.8.4. 2013-01-24. URL: https://www.cs.ubc.ca/research/flann/uploads/FLANN/flann_manual-1.8.4.pdf (visited on 2018-03-05).

[12] Travis E. Oliphant. *Guide to NumPy*. 2006-12-07. URL: http://ns.ael.ru/ports/distfiles/numpybook.pdf (visited on 2018-03-21).

[13] Edouard Oyallon and Julien Rabin. "An Analysis of the SURF Method". In: *Image Processing On Line* 5 (2015), pp. 176–218. DOI: 10.5201/ipol.2015.69.

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[15] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Draft. Springer, 2010-09-03. URL: http://szeliski.org/Book/drafts/SzeliskiBook_20100903_draft.pdf (visited on 2018-01-14).

[16] *The Python Language Reference*. Version 3.6. Python Software Foundation. 2016-12-23. URL: https://docs.python.org/3.6/reference/index.html (visited on 2018-03-21).

[17] Zhou Wang et al. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004-04), pp. 600–612. ISSN: 1057-7149. DOI: 10.1109/TIP.2003.819861.

[18] Stewart Woods. *Eurogames: The Design, Culture and Play of Modern European Board Games*. McFarland & Company, Inc., 2012. ISBN: 978-0-7864-6797-6.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, ganz oder in Teilen noch nicht als Prüfungsleistung vorgelegt, und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet.

Bayreuth, den 23.04.2018 .....