EasyTikZ

converting "Kritzeleien" to TikZ-code

Robin Holländer
108014394238

Niklas Heyne
108013208773

19.07.2019

1 Introduction

The desire to support a written publication with graphical figures presents the author with the necessity to choose a suitable format for said graphics. When comparing the two major classes *raster graphics* and *vector graphics*, the advantages of the latter over the former in presenting abstracted concepts like diagrams or line graphs become obvious. *Vector graphics* are scale-invariant and do not compromise information by blurring. When using a limited number of colors and parameterized shapes, *vector graphics* take up much less storage space than comparable *raster graphics* and minor changes like changing the color of shapes of slightly altering their outline can be accomplished effortlessly by a trained user.
With the production of scientific publications often being a long-term development process with several contributors, keeping the majority of the project version controlled greatly supports the advantages of collaboration.
TikZ is a markup language for the description of *vector graphics*, which can be placed directly into LaTeX documents. The advantages of straight-forward parameterization and version control come at the price of a cumbersome user interface and a laborious process of graphics design.

Out project was meant to facilitate the process of using *TikZ* by developing a software solution for the conversion of hand-drawn images into human-readable *tikZ*-code. The main focus was not to produce the most detailed representation of the input but rather to reliably provide a basic scaffolding as a base for further refinement.

Over the course of three months, the scope we initially expected for our project hat to be reduced several times due to our unfamiliarity with the programming language, over-estimation of the applicability of existing software and some strategic mistakes we made during the development process.

2 requirements

The initial list of features, which was developed in cooperation with our supervisor, was partitioned into necessary, expected and treat-features for each of the two modules.

raster graphics → semantic model

necessary	expected	treats
• detect and recognize a limited number of shapes • detect connections between shapes • detect and recognize labels inside of shapes • detect freely positioned text • basic automatic alignment of structures	• recognize arbitrary polygons ≤ 6 vertices (within reasonable limits) • robustness against discontinuous lines • better automatic alignment of structures • recognize specific parameters for shapes (rounded corners, fixed angle rotations, line thickness) • recognize relative text position in shapes	• allow for parameterized alignment thresholds • recognize groups of identical shapes • recognize different endpoints of connectors

semantic model → TikZ code

necessary	expected	treats
• human readable • intuitive structure of code • generation of shapes • generation of connections • generation and targeted placement of text	• easily adjustable cosmetic preferences line thickness, possible grid etc. • code optimization	• use loops to better parameterize groups of shapes • extend adjustment capabilities to store variables for group parameters

3 General Workflow

Our Project is comprized in two independent modules, the *recognition-* and *TikZ-generator* module, which interface via a representation of the processed diagram. Both modules each complete a number of sub-tasks in order to achieve their gross purpose.

3.1 recognition module

The *recognition module* works by completing the following sub-tasks:

1. **Image preprocessing**

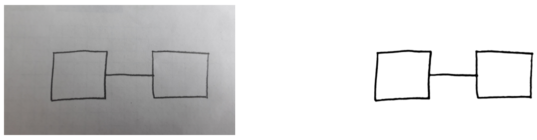


Figure 1: input image before and after preprocessing (without cropping and scaling)

The input image is likely to be an RGB-image of a hand-drawn diagram on a piece of paper or a whiteboard, which was taken with a digital camera of medium to low quality and under inconsistent lighting conditions.
In comparison with images obtained by scanning a piece of paper on a high resolution flatbed scanner, medium quality digital photos bring with them a number of adverse conditions for reliable image recognition tasks, namely a potentially strong lighting gradient across the image, sub-optimal contrast due to inconsistent color representation, high amounts of noise and varying resolutions. Further difficulties are induced by special qualities of the pictured diagrams because of the difference in line thickness, the poor signal-to-noise ratio of a picture taken of a homogeneous background with a small foreground area where lines are drawn and the general inconsistency and unpredictability of human hand-drawing.
The image preprocessing step aims to provide the *recognition module* with input of a standardized format by rescaling the image, adjust the relative size of the image and the drawn diagram and separating content from background via binarization.

2. **Shape recognition**
Having binarized the input image, drawn shapes have to be recognized to become vertices of the final graph structure. During the shape recognition step the binary input image is further modified to distinguish between those pixels that make up shapes and those that make up connections between them. A binary image containing only the filled shapes as foreground is then handed to a contour finding algorithm and a Polygon Approximation algorithm in succession to obtain parameterized polygonal shapes. The shape image is also used with a Hough-like transform to find parameterized circular structures.

3. **Connection recognition**
In order to obtain the lines that make up connections between shapes, the shape image from the shape recognition step is used to erase the shape outlines and leave a binary image with just the remaining lines as foreground.
On the line image we detect corners (including endpoints and intersections) of lines and perform

a line detection via the probabilistic Hough-transform. In order to remove duplicates and noise induced false detections, we have each of the resulting hough-lines vote for the combinatorial connection of two corners that it is most similar to, according to the Metric for line segments proposed by [Nacken, 1993]. The resulting collection of line segments, after thresholding at a specific amount of support, is a subset of all possible embeddings of undirected connections between two of the previously detected corners.

4. **Graph construction**
   The interfacing *Diagram* class against which the *recognition module* was implemented, maintains a collection of Nodes and a collection of Connections that each connect two of the nodes. To provide the Diagram class with this connectivity information, the Graph construction step first assigns each endpoint of each detected line segment with up to one suitable shape before merging remaining free endpoints to form intermediate nodes, creating a collection of graph embeddings.

5. **Connectivity detection**
   The collection of graph embeddings is traversed via depth first search from each unvisited Shape, adding encountered intermediate nodes to a connection object until another shape is found and the completed connection with a starting and ending shape and a list corner embeddings in-between can be stored.

6. **Diagram construction**
   The shape nodes from the graph are converted to their respective counterparts used in the *Diagram* class before shapes and constructed connections are inserted into a *Diagram* object and handed to the *TikZ-generator*.

3.2 TikZ-generator module

The *TikZ-generator module* has one public method which completes the following sub-tasks.

1. **Diagram alignment**
   In order to improve the look and readability of generated graphs, the *TikZ-generator* first calls a method on the *Diagram* object which aligns the coordinates of all nodes and intermediate corners of connections. This method can be called with different *AlignmentOptions* as parameter, each implementing a different process of alignment.

2. **Generation of optional output**
   With command line arguments the user can choose what code will be generated in addition to the basic graph. If cosmetic variables are added, the graph generation will also be altered to include *tikzstyle* interfaces in its elements. For each supported type of element a unique *tikzstyle* line will be added to easily manipulate presentation. Furthermore, commented out TikZ-code drawing the grid to which all coordinates were aligned is added as part of the cosmetic variables option. The *TikZ-generator* works by continuously appending information as strings to one singular string which eventually will be printed to a file as output. Because of this, the code for LaTeX and TikZ environments has to be split appropriately so it can envelop the generated graph.

3. **Generation of TikZ graph**
   The graph itself is generated by appending code for all rectangles, circles, polygons and then

connections described in the *Diagram* object. Diamonds are considered rectangles and will not be sorted differently. Every shape generated has a minimum size, root coordinates, an identifier and a field for text. Depending on user input this field is either left blank or it contains the shape's identifier for easier association of TikZ-code and compiled picture. Interfaces for *tikzstyle* are not added if cosmetic variables are disabled. If applicable, the second parts of the LaTeX and TikZ environments are the last bits of information which will be added to the output string.

4. **Printing output to file**
   The different elements of the desired output are already part of the virtual string, which now is saved as "EasyTikZ.txt".

4 Image → semantic model

4.1 Image Processing

4.1.1 Thresholding methods

To process the drawn lines in the input image, the foreground(drawn lines) had to be separated from the background(the rest of the image). Given the common practice of drawing with ink much darker than the drawn-on surface, we searched for a thresholding method to determine for each pixel whether it belonged to the fore- or background.
To achieve a reliable binarization of the input image several thresholding methods were investigated. The methods were mostly prototyped and tested in MATLAB before their integration into the project.

- **Otsu's Thresholding method**
  Otsu's thresholding method is a popular method for finding a threshold for the binarization of grayscale images. It works by exhaustively searching all possible thresholds for the one that minimizes the intra-class-variance of the two resulting classes. The Algorithm is frequently used in the separation of foreground- and background pixels in microscopic images. As most thresholding algorithms, Otsu's method requires the input image to feature a bimodal histogram with distinguishable peaks for foreground and background pixel values. Besides the uneven lighting on manually taken photos, the low signal-to-noise ratio in the expected input images (largely more background than foreground pixels) make the desired foreground pixels become small noise peaks in the stretched out background spectrum. Figure 2 shows an expected input image and a microscopy image of a blood smear(a typical use case for Otsu's method) as well as their respective histograms. One can easily see that the noise peaks in the large background blob of the first histogram are much taller and more defined than the small bulge at the lower end of the lightness spectrum. The second image produces similarly sized peaks with a clear minimum in-between.

- **Adaptive Thresholding** The adaptive thresholding method produces an individual threshold for each pixel by taking into account its local neighborhood of pixels. The idea is to account for slowly varying lighting intensity across the image. We discarded this method due to a large number of small artifacts which appeared at the borders of the tiles as an effect of the built-in OpenCV function. The morphological denoising of the binarized image would have compromised the integrity of the shape outlines, rendering the shape detection method ineffective.

- **Rosin's Algorithm** Rosin's algorithm works by drawing a straight line from the highest value of the histogram to the end of its tail and choosing the value of the histogram bin the top of which is furthest from the line. Rosin's algorithm (aka maximum deviation algorithm) is said to perform well on unimodal histograms which are infamous for posing a challenge for most thresholding algorithms. During our tests, the tried implementation [Khan, 2014] performed mostly identical to Otsu's method while occasionally producing ridiculous thresholds outside of the value range.
  With Rosin's algorithm having originally been proposed as a method for thresholding edge images, we held the irregular line thickness responsible and did not further pursue this option.

- **Gradient magnitude thresholding** We hoped to be able to threshold an edge image created by using the Canny edge detector or bare Laplace- and Sobel filters but as each line would

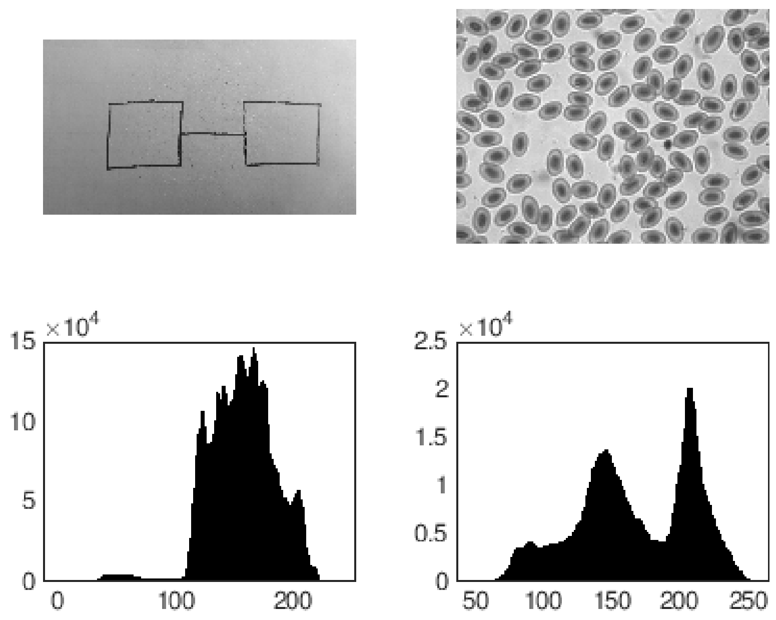


Figure 2: **left side:** An unevenly lit image with low SNR and its histogram **right side:** An evenly lit microscopy image with higher SNR and it's bimodal histogram.

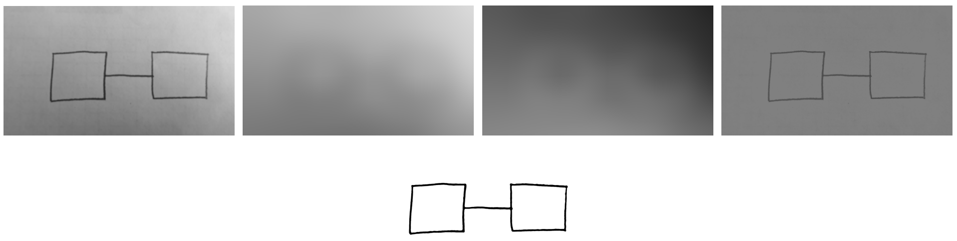


Figure 3: **top row:** input image, extracted gradient, inverted gradient, mean of inverted gradient and input image; **bottom row:** mean image thresholded with Otsu's method

have produced high values of gradient magnitude on the rising as well as the falling edge, we would have had to adapt our shape- and edge finding algorithms which would have taken too much time.

- **Gradient subtraction**
  We developed a method of binarization which yields good results in our MATLAB implementation. The procedure, which is illustrated in figure 3, consists of extracting the lighting gradient by convolving the input image with a gaussian kernel of very large standard deviation and computing the mean of the input image and the inverted gradient. After denoising with a small Gaussian filter, Otsu's thresholding method can be applied to separate foreground and background.
  Unfortunately, the quality of the result depends on the two gaussian kernels having the right size for the resolution of the input picture. With our cropping and rescaling step being applied to the binarized image, we could not binarize the already cropped and rescaled version. The Process remains unimplemented in our final solution.

The difficulties in finding a suitable thresholding algorithm forced us to rely on using Otsu's thresholding method which limits input images to either being evenly lit(scanned) or already binarized(drawn digitally).

4.2 Detection of Shapes

OpenCV provides a reliable toolchain for the recognition of polygonal shapes. A common workflow is to first find contours in a binary image via an algorithm proposed by [Suzuki et al., 1985] and then approximate Polygons of lower vertex counts by using The Douglas-Peucker algorithm [Douglas and Peucker, 1973]. While the OpenCV library provides tested and efficient implementations of those algorithms,

The main complications imposed by the nature of the problem were the following:

- **variable line thickness**
  The standard way of finding linear structures in an image is to perform the hough-transform on the output of an edge detector(e.g. Canny) which assigns each possible line from a defined parameter space an intensity value, which after thresholding yields the parameters of the most clearly defined lines in the image.

  Running an edge detector on our input image, which – except for the varying line thickness – was an edge image, to begin with, however, yielded two limiting outlines for each linear structure, which introduced the necessity for a method to cluster the resulting hough lines.

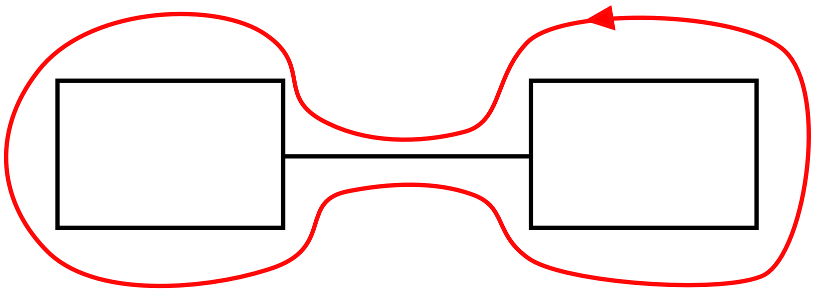


Figure 4: Dumbbell shaped outline of two connected polygon outlines

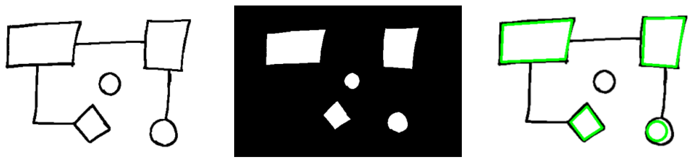


Figure 5: Caption

- **distinguishing between connections and shape outlines**
  As there is no difference between the lines that make up shapes and those that make up the connections in-between, the contour finding algorithm from [Suzuki et al., 1985] could not simply be used on the unmodified input image. The results would have been Approximations of the outlines of connected components as shown in figure 4.

The problem of varying line thickness alone could relatively easily be tackled using heuristics for morphological opening and closing steps during image preprocessing, which would reduce line thickness. However, our solution for telling shapes from connections required all shapes to be closed completely. Therefore we could not risk to morphologically damage the shape outlines.
The first idea was the construction of shapes from detected linear structures in the image but the limited time made it unfeasible to achieve sufficiently reliable line segment detection and develop an algorithm to construct geometric shapes from them without accepting large errors and jeopardizing the usefulness of the application.
Before we abandoned this first approach, we implemented the line segment clustering algorithm by [Jonk and Smeulders, 1995], which is based on a modified version of the metric described in [Nacken, 1993]. We would later use the metric itself to support the detection of lines.

Our solution was to require all shapes to be completely closed, while all cycles in the drawn graph must have at least one opening. This way we could use a regular bucket-fill algorithm from one of the corners of the input image to color all background pixels on the outside of shapes in the color of the lines and leave only the shape insides untouched. We could then perform shape recognition on the insides of the shapes as illustrated in figure 5.

4.3 Detection of lines

Having constructed the binary image with only the insides of the shapes having foreground color, dilating(inflating) those shapes to cover the shape outlines and performing a logical **and** of its inverse and the original binary image produced an image which contained only the lines which connected

the shapes.
On the resulting image the *Harris corner detection algorithm*, which is implemented in OpenCV, is used to detect endpoints, corners, and intersections on the binary edge image. Those corners will serve as possible endpoints for line segments to reduce the number of possible candidates for connecting edges.
In order to detect the line segments which make up the inter-shape connections, we use the *probabilistic Hough Line function*, which is also part of OpenCV, to directly find edges on the binary edge image. The possibly high line thickness leads to each original line giving rise to the detection of several similar hough lines, which have to be filtered to yield useful information.
To find the line segments which make up the connections, we have each hough line "vote" for the connection of two detected corners, that it is most similar to, according to the metric for line segments which was proposed by Peter Nacken in [Nacken, 1993].
The metric takes into consideration the difference in angle as well as the distance between the centers of line segments. After keeping only those corner connections with sufficient support from detected line segments, the remaining line segments and shapes are used to create a graph.

4.4 Construction of the semantic model

Having detected the desired shapes and line segments from which the connectors would have to be created, a method for constructing the finished graph was needed. As all detected edges are connections between previously detected corners, we needed to distinguish between corners that represented the intersection of a connection and the outline of a shape on the one hand and the intersection of connections on the other.

Incidence of shapes and lines

The method we developed to associate a shape and it's incident edges is based on the proximity of the edges' endpoints to the center of the shape and on the length of the projection of the shape's centroid onto the line through the line segment in question. In order to be accepted as an incident edge of a given shape, an endpoint of the line has to be closer to its centroid than the centroid is to the closest vertex of the outline. Also, the infinite line on which the line segment is situated has to pass through the circle around the centroid that is defined by the closest center of one of the outline's edges. The method is illustrated in figure 7. In the first step of graph construction, each detected shape is wrapped in a *NodeShape* object that provides the functionality of a graph node. It is then linked to one valid endpoint of each viable edge by iterating over all endpoints and testing for the above-mentioned criteria. Left endpoints are favored which causes the fringe case illustrated in figure 7 where a line, being completely inside the outer radius of a shape on its right side and having its left endpoint assigned to it, rejects a shape on its left side which would otherwise be viable.

Construction of corners and intersections

To enable our algorithm to connect shapes via the detected line segments, several line segments had to be connected at the endpoints where they form corners and intersections. We chose not to handle lines with free endpoints because the necessity of connecting corners and shapes helped in weeding out the still substantial number of falsely detected line segments.
We construct the intersections by iterating over all pairs of endpoints of line segments and merge loose endpoints into corners if their distance is below a user-defined limit. Endpoints are preferably merged into existing corners before they are compared with other singular endpoints.

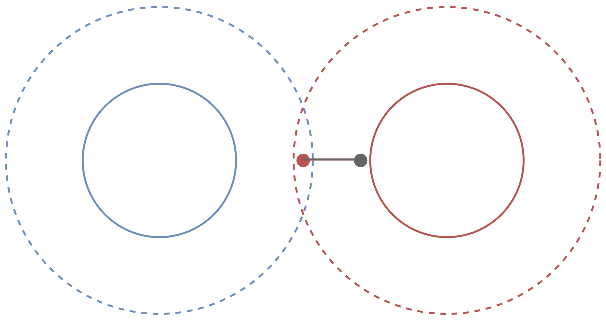


Figure 6: An example where a large outer radius will cause wrong connection of shapes and edges if the right shape happens to be processed first

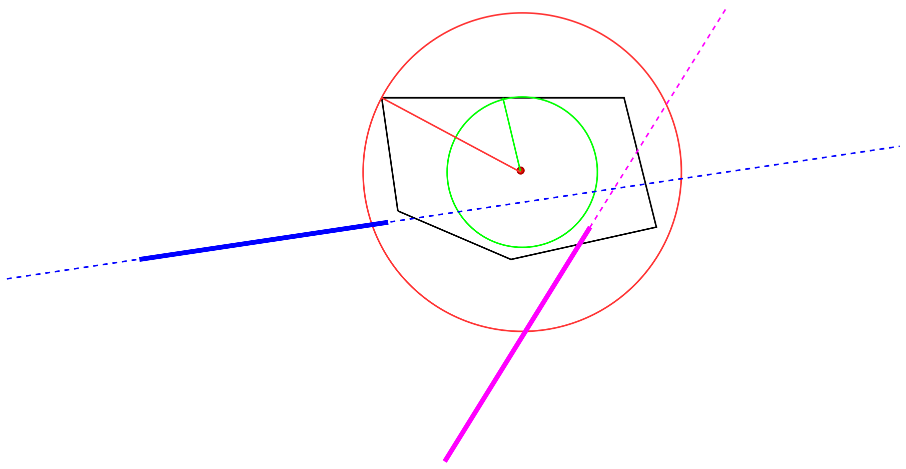


Figure 7: A polygonal shape with it's inner radius (green), outer radius(red), an accepted line segment(solid blue) and a rejected line segment(solid purple). The lines which are defined by the line segments are displayed as dashed lines in the respective color. The solid blue line segment is accepted because it's right endpoint lies within the outer radius and it's infinite line passes through the inner circle. The purple line segment is rejected because it does not pass through the inner circle.

The *NodePoint* class wraps the constructed corner coordinates to provide the functionality of graph nodes.

Graph traversal

The *depth first search* is carried out by the *Node* objects wrapping shaoes and corner coordinates. To cover all unconnected graph components, a DFS is started from every shape. A common container of *Connection* objects exists, which can be accessed by using a particular *Node* object n as a key. In case of a hit, the stored *Connection* always starts with a shape and ends with the last *Node* encountered before n.
When assuming the role of the *current node* during the DFS, a *NodeShape* will either finish an unfinished *Connection* by emplacing itself at the end and storing the *Connection* to be returned or, if it has no *Connection* associated to itself, or begin a new DFS. A new DFS is also started if a *Connection* was finished and the adjeacent Node from where the last *Connection* came will then be marked and therefore ignored.
NodePoints, which wrap corners and intersections, cannot finish *Connections* and will only append themselves a copy of their associated *Connection* and associate that copy with all unmarked, adjacent *Nodes*.

Following this procedure finds pairs of Shapes which are connected by a sequence of line segments, which in turn are connected by there endpoints being merged into corners. Each resulting *Connection* references the two *NodeShapes* it connects, as well as an optional list of coordinates through which the connecting line should pass.

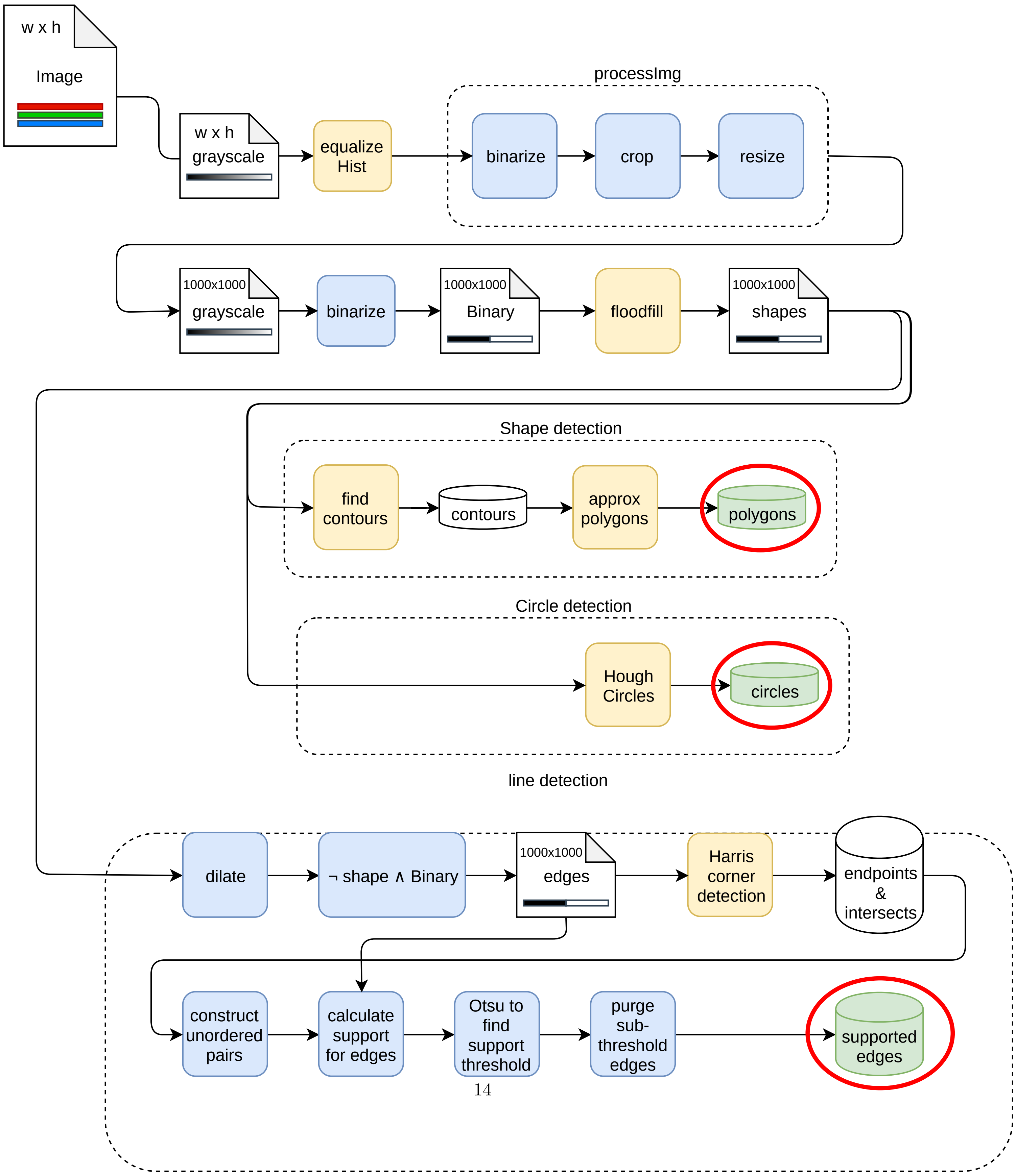


Figure 8: Image Preprocessing

5 Semantic model → TikZ code

Compared to the complexity of constructing the semantic model based on an input image, the challenges and problems encountered whilst working on converting that model to TikZ code are trivial.

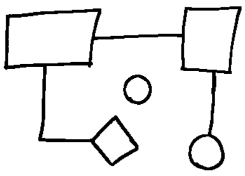
5.1 Diagram Modelling

The *Diagram* class is designed with the generation of TikZ code in mind. After we evaluated what would be necessary information in order to have worthwhile output we formulated basic requirements. Every element has to be easily accessible whilst maintaining the ability to be passed between methods as part of a group of elements. New types of elements added on a later date should be easy to integrate into existing collections and methods.
We decided to create the class *Shape* dealing with the most basic information every generated shape would have, and to implement derivatives as needed. As a result of that, more general operations are able to process all shapes as a collection, regardless of their specialized attributes. In order to maintain modularity, the collection of shapes is saved as an unordered map with the type of shape as key and a vector of shapes of corresponding type as value. New keys and related shapes can be added effortlessly. Objects based on the *Connection* class are stored separately in a simple vector and thus far there is no support for derivatives.
Later in the project we added a method for calling code from *AlignmentOptions* to the *Diagram* class, as the values which are to be aligned belong to *Diagram*. This also keeps in mind our goal of modularity, with the method accepting every class inheriting from *AlignmentOption* as parameter.

5.2 TikZ Code Generation

The *TikzGenerator* class writes LaTeX/TikZ code to "EasyTikZ.txt" based on the given *Diagram*, *AlignmentOption* and several other parameters mostly comprising flags. Whilst this basic idea remained the same during the timespan of the project, the first implementation was deemed unsuitable for our scope. The original *TikzGenerator* would draw single lines in TikZ in order to create shapes visually, instead of using the existing node/shape functions the current iteration works with. The reasoning behind the first method was that drawing the desired outcome line by line is very modular. It quickly became apparent though, that this approach was too complicated even after several improvements to the system. Adjustments regarding layout, connections, visual appearance and other properties would be convoluted and time-consuming, defeating our stated goal of reduction of user workload.
The current *TikzGenerator* makes use of TikZ' node system. Generated shapes are easier to connect thanks to their unique identifiers and the connection won't be broken by adjusting the graphs layout. The different parameters available to nodes facilitate further modifications after the TikZ code has been generated and generally help usability. The usage of *tikzstyles* is also aided by the new output format.
Before converting the information stored in the *Diagram* object to TikZ code, a method responsible for alignment is called on aforementioned object. Our first *AlignmentOption* would move the root coordinates of nodes onto a grid whose size was determined by the smallest shape. This lead to very small grid-sizes whenever a small shape was part of the diagram, resulting in less effective alignment. The current default *AlignmentOption* is based off that first prototype, but instead of only checking for the smallest shape, which we originally did to avoid major displacement of small elements, its grid-sizes are based off the average size of all shapes.

input image



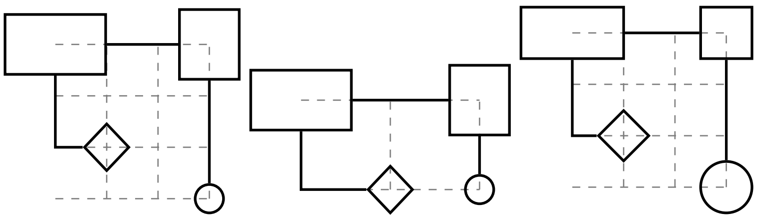
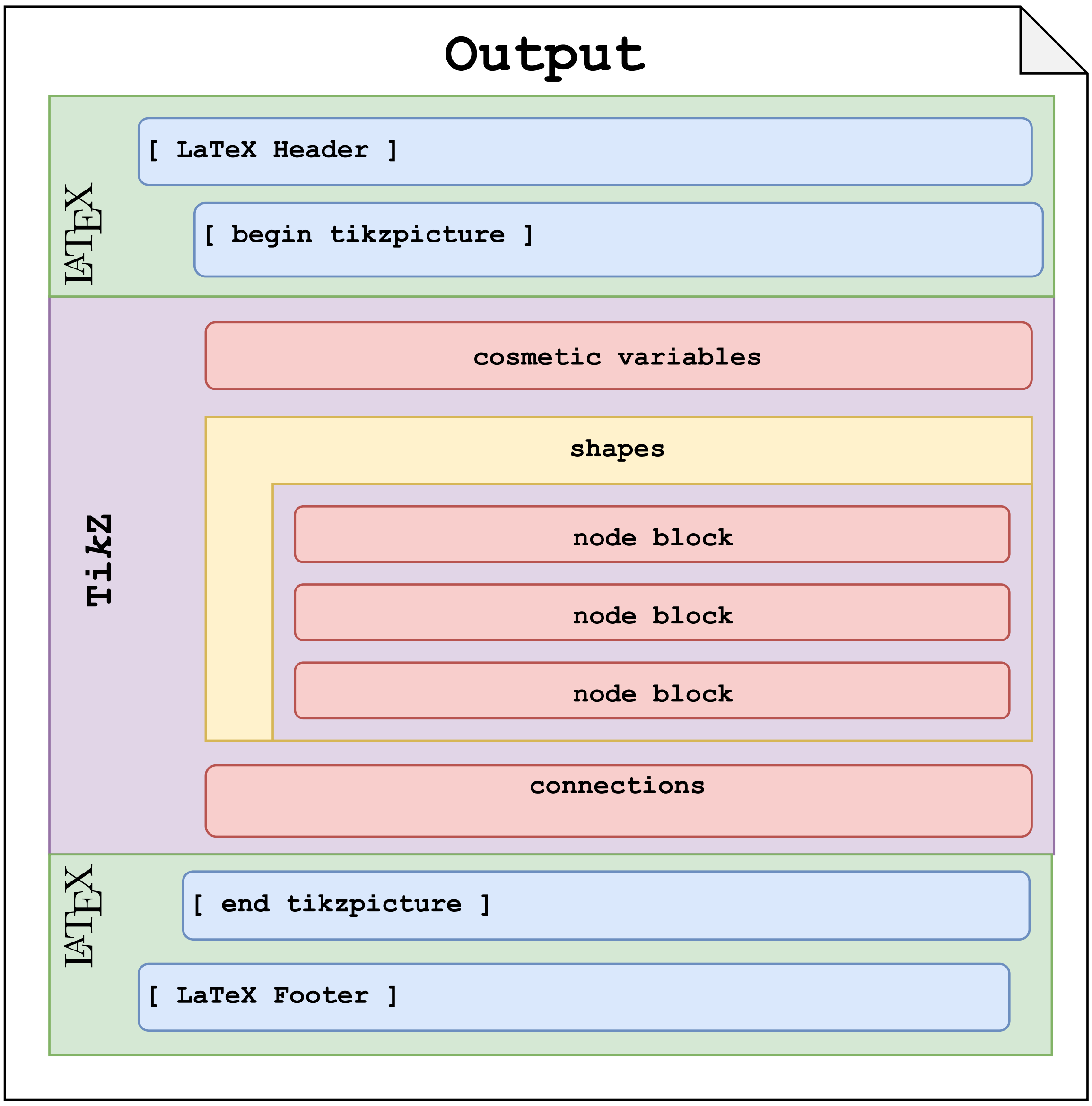


Figure 9: :
DefaultAlign, ManualAlign(1.18,1.18), SizeAlign

We have implemented two more derivatives of *AlignmentOption*; *ManualAlign* and *SizeAlign*. The former enables the user to determine grid-sizes for alignment manually, the latter adjusts the sizes of shapes depending on the automatically determined grid, in addition to what *DefaultAlign* does. The generation of the TikZ code itself as well as saving that output to a file is a simple task and has been explained before. Again it is to be noted that the user can specify whether they want lightweight output with just the TikZ picture or rather code for an entire LaTeX document, with different levels of detail inbetween.

5.3 Output

The output format is meant to be as intuitive as possible with the goal of being a useful foundation.



6 Current state

7 Report

- **Did the formulation of your task or your perception of it change over time?**
  Yes. In the beginning ourselves as well as our supervisor underestimated the complexity of the problem and the effort required to deal with it. In combination with our initial overestimation of our capabilities and the applicability of several existing software solutions, an adjustment of expectations towards the final product had to be made. Core requirements remained unchanged throughout the project.

- **Would you classify your scheduling as adequate? How often did you have to adjust your planning and why?**
  No. As outlined earlier, our flawed perception of the expected workload lead us to schedule achievement of certain features prematurely.

- **What methods did you implement to support your collaboration and how did they perform?**
  We set off using the Kanban software *Trello* to keep track of our tasks, categorize them based on urgency and assign them to a specific member of the team. We also maintained a readme on our Gitlab with links to resources for research on relevant topics and possible inspirations for additional features. The documents our team wrote were shared across all members with the help of overleaf.com, an online service for collaborating on LaTeX documents, where we also experimented with TikZ code.

- **How well did your coordination work? Which problems arose and how did you tackle them?**
  We upheld a steady state of active communication which resulted in very little organizational issues or mis-information. Our small team size was beneficial in keeping all members sufficiently informed and engaged at all times.

- **Did you feel sufficiently qualified for the project? Which skills did you have to acquire during the project? What did you learn from your experience?**
  Our lack of familiarity with the C++ programming language and the best practices for its application proved to be a major challenge, greatly hindering progress in the early stages of the project. Thanks to our supervisor's experience and ongoing support we were able to improve our skills substantially, albeit not without causing a delay on actual progress resulting in a reduction of scope. We also had to familarize ourselves with opencv and several image processing techniques.

- **How did you try to achieve maximum efficiency and personal reliability?**
  We established a regular meeting twice a week to discuss current challenges, proposed solutions and the interfacing of our respective submodules.

- **Would you approach your project differently if you were to begin anew considering the lessons learned?**
  Based on the knowledge that most software we planned on using doesn't perform as we hoped it would, yes. The development should be more focused, diverting less attention towards the introduction of new feature before thoroughly finishing and testing existing ones. It also is of

vital importance to outline the implementation of features in way greater detail before actually working on it.

- **What were the challenges caused by your group's dynamics and how did you handle them?**
  Our group did work on a smaller project before so we did not have to get acquainted with each other. Akin to said project we had, partly due to differences in experience and knowledge regarding actual programming, diverging opinions regarding the amount of features we would strive for. In order to avoid the repetition of mismanagement and a resulting greatly increased workload, we spend more time on formulating feature-based goals and their importance, as well as discussing them with our supervisor. One member of the team also was heavily invested in the project's topic and more familiar and comfortable with the tools required, being very motivated and at times overly optimistic.

References

[Douglas and Peucker, 1973] Douglas, D. H. and Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122.

[Jonk and Smeulders, 1995] Jonk, A. and Smeulders, A. W. (1995). An axiomatic approach to clustering line-segments. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 386–389. IEEE.

[Khan, 2014] Khan, S. (2014). Rosin unimodal thresholding. `https://de.mathworks.com/matlabcentral/fileexchange/45443-rosin-thresholding`. Accessed: 2019-07-10.

[Nacken, 1993] Nacken, P. F. (1993). A metric for line segments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(12):1312–1318.

[Suzuki et al., 1985] Suzuki, S. et al. (1985). Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, 30(1):32–46.

8 Appendix

8.1 Progress

18.04. - 24.04.

Niklas	Robin
• got opencv working • implemented(or customized) basic shape recognition • set up git submodule for the documentation document which turned out to be of no use at all • did research and updated readme	• studied TikZ and different methods for shape generation

25.04. - 01.05.

Niklas	Robin
• experimented with different thresholding methods • adapted existing code to be C++ rather than C. • familiarized with the openCV Hough transform as an alternative or support for built in shape detection • created diagram for visualization of TikZ output format	• developed a formalized structure for the TikZ-output • wrote classes ”Shape” and its derivative ”Rectangle” • wrote Class tikzGenerator to output a TikZ snippet of parameterized rectangles.

02.05. - 09.05.

Niklas	Robin
• replaced openCV shape recognition with hough transformation • included Harris corner detection in the shape recognition process • searched for ways to distinguish between connectors and shapes • implemented line segment metric as proposed by Nacken in 1993 • implemented line clustering method by Jonk & Smeulders • began implementation of graph representation	• cleaned up code, implemented new knowledge regarding best practices • reworked TikZ code generation, now using nodes instead of free drawing • reworked the way tikzGenerator handles Diagram input • reworked (classes containing information) Shape, Rectangle and Diagram • implemented Connection class and generation of connections between nodes • familiarized with current implementation of image recognition and new methods

09.05. - 20.05.

Niklas	Robin
• familiarized with boost graph library as alternative to implementation from scratch • implemented boost graph with custom vertex-and edge properties • implemented custom vertex discovery handler for use in boost depth first search	• Converted all pointers and cleaned up code structure • Reworked passing of Shapes between classes • struggled with installing opencv on old hardware

27.05 - 03.06

Niklas	Robin
• —	• Restructured Diagram • Added framework for basic grid-based alignment

03.06 - 10.06

Niklas	Robin
• Modified main.cpp to produce first ever complete runthrough • Cleaned code, fixed warnings	• Implemented new modular structure for AlignmentOptions • Reworked all files dealing with alignment

10.06 - 17.06

Niklas	Robin
• Added testfile and cleaned up main • Added support for command line flags for tex- and tikz-environment • Implemented circle detection via Hough circle transform	• Added framework for intermediateCoordinates of Connections • Added framework for rectangles treated as diamonds • Added support for tex- and tikz-environment flags

17.06 - 24.06

Niklas	Robin
• Found and fixed rescaling error • Implemented construction of graphs for connectivity modelling • Implemented base for DFS and Connection generation • Started work on intermediate corner detection	• Added support for Circle generation • Tweaked the way nodes are inserted into Diagram, tweaked output and alignment

24.06 - 01.07

Niklas	Robin
• Implemented construction of Shape objects from polygons, rectangles and circles • Implemented construction and traversal of graph structure and generation of connections • Fixed Edge construction as well as multiple errors in DFS • Worked on Node→Shape-derived conversion	• Added support for regular polygon generation • Added support for Diagrams without any rectangles, adjust DefaultAlign accordingly • Added scaling of size of shapes to DefaultAlign, improved automaticGridSize • Worked on Node→Shape-derived conversion

01.07 - 08.07

Niklas	Robin
• Fixed object slicing • Tweaked outer radius computation for improved circle connectability • Fixed many bugs	• Added support for alignment of intermediate coordinates, reworked structure of AlignmentOption • Added support for manual gridSize

08.07 - 15.07

Niklas	Robin
• Cleaned up command line parameters and enabled user specific grid size for ManualAlign • Added additional command line parameters • Added config file for command line parameter defaults, can be set via command line flag	• Tweaked default values for possibly omitted flags, tweaked DefaultAlign and ManualAlign

15.07 - 22.07

Niklas	Robin
• Wrote and commented new thresholding method	• Added generation of cosmetic variables • Added additional command line parameters, tweaked some old ones • Added SizeAlign, removed scaling of size from DefaultAlign and ManualAlign

22.07 - 29.07

Niklas	Robin
• —	• —