Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Theoretische Informatik

The Rhythm of Relations of Color and Size: Detecting Rectangles in Paintings Created by Piet Mondrian

Finn Pauls
Matrikelnummer: 4788442
finn.pauls@fu-berlin.de

Betreuer: M.Sc. Martin Skrodzki
Eingereicht bei: Prof. Dr. Wolfgang Mulzer
Zweitgutachter: Prof. Dr. Konrad Polthier

Berlin, 1. Oktober 2018

Abstract

The arrangement of compositional elements in abstract paintings by Piet Mondrian have been of interest to researchers for a long time. Despite this interest, there are no publications examining these paintings numerically across a broad pool of data. This thesis tries to change this lack of data by describing an algorithm for detecting rectangles in Mondrian's compositions from 1920 to 1937. It also provides a descriptive analysis of the resulting data. While there are no hints that Mondrian used specific aspect ratios, a bias for positioning colors could be revealed.

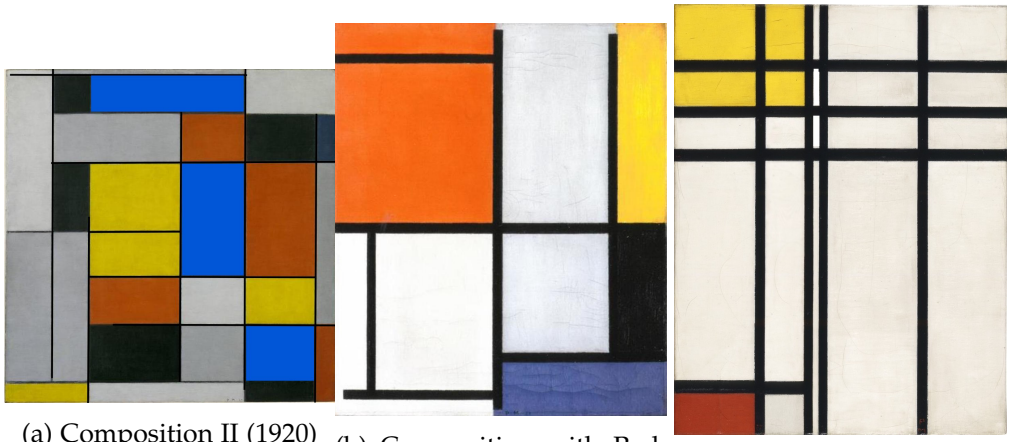
Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

1. Oktober 2018

Finn Pauls

Contents



(a) Composition II (1920)

(b) Composition with Red, Blue and Yellow (1930)

(c) Opposition of Lines: Red and Yellow (1937)

Figure 1: Three examples of Mondrian paintings from 1920, 1930 and 1937.

1 Introduction

1.1 Piet Mondrian

The oeuvre of Piet Mondrian (1872-1944) consists of figurative paintings as well as abstract compositions. Up until 1910, his works only depicted naturalistic scenes like churches, trees, windmills or landscapes. From 1911 onwards his paintings remain representational but are increasingly painted in a more abstract way.

Around 1914, his compositions begin to be purely abstract and in 1917 he cofounds the artistic movement and group *De Stijl*, also known as *Neoplasticism*. One of the primary objectives of the group is to reform art by "abolishing natural form" [20] altogether.

At some point around 1920, Mondrian further restricted his compositional elements to rectangles and straight black lines. He only allowed lines to run horizontally or vertically and only used primary colors (red, blue and yellow) and non-colors (black, white and grey). See Figure 1 for examples of Mondrian paintings with these elements.

Another founding member of *De Stijl*, Georges Vantongerloo, used methods inspired by mathematics for his compositions. One example of these paintings is *Composition Derived from the Equation y = -ax2 + bx + 18*.

Mondrian, on the other hand, is known to only use intuition when creating his works, moving compositional elements around for weeks, seeking a balanced composition, until he was satisfied with the result. [6]

Still, many people have been interested in the question if Mondrian's non-figurative paintings utilize certain compositional rules or techniques that the painter might have used unconsciously. One approach to finding structure in Mondrian's paintings is to create art that resembles his compositions using computers.

1.2 Related Works

There have been multiple attempts at creating art that resembles Mondrian's compositions using computers. Feijs (2004) [7] presents different techniques for generating images resembling non-figurative Mondrian paintings from different periods. He concludes that different kinds of algorithms for generating images can be used to formalize the distinction between different types of Mondrian paintings.

Skrodzki and Polthier (2018) [16] use computer models to generate Mondrian-inspired three-dimensional pieces using KdTree data structures. They note however that while their results are similar to Mondrian paintings, they do not always resemble Mondrian paintings since proportions are chosen randomly and not deliberately.

In 1968 Hill [8] tried to study the organization of the compositional elements used by Mondrian. He analyzed the network topology of Mondrian's paintings. One of his findings shows the avoidance of symmetry on a structural level. He also criticized earlier work for the inaccuracy of measurement and the lack of statistics and called for a more substantiated analysis of the paintings.

Some sources [3, 4] suggest that Mondrian used golden rectangles in his paintings. A golden rectangle is a rectangle where the ratio between the sides lengths is the golden ratio $\phi \approx 1.618$. Other authors [11, 13] however refute those claims, finding fault with the lack of evidence, which consists of mostly exemplary superimposing golden rectangles on paintings. Despite this disagreement, there have been very few attempts to actually evaluate this question statistically. In a poster from 2017, Tanaka and Miyanaga [18] examine the use of rectangle ratios, specifically the golden and silver ratio, in 10 late Mondrian paintings. They conclude that Mondrian actually had a preference for the silver ratio $\delta_S \approx 2.414$ instead.

However, to the author's knowledge, there is no publication examining the question of the use of certain special ratios on Mondrian's paintings with a larger data set.

1.3 Scope of the Thesis

Providing numeric data for closing the lack of foundation for analysis on the neoplastic compositions by Mondrian is the main objective of this thesis. The goal is to create a program that uses methods of Computer Vision to extract the compositional elements from images of the paintings. The program is restricted to paintings from 1920 to 1937, which all use black lines to separate rectangles.

In order to have a uniform and exhaustive representation of the works in question, the *Catalogue Raisonné* [10] was chosen as a source. It includes images of the complete works of Piet Mondrian together with unique identifiers. For the works of Piet Mondrian, this is particularly important as many works are named very similarly or even have the same name.

There are 178 works listed in the catalog for that time period. Of these works, 150 were paintings while the other 28 were sketches, unfinished works or plastic art. For 12 paintings, Mondrian used a tilted canvas, often referred to as *Diamond Compositions*. For the sake of simplicity, these paintings were excluded as well. Additionally, 20 paintings in the catalogue had either only a greyscale image or no image available at all. These paintings were also excluded. The remaining 118 images were selected

for the algorithm. They were scanned and subsequently cropped so that only the painting and no frame was visible. The images were scaled so that the longer side would be 1000 pixels.

Since the distinction between lighter grey to white and darker grey to black is difficult even for a human observer, the detection of non-colors is restricted to white and black only. Additionally, the lines are simplified to not have any thickness, so that they can be represented solely by their start and end points.

2 Fundamentals

This section will specify the different definitions used for conceptualizing the program (2.1). It is also describing different existing methods and algorithms that were used (2.2). Lastly, it outlines the concept for detecting the structure of rectangles in Mondrian's abstract paintings (2.3).

2.1 Definitions

An Image X with width w and height h is defined to be the set I.

$$I = \{p_{xy} | x \in \{1, 2, \dots w\}, y \in \{1, 2, \dots h\}\}$$

An RGB Image I is an Image with the following p_{xy}:

$$p_{xy} = (r_{xy}, g_{xy}, b_{xy}) \in \{0, 1, ..., 255\} \times \{0, 1, ..., 255\} \times \{0, 1, ..., 255\}$$

A Greyscale image is an Image with $p_{xy} \in \{0, 1, \dots 255\}$ and a Binary Image B is an Image with $p_{xy} \in \{0, 1\}$, where 0 is also referred to as *black* and 1 as *white*.

2.2 Used Algorithms and Methods

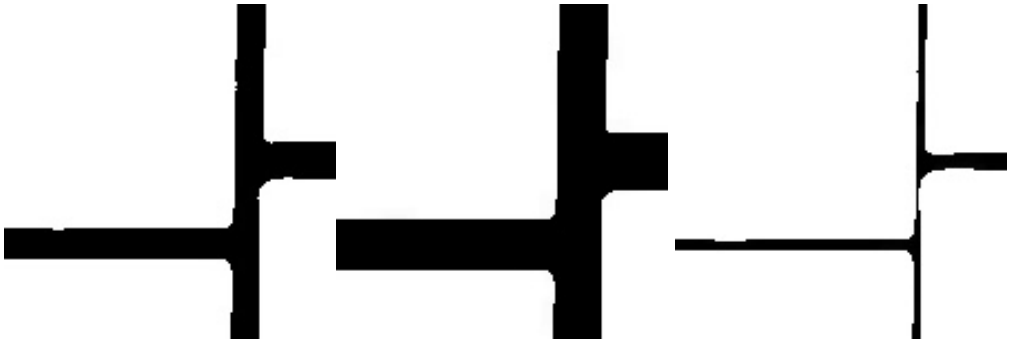
For preprocessing the images, a combination of methods are used: *Image Morphology*, *Thresholding*, *Contrast Limited Adaptive Histogram Equalization (CLAHE)* and *Masks*. These algorithms are provided by the Python distribution of the Open Source library OpenCV [5].

2.2.1 Image Morphology

Image Morphology is based on Mathematical Morphology from the mathematical field of set theory. For the purpose of this algorithm, two basic morphological operators are used. They are going to be explained briefly here, without going into detail of their mathematical definition. A more formal definition and further reading can be found in Aguardo (2012) [1].

Given a binary image B and a smaller binary image B_s called the *Structuring Element*, with a dedicated center point $c \in B_s$, two basic operations can be applied: Erosion and Dilation. Figure 2 illustrates the two operations on a 150×150 pixel example image with a a Structuring Element of 10×10.

Erosion $B \ominus B_s$ is equivalent to going through all of the white pixels of B and translate B_s by its center point $c \in B_s$ to the pixel's position. The pixel is only kept



(a) Input (b) Erosion (c) Dilation

Figure 2: Demonstration of the Erosion and Dilation operation on an example input image using a quadratic Structuring Element.

white if all of the translated points of B_s that are not positioned outside B correspond to white pixels in the image. Otherwise, the pixel is turned black. Therefore the white areas in B are reduced. [17]

Similarly the *Dilation* operation $B \oplus B_s$ translates the center point of the Structuring Element to every pixel $p \in B$. For each of these pixels, all the neighboring pixels in B that intersect with the translated B_s are turned white. The total area of white pixels is increased. [17]

Image Morphologies have multiple uses like increasing certain areas for better recognition or separating larger shapes of an image. For this thesis, they will be helpful for correcting interruptions in the black lines of the image as well as for separating black areas that are bigger than typical lines.

2.2.2 Thresholding

Thresholding $T(G) = \{t_{xy} \in \{0,1\} | g_{xy} \in G\}$ is a process for turning a greyscale image G with pixels g_{xy} into a binary image B using a threshold $t \in \{0,1,\dots 255\}$. Each pixel t_{xy} is assigned a black or white value given whether it is below or above the threshold.

$$t_{xy} = \begin{cases} 0 & \text{if} \quad g_{xy} \leq t \\ 1 & \text{if} \quad g_{xy} > t \end{cases} \tag{1}$$

It is usually used to separate out regions of interest in an image. For the algorithm presented in this paper, these regions will be the black lines of the painting.

2.2.3 Contrast Limited Adaptive Histogram Equalization (CLAHE)

A *Histogram* of an image is the distribution of brightness values in an image. *Histogram equalization* is the process of adjusting the contrast of an image using the Histogram. Global histogram equalization changes the brightness of each pixel in the image using the overall histogram of the image. [15]

Adaptive histogram equalization (AHE), on the other hand, only uses the histogram of a specified area around any given pixel to adjust its brightness. It therefore is able

to increase the local contrast better.

However, *AHE* tends to overamplify noise in an image. Fairly uniform regions of the same brightness have high histogram peaks and are turned into noisy patterns. *Contrast Limited Adaptive Histogram Equalization (CLAHE)* limits this effect by setting a maximum value for the values of the histogram. [14]

2.2.4 Masks

Mask operations are related to elementary arithmetics. Given two images X_a, X_b with the same width and the same height, each pixel a_{xy} in X_a is added, substracted, multiplied or divided with b_{xy} of X_b. For example the operation $X_a + X_b$ denotes the set

$$X_a + X_b = \{\min(255, p_{xy} + q_{xy}) | p_{xy} \in X_a, q_{xy} \in X_b\}.$$

An additional operation on a binary image B_a is the opposite or inversion $\neg B_a$, which is equivalent to a bitwise not.

$$\neg B_a = \{\max(0, p_{xy} - 1) | p_{xy} \in B_a\}$$

These operations will be helpful in combining results from different parts of the algorithm or removing certain areas like black rectangles because later steps should ignore them.

2.3 Concept

The aim for the detection algorithm is to take a cropped Mondrian painting as an input and detect all the rectangles and their colors in it.

The input is an RGB Image I_1 of a Mondrian painting. The expected output is a list of rectangles with their position, size, and color as they are compositionally seen in the painting. All rectangles combined are expected to completely dissect the original image. The thick black lines in the paintings are seen as one-dimensional line segments that are positioned in the middle of the lines in the painting.

The algorithm itself can be separated into two different phases: Image preprocessing (2.3.1) and recognition of rectangles (2.3.2).

The goal of the first preprocessing phase is to obtain a binary image in which black represents the lines of the painting while white represents the inner area of the rectangles. The different steps of the preprocessing phase are visualized in Figure 3.

The second phase then takes this binary image and returns a list of found rectangles. Additionally, for recognizing the colors of the rectangles, the original input image is used.

2.3.1 Image Preprocessing

At first, a Gaussian blur is applied to the input RGB Image I_1 to reduce artifacts that the input images might include. The result of the blur is I_2. These artifacts have several possible origins: They might be craquelure or fading of darker areas in the paintings. But they might also be results of the photography and scanning the images.

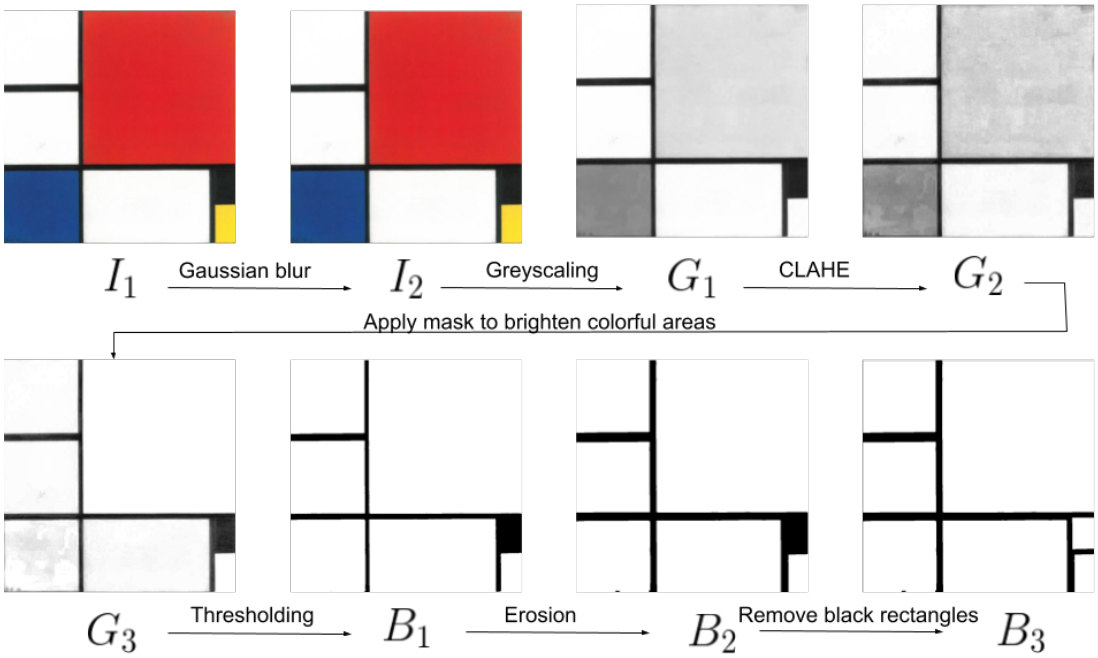


Figure 3: Preprocessing Steps on *Composition II in Red, Blue, and Yellow from 1929*

Next, the RGB input image is decomposed into a Greyscale Image G_1 by using the maximum value of the RGB triplet for each pixel in I_2:

$$G_1 = \{p_{xy} \in \{0, \ldots, 255\} \mid (r_{xy}, g_{xy}, b_{xy}) \in I_2 \text{ and } p_{xy} = \max\{r_{xy}, g_{xy}, b_{xy}\}\}. \quad (2)$$

To normalize the brightness distribution of G_1 and increase the contrast of the darker regions, histogram equalization is applied to G_1. For this we use *Contrast Limited Adaptive Histogram Equalization (CLAHE)*. In contrast to non-adaptive and ordinary adaptive histogram equalization algorithms, this method prevents the overamplification of noise as reasoned above (2.2.3). The result of the equalization is called G_2. [14]

After the normalization is applied, colorful areas of the image are further brightened to make them stand out against the black areas of the paintings. This way, for example, darker blue areas can be better distinguished from black areas. Since the difference between the maximum values G_1 and the minimum value G_{min} of the RGB triplet is related to the colorfulness of a pixel, we calculate the colorfulness for every pixel of the input image. The resulting Mask G_c from the calculation is added to the contrast-normalized image G_3:

$$G_{min} = \{p_{xy} \in \{0, \ldots, 255\} \mid (r_{xy}, g_{xy}, b_{xy}) \in I_2 \text{ and } p_{xy} = \min\{r_{xy}, g_{xy}, b_{xy}\}\} \quad (3)$$

$$G_c = G_1 - G_{min} \quad (4)$$

$$G_3 = G_2 + G_c \quad (5)$$

Now the image G_3 with the elements g_{xy} is converted into a binary image using Thresholding $B_1 = T(G_3)$. The threshold value t needs to be chosen to optimally

separate between the darker lines and the lighter rectangles in the image. See 3.5 for how that value was determined in practice.

Since the thresholding step might leave some interruption in the black lines, the white areas of the image are reduced: An Erosion $B_2 = B_1 \ominus B_e$ is applied. In this case, the structuring element B_e is a $N \times N$ white binary Image. Using the Erosion some accidental interruptions in the lines can be restored. However, choosing a Structuring Element that is too big might result in loss of information by filling smaller white rectangles black. Therefore the size N needs to be chosen carefully.

The resulting image B_2 now separates the darker parts of the image fairly well. But the goal is to detect black lines and Mondrian paintings also include filled black rectangles. To remove these black rectangles, a mask is created that applies a Dilation $B_m = B_2 \oplus B_d$ with a larger Structuring Element B_d ($M \times M$) compared to B_e on the image. The size of B_d is chosen so most of the lines in the paintings are removed, only leaving inner areas of black rectangles. The inverse of the resulting mask B_m is then removed from the image of the last step $B_3 = B_2 - B_m$. Hence only the outlines of the black rectangles, as well as the black lines, remain in B_3, which is the output of the first phase.

2.3.2 Detection and Recognition of Rectangles

Since the rectangles in the painting are defined by the horizontal and vertical lines in the image, the detection starts by finding all of those lines in the output binary image B_3 from the previous phase.

Now all uninterrupted vertical sequences V and horizontal sequences H of black pixels with a minimum length ℓ are considered. You can think of it as a lossy run-length encoding once horizontally and once vertically. The selection of the parameter ℓ is discussed in 3.5.

This means that for every structural line in the painting, multiple line segments are recognized. For example, a line that is 50 pixels wide would be recognized as 50 lines. Therefore, as a next step, parallel lines close to each other are merged into a single line. As long as parallel lines are within a certain distance d from each other, they are merged into one line. For horizontal lines, the resulting line will have the average y value of all those lines and the minimum and maximum x values as starting and end points.

From the conceptional view used in this thesis, the ends of lines always touch another line or the edge of the painting. However, the remaining lines now might overlap slightly or not even connect to the next line.

Now the two ends of every horizontal line $h \in H$ are considered. For each end $e = (x_e, y_e)$ find the closest vertical line $v \in V$ that could touch e by only translating it horizontally.

Since a rectangle is defined by the black lines in the paintings, as well as the edges of the painting, the lines of the edges to the H and V are added as well.

All lines should now represent the structure of the painting. The desired output, however, is a list of rectangles. Every rectangle in the image can be defined through a set of four different corners: top-left, top-right, bottom-left and bottom-right. These corners are always intersections of two lines, either fully crossing or touching (T-

crossing). The corners and their types can be identified.

Next, four different corners are combined into a rectangle by finding matching corners that belong to one rectangle. This is done by iterating through the top-left corners (x, y) and finding the closest top-right corner to the right (x_r, y) and the closest bottom-left corner below (x, y_b). The rectangle is then defined by the position of the top-left corner and its width and height $(x, y, x_r - x, y_b - y)$.

The colors are determined for each rectangle in the list. For this purpose, the rectangle is clipped from the original image I_1 and the average color of the selection is calculated. This color is then reduced to either black, white, red, blue or yellow.

Listing 1: Flow and structure of the program

```
input: directory d,
    threshold t,
    erosion kernel size N,
    dilation kernel size M,
    minimum line length l,
    maximum line width d,

for every file f in d,
  read original image from file f
  binary = preprocessing(original)
  output = detect_rectangles(binary, original)
  check_image = create image from output and overlay it on
     original
  write check_image to disk
  write output to json file

function preprocessing
  input: RGB image of a Mondrian painting image_1
  output: Binary image where only the lines are black

  image_2 = blur(image_1)
  r,g,b = split_channels(image_2)
  grey_1 = max(r,g,b)
  grey_min = min(r,g,b)
  grey_2 = clahe(g1)
  grey_c = grey_1 - grey_min
  grey_3 = grey_2 + grey_c
  binary_1 = threshold(grey_3, t)
  binary_2 = erode(binary_1, N)
  binary_m = dilate(binary_1, M)
  binary_3 = binary_2 + bitwise_not(binary_m)
  return binary_3

function detect_rectangles
  input:
    Binary image where only the lines are black binary_image,
    Original RGB image of that painting original_img
  output: List of rectangles and their colors, dimensions, and
     positions

  (h1, v1) = detect_lines(binary_image, l)
  (h2, v2) = reduce_lines(h1, v1, d)
  (h3, v3) = remove_lines_close_to_border(h2, v2)
  (h4, v4) = add_border_lines(h3, v3)
  (h5, v5) = connect_lines(h4, v4)
  (tl, bl, br, tr) = find_corners(h5, v5)
  rects = find_rectangles(tl, bl, tr))
  rects_with_color = find_colors(rects, original_img)
  return rects_with_color
```

3 Implementation

This section describes the implementation and development of the concept described above. First, the structure of the software is presented (3.1), then challenges in line detection (3.2) and performance (3.3) are discussed. Also the iterative development approach (3.4) and the related selection of the parameters (3.5) is explained. Lastly, the used tools are evaluated (3.6).

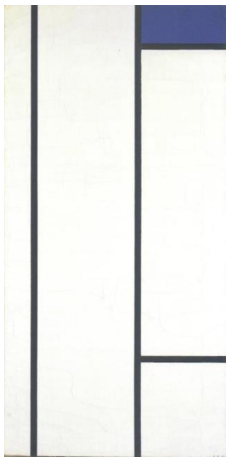
The program was implemented with a command line interface in *Python 3.7* using the *OpenCV 3.4.1* and *NumPy 1.15.2* libraries. OpenCV provides the different image processing methods described in 2.2 as well as other methods for reading, writing and manipulating image files. NumPy, which is part of the SciPy package, is a package for scientific numerical computing [9].

3.1 Structure

The structure of the main algorithm could be closely modeled after the concept. The separation between preprocessing (2.3.1) and detection of the rectangles (2.3.2) is also reasonable in the implementation.

The program sketched out in Listing 1 takes a directory and the different parameters as input. The output of the main algorithm is a list of rectangles and their respective dimensions, colors, and positions. An image is created from the resulting data and overlaid over the original image and saved to disk. This can be used to validate the result of the algorithm. The data from the rectangles is encoded in a *JSON* file for each image. *JSON* was chosen, since it is a popular flexible data-interchange format that is also human-readable. An example for such a *JSON* file can be seen in Figure 4.

As an example, Listing 2 shows the source code of the function `find_rectangles`, a subroutine of `detect_rectangles`. It detects rectangles from a list of top-left, bottom-left and top-right positions. The length of each of these lists is the same and is expected to be the number of rectangles. The program first sorts the top-right corners by their x position and the bottom-left corners by their y position. When iterating



```
{
  "id": "B244",
  "height":1000,
  "width":489,
  "rectangles":[
    {"x":294,"y":91,"w":195,"h":691,"c":"white"},
    {"x":294,"y":782,"w":195,"h":218,"c":"white"},
    {"x":64,"y":0,"w":230,"h":1000,"c":"white"},
    {"x":294,"y":0,"w":195,"h":91,"c":"blue"},
    {"x":0,"y":0,"w":64,"h":1000,"c":"white"}
  ]
}
```

Figure 4: Source image and *JSON* representation for *Composition no. I (1934)*

through the top-left corners, their x position are going to be the same as the matching bottom-left corner, apart from a larger y-coordinate. So only these candidates are selected from the `b_l` list using an iterator expression. Since the corners are ordered from top to bottom, the matching corner is going to be the first returned by the iterator. Therefore the iterator is only called once with `next`. The same logic is applied to bottom-left corners. Using the coordinates of the matching top-right and bottom-left corners, the rectangle is calculated and added to a list of rectangles as a tuple. That list is then returned by the function.

Listing 2: Function for constructing rectangles from corners

```
def find_rectangles(t_l, b_l, t_r):
    t_r.sort(key=lambda pos: pos[0])
    b_l.sort(key=lambda pos: pos[1])
    rectangles = []
    for x,y in t_l:
        x2,_ = next(c for c in t_r if c[1] == y and c[0] > x)
        _,y2 = next(c for c in b_l if c[0] == x and c[1] > y)
        w = x2 - x
        h = y2 - y
        rectangles.append((x,y,w,h))
    return rectangles
```

3.2 Line Detection

Using a custom algorithm for the line detection was not the first choice. First, a common technique, the Hough Transform, was used. Hough Transforms rotate a line over every pixel of an image with an angle θ for every step. For each rotation black pixels on that lines are used as votes for that line. Then lines above a certain threshold of votes are considered. Using $\theta = \frac{\pi}{2}$ to only consider horizontal and vertical lines, this method did not give accurate enough results. The algorithm would, for example, sometimes detect a vertical line between two close horizontal ones. Since gaps between lines are not expected after the preprocessing, this method is not suitable in this context.

It became clear that it would be simpler and more accurate to construct the lines by iterating through the image. To find horizontal lines H from 2.3.2, the image is scanned vertically line by line. The iteration runs through the pixels of each horizontal line. Uninterrupted sequences of black pixels are added to a list of horizontal lines. Only sequences with a specified minimum length ℓ are added. The minimum length should be slightly larger than the maximum width of the lines in the image. Vertical lines V are recognized respectively by iterating through the lines from left to right.

3.3 Performance

The performance of the algorithm was measured by timing the duration for each image. The first implementation of the algorithm took about 1100 milliseconds per iteration. By running timings on different parts of the processing, the `detect_lines` function was discovered as a bottleneck. It was accountable for about 93 percent of

the processing time.

This was the only place in the program were an iteration through all of the pixels in the image took place using Python. All other operations were done using OpenCV or NumPy. They also iterate through all of the pixels but using optimized C code.

To reduce the time of this step this function was reimplemented in a compiled, more low-level language. Rust was chosen using the *rust-cpython* library for bindings. Using the same algorithm implemented in Rust, the time of this step was reduced from about 1020 to 40 milliseconds. The runtime of the program could be reduced by a factor of 13.

This time could have been further reduced by running multiple parallel iterations over the different rows or columns. However, for this purpose the achieved time was sufficient.

3.4 Iterative approach

When developing the program, the different processing steps of the algorithm were visualized by providing multiple output images for each input image. The output images from the preprocessing phase correspond to the images shown in Figure 3. Example details of the same image from the second phase can be seen in Figure 5. These images allow for an inspection of the different intermediary results created in the process. Together with the superimposed output image, these could be used to review the correctness of the algorithm.

This visual feedback gave insights into problems of the algorithm like small bugs or conceptional problems. For example, the preprocessing step to increase the brightness of areas that are more colorful was introduced because there were issues of discriminating darker blue areas from black ones. This way the development of the algorithm could follow an iterative approach.

These images also provided a basis for the adjustment of the parameters of the different steps.

(a) `detect_lines` (b) `reduce_lines` (c) `find_corners` (d) `find_colors`

Figure 5: Details of example output images with the corresponding function from Listing 1 after which they were created.

3.5 Parameter Selection

By running the program on the images and looking at the output images for the different steps, the reasonable values for the parameters of the program can be determined.

Through comparing the Thresholding output for each image to the input image, the threshold value $t = 110$ could be determined to give reasonable results.

To better fine-tune the parameters, a subset of input images was selected and the parameters manually changed until the result was correct. The resulting *JSON* files with the correct results were moved to another directory *detected*. Then a step was added to the program that would always compare the computed result with the data in that directory if available. When the result differed, it would print out a warning.

Using this output the minimum length ℓ, the maximum width d and the size of the erosion kernel N as well as the dilation kernel M were changed to maximize the number of recognized images from that set. The resulting parameters were $l = 60$, $d = 70$, $N = 11$ and $M = 40$.

3.6 Evaluation of Tools

Reflecting about the used tools, OpenCV would certainly be chosen again. It supports a variety of established computer vision algorithms, is well documented and has a variety of learning resources. Since OpenCV primarily supports C++ and Python interfaces, the programming language choice was one of these options. While using the compiled language C++ could have increased performance over the interpreted Python, it would also have lacked some of its benefits: Python's readability and user-friendly data-structures made it a good choice for an iterative development.

4 Results

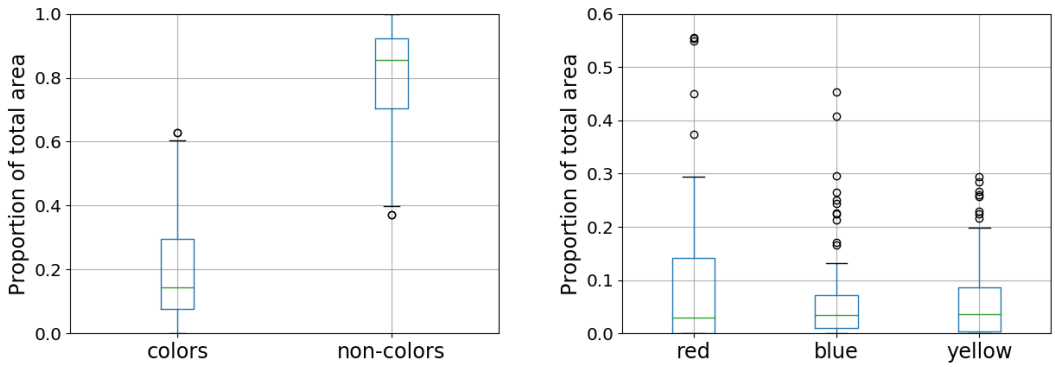
All 118 images were processed by the detection program and subsequently checked for accuracy by comparing the result with the original. The resulting dataset consists of 1316 rectangles and their respective size, colors, and paintings.

Some descriptive analysis about the use of color (4.1) and ratios (4.2) was performed.

4.1 Colors

On average, neglecting the area used by lines, Mondrian paintings consist of 79.7% non-colors (black, white) and 20.3% colors (red, blue, yellow) ($\sigma = 17.0\%$), see Figure 6a. While the median percentage of each color appears to be rather similar, the distribution of red is more widespread as can be seen in Figure 6b.

To see if different colors of rectangles have overall preferred positions in the compositions, the dataset of all rectangles and their respective positions, width and colors is used. The calculated centers of all of these rectangles are plotted by color. The visualisation in Figure 7 shows the estimated probability density function of the positions using *Kernel density estimation (KDE)* with an Gaussian kernel selected by Scott's method [19]. The first notable characteristic is that the color red appears to have a



(a) (b)

Figure 6: The preference for non-colors in the paintings is evident in (a). The Median proportion of the colors in (b) are very similar, but red has a wider distribution.

substantial bias to the top-left corner. Similarly blue has a bias to the bottom-right corner but is comparatively more evenly spread.

4.2 Ratios

For all rectangles, the aspect ratio of the longer to the shorter side was calculated. Figure 8a shows the estimated probability density of the ratios in Mondrian's rectangles compared to the ratio of a set of 10000 random rectangles. Figure 8b shows all rectangles plotted by their sides, as well as lines for certain proposed ratios. The data does not show peaks for the golden or silver ratio. This supports the rejection by Livio (2002) [11] and Markowsky (1992) [13] of the golden rectangle claim by Bouleau (1963) [4] and Bergamini (1980) [3].

5 Discussion

The developed program is able to detect rectangles in a class of abstract Mondrian paintings. Human control and intervention are still necessary to make sure the result is correct. In 23 cases the input image was manually edited to trace lines that were not detected. Otherwise, all images were detected successfully by adjusting the input parameters. From the dataset of all chosen images, 77% of the images could be detected without changing the default parameters.

A first look at the resulting data showed some interesting findings that deserves further investigation. The dataset that was obtained from the images will be made freely available for other applications, from statistical analysis to machine learning.

The data does not suggest a preferred use of the golden or silver ratio in rectangles. There only seems to be a general tendency towards squares. Justifying continued research into this direction should depend on findings in the psychology of aesthetics. So far the studies about the visually pleasing nature of golden rectangles appear to contradict each other. Perceptional studies about the silver ratios are missing entirely.

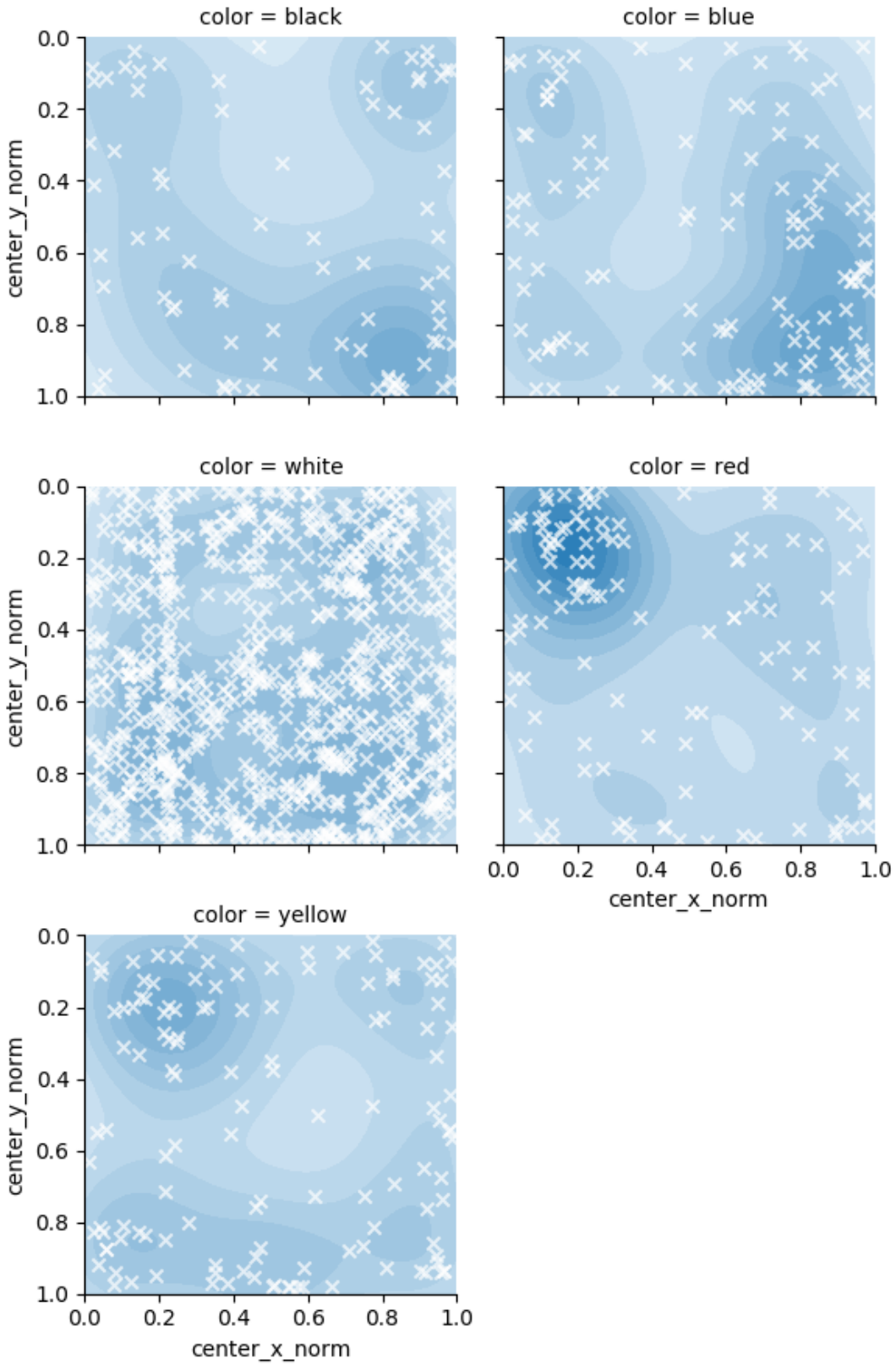
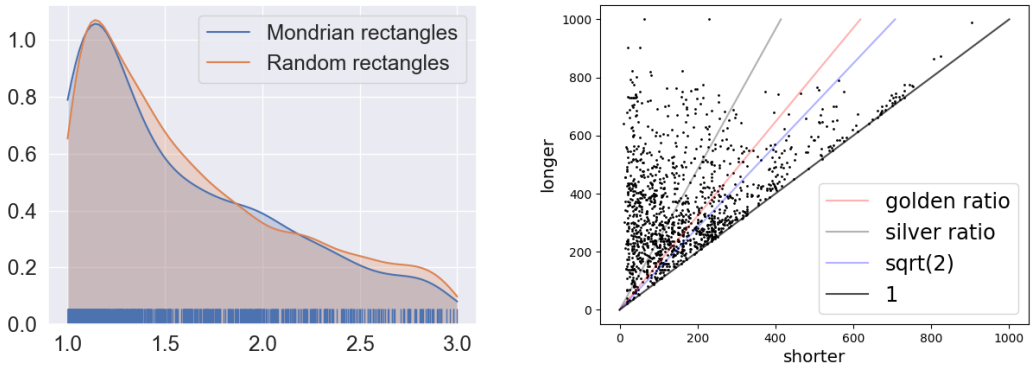


Figure 7: Scatter plots and KDE for each color



(a) (b)

Figure 8: There appears to be no specific ratio used preferentially by Mondrian.

However, research in ratios of other components, like the positioning of lines or the size of rectangles to each other might lead to interesting results.

In "Art and Visual Perception" Rudolf Arnheim [2] notes that "an object in the upper part of the composition is heavier than none in the lower; and location at the right side makes more weight than location on the left." Winner (1987) [21] psychologically tested these principles by showing abstract images and their horizontally or vertically flipped counterparts. Their findings support the up-down principle, but not the left-right principle.

Arnheim also notes that the color red is heavier than the color blue. A study from [12] found some support for this claim when showing design-trained participants modified Mondrian paintings where some colors were swapped.

It would be interesting to explore if the found preference by Mondrian to use red in the top-left corner could be explained by a combination of these proposed principles of balance or whether it is merely coincidental or habitual.

Bibliography

[1] Alberto Aguado. *Feature Extraction and Image Processing for Computer Vision.* Academic Press, 2012.

[2] Rudolf Arnheim. *Art and Visual Perception: A Psychology of the Creative Eye*. Univ. of Calif. Press, 1965.

[3] D. Bergamini and Time-Life Books. *Mathematics*. Life science library. Time-Life Books, 1980.

[4] C. Bouleau. *The Painter's Secret Geometry: A Study of Composition in Art*. 1963.

[5] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[6] Susanne Deicher. Piet Mondrian: Protestantismus und Modernität, 1995.

[7] Loe Feijs. Divisions of the Plane by Computer: Another Way of Looking at Mondrian's Nonfigurative Compositions. 37(3):217–222, 2004.

[8] Anthony Hill. Art and Mathesis: Mondrians Structures. *Leonardo*, 1(3):233–242, 1968.

[9] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[10] Joop M Joosten, Piet Mondrian, and Robert P Welsh. *Catalogue Raisonné of the Work of 1911-1944*. Prestel, 1998.

[11] Mario Livio. *The Golden Ratio*. 2002.

[12] Paul Locher, Kees Overbeeke, and Pieter Jan Stappers. Spatial balance of color triads in the abstract art of Piet Mondrian. *Perception*, 34(2):169–189, 2005.

[13] George Markowsky. Misconceptions about the Golden Ratio. *The College Mathematics Journal*, 23(1):2, 1992.

[14] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, and Karel Zuiderveld. Adaptive Histogram Equalization and Its Variations. *Computer vision, graphics, and image processing*, 39(3):355–368, 1987.

[15] Linda G Shapiro. *Computer vision*. Prentice Hall, Upper Saddle River, NJ, 2001.

[16] Martin Skrodzki and Konrad Polthier. Mondrian Revisited: A Peek into the Third Dimension. In *Bridges 2018*, pages 99–106, 2018.

[17] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, CA, USA, 1997.

[18] Sayaka Tanaka and Michiyo Miyanaga. A Study on the Composition on the Works of Piet Mondrian during his days in London and New York. Poster presented at Asian Forum of Graphic Sciences (AFGS), 2017.

[19] George Terrell and David Scott. Variable Kernel Density Estimation. 1992.

[20] Wikisource. Manifest i of the style 1918 — wikisource,, 2015. [Online; accessed 18-September-2018].

[21] Ellen Winner, Jacqueline Dion, Elizabeth Rosenblatt, and Howard Gardner. Do Lateral or Vertical Reversals Affect Balance in Paintings? *Visual Arts Research*, 13(2):1–9, 1987.