# Perspective Check in Paintings

*M1 Internship*

Yoann Coudert–Osmont

*Supervised by*

Elmar Eisemann  Ricardo Marroqium

May - July 2019

## Introduction

In a Delft museum, paintings by Pieter de Hooch are on display. It is quite easy to notice that some pairs of paintings are very similar (e.g. Figure1). One hypothesis that would explain this strangeness is the possibility that another painter copied Pieter de Hooch's paintings. Looking at the pairs in more detail, we can notice that quite often one painting respects the rules of perspective well and the other does not respect them. This gives credibility to the hypothesis of the existence of another painter. But humans tend to be biased to check that in each pair, one painting respects the rules of perspective and the other does not. Then comes the need

Figure 1: A pair of similar paintings

to create a tool that allow us to check whether the perspective is respected in a painting with as little human intervention as possible. The goal of this internship was therefore to create a graphical interface to verify the perspective in a painting with as much automation as possible. Tools already exist to find lines in an image. The most common and the one I used is the Hough transform [DH72]. I will describe the algorithms used to create my interface in this report. In most paintings there is a tiled floor and it is mainly the lines of the tiles that I studied. The development of a graphical user interface with minimal user control was necessary because there is no guarantee that a fully automated program will do exactly what we want. Part of the internship was used to learn how to make a graphical interface with Qt and to design an interface that is easily usable. But I'm not going to describe how this interface works, I'll just give some implemented features inside this report.

# Contents

# 1 Reminders on the rules of perspective

The main rule of perspective is that when you make a drawing, the parallel lines must all cross at the same point. We call this point a vanishing point. In addition, all vanishing points must be on the same line. This line is the vanishing line and corresponds to the horizon.

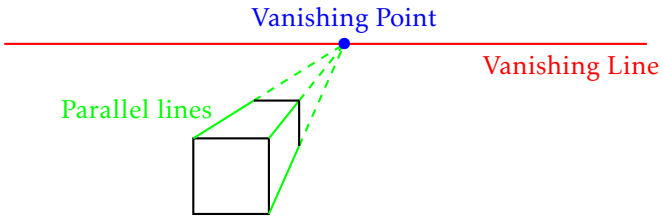



Figure 2: Illustration of the perspective rules

## 1.1 Special case of a tiled floor

This subsection will be useful for the last section of this report. There are two interesting equations to record about tiled floors linking tile lines with their diagonals. Indeed I will try to build a new painting that perfectly respects the perspective. But to do this, it is necessary to know the properties that the vanishing points of a tiled floor must respect.

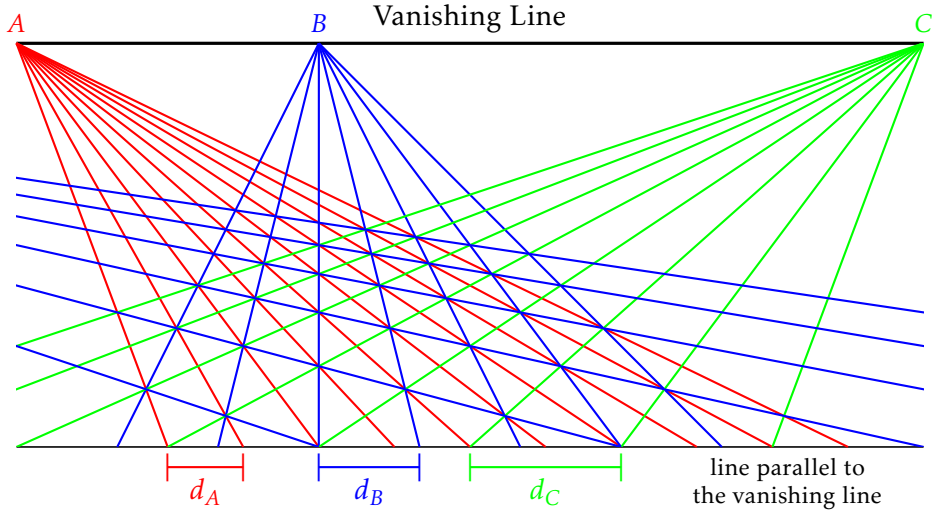



Figure 3: Tiled floor in perspective

In the Figure3, blue lines represent a tiled floor in perspective. We will always assume that tiles are squares. Then all the parallel lines are equally spaced. In perspective, this implies that if a line parallel to the vanishing line is drawn, then the tile lines intersect with it at regular intervals. We write these intervals $d_B$ for the depth lines, and $d_A$ and $d_C$ for the diagonals. $A$, $B$ and $C$ are the vanishing points located on the vanishing line (note that here the vanishing line is horizontal but it may not be). Finally here are the two equations respected by such a tiled floor:

$$B = \frac{d_C A + d_A C}{d_A + d_C} \qquad (1) \qquad\qquad d_B = 2\frac{d_A d_C}{d_A + d_C} \qquad (2)$$

# 2 Description of the algorithm

## 2.1 General description

Finding lines in an image can be broken down into several steps. For my part, I divided this task into four main steps. First I change the color space to switch from RGB space to CIE Lab space [CF97]. Then I eliminate the noise and smooth the image while keeping the areas with high gradient. In the third step I compute in each pixel of the image the direction and amplitude of the gradient. I leave some user control over the value of some thresholds used in this step. In addition, the user has the possibility to erase certain parts of the gradient that are not interesting to study perspective. Finally, the last step consists in applying the Hough transform to the gradient. The local maxima of this transformation then give us the lines contained in the image.

Once the user has obtained the lines of the image he has the possibility to make groups in order to calculate the vanishing points, the vanishing line but also to calculate the diagonals of a tiled floor as we could see in the previous part. Once the user has finished this part he can judge if the perspective is well respected.

## 2.2 Color Space

At the beginning of my experiments I did not make any color space changes. The gradient of the image was calculated in the RGB space well known to all. In this space the colors are represented by three bytes, representing respectively the intensity of red, green and blue in the color. Unfortunately for some paintings, the amplitude of the gradient obtained did not necessarily match what I observed with the naked eye. My algorithm could return a high gradient where I could only see a slight color change and return a low gradient where I could see a boundary between two distinct colors. One of the concerns of RGB space, for example, is that it does not take into account the fact that the human eye is less sensitive to the color blue than to the color green.

The CIELAB space [CF97] allows us to correct this problem. Indeed, the Euclidean distance in this space corresponds well to the difference in color that a human observes with his eyes. It is a space more in line with human vision. The three coordinates of CIELAB represent the lightness of the color ($L^* = 0$ yields black and $L^* = 100$ indicates diffuse white), its position between red/magenta and green ($a^*$, negative values indicate green while positive values indicate magenta) and its position between yellow and blue ($b^*$, negative values indicate blue and positive values indicate yellow). Thus the difference between two colors $(L_1^*, a_1^*, b_1^*)$ and $(L_2^*, a_2^*, b_2^*)$ is given by the Euclidean distance:

$$\Delta E = \sqrt{(L_1^* - L_2^*)^2 + (a_1^* - a_2^*)^2 + (b_1^* - b_2^*)^2}$$

I will not give the formulas for switching from RGB to CIELAB, but I can give the intermediate steps. There is first a non-linear transformation from RBG to sRGB then a linear transformation to the XYZ space and finally a non-linear transformation from XYZ to CIELAB. Once this change of space was implemented, the results on some paints improved considerably.

## 2.3 Smoothing

Smoothing is often important in image processing because it allows noise to disappear. There are many filters known to perform smoothing. There is for example the median filter which replaces each pixel by the median value on a neighborhood, or the very used Gaussian filter which has the particularity to be applied with a low complexity thanks to a Fourier transform. The Gaussian filter performs a weighted average of the neighboring pixels. This weighting is done using a Gaussian with a standard deviation $\sigma$. Averages are taken individually on each component of the color space. If we consider an image $I : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ which associates real values to integer coordinates, then applying a Gaussian filter of standard deviation $\sigma$ gives the following image $I'$:

$$I'(z) = \frac{1}{W} \sum_{z' \in \Omega_z} I(z') \exp\left(-\frac{\|z - z'\|^2}{2\sigma^2}\right)$$

Where $\Omega_z$ is a window around $z \in \mathbb{N}^2$ and $W = \sum_{z' \in \Omega_z} \exp\left(-\frac{\|z-z'\|^2}{2\sigma^2}\right)$ is a factor of normalization. Since it is a convolution product, this can be computed in the Fourier space with a point-wise product. That's the big advantage of this filter. The problem is that it could erase borders that would be useful for finding the lines in the image.

One solution to overcome these important edge disappearances is to use a bilateral filter [TM98]. This filter is defined using two functions $f : \mathbb{N}^2 \times \mathbb{N}^2 \to \mathbb{R}$ and $g : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and gives the following image:

$$I'(z) = \frac{1}{W} \sum_{z' \in \Omega_z} I(z) f(z, z') g(I(z), I(z'))$$

Where $\Omega_z$ is still a window around $z$ and $W = \sum_{z' \in \Omega_z} f(z, z') g(I(z), I(z'))$. Thanks to the $g$ function, not only the position of the other pixels is taken into account, but also the value of these pixels. The idea we can then have is to use this function $g$ to ignore pixels that have a too different color in order not to smooth the borders between two different colors. Taking two Gaussian functions for f and g is then common practice and that is the choice I made:

$$f(z, z') = \exp\left(-\frac{\|z - z'\|^2}{2\sigma_f^2}\right) \qquad g(u, u') = \exp\left(-\frac{\|u - u'\|^2}{2\sigma_g^2}\right)$$

The two standard deviations ($\sigma_f$ and $\sigma_g$) and the size of the neighborhood are automatically calculated according to certain image characteristics.

Original image Gaussian filter Bilateral filter

Figure 4: Comparison of smoothing filters

It can be seen in the Figure4 that the bilateral filter keeps the edges clean between the tiles and that the noise inside the tiles is erased. This filter therefore does exactly what we want it to do, unlike the Gaussian filter which smooths the edges between the tiles and therefore makes it difficult to accurately locate the edges. On the other hand because of the function $g$, the filter is not a convolution product and it is therefore no longer possible to go through the Fourier space in order to calculate the results in a fast way. That's the only downside of this filter. To compensate for this calculation time which can be long and even to simplify the steps that will follow, I leave the user the possibility to select with a lasso the interesting area to study in the painting (Figure5).

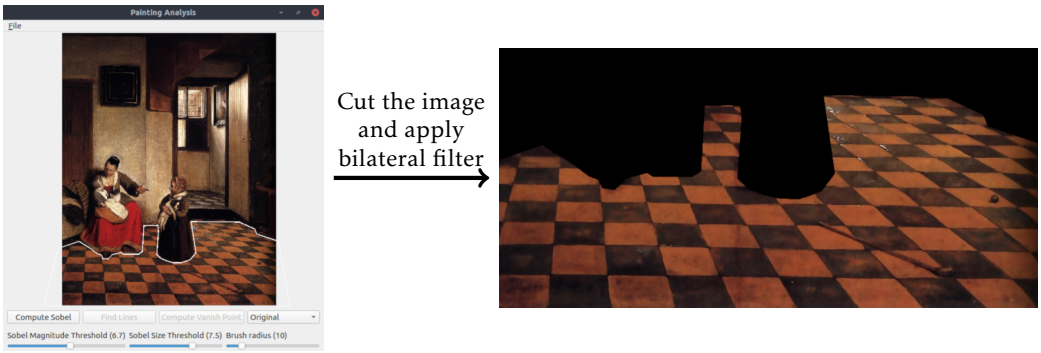



Figure 5: The user has the possibility to select the interesting zone.

## 2.4 Gradient

To compute the gradient of the image, I used a Sobel filter of size 3 by 3. This filter calculates the amplitude of the gradient along the x-axis and the y-axis and then deduces the gradient norm and direction from it. This is done for each component $C$ of the CIELAB space. I denote by $*$ the convolution product. We then have:

$$G_{C,x} = \begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix} * C \qquad G_{C,y} = \begin{bmatrix} -3 & -10 & -3 \\ 0 & 0 & 0 \\ 3 & 10 & 3 \end{bmatrix} * C$$

Where $G_{C,x}$ and $G_{C,y}$ are are respectively the discrete horizontal and vertical gradients of the component $C$. We can then compute the norm of the gradient in this component:

$$G_C = \sqrt{G_{C,x}^2 + G_{C,y}^2}$$

As only the direction of the gradient and not the orientation of the gradient interests us, we want to compute the angle of the gradient modulo $\pi$ instead of $2\pi$. We then defines $(G'_{C,x}, G'_{C,y})$ equal to $(G_{C,x}, G_{C,y})$ if $G_{C,x} \geqslant 0$ or to $(-G_{C,x}, -G_{C,y})$ otherwise. Finally, the norm $G$ and the angle $\Theta$ of the gradient are obtained as follows:

$$G = \sqrt{G_{L^*} + G_{a^*} + G_{b^*}} \qquad \Theta = \arg\left(G_{L^*}G'_{L^*,x} + G_{a^*}G'_{a^*,x} + G_{b^*}G'_{b^*,x} + i\left(G_{L^*}G'_{L^*,y} + G_{a^*}G'_{a^*,y} + G_{b^*}G'_{b^*,y}\right)\right)$$

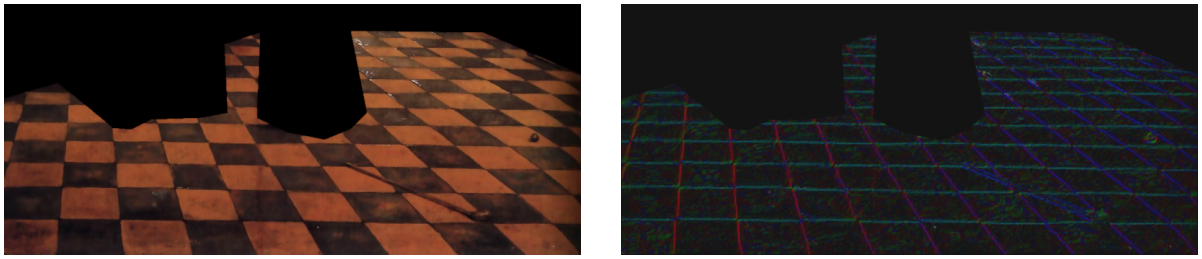


Figure 6: Result of the Sobel filter

In the Figure6 we can see the result obtained on the same painting used in the previous part about smoothing (I will use this painting throughout the explanation of the algorithm). The image on the left is the smoothed paint from the previous part and the one on the right is the image representing the gradient. For each pixel, the intensity of the color represents the norm of the gradient. In the image I give, the intensity is proportional to the square root of the norm. This choice was made so that it could be seen that despite the smoothing done previously, the gradient norm is not zero inside the tiles. The color gives the angle of the gradient. Red for $0 + \mathbb{Z}\pi$, green for $\pi/3 + \mathbb{Z}\pi$ and blue for $2\pi/3 + \mathbb{Z}\pi$.

**Automatic cleaning** To eliminate the remaining noise and make the rest easier, I clean the image a little bit. To do so, I apply a threshold that will zero all pixels in the gradient whose intensity is less than a certain value. Then I compute the connected components of the remaining pixels and zero all pixels of the connected components that have a size smaller than a minimum size. I have a function that calculates the intensity threshold and minimum size of the related components based on the characteristics of the image. Unfortunately, these two values are not always perfect. I therefore allow the user to modify these values if necessary using sliders. The effect of this cleaning is illustrated in Figure7.

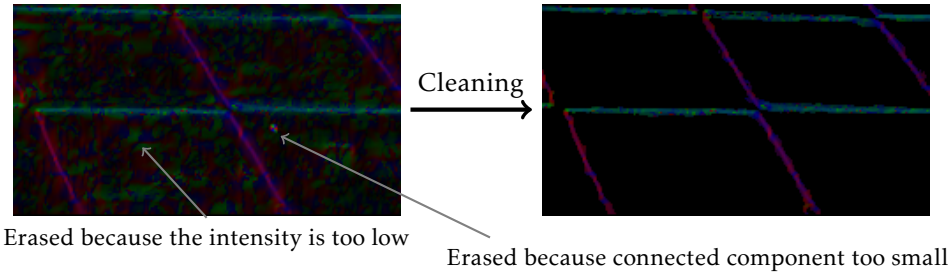



Figure 7: Cleaning the gradient image

**Smoothing** It can be noted that the direction of the gradient is not perfect. Some isolated pixels, or even groups of pixels, do not have a good direction. However, in the following I will use this direction to find the lines. That's why I smooth the direction in the gradient image. The Algorithm 1 at the top of the next page is the function that gives the new smoothed direction of a pixel of the gradient.

**Algorithm 1** Gradient Smoothing

$$M \leftarrow \begin{bmatrix} 0.0925 & 0.12 & 0.0925 \\ 0.12 & 0.15 & 0.12 \\ 0.0925 & 0.12 & 0.0925 \end{bmatrix}$$

**function** SmoothGrad($G, \Theta, x, y$)
  $a, b \leftarrow 0, 0$
  **for** $i, j \in \{-1, 0, 1\}^2$ **do**
    $a \leftarrow a + M[j+1][i+1] \times G[x+i][y+j] \times \cos(2 \times \Theta[x+i][y+j])$
    $b \leftarrow b + M[j+1][i+1] \times G[x+i][y+j] \times \sin(2 \times \Theta[x+i][y+j])$
  **end for**
  $\Theta[x][y] \leftarrow \arg(a + ib) / 2$
**end function**

The function computes a weighted average of the gradient angle $\Theta$ in a neighborhood of size $3 \times 3$ around the pixel $(x, y)$. The weight depends on the distance (thanks to the $M$ matrix) and the amplitude of the gradient ($G$). In addition, the factor 2 that appears in the cosine and sinus comes from the fact that we consider angles modulo $\pi$. Finally, because a single step of this smoothing is not enough, this algorithm is called several times. For example, for the paint that we use as an example, we apply 5 times the smoothing. For higher resolution images (because this one is low resolution), I can apply up to 15 smoothing steps for the gradient. The Figure8 shows the result of the previous algorithm.

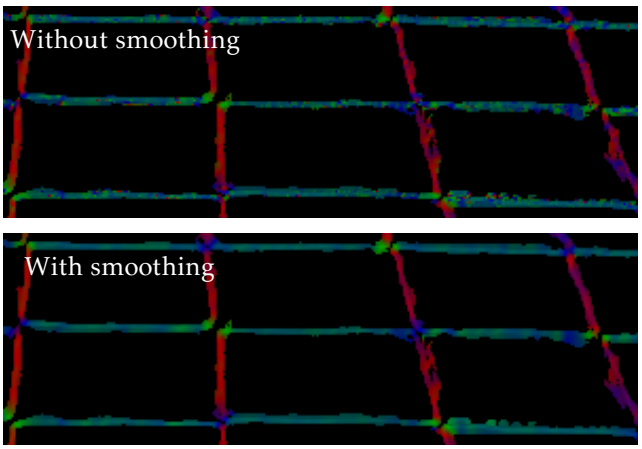



Figure 8: Smoothing gradient result

**Manual cleaning** Because there may be objects on the floor of the paint or sometimes because the noise is too high (or because we can see some damages on the paints of high quality) there may be undesirable parts left in the gradient image. That's why I'm leaving user control again. There is the possibility of using a brush to remove unwanted parts. The user can choose the size of the brush and erase bad brush strokes that he could have given. The Figure9 illustrates this tool. White pixels are the erased gradient pixels.

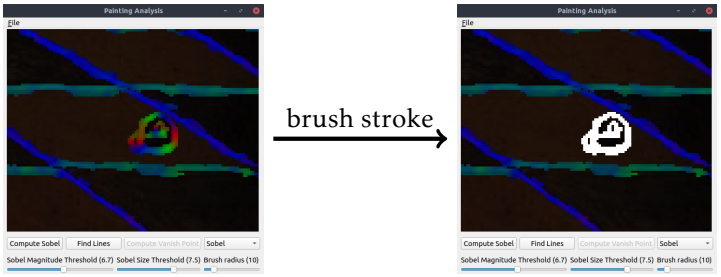



Figure 9: Usage of the brush

## 2.5 Hough Transform

**A first try** Before using Hough's transform I tried something else that I will describe in a few lines in this paragraph. We are trying to find the main lines that make up the image. My idea was to compute connected components on the pixels of the gradient. I considered two pixels, adjacent if they touched each other and if their gradient angle difference was less than a certain value. Then I performed a PCA (Principal Component Analysis) [Shl14] on each connected component in order to obtain equations of lines corresponding to the main direction of each component. Then I merged all the connected components for which the guide lines were close and re-performed a PCA again to obtain new, more precise line equations.

But the big problem is that the principal component analysis was not very good on the small connected components and some lines were mainly formed by a succession of small connected components especially in paintings with very small tiles. Then there is a butterfly effect, some components merge when they should not, and others do not merge when they should. The results were therefore not very good and I implemented the Hough transform [DH72] which gave much better results.

**Parameterization** We often represent a line with two parameters $a$ and $b$, by the equation:

$$y = ax + b$$

But this representation has some drawbacks. The parameters are not bounded if we consider all the lines passing through a rectangle. Fortunately, Duda and Hart have designed another representation that allows us to have bounded parameters $\rho$ and $\theta$:

$$\rho = x\cos(\theta) + y\sin(\theta)$$

Thus $\rho$ represents the distance between the line and the origin except for the sign and $\theta$ represents the angle of the line normal. Let's call $D$ the length of the diagonal of the image. In the literature we generally take $\theta$ in $[0;\pi[$ and $\rho$ in $[-D;D]$. For my part, I preferred to take $\theta$ in $[-\pi/2;\pi]$ and $\rho$ in $[0;D]$. These domains cover the entire rectangle of the image, *i.e.* $[0;W)\times[0;H)$ where $W$ and $H$ are the width and the height of the image.

**Algorithm** The idea of the algorithm is to build a new image $H^*$, of size $R \times T$ where $R$ and $T$ are fixed using the dimensions of the original image. Then for each pixel of position $(x,y)$ of the gradient with a non-zero intensity, and for each $t \in [0;T)$ we compute the distance at the origin, $\rho$ of the line of direction $\theta = 3\pi/2 \times t/T - \pi/2$ passing by $(x,y)$. We then let $r = R \times \rho/D$ and we add a positive value to the pixel of coordinates $(r,t)$ in the image $H^*$ (Algorithm 2).

**Algorithm** 2 Hough Transform

**function** HOUGH($G, \Theta$)
  $H^* \leftarrow$ Null image of size $R \times T$
  **for** $(x, y) \in [0, W) \times [0; H)$ with $G[x][y] > 0$ **do**
    **for** $t \in [0; T)$ **do**
      $\theta \leftarrow 3\pi/2 \times t/T - \pi/2$
      $\rho \leftarrow x.\cos(\theta) + y.\sin(\theta)$
      **if** $\rho \geqslant 0$ **then**
        $r \leftarrow \rho \times R/D$
        $H^*[r][t] \leftarrow H[r][t] + f(G, \Theta, \theta, x, y)$
      **end if**
    **end for**
  **end for**
  **return** $H^*$
**end function**

When Hough transform is computed, lines are local maximums of $H^*$. To obtain these local maximums, I sort all the pixels of the transform in descending order and then take the pixels one by one and add them to a list $L$ of lines if and only if the line corresponding to the pixel in question is far enough away from all the other lines already contained in the list $L$.

The value $H^*[r][t]$ then corresponds approximately to the number of pixels of the gradient on the line parameterized by $r$ and $t$. I say approximately because of the function $f$. This function allows us to take into account the intensity of the gradient and the difference between the angle of the line and the angle of the gradient. I have designed the following function:

$$f(G, \Theta, \theta, x, y) = \left(1 - \left|\sin\left(\theta - \Theta[x][y]\right)\right|^{\frac{2}{3}}\right) \times \left(1 + \frac{3}{2}\frac{G[x][y]}{\max(G)}\right)$$

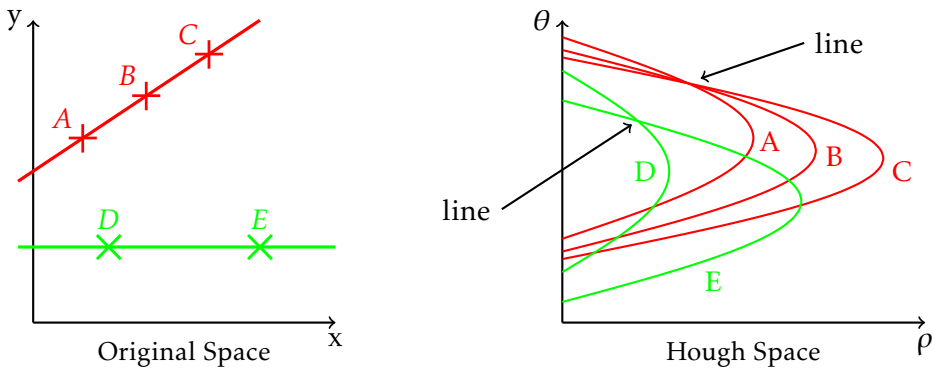



Figure 10: Hough Transform Example

The Figure10 shows the curves where the Hough transform is strictly positive for five points sampled on two lines. The local maximums are located where the curves intersect. The two intersection points observed therefore correspond to the two lines that were sampled.

**Improvements** This algorithm can be improved in many ways to reduce the computation time but also to improve the final result. Here are the main improvements I have implemented:

- We don't have to go through the whole spectrum of $\theta$ but just limit ourselves to a smaller window around $\Theta[x][y]$. This limits the computation time and prevents some pixels from contributing to a line that does not correspond at all to the orientation of the gradient.
- After computing $\rho$ for a given pixel and direction $\theta$, instead of taking $r$ as a rounding of $\rho \times R/D$ we can take the two integers $r_1 = \lfloor \rho \times R/D \rfloor$ and $r_2 = \lceil \rho \times R/D \rceil$ and add a portion from $f(G, \Theta, \theta, x, y)$ to $H[r_1][t]$ and $H[r_2][t]$.
- The values $H[r][t]$ can be readjusted according to the distance that the line parameterized by $r$ and $t$ travels in the cut area of the image (see Figure5).
- Once you have found that the pixel $(r, t)$ of the Hough transform corresponds to a local maximum and therefore to a line, you can use the values of the Hough transform in a small neighbourhood of $(r, t)$ to readjust the parameters $\rho$ and $\theta$ of the corresponding line.


Figure 11: Hough Transform on painting

On the Figure11 we can see what the transform gives on the painting we have been following since the beginning. Only the 20 best lines are displayed in the interface after clicking on the button to find the lines. But the user has the possibility to delete the lines that do not interest him with the right click and to add the lines that the Hough transform found without them being displayed by right-clicking approximately where the desired line should be. The user also has the possibility to drag the mouse to indicate approximately the direction of the line if the position of the click is not really accurate. As mentioned in the subsection 2.1, the user can then group the lines together to compute the vanishing points (they are computed by minimizing the average square distances with the group lines) and also to compute the diagonals of the tiles. Diagonals are calculated by applying a PCA [Shl14] on the intersections of the horizontal and depth lines. Finally, it is possible to save the result obtained in SVG format. The Figure12 shows the result obtained for the painting we are following. It can then be seen that even if the vanishing line is not very horizontal, the perspective seems to be respected since the parallel lines all intersect well at the same point and all these vanishing points are approximately on the same line.

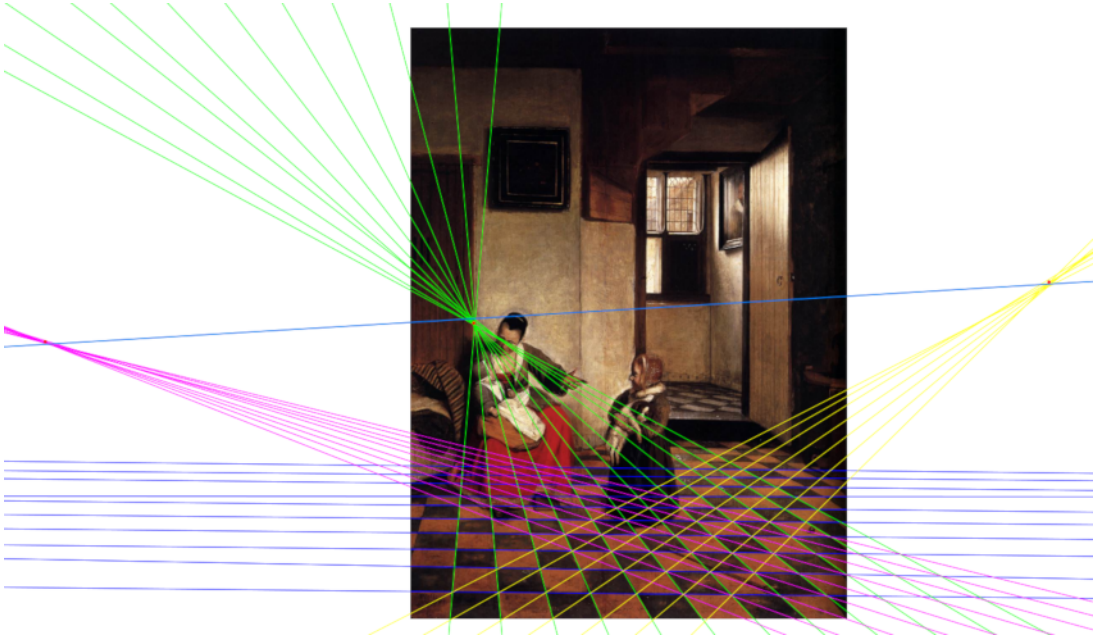


Figure 12: Final Result

Result on the painting that looks like this one is given in the appendices. I also give for some of them the difference obtained between the lines drawn by my algorithm and the lines drawn by Ricardo. Finally, we would also like to know how close or far away the painting is to a perfect perspective. Indeed, we would like to know if it is possible that the artist made small inaccuracies or if he did not completely take into account the rules of perspective. Moreover, my algorithm is not necessarily very precise and therefore it can be interesting to see that a perfect perspective can be very close to the painting. This is the purpose of the next section.

# 3 Attempt to redraw the paintings with a perfect perspective

Elmar suggested I use the paper *Structure preserving manipulation of photographs* [OBBT07] that allows us to modify an image by using its gradient. The idea is then to do the following steps starting from the lines obtained in the previous section:

- Compute new lines for the tiles that would all intersect exactly at a single vanishing point and remain as close as possible to the previously obtained lines.
- Modify the gradient of the image so that the high intensity pixels are on the new lines.
- Use Poisson editing [PGB03] to obtain the new image with perfect perspective from the new gradient.

Unfortunately, I didn't get anything conclusive for this second step. I will therefore only describe what I was able to do for the first step.

For the first step I parameterize the perspective with the coordinates of the points $A$, $B$ and $C$ and with the distances $d_A$, $d_B$ and $d_C$ (see Figure3) between the intersections of the paintings lines with a line parallel to the vanishing line and located at a distance $d_{trans}$ from it. Thanks to the equations (1) and (2) we can remove $B$ and $d_B$ from our parameterization. We then have 6 parameters to optimize:

$$x_A, y_A, x_C, y_C, d_A, d_C$$

I have broken down the function that these parameters should minimize into the sum of three sub-functions explained in the next paragraphs. In the following I will use the notation $c^{te}$ to designate a constant whose value won't be given in order not to overload the expression with uninteresting details. And I will also use $v$ to designate an expression that does not depend on the parameters to be minimized but that is dependent on the lines of the painting to which we are trying to be the closest.

In order to have a horizontal vanishing line and in order to have well oriented tiles, I try to minimize the following value:

$$R = \frac{(y_C - y_A)^2}{(x_C - x_A)^2 + (y_C - y_A)2} \times \left(v.(d_C - d_A)^2 + c^{te}\right) \tag{3}$$

The left-side of the multiplication is null when the vanishing line is horizontal and strictly positive otherwise. The right-side try to minimize the difference between $d_C$ and $d_A$ because if these two values are equal then tiles are correctly oriented (horizontal lines of the floor are parallel to the vanishing line).

So that the vanishing points do not diverge too much, I minimize also this expression:

$$S = v.(x_A - x_{A_0})^2 + v.(y_A - y_{A_0})^2 + v.(x_B - x_{B_0})^2 + v.(y_B - y_{B_0})^2 + v.(x_C - x_{C_0})^2 + v.(y_C - y_{C_0})^2 \tag{4}$$

Where $B$ and $d_B$ are still given by the equations (1) and (2). The points $A_0$, $B_0$ and $C_0$ are the vanishing points obtained in the previous section, the ones that are represented by red dots in the Figure12. This expression $S$ is simply the sum of the square distances of the new vanishing points to the old ones.

The last expression to minimize is the most complicated one. With the two previous expressions $R$ and $S$ we only take care of the vanishing line and the vanishing points but we don't take care of the position of the lines in the paintings. In order not to have something too complicated to optimize, I decided to make sure that for each group of parallel lines, the endpoints of the right-most line and the left-most line match as closely as possible with the endpoints of initial lines.

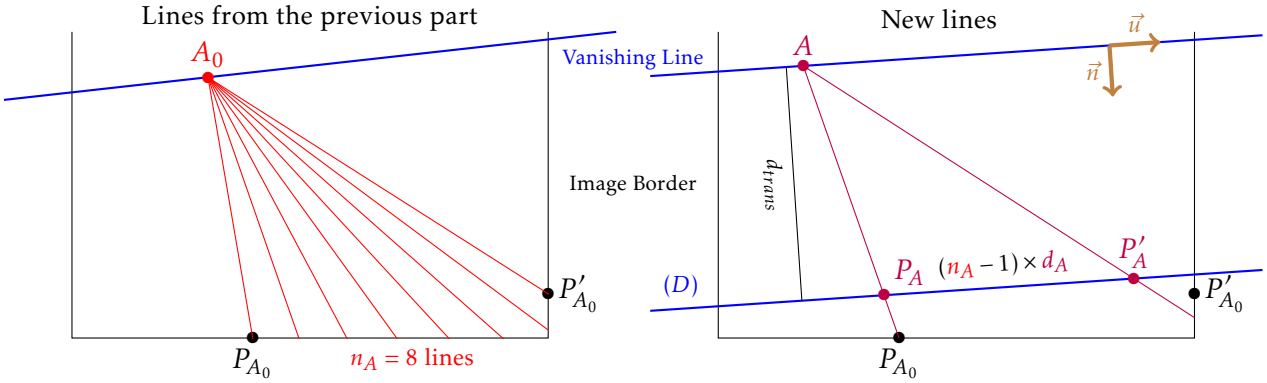



Figure 13: Constructions of a few points for the function to be minimized

We call $P_{A_0}$ the endpoint of the left-most line whose vanishing point is $A_0$ and $P'_{A_0}$ the endpoint of the right-most line whose vanishing point is also $A_0$. We define $P_{B_0}$, $P'_{B_0}$, $P_{C_0}$ and $P'_{C_0}$ in the same way. Here I will only explain what I minimize for the lines whose vanishing points is $A$, but you can replace $A$ with $B$ or $C$ for the rest of the paragraph. We can also define the unit vanishing line direction vector $\vec{u}$ by the normalization of $\overrightarrow{AC}$. Then we can obtain the normal unit vector $\vec{n}$ by a simple rotation of minus 90 degrees. Now our goal is to compute the distance between $P'_{A_0}$ and the new right-most line assuming that the left-most line passes through $P_{A_0}$. To do this, we use the fact that the intersections of lines whose vanishing point is $A$ with the line $(D)$ (the one parallel to the vanishing line and located at a distance $d_{trans}$ from it) are spaced from the distance $d_A$. Thus we define $P_A$ as the intersection between $(D)$ and $(AP_{A_0})$:

$$P_A = A + d_{trans} / \left(\vec{n}.\overrightarrow{AP_{A_0}}\right) \times \overrightarrow{AP_{A_0}}$$

Therefore the right-most line passes through the point $P'_A$ defined by:

$$P'_A = P_A + (n_A - 1) \times d_A \times \vec{u}$$

Where $n_A$ is the number of lines whose vanishing points is $A$. We finally get the wanted distance:

$$dist\left((AP'_A), P'_{A_0}\right) = \left|\det\left[\overrightarrow{P'_A P'_{A_0}} \quad \overrightarrow{AP'_A}\right]\right| / \left\|\overrightarrow{AP'_A}\right\|$$

The last term that we need to minimize is then:

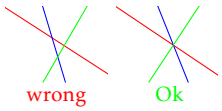
$$T = dist\left((AP'_A), P'_{A_0}\right)^2 + dist\left((BP'_B), P'_{B_0}\right)^2 + dist\left((CP'_C), P'_{C_0}\right)^2 \tag{5}$$

Thanks to the equation (3), (4) and (5) we obtain the function to minimize:

$$F(x_A, y_A, x_C, y_C, d_A, d_C) = c^{te}.R + v.S + v.T$$

This minimization is done using the Newton Raphson's method. With my way of initializing variables (which I will not detail) it generally takes no more than 3 steps of this method to converge.

Once the six parameters have been optimized, we actually obtain lines that remain close to the initial lines. But there is one thing wrong with it. Lines from the vanishing points $A$, $B$ and $C$ do not all intersect at the same time. There is a slight difference between the intersections of these lines two by two. I then slightly shift each group of lines along the direction of the vanishing line to ensure that the three groups intersect at the same time. Once this problem is solved, we obtain the result in the Figure14.



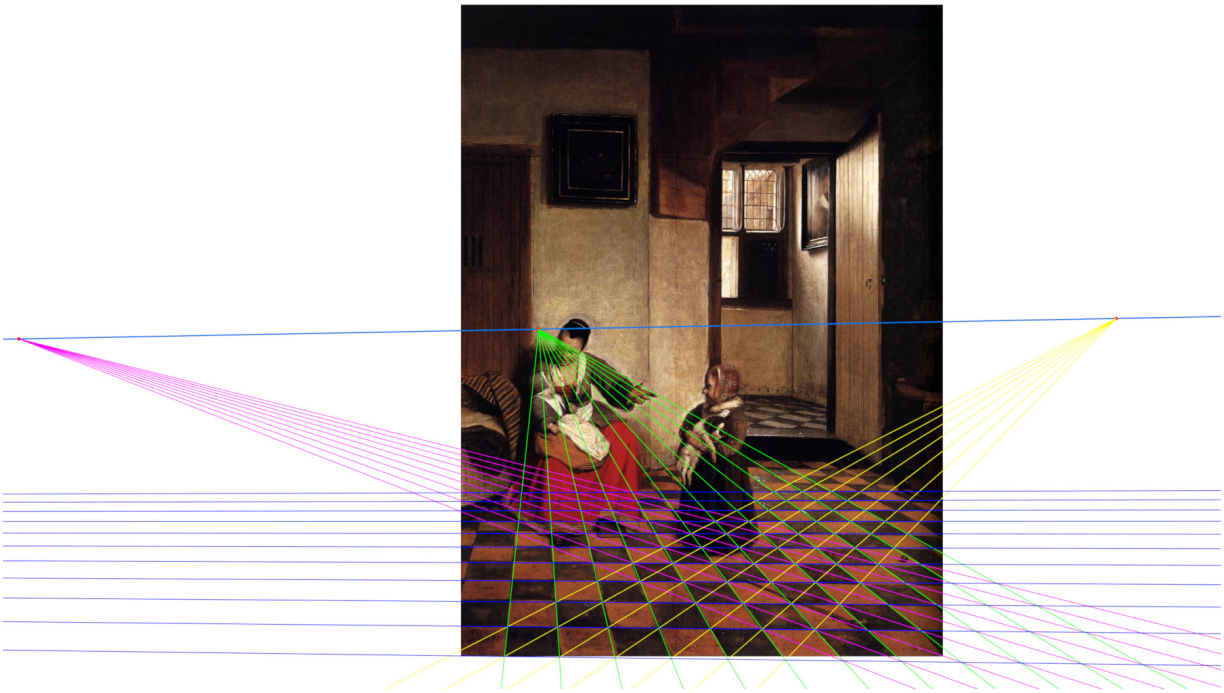



Figure 14: Result of the perfect perspective

It can be seen that these new lines match the tiles of the paint well. Unfortunately, when trying on other paints, the result obtained is not always great. It is then necessary to modify some constants in the function to be minimized to ensure that a perfect perspective really does not fit with the paint because by changing the constants it is sometimes possible to obtain a perfect perspective that fits well. To fix this we could add sub-functions to minimize like $T$ but on extreme points of different lines rather than the right-most one. On the other hand, doing this increases the computing time.

# 4 Conclusion

At the end of this internship I have obtained a graphic interface that is quite easy to use and allows us to do the desired work without human bias. Indeed, for some paints, I obtained different results from what Ricardo had obtained by hand. Of course, there is still much room for improvement. When the lines are not quite straight in the painting, the algorithm will usually find a line that matches a portion of the painting's line but not the entire painting's line. I tried to use a PCA on the pixels close to the found lines but the result was not great. Moreover, the thresholding after the Sobel filter is certainly not the best that can be done, one could imagine using a threshold that depends on local and not global intensity. Indeed currently the contours in a shaded area are less strong than the illuminated contours which is not desirable. Finally, the last part, which was supposed to consist of redrawing the paint, was not completed. This is clearly an avenue to explore.

# References


[CF97] Christine Connolly and Thomas Fliess. A study of efficiency and accuracy in the transformation from RGB to CIELAB color space. *IEEE Trans. Image Processing*, 6(7):1046–1048, 1997.

[DH72] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, 1972.

[OBBT07] Alexandrina Orzan, Adrien Bousseau, Pascal Barla, and Joëlle Thollot. Structure-preserving manipulation of photographs. In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering 2007, San Diego, California, USA, August 4-5, 2007*, pages 103–110, 2007.

[PGB03] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. *ACM Trans. Graph.*, 22(3):313–318, 2003.

[Shl14] Jonathon Shlens. A tutorial on principal component analysis. *CoRR*, abs/1404.1100, 2014.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *ICCV*, pages 839–846, 1998.


# A Source Code

Here is my git repository: `https://github.com/Nanored4498/Painting-Analysis`. The source code of the interface is in the folder `src` and there is Python script in the folder `redraw` for the section 3 about the perfect perspective.

## B Result on the other painting of the pair

Figure 15: Results on the painting that looks similar to the one in the Figure12

In this painting (Figure15) the perspective also seems to be well respected. The observation that in each pair, one painting respects perspective and the other not, is then not realy verified for the pair that illustrated this report. However, by observing the perfect perspective closest to this painting (Figure16), we can see that it does not quite match the painting. We can say that the perspective remains verified but that it is not as good as in the first painting of this pair.

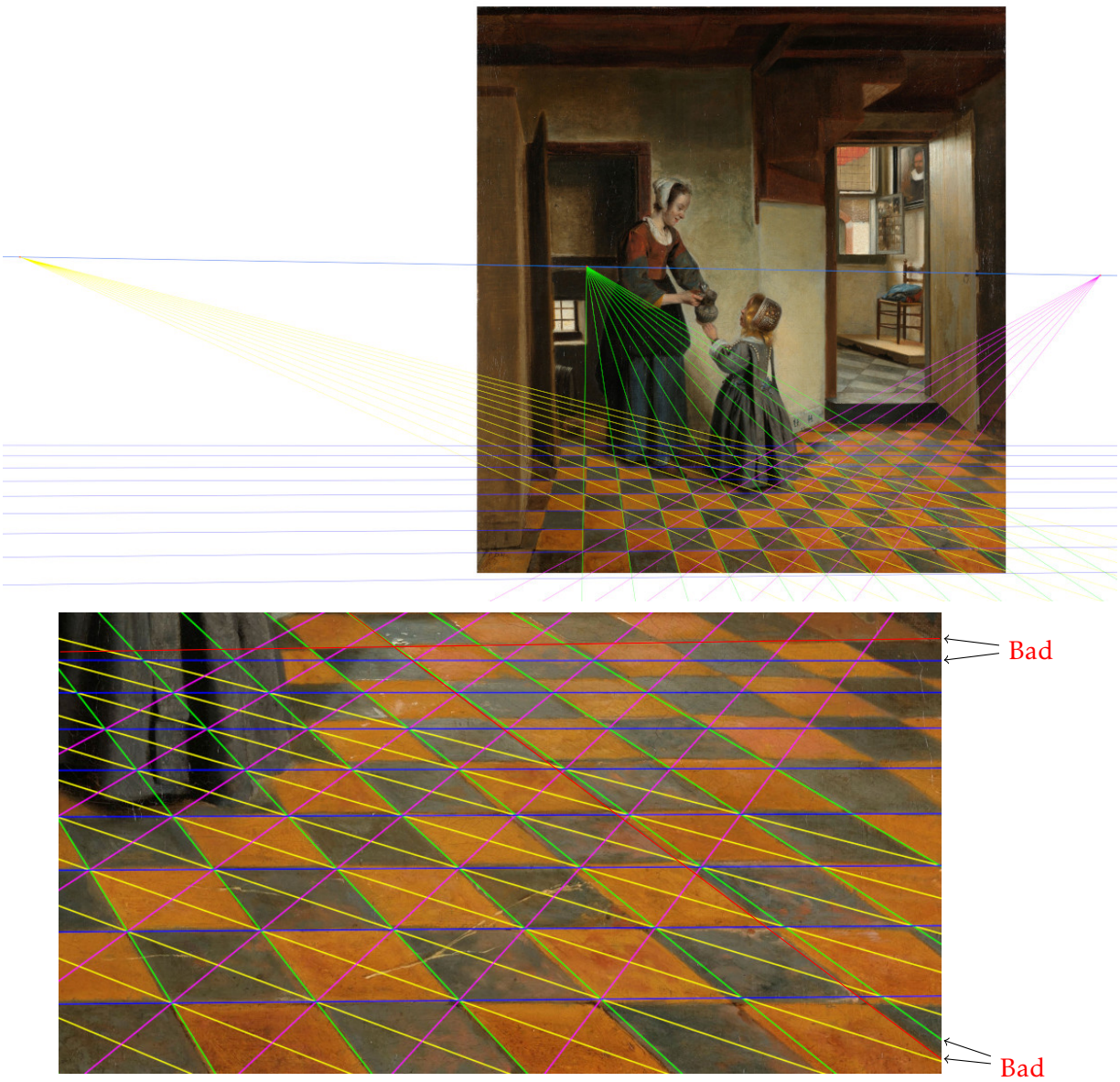



Figure 16: Perfect perspective obtained on the other painting of the pair