

**Image Processing**

**Project Documentation**

**Red Eye Detection and Removal From**

**Digital Images**

**Student:** Anca Mihai

**Group:** 30432

**Assistant Teacher:** Horațiu Florea

**Academic Year:** 2023-2024

**1. Introduction**

In dimly lit environments, individuals often utilize flash photography to illuminate subjects. However, the use of a flash can lead to an occurrence known as the *red-eye* effect. This phenomenon is caused by the flash reflecting off the blood vessels behind enlarged pupils and subsequently reaching the camera lens, resulting in the eyes appearing red in photographs.

Red-eye effect will never occur if *we do not use flash light* when we take a picture. Another light source can be used to control brightness instead of flash light. But it is *impractical* to carry another lighting device and most of people use flash light when it is dark.

Red-eye removal methods are categorized into two classes: *physics-based* and *software-based*. A *physics-based* method is to *prevent* red-eye effect: the distance between *flash and lens* can be increased so that the lens is located outside the red-eye beacon. However, the size of a camera *should be large enough*. Or the size of pupil can be made *small* by using *pre-exposure* flash. Camera will consume much power and people feel annoyed when seeing the pre-exposure flash.

A *software-based* method post-processes digital photographs using algorithms to remove existing red-eye in them. Algorithms for red-eye removal are researched by many corporations and laboratories: a lot of image editing software tools offer the function of red-eye removal and some companies developed the software for red-eye removal to apply to their products. However, a lot of these tools are implemented with *automatic* algorithms, that have *poor pupil segmentation* which leads to unnatural red eye correction: sometimes they correct red eye pixels too aggressively, *darkening eyelid areas*; or too conservatively, leaving *many red eye pixels uncorrected*.

In general, red-eye removal algorithms are composed of two parts: red-eye detection and red-eye correction.

1. *Red eye detection*:

Red eye detection strategies can be broadly divided into more classes. One of them assumes that *candidate eye regions* are somehow *identiﬁed*, either manually or automatically. In my solution, it is assumed there is a rectangle selection of the area on the image where the eyes are placed.

In most approaches the color portion of the image, the candidate to contain a pupil, is converted into a new image. It is typically a *gray scale* one, usually deﬁned as the *redness map*, and different transformations can be adopted to generate it. The candidate red pupils are usually located binarizing the redness map using, for instance, empirically deﬁned *thresholds*. Morphological ﬁlters or geometric constraints and other considerations described are then usually adopted to discriminate from true red pupils and other red spots.

As examples, according to *Benati [5]*, to identify red eye pixels, ﬁrst a threshold in the HLS color space is applied, then the pixels are grouped into spatially contiguous regions, and a score is assigned to each region based on size, shape, color and brightness. The region with the highest score corresponds to the pupil to be corrected.

Furthermore, the algorithm developed by *Gasparini and Schettini [3]* looks for red eyes in the regions with high value of redness, deﬁned as:

*Redness = (max (0, (2R-(G+B))/R))^2*

Then, to limit the number of false hits the algorithm exploits some geometric constraints: in particular, the *percentage ratio* between the *area of the candidate red* eye and the *whole face*, the *red pixel spatial distribution* and the *roundness of the region* considered.

Based on the methods presented above, my main solution is the following:

* redness: the algorithm developed by *Gasparini and Schettini*
* geometric shape: for detecting the pupil, from the red regions detected above will be considered the most centric ones who have the shape similar to the one of a circle and are big enoguh (such that to not consider other red elements which could be characteristic for a face – ex: red skin tone); for validating the geometric constraints (roundness) *Hough algorithm* can be used

2. *Red eye correction*:

Most red-eye correction algorithms *desaturate* *red color* component from red-eyes. Of course, the algorithms distinguish themselves from other algorithms by proposing the method that makes corrected red-eyes more natural.

*Patti[6]* proposed a simple red eye color correction where all the detected red eye pixels are replaced by a gray value of 0.8 of their *luminance value*. This factor is experimentally determined that yields a natural correction of the defective pixels. Before applying this color correction, a morphological pruning is performed on the mask, to avoid the correction of non-pupil regions, such as eyelids.

Other approaches adopt very simple corrections such as *Wu [7]*, where the red color of the defected eyes are simply substituted by black. Corrections of this type could be very dangerous leading to a processed image which is even worse than the defective original.

Moreover, *Gaubatz and Ulichney[8]* desaturated red color in proportion to redness in order to soften the boundary of red-eye.

Lastly, the approach that I will also use is the one of *Held[9]*: using a *correction mask* obtained by smoothing the red pupil binary map with a *Gaussian ﬁlter*. This correction mask m(i, j) can also be considered as the *probability* that a certain pixel (i, j) belongs to a red-defect region or not. Pixels approaching to the eye boundaries receive a gradually decreasing probability, allowing for a smooth change between corrected and uncorrected regions. The correction for the defects is performed on the red channel as follows:

*Rnew(i, j ) = R(i, j)−m(i, j)∗(R(i, j)−min(G(i, j), B(i, j)))*

Thus, if the probability of a pixel belongs to a red eye defect is 0, then the correction factor is 0 as well. Otherwise, the red channel will be pulled toward the minimum of both the blue and green channels. To avoid an unpleasant color shift, the correction is adjusted in case of too large a difference between the blue and green channels, as indicated by the following equations:

*if G>Rnew then Gnew = (Rnew +B)/2*

*if B > Rnew then Bnew = (Rnew +G)/2*

**2. Dataset**

The dataset I choose to use in developing this project consists of 10 images containing instance of red-eye occurrences in humans. These images were obtained from publicly accessible repositories, some of them even can be found in the examples offered by the research papers mentioned in the previous chapter. The images from this set were selected because they offer an accurate representation of a red eye generated by a digital camera and because they provide a wide range of scenarios (backgrounds included) and lightning environments: in this way they support thorough examination and assessment of the algorithm I propose for red-eye detection and correction.

C++ programming language along with the OpenCV library will be used for the implementation, which will be tested with the dataset mentioned right before. The powerful capabilities of OpenCV in image processing and computer vision make it suitable for effectively perform tasks related to detecting and correcting red-eye issues.

**3. Description of method**

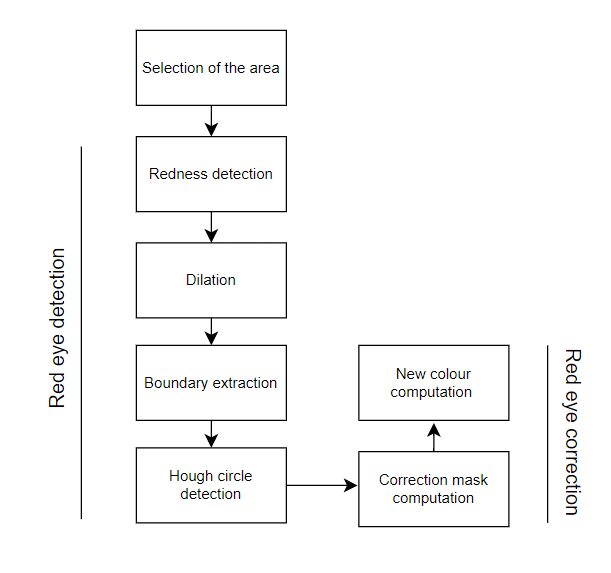
****

Fig 1: Diagram of the components of the method

The approach I used for solving the problem consists of several steps:

**1.** *Selection of the area*: the user must first select the region of interest in which the detection of eyes is considered; with the help of mouse callbacks, the left-up and right-down corner points are detected and, based on them, the rectangle which represents the selected area is determined.

**2.** *Redness detection*: using the formula: *Redness = (max (0, (2R-(G+B))/R))^2,* the points will have their redness determined and those with a bigger value than a determined threshold will be considered “red enough” and will be white in a new grayscale intermediary image; the other points will be black in the intermediary image

**3.** *Dilation*: on the resulting image, in order to improve the shape of the detected regions, a *dilation* using a *4 neighbors-structuring element* is used

**4**. *Boundary extraction*: on the newly-obtained result, the contours of the detected red shapes are extracted by using the difference between the result and the result if it was put through *erosion*

**5*.*** *Hough circle detection* (reference – *[10]*)**:**

The general equation of a circle is as follows: (𝑥 − 𝑎)^2 + (𝑏 − 𝑦)^2 = 𝑟^2. Using the basics of trigonometry and a given radius, any point on a circle can be calculated by:

𝑎 = 𝑥 − 𝑅𝑐𝑜𝑠(𝑡)

𝑏 = 𝑦 − 𝑅𝑠𝑖𝑛(𝑡)

In most cases, the actual radius of the circle is not known: a voting-based algorithm will be used:

***I****.*

*1*: Initialize: accum [Rows][Cols][Radius] = 0

Initialize: sin [] and cos [] loop up table arrays for every angle n from 0 to 360 degrees

*2*: for each x in Row do

*3*: for each y in Cols do

*4*: if cell(x,y) != 0 then //***Look for edge***

*5*: for each r in Radius do ***// the Interval [minRadius,maxRadius] is determined based on the size of the original image***

*6*: for each 𝑛 ∈ (0,360) do

*7*: b = y – r \* sin[n]

*8*: a = x – r \* cos[n]

*9*: if a 𝜖 (Rows, Cols) and b 𝜖 (Rows, Cols) then

*10*: accum[x][y][r] += 1 //***Voting***

*11*: end if

*12*: end for

*13*: end for

*14*: end if

*15*: end for

*16*: end for

***II***.

*Kernel Size 𝐾 > 0* is the size of the window to search through;

*Circle Threshold C* is the threshold required to consider a vote a pixel.

*I\_Dst(Rows,Cols)* is the destination image.

*1*: Initialize pixel which will keep track of the highest vote for the pixel in the

accumulator array

Initialize *x0,y0,r0* which will keep track of the index of the highest voted pixel

Initialize *temp* which will temporarily hold the highest vote

*2*: for each x in Row do

*3*: for each y in Cols do

*4*: pixel = 0, temp = 0

*5*: for each i in K do

*6*: for each k in K do

*7*: for each r in Radius do

*8*: temp = accum[x+i][y+j][r]

9: if temp > pixel then

*10*: pixel = temp

*11*: x0 = x+i

*12*: y0 = y+j

*13*: r0 = r

*14*: end if

*15*: end for

*16*: end for

*17*: if 𝑝𝑖𝑥𝑒𝑙 > 𝐶 then

*18*: for each 𝑛 ∈ (0,360) do

*19*: b = y0 – r0 \* sin[n]

*20*: a = x0 – r0 \* cos[n]

*21*: if a and b 𝜖 (Rows, Cols) then

*22*: I\_Dst(a,b) = 255

*23*: end if

*24*: end for

*25*: end if

*26*: end for

*27*: end for

Depending on the performance of the edge detection procedure, many centers can be detected causing false detections. That’s why a threshold (depending on the size of the original image) is set to allow only votes higher than that threshold to be counted as centers. However, this could be insufficient because the centers for different radii could still be present: a window is used to traverse through the accumulator array at different radii and select only the local maximum for that window. After this step, duplicated circles are eliminated by taking into account the minimum distance between two different centers. Lastly, the two circles the closest to the center of the selected area are selected as the red eyes.

**6.** *Correction mask computation* – In case two circles representing the eyes are detected, they are filled; borders of size 1-2 pixels (depending on the size of the radii of the detected eyes) are possible to be added in order to ensure that some potential red pixels ignored by the steps above are also corrected. For a smooth change between corrected and uncorrected regions, a Gaussian blur filter is applied with the kernel size = 5.

**7.** *New colour computation* **–** for the points different from back in the intermediary result the following formulas are applied on the original selected area:

*Rnew(i, j ) = R(i, j)−m(i, j)∗(R(i, j)−min(G(i, j), B(i, j)))*

*if G>Rnew then Gnew = (Rnew +B)/2*

*if B > Rnew then Bnew = (Rnew +G)/2*

where *m(i,j) = value of point/255 (value of white)*

**References**

**[1]** Seunghwan Yoo and Rae-Hong Park. Red-eye detection and correction using inpainting in digital photographs. *Consumer Electronics, IEEE Transactions on*, 55:1006 – 1014, 09 2009

**[2]** Francesca Gasparini. *Automatic red-eye removal for digital photography*. 09 2008

**[3]** F. Gasparini and R. Schettini, *“Automatic redeye removal for smart enhancement of photos of unknown origin,”* *Lecture Notes in Computer Sciences*, vol. 3736, pp. 226–233, 2006.

**[4]** Sepideh Samadzadegan. *Automatic and Adaptive Red Eye Detection and Removal - Investigation and Implementation*. 05 2012

**[5]** P. Benati, R. Gray, and P. Cosgrove, *“Automated detection and correction of eye color defects due to ﬂashillumination,”* US Patent 5748764, May 1998.

**[6]** A. Patti, K. Konstantinides, D. tretter and Q. Lin, *"Automatic digital redeye reduction," Image Processing*, vol. 3, pp. 55-59, 1998.

**[7]** D. Wu, *“Automatic red eye removal,”* US Patent 0232481, October 2005.

**[8]** M. Gaubatz and R. Ulichney*, “Automatic red-eye detection and correction,”* in Proc. IEEE Int. Conf. Image Processing, vol. 1, pp. 804–807, Rochester, NY, Sep. 2002.

**[9]** A. Held, “*Model-based correction of red eye defects*,” in Proceedings of the IS&T/SID Tenth Color Imaging Con-ference: Color Science and Engineering Systems, Technologies, Applications, (Scottsdale, Arizona), pp. 223–228, November 2002.

**[10]** C. Lemus, “*A Circle Hough Transform implementation using High-level Synthesis,*”, University of Nevada, Las Vegas, December 2020.