Red-Eye Correction and Object Removal
EE604A

Aditya Vikram(14040)

1 Red Eye Correction

1.1 Problem Statement

Given a frontal face image, find and correct the red eye affected regions.

1.2 Red Eye Effect

The red-eye effect is the appearance of red pupils commonly caused in flash photography, when the light of the flash is reflected off the blood vessels in the choroid at the back of the eyeball and comes out through the pupil as red. This reflected light is recorded by the camera. Photographs of children are more prone to red-eye effect because children's eyes have a rapid dark adaption as the pupils enlarge sooner in low light and the red pupil gets enlarged. Red-eye effect ruins a picture, and the following method is aimed at automating the process of red-eye correction.

1.3 Procedure

1. First, we detect the eyes from the face image using OpenCV's Haar feature-based cascade classifier. We load the pre-trained classifier from the XML file on opencv's git repository[3]. Subsequent processing occurs only on these eye region(s).

2. In each of these eye regions, we compute values of a *redness* parameter defined in [4] as

$$redness(i,j) = \frac{R(i,j)^2}{G(i,j)^2 + B(i,j)^2 + K}$$

   where R,G and B are the red, green and blue components of the image at pixel (i, j) and K is meant to avoid singularities (I set K = 1).

3. Based on the redness values, a strong threshold (default value 4) is set to form a binary mask. This is done to avoid false positives due to non-eye regions areas which have comparable redness.

4. *Region growing* is applied to the obtained mask with a weak threshold (default value 2) as an inclusion criteria. Basically, for each point in the above mask, all points in its 8-connect neighborhood having $redness > weak\_threshold$ are recursively included in the mask. This grows the red eye mask to include the whole red eye region, and also avoides false positives and noise which could have come in had we just thresholded at $weak\_threshold$ in the previous step.

5. The gaps and holes in the above mask are filled using closing operation with a circular mask. Remaining gaps in binary mask (say M) are filled [6] by applying *Flood-fill* from point $(0, 0)$, followed by inverting the image, and taking a bitwise or with M. At this point, all holes in the mask have been filled.

6. In the mask obtained, *shape filtering* [5] is performed to remove the non eye shaped red patches. This step is based on the following heuristics for the shape of an eye:

$$\begin{aligned} k_w^{min} W \leq w_i \leq k_w^{max} W \\ k_h^{min} H \leq h_i \leq k_h^{max} H \\ \frac{1}{2} \leq \frac{w_i}{h_i} \leq 2 \\ \text{and } a_i \geq \frac{w_i h_i}{2} \end{aligned}$$

   where w_i, h_i and a_i denote the width, height and area of the i-th probable eye patch, with W and H signifying the width and height of the eye bounding box. I experimented around with values of the constants and found

that most of the false positive are removed at $k_w^{min} = k_h^{min} = 0.1, k_w^{max} = k_h^{max} = 0.5$. All white patches not satifying these criteria are removed.

7. We correct the white areas in the final mask using

$$R(i,j) = G(i,j) = B(i,j) = \frac{G(i,j) + B(i,j)}{2} \quad \forall (i,j) \text{ such that } mask(i,j) = 1$$

This is our output image.

Figure 1: Red-Eye correction in action. (From left to right, top to bottom) (a) Input Image (b) Eye detection (c) Final corrected image (d) One of the eye masks after step 3 (e) The same mask after region growing (f) Same mask after filling holes

2 Object Removal

2.1 Problem Statement

Remove an input selected region in an image by filling it out the with background texture and color.

2.2 Introduction

Many a times, images contain some unwanted objects like photobombs, non-aesthetic or out of place objects, dust spots on old photographs, etc. Object removal algorithms come in handy in such situations, when you want to get rid of something in an image like it was never present. I have implemented the object removal algorithm proposed by Criminisi, A., Pérez, P., & Toyama, K. [7].

Before them, two types of algorithms existed: one where two dimensional textures were repeated with some probability, and the other *inpainting* techniques which pay attention to linear structures. Their proposed algorithm combined these two techniques, where they do replicate the texture, but in the order similar to the inpainting technique. For determining the filling order of the removed area, two terms: *confidence* and *data* are computed. Confidence signifies the algorithm's confidence of a point having its true pixel value, and the data term incorporates the linear structure idea of inpainting techniques. The next section describes the notations used in the paper and the algorithm that follows.

2.3 Notations

I	:	Inpur image
Ω	:	Target region, the region to be filled
$\mathrm{d}\Omega$	:	Fill front, the contour of the target region
Φ	:	Source region, the region to be filled $(\Phi = I \setminus \Omega)$
p	:	Pixel under consideration
Ψ_p	:	Square patch (default size 9×9) centred at p

2.4 Procedure

2.4.1 Finding Fill front

To find the fill front $\mathrm{d}\Omega$, we have to essentially detect the edges in a target mask, i.e. initialize a target mask T with $T(i,j) = 1$ if $(i,j) \in \Omega$. Then, we convolute T with a laplacian mask to get T'. Finally, the fill front is $\mathrm{d}\Omega = \{(i,j) : T'(i,j) > 0\}$.

2.4.2 Confidence and Data terms

The confidence term signifies the amount of reliable information surrounding a pixel. More the confidence value, more is the probability of greater number of pixels filled around that point. It is logical to fill the pixels having higher confidence first. Formally, the confidence term of a pixel is defined as the average of its neighboring confidence terms lying in the source region:

$$C(p) = \frac{\sum_{q \in \Psi_p \cap \Phi} C(q)}{area(\Psi_p)}$$

Initially, we set $C(p) = 1 \; \forall p \in \Phi$, and $C(p) = 0 \; \forall p \in \Psi$. It can be seen that as we fill the region inwards, the confidence decreases monotonically because lesser number of neighboring pixels are filled, and hence more zeros occur in the sum in numerator.
The data term encourages the formation of linear structures in the region. It prioritizes pixels which lead to propagation of edges. Formally, data term is defined for the pixels on the fill front as a normalized component of the image gradient along the normal to the fillfront:

$$D(p) = \frac{|\nabla I_p \cdot \vec{n_p}|}{\alpha}$$

where ∇I_p is the image gradient at pixel p and $\vec{n_p}$ is the normal to the fill front at p, $\alpha = 255$ is a normalization constant. Thus, $D(p)$ will be larger when the gradient is along the normal, i.e. when there is an edge normal to the fill front.
The priority value of a pixel is defined as the product of the confidence and the data terms:$P(p) = C(p)D(p)$. Thus, we use a balanced priority term, which prioritizes both confidence and linear structure propagation.

2.4.3 Texture Propagation

For the pixel p with maximum priority $P(p)$ on the fill front, we find a patch in the input source region having maximum similarity with the patch centred at p. In the paper, and my implementation, patch MSE has been used as a similarity criterion, but to improve the results, we can use SSIM as our similarity criterion. So we find a pixel $\hat{q}$ such that

$$\Psi_{\hat{q}} = \underset{\Psi_q \in \Phi_0}{\arg\min} \; d(\Psi_q, \Psi_p)$$

where $d(\cdot)$ represents the patch MSE here. Also, Φ_0 represents the original source region, i.e. we find the patch only in the source region in the input, not among the image part we've reconstructed. Then, we just copy all the pixel values from $\Psi_{\hat{q}}$ to corresponding pixels in Ψ_p for pixels, which are in the target region Ω.

Also, the confidence values of pixels in Ψ_p are updated as $C(p') = C(p) \;\; \forall p' \in \Psi_p$.

Algorithm: Inpainting(I, Ω)

$\Phi := I \setminus \Omega$;
$C(p) := 1 \; \forall p \in \Phi$;
$C(p) := 0 \; \forall p \in \Psi$;
Compute Fill front $\mathrm{d}\Omega$;
while $\mathrm{d}\Omega$ *is not Empty* **do**
    Update $C(p) \; \forall p \in \mathrm{d}\Omega$;
    Compute $D(p) \; \forall p \in \mathrm{d}\Omega$;
    $P(p) := C(p)D(p) \;\; \forall p \in \mathrm{d}\Omega$;
    $p := \arg\max_{x \in \mathrm{d}\Omega} P(x)$;
    Compute the most similar patch $\Psi_{\hat{q}}$ to Ψ_p;
    Copy graylevel values for $p' \in \Psi_p \cap \Omega$ from corresponding pixels in patch $\Psi_{\hat{q}}$;
    Compute Fill front $\mathrm{d}\Omega$;
end

Figure 2: Image inpainting in action using patch size 15×15: (a) Original image (b) Input mask using GUI (c)-(e) Stages of inpainting with choosen square patches in green (f) Final image with mushoom removed

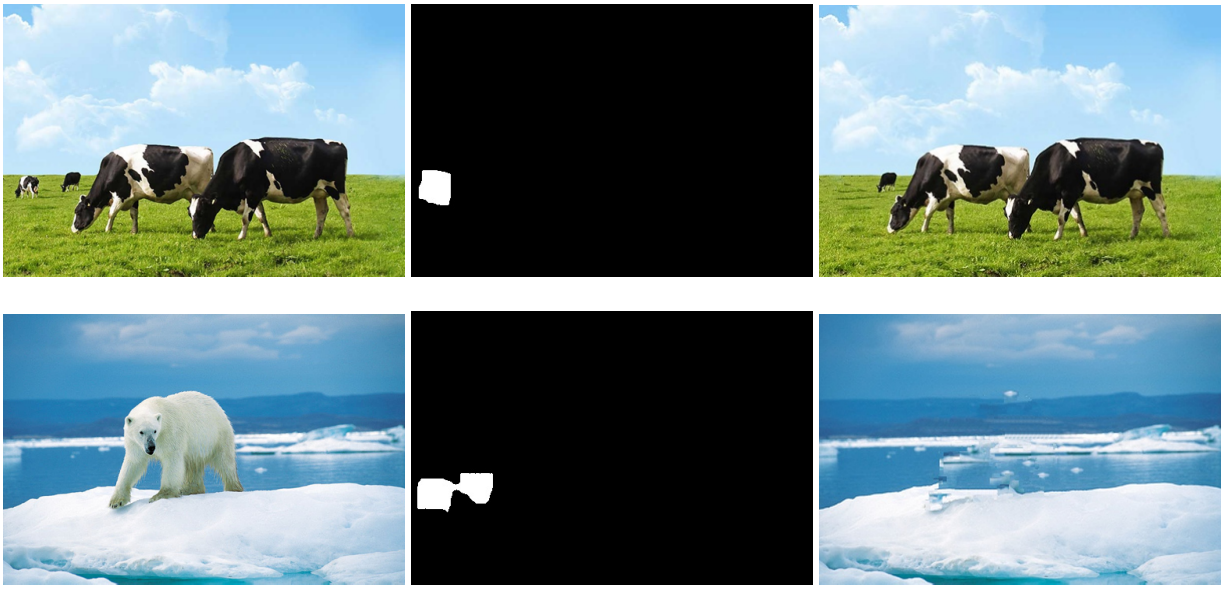
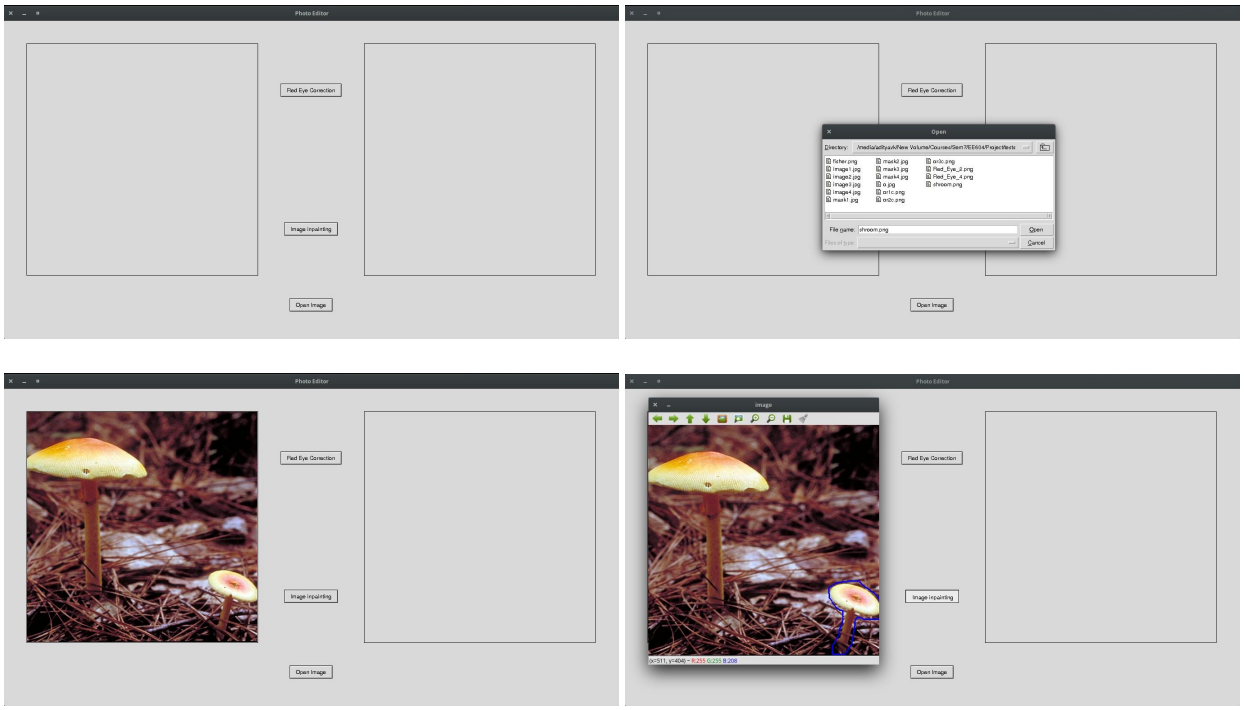
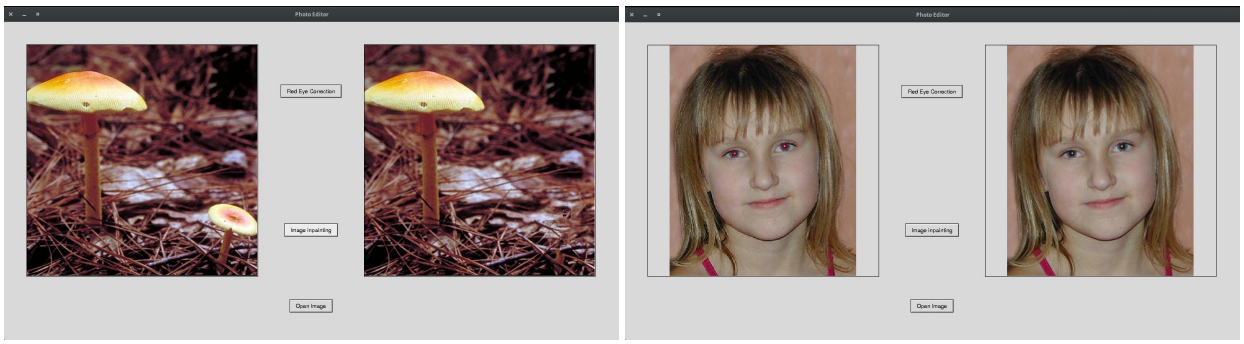


Figure 3: Image inpainting output for different test images using patch size 9×9

3 Implementation

The assignment has been coded in *Python3*, using the OpenCV library *cv2* [2] for image processing functions and mouse interaction for object selection. The GUI has been developed with *tkinter* package.
The code for the assignment can be found at : https://github.com/adityavk/IP
The GUI can be seen in action in the following snapshots:





Acknowledgement

The success of this assignment required a lot of guidance from many people and I am fortunate to have got all this along the way. I sincerely thank Prof. Tanaya Guha for providing me this opportunity which made me apply the concepts learnt in the lectures, and for her assistance throughout.
Besides, I would like to thank the people involved in developing OpenCV's tutorials [9] and documentation which were really helpful while building the application. I am also grateful to WEI Wen [8] for his python implementation of the image inpainting algorithm [7], which I referred to for an overview of the code. I would also like to thank Hemant Kumar for his useful insights and inspiring discussions for optimizing the program. Also, I would like to express my gratitude to Gaurav Verma for providing me with valuable information in a comprehensive way.
Last but not the least, I would like to thank the Stack Exchange team for putting together a wonderful platform, and the awesome Stack Overflow community for their help when I was stuck.

References

[1] Itseez. *The OpenCV Reference Manual*, 2.4.9.0 edition, April 2014.

[2] Itseez. Open source computer vision library. https://github.com/itseez/opencv, 2015.

[3] Pretrained OpenCV Haar cascade classifiers. https://github.com/opencv/opencv/tree/master/data/haarcascades

[4] Gaubatz, M., & Ulichney, R. (2002). Automatic red-eye detection and correction In *Image Processing. 2002. Proceedings. 2002 International Conference on* (Vol. 1, pp. I-I). IEEE.

[5] Revathy, M. P., & Aditanar, S. (2014). Red Eye Detection and Correction.

[6] Filling holes in an image using OpenCV. https://www.learnopencv.com/filling-holes-in-an-image-using-opencv-python-c/

[7] Criminisi, A., Pérez, P., & Toyama, K. (2004). Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on image processing, 13*(9), 1200-1212.

[8] Python implementation of Image inpainting by WEI Wen. https://github.com/veslam/Exemplar-Based-Inpaining-Python

[9] Mouse as a Paint-Brush. https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_mouse_handling/py_mouse_handling.html