DETECÇÃO DE OLHOS VERMELHOS EM IMAGENS DIGITAIS

LUIZ FILLIPE OLIVEIRA MORAIS
UFES Centro Universitário Norte do Espírito Santo
E-mail: luiz.morais@edu.ufes.br

Resumo: Esse artigo descreve o estudo do algoritmo usado para a detectar os "efeitos de olhos vermelhos" causados pela reflexão da luz no fundo do olho.

Palavras-chave: Processamento de Imagens, olhos vermelhos, reflexão, detecção de olhos vermelhos.

1 Introdução

Os olhos vermelhos em imagens digitais era um problema muito comum causado pelo flash da câmera fotográfica, e o a luz brilhante do flash era refletida pelos vasos sanguíneos do olho, dando ao olho um tom vermelho não natural. Nesse trabalho, iremos propor um sistema automático de detecção de olhos vermelhos e o combinaremos com um algoritmo de correção a fim de resolver o problema.

2 Desenvolvimento

Nesse estudo, analisaremos a imagem já registrada, ou seja, uma digitalização da imagem em um formato especifico a qual o programa possa ler. Trabalharemos no campo de espectro RGB que é uma abreviatura do sistema de cores aditivas em que o Vermelho (**R**ed), o Verde (**G**reen) e o Azul (**B**lue) são combinados de várias formas afim de produzir um largo espectro cromático. O seu propósito é a reprodução de cores em dispositivos eletrônicos como monitores de TV e computador, retroprojetores, scanners e câmeras digitas.

Uma cor no modelo de cores RGB pode ser descrita pela indicação da quantidade de vermelho, verde e azul que contém. Cada um pode varia entre o mínimo (0) que é completamente escuro e o máximo (255) que é completamente intenso. A combinação da cor preta é resultado se todas estiverem no mínimo, e para a cor branca se todas estiverem no máximo. Já se todas os valores forem próximos um dos outros teremos como resultante a cor cinza.

No espaço rgb ou rgb normalizado, uma cor é representada pela proporção de vermelho, verde e azul na cor, e não pela intensidade de cada um. Como essas proporções devem sempre somar um total de 1, podemos citar apenas as proporções vermelha e verde da cor e calcular o valor azul, se necessário.

A conversão de RGB do espaço de cores para o espaço de cores rgb é mostrada na *Equação 1.1* em que a soma das componentes r + g + b é igual a 1. O rgb normalizado é utilizado em visão computacional, desacopla cor e intensidade além de distinguir objetos pela cor independente da luminosidade do ambiente. Este é o espaço de cor rgb utilizado por *Luo et al.* em sua equação.

$$r = \frac{R}{R+G+B} \qquad g = \frac{G}{R+G+B} \qquad b = \frac{B}{R+G+B} \qquad \text{Equação 1.1}$$

3 Detecção

Existem basicamente dois tipos de detecção de olhos vermelhos, o primeiro sendo necessário reduzir manualmente a área de seleção da região ocular onda há a concentração de vermelho e aplicar o algoritmo desejado para corrigir o problema (*Figura 1*). Uma outra abordagem é digitalizar toda a imagem sem uma redução prévia do espaço de procura e após isso são submetidas a uma etapa de verificação para obter o resultado final de confirmação, geralmente baseada na cor, contraste, geometria, presença de pele, brilho etc. *(Figura 2)* e/ou verificar as regiões vermelhas candidatas pertencem a olhos obtidos anteriormente de um banco de dados *(Figura 3)*.

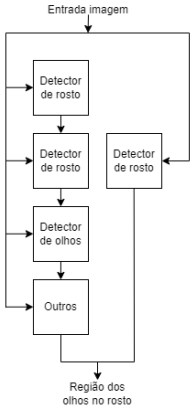

Figura 1- Fluxograma das possíveis combinações de redução da área ocular

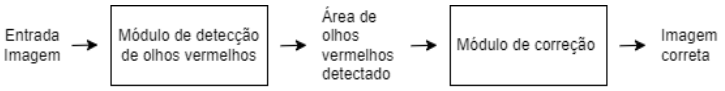

Figura 2 – Fluxograma de procura de regiões oculares avermelhadas.

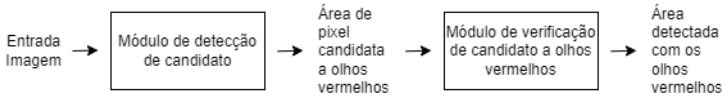

Figura 3- Fluxograma da etapa de verificação da área encontrada.

Para detectarmos os olhos vermelhos das imagens digitalizadas, utilizaremos a *Equação 1.2* fornecida pelo autor *Luo et al.* Que define uma triagem preliminar em que as áreas candidatas a olhos vermelhos são detectadas principalmente na base de vermelhidão e contraste realizando vários testes de verificação local. Em particular, cada área candidata de olhos vermelhos que não correspondem a um tom isolado numa pelo, a região do tom é filtrada a partir do mapa final dos olhos vermelhos.

Para obter os parâmetros para a segmentação dos olhos vermelhos, é usado geralmente uma amostra de imagens de olhos vermelhos reais que são manualmente selecionados a área para gerar os dados. Em seguida, uma análise de histograma é feita para otimizar o processo.

O algoritmo *Luo et al.* Define duas cores de olhos vermelhos, uma superfície de baixo contraste e uma de alto contraste. O alto contraste utiliza as curvas de contorno para separar agressivamente os pixels vermelhos dos olhos de pixels não vermelhos, já o baixo contraste utiliza faz a separação por meio da região da pele.

Equação 1.1 nos diz que a detecção é feita utilizando o pixel Vermelho ao quadrado sobre a soma de Vermelho + Verde + Azul + 1 ao quadrado. Essa é a estratégia utilizada por *Luo et al.*

$$\frac{r^2}{(r+g+b+1)^2} \qquad \text{Equação 1.2}$$

4 Validação

Agora com os candidatos a áreas de olhos vermelhos, podemos utilizar uma validação para obter o resultado corretamente. Basicamente, esse processo segue vários critérios como as restrições geométricas por forma e relação entre tamanhos de distância um do outro (se houver), a verificação dos pares de olhos vermelhos como a comparação entre a pupila vermelha e a íris. Ainda na detecção de olhos vermelhos, temos a distancia entre as regiões candidatas, sua diferença de cor e etc.

Esses processos podem ser combinados para obter um melhor resultado de áreas candidatas a conterem os olhos vermelhos. O *Luo et al.* Desenvolveu um módulo de verificação com base na análise de múltiplas características em paralelo, utilizando uma estrutura de aprendizagem de máquina (machine learnig).

5 Dificuldades e soluções adotadas

Em relação ao desenvolvimento do programa foi bem tranquilo devido ao fato de cada implementação ter sido desenvolvida de forma sequencial ao longo do semestre. Um problema que surgiu foi na implementação do limiar de LiLee em que o código escrito em C++, em que se utilizar uma variável do tipo *Byte* e em JAVA houve a necessidade de substituí-la pela variável do tipo *Inteiro*.

6 Experimentos

Agora com a parte teórica já fundamentada, podemos utilizar as formulações vistas para tentarmos encontrar uma possível detectação de olhos vermelhos. Para isso foi utilizado a implementação do algoritmo na linguagem de programação JAVA e com a ajuda do JAVA Swing foi possível criar uma GUI (do inglês, interface gráfica do utilizador) para a melhor a visualização das imagens.

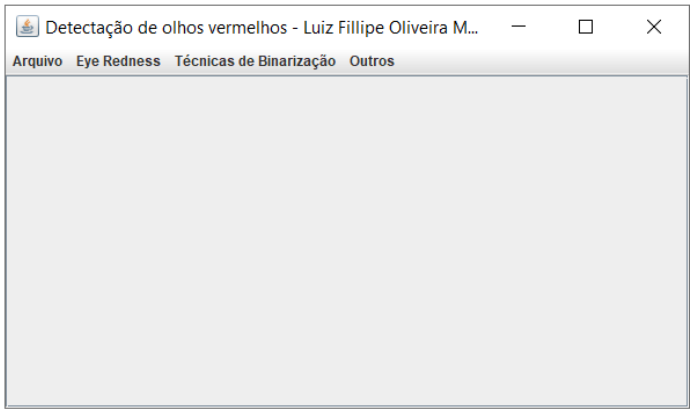

Figura 4- Interface do programa

Com isso, podemos já a começar a testar o software para ver como ele se comporta com diferentes tipos de imagens. Para facilitar, o programa conta com uma função de importação, analise e resultado em lote o que facilita a agilidade de análise.

6.1 Primeiro Experimento

Para o primeiro experimento, foi utilizado uma imagem encontrada da internet *(figura 5)* a qual se remete a uma criança com os olhos avermelhados causados pelo reflexo dos vasos sanguíneos pelo flash das câmeras fotográficas.

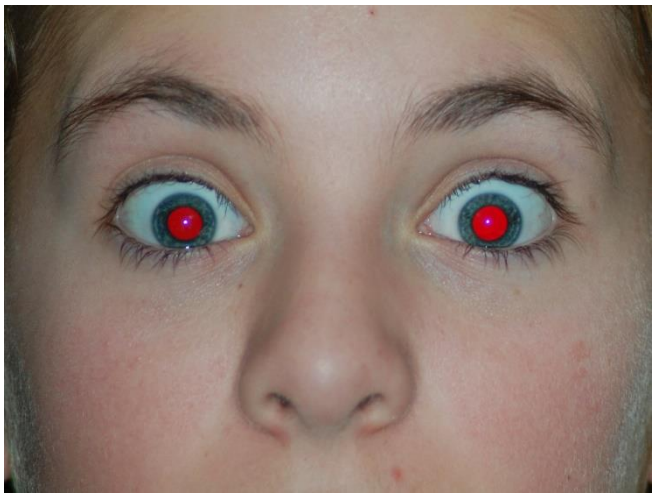

Figura 5- Imagem original

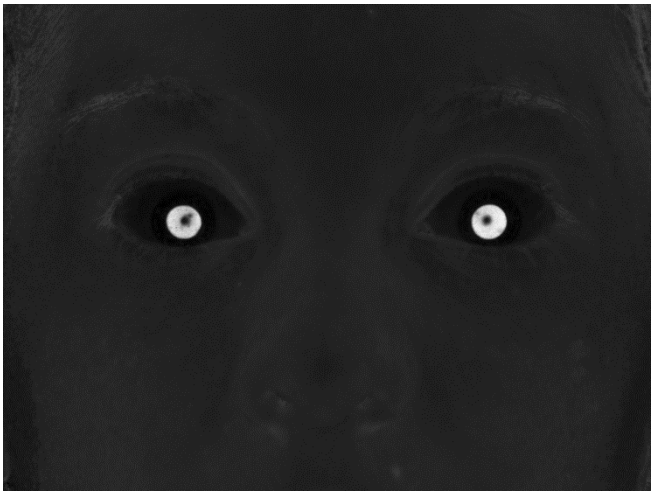

Figura 6 - Imagem após o processamento de O Luo et al

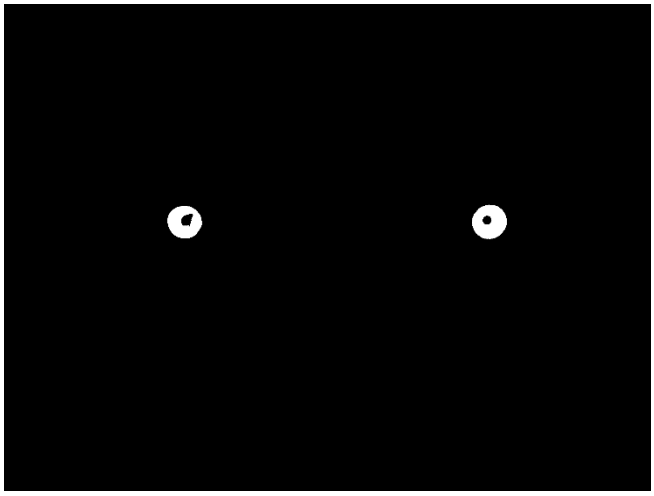

Figura 8 - Imagem binarizado pelo método de OTSU

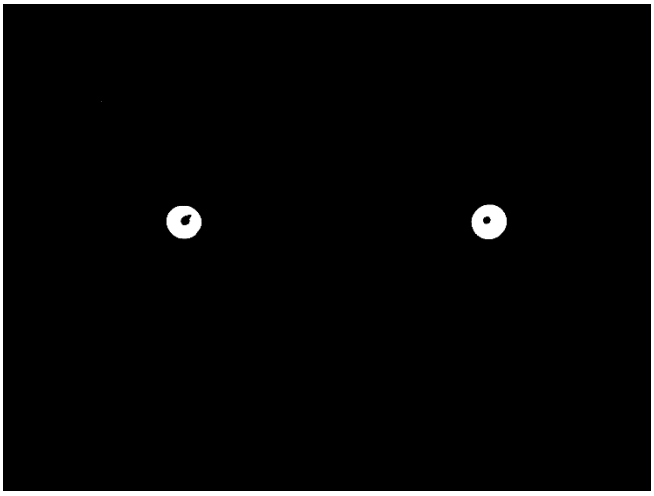

Figura 7 - Imagem binarizada pelo limiar de LiLee

6.2 Segundo Experimento

O segundo experimento se refere a uma framboesa, logo veremos como o software se comporta a imagens a qual não há uma pessoa mas sim um objeto com tons avermelhados.

Figura 10 - Imagem original

Figura 9 - Detecção utilizando o algoritmo de Luo et al.

Figura 12 - Binarização pelo método de Otsu

Figura 11 - Binarização pelo método de entropia de LiLee

7 Implementação do índice de cor redness e a binarização de LiLee

Com base na *equação 1.2* fornecida por *Luo et al* podemos colocar em prática o seu estudo. Para isso, conforme o código disponível abaixo foi necessário respeitar o espaço de cor definido pelo autor, sendo esse o rgb normalizado.

```
public class ProcessamentoImagem {

          public static BufferedImage riRedEye(BufferedImage imagem) {
                    double max = -Double.MAX_VALUE;
                    double min = Double.MAX_VALUE;
                    double ri;

                    for(int i = 0; i < imagem.getWidth(); i++) {
                     for(int j = 0; j < imagem.getHeight(); j++) {
                         Color c = new Color(imagem.getRGB(i, j));
                         double R = new Double(c.getRed());
                         double G = new Double(c.getGreen());
                         double B = new Double(c.getBlue());

                         double r = R / (R+G+B);
                         double g = G / (R+G+B);
                         double b = B / (R+G+B);


                         ri = Math.pow(r,2) / (Math.pow((r+g+b+1),2));

                         if (ri < min) min = ri;
                         if (ri > max) max = ri;
                              }
                    }

                    for(int i = 0; i < imagem.getWidth(); i++) {
                     for(int j = 0; j < imagem.getHeight(); j++) {
                         Color c = new Color(imagem.getRGB(i, j));
                         double R = new Double(c.getRed());
                         double G = new Double(c.getGreen());
                         double B = new Double(c.getBlue());

                         double r = R / (R+G+B);
                         double g = G / (R+G+B);
                         double b = B / (R+G+B);

                         ri = Math.pow(r,2) / (Math.pow((r+g+b+1),2));

                         if(((r + g) <= 0)){
                                  Color novo = new Color(0, 0, 0);
                                  imagem.setRGB(i, j, novo.getRGB());
                         } else {
                                  double riNormalizado = 255 * ((ri - min) / (max - min));
                                  int riBW = (int) riNormalizado;

                                  Color novo = new Color(riBW, riBW, riBW);
```

```
                                imagem.setRGB(i, j, novo.getRGB());
                            }
                        }}
                        return imagem;
                }
```

A seguir, apresentamos um fragmento do programa que se concentra na etapa de processamento da imagem após a detecção pelo método de *Luo et al*. Este trecho específico é encarregado pela binarização da imagem utilizando o método da entropia de LiLee. É importante destacar que o código foi escrito na linguagem Java, uma das mais populares e amplamente utilizadas na programação de aplicativos.

```
public static BufferedImage BinarizacaoEntropiaLiLee(BufferedImage img)
{
        BufferedImage res = new BufferedImage(img.getWidth(), img.getHeight(),
img.getType());
        // ROTINA de de Otsu TRADICIONAL

        //Armazenando o tamanho da imagem
        int Largura = img.getWidth();// Largura da imagem
        int Altura = img.getHeight();// Altura da imagem
        int col, lin, i, cinza;
        double totalPixel= (double) Largura * Altura;
        double [] proba = new double[256];
        int [] histogram = new int[256];
        int k, uiLimiar;

        // inicializacao do Histograma
        for(i=0; i < 256; i++){
                histogram[i]= 0;
        }

        // calculo do Histograma
        for( lin = 0; lin < Altura; lin++) {
                for( col = 0; col < Largura; col++) {
                        Color x = new Color(img.getRGB(col,lin));
                        cinza = (int)((x.getGreen() + x.getRed() + x.getBlue())/3);
                        histogram[cinza]++;
                }
        }
          long soma1, soma2, soma3, soma4;
          long total1, total2;
          double[] mi1 = new double [256], mi2 = new double[256];
          double[] ni1 = new double [256], ni2 = new double[256];
          double[] ni = new double [256];

          int l,m;
          total1    = 0;
          total2    = 0;
          for (l = 0; l < 256; l++)
          {
             ni1[l] = 0;
             ni2[l] = 0;
```

```
            total1 += l * histogram[l];
            total2 += histogram[l];
          }
          soma1     = 0;
          soma2     = 0;
          soma3     = 0;
          soma4     = 0;
          for (l = 1; l < 256; l++)
          {
                    soma1 += (l-1) * histogram[l-1];
                    soma2 += histogram [l-1];
                    soma3 = total1 - soma1;
                    soma4 = total2 - soma2;

                    mi1[l] = (soma2 != 0) ? ((double) (soma1)) / soma2 : 0;
                    mi2[l] = (soma4 != 0) ? ((double) (soma3)) / soma4 : 0;

                    for (m = 1; m <   l; m++)
                   ni1[l] += (mi1[l] != 0) ? (m * histogram[m] * Math.log(m/mi1[l])) : 0.;

                    for (m = l; m < 256; m++)
                   ni2[l] += (mi2[l] != 0) ? (m * histogram[m] * Math.log(m/mi2[l])) : 0.;
                    ni[l] = ni1[l] + ni2[l];
          }
          // busca do minimo = limiar
          uiLimiar = 1;
          for (l = 2; l < 256; l++)
          {
                    if (ni[l] < ni[uiLimiar])
                              uiLimiar = (int)l;
          }
                    //Cria a  imagem  binarizada
                    // Aloca a Matriz
                    int [][] pBufferbinario = new int[Altura][Largura]; //Cria um PONTEIRO
para a  imagem  binarizada

                    for( lin = 0; lin < Altura; lin++) {
                              for( col = 0; col < Largura; col++) {
                              Color x = new Color(img.getRGB(col,lin));
                                        cinza = (int)((x.getGreen() + x.getRed() +
x.getBlue())/3);
                                        if (cinza > uiLimiar){
                                                  pBufferbinario[lin][col] = 1;
                                        } else {
                                                  pBufferbinario[lin][col] = 0;
                                        }
                              }
                    }
                    //Aqui Gera a imagem binária
                    for( lin = 0; lin < Altura; lin++) {
                              for( col = 0; col < Largura; col++){
                                        int atual = pBufferbinario[lin][col]* 255;
                                        Color novo = new Color(atual, atual, atual);
```

```
                                    res.setRGB(col,lin, novo.getRGB());
                                }
                            }
                            return res;
                    }
```

8 Conclusões

Em conclusão, este programa de detecção de olhos vermelhos é uma ferramenta útil para identificar automaticamente problemas comuns em fotografias, como olhos vermelhos. Ele pode ajudar os usuários a corrigir rapidamente esses problemas e melhorar a qualidade das imagens. No entanto, é importante lembrar que a precisão desses programas pode variar e que eles não são capazes de corrigir todos os problemas em uma imagem. Portanto, é importante avaliar cuidadosamente a qualidade da detecção de olhos vermelhos antes de utilizar o programa em questão.

9 Referências

F. Gasparini and R. Schettini, “A review of redeye detection and removal in digital images through patents,” Recent Patents on Electrical Engineering, pp.1-9,2009.