Relatório dos exercícios propostos

Aluno: Mateus Lacerda Soares

Todos os códigos e imagens encontram-se em https://github.com/MateusLacerda/PDI

3. Manipulando pixels em uma imagem

Exercício 1

O programa regions.cpp solicita ao usuário as coordenadas dos pontos P1 e P2 para negativar a parte da imagem delimitada por esses dois pontos. Primeiramente o programa certifica-se se o ponto P1 está mais a cima e a esquerda do ponto P2, pois caso não esteja terá problema no for, pois a variável vai crescer até encontrar a outra, e se for maior que a outra nunca a encontrará. Após certificado isso, basta percorrer toda a imagem começando do ponto P1 e indo até o ponto P2 invertendo a cor de cada pixel com a operação pixel = 255 – pixel. Além disso, toda a lógica do programa está dentro de um while que perguntar se o usuário quer repetir o processo. O resultado e o programa encontram-se abaixo.

Figura 1: Resultado do programa regions.cpp

```
#include <iostream>
#include <cv.h>
#include <highgui.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main(int, char** ){
	Mat imagem;
	int x,y,x1,y1,x2,y2, temp;
	char continuar = 's';

	while (continuar=='s'){
```

```
		imagem = imread("biel.png", CV_LOAD_IMAGE_GRAYSCALE);
		if (!imagem.data){
			cout << "Imagem não abriu\n";
		}

		cout << "Insira os valores da coordenada do ponto 1 (x1,y1): ";
		cin >> x1;
		cout <<"Agora y1: ";
		cin >> y1;
		cout << "Agora insira os valores da coordenada do ponto 2 (x2,y2): ";
		cin >> x2;
		cout << "Agora y2: ";
		cin >> y2;
		cout << "\n";

		// fazer com que o ponto 1 esteja sempre mais a esquerda e mais a cima do ponto 2

		if (x1>x2){
			temp = x1;
			x1 = x2;
			x2 = temp;
		}

		if (y1>y2){
			temp = y1;
			y1 = y2;
			y2 = temp;
		}


		namedWindow("Janela", WINDOW_AUTOSIZE);

		for (x=x1; x<=x2;x++){
			for (y=y1; y<=y2; y++){
				imagem.at<uchar>(x,y) = 255 - imagem.at<uchar>(x,y);
			}
		}
		imshow("Janela", imagem);
		waitKey();
		destroyWindow("Janela");

		cout << "Quer continuar? (s/n) ";
		cin >> continuar;
	}
	return 0;
}
```

Exercício 2

O programa trocaregioes.cpp usa 4 for's para trocar os quadrantes da imagem "biel.png". O programa e o resultado podem ser vistos abaixo

Figura 2: Resultado do programa trocaregioes.cpp

```
#include <iostream>
#include <cv.h>
#include <highgui.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main(int, char** ){
	Mat imagem, imagem2;
	int x,y;

	imagem = imread("biel.png",CV_LOAD_IMAGE_GRAYSCALE);
	imagem2 = imagem.clone();

	// colocando o quarto quadrante no segundo
	for (x = 1; x<=128;x++){
		for (y = 1; y<=128;y++){
			imagem2.at<uchar>(x,y) = imagem.at<uchar>(x + 128,y + 128);
		}
	}
	//colocando o segundo quadrante no quarto
	for (x = 129; x<=256;x++){
		for (y = 129; y<=256;y++){
			imagem2.at<uchar>(x,y) = imagem.at<uchar>(x - 128,y - 128);
		}
	}
	//colocando o primeiro quadrante no terceiro
	for (x = 129; x<256;x++){
		for (y = 1; y<=128;y++){
			imagem2.at<uchar>(x,y) = imagem.at<uchar>(x - 128, y + 128);
		}
	}
	//colocando o terceiro quadrante no primeiro
	for (x = 1; x<=128;x++){
		for (y = 129; y<=256;y++){
			imagem2.at<uchar>(x,y) = imagem.at<uchar>(x + 128,y - 128);
		}
	}
```

```
        namedWindow("janela",WINDOW_AUTOSIZE);
        imshow("janela", imagem2);
        waitKey();
        return 0;
}
```

4. Preenchendo regiões

Exercício 1

A limitação desse número (255) é por causa do tipo da variável que está sendo usada (unsigned char). Para resolver esse problema, poderíamos usar um variável do tipo inteiro ou float para aumentar o limite da contagem. Contudo, visualmente a saída do programa seria prejudicada, pois nosso olho não consegue diferenciar tantos tons de cinza, porém para a visão computacional isso não seria problema.

Exercício 2

O programa exercicio4_2.cpp abre o arquivo “bolhas.png” e primeiramente pinta as bordas com o valor de 255, assim estará conectando todas as bolhas que estão na borda. Feito isso, chamamos a função “floodFill()” passando como parâmetro o valor 0 de cor a primeira coordenada, que está conectado com todas as bolhas da borda, e pintando assim estaremos eliminando estas. Após isso chamamos novamente a função “floodFill” e pintamos o fundo com a cor 10. O próximo passo é rotular as bolhas, percorrendo toda a imagem procurando o valor 255, e quando encontrado muda o seu valor. No final, saberemos quantas bolhas há, porém as bolhas com furo continuarão lá, então percorremos a imagem mais uma vez procurando o valor 0, pois estas regiões são os furos das bolhas que não foram pintados pois não estão conectados com o fundo. Uma vez encontrado o valor 0, basta chamarmos a função floodFill para pinta o furo e o pixel vizinho, pois este será a bolha que contém o furo. Os resultados e código se encontram abaixo.

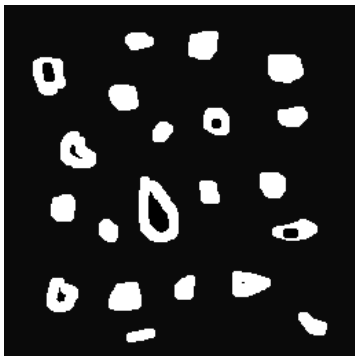

Figura 3: Programa exercicio4_2.cpp: bolhas nas bordas eliminadas

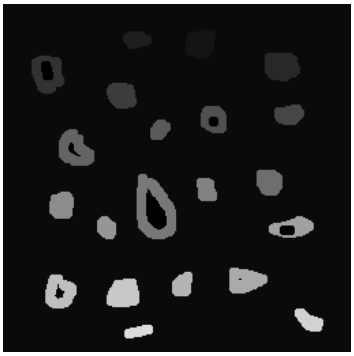

Figura 4: Programa exercicio4_2.cpp: bolhas rotuladas

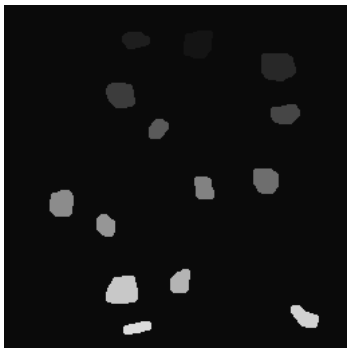

Figura 5: Programa exercicio4_2: bolhas com furo eliminadas

```
#include <iostream>
#include <opencv2/opencv.hpp>
```

```
using namespace cv;

int main(int argc, char** argv){
  Mat imagem;
  int largura, altura;
  int n;
  CvPoint p;

  imagem = imread(argv[1], CV_LOAD_IMAGE_GRAYSCALE);

  largura = imagem.size().width;
  altura = imagem.size().height;

  // pintando a borda de 255 para conectar as bolhas das bordas
  for (int x = 0; x < altura ; x++){
    for (int y = 0; y < largura ; y++){
      if (x == 0 || y == 0 || x == altura-1 || y == largura-1){
        imagem.at<uchar>(x,y) = 255;
      }
    }
  }

  //pintando o fundo de 10
  p.x = 0;
  p.y = 0;
  floodFill(imagem, p, 0);
  floodFill(imagem, p, 10);

  imshow("sem bolhas nas bordas",imagem);

  //rotulando os objetos
  n = 10;
  for (int x = 0; x < altura ; x++){
    for (int y = 0; y < largura ; y++){
      if ( imagem.at<uchar>(x,y) == 255 ){
        //achou objeto
        n = n + 10;
        p.x = y;
        p.y = x;
        floodFill(imagem, p, n);
      }
    }
  }

  printf("Número de objetos=%d\n", (n-10)/10);

  imshow("rotulados", imagem);

  //verificando se há algum 0 (furo) e apagando o objeto com furo
  for (int x = 0; x < altura ; x++){
    for (int y = 0; y < largura ; y++){
      if ( imagem.at<uchar>(x,y) == 0 ){
        if (imagem.at<uchar>(x,y-1) != 10){
          //imagem.at<uchar>(x,y) = 10;
          p.x = y-1;
          p.y = x;
          floodFill(imagem, p, 10);
          p.x = y;
          p.y = x;
          floodFill(imagem, p, 10);
        }
      }
```

```
  }
 }

 imshow("Bolhas sem furo",imagem);
 waitKey();
 return 0;
}
```

5. Manipulação de histogramas

Exercício 1

O programa *equalize.cpp*, ao pegar cada quadro do vídeo, realiza a equalização da imagem antes de exibi-la. Para tal, usamos a função *equalizeHist()* para equalizar o histograma da imagem desejada. Assim, pegamos o programa *histograma.cpp* como base e adicionamos essa função para equalizar cada canal de cor da imagem. Adicionamos o seguinte trecho de código:

```
//equalizando o histograma da imagem capturada
equalizeHist(planes[0], planes[0]);
equalizeHist(planes[1], planes[1]);
equalizeHist(planes[2], planes[2]);
```

As imagem abaixo mostram uma imagem não equalizada e outra equalizada

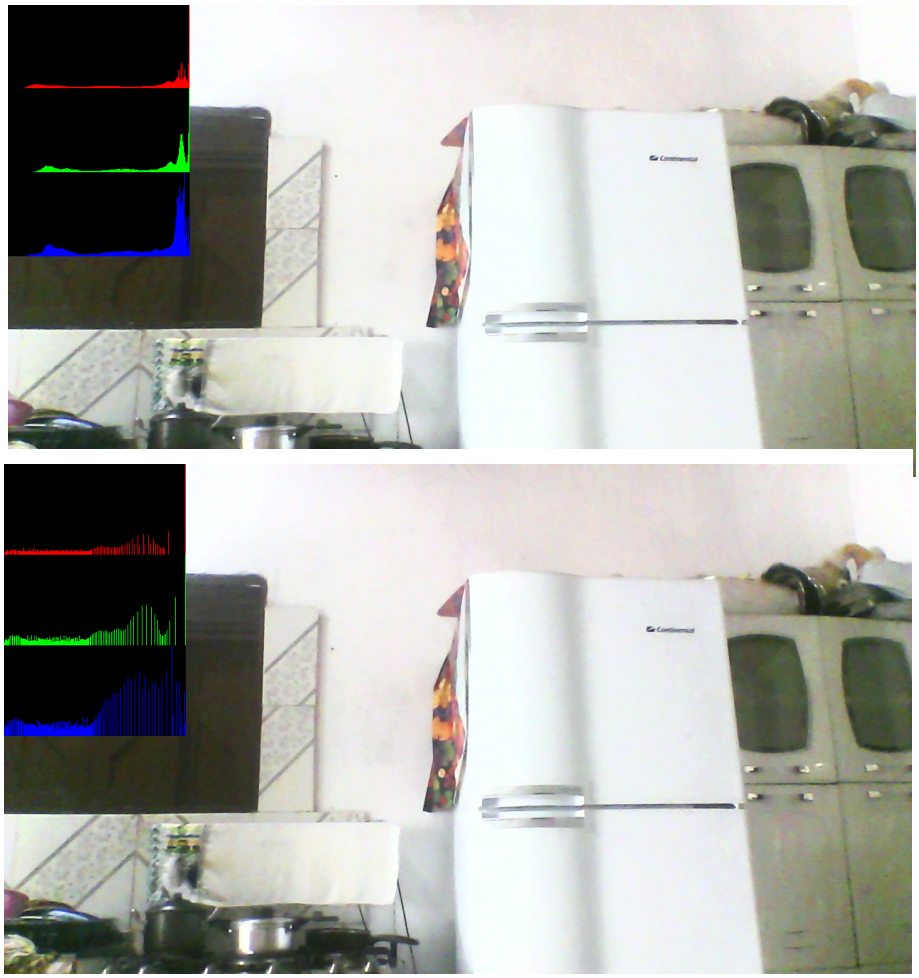

Figura 7: Imagem equalizada

Exercício 2

O programa *motionDetector.cpp* consiste em capturar duas imagens e em seguida calcular os histogramas dos canais de cores das mesmas, e então comparar esse dois histogramas para verificar se há a presenta de movimento. Para implementar essa funcionalidade, adicionamos o seguinte trecho de código no programa *histograma.cpp*:

```
if (Compare(histImgR, histImgR2, histh, histw)){
  if (Compare(histImgG, histImgG2, histh, histw)){
    if (Compare(histImgB, histImgB2, histh, histw)){
        cout << "Movimento"<< endl;
    }
  }
}
```

e a implementação da função *Compare()*:

```
bool Compare(Mat hist1, Mat hist2, int histh, int histw){
  int cont = 0;
  for (int i = 0 ; i < histw ; i++){
    for (int j =0 ; j < histh ; j++){
      if (abs(hist1.at<uchar>(i,j) - hist2.at<uchar>(i,j)) > (uchar)50){
        cont++;
      }
    }
  }
  if (cont > 280){
    return true;
  }else{
    return false;
  }
}
```

Como a câmera usada no meu computador insere bastante ruído na imagem, colocamos uma marguem grande de erro para a verificação da igualdade entre os histogramas.

6. Filtragem no domínio espacial I

Exercício 1

No programa *laplgauss.cpp* adicionamos a funcionalidade ao programa *filtroespacial.cpp* de ter a opção de aplicar os filtros laplaciano e gaussiano na mesma imagem. Para tal, inserimos o seguinte trecho de código:

```
if (laplgauss){
     mask = Mat(3, 3, CV_32F, gauss);
     scaleAdd(mask, 1/16.0, Mat::zeros(3,3,CV_32F), mask1);
     mask = mask1;
     filter2D(frame32f, frameFiltered, frame32f.depth(), mask, Point(1,1), 0);
     mask = Mat(3, 3, CV_32F, laplacian);
     frame32f = frameFiltered.clone();
     filter2D(frame32f, frameFiltered, frame32f.depth(), mask, Point(1,1), 0);
     imwrite("Aplicacao_do_laplgauss.png",frameFiltered);
   }else{
     filter2D(frame32f, frameFiltered, frame32f.depth(), mask, Point(1,1), 0);
   }
```

[…]

```
case 'n':
   menu();
     imwrite("Entrada_do_laplgauss.png", frame);
     laplgauss = true;
     break;
```

As imagens abaixo mostram o resultado da aplicação:

Figura 9: Saída do programa laplgauss.cpp

A visão está prejudicada por causa da baixa qualidade da câmera, porém o que observamos é um realce nas bordas das figuras da imagem.