Processamento Digital de Imagens

Table of Contents

1 Manipulando pixels em uma imagem

Manipular pixels em uma imagem envolve a capacidade de alterar individualmente os elementos de cor que compõem a imagem. Cada pixel contém informações sobre sua cor específica, como vermelho, verde e azul (RGB), além de valores de transparência em alguns casos. Ao manipular os pixels, é possível realizar uma variedade de transformações na imagem, como ajustar o brilho, a saturação, o contraste, aplicar filtros, redimensionar ou recortar. Essas manipulações permitem corrigir imperfeições, realçar detalhes, criar efeitos especiais, entre outras possibilidades. A manipulação de pixels é uma técnica fundamental em áreas como processamento de imagem, design gráfico, edição de fotos e criação de arte digital. Com a ajuda de bibliotecas de processamento de imagem, é possível acessar e modificar os valores dos pixels em uma imagem, abrindo caminho para inúmeras possibilidades criativas e práticas.

1.1 Exercicio 1.1 - Filtro Negativo (regions.cpp)

O filtro negativo é um mecanismo que busca eliminar conteúdos indesejados, inapropriados ou prejudiciais por meio da aplicação de técnicas de análise e restrição. Ele é amplamente utilizado em diversas áreas, como tecnologia da informação, mídias sociais e segurança, visando proteger os usuários e garantir a qualidade do conteúdo. No entanto, é necessário ter cautela para evitar restrições excessivas e preservar a liberdade de expressão.

Com isso, crie um programa chamado regions com base no código de exemplo "pixels.cpp". O programa deve pedir ao usuário as coordenadas de dois pontos, P1 e P2, que estão dentro dos limites do tamanho da imagem. Em seguida, ele deve exibir a imagem fornecida, mas com a região definida pelo retângulo formado pelos pontos P1 e P2 exibindo o negativo da imagem nessa área específica. Esse efeito produzirá uma região destacada com cores invertidas em relação ao restante da imagem.

1.1.1 Código e Resultado.

regions.cpp

```
// Renato Emanuel Medeiros de Lira
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main(int, char** argv) {

    Mat image;
    image=imread(argv[1], IMREAD_GRAYSCALE);

    if (image.empty()) {
        cout << "Imagem não foi carregada" << endl;
        return 1;
    }

    Point p1, p2;

    cout << "Tamanho da imagem: " << image.rows << "x" << image.cols << endl;

    cout << "Digite as coordenadas do ponto P1:" << endl;
    cin >> p1.y >> p1.x;

    do{
    cout << "Digite as coordenadas do ponto P2:" << endl;
    cin >> p2.y >> p2.x;
    if (p1.y >= p2.y || p1.x >= p2.x){
        cout << "Ambas coordenadas do ponto P2 tem que ser maiores que as coordenadas
do ponto P1, escreva novamente!"<<endl;
    }
    }while(p1.y >= p2.y || p1.x >= p2.x);

    for (int i = p1.x; i < p2.x; i++) {
        for (int j = p1.y; j < p2.y; j++) {
            image.at<uchar>(i, j) = 255 - image.at<uchar>(i, j);
        }
    }
```

```
    namedWindow("janela", WINDOW_AUTOSIZE);
    imshow("janela", image);
    waitKey(0);

    imwrite("Regions.png", image);

    return 0;
}
```

Abaixo temos a imagem original biel e em seguida a imagem resultante do código regions. As Coordenadas escolhidas foram P1 (90, 180) e P2 (90, 180).

Figure 1. Imagem original

Figure 2. Saída do programa regions.cpp

1.2 Exercicio 1.2 - Troca Regiões (trocaregioes.cpp)

Desenvolva um programa chamado [trocaregioes.cpp] com base no código de exemplo "pixels.cpp". O programa deve trocar os quadrantes diagonais na imagem. Para realizar essa troca, utilize a classe "Mat" e seus construtores para criar as regiões que serão intercambiadas. Ao executar o programa, a imagem exibida terá os quadrantes diagonais trocados.

1.1.2 Código e Resultado.

trocaregioes.cpp

```
// Renato Emanuel Medeiros de Lira
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main(int, char** argv) {

  Mat image;
    image=imread(argv[1], IMREAD_GRAYSCALE);

    if (image.empty()) {
        cout << "Imagem não foi carregada" << endl;
        return 1;
    }

  int x = image.rows;
  int y = image.cols;

  Mat temp = image.clone();

  Rect region1(0, 0, y/2, x/2);
  Rect region2(y/2, 0, y/2, x/2);
  Rect region3(0, x/2, y/2, x/2);
  Rect region4(y/2, x/2, y/2, x/2);

  Mat roi1 = temp(region1);
  Mat roi2 = temp(region2);
  Mat roi3 = temp(region3);
  Mat roi4 = temp(region4);

  roi3.copyTo(image(region2));  // R3 -> R2
  roi4.copyTo(image(region1));  // R4 -> R1
  roi1.copyTo(image(region4));  // R1 -> R4
  roi2.copyTo(image(region3));  // R2 -> R3

  imshow("janela", image);
  waitKey(0);

  imwrite("trocaregioes.png", image);

  return 0;
}
```

Abaixo temos a imagem original biel e em seguida a imagem resultante do código trocaregioes, na

qual os quadrantes foram trocados de lugar.

Figure 3. Saída do programa trocaregioes.cpp

2. Serialização de dados em ponto flutuante via FileStorage

A serialização de dados em ponto flutuante usando o FileStorage envolve a gravação e leitura de valores numéricos em formato de ponto flutuante em um arquivo estruturado. Ao utilizar a classe FileStorage, é possível gravar os dados em um formato como YAML ou XML, permitindo a preservação da precisão fracionária. A leitura dos dados serializados também é facilitada pela classe FileStorage, tornando possível acessar e utilizar os valores gravados posteriormente. Esse processo é especialmente útil em cenários onde é necessário armazenar e recuperar informações numéricas precisas, como em aplicações científicas, processamento de imagens e simulações.

2.1. Exercicio 2 - filestorage.cpp

Utilizando o programa "filestorage.cpp" como base, crie um novo programa que gere uma imagem de 256x256 pixels contendo uma senóide horizontal com 4 períodos e amplitude de 127, semelhante à figura apresentada na [filestorage_exemplo]. Grave a imagem em formato PNG e em formato YML. Em seguida, compare os arquivos gerados, extraindo uma linha de cada imagem gravada e calculando a diferença entre elas. Trace um gráfico dessa diferença ao longo da linha correspondente nas imagens. Observe os resultados e faça as observações relevantes.

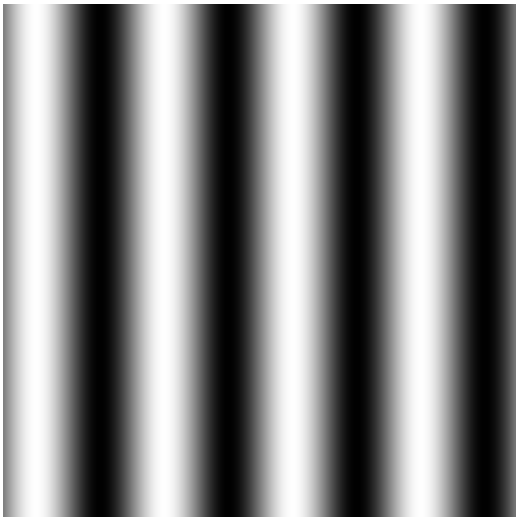

Figure 4. Imagem de uma senóide horizontal com 8 periodos

2.1.1. Código e Resultado.

filestorage.cpp

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

using namespace std;
using namespace cv;

int main() {
    int side = 256;
    int periodos = 4;

    Mat image(side, side, CV_8UC1);

    for (int i = 0; i < side; i++) {
        for (int j = 0; j < side; j++) {
            float value = 127 * sin(2 * CV_PI * periodos * j / side) + 128;
            image.at<uchar>(i, j) = static_cast<uchar>(value);
        }
    }

    string filenameYML = "senoide.yml";
    string filenamePNG = "senoide.png";

    FileStorage fs(filenameYML, FileStorage::WRITE);
    fs << "mat" << image;
    fs.release();

    imwrite(filenamePNG, image);

    FileStorage fs_read(filenameYML, FileStorage::READ);
    fs_read["mat"] >> image;
    fs_read.release();
```

```
    imshow("image", image);
    waitKey();

    return 0;
}
```

Abaixo temos a imagem resultante do código filestorage, na qual é representado uma senoide horizontal com 4 periodos e amplitude de 127. A imagem é semelhante a figura [filestorage_exemplo] usada como exemplo.

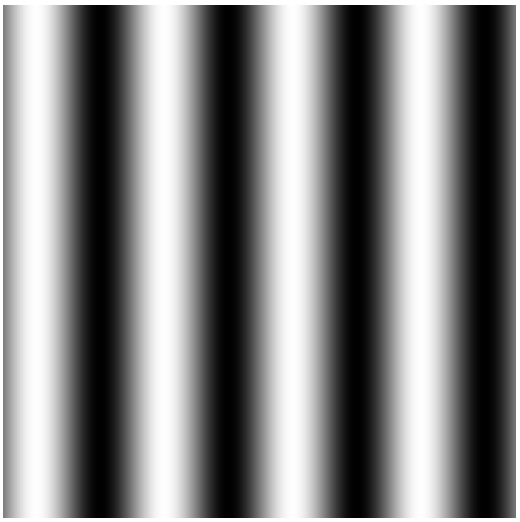

Figure 5. Saída do programa filestorage.cpp

3. Decomposição de imagens em planos de bits

A decomposição de imagens em planos de bits é um processo onde uma imagem é separada em camadas binárias, cada uma representando um plano de bits específico. Isso permite analisar e manipular individualmente cada componente da imagem, seja em escala de cinza ou em cores, como vermelho, verde e azul. Essa técnica é útil em diversas aplicações, como processamento de imagem, análise de padrões e compressão de dados, possibilitando identificar características específicas e realizar manipulações precisas na intensidade de cor de cada pixel.

3.1. Exercicio 3 - esteg-encode.cpp

Escreva um programa que seja capaz de recuperar uma imagem codificada resultante de esteganografia. No processo de recuperação, os bits menos significativos dos pixels da imagem fornecida devem ser utilizados para compor os bits mais significativos dos pixels da imagem recuperada. O programa deve receber como parâmetro de linha de comando o nome do arquivo da imagem resultante da esteganografia

3.1.1. Código e Resultado.

esteg.cpp

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

using namespace std;
using namespace cv;

int main(int argc, char** argv) {
    Mat imagemCodificada, imagemEscondida, imagemPortadora, imagemDecodificada;
    Vec3b valCod, valEsc, valPort, valDec;
    int nbits = 5;

    // Recebendo a imagem codificada;
    imagemCodificada = cv::imread(argv[1], cv::IMREAD_COLOR);

    if (imagemCodificada.empty()) {
        cout << "Imagem nao carregou corretamente" << endl;
        return -1;
    }

    // Inicializando as matrizes clonadas
    imagemEscondida = Mat(imagemCodificada.size(), imagemCodificada.type());
    imagemPortadora = Mat(imagemCodificada.size(), imagemCodificada.type());
    imagemDecodificada = Mat(imagemCodificada.size(), imagemCodificada.type());

    // Realizando a decodificação da imagem;
    for (int i = 0; i < imagemCodificada.rows; i++) {
        for (int j = 0; j < imagemCodificada.cols; j++) {
            valCod = imagemCodificada.at<Vec3b>(i, j);

            valEsc[0] = valCod[0] << nbits;
            valEsc[1] = valCod[1] << nbits;
            valEsc[2] = valCod[2] << nbits;

            imagemEscondida.at<Vec3b>(i, j) = valEsc;

            valPort[0] = valCod[0] >> nbits << nbits;
            valPort[1] = valCod[1] >> nbits << nbits;
            valPort[2] = valCod[2] >> nbits << nbits;

            imagemPortadora.at<Vec3b>(i, j) = valPort;

            valDec[0] = valCod[0] >> (8 - nbits) << (8 - nbits);
            valDec[1] = valCod[1] >> (8 - nbits) << (8 - nbits);
            valDec[2] = valCod[2] >> (8 - nbits) << (8 - nbits);

            imagemDecodificada.at<Vec3b>(i, j) = valDec;
        }
    }
```

```
    imwrite("imagemEscondida.png", imagemEscondida);
    imwrite("imagemPortadora.png", imagemPortadora);
    imwrite("imagemDecodificada.png", imagemDecodificada);

    return 0;
}
```

Esse programa recupera a imagem codificada a partir dos bits menos significativos dos pixels da imagem esteganografada fornecida. Os bits 6 e 7 dos canais de cor de cada pixel são zerados para compor os bits mais significativos da imagem recuperada. Abaixo temos a imagem Decodificada imagemDecodificada, a imagem escondida imagemEscondida e a imagem portadora imagemPortadora, todas sendo saída do código esteg.

Figure 6. Saída do programa esteg.cpp, imagem Decodificada

Figure 7. Saída do programa esteg.cpp, imagem Portadora

Figure 8. Saída do programa esteg.cpp, imagem Escondida

4. Preenchendo regiões

Uma tarefa comum em processamento de imagens e visão computacional é contar a quantidade de objetos em uma cena. Para isso, é necessário identificar os aglomerados de pixels associados a cada objeto. Geralmente, trabalha-se com imagens binárias, em que cada pixel tem o valor 0 para o

fundo da imagem e 255 para os objetos. Cada aglomerado de pixels é considerado um objeto separado, e esse processo é amplamente utilizado para contagem de objetos em imagens.

4.1. Exercicio 4 - labeling.cpp

O programa "labeling.cpp" apresenta uma limitação quando há mais de 255 objetos na cena devido à utilização de pixels de 8 bits para rotular os objetos. Para solucionar esse problema, é necessário utilizar uma estrutura de dados com maior capacidade de armazenamento, como uma matriz de inteiros de 32 bits. Além disso, é possível aprimorar o algoritmo de contagem para identificar regiões com ou sem buracos internos, removendo os objetos que tocam as bordas da imagem e contando separadamente os buracos internos de cada objeto identificado.

4.1.1. Código e Resultado.

labeling.cpp

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

using namespace cv;

void removeBordas(Mat& image) {
    int width = image.cols;
    int height = image.rows;

    // Remover objetos das bordas
    for (int i = 0; i < width; i++) {
        floodFill(image, Point(i, 0), 255);
        floodFill(image, Point(i, height - 1), 255);
    }

    for (int i = 0; i < height; i++) {
        floodFill(image, Point(0, i), 255);
        floodFill(image, Point(width - 1, i), 255);
    }
}

void contarObjetos(Mat& image, int& nObjects, int& nBuracos) {
    int width = image.cols;
    int height = image.rows;

    nObjects = 0;
    nBuracos = 0;

    // Alterar o background
    floodFill(image, Point(0, 0), 100);

    // Contar objetos e buracos
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
```

```
            if (image.at<uchar>(i, j) == 0) {
                // Contar buraco
                nBuracos++;
                floodFill(image, Point(j, i), nBuracos);
            } else if (image.at<uchar>(i, j) == 255) {
                // Contar objeto
                nObjects++;
                floodFill(image, Point(j, i), 200);
            }
        }
    }
}

int main(int argc, char** argv) {
    Mat image = imread(argv[1], IMREAD_GRAYSCALE);

    if (!image.data) {
        std::cout << "Imagem nao carregou corretamente\n";
        return -1;
    }

    removeBordas(image);

    int nObjects, nBuracos;
    contarObjetos(image, nObjects, nBuracos);

    std::cout << "A figura tem " << nObjects << " bolhas\n";
    std::cout << "A figura tem " << nBuracos << " buracos\n";

    imshow("image", image);
    imwrite("labeling.png", image);
    waitKey();

    return 0;
}
```

O código labeling está contando certo o número de bolhas com buracos, mas parece que houve um problema na lógica utilizada para identificar as bolhas sem buracos. A abordagem utilizada não está correta e acabou removendo todas as bolhas sem buracos da imagem. Na imagem [fig_labeling], podemos ver melhor o que aconteceu

5. Manipulação de histogramas

5.1. Exercicio 5 - histogram.cpp

5.1.1. Código e Resultado.

6. Filtragem no domínio espacial I

6.1. Exercicio 6 - filtroespacial.cpp

6.1.1. Código e Resultado.

7. Filtragem no domínio espacial II

7.1. Exercicio 7 - addweighted.cpp

7.1.1. Código e Resultado.

8. A Tranformada Discreta de Fourier

8.1. Exercicio 8 - dftimage.cpp

8.1.1. Código e Resultado.