UNIVERSIDADE FEDERAL DO MARANHÃO – UFMA

CAMPUS DE SÃO LUÍS - CIDADE UNIVERSITÁRIA

BACHARELADO INTERDISCIPLINAR EM CIÊNCIA E TECNOLOGIA – BICT

Laboratório De Programação

TURMA: 01 (2023.02)

ALUNOS:

Emanuel Lopes Silva - 2021017818

Thales Aymar Fortes De Souza - 2021018145

Processamento de Imagem em C

PROFESSOR:

Thales Levi Azevedo Valente

SÃO LUÍS - MA

2023

Documento de Explicação do Código de Emanuel e Thales

- **Filtro Negativo:** O filtro busca aplicar o efeito "negativo" na imagem, invertendo suas cores. O Loop **"for (int i = 0; i < height * width * channels; i++)"** percorre cada byte da imagem. ele itera através de cada byte na imagem, pois a condição **"i < height * width * Channels"** garante que o loop percorra todos os bytes da imagem. Após isso, a parte **"img[i] = 255 - img[i]"** está presente. Ele realiza a operação de filtro negativo em cada byte da imagem. Subtrai-se o valor do byte atual de 255, que é o valor máximo possível para um byte (8 bits). Isso resulta no inverso do valor original, criando assim o efeito negativo, aplicando as cores inversas de cada byte na imagem.

- **Filtro Cinza:** O filtro busca converter efetivamente uma imagem colorida para uma versão em escala de cinza, deixando todos os canais em cor de cinza, com uma variação de iluminação. Inicialmente, temos o **"for (int k = 0; k < width * height * channels; k += channels)"**; Este é um loop que percorre cada pixel da imagem, avançando de acordo com o número de canais de cor. Isso garante que cada pixel seja processado corretamente. Logo após, temos a **"unsigned char Cinza = (unsigned char)(0.3 * img[k] + 0.59 * img[k + 1] + 0.11 * img[k + 2]);"** Aqui é aplicada a fórmula de conversão de cor RGB para escala de cinza. Os coeficientes 0.3, 0.59 e 0.11 representam as ponderações dos canais vermelho, verde e azul, respectivamente. Tal fórmula segue a ideia fornecida pelo próprio professor, Gray = 0.3*R + 0.59*G + 0.11*B. Após a variável "Cinza" receber o valor calculado em determinada iteração de "k", o seguinte código é aplicado: **" img[k] = Cinza;" , " img[k + 1] = Cinza;" , "img[k + 2] = Cinza;"** , essas linhas aplicam o valor calculado para o canal vermelho, verde e azul do pixel, transformando-o em um tom de cinza, assim finalizando a aplicação do filtro.

- **Filtro Blur:** Para se aplicar um filtro, aplica-se uma máscara que passa por todos os pixels da imagem e modifica-os de acordo com os pixels vizinhos. Esse novo valor do pixel é calculado aplicando uma média ponderada dos seus pixels vizinhos(um pixel é vizinho dele mesmo), em que os pesos são determinados pela máscara. Uma máscara também é chamada de Kernel. No caso do filtro blur, esse Kernel é dado por {(1,1,1), (1,1,1), (1,1,1)}, e como todos os pesos são 1, a divisão é feita por 9. Primeiramente, ocorre a inicialização do valor do elemento do kernel, usado para a operação de borramento **"float v = 1.0 / 9.0;"** Após isso, se inicializa o kernel float "**kernel[3][3] = {{v, v, v},{v, v, v}, {v, v, v}};**". Logo após, ocorre o processo de alocação de memória para a imagem temporária que será usada no processo de borramento, por meio do **"unsigned char *temp = malloc(width * height * 3)"** e depois ocorre a cópia a imagem original para a imagem temporária com o **"memcpy(temp, img, width * height * channels);"** Com isso, a aplicação do filtro blur começa, com um loop para percorrer os pixels da imagem, excluindo a borda representado por : **"for (int i = 1; i < height - 1; ++i) {for (int j = 1; j < width - 1;**

++j) {”, Dentro desse loop, ocorre o loop para processar os canais de cor (R, G, B) , visto como : **“for (int k = 0; k < 3; ++k) {“** e dentro deste loop, a variável soma, que é a variável para armazenar a soma para cada pixel, é declarada ao valor zero. Esse processo é representado por **“float soma = 0;”**. Após isso, têm-se o loop para percorrer a vizinhança do pixel atual e aplicar o kernel, escrito como **“for (int di = -1; di <= 1; ++di) { for (int dj = -1; dj <= 1; ++dj) {”**, e dentro do loop, o processo de obtenção da soma e aplicação do Kernel na soma ocorre, borrando a "imagem temporária", com o código **“soma += kernel[dj + 1][di + 1] * temp[((i + di) * width + (j + dj)) * 3 + k];”** Após obter o valor da soma, se atribui o valor calculado à imagem original após arredondar o resultado, por meio do código: **“img[(i * width + j) * 3 + k] = (unsigned char)round(soma);”**, após isso, os loops são fechados, e o filtro blur foi aplicado a imagem original, quando todos os loops de i e j forem terminados.

- **Filtro Sobel:** O filtro Sobel só funciona com a aplicação do Filtro cinza previamente. Então, todas as modificações citadas acima, da parte do Filtro Cinza, serão aplicadas à imagem antes do código a seguir. Inicialmente, se tem **“int soma1; int soma2;”** , que serve para a declaração de variáveis para armazenar as somas dos produtos com os kernels Sobel, sendo “soma 1” para o Kernel horizontal e a “soma2” para o kernel vertical. Após isso, temos a inicialização dos kernels Sobel para detecção de bordas horizontal e vertical, por meio do código:
  “float kernelx[3][3]={{ -1, 0, 1 },{ -2, 0, 2 },{ -1, 0, 1 }};”
  “float kernely[3][3]={{ -1, -2, -1 },{ 0, 0, 0 },{ 1, 2, 1 }};”
  Após isso, ocorre a inicialização de imagens temporárias para o processo de soma e aplicação do Sobel : **“unsigned char *temp = malloc(width * height * 3); unsigned char *temp2 = malloc(width * height * 3);”.** Logo após, ocorre a cópia a imagem original para as temporárias, por meio da **“for (int i = 0; i < width * height * 3; ++i) {temp[i] = img[i]; } for (int i = 0; i < width * height * 3; ++i) {temp2[i] = img[i];}”**, por meio de 2 loops que percorrem cada byte da imagem, um loop para a imagem temporária 1 e o segundo loop para a imagem temporária 2. Logo após a cópia da imagem original para as temporárias, o processo de aplicação do filtro sobel começa, com um loop para percorrer os pixels da imagem, excluindo a borda representado por : **“for (int i = 1; i < height - 1; ++i) {for (int j = 1; j < width - 1; ++j) {”**, Dentro desse loop, ocorre o loop para processar os canais de cor (R, G, B) , visto como : **“for (int k = 0; k < 3; ++k) {“** e dentro deste loop, as variáveis somas declaradas mais cedo são atribuídas ao valor zero. Ademais, cria-se o loop para percorrer a vizinhança do pixel atual, por meio do código **“for (int di = -1; di <= 1; ++di) { for (int dj = -1; dj <= 1; ++dj) {”** e dentro deste loop, obtém-se o valor do pixel da vizinhança, por meio do **“temp[((i + di) * width + (j + dj)) * 3 + k];”**, e os kernels Sobel horizontal e vertical são aplicados. O kernel Sobel horizontal é aplicado na soma1 e o vertical a soma2, por meio de uma multiplicação do kernel com o valor do pixel da vizinhança : **“soma1 += kernelx[dj+1][di+1] * temp[((i + di)* width + (j + dj)) * 3 + k];**
  soma2 += kernely[dj+1][di+1] * temp[((i + di)* width + (j + dj)) * 3 + k];” Após a aplicação do Kernel Sobel, ocorre o cálculo da magnitude da borda usando a

distância euclidiana **“int magnitude = (int)sqrt((soma1 * soma1) + (soma2 * soma2));”** e logo após a limitação da magnitude a 255 bytes (valor máximo do byte): **“if (magnitude > 255) { magnitude = 255;}”** após isso, começa-se o processo de atribuir o valor calculado na imagem temporária para o canal k, por meio do código: **“temp2[(i * width + j) * 3 + k] = (unsigned char)magnitude;”**, e logo após, a atribuição da imagem temporária na imagem original, aplicando assim o valor da magnitude na imagem original, assim aplicando o filtro Sobel, no código **“img[(i * width + j) * 3 + k] = temp2[(i * width + j) * 3 + k];”**. Dessa forma, o Filtro Sobel foi aplicado na imagem original, ao receber as informações guardadas na imagem temporária 2.