Fundamentos de Processamento de Imagens

Trabalho 1

Nome: João Pedro Cosme da Silva / Cartão 00314792

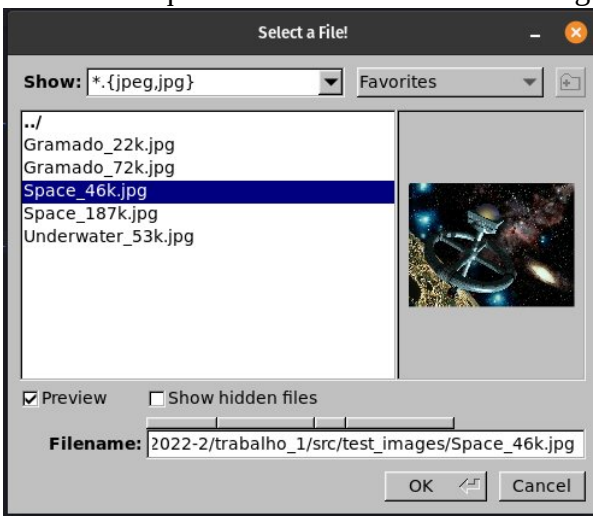
Resumo

Este trabalho tem como objetivo a familiarização com conceitos vistos nas primeiras aulas da disciplina de Fundamentos de Processamento de Imagens. O desenvolvimento deste trabalho auxiliou tanto na obtenção de conhecimento teórico sobre as operações ponto a ponto vistas em aula, como também uma interação em primeira mão com o desenvolvimento dessas funções e manipulação dessas estruturas através de código. Para o desenvolvimento do trabalho, utilizei a linguagem de programação Rust, com o propósito de aprender esta linguagem que, por seus conceitos de *ownership* (como alternativa para coletor de lixo ou alocação de memória manual), que facilitam o gerenciamento de memória, por sua velocidade, visto que é uma linguagem compilada, e por sua confiança e popularidade recente (tendo já sido habilitada no kernel do Linux a partir de 2022). A seguir, serão descritas as funcionalidades desenvolvidas e seu uso.

Parte 1 – Leitura e Gravação de Arquivos de Imagens

a. Ler e gravar arquivos de imagens;

Para esta etapa, é apresentada a versão final do programa desenvolvido. Ao inicia-lo, o usuário é apresentado ao um *prompt* solicitando que o mesmo selecione uma imagem para ser modificada:



Neste caso, a imagem é carregada para a edição e em seguida é exibida:

Já na comparação entre os arquivos base, podemos tomar como exemplo o arquivo *Space_187k.jpg*. Originalmente este arquivo possui o tamanho de 191.3KB, enquanto sua contrapartida salva pelo programa desenvolvido, possui 54.3KB. Esta diferença de tamanho pode ser explicada, esta diferença se dá devido ao algoritmo de compressão JPEG que, por se tratar de um algoritmo *lossy*, se compromete a perder a qualidade em troca de um menor tamanho final de arquivo. Mesmo pelo arquivo original já ser um JPEG, a biblioteca implementada executa novamente este algortimo para ser salvado, gerando então essa diferença.

Parte II – Leitura, Exibição e Operação sobre Imagens

a) Espalhamento Horizontal e Vertical

Na interface de meu programa, há um botão para cada um dessas ações, que permite executar ambas as ações, seguem abaixo exemplos de resultados de operações:

Flip Horiontal

Flip Vertical

Para o espalhamento horizontal, foi utilizado o seguinte algoritmo onde, como sabemos que é uma inversão horizontal e, como cada pixel ira ser substituido pelo seu oposto, podemos executar um loop onde executamos ao mesmo tempo a inserção de dois pixeis na imagem de saida:

```rust
fn horizontal_flip(image: &ImageBuffer<Rgb<u8>, Vec<u8>>) -> ImageBuffer<Rgb<u8>, Vec<u8>> {
    let width: u32 = image.width();
    let half: u32 = width / 2;
    let mut output: ImageBuffer<Rgb<u8>, Vec<u8>> = ImageBuffer::new(width, image.height());
    for x: u32 in 0..half {
        for y: u32 in 0..image.height() {
            output.put_pixel(x, y, pixel: image.get_pixel(x: width - x - 1 as u32, y).clone());
            output.put_pixel(x: width - x - 1, y as u32, pixel: image.get_pixel(x, y).clone());
        }
    }
    return output;
}
```

Já para o espelhamento vertical, seguindo a mesma filosofia, o algoritmo é o seguinte:

```rust
fn vertical_flip(image: &ImageBuffer<Rgb<u8>, Vec<u8>>) -> ImageBuffer<Rgb<u8>, Vec<u8>> {
    let width = image.width();
    let height = image.height();
    let mut output: ImageBuffer<Rgb<u8>, Vec<u8>> = ImageBuffer::new(width, image.height());
    for x in 0..width {
        for y in 0..height / 2 {
            output.put_pixel(x, y, image.get_pixel(x, height - 1 - y).clone());
            output.put_pixel(x, height - y - 1 as u32, image.get_pixel(x, y).clone());
        }
    }
    return output;
}
```

Em ambos os casos, considerando que Rust é uma linguagem compilada, temos uma execução praticamente instantânea para todas as imagens do conjunto de teste. Além disso, por fazermos uma troca direta entre pares de pixeis opostos no eixo desejado (seja horizontal ou vertical), temos a garantia de que imagens com cardinalidade impar em um dos eixos estará seguramente coberta, já que os pixeis no centro não serão afetados.

b) Conversão de imagem colorida para tons de cinza (luminância).

Conforme visto, a interface oferece um botão que aplica o cálculo de luminancia sobre a imagem base, neste caso, o resultado é o seguinte:

O cálculo de luminância foi feito conforme o algoritmo utilizado na definição do trabalho:

```
fn to_grayscale(pixels: &[u8; 3]) -> u8 {
    let red = pixels[0] as f64;
    let green = pixels[1] as f64;
    let blue = pixels[2] as f64;

    let new_val = 0.299 * red + 0.587 * green + 0.114 * blue;
    let new_val = new_val as u8;
    return new_val;
}
```

Para o caso do cálculo seja aplicado em uma imagem já do tipo Ri = Gi = Bi = Li, nada irá mudar, já que a equação dada por :

L = 0.299*R + 0.587*G + 0.114*B

Será reduzida a:

L = 0.299*L + 0.587*L + 0.114*L

L = L

Dessa forma, sucessivas aplicações do cálculo de luminância não irão afetar uma imagem em tons de cinza.

c) Quantização (de tons) sobre as imagens em tons de cinza

Na interface apresentada anteriormente, podemos ver que existe um botão para realizar a quantização da imagem baseada em um número de entrada que representa o numero de tons a ser utilizados. A seguir, alguns exemplos de Quantização:

Quantização com 15 tons

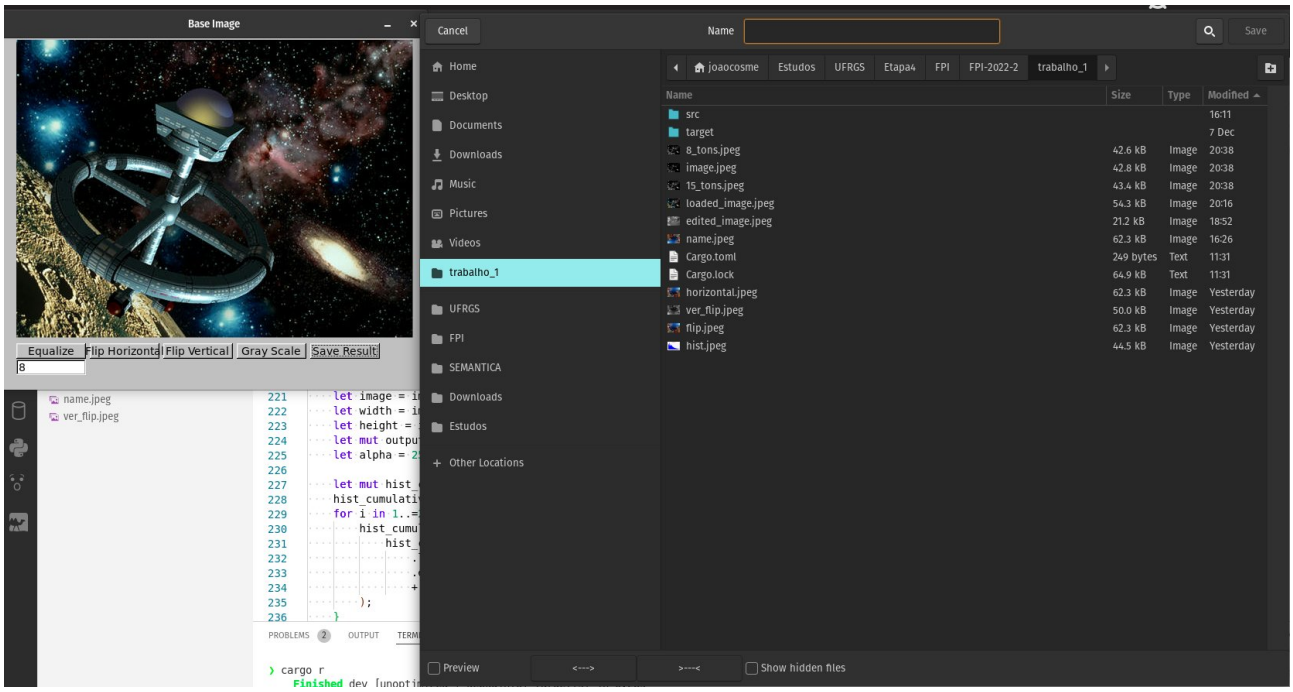
Aqui, já podemos perceber um maior ruido em áreas anteriormente suaves na imagem base, isto ocorre devido a redistribuição de tons em um numero menor de bins (que, para a representação acurrada de 256 tons, irá ter um contraste maior entre bins próximos se comparado a um histograma com 256 bins).

Quantização com 8 Tons

Com 8 tons, o efeito de ruído se torna ainda mais perceptível. Diversas areas da imagem já apresentam praticamente nenhuma suavização de tons e borrões se tornam comuns.

d) Salvamento da imagem resultante das operações realizadas em um arquivo JPEG.

Ao fim das edições, é possivel utilizar o botão de Salvar para iniciar uma tela de escolha de diretorio e nome do arquivo para que o resultado de nossas operações possa ser salvo:



Conclusão

Ao longo deste trabalho, tive a oportunidade de implementar as operações propostas e ganhar um entendimento mais profundo sobre elas. Algumas mais simples, como cálculo de luminância e espelhamento horizontal foram fáceis devido a naturalidade de suas operações. Já a quantização de imagens foi um desafio maior visto que combina diversas possibilidades: cálculo de histograma, histograma cumulativo e ainda a tradução para um número de tons diferente do original.

Outra dificuldade enfrentada foi a novidade de aprender Rust, que apresenta suas peculiaridades, junto com o desenvolvimento de GUIs utilizando-a. Neste trabalho, usei a biblioteca FLTK, e tive algumas dificuldades com sua sintaxe e maneira de declarar ações, e planejo refatorar uma boa parte do código de interface para a próxima etapa.