Relatório Alquimia em C

Caio Ferreira, Pedro Hartmann
Faculdade de Ciência da Computação — PUCRS

28 de abril de 2024

Resumo

O primeiro trabalho da disciplina de Programação de Baixo Nível requer que os alunos construam um código em C que receba duas imagens como parâmetro, retirando as cores da primeira imagem (“origem”) e montando uma terceira imagem (“saída”) com base na segunda imagem (“desejada”). Para isso, foi implementado a biblioteca “SOIL”, cuja permite o desenvolvimento do código através do uso de suas propriedades. Este artigo reporta 2 soluções encontradas, junto de algumas sugestões de melhorias para o código apresentado.

Introdução

O problema é introduzido aos alunos explicando a origem e a definição de alquimia. Dentro dessa primeira seção, é esclarecido que as imagens utilizadas para o processo sempre terão a mesma área, portanto, terão a mesma quantidade de pixels quando representadas digitalmente. Em seguida, é explicado o uso da biblioteca SOIL, esclarecendo que cada pixel da imagem é representado por três caracteres sem sinal: r, g e b (correspondentes aos valores de vermelho, verde e azul, respectivamente).

O código base fornecido apresenta dois structs para uso no programa: “RGBpixel”, o qual contém os três caracteres sem sinal r g b; e “Img”, contendo dois valores inteiros correspondentes à largura e à altura da imagem, e um ponteiro RGBpixel, caracterizando um vetor de pixels. Esse vetor armazena todos os pixels e seus valores rgb presentes na imagem, facilitando a comparação entre duas imagens.

Além disso, o código base também possui um método “cmp”, que recebe dois ponteiros de um tipo qualquer (void). Dentro do método, é imposto que esses ponteiros são do tipo RGBpixel, e seus valores rgb são atribuídos a variáveis dentro da função a fim de determinar qual dos dois pixels é mais escuro. A partir dessas variáveis, é executada uma série de comparações entre os valores, determinando se um pixel é mais ou menos escuro que o outro. A utilidade desse método está na implementação da função “qsort”[1], fornecida pela biblioteca stdlib.io da própria linguagem C.

Para solucionarmos o problema, é necessário realizar várias comparações entre as duas imagens fornecidas por parâmetro (“origem” e “desejada”), tendo que o programa deve encontrar o pixel mais próximo em “origem” do correspondente em “desejada”. Após caminhar a imagem

desejada e encontrar os pixels correspondentes na origem, é montada uma terceira imagem de “saída” utilizando esses pixels, concluindo o processo de alquimia.

Primeira solução

Após pensarmos sobre como resolver o problema, decidimos fazer uma solução custosa em termos de eficiência, mas que nos garantiria um resultado correto. Sem a utilização da função de qsort, o programa cria um vetor de RGBpixels (com tamanho igual à área das imagens) e um RGBpixel temporário. Após isso, são executados dois laços de repetição aninhados, onde o primeiro percorre a imagem desejada e o segundo percorre a imagem de origem, comparando as somas dos valores de vermelho, verde e azul dos dois pixels, armazenando o pixel com o menor resultado da comparação na variável temporária e adicionando-o ao vetor. Ao fim da execução dos laços de repetição, a imagem de saída recebia o vetor criado e a imagem era demonstrada na janela.

O resultado gerado por esse algoritmo é reconhecível, no entanto, as colorações são muito distorcidas, pois o método de comparação encontra um pixel da cor errada mas com uma comparação igual a zero (fazendo o programa interpretar como se os pixels fossem iguais), colocando esse pixel na imagem erroneamente. Além disso, conforme foi mencionado, essa implementação é extremamente custosa, tendo que percorrer a imagem de origem uma quantidade correspondente à área das imagens. Em casos menores, o tempo de execução não é muito significativo, mas piora proporcionalmente à área das imagens. Isso poderia ser melhorado com busca binária e o uso da função de quicksort, utilizadas na próxima solução.

Segunda solução

Para a segunda solução, tentamos arrumar dois problemas da primeira solução: o programa identificando 2 pixels diferentes como iguais, e o fato de o algoritmo não ser eficiente. A fonte do primeiro problema era que tinhamos tentado implementar uma função de comparação que funcionaria com uma outra situação agora descartada, e resolvemos utilizar a função fornecida com o código base. Para resolver a questão da eficiência, achamos que seria uma boa ideia chamar a função qsort na imagem de origem das cores, e realizar uma pesquisa binária na imagem ordenada para achar o pixel mais próximo. Após rodar o programa, vimos que o resultado é muito parecido com o exemplo fornecido no enunciado do problema, e a eficiência do programa aumentou dramaticamente.

```c
Img sorted = pic[ORIGEM];
sorted.pixels = calloc(tam, sizeof(RGBpixel));
memcpy(sorted.pixels, pic[ORIGEM].pixels, sizeof(RGBpixel) * tam);

// Realiza quicksort em sorted.pixels
qsort(sorted.pixels, tam, sizeof(RGBpixel), cmp);

RGBpixel *ptra = pic[DESEJ].pixels;
RGBpixel montagem[tam];
clock_t start = clock();
// percorre pic[DESEJ].pixels
for (int i = 0; i < tam; i++, ptra++) {

    int low = 0, high = tam, result;
    while (low < high) {
        result = low + ((high - low) / 2);

        int found = 0;
        switch (cmp(ptra, &sorted.pixels[result])) {
            case 0:
                found = 1;
                break;
            case 1:
                low = result + 1;
                break;
            case -1:
                high = result - 1;
                break;
        }
        if (found) {
            break;
        }
    }

    montagem[i] = sorted.pixels[result];
```

Porém, a implementação atual ainda possui um problema que não conseguimos identificar a fonte e nem a solução: imagem de saída possui “manchas” de uma cor incorreta em alguns lugares, deixando o resultado um pouco distorcido, conforme exemplificado na imagem a seguir.

Resultado da transmutação entre Mona Lisa e Persistence of Memory usando o algoritmo em Solução 2.

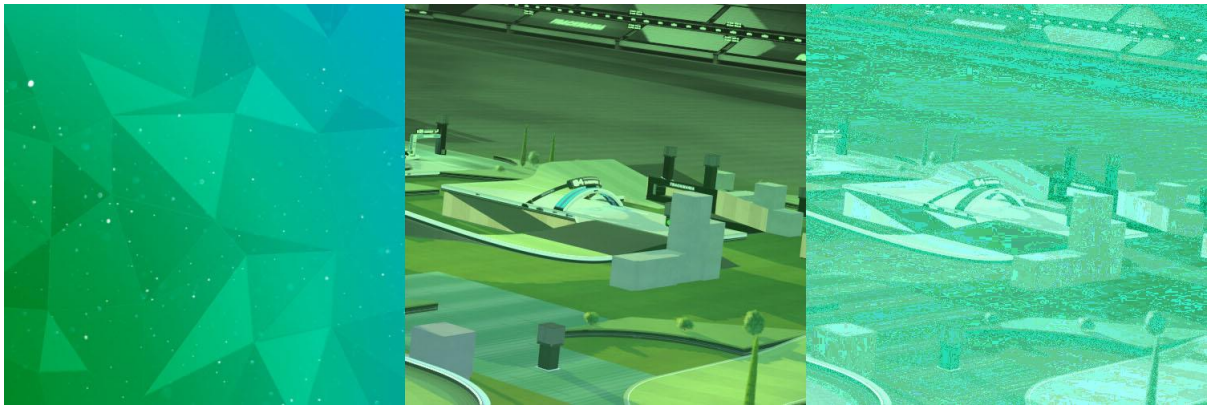
Pode-se observar que existem pixels vermelhos em meio aos azuis.

Resultados

Imagem da esquerda: origem, utiliza-se apenas suas cores.

Imagem do meio: desejada, imagem a ser formada.

Imagem da direita: saída, montagem da segunda imagem usando as cores da primeira.



Conclusões

Ao analisar os resultados dos testes com outras imagens, pode-se observar que ainda existe muito ruído nas imagens de saída. Após a análise do resultados, acreditamos que a causa possa estar relacionada ao modo em que é realizada a comparação entre pixels quando as cores são muito distintas (como pode-se perceber no terceiro exemplo em Resultados, ou nos pixels vermelhos na figura da segunda solução), fazendo com que o algoritmo armazene o pixel encontrado no meio do quicksort. Entretanto, infelizmente, não soubemos solucionar o problema.

Em questões de melhoria do algoritmo, a sua eficácia encontra-se boa, pois, em comparação com a primeira solução, a segunda solução é muito melhor em relação ao tempo de execução, uma vez que a primeira demorava quase 10 segundos para o processo, e a segunda o faz em menos de meio segundo.

Para o problema da comparação dos pixels, pensamos em uma possível solução a qual não conseguimos implementar:

1. Altera-se o struct RGBpixel para que armazene um int index, correspondente à sua posição no vetor de pixels.
2. Realiza-se quicksort nas imagens de origem e desejada.
3. Compara-se os pixels entre os arrays ordenados.
4. Monta-se a imagem de saída com base nas comparações.

Não estamos certos se essa ideia funcionaria, mas, além de possivelmente melhorar a eficiência do algoritmo ainda mais, poderia arrumar o problema da comparação entre pixels.

Referências

[1] Microsoft Learn, “qsort”. Disponível em: https://learn.microsoft.com/pt-br/cpp/c-runtime-library/reference/qsort?view=msvc-170