Este é o CS50x

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://orcid.org/0000-0001-5338-2522)
(https://www.quora.com/profile/David-J-Malan)
(https://www.reddit.com/user/davidjmalan) (https://twitter.com/davidjmalan)

Filtro

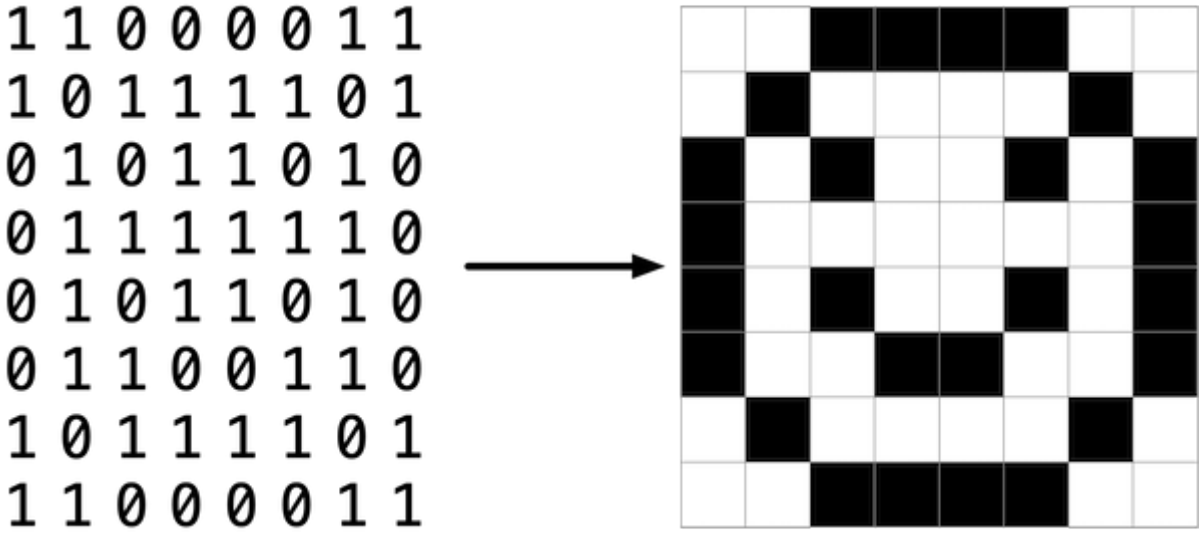
Implemente um programa que aplique filtros a BMPs, conforme a seguir.

```
$ ./filter -r image.bmp reflected.bmp
```

Fundo

Bitmaps

Talvez a maneira mais simples de representar uma imagem seja com uma grade de pixels (ou seja, pontos), cada um dos quais pode ser de uma cor diferente. Para imagens em preto e branco, precisamos, portanto, de 1 bit por pixel, já que 0 pode representar preto e 1 pode representar branco, como mostrado a seguir.



Nesse sentido, então, uma imagem é apenas um bitmap (ou seja, um mapa de bits). Para imagens mais coloridas, você simplesmente precisa de mais bits por pixel. Um formato de arquivo (como BMP (https://en.wikipedia.org/wiki/BMP_file_format) , JPEG (https://en.wikipedia.org/wiki/JPEG) ou PNG (https://en.wikipedia.org/wiki/Portable_Network_Graphics)) que suporta "cores de 24 bits" usa 24 bits por pixel. (BMP, na verdade, suporta cores de 1, 4, 8, 16, 24 e 32 bits.)

Um BMP de 24 bits usa 8 bits para significar a quantidade de vermelho na cor de um pixel, 8 bits para significar a quantidade de verde na cor de um pixel e 8 bits para significar a quantidade de azul na cor de um pixel. Se você já ouviu falar em cores RGB, bem, aí está: vermelho, verde, azul.

Se o R, G e B valores de alguns pixel em uma BMP são, digamos, `0xff`, `0x00`, e `0x00` em hexadecimal, que pixel é puramente vermelho, como `0xff` (também conhecido como `255` em decimal) implica "um monte de vermelho", enquanto `0x00` e `0x00` implicam "Sem verde" e "sem azul", respectivamente.

Um pouco (mapa) Mais Técnico

Lembre-se de que um arquivo é apenas uma sequência de bits, organizados de alguma forma. Um arquivo BMP de 24 bits, então, é essencialmente apenas uma sequência de bits, (quase) cada 24 dos quais representam a cor de algum pixel. Mas um arquivo BMP também contém alguns "metadados", informações como a altura e a largura de uma imagem. Esses metadados são armazenados no início do arquivo na forma de duas estruturas de dados geralmente chamadas de "cabeçalhos", não devem ser confundidos com os arquivos de cabeçalho de C. (Aliás, esses cabeçalhos evoluíram com o tempo. Esse problema usa a versão mais recente do formato BMP da Microsoft, 4.0, que estreou com o Windows 95.)

O primeiro desses cabeçalhos, chamado `BITMAPFILEHEADER`, tem 14 bytes de comprimento. (Lembre-se de que 1 byte é igual a 8 bits.) O segundo desses cabeçalhos, chamado `BITMAPINFOHEADER`, tem 40 bytes de comprimento. Imediatamente após esses cabeçalhos está o bitmap real: uma matriz de bytes, triplos dos quais representam a cor de um pixel. No entanto, o BMP armazena esses triplos ao contrário (ou seja, como BGR), com 8 bits para o azul, seguidos por 8 bits para o verde, seguidos por 8 bits para o vermelho. (Alguns BMPs também armazenam todo o bitmap de trás para frente, com a linha superior de uma imagem no final do arquivo BMP. Mas armazenamos os BMPs desse conjunto de problemas conforme descrito aqui, com cada linha superior do bitmap primeiro e a linha inferior por último.) palavras, se convertêssemos o smiley de 1 bit acima em um smiley de 24 bits, substituindo o vermelho por preto, um BMP de 24 bits armazenaria este bitmap da seguinte forma, onde `0000ff` significa vermelho e `ffffff` significa branco; destacamos em vermelho todas as instâncias de `0000ff`.

```
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
```

Como apresentamos esses bits da esquerda para a direita, de cima para baixo, em 8 colunas, você pode realmente ver o emoticon vermelho se der um passo para trás.

Para ser claro, lembre-se de que um dígito hexadecimal representa 4 bits. Conseqüentemente, `ffffff` em hexadecimal realmente significa `111111111111111111111111` em binário.

Observe que você pode representar um bitmap como uma matriz bidimensional de pixels: onde a imagem é uma matriz de linhas, cada linha é uma matriz de pixels. Na verdade, é assim que escolhemos representar imagens de bitmap neste problema.

Filtragem de Imagens

O que significa filtrar uma imagem? Você pode pensar em filtrar uma imagem como pegar os pixels de alguma imagem original e modificar cada pixel de forma que um efeito específico seja aparente na imagem resultante.

Tons de Cinza

Um filtro comum é o filtro "escala de cinza", onde pegamos uma imagem e queremos convertê-la em preto e branco. Como isso funciona?

Lembre-se de que se os valores de vermelho, verde e azul estiverem todos definidos como `0x00` (hexadecimal para `0`), o pixel é preto. E se todos os valores forem definidos como `0xff` (hexadecimal para `255`), o pixel é branco. Contanto que os valores de vermelho, verde e azul sejam todos iguais, o resultado será tons de cinza variados ao longo do espectro preto e branco, com valores mais altos significando tons mais claros (mais perto de branco) e valores mais baixos significando tons mais escuros (mais perto de Preto).

Portanto, para converter um pixel em tons de cinza, só precisamos ter certeza de que os valores de vermelho, verde e azul são todos iguais. Mas como sabemos que valor devemos criá-los? Bem, é provavelmente razoável esperar que, se os valores originais de vermelho, verde e azul fossem todos muito altos, o novo valor também deveria ser muito alto. E se os valores originais fossem todos baixos, o novo valor também deveria ser baixo.

Na verdade, para garantir que cada pixel da nova imagem ainda tenha o mesmo brilho ou escuridão geral da imagem antiga, podemos obter a média dos valores de vermelho, verde e azul para determinar qual tom de cinza deve ser criado para o novo pixel.

Se você aplicar isso a cada pixel da imagem, o resultado será uma imagem convertida em tons de cinza.

Sépia

A maioria dos programas de edição de imagem oferece suporte a um filtro "sépia", que dá às imagens uma aparência antiga, fazendo com que toda a imagem pareça um pouco marrom-avermelhada.

Uma imagem pode ser convertida em sépia tomando cada pixel e computando novos valores de vermelho, verde e azul com base nos valores originais dos três.

Existem vários algoritmos para converter uma imagem em sépia, mas, para esse problema, pediremos que você use o seguinte algoritmo. Para cada pixel, os valores da cor sépia devem ser calculados com base nos valores da cor original conforme a seguir.

```
sepiaRed = .393 * originalRed + .769 * originalGreen + .189 * originalBlue
sepiaGreen = .349 * originalRed + .686 * originalGreen + .168 * originalBl
sepiaBlue = .272 * originalRed + .534 * originalGreen + .131 * originalBlu
```

Obviamente, o resultado de cada uma dessas fórmulas pode não ser um número inteiro, mas cada valor pode ser arredondado para o número inteiro mais próximo. Também é possível que o resultado da fórmula seja um número maior que 255, o valor máximo para um valor de cor

de 8 bits. Nesse caso, os valores de vermelho, verde e azul devem ser limitados a 255. Como resultado, podemos garantir que os valores de vermelho, verde e azul resultantes serão números inteiros entre 0 e 255, inclusive.

Reflexão

Alguns filtros também podem mover pixels. Refletir uma imagem, por exemplo, é um filtro em que a imagem resultante é o que você obteria colocando a imagem original na frente de um espelho. Portanto, quaisquer pixels no lado esquerdo da imagem devem terminar no lado direito e vice-versa.

Observe que todos os pixels originais da imagem original ainda estarão presentes na imagem refletida, mas esses pixels podem ter sido reorganizados para estar em um local diferente na imagem.

Borrão

Existem várias maneiras de criar o efeito de desfocar ou suavizar uma imagem. Para este problema, usaremos o “desfoque de caixa”, que funciona pegando cada pixel e, para cada valor de cor, dando a ele um novo valor calculando a média dos valores de cor dos pixels vizinhos.

Considere a seguinte grade de pixels, onde numeramos cada pixel.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

O novo valor de cada pixel seria a média dos valores de todos os pixels que estão dentro de 1 linha e coluna do pixel original (formando uma caixa 3x3). Por exemplo, cada um dos valores de cor para o pixel 6 seria obtido pela média dos valores de cor originais dos pixels 1, 2, 3, 5, 6, 7, 9, 10 e 11 (observe que o próprio pixel 6 está incluído no média). Da mesma forma, os

valores de cor para o pixel 11 seriam obtidos pela média dos valores de cor dos pixels 6, 7, 8, 10, 11, 12, 14, 15 e 16.

Para um pixel ao longo da borda ou canto, como o pixel 15, ainda procuraríamos todos os pixels em 1 linha e coluna: neste caso, os pixels 10, 11, 12, 14, 15 e 16.

Começando

Veja como baixar o "código de distribuição" desse problema (ou seja, código inicial) em seu próprio CS50 IDE. Faça login no CS50 IDE (https://ide.cs50.io/) e, em uma janela de terminal, execute cada um dos itens (https://ide.cs50.io/) abaixo.

- Execute `cd ~` (ou simplesmente `cd` sem argumentos) para garantir que você está em seu diretório inicial.
- Execute `mkdir pset4` para fazer (ou seja, criar) um diretório chamado `pset4`.
- Execute `cd pset4` para mudar para (ou seja, abrir) esse diretório.
- Execute `wget https://cdn.cs50.net/2020/fall/psets/4/filter/less/filter.zip` para baixar um arquivo ZIP (compactado) com a distribuição desse problema.
- Execute `unzip filter.zip` para descompactar esse arquivo.
- Execute `rm filter.zip` seguido por `yes` ou `y` para excluir esse arquivo ZIP.
- Execute `ls`. Você deve ver um diretório chamado `filter`, que estava dentro desse arquivo ZIP.
- Execute `cd filter` para mudar para esse diretório.
- Execute `ls`. Você deverá ver a distribuição deste problema, incluindo `bmp.h`, `filter.c`, `helpers.h`, `helpers.c`, e `Makefile`. Você também verá um diretório chamado `images`, com algumas imagens de bitmap de amostra.

Entendimento

Vamos agora dar uma olhada em alguns dos arquivos fornecidos a você como código de distribuição para entender o que há dentro deles.

`bmp.h`

Abra `bmp.h` (clicando duas vezes no navegador de arquivos) e dê uma olhada.

Você verá as definições dos cabeçalhos que mencionamos (`BITMAPINFOHEADER` e `BITMAPFILEHEADER`). Além disso, que define arquivos `BYTE`, `DWORD`, `LONG`, e `WORD`, tipos de dados normalmente encontrados no mundo da programação do Windows. Observe como eles são apenas apelidos para primitivos com os quais você (espero) já esteja familiarizado. Parece que `BITMAPFILEHEADER` e `BITMAPINFOHEADER` fazer uso desses tipos.

Talvez o mais importante para você, este arquivo também define um `struct` chamado `RGBTRIPLE` que, simplesmente, "encapsula" três bytes: um azul, um verde e um vermelho (a ordem, recorde, em que esperamos encontrar triplos RGB realmente no disco) .

Por que eles são `struct` úteis? Bem, lembre-se de que um arquivo é apenas uma sequência de bytes (ou, no final das contas, bits) no disco. Mas esses bytes são geralmente ordenados de forma que os primeiros representem algo, os próximos representem outra coisa e assim por diante. Os “formatos de arquivo” existem porque o mundo padronizou o significado de bytes. Agora, poderíamos simplesmente ler um arquivo do disco para a RAM como um grande array de bytes. E poderíamos apenas lembrar que o byte em `array[i]` representa uma coisa, enquanto o byte em `array[j]` representa outra. Mas por que não dar nomes a alguns desses bytes para que possamos recuperá-los da memória com mais facilidade? É exatamente isso que as estruturas nos `bmp.h` permitem fazer. Em vez de pensar em algum arquivo como uma longa sequência de bytes, podemos pensar nele como uma sequência de `struct` s.

`filter.c`

Agora, vamos abrir `filter.c`. Este arquivo já foi escrito para você, mas há alguns pontos importantes que devem ser observados aqui.

Primeiro, observe a definição de `filters` na linha 11. Essa seqüência diz ao programa que os argumentos de linha de comando admissível para o programa são: `b`, `g`, `r`, e `s`. Cada um deles especifica um filtro diferente que podemos aplicar às nossas imagens: desfoque, tons de cinza, reflexo e sépia.

As próximas linhas abrem um arquivo de imagem, certifique-se de que seja realmente um arquivo BMP e leia todas as informações de pixel em um array 2D chamado `image`.

Role para baixo até a `switch` instrução que começa na linha 102. Observe que, dependendo do que `filter` escolhemos, uma função diferente é chamada: se o usuário escolher o filtro `b`, o programa chama a `blur` função; se `g`, então `grayscale` é chamado; se `r`, então `reflect` é chamado; e se `s`, então, `sepia` é chamado. Observe também que cada uma dessas funções toma como argumentos a altura da imagem, a largura da imagem e a matriz 2D de pixels.

Estas são as funções que você (em breve!) Implementará. Como você pode imaginar, o objetivo é que cada uma dessas funções edite o array 2D de pixels de forma que o filtro desejado seja aplicado à imagem.

As linhas restantes do programa pegam o resultado `image` e os gravam em um novo arquivo de imagem.

`helpers.h`

A seguir, dê uma olhada em `helpers.h`. Este arquivo é bastante curto e fornece apenas os

A seguir, dê uma olhada em `helpers.h`. Este arquivo é bastante curto e fornece apenas os protótipos de funções para as funções que você viu anteriormente.

Aqui, observe o fato de que cada função recebe um array 2D chamado `image` como argumento, onde `image` é um array de `height` muitas linhas e cada linha é em si um outro array de

`width` muitos `RGBTRIPLE`s. Portanto, se `image` representa a imagem inteira, `image[0]` representa a primeira linha e `image[0][0]` representa o pixel no canto superior esquerdo da imagem.

`helpers.c`

Agora, abra `helpers.c`. É aqui que `helpers.h` pertence a implementação das funções declaradas em . Mas note que, agora, as implementações estão faltando! Esta parte é com você.

`Makefile`

Finalmente, vamos dar uma olhada `Makefile`. Este arquivo especifica o que deve acontecer quando executamos um comando de terminal como `make filter`. Considerando que os programas que você pode ter escrito antes estavam confinados em apenas um arquivo, `filter` parece usar vários arquivos: `filter.c`, `bmp.h`, `helpers.h`, e `helpers.c`. Portanto, precisaremos dizer `make` como compilar esse arquivo.

Tente compilar `filter` por si mesmo indo para o seu terminal e executando

```
$ make filter
```

Em seguida, você pode executar o programa executando:

```
$ ./filter -g images/yard.bmp out.bmp
```

que pega a imagem em `images/yard.bmp` e gera uma nova imagem chamada `out.bmp` após executar os pixels por meio da `grayscale` função. `grayscale` ainda não faz nada, portanto, a imagem de saída deve ser igual à do jardim original.

Especificação

Implemente as funções de `helpers.c` forma que um usuário possa aplicar filtros de escala de cinza, sépia, reflexão ou desfoque às imagens.

- A função `grayscale` deve pegar uma imagem e transformá-la em uma versão em preto e branco da mesma imagem.
- A função `sepia` deve pegar uma imagem e transformá-la em uma versão sépia da mesma imagem.

- A `reflect` função deve pegar uma imagem e refleti-la horizontalmente.
- Finalmente, a `blur` função deve pegar uma imagem e transformá-la em uma versão desfocada da mesma imagem.

Você não deve modificar nenhuma das assinaturas de função, nem deve modificar nenhum outro arquivo que não seja `helpers.c`.

Passo a passo

Observe que existem 5 vídeos nesta lista de reprodução.

Uso

Seu programa deve se comportar de acordo com os exemplos abaixo.

```
$ ./filter -g infile.bmp outfile.bmp
```

```
$ ./filter -s infile.bmp outfile.bmp
```

```
$ ./filter -r infile.bmp outfile.bmp
```

```
$ ./filter -b infile.bmp outfile.bmp
```

Dicas

- Os valores de um pixel de `rgbtRed`, `rgbtGreen` e `rgbtBlue` componentes são todos os inteiros, por isso certifique-se de arredondar os números de ponto flutuante para o número inteiro mais próximo ao atribuir-lhes um valor de pixel!

Testando

Certifique-se de testar todos os seus filtros nos arquivos de bitmap de amostra fornecidos!

Execute o seguinte para avaliar a exatidão do seu código usando `check50`. Mas certifique-se de compilar e testar você mesmo!

```
check50 cs50/problems/2021/x/filter/less
```

Execute o seguinte para avaliar o estilo do seu código usando `style50`.

```
style50 helpers.c
```

Como enviar

Execute o procedimento a seguir, fazendo login com seu nome de usuário e senha do GitHub quando solicitado. Por segurança, você verá asteriscos (`*`) em vez dos caracteres reais em sua senha.

```
submit50 cs50/problems/2021/x/filter/less
```