Processamento de Imagens

Processamento de Imagens de Textos

Professor: Bruno Dembogurski
Disciplina: Processamento de Imagens — TM438
Curso: Ciência da Computação
Deived William da Silva Azevedo, 2016780094
Handy Claude Marie Milliance, 2014780321

Introdução

O objetivo deste trabalho é processar imagens de texto visando recortar as letras das palavras. Para realizar essa tarefa, usamos algumas das técnicas e algoritmos de processamento de imagens que foram apresentados ao longo de todo o curso.

Para realização do recorte seguimos os seguintes passos

1. Imagem Load.
2. Conversão dos pixels RGB para escala de cinza.
3. Suavização da Imagem.
4. Binarização
5. Detecção de Bordas
6. Identificação das Bordas.
7. Para cada borda encontrada calcular um Bounding Rec
8. Desenhar os recortes.

Pré-requisitos

python3, mahotas, numpy, opencv

Imagem Load

Para carregamento da imagem em memória, utilizamos o pacote OpenCV na segunda versão, a função *imread*.

```
import cv2
import numpy as np
import mahotas
```

```
#open target image
target_img = cv2.imread('asserts/test_text.jpg')
#open alphabet image
alpha_img = cv2.imread('asserts/alpha.png')
```

Essa função, nada mais, nada menos, monta um array com o tamanho da imagem e a profundidade de cor. Por exemplo, para uma imagem *bitmap,* com dimensão de 1024x768, e profundidade de 24 bits de cor, o shape deste array seria (1024, 768, 3).

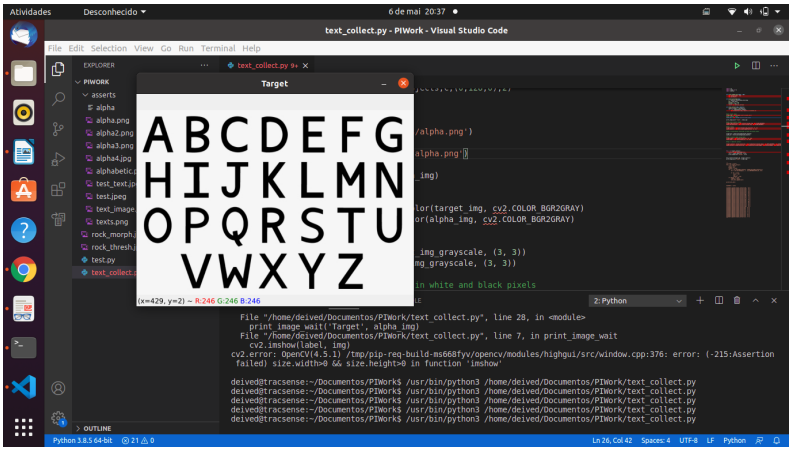

Figura 1: Carregando uma imagem usando OpenCV

GrayScale Converter

O segundo passo, foi a conversão da imagem colorida para tons de cinza. O propósito desta conversão é poder representar a intensidade de cada pixel em um único valor, ao invés de três, assim podemos ver quão claro está cada pixel. Essa conversão é muito importante para as próximas etapas.

Método

Y ← 0.299 * R + 0.587 * G + 0.114 + B

```
#gray scale convert images
target_img_grayscale = cv2.cvtColor(target_img, cv2.COLOR_RGB2GRAY)
alpha_img_grayscale = cv2.cvtColor(alpha_img, cv2.COLOR_BGR2GRAY)
```

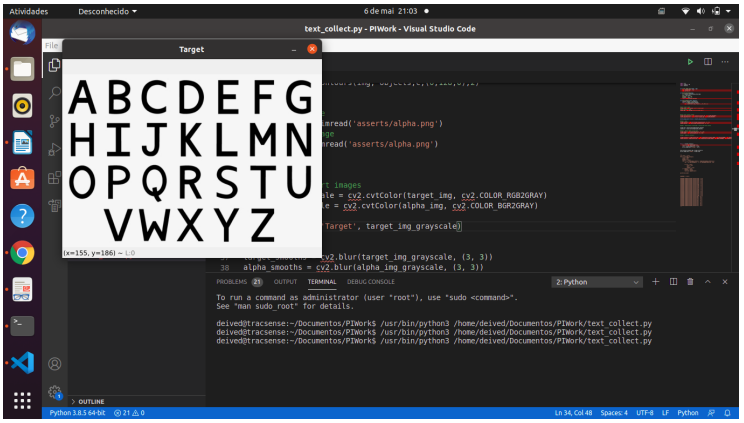

Figura 2: Imagem em tons de cinza.

Suavização da Imagem (Aplicação de um HPF)

A suavização foi aplicada a fim de remover alguns ruídos da imagem, para isso utilizamos um filtro passa alta, Filtro Bilateral. Este filtro é altamente eficaz na remoção de ruído preservando as bordas, pois a detecção de bordas é essencial para o nosso problema.

```
#Step 2: Bilateral filter
target_smooths = cv2.bilateralFilter(target_img_grayscale, 10, 75, 75)
alpha_smooths = cv2.bilateralFilter(alpha_img_grayscale, 10, 75, 75)
```

Binarização (Conversão Preto & Branco)

Nesta etapa nós simplesmente converte a imagem que está em escala de cinza para preto e branco. No ponto (x,y) onde a intensidade é maior que um limiar (*Otsu's Method*) escrevemos 255, ou seja, pixel totalmente aceso, caso contrário, esta posição recebe 0 que é apagado. Por fim, para deixar as bordas visíveis aplicamos um *bitwise not* pixels para inversão, assim teremos as arestas visíveis.

```
def BinarizationImg(img):
    T = mahotas.thresholding.otsu(img)
    bin = img.copy()
    bin[bin > T] = 255
    bin[bin < 255] = 0
    return cv2.bitwise_not(bin)
```

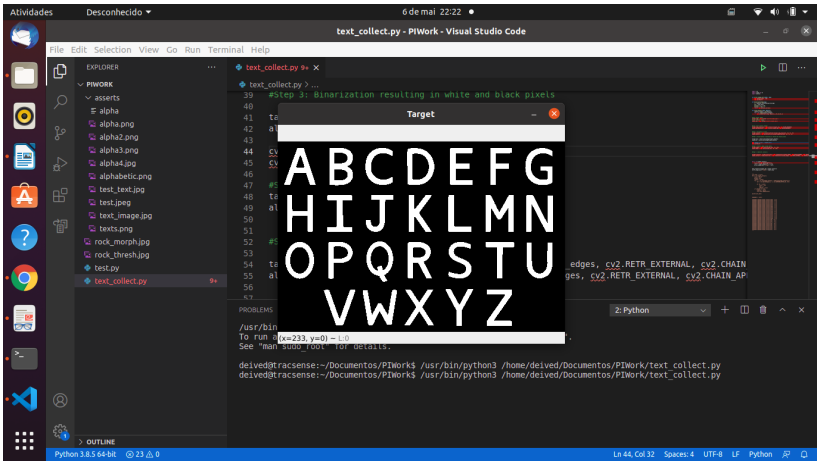

Figura 3: Binarização da Imagem (*Black and White)*

Detecção de Bordas

Para detecção de bordas, usamos o algoritmo de *Canny* que é um algoritmo muito famoso para este propósito. A borda de uma imagem pode apontar em diferentes direções, então o algoritmo de Canny usa quatro filtros para detectar horizontais, verticais e diagonais nas bordas da imagem desfocada. O operador de detecção de borda (Roberts , Prewitt , Sobel , por exemplo) retorna um valor para a primeira derivada na direção horizontal (Gy) e na direção vertical (Gx).

```
#Step 4: Detect edges using Canny
target_edges = cv2.Canny(target_bin, 70, 150)
alpha_edges = cv2.Canny(alpha_bin, 70, 150)
```

Os valores 70 e 150 são os valores para o limiar min e max respectivamente.

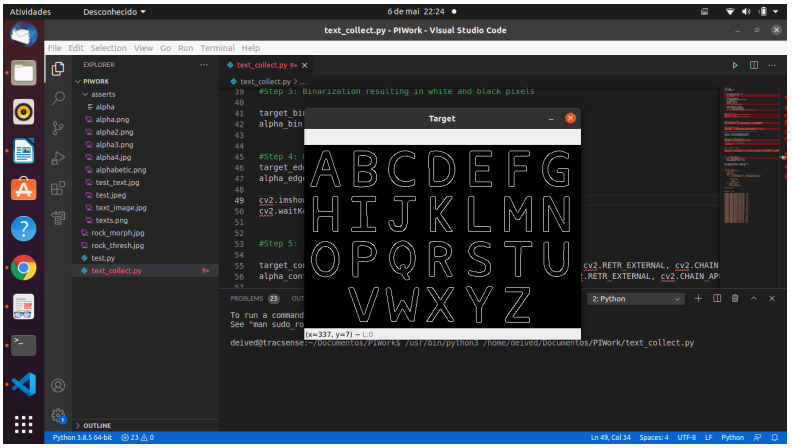

Figura 4: Resultado do algoritmo de Canny

Procurar Contornos

Nesta etapa basicamente procuramos todos os contornos na imagem, ou seja, todos os pontos vizinhos — ao longo da fronteira, que têm a mesma intensidade.

```
#Step 5: Identify contours

target_contours, target_hierarchy = cv2.findContours(target_edges,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
alpha_contours, alpha_hierarchy = cv2.findContours(alpha_edges,
cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

O parâmetro cv2.CHAIN_APPROX_SIMPLE remove todos os pontos redundantes e comprime o contorno, economizando memória.

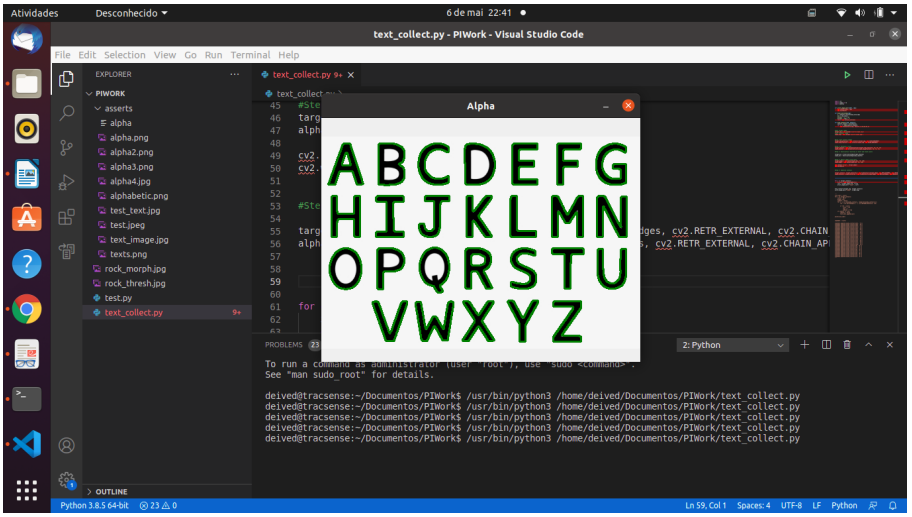

Figura 5: Contornos.

Bounding Rects & Recorte

Nesta etapa é realizado um *bounding rect* em todos os contornos a fim de calcular um ponto “x” ,” y” e um comprimento “w” (largura) e “h” (altura). Com estes valores, é possível fazer o recorte de cada contorno, pois basta lembrar que nossa imagem é um array, então podemos selecionar o ponto exato, isto é, “x” e “y”, e somar este ponto com a largura e altura, respectivamente.

```
    for c in target_contours:
        x,y,w,h = cv2.boundingRect(c)
        curt = target_img[y:y+h, x:x+w]
        print_image_wait('Target', curt)
```

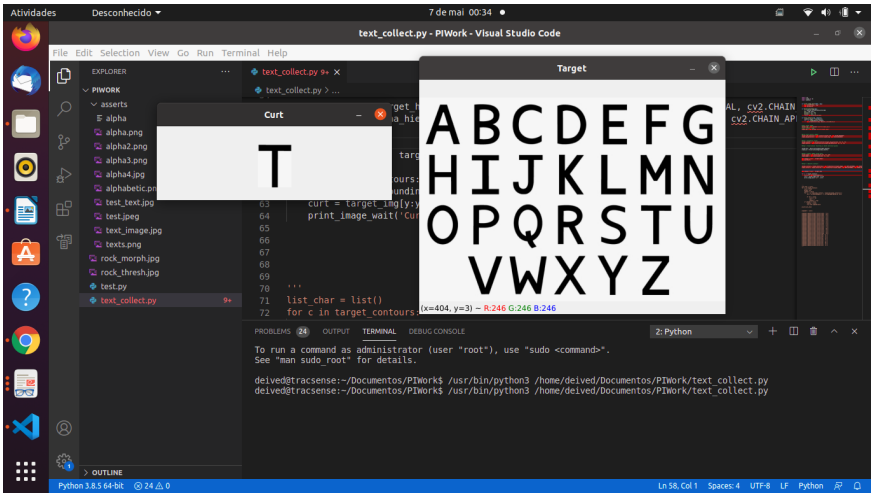

Figura 6: Recorte de uma letra.

A tecla ESC seleciona outro recorte.

Obs: Tentamos fazer a opção de reconhecer a letra carregando uma imagem “alfabeto”, e comparar os contornos com a imagem alvo a fim de calcular a similaridade entre os contornos. Quanto mais próximo de zero é o valor de “ret”, mais similaridade há entre os contornos “c” e “l”, como mostra o exemplo abaixo:

```
ret = cv2.matchShapes(c, l, cv2.CONTOURS_MATCH_I3, 0.0)
```

Exibição das imagens na tela

O algoritmo exibe uma imagem na tela, a cada etapa do processamento, e no final exibe cada letra recortada. Para cada letra recortada que é exibida na tela, deve-se fechar a janela onde a letra está sendo exibida, e em seguida é exibida uma nova letra. O algoritmo funciona para todas as imagens que estão na pasta, mas é mais eficiente no caso das imagens que são compostas por palavras ou letras maiores.