3x3 Median Filter

Tuesday, 26 April, 2022 9:53 AM

① PNG → .dat

```python
import numpy as np
from PIL import Image, ImageOps

# image can be png or jpg format
loc = "./data/"
fname = "input.png"
im = Image.open(loc + fname)
width, height = im.size
outdat = "input.dat"

with open('./data/dimension.dat', 'w') as f:
    for item in [width, height]:
        f.write("%s\n" % item)

def write_dat (f, pix_val):
    for i in range (0, len(pix_val)):
        f.write(str(pix_val[i])+"\n")
    f.close()

# get pixel values
print(">> Opening and processing image "+loc + fname+ " width "+str(width)+", height "+str(height))
im = Image.open(loc+fname, 'r').convert('RGB')
pix_val = list(ImageOps.grayscale(im).getdata())
# pix_val = list(im.getdata())
print(">> Total number of pixels: "+str(len(pix_val)))

# write dat file
print(">> Dat file written at "+loc+outdat)
f = open(loc + outdat, 'w')
write_dat (f, pix_val)

# grayscale input image
newimage = Image.new('L', (width,height))
newimage.putdata(np.array(pix_val))
newimage.save(f"{loc}gray_{fname}")
```

Import libraries

parent directory

image png to be converted to .dat

open the image

get dimensions/resolution

output file

save dimension/resolution info in dimension.dat to be used/read

callable function: write the pixel values in file variable f

ignore the RGB, this is written by Dr. Hadi

grayscale the input image.

call the function above

create a blank png and store/write the pixels into it

② 3x3 Median Filter in C++

```cpp
#include <iostream>
#include <chrono>
#include <cstdlib>
#include <cmath>

using namespace std;

void bubbleSort (int inputArray[], int arraySize)
{
    int i, j;

    for (i = 0; i < arraySize - 1; i++ ){
        for (j = 0; j < arraySize - i - 1; j++) {
            if (inputArray[j] > inputArray[j + 1]) {
                swap(inputArray[j], inputArray[j + 1]);
            }
        }
    }
}

void insertionSort (int inputArray[], int arraySize)
{
    int i, j, key;

    for (i = 1; i < arraySize; i++) {

    key = inputArray[i];
    j = i - 1;

    while (j >= 0 && inputArray[j] > key) {
        inputArray[j + 1] = inputArray[j];
        j = j - 1;
    }

    inputArray[j + 1] = key;

    }
}

int main(){

    cout << ">> Reading data..." << endl;
    FILE *fin, *fdim, *fout, *fout_m, *ftime, *fspeed;

    // store image dimensions as dim
    int dim[2];
    fdim=fopen("./data/dimension.dat","r");
    fscanf(fdim, "%d", &dim[0]);
    fscanf(fdim, "%d", &dim[1]);
    fclose(fdim);

    // declare width and height
    int width = dim[0], height = dim[1], row, col, pixel_pointer=0;

    cout << ">> Allocating memory..." << endl;
    // store input.dat as vec
    int *vec = (int*) malloc(width*height * sizeof(int)); // Dynamically allocated (width x height) empty matrix space

    fin=fopen("./data/input.dat","r");
    for (int i = 0; i < width*height; i++) {
```

Include libraries

Bubble sort algorithm

Insertion sort algorithm

declare files as variables

get the image's resolution/dimensions and store in dimension.dat

create dynamic matrix named vec

```
    cout << ">> Allocating memory..." << endl;
    // store input.dat as vec
    int *vec = (int*) malloc(width*height * sizeof(int)); // Dynamically allocated (width x height) empty matrix space

    fin=fopen("./data/input.dat","r");
    for (int i = 0; i < width*height; i++) {
        fscanf(fin, "%d", &vec[i]);
    }
    fclose(fin);

    // declare storage to store image data (not working directly on image data in vec,
    // allowing the choise of whether to use preprocessing such as padding)
    int N = 2000;
    int *img_array = (int *)malloc(N * N * sizeof(int)); // Dynamically allocated (2000x2000) empty matrix space

    cout << ">> Preprocessing data..." << endl;

    // set all elements = 0
    for (row = 0; row < N; row++){
        for (col = 0; col < N; col++){
            img_array[row*N + col] = 0; // equivalent to img_array[row, col]
        }
    }

    // store pixel data into img_array matrix
    // // // With zero padding
    // for (row = 1; row < height-1; row++){
    //   for (col = 1; col < width-1; col++){
    //     img_array[row*N + col] = vec[pixel_pointer];
    //     pixel_pointer++;
    //   }
    // }

    // // No padding
    for (row = 0; row < height; row++){
        for (col = 0; col < width; col++){
            img_array[row*N + col] = vec[pixel_pointer];
            pixel_pointer++;
        }
    }

    // 3x3 median filter main algorithm

    // int window[9];
    int *window = (int*)malloc(9 * sizeof(int));
    double t, speed, sum_t=0, sum_speed=0;

    cout << ">> Performing 3x3 median filter, please wait..." << endl;

    fout=fopen("./data/out.gold.dat","w");
    fout_m=fopen("./data/out.m.dat","w");
    ftime=fopen("./data/time_ns.dat","w");
    fspeed=fopen("./data/speed_ns.dat","w");

    for(row = 1; row <= height; row++)
    {
        for(col = 1; col <= width; col++)
        {
            // Window = 3x3 matrix, centered at (row,col)
            window[0] = img_array[(row-1)*N + (col-1)];
            window[1] = img_array[(row-1)*N + (col)];
            window[2] = img_array[(row-1)*N + (col+1)];
            window[3] = img_array[(row*N) + (col-1)];
            window[4] = img_array[(row*N) + (col)];
            window[5] = img_array[(row*N) + (col+1)];
            window[6] = img_array[(row+1)*N + (col-1)];
            window[7] = img_array[(row+1)*N + (col)];
            window[8] = img_array[(row+1)*N + (col+1)];

            //sort window array (pick any sorting algorithm above)
            auto start = chrono::steady_clock::now(); // Start timer
            insertionSort(window, 9);
            auto end = chrono::steady_clock::now(); // Stop timer
            t = chrono::duration_cast<chrono::nanoseconds>(end - start).count(); // get nanoseconds taken
            sum_t += t; // add to total time taken (still in nanoseconds)
            speed = std::isfinite(9.0/t) ? 9.0/t : speed; // Speed = (number of pixels in the window / time taken) = pixels per nanosecond
            sum_speed += speed;
            fprintf(ftime,"%.0f\n",t); // Write nanoseconds (per window) to time.dat
            fprintf(fspeed,"%.8f\n",speed); // write speed (pixels per nanosecond) to speed.dat
            fprintf(fout,"%d\n",window[4]); // Write median into out.gold.dat
            fprintf(fout_m,"%d\t",window[4]); // Write median into out.m.dat
        }
        fprintf(fout_m,"\n");
    }

    double pps = sum_speed*1e9/(width*height); // total speed / total pixels = average pixels per second

    fclose(fout);
    fclose(fout_m);

    free(img_array);
    free(vec);
    free(window);

    cout << ">> Time taken per pixel (nanoseconds) written as /data/time_ns.dat." << endl;
    cout << ">> Speed per pixel (pixels per nanoseconds) written as /data/speed_ns.dat." << endl;
    cout << ">> Saved processed output pixels matrix as /data/out.m.dat and flattened as /data/out.gold.dat." << endl;
    cout << endl << "- - - Time profiling - - -" << endl << ">> Total time taken = " << sum_t*1e-9 << " seconds." << endl;
    cout << ">> Processing speed = " << pps << " pixels per second." << endl;

    return 0;
}
```

create dynamic matrix named vec

store input.dat into vec

create a 2000x2000 dynamic empty matrix space for the input to be placed and worked on

store the input image pixels one by one, row by row into the img_array.

with zero padding

no zero padding

pixel pointer movement

create the 3x3 window

variables for speed and time profiling

create those files

0	1	2
3	4	5
6	7	8

median

elapsed time in nanoseconds (end minus start)

total time taken

speed to get that median

total speeds

n pixels

speed 1 | speed 2 | speed 3 | ... | speed n

$$\text{avg speed} = \frac{\sum\left[\text{speed (pixels/ns)}\right] \times (1\times10^9)}{\text{width} \times \text{height}}$$

$$\frac{\frac{1}{\text{time 1}} + \frac{1}{\text{time 2}} + \frac{1}{\text{time 3}} + \cdots + \frac{1}{\text{time n}}}{n}$$

speed = 1/time

num. pixels

= average speed

(3) .dat → PNG

dat_to_img.py 3

③ .dat → PNG

dat_to_img.py 3

```python
import numpy as np
from PIL import Image

input_dat = './data/v_outdata.dat'
im = Image.open('./data/input.png')
width, height = im.size
output_png = "output.png"

with open(input_dat, 'r') as file:
    data = file.readlines()

for i in range(len(data)):
    # data[i] = int(data[i].replace(';\n',''))
    data[i] = int(data[i])

data = np.array(data)

newimage = Image.new('L', (width,height))
print(f">> Opening and processing dat file {input_dat} width {width}, height {height}")
print(f">> Total number of pixels: {width*height}")
print(f">> PNG file generated as {output_png}")
newimage.putdata(data)
newimage.save(f"./{output_png}")
newimage.save(f"./data/{output_png}")
```

Import necessary libraries

read the vertical .dat output file

declare output png

read the dat line-by-line (default is read as string format)

convert data into integer format

convert into Numpy array

create new blank PNG ('L' for grayscale, dunno why)

put the pixels in

save at 2 separate locations

④ Plot speed with Python

plot_speed.py

```python
import numpy as np
import matplotlib.pyplot as plt

def smooth(y, box_pts):
    box = np.ones(box_pts)/box_pts
    y_smooth = np.convolve(y, box, mode='same')
    return y_smooth

print(">> Reading speed data...")

speedData = [element * 1000 for element in list(map(float, open('./data/speed_ns.dat', 'r').read().splitlines()))]

print(">> Plotting the data...")

plt.figure(figsize=(20,15))
plt.plot(speedData, color='blue', linewidth=1)
plt.plot(smooth(speedData, 5000), color='red', linewidth=1)
plt.legend(['Speed','Smoothed'], loc='upper right')
# plt.ylim(0, 0.1)
plt.grid(True)
plt.title("3x3 Median Filter - Processing Speed\n", fontsize=30)
plt.xlabel("\nPixel number", fontsize=20)
plt.ylabel("Speed (megapixels per second)\n", fontsize=20)
plt.savefig("Speed.png")
# plt.show()

print(">> Speed per pixel plotted and saved as Speed.png.")
```

Smoothing algorithm using Numpy convolution.

read speed.dat and x1000 for all, so pixels per ns becomes megapixels per second

original data, blue line

smoothing window size = 5000, red line

graph size

grid on

graph title

axis labels

save plot as png

Example:

$$\frac{0.03\ px}{1\ ns} = \frac{0.03\ px}{1\times10^{-9}\ s} = \frac{0.03\times10^{9}\ px}{1\ s}$$

$$\downarrow$$

$$\frac{0.03\times10^{3}\times10^{6}\ px}{1\ s} = \frac{0.03\times10^{3}\ megapixels}{1\ s}$$

Hence why x1000.

DONE