CMP2090M

Object-Oriented Programming Assignment Report

Daniel Dixon

Student ID: 16602092

1. Introduction

For this project we were given the brief to develop software that is able to manipulate and resize images based on several established techniques. The code was written using Visual Studio 2015 and when Running the code please use the executable found in the 64-bit release folder.

2. Programme Structure

Main Menu and User choice:
The program initially starts in Assessment1Source.cpp. The user is shown the initial display in the main function before going to the full main menu in the user choice function, these two functions were separated to give the option of coding in other functions later. The user then chooses whether to do Image manipulation or Image scaling, either option has separate Images to interact with so separating them allows for quicker testing. Both the main and User Choice Functions are part of the Image class as they have not diversified into either Image manipulation or scaling yet.

Image Manipulation:
If the user chooses to manipulate the images three functions are called to get mean, median and sigma respectively. These functions and all functions taking place in this section are located in the manipulation class. All Images in the dataset for manipulations are read into an image array. While doing this, data is displayed about the images being read in such as width, height, binary colour depth, filename and time taken to read the images, all of which is stored in an outside text file. The images are then manipulated using their respective algorithms and then outputted to storage. After this the user can decide whether to view the specific details of an image that had been read in, either from the original dataset or the three manipulated images.

Image Scaling:
The user may decide to do the image scaling before or after the manipulations but for either choice the Zoom class is used. The Zoom class inherits from the Image class and starts the loading of images as soon as the zoom2x4x function is entered. The user gets no further input and the code first zooms the original image by a scale factor of 2 and 4 before doing the same to the region of interest.

3. Algorithms

Mean Blending:

```
71      for (int b = 0; b < img_Array[0].w * img_Array[0].h; ++b) {
72          for (int a = 0; a <= 12; ++a) {
73
74              img_Array[13].pixels[b] += img_Array[a].pixels[b];
75          }
76      }
77      for (int c = 0; c < img_Array[0].w * img_Array[0].h; ++c) {
78          img_Array[13].pixels[c] /= 13.0f;
79      }
```

To calculate the mean of all the images the += operator is overloaded to calculate the Sum of all R, G and B values with one line of code, this is nestled inside two for loops that go to each pixel and in all thirteen Images. Once the sum is calculated and stored into the part of the array used for mean a second loop and overloading of the /= operator are used to divide every pixel by thirteen, the number of images in the dataset.

Median Blending:

```
92      float red[13] = {}, green[13] = {}, blue[13] = {};
93      //for (int d = 0; d < 13; ++d) {
94      //  MedVect.push_back(img_Array[d]);
95      //}
96      //vector<Image>::iterator it1;
97      for (int d = 0; d < 3264 * 2448; ++d) {
98          for (int e = 0; e < 13; ++e)//it1 = MedVect.begin(); it1 !=
99          {
100             red[e] = img_Array[e].pixels[d].r;
101             green[e] = img_Array[e].pixels[d].g;
102             blue[e] = img_Array[e].pixels[d].b;
103         }
104         sort(red, red + 13);
105         sort(green, green + 13);
106         sort(blue, blue + 13);
107
108         img_Array[14].pixels[d].r = red[6];
109         img_Array[14].pixels[d].g = green[6];
110         img_Array[14].pixels[d].b = blue[6];
111     }
```

Three float arrays are created to store the float values of R G and B for each pixel, these would then be sorted before the middle value being inputted into the part of the array used to store the median. This repeats for every pixel.

Sigma Clipping:

```
127     Image *Temp1 = new Image(3264, 2448);
128     //(*Temp1).pixels = {0};
129     Image *Temp2 = new Image(3264, 2448);
130     Image *StandDev = new Image(3264, 2448);
131     vector <float> RStore;
132     vector <float> GStore;
133     vector <float> BStore;
134     float medr, medg, medb;
135     for (int j = 0; j < 3264 * 2448; ++j) {
136         RStore.clear();
137         GStore.clear();
138         BStore.clear();
139         for (int k = 0; k <= 12; ++k) {
140             (*Temp1).pixels[j].r += ((img_Array[k].pixels[j].r - img_Array[14].pixels[j].r) * (img_Array[k].pixels[j].r - img_Array[14].pixels[j].r));
141             (*Temp1).pixels[j].g += ((img_Array[k].pixels[j].g - img_Array[14].pixels[j].g) * (img_Array[k].pixels[j].g - img_Array[14].pixels[j].g));
142             (*Temp1).pixels[j].b += ((img_Array[k].pixels[j].b - img_Array[14].pixels[j].b) * (img_Array[k].pixels[j].b - img_Array[14].pixels[j].b));
143         }
144         (*Temp1).pixels[j] /= 13.0f;
145         (*StandDev).pixels[j].r = sqrt((*Temp1).pixels[j].r);
146         (*StandDev).pixels[j].g = sqrt((*Temp1).pixels[j].g);
147         (*StandDev).pixels[j].b = sqrt((*Temp1).pixels[j].b);
148         (*Temp1).pixels[j].r = img_Array[14].pixels[j].r + (sigma * (*StandDev).pixels[j].r);
149         (*Temp1).pixels[j].g = img_Array[14].pixels[j].g + (sigma * (*StandDev).pixels[j].g);
150         (*Temp1).pixels[j].b = img_Array[14].pixels[j].b + (sigma * (*StandDev).pixels[j].b);
151         (*Temp2).pixels[j].r = img_Array[14].pixels[j].r - (sigma * (*StandDev).pixels[j].r);
152         (*Temp2).pixels[j].g = img_Array[14].pixels[j].g - (sigma * (*StandDev).pixels[j].g);
153         (*Temp2).pixels[j].b = img_Array[14].pixels[j].b - (sigma * (*StandDev).pixels[j].b);
154         for (int l = 0; l <= 12; ++l) {
155             if (img_Array[l].pixels[j].r > (*Temp1).pixels[j].r || img_Array[l].pixels[j].r < (*Temp2).pixels[j].r) {
156                 img_Array[l].pixels[j].r = 0.0f;
157             }
158             if (img_Array[l].pixels[j].g > (*Temp1).pixels[j].g || img_Array[l].pixels[j].g < (*Temp2).pixels[j].g) {
159                 img_Array[l].pixels[j].g = 0.0f;
160             }
161             if (img_Array[l].pixels[j].b > (*Temp1).pixels[j].b || img_Array[l].pixels[j].b < (*Temp2).pixels[j].b) {
162                 img_Array[l].pixels[j].b = 0.0f;
163             }
```

```
170         size_t sizer = RStore.size();
171         size_t sizeg = GStore.size();
172         size_t sizeb = BStore.size();
173         sort(RStore.begin(), RStore.end());
174         sort(GStore.begin(), GStore.end());
175         sort(BStore.begin(), BStore.end());
176
177         if (sizer % 2 == 0) medr = (RStore[sizer / 2 - 1] + RStore[sizer / 2]) / 2;
178         else medr = RStore[sizer / 2];
179         img_Array[15].pixels[j].r = RStore[medr];
180
181         if (sizeg % 2 == 0) medg = (GStore[sizeg / 2 - 1] + GStore[sizeg / 2]) / 2;
182         else medg = GStore[sizeg / 2];
183         img_Array[15].pixels[j].g = GStore[medg];
184
185         if (sizeb % 2 == 0) medb = (BStore[sizeb / 2 - 1] + BStore[sizeb / 2]) / 2;
186         else medb = BStore[sizeb / 2];
187         img_Array[15].pixels[j].b = BStore[medb];
188     }
```

```
ixels[j].r);
ixels[j].g);
ixels[j].b);
```

The sigma clipping was more complex in its design as it first has to calculate the standard deviation in several parts before storing all the parts for the equation m+-(alpha * sigma) into the two Temp Images. Once done all the images have the equation applied to them and any that are outside the temp image ranges are set to zero. Any pixel RGB values that are inside this range are then sent to a vector to be sorted and the median of the remaining values calculated

Zoom Scaling:

```
oid Zoom::near_Neighb(vector<Image> Vect, int times) {
   auto start = chrono::high_resolution_clock::now();
   int w1, h1, w2, h2;
   int store = 1;
   double px, py;
   w1 = Vect[0].w;
   h1 = Vect[0].h;
   w2 = w1 * times;
   h2 = h1 * times;
   double x_ratio = w1 / (double)w2;
   double y_ratio = h1 / (double)h2;
   if (times == 2) { store = 1; }
   else if (times == 4) { store = 2; }
   for (int i = 0; i < h2; i++) {
       for (int j = 0; j < w2; j++) {  //Completing nearest neighbour processing
           px = floor(j * x_ratio);
           py = floor(i * y_ratio);
           Vect[store].pixels[(i*w2)+j]= Vect[0].pixels[(int)((py*w1) + px)]; // V1[
       }
   }
```

As shown above the width and height of the two images are taken and then their ration is computed, this ratio is then used in the for loop to copy over the original image to the zoomed image.

4. Results

4.1 Part A Manipulation Comparisons

Sigma Clipped: Value

As can be seen above all Images look fairly blurry, however the sigma shows the clearest image as the edges of different objects have been reduced to black

4.2 Part B Scaling Comparisons

The Zoom Algorithm does not show much difference when displayed on either photoshop or Gimp as the image is rescaled by those programs but the images do tend to become more pixelated the more zoomed in the picture becomes. This would have been rectified if another zoom algorithm had been implemented

4.3 Stretch Exercises

Optimisation: Several attempts were made to improve the efficiency of the code, including pushing the Image objects straight into an array rather than creating them, then reading and then storing them. This cut the code down while simultaneously increasing efficiency. Header Guards were also used to speed up compilation at run time. Several other attempts were made in areas of the code that bottlenecked such as using different sorting algorithms to find the median and using different code to write the data to the images, these were either unsuccessful in improving speed or could not be created successfully.

Zoom Algorithm:
An attempt was made to use bilinear interpolation to gain a better image but coding in the time given was unsuccessful.

ROI:
A Region of interest was created in the top right of the original image, this was then zoomed into successfully as asked in the brief

5. Discussion & Conclusion

Manipulation: The code works effectively and although it has caused a lot of errors in it's creation is now relatively unbreakable, due to it running all manipulations in one go, error checking was very time consuming
Zoom: The zooming of images worked quite well and apart from issues with polymorphism runs wells.

Optimisation: The optimization went well in some areas but in others a lot of time was used up trying to get them working when they never would. A lot of focus on the biggest bottle neck, the Writeppm could have been used better elsewhere.
ROI: The same as zoom, runs efficiently and with few errors.