CS900 Lab - Day 06

July 7th, 2003

Instructor: Thomas Hofmann

New Topics Covered

Image Processing – Lecture: Day 6
Two-dimensional Arrays, Lecture: Day 4+6, King: Chapter 8, in particular 8.2

Installing

Copy all files from the directory /course/cs900/day06/src by typing

```
cd
mkdir day06
cd day06
cp /course/cs900/day06/src/* .
```

This should include the files g2.h, g2_X11.h, libg2.a, lena.pgm, einstein.pgm, images.c. Make sure that you have copied over the files by typing

```
ls -la
```

Introduction

In this project you will write programs for image processing. Image processing refers to any operation that acts to improve, correct, analyze, or in some way change an image. For our purposes we will be manipulating grayscale images through a series of transformations and filters. This will improve your skills in dealing with two-dimensional arrays. The visualization will enable you to clearly distinguish between the original image and the manipulated image to determine the functionality of your program.

For all programs that you write, you will need to use the g2 library in order to display the two-dimensional array as a grayscale image. Hence, the compilation command will look like this (with foo being replace by your program's name):

```
gcc foo.c -o foo -Wall -L. -L/usr/X11R6/lib -lm -lg2 -lX11
```

Problem 1 – Inverted Images

Using images.c as a template, write a program that inverts the original image and displays it. You should load images.c in xemacs, then save it under a new name, such as invert.c and continue to edit this source file.

You will see that images.c includes a number of headers, referring to all kinds of libraries, in particular graphics libraries. Don't change these lines. Images will be stored in two-dimensional arrays,

```
int image1 [NUM_IMAGE_ROWS] [NUM_IMAGE_COLS] ;
int image2 [NUM_IMAGE_ROWS] [NUM_IMAGE_COLS] ;
```

The image sizes are specified as constants using the #define syntax. You can read in two different images,

```
read_image("lena.pgm",image1);
```

or

```
read_image("einstein.pgm",image1);
```

it's up to you to use either one of them (or both).

You have to create a window for display with the command

```
win = create_window();
```

Here win will be a handle to the created window, which you can subsequently use to display images using the function call

```
display_image(image1,win);
```

The program also includes a line

```
getchar();
```

Which is useful to interrupt the program execution until the user hits <ENTER>.

Problem 2 – Geometrical Image Transformations

Using images.c as a template, write a program that geometrically transforms the original image. You should load images.c in xemacs, then save it under a new name, such as geometric.c and continue to edit this source file.

a) Create a second image that is the original one flipped upside down and display the image
b) Create a version of the image that is rotated by 180 degrees and display the result.

Problem 3 – Quantization

Follow the same steps as above creating a C program quantize.c.

Now implement a pixel by pixel quantization. This means that each pixel value is replaced by a value from a smaller set of values, e.g.{0, 64, 128, 191, 255}. Map each pixel value x into the closest quantized value available. For example, a grayscale value of 180 would be mapped to the quantized value 191, a value of 240 to 255, etc. Display the resulting image(s). Implement your program in a file quantize.c.

Problem 4 – Mosaic Images

A mosaic is a block with the same luminance or color. Imagine the original image of size 256 by 256 is decomposed (in your mind) into non-overlapping squares of size 8 by 8 (hence there are 32 x 32= 1024 such squares). Within each square, replace the pixel value of each pixel by the average pixel value of the square it belongs to. This should look like a mosaic image, the block structure should be clearly visible. Implement your program in a file mosaic.c.

Problem 5 – Image Filtering

Implement the 2-dimensional convolution discussed in the lectures. Write your program in a file filter.c. Filter images with the blurring filter and the edge detection (Lapalacian) filter. Display the results.

Remember that a blurMask can be defined as

```
int blurMask[][] = {
      {  1,  4,  6,  4,  1 },
      {  4, 16, 25, 16,  4 },
      {  6, 24, 36, 24,  6 },
      {  4, 16, 25, 16,  4 },
      {  1,  4,  6,  4,  1 }
}
int normal = 256;
```

whereas the Laplacian filter can be implemented with the mask

```
int edgeMask[][] = {
      {  0,  0, -1,  0,  0 },
      {  0, -1, -2, -1,  0 },
      { -1, -2, 16, -2, -1 },
      {  0, -1, -2, -1,  0 },
      {  0,  0, -1,  0,  0 }
}
int normal = 16;
```

Problem 6 – Artistic Filters

Implement some artistic image filters. Start with the ice-melt effect which uses the following conditional swap function: Pick a point at random. Exchange the pixel value with pixel value above, if value is smaller than the one above, otherwise do nothing.

Here a code sniplet for the cruial part of the program:

```
rndrow = (rand() % (NUM_IMAGE_ROWS-1))+1 ;
rndcol =  rand() %  NUM_IMAGE_COLS       ;
if (img[rndrow][rndcol] < img[rndrow-1][rndcol])
{
    tmp                   = img[rndrow][rndcol];
    img[rndrow][rndcol]   = img [rndrow-1][rndcol];
    img[rndrow-1][rndcol] = tmp;
}
```

Think about other effects, for example by randomly shifting neighboring pixels, trying to get a fish-eye effect, relief map, etc.