Writing digital image processing (dip) software

Image format

origin

\[ f(x,y) \]
Some basic terminology

- **bpp** (bit depth)
  - bits per pixel (sometimes bytes per pixel)
- **rgb**
  - red, green, blue... ie, color image
- **grayscale or monochrome**
  - intensity only image
  - not to be confused with black and white image
- **bi-tonal**
  - black and white image
- **raw image, raw format**
  - uncompressed image
- **depth**
  - refers to the number of "bands" in an image
  - grayscale has 1 band (depth=1)
  - rgb has 3 bands (depth=3)
- **scanline**
  - an individual row of an image

Image format

```
origin
```

```
x
```

```
y
```

```
scanline
```

```
image of fish
```
Data structure

- **Necessary Information**
  - Width, Height (# Columns, # of Rows)
  - bits/bytes per pixel
    - 1 byte
    - 3 bytes
    - 4 bytes
    - May just 1 bit (black and white)
  - Image data (array)

Data/Class structure

Example

```cpp
class ImageClass
{
  public:
    int width, height;
    int bytes_per_pixel;   // you may want to have bits_per_pixel
                           // instead of bytes
    unsigned char *imaData;  //

    ImageClass();
    ...
    ...
}
```
Typical for this class

- Gray level is [0-255]
  - 8 bits per pixel (unsigned char)

```cpp
Class ImageClass
{
    public:
        int width, height;
        unsigned char *imaData;  // Gray level assumed to be [0-255]

        ImageClass();
        ...
        ...
}
```

Indexing your image

- How you are going to index your image?
  - F(x,y) or F(y,x)?

- Personally, I like F(y,x)
  - Why?
    - Matrix notation, row, col
      - imaData[j][i] // row, col

  - We tend to think in terms of scanlines (rows) and not columns, hence matrix notation is more appropriate
Representing your image?

- 1D or 2D array
- Say you have a 512x512 image:
  - 1D array
    - unsigned char imaData[ 512 * 512 ];
  - 2D array
    - unsigned char imaData[ 512 ][ 512 ];

Image size

- unsigned char imaData[512][512]
  - compile-time statement

- Unfortunately you generally do not know the size of your image in advance.

- YOU WILL NEED DYNAMIC MEMORY ALLOCATION
Dynamic Memory Allocation

- 1D case (assuming grayscale)

```cpp
Public ImageClass
{
  int width, height;
  unsigned char *imaData;

  ImageClass(int _w, int _h)
  {
    this->width = _w;
    this->height = _h;
    imaData = new unsigned char[_w * _h]; // allocate an array of pixels
  }

  ~ImageClass()
  {
    delete [] imaData;
  }
}
```

- 2D case (assuming grayscale)

```cpp
Public ImageClass
{
  int width, height;
  unsigned char **imaData;

  ImageClass(int _w, int _h)
  {
    this->width = _w;
    this->height = _h;
    imaData = new unsigned char *[_h]; // Allocate an array of points
    for (int j=0; j < w; j++)
      imaData[j] = new unsigned char[_w]; // Allocate individual arrays of "pixels"
  }

  ... (con't)
```
Dynamic Memory Allocation

- 2D case (assuming grayscale)

...  
...  
~ImageClass() // cleanup! 
{  
  for(int j=0; j < height; j++)  
    delete [] imaData[j];  
  delete [] imaData;  
}  
}

I personally like the 1D representation. Why?

---

1D memory as a 2D image

Logical representation

1
2
3
...
N

M (scanline size)

Physical representation in memory

1 2 3 ...

M

N
1D memory as a 2D image

- Indexing 1D memory

  Coordinates \((x, y)\) [or \((j, i)\)]

  \[-imaData[y \times width + x]\]
Images with depth > 1

- Individual bands are “packed” into one array

\[
\begin{array}{cccccccccccccccc}
\text{r} & \text{g} & \text{b} & \text{r} & \text{g} & \text{b} & \text{r} & \text{g} & \text{b} & \text{r} & \text{g} & \text{b} & \text{r} & \text{g} & \text{b} \\
\end{array}
\]

Images with depth > 1

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & \ldots & N \\
\end{array}
\]

\(M \times \text{depth}\)

(Scanline size)

Physical representation in memory

\[
\begin{array}{cccccccccccccccc}
1 & 2 & 3 & \ldots & N \\
\end{array}
\]

\(M \times \text{depth}\)
Images with depth > 1

• \texttt{imaData[ (y*width+x) * depth ]}

• this places you at the start of the “pixel”

\begin{verbatim}
\begin{verbatim}
some_function()
\{ . . .
    unsigned char r, g, b;
    int offset = (y*width*x)*depth;
    r = imaData[offset + 0];
    b = imaData[offset + 1];
    g = imaData[offset + 2];
    . . .
\}
\end{verbatim}
\end{verbatim}

Alternatives

• RGB image

\begin{verbatim}
\begin{verbatim}
struct rgbPixel // declare a structure big enough to
{       // hold the “RGB pixel”
    unsigned char r, g, b;
};
rgbPixel *imaData = new rgbPixel[w*h];
\end{verbatim}
\end{verbatim}

im\texttt{aData[ j * w + i \}].r = . . . // red value
im\texttt{aData[ j * w + i \}].g = . . . // green value
im\texttt{aData[ j * w + i \}].b = . . . // blue value
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
Other types of images

- Floating point representations
  - "float" image

- Pixel = float value;
  
  ```c
  float *imaData = new float[w * h];
  ```

- Very useful for image processing
  - You generally have to convert this to a monochrome (or rgb) image to view it!

Some issues to be aware of

- Let say you are going to "add" pixels "a" and "b" (c = a + b)

  ```c
  unsigned char a, b, c;

  a = imaData1[j * w + i];
  b = imaData2[j * w + i];

  c = a + b;    // What is the problem with this?
  // say a = b = 130;
  ```
Some issues to be aware of

• Be careful of overflow!

- unsigned char a, b, c;
- a = 130;
- b = 130;

\[ c = a + b; \quad c = 4!!!! \ (260 \mod 256) \]

Effects of overflow

+50 to each pixel
Avoiding overflow

```
{ ... 
    int a, b, c;
    a = (int) imaData1[ j * w + i ];
    b = (int) imaData2[ j * w + i ];
    c = a + b;       // no overflow problem
Now, you have to make a choice! What do you do with values outside the grayscale [0-255]?
```

Handling overflow

+50 to each pixel

Restrict the range to [0-255]
Handling overflow/underflow

• restrict to [0-255]
  if (c < 0)
    c = 0;
  if (c > 255)
    c = 255;

• Create a new image with more than 8 bits per pixel (per band)
  - float image

• Depends on the application

Viewing a “float” image

• Say you have done some processing and now have a float image

• “pixel” values  [f_min, f_max]
  - no restrictions on f_min and f_max
  - (other than they are finite)

• Need to rescale the image for visualization
  - float value mapped to [0 to 255]
“Rescaling” for visualization

... (Example Code) ...

unsigned char *grayIma = unsigned char [w*h]; // Grayscale Image

float minI = floatIma[0], maxI = floatIma[0]; // Initialize with first value of image

for(int i=0; idx < w*h; i++) // find min and max
{
    if (floatIma[idx] < minI)
        minI = floatIma[idx];
    if (floatIma[i] > maxI)
        maxI = floatIma[i];
}

float scaleFactor = 255.0 / (maxI – minI); // Determine the scale factor

// scale image
for (int idx=0; idx < w*h; idx++)
{
    float value = scaleFactor * (floatIma[i] - minI);
    if (value > 255) value = 255; // cap the values
    if (value < 0) value = 0; // should never happen but just in case
    grayIma[idx] = (unsigned char) value;
}

...

Typical IP diagram

- Gray Image ➔ Process ➔ Float Image
- Gray Image
More issues

\[ \text{Image} \xrightarrow{\text{process}} \text{Image} \]

Different size

How do you sample \( M \times N \) to generate \( S \times T \)?

Image boundaries

- Some local processing uses neighboring pixels

Value “\( t \)” depends on “\( s \)” and its neighbors
Image boundaries

- What do you do in the boundaries?

"s" neighbors do not exist!

One solution: Ignore boundary

Resulting image is slightly smaller.
Do you create a smaller image, or just put "0" in the border?
Another solution: reflect the boundaries

Reflecting the boundaries

reflecting a boundary

\[ p = \text{imaData}[ \text{abs}(y) \times w + \text{abs}(x) ]; \]

\[ \ldots \]
Basic functionality

• `createImage(int w, int h)`
  - Create a new image

• `readImageFromFile()`
  - Read image from a file

• `writeImageToFile()`
  - Write image to a file

PNM files

• *Portable Any Map* files
  - Very basic and easy to understand file format for raw images

• In particular, you should know
  - Portable Gray Map (PGM)
  - Portable Pixel Map (PPM)
  - Two types:
    • ASCII Format and Binary Format
PGM and PPM file format

- Magic Number
- Comments (if any) (denoted by a '#' )
- Width Height
- Max Value
- Data
  - Either numbers in ASCII format separated by spaces and/or CR
  - Or binary packed data.
  - PPM is "R G B  R G B  R G B" format
  - PGM is "I I I I"

Example

P3
# created by Yang
# 08-30-2004 for CS635
8 8
255
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0
Example

• Magic Numbers
  - P3  (PPM  ASCII format )
  - P6  (PPM  Binary format)
  
  - P2  (PGM  ASCII format)
  - P5  (PGM  Binary format)

** For Binary format (P{5,6}), max value is always 255 **

Accessing the image data

• You can make your “image data” public
• Goes against some OO philosophies
  - but it makes life easy

public ImageClass
{
  public:
    unsigned char *imaData;
}

Accessing the image data

• More appropriate OO style

- unsigned char getPixel(int x, int y)
  { return imaData[y*this->width+x]; }

- void setPixel(int x, int y, unsigned char value)
  { imaData[y*this->width+x] = value; }

  // May be a good idea to check the values of x
  // and y before you access the data
  // (0 <= x < width) AND (0 <= y < height)!!

Working buffers

Gray Image → Process → Float Image

Working Buffer

Gray Image
Working buffers

Problems w/ working buffers

- Need to allocate memory
- If the process is called often, this can waste lots of time
- Some tricks
Static working buffer

... (Example Code)...

```c
void process(int w, int h)
{
    static bool firstEntry = true;
    static float *workBuf = NULL;

    if (firstEntry)                                                 // Trade a little "overhead" logic
        {                                                               // for memory management
            // allocate the buffer
            workBuf = new float[w * h];                  // bad programming style, because
            firstEntry = false;                                  // this memory is never cleanup until
        }                                                               // the end of the program
    //   It is a trade-off . . .

    // Don't forget initialize this to Zero
    for(int i=0; i < w*h; i++)
        workBuf = 0;

    . . . IP code . . .
}
```

Tips on writing programs
Use command line args

- Command line arguments
  - binary “filename”
  - binary “filename” 100 200

- Don’t assume or hardcode a fixed filename!

- a.out (automatically reads in “image.pgm”) (this is very bad 😞)

Write useful executables

- Give the user a usage statement!

  - For example:
    - If you write a program that takes as input from the command line, a filename and a threshold number between [0-255]

    - If someone types “binary ”

    - Your program shouldn’t crash!!!

    - Your program should respond:
      - > USAGE: binary “filename” Threshold_Number
Handle bad filenames

- binary badfile.pgm
  - Don’t try to read from this file!!!
  - Have an error message
    - > Error when reading "badfile.pgm". Abort.

Sometimes errors cannot be avoided.
But, you should handle obvious user mistakes

Comment your code

- Give the general idea of your “logic” via the comments
- Especially tricky pieces of code!

```c
for(i=0; i < 256; i++)
  for(j=0; j < 256; j++)
    x = (((i & 0x8)==0)^ ((j & 0x8)==0))*255;
```

// statement produces a repeated checkerboard
// pattern such that:
// wb|wb|wb
// bw|bw|bw
// where "w" or "b" is an 8x8 block of all black or all white
Things to remember for IP

• Be mindful of your indices! You can get into memory problems very quickly

• If you get a SEGFAULT, you have memory problems; probably a bad index.
  (and it isn't the compiler or the computer)

• Don’t forget to free memory for “working buffers”/“temp images”

• Avoid “hard-coding” values!