Image Processing – The Programming Fundamentals

First Edition
Revision 1.0

A text to accompany

Image Apprentice

The C/C++ based Image Processing Learner's Toolkit.

Image Processing – The Programming Fundamentals

Prerequisites: A brief knowledge of C/C++ language(s).

This text describes the basic computer programming (C/C++) techniques required to begin practical implementations in Digital Image Processing. After reading the text, the reader would be in a position to understand and identify:

- How uncompressed images are stored digitally
- Typical data structures needed to handle digital image data
- How to implement a generic image processing algorithm as function

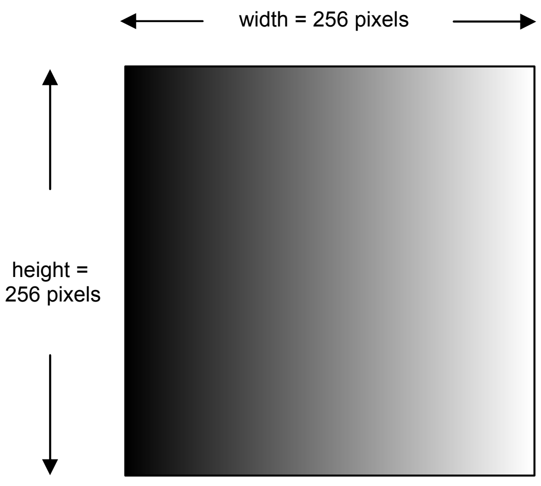
Mathematically, an image can be considered as a function of 2 variables, *f(x, y)*, where *x* and *y* are spatial coordinates and the value of the function at given pair of coordinates *(x, y)* is called the *intensity value*.

The programming counterpart of such a function could be a one or two dimensional array. **Code Snippet 1** and **Code Snippet 2** show how to traverse and use 1-D and 2-D arrays programmatically. Both of them essentially can represent an image as shown in **Figure 1**.

width = 256

0	1	2	3	4	5	. . . .	253	254	255
0	1	2	3	4	5	. . . .	253	254	255
0	1	2	3	4	5	. . . .	253	254	255
0	1	2	3	4	5	. . . .	253	254	255
⋮	⋮	⋮	⋮	⋮	⋮	⋮ . . . . ⋮	⋮	⋮	⋮
0	1	2	3	4	5	. . . .	253	254	255

(A)



(B)

Figure 1: Values in the grid represent the grayscale intensities of the image on the right.
(A) Intensitiy values shown in a grid of same dimension as image **(B)** Image as seen on monitor

```
//  Following code declares a 1-D array of type 'unsigned byte' having
//  size 256*256. It then puts values from 0 through 255 in each row.

int width, height;            // width and height of image
int offset;                   // num of elements traversed in array
int value;                    // image intensity value
width = height = 256;
value = offset = 0;
unsigned byte array_1D[height * width];

for(int j=0; j<height; j++)          // traverse height (or rows)
{
      offset = width * j;            // modify offset travelled
      for(int i=0; i<width; i++)     // traverse width (or columns)
      {
            array_1D[offset + i] = value++;     // update value at
                                                // current index  i.e.
                                                // (offset+i)
      }
      value = 0;
}
```

Code Snippet 1

```
//  Following code declares a 2-D array of type 'unsigned byte' having
//  size 256*256. It then puts values from 0 through 255 in each row.

int width, height;            // width and height of image
int value;                    // image intensity value
width = height = 256;
value = 0;
unsigned byte array_2D[height][width];

for(int j=0; j<height; j++)          // traverse height (or rows)
{
      for(int i=0; i<width; i++)     // traverse width (or columns)
      {
            array_2D[j][i] = value++;      // update value at
                                           // current  (i, j)
      }
      value = 0;
}
```

Code Snippet 2

The 'for' loop in **Code Snippet 1 and 2** can be visualized as shown in **Figure 2.**

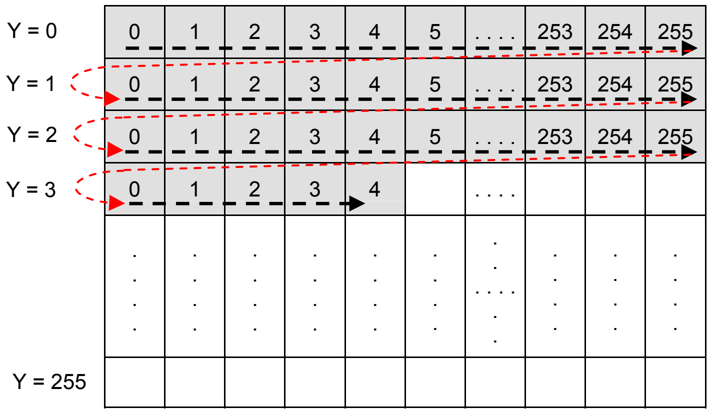


Figure 2: Array traversal in a 'for' loop. Note that rows are being accessed one after the other. This is known as '*Row Major Traversal*'. The graphic suggests that in the current iteration, x = 4 and y = 3 both starting from 0.

Dynamic Memory Allocation in C++

In **Code Snippet 1 and 2,** memory for the arrays is being allocated statically. This means that the compiler knows the array size during compilation process. However, in most image processing algorithms, the dimensions of the image (the width and height) are not known in the compile time. The application gets to know about it only in the run time when the user opens an image (i.e. supplies it to the program). This gives way to the need of allocating memory '*dynamically*'. Dynamic memory allocation in C/C++ requires the programmer to deallocate the memory too. The step by step methods to allocate/deallocate memory using C++ keywords 'new' and 'delete' are as follows:

1-D Array:

This is achieved using *a pointer to the appropriate data type* ('`unsigned byte`' in this case). Code other than that in **Code Snippet 1 and 2** is written in **bold** typeface.

```
int width, height;              // width and height of image
int offset;                     // num of elements traversed in array
int value;                      // image intensity value
value = offset = 0;

unsigned byte* array_1D = NULL;     // Pointer to unsigned byte

// Get width and height from user, say using scanf().
// This could have been from the user interface or just anywhere.
scanf("%d", &width);                // Get width from user input
scanf("%d", &height);               // Get height from user input

// Now we have the user supplied width and height
// Utilize user supplied width and height for dynamic allocation
array_1D = new unsigned byte[height*width];
if(array_1D == NULL) return;        // return if memory not allocated

for(int j=0; j<height; j++)         // traverse height (or rows)
{
      offset = width * j;           // modify offset travelled
      for(int i=0; i<width; i++)    // traverse width (or columns)
      {
            // update value at current index i.e. (offset+i)
            array_1D[offset + i] = value++;
      }
      value = 0;
}

// After complete usage, delete the memory dynamically allocated
delete[] array_1D;
```

Code Snippet 3

2-D Array:

This is achieved using *a pointer to a pointer (or, double pointer) to the appropriate data type* (‘`unsigned byte`’ in this case). Code other than that in **Code Snippet 1 and 2** is written in **bold** typeface.

```
int width, height;              // width and height of image
int value = 0;                  // image intensity value

unsigned byte** array_2D = NULL;     // Ptr to a Ptr to unsigned byte
// Following could have been from the user interface or just anywhere.
scanf("%d", &width);                 // Get width from user input
scanf("%d", &height);                // Get height from user input

// Allocation is going to be a 2 step process. First allocate a
// '1-D array of pointers' [NOTE THIS] of length 'height'
array_2D = new unsigned byte*[height];
if(array_2D == NULL) return;         // return if memory not allocated
// For each of these pointers, allocate memory of size 'width'
for(int j=0; j<height; j++)
{
      array_2D[j] = new unsigned byte[width];
      if(array_2D[j] == NULL) return;// return if memory not allocated
}

for(j=0; j<height; j++)         // traverse height (or rows)
{
      for(int i=0; i<width; i++)     // traverse width (or columns)
      {
            array_2D[j][i] = value++;// update value at current (i, j)
      }
      value = 0;
}

// After complete usage, delete the memory dynamically allocated
for(j=height-1; j>= 0; j--) delete[] array_2d[j]; //delete rows
delete[] array_2d;//delete the pointer to pointer
```

Code Snippet 4

Here's the explanation (**Figure 3**) with an example of a 2D array of width = 6 and height = 5:

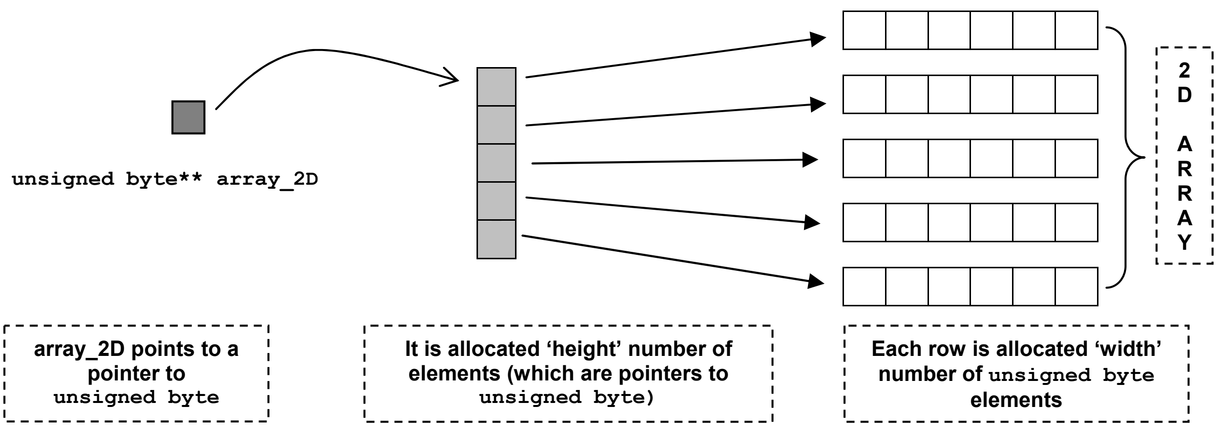
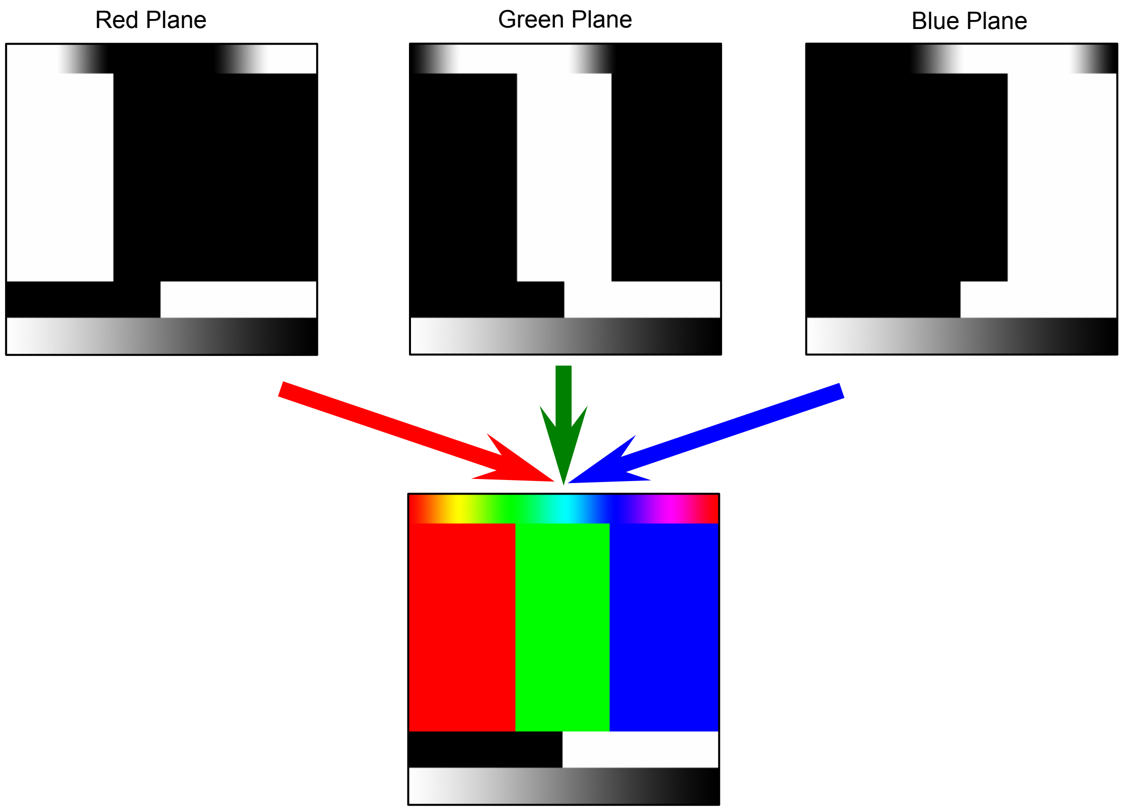

unsigned byte** array_2D
2 D ARRAY
array_2D points to a pointer to unsigned byte
It is allocated ‘height’ number of elements (which are pointers to unsigned byte)
Each row is allocated ‘width’ number of unsigned byte elements

Figure 3: Steps involved in dynamic memory allocation for a 2D array

Color Planes explained

Each basic colot (Red, Green and Blue) is displayed separately and is stored as an array of `unsigned byte` and hence the nomenclature 'Color Plane'. This means each of R, G, and B plane has range of [0, 255]. As **Figure 4** suggests, when the 3 planes get merged on the display device, they appear like a color image. If all the 3 color planes have same values at corresponding coordinates, the image gives the impression of a grayscale image. As there is same information in all the 3 planes, while storing a grayscale image we store the information of only 1 plane that is later replicated 3 times when displayed.



$B_{0,0}$	$G_{0,0}$	$R_{0,0}$	$B_{0,1}$	$G_{0,1}$	$R_{0,1}$	. . . .	$B_{0,w-1}$	$G_{0,w-1}$	$R_{0,w-1}$
$B_{1,0}$	$G_{1,0}$	$R_{1,0}$	$B_{1,1}$	$G_{1,1}$	$R_{1,1}$	. . . .	$B_{1,w-1}$	$G_{1,w-1}$	$R_{1,w-1}$
$B_{2,0}$	$G_{2,0}$	$R_{2,0}$	$B_{2,1}$	$G_{2,1}$	$R_{2,1}$	. . . .	$B_{2,w-1}$	$G_{2,w-1}$	$R_{2,w-1}$
$B_{3,0}$	$G_{3,0}$	$R_{3,0}$	$B_{3,1}$	$G_{3,1}$	$R_{3,1}$	. . . .	$B_{3,w-1}$	$G_{3,w-1}$	$R_{3,w-1}$
⋮	⋮	⋮	⋮	⋮	⋮	. . . .	⋮	⋮	⋮
$B_{h-1,0}$	$G_{h-1,0}$	$R_{h-1,0}$	$B_{h-1,1}$	$G_{h-1,1}$	$R_{h-1,1}$	. . . .	$B_{h-1,w-1}$	$G_{h-1,w-1}$	$R_{h-1,w-1}$

$B_{0,0}$	$G_{0,0}$	$R_{0,0}$	**$B_{0,1}$**	$G_{0,1}$	$R_{0,1}$	. . . .	$B_{0,n-1}$	$G_{0,n-1}$	$R_{0,n-1}$

Figure 4: **[A]** R, G, and B planes when merged optically on display device, give rise to a color image. **[B]** Layout in the memory while storing an RGB image in a: **(1)** 2D Array **(2)** 1D Array

Range of colors in various color depths

When displayed on a monitor, image data is sent as `unsigned byte`. The minimum and maximum values of an `unsigned byte` are 0 (all 8 bits are 0) and 255 (all 8 bits are 1). Hence an `unsigned byte` can have a maximum possible of 256 different values. Keeping this in mind, we have the following:

0 | 0 | 0 | 0 | 0 | 0 | 0 | 0	1 | 1 | 1 | 1 | 1 | 1 | 1 | 1
`unsigned byte`	`unsigned byte`
Decimal equivalent = 0 (i.e. minimum value)	Decimal equivalent = 255 (i.e. maximum value)

<u>24 Bit or RGB images</u>: Each component (R, G and B) constitute one `unsigned byte` and can have 256 different values. Hence the total number of colors in an RGB image can be 256*256*256 i.e. 16777216 or 16 million colors. *Bytes consumed per pixel* here is 3.

<u>8 Bit images</u>: As discussed above, they can have a maximum of 256 different shades. *Bytes consumed per pixel* here is 1.

<u>1 Bit images</u>: 2 possible values, either 0 or 1.

1D to 2D array copying (and vice versa)

At times situation may arise when we have to copy the contents of a 1D array to a 2D array. **Code Snippet 5** shows how to do that.

```
// This assumes that we have a defined and populated 1D array
// declared as 'array_1D' of type 'unsigned byte' (can be different)
// and has dimension width*bytesPerPixel*height
// where 'width' and 'height' are the physical size of the image.
// bytesPerPixel = 3 (if RGB or 24 Bit image)
// bytesPerPixel = 1 (if grayscale or 1 Byte image)

// Declare a double pointer of the type you want.
// This will point to the 2D array.
unsigned byte** array_2D;

// Now allocate memory as in Code Snippet 4
array_2D = new unsigned byte*[height];
if(array_2D == NULL) return;         // return if memory not allocated
// For each of these pointers, allocate memory of size 'width'
for(int j=0; j<height; j++)
{
      array_2D[j] = new unsigned byte[width*bytesPerPixel];
      if(array_2D[j] == NULL) return;// return if memory not allocated
}
```

Code Snippet 5 **(continued on next page…)**

```
// Copy contents from the existing 1D array named array_1D
int offset = 0;
for(j=0; j<height; j++)       // traverse height (or rows)
{
      offset = width * bytesPerPixel * j;
      for(int i=0; i<width*bytesPerPixel; i++)  // traverse width
      {
            array_2D[j][i] = array_1D[offset + i];    // update value at
                                                      // current (i, j)
      }
}

// After complete usage, delete the memory dynamically allocated
for(j=height-1; j>= 0; j--)
      delete[] array_2d[j]; //delete rows of pointers
delete[] array_2d; //delete the pointer to pointer
```

Code Snippet 5

Code Snippet 6 shows how to copy the contents of a 2D array to a 1D array:

```
// This assumes that we have a defined and populated 2D array
// declared as 'array_2D' of type 'unsigned byte' (can be different)
// and has size (width * bytesPerPixel) * height
// where 'width' and 'height' are the physical size of the image.
// bytesPerPixel = 3 (if RGB or 24 Bit image)
// bytesPerPixel = 1 (if grayscale or 1 Byte image)

// Declare a pointer of the type you want.
// This will point to the 1D array
unsigned byte* array_1D;

// Now allocate memory for array_1D as in Code Snippet 3
array_1D = new unsigned byte[height*width*bytesPerPixel];
if(array_1D == NULL) return;         // return if memory not allocated

// Copy contents from the existing 2D array named array_2D
int offset = 0;
for(int j=0; j<height; j++)          // traverse height (or rows)
{
      offset = width * bytesPerPixel * j;
      for(int i=0; i<width*bytesPerPixel; i++)  // traverse width
      {
            array_1D[offset + i] = array_2D[j][i];    // update value at
                                                      // current (i, j)
      }
}

// After complete usage, delete the memory dynamically allocated
delete[] array_1d; //delete the pointer to pointer
```

Code Snippet 6

Extracting individual R, G, B Planes from composite data

Assuming that one has a composite array of image data in the format BGRBGRBGR… at times there might arise need to process each color plane (Red, Green, and Blue) separately. Go through **Code Snippet 7** to see how to do this.

```
// This assumes that we have a defined and populated 1D array
// declared as 'array_1D' of type 'unsigned byte' (can be different)
// and has size width * bytesPerPixel * height
// where 'width' and 'height' are the physical size of the image.
// bytesPerPixel = 3 (as the it is an RGB or 24 Bit image)

// Declare three 2D arrays to store R,G, and B planes of image.
unsigned byte **arrayR_2D, **arrayG_2D, **arrayB_2D;

// Now allocate memory for array_1D as in Code Snippet 3
// To avoid repetition, we create a function to do this, Code Snippet 8.
arrayR_2D = Allocate2DArray(arrayR_2D, width, height);
arrayG_2D = Allocate2DArray(arrayG_2D, width, height);
arrayB_2D = Allocate2DArray(arrayB_2D, width, height);

// return if memory not allocated
if(arrayR_2D == NULL || arrayG_2D == NULL || arrayB_2D == NULL) return;

// Extract R,G,B planes from the existing composite 1D array
int offset = 0;
int counter = 0;
for(int j=0; j<height; j++)          // traverse height (or rows)
{
       offset = width * bytesPerPixel * j;
       for(int i=0; i<width*bytesPerPixel; i+=bytesPerPixel) // width
       {
              arrayB_2D[j][counter++] = array_1D[offset + i+0];
              arrayG_2D[j][counter++] = array_1D[offset + i+1];
              arrayR_2D[j][counter++] = array_1D[offset + i+2];
       }
       counter = 0;
}

// After complete usage, delete the memory dynamically allocated
DeAllocate2DArray(arrayR_2D, height);
DeAllocate2DArray(arrayG_2D, height);
DeAllocate2DArray(arrayB_2D, height);
```

Code Snippet 7

To put 3 arrays containing R, G, and B data into one single composite array in the BGRBGRBGR… order, one just has to modify the lines inside the 2 for loops in Code Snippet 7 as below:

```
array_1D[offset + i+0] = arrayB_2D[j][counter++];
array_1D[offset + i+1] = arrayG_2D[j][counter++];
array_1D[offset + i+2] = arrayR_2D[j][counter++];
```

Code Snippet 8 has dynamic memory allocation in form of a function. It is always a good practice to write reusable functions in programming.

```
unsigned byte** Allocate2DArray(unsigned byte** buffer, int w, int h)
{
      // Allocate memory as in Code Snippet 4
      buffer = new unsigned byte*[h];
      if(buffer == NULL) return;    // return if memory not allocated
      // For each of these pointers, allocate memory of size 'w'
      for(int j=0; j<h; j++)
      {
            buffer[j] = new unsigned byte[w];
            if(buffer[j] == NULL) return;// return if not allocated
      }
      return buffer;
}

void DeAllocate2DArray(unsigned byte** buffer, int h)
{
      // After complete usage, delete the memory dynamically allocated
      for(int j=h-1; j>= 0; j--) delete[] buffer[j]; //delete rows
      delete[] buffer; //delete the pointer to pointer
}

int main()
{
      // Declare a 2D array.
      unsigned byte **array;

      // Get width and height from user, say using scanf().
      // Following could have been from the user interface
      // or just anywhere.
      scanf("%d", &width);          // Get width from user input
      scanf("%d", &height);         // Get height from user input

      // now call the function to allocate memory in 2D.
      array = Allocate2DArray(array, width, height);

      // return if memory not allocated
      if(arrayR_2D == NULL || arrayG_2D == NULL || arrayB_2D == NULL)
            return;

      // Now you can use the new 2D array in a way you like

      // After complete usage, delete the memory dynamically allocated
      DeAllocate2DArray(arrayR_2D, height);

      return 0;
}
```

Code Snippet 8

Normalization of data to the range [0,255]

At times, you have data that is not in the range [0, 255] but in some other range. As computer displays need data in the range [0, 255], the out-of-range data needs to be *normalized* before it can be sent to the display device.

```
void GetMinMax(double* buffer, int size, int* min, int* max)
{
	for(int i=0; i<size; i++)
	{
		if(data[i] > *max) *max = data[i];
		if(data[i] < *min) *min = data[i];
	}
}

// This normalizes between range [0, 255]
void Normalize(double* data, int size)
{
	double max, min;
	min = max = data[0];

	GetMinMax(data, size, &min, &max);

	for(int i=0; i< size; i++)
	{
		data[i] = (255.0*(data[i]-min)) / (max-min);
	}
}

int main()
{
	// Suppose we have a populated array, 'array_1D', of type double
	// and length 'size' and it has range other than [0, 255]
	Normalize(array_1D, size);

	// after this call, it has been normalized to range [0, 255]
}
```

Code Snippet 9

References

1. Image Apprentice, The C/C++ based Image Processing Learner's Toolkit; http://www.adislindia.com
2. 5 step guide to build Image Apprentice Plugins; http://adislindia.com/documents/5_step_guide.pdf
3. C++ Language Tutorial; http://www.cplusplus.com/doc/tutorial/