EECS 22: Assignment 4

Prepared by: Huan Chen, Prof. Rainer Doemer

10/28/2016

Due on Thursday 11/17/2016 6:00pm.

Contents

1 Digital Image Processing (DIP) 1

1.1 Initial Setup . . . 2

1.2 Add Support for Different Image Sizes . . . 2

1.2.1 Use Pointers in One Dimensional Memory Space instead of Arrays with Two Dimensions . . 2

1.2.2 The Image.c Module . . . 2

1.2.3 Read and Save Image Files . . . 3

1.2.4 Modify HW2 and HW3's DIP Function Implementations . . . 4

1.2.5 Test.c Module . . . 5

1.3 Advanced DIP operations . . . 5

1.3.1 Overlay . . . 5

1.3.2 Rotate . . . 6

1.3.3 Crop . . . 8

1.3.4 Resize . . . 9

1.3.5 MetalCircle . . . 10

1.4 Test All Functions . . . 11

1.5 Extend the Makefile . . . 11

1.6 Use "Valgrind" Tool to Find Memory Leaks and Invalid Memory Accesses . . . 11

2 Implementation Details 12

2.1 Function Prototypes . . . 12

2.2 Pass in the Pointer of the struct IMAGE . . . 12

3 Budgeting Your Time 13

4 Script File 13

5 Submission 13

6 Grading (100 pts + 10 pts Bonus) 14

1 Digital Image Processing (DIP)

In this assignment, you will learn to use dynamic memory allocation. Based on the program *PhotoLab* for Assignment 3, you will redesign your DIP operations to support varying image sizes. Then, you will add more DIP operations whose resulting images will differ in size compared to the original one. Thus you can use your *PhotoLab* program to perform the DIP operations on any of your own pictures.

1.1 Initial Setup

You can reuse and adjust your hw3's DIP implementations for this assignment (if it is bug free).
Alternatively, make a directory *hw4* and copy the following files from directory *~eecs22/public* to it.

```
mkdir hw4
cd hw4
cp ~eecs22/public/Image.h .
cp ~eecs22/public/FileIO.h .
cp ~eecs22/public/FileIO.c .
cp ~eecs22/public/Constants.h .
cp ~eecs22/public/Test.h .
cp ~eecs22/public/Test.c .
cp ~eecs22/public/EH.ppm .
cp ~eecs22/public/balloon.ppm .
```

- **Image.h** is the header file for the new structure definition and pixel mapping functions declarations, used in Section 1.2.2.
- **Test.h**, **Test.c** contains `AutoTest(void)`, should be used by `main()`.

1.2 Add Support for Different Image Sizes

To support varying image sizes, we cannot define and pass three fixed size arrays to the DIP operation functions as in previous assignments. Instead, we need to use dynamic memory allocation to claim three blocks of memory whose size will be decided at run time, then pass the pointers pointing to an image structure to the DIP functions, since the size of the input image cannot be determined at compile time.

1.2.1 Use Pointers in One Dimensional Memory Space instead of Arrays with Two Dimensions

In this assignment, we use three pointers to type *unsigned char* for the color intensity values for each pixel instead of three fixed sized arrays. However, pointers only point to a memory space in one dimension. Therefore, we need to map 2-tuple coordinates of pixels to a single value index corresponding to the pixel color information in memory.

For example, we have an image of size 10×5, and three pixels (0, 0), (9, 4), and (6, 4). We assume **row major** for the image storage in this program. Therefore, the index value for pixel (0, 0) in the one dimensional storage space will be 0; the index value for pixel (9, 4) in the one dimensional storage space will be $49 = 9 + 4 * 10$; and the index value for pixel (6, 4) in the one dimensional storage space will be $46 = 6 + 4 * 10$.

In general, the index value for the pixel (*x*, *y*) in an image of size **WIDTH**×**HEIGHT** in the one dimensional storage space will be *x* **+** *y* *** WIDTH**.

1.2.2 The Image.c Module

Please implement module **Image.c** (see provided **Image.h**) to handle basic operations on the image. **Image.h** is the header file for the new structure definition and pixel mapping functions declarations.

- The **IMAGE** struct: used to aggregate all the information of an image, defined in **Image.h**.

```
typedef struct {
  unsigned int Width;  /* image width */
  unsigned int Height; /* image height */
  unsigned char *R;    /* pointer to the memory storing all the R intensity */
                       /* values */
  unsigned char *G;    /* pointer to the memory storing all the G intensity */
                       /* values */
  unsigned char *B;     /* pointer to the memory storing all the B intensity */
                        /* values */
} IMAGE;
```

- Define the functions to get and set the value of the color intensities of each pixel in the image. Please use the following function prototypes (provided in **Image.h**) and define the functions properly in **Image.c**.

  ```
  /* Get the color intensity of the Red channel of pixel (x, y) in image */
  unsigned char GetPixelR(const IMAGE *image, unsigned int x, unsigned int y);

  unsigned char GetPixelG(const IMAGE *image, unsigned int x, unsigned int y);

  unsigned char GetPixelB(const IMAGE *image, unsigned int x, unsigned int y);

  /* Set the color intensity of the Red channel of pixel (x, y) in image to r */
  void SetPixelR(IMAGE *image, unsigned int x, unsigned int y, unsigned char r);

  void SetPixelG(IMAGE *image, unsigned int x, unsigned int y, unsigned char g);

  void SetPixelB(IMAGE *image, unsigned int x, unsigned int y, unsigned char b);
  ```

  The mapping from the 2-tuple coordinates (x, y) to the single index value for the one dimensional memory space will be taken care of in these functions. Please call these functions in your DIP functions for setting / getting the intensity values of the pixels.
  NOTE: By using pointers in one dimensional memory space, you need to modify the statements in your functions for array elements' indexing with the pixel setting/getting functions accordingly. For example:

  - In Assignment 3, we got the pixel's color value by indexing the element from the two-dimensional array:
    tmpR = R[x][y];
  - Now, we need to get the pixel's color value by calling the getting function:
    tmpR = GetPixelR(image, x, y);
  - In Assignment 3, we set the pixel's color value by indexing the element from the two-dimensional array:
    R[x][y] = r;
  - Now, we need to set the pixel's color value by calling the setting function:
    SetPixelR(image, x, y, r);

  By using the setting/getting functions, we can keep the two-dimensional coordinate system as in Assignment 2 and Assignment 3.

  Please make sure to include the header file **Image.h** properly in your source code files and header files.

- Add **assertions** in these functions to make sure the input image pointer is valid, and the set of pointers to the memory spaces for the color intensity values are valid too. Last but not least, add assertions to ensure that the coordinates are within the valid ranges for the image.

- Please extend/adjust your **Makefile** accordingly: 1) add the target to generate **Image.o** and **Test.o** and 2) add **Image.o** and **Test.o** when generating **PhotoLab** and **PhotoLabTest**.

1.2.3 Read and Save Image Files

Refer to **FileIO.h** for the defined functions for file I/Os.

- `IMAGE *LoadImage(const char *fname)`
  This function reads the file *fname.ppm* and returns the image pointer if loaded successfully, otherwise returns

NULL. The color intensities for channel red, green, and blue are stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of the returned IMAGE pointer respectively. The memory space of the image is created in this function by a function call to CreateImage(), see below.

- `int SaveImage(const char *fname, const IMAGE *image)`
  This function saves the color intensities of the red, green, and blue channel stored in the memory spaces pointed to by member pointers *R*, *G* and *B* of *image* into the file *fname.ppm*. This function returns an error code if something goes wrong. Handle it by letting the user know that the image was not saved.

Please implement the two functions to handle the memory allocation and deallocation in **Image.c**, declared in **Image.h**.

```
/* Allocate the memory space for the image structure         */
/* and the memory spaces for the color intensity values.     */
/* Return the pointer to the image, or NULL in case of error */
IMAGE *CreateImage(unsigned int Width, unsigned int Height);

/* Release the memory spaces for the pixel color intensity values */
/* Deallocate all the memory spaces for the image                 */
/* Set R/G/B pointers to NULL                                     */
void DeleteImage(IMAGE *image);
```

IMPORTANT: The LoadImage() function needs the CreateImage() function inside to allocate the memory space. Therefore, you should implement the CreateImage() and DeleteImage() functions correctly before you use the LoadImage() and SaveImage() functions. `malloc()` and `free()` must be called within CreateImage() and DeleteImage() only.

1.2.4 Modify HW2 and HW3's DIP Function Implementations

Most of our functions need to be refined by taking the IMAGE structure as a parameter which contains all the information about the image, i.e. your DIP function prototypes should look like below:

```
/* DIPs.h */
IMAGE *Negative(IMAGE *image);
IMAGE *ColorFilter(IMAGE *image, int target_r, int target_g, int target_b,
  int threshold, int replace_r, int replace_g, int replace_b);
IMAGE *Edge(IMAGE *image);
IMAGE *HFlip(IMAGE *image);
IMAGE *Vmirror(IMAGE *image);
IMAGE *Zoom(IMAGE *image);

/* Advanced.h */
IMAGE *Shuffle(IMAGE *image);
IMAGE *Posterize(IMAGE *image, int rbits, int gbits, int bbits);
IMAGE *AddNoise(IMAGE *image, int n);
IMAGE *Overlay(IMAGE *inputImage, const IMAGE *overlayImage, int x_offset, int y_offset,
  unsigned char backgroundR, unsigned char backgroundG, unsigned char backgroundB);
IMAGE *Rotate(IMAGE *image, int degree);
IMAGE *Crop(IMAGE *image, int x, int y, int W, int H);
IMAGE *MetalCircle(IMAGE *image, int centerX, int centerY, int outerRadius,
  int borderWidth);
```

IMPORTANT: Note the changes in the return types and the function arguments!
NOTE: Add **assertions** in **ALL** these DIP functions to make sure the input image pointer is valid.

1.2.5 Test.c Module

Include **Test.h** in your **PhotoLab.c**, otherwise `AutoTest(void)` cannot be called in `main()`, you should use the provided `AutoTest(void)`.

1.3 Advanced DIP operations

In this assignment, implement the advanced DIP operations described below in **Advanced.c** (**Advanced.h**).

The user should be able to select DIP operations from a menu as the one shown below:

```
Please make your choice:
--------------------------------
1:  Load a PPM image
2:  Save an image in PPM and JPEG format
3:  Make a negative of an image
4:  Color filter an image
5:  Sketch the edge of an image
6:  Flip an image horizontally
7:  Mirror an image vertically
8:  Zoom an image
9:  Add noise to an image
10:  Shuffle an image
11:  Posterize an image
12:  Overlay
13:  Rotate clockwise
14:  Crop
15:  Resize
16:  MetalCircle
17:  Test all functions
18:  Exit
```

NOTE: Your program should:

- print ”No image to process!” when menu item 2 - 16 is chosen but the input image pointer is NULL (such as loading image failed or no image is loaded).
- print ”Invalid selection!” when none of menu item 1 - 18 is chosen.
- print ”You exit the program.” and exit properly whenever the user inputs 18.
- print ”AutoTest failed, error code RC.” (replace RC with return code from `AutoTest`) if `AutoTest` returns a non-zero code, otherwise print ”AutoTest finished successfully.”.

There should be no memory leaks when:

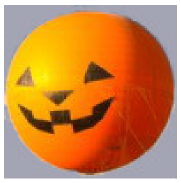
- the user chooses to load an image multiple times then exit menu without doing any DIPs.
- the user chooses to load an image and choose some DIPs then exit menu without saving it to file.

1.3.1 Overlay

This function overlies the current image with a second image. In our program, we will put a balloon image on the original image.

First, you load the second image before the overlay function (use *LoadImage(”balloon”)* in our case), without prompting the use to input the overlay image name. In this assignment, the second image is **balloon.ppm** (98×98 pixels, smaller than the original image). Then prompt the user to enter the position of the overlay with coordinates (x, y) and

(a) Original image


(b) A second image

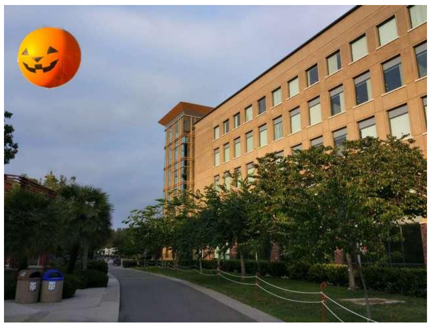

(c) Overlay the image at (20, 30)

Figure 1: An image and the overlayed image.

the background R/G/B values of the second image to be ignored.

Take a look at Fig. 1(b). To achieve the overlay effect, we should not select the background of balloon image (namely the cloudy sky area, as it does not match well with the original image's blus sky background). Instead, treat it as transparent color. That is, each of the non-background pixels in the second image will be overlaid to a position in the original image, while the background pixels will not. Check whether a pixel in image Fig. 1(b) is a background pixel or not by the RGB values of this pixel. More specifically for the balloon image, if a pixel's RGB values satisfy ($|r - backgroundR| \leq 5$, $|g - backgroundG| \leq 5$, $|b - backgroundB| \leq 5$), consider it as the cloudy sky background pixel, this pixel should not be put onto the original image. Here we set background R/G/B to 164/163/179, threshold to 5 (actually, this is a color filter operation).

Implement the following function to do this DIP.

```
IMAGE *Overlay(IMAGE *inputImage, const IMAGE *overlayImage, int x_offset, int y_offset,
  unsigned char backgroundR, unsigned char backgroundG, unsigned char backgroundB);
```

Here, x_offset, y_offset are the overlay cordinates, *backgroundR*, *backgroundG*, *backgroundB* are the background color to be ignored in the second image when it is overlaid to the original image.

NOTE: the 'const' indicates the overlay image cannot be modified. Pay attention to the coordinates after applying the offset values! Any pixel outside of the valid ranges of the original image should be ignored (otherwise invalid memory access will occur).

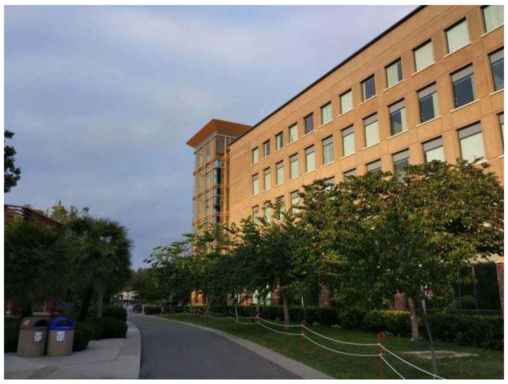
Once user chooses this option, your program's output wll be like:

```
Please make your choice: 12
Please input X coordinate of the overlay image: 20
Please input Y coordinate of the overlay image: 30
Please input background R: 164
Please input background G: 163
Please input background B: 179
balloon.ppm was read successfully!
"Overlay" operation is done!
/* ... print menu again and wait for the user's next input */
```

The image is saved with the name 'overlay' after this step.

1.3.2 Rotate

This function rotates the image by 90, 180 or 270 degrees clockwise. The size of the image will be the same, but the width (height) of the new image may be the same as its original height (width).


(a) Original image

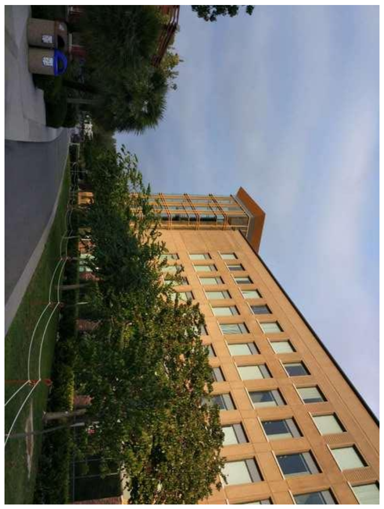
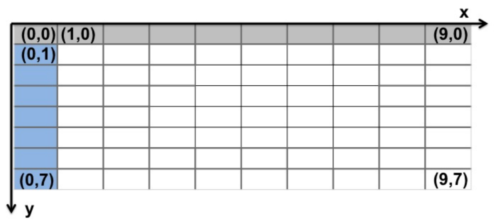
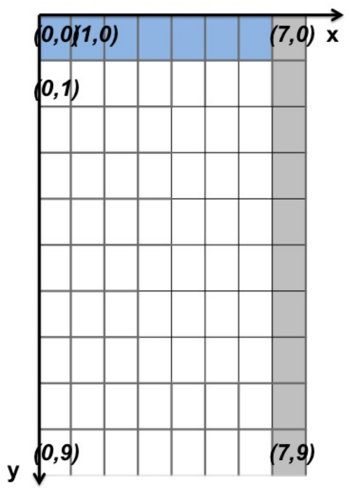

(b) Rotated image

Figure 2: An image and its rotated counterpart.

NOTE: As shown in Fig. 3, row pixel indices increase as they go down while the column pixel indices increase as they go to the right. Pixel indices are integer values ranging from 0 to the length of the row or column minus one. The top left pixel's coordinate is (0, 0), and the bottom right pixel's coordinate is (image→Width - 1, image→Height - 1).



(a) Coordinates for the original image



(b) Coordinates for the rotated image

Figure 3: Coordinates of the original image and its rotated counterpart.

First, you need to find how the pixel coordinates map from the original image (x, y) to the rotated image (x', y'). Then, you need to set the color of the pixel at (x', y') in the new image to the color of the pixel at (x, y) in the original image.

You need to implement the following function to do this DIP.

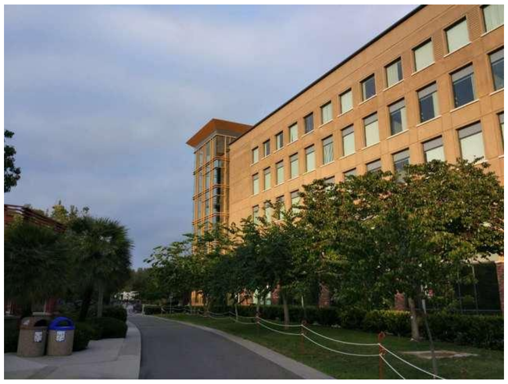
```
IMAGE *Rotate(IMAGE *image, int degree);
```

Fig. 2 shows an example of rotating 90 degree clockwise. Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 13
Please input the rotate degree (90, 180 or 270): 90
"Rotate" operation is done!
/* ... print menu again and wait for the user's next input */
```

The image is saved with the name 'rotate' after this step.

1.3.3 Crop


(a) Original image

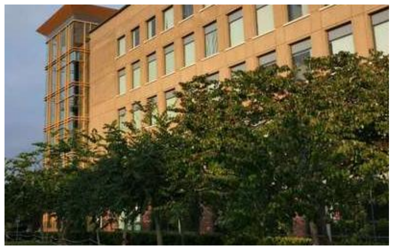

(b) Cropped image

Figure 4: The original image and the cropped EH image.

This function crops the image based on a set of user inputs. The user will indicate a starting pixel in the image by entering an x and y offset. Then the user specifies the size of cropping by entering how many pixels the user wants to crop in the x and y directions. If the crop amount exceeds the image width or height (or both), the returned image will only crop up to the maximum length of the original image.

NOTE: This means that the picture should only allocate the minimum amount of memory needed to store the image.

You need to implement the following function to do this DIP.

```
IMAGE *Crop(IMAGE *image, int x, int y, int W, int H);
```

Fig. 4 shows an example of this operation (the size of the cropped image is 400×250). Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 14
Please enter the X offset value: 200
Please enter the Y offset value: 150
Please input the crop width: 400
Please input the crop height: 250
"Crop" operation is done!
/* ... print menu again and wait for the user's next input */
```

The image is saved with the name 'crop' after this step.

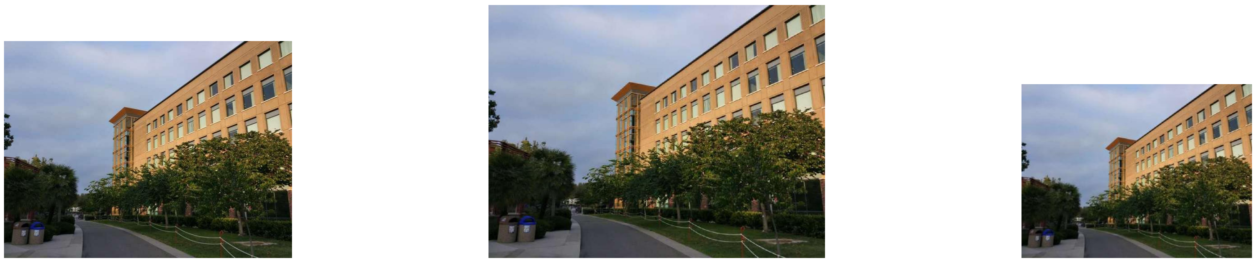

(a) Original image
(b) resized to a bigger image (percentage = 170)
(c) resized to a smaller image (percentage = 60)

Figure 5: An image and its resized bigger and resized smaller counterparts.

1.3.4 Resize

You need to implement the following function for this DIP.

```
IMAGE *Resize(IMAGE *image, int percentage);
```

This function resizes the image with the scale of *percentage*.

- *percentage* $==$ 100, the size of the new image is the same as the original one.
- *percentage* $<$ 100, the size of the new image is smaller than the original one.
- *percentage* $>$ 100, the size of the new image is larger than the original one.

More specifically, we scale *percentage* as follows:

- $Width_{new}$ = $Width_{old}$ * (percentage / 100.00);
- $Height_{new}$ = $Height_{old}$ * (percentage / 100.00);

If *percentage* is greater than 100, we need to duplicate some pixels from the original image to the new larger one. Assume (x', y') are the coordinates for the position of the pixel in the new image while (x, y) are the coordinates for the position of the pixel in the original image. Then, copy the color of the pixel(x, y) in the original image to pixel (x', y') in the new image Note that:

```
    x' = x * (percentage / 100.00);
    y' = y * (percentage / 100.00);
```

If *percentage* is less than 100, we will have fewer pixels in the new smaller image than in the original image. Therefore, we need to average the values of the color intensities of multiple pixels from the original image. Otherwise, we lose too much information from the original image. We use this average value as the color intensity of the pixel in the smaller image.

To demonstrate, each grid element represents one pixel in the image, as shown in Figure 6. We **average** the value of the color intensities for all the red edged pixels in the original image (from (x_1, y_1) to ($x_2 - 1$, $y_2 - 1$)) and use this average as the red color intensity of the pixel (x, y) in the new image, where:

```
    x1 = x / (percentage / 100.00);
    y1 = y / (percentage / 100.00);
    x2 = (x + 1) / (percentage / 100.00);
    y2 = (y + 1) / (percentage / 100.00);
```

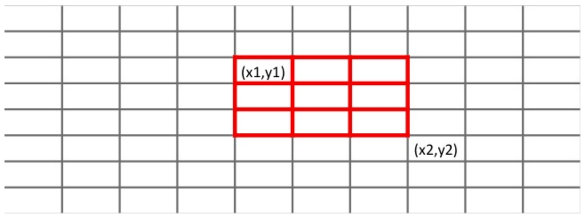


Figure 6: Pixels mapping from the bigger original image to the smaller new image

HINT: For enlarging the image, it is easier to iterate over the target image (not over the original image). **NOTE:** The *Resize()* function will consume the input image and return a new image with the new size. Please delete and create the image data structures properly in this function.

Figure 5 shows an example of this operation. Once the user chooses this option, your program's output should like this:

```
Please make your choice: 15
Please input the resizing percentage (integer between 1~500): 170
"Resizing the image" operation is done!
/* ... print menu again and wait for the user's next input */
```

These two images are saved after this operation:

1. 'bigresize': a bigger image with scale *percentage* = 170.
2. 'smallresize': a smaller image with scale *percentage* = 60.

1.3.5 MetalCircle

(a) Original image

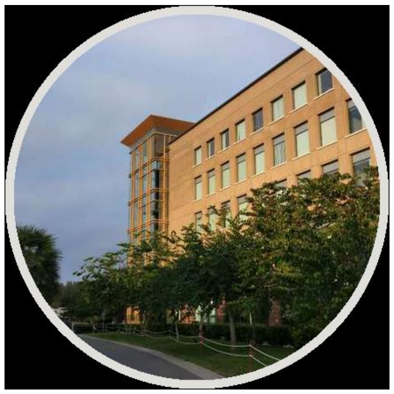

(b) Metal circle image

Figure 7: The original image and the metal circle EH image.

This function adds a metal circle to the image and puts the output image in a square. The user specifies the pixel (by inputing x, y offsets) as the center of the circle, inputs the outer radius and the circle's border width. Finally, the circle should be in the square.

The distance between pixels is defined as $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$. For each pixel in original image, if its distance to the center pixel is greater than or equal to outer radius, set it to black; else if its distance is greater than or equal to inner

radius (inner radius = outer radius - border width), set it to metal color (r = 224, g = 223, b = 219). The output square image's width and height should both be (2 * outerRadius + 1). For example, when outer radius = 220, both output image's width and height should be 441, since circle center pixel takes up one position. For example, the distances between R[x][y] and R[x][y - 220], between R[x][y] and R[x][y + 220] are both 220, however, this circle contains 441 rows from row [y - 220] to row [y + 220].
Implement this function:

```
IMAGE *MetalCircle(IMAGE *image, int centerX, int centerY,  int outerRadius,
  int borderWidth);
```

Fig. 7 shows an example of this operation. Once the user chooses this option, your program's output should look like this:

```
Please make your choice: 16
Please enter the X offset value: 200
Please enter the Y offset value: 150
Please input the outer radius: 220
Please input the border width: 10
"MetalCircle" operation is done!
/* ... print menu again and wait for the user's next input */
```

The image is saved with the name 'circle' after this step.

1.4 Test All Functions

Use the provided `AutoTest(void)` function to test all the DIP operations.

1.5 Extend the Makefile

- Add *Image.h*, *Image.c* to your **Makefile** and adjust it properly.
- Add *Test.h*, *Test.c* to your **Makefile** and adjust it properly.
- Generate 2 executable programs
  1. *PhotoLab* with the user interactive menu and the DEBUG mode off.
  2. *PhotoLabTest*, an executable that just calls *AutoTest(void)* function (with the DEBUG mode on).

  Define two targets to generate these 2 programs respectively in addition to *all* and *clean*. You may define other targets as needed.

1.6 Use "Valgrind" Tool to Find Memory Leaks and Invalid Memory Accesses

Valgrind is a multipurpose code profiling and memory debugging tool for Linux. It allows you to run your program in *Valgrind*'s own environment that monitors memory usage, such as calls to malloc and free. If you use uninitialized memory, write over the end of an array, or forget to free a pointer, *Valgrind* will detect it. You may refer to `http://valgrind.org/` for more details about the *Valgrind* tool.

In this assignment, please use the following command to check the correctness of your memory usages:

```
  valgrind --leak-check=full program_name
```

If there is no problem with the memory usage in your program, you will see information similar to the following upon completion of your program:

```
==xxxxx==
==xxxxx== HEAP SUMMARY:
==xxxxx==     in use at exit: 0 bytes in 0 blocks
==xxxxx==   total heap usage: 129 allocs, 129 frees, 20,476,437 bytes allocated
==xxxxx==
==xxxxx== All heap blocks were freed -- no leaks are possible
==xxxxx==
==xxxxx== For counts of detected and suppressed errors, rerun with: -v
==xxxxx== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 6 from 6)
```

Compile your program with the *"-g"* option in *gcc* to enable detection of memory usage problems in your program.

If there are problems with your program's memory usage, *Valgrind* will provide you with information about the problem and where to fix it.

For your final submission, your program should be free of warnings and free of any errors reported by *Valgrind*.

2 Implementation Details

2.1 Function Prototypes

Please implement **all** functions in **Advanced.c** and **Image.c**, adjust `main()`:
You can define other help functions as needed. **Do not** modify the provided function definitions in *Image.h* and `AutoTest(void)` in *Test.c*, as well as *Test.h*.

2.2 Pass in the Pointer of the struct IMAGE

In the main function, define the struct variable *image* of type IMAGE. It will contain the following image information: *Width*, *Height*, pointers to the memory spaces for all the color intensity values of the *R*, *G*, *B* channels.

When any of the DIP operations are called in the main function, the address of this *image* variable is passed into the DIP functions. Thus, the DIP functions can access and modify the contents of this variable.

In your DIP function implementation, there are two ways to save the target image information. Choose the better and easier one.

Option 1: Using Local Variables Define local variables of type IMAGE to save the target image information. For example:

```
IMAGE *DIP_function_name(IMAGE *inputImage)
{
  IMAGE *outputImage = NULL;
  outputImage = CreateImage(...);
  ...
  DeleteImage(inputImage);
  inputImage = NULL;
  return outputImage;
}
```

Make sure you create and delete the image space properly.

Then, at the end of each DIP function implementation, you can copy the data in *outputImage* over to *inputImage*, or delete the incoming image and return the new one.

Option 2: in Place Manipulation Sometimes you do not have to create new local array variables to save the target image information. Instead, you can just manipulate on *image→R*, *image→G*, *image→B* directly. For example, in the implementation of the Negative() function, you can assign the result pixel value directly back to the pixel entry.

NOTE: Please always call SetPixelR (SetPixelG, SetPixelB) function to set the pixel color value and GetPixelR (GetPixelG, GetPixelB) function to read the pixel color value.

3 Budgeting Your Time

This assignment's workload is **heavier** than the previous one. Suggested steps:

- Week 1:
  1. Change the implementations of HW2 and HW3's DIP functions to fit this assignment.
  2. Adjust the **Makefile** with the targets for the new module.
  3. Implement functions in **Image.c**.
  4. Implement one advanced DIP function if possible.
- Week 2:
  1. Implement all the advanced DIP functions.
  2. Use *Valgrind* to check memory usages. Fix any errors and warnings if any complained by *Valgrind*.
  3. Script the result of your programs and submit your work.

IMPORTANT: As 11/11/2016 is holiday, no labs, so utilize your time well on 11/04/2016.

4 Script File

To demonstrate that your program works correctly, perform the following steps and submit the log as your script file:

1. Type `script` to start your script.
2. Type `make` to generate *PhotoLabTest* and *PhotoLab*.
3. Type `./PhotoLab`, input 17 to run `AutoTest` then input 18 to exit.
4. Type `valgrind --leak-check=full PhotoLabTest` to run *PhotoLabTest* under *Valgrind*.
5. Type `make clean` to clean all the object files, generated .ppm files and executable programs.
6. Type `exit` to stop the script.
7. Type `mv typescript PhotoLab.script` to rename the script file as required.

NOTE: make sure you use exactly the same names as shown in the above steps when saving modified images! The script file is important, and will be checked in grading; you must follow the above steps to create the script file. ***Please don't open any text editor while scripting !!!***

5 Submission

Go to the parent directory of your *hw4* folder, turn in your homework by running:

```
~eecs22/bin/turnin.sh
```

Your *hw4* folder should contain *PhotoLab.script*, *PhotoLab.txt* (used to **briefly** describe your implementations), *PhotoLab.c*, *Image.c*, *Image.h*, *Constants.h*, *DIPs.c*, *DIPs.h*, *FileIO.c*, *FileIO.h*, *Advanced.c*, *Advanced.h*, *Makefile*, *Test.h*, *Test.c*.

6 Grading (100 pts + 10 pts Bonus)

Scores breakdown

- Makefile (10 pts)
- CreateImage, DeleteImage (5 pts each, 10 pts, points will be deducted if no proper error handling in CreateImage or failing to set R/G/B pointers to NULL in DeleteImage)
- GetPixelR, GetPixelG, GetPixelB, SetPixelR, SetPixelG, SetPixelB (2 pts each, 12 pts)
- HW2 and HW3's DIP functions reimplementation (2 pts each, 18 pts)
- Overlay, Rotate, Crop (10 pts each, 30 pts)
- Resize (5 pts for smallresize, 5 pts for bigresize) (10 pts)
- Menu (5 pts)
- Valgrind 0 errors for *PhotoLabTest* (5 pts)
- MetalCircle (Bonus: 10 pts)

NOTE: Partial credit based on quick code review will be given if:

- `make` failed, or target *PhotoLabTest* or *PhotoLab* cannot be generated, or any errors or warnings arise
- the output image (ppm or jpg) is incorrect
- the menu is incorrect (such as cannot exit properly, or `AutoTest` function failed, or memory leaks under the test cases described in Section 1.3)
- Valgrind generates errors

In these cases, grading will rely solely on a brief source code review with a rough estimate of how much usable code exists. The score received is not negotiable.

TIPS: Each of the above DIPs in `AutoTest` generates a definite output, except for `Shuffle` and `AddNoise` (with random results). Thus, use `diff` command to verify your output images. For example:

```
diff ~eecs22/public_html/negative.jpg ~YourID/public_html/negative.jpg
```

It is your responsibility to verify the output images are correct. Failing to do so may cause you lose credits.