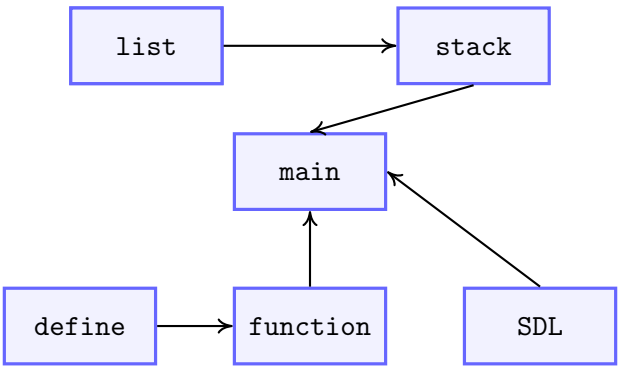
PRIM Project - Small Graphic Language

Richard CHEAM

Table of contents

---

1 Structural designation of the program

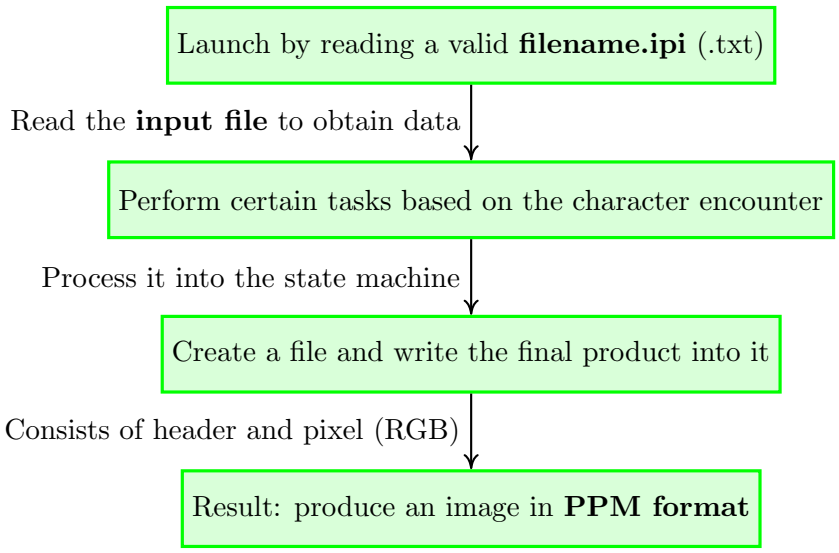


The program contains 5 modules comprising:

- `define`: contains each data structure of the program which tells how each data type is made and is needed for `function`.
- `list`: contains the structure of linked list which are needed for the implementation of `stack`.
- `stack`: since stack is implemented with linked list in this program, this module requires the interface of the `list` module for the linked list structure.

- `function`: describes numerous of functions which are needed to run the program as well as of the implementation of each function and is needed for `main`.
- `SDL`: this library *#include <SDL2/SDL.h>* allows the program to perform certain functions in order to produce the graphical interface which demonstrates the construction of the image to the user, but beware that in order to use this library, the existence of the *include* and *lib* folder is taken into account. They are both need to be with the project folder.
- `main`: comprises of the code which allows the program to read the input file precisely **filename.ipi**, read each letter in the input file and do certain task and write the final output as ppm file.

2 Synopsis of the program's operation



A brief description of the project/program context:

The program prompts the user to input the **file.ipi(txt)** which contains the number on the first line which is the size of the image to be produced, and as the image is squared, then the width and the height of the image is equal. The next line of the **input file** consists of characters that the program has to read the file and get those characters. As a result the program will perform a certain tasks for each letter it encounters. For instance, as there are **20 characters** in total that the program could read and perform the given task; so if the character is 'r' the program will then add the red color into the bucket of colors of the machine. Hence, in each procedure, the program building an image in baby steps. Afterwards, the program obtains the pixel of the final product, and it then opens a new file with the **.ppm extension** and writes those **pixels(RGB)** into the file. Correspondingly, the final product would be an image (the file created early on) and the image is in **PPM format**. And for **the graphical interface** it will then assemble simultaneously with the 4 characters 'l' 'f' 'e' 'j' in a loop to show each process of the image construction. The illustration of the image is then pops out as a window of the image size namely **"WIN"**.

3 Clarification of choices made

First and foremost, *data structure* is the essential component to begin with. The data types are describe below:

- **position**: made of x-axis and y-axis , to be more precise, horizontal and vertical respectively just like the cartesian coordinate which grants us the capability to keep track of marked and current position of the cursor.
- **direction**: consists of 4 cardinal direction including North, East, South, West as we have to move the cursor around from one direction to another. For instance, when plotting a line.
- **color and opacity**: three colors and a opacity are taken into account. Due to the fact that a data type *unsigned char* is ranged from 0 to 256 - 1 which is exactly from 0 to 255; hence, it ensures that it will not exceed 255 as it will wrap around to 0 if the value is bigger than 255 and the other way around if it happens to be a negative number.
- **pixel**: comprises of color and the opacity above which we can easily get access to the RGB and the opacity of a pixel.
- **layer**: contains numerous of pixels which together become one layer and by assigning it as a from of matrix or 2D array, it allows us to monitor the pixel of each coordinate. To illustrate, pixel$[x_1][y_1]$, where x_1 and y_1 can be any position within the layer.
- **machine**: holds a bunch of data structures above and 3 integers which allows us to keep track of the number of colors, opacities, and layers of the state machine. But, in order to possess of a bucket, assigning it as an array[MAX_VALUE] is one of the other ways to store a stack of colors, doses of opacity, and layers. *MAX_COLORS* and *MAX_OPACITIES* are very large in virtue of handling the overflow of the buckets as we do not really know the maximum value of colors and opacities to be added. And as for the *MAX_LAYERS*, since we know that there are only 10 layers in total, the number is not too really big. But, in the **machine**, the stack of layers were set to *MAX_LAYERS + 1* seeing that when printing the number of layers after adding more than 10 it starts all over again from 0, so here *MAX_LAYERS + 1* hands us the *number_of_layers* that remain the same after 10 layers were added; thus, it is the solution for visualization purposes and it will not hurt just to increase the size of array by 2 because the index starts from 0.

Beside the data structure, visualization of the code is another paramount point to be considered. I divided the code into 3 identical parts such as:

- `define`: the way data structures were built and the definition of each *MAX VALUE*.
- `function`: the implementation of each function in .c file and description of each function as well in .h file.
- `prog(main)`: the procedure of program from the beginning until the final product (image.ppm) is produced.

At last, I chose to assign some important global variables:

- **width, height**: initialized in order to store the width and height of the image from the input file (.ipi). Even though, the image is squared (width = height), by assigning width and height makes it seems more visualized; plus, more adaptable in case the difference of width and height occurs.
- ***mac**: a pointer of machine being allocated for the size of machine is extremely important here in the program in terms of function and memory used.

- **old and new**: variables pixel used in *fill* function.
- ***l and *c0**: layers that are used in most of the crucial functions such as *fill*, *fusion*, *cutting*, and so on. The pointer ***l** points to the address of the topmost layer of stack of layers in the machine, and for ***c0**, it points to the one below the top. They are assigned in the beginning of the while loop as being initialized outside the loop will be run into errors.
- **c and buf[256]**: variable **c** is to store a character from the input file using *getc(FILE *stream)* or *getchar(void)* depends on the way we read the file as to response to bonus questions (description in prog.c). And for buf[256], since we work with 0s and 1s, the program simply practice this. A character typically has 1 byte, 8 bits, and 256 potential values.

4 The arisen of technical issues and their fixes

It is an undeniable fact that implement such program will inevitably face tons of problem, but some of the most significant bugs are:

- **get_current_pixel** : when calculating the average of each color I forgot to initialize the condition to check when there is no color in the bucket, precisely there is nothing to find in the place due to the fact that there is no color; hence, *floating point exception* occurs. Solution to this is to set condition to automatically assign average value to 0 if we happen to have 0 color in the bucket.
- **add_layer** : as describe in the command in the code at the **add_layer** function, we normally should add the number of layer first before assigning value (pixel and opacity) into it. But, stack of layer is starting from 0. Solution to this is by assigning value first then increment the layer by 1 afterward.
- **keep track number of layers**: logically as the array starts from 0 then it does make sense if we assign the max stack of the layer to 9 meaning 0-9, hence 10 in total. By doing this it works fine with the image but when I tried to print the number of layers, it went from 1 to 9 then reset from 0 $1 \ldots 9$ again repeatedly whenever it reaches the maximum layer; hence, to fix this I assigned max layer to 10 and in the initial condition of the state machine I added 1 to max layer.
- **move cursor**: naturally, the north direction is the upper y-axis, so I did y++ when it moves one step. However, once I came upon it and with the aid of explanations on the internet, I discovered that the visual screen is cartesian but reversed such that the top is negative. Thus, y– is the right one, and it is not the case for x-axis, it stays the way it really was.
- **plotting_lines**: I first used while loop to repeat d times by saying while(d $>$ 0) and decrement d by 1 (d–), so each time of the iteration the value of d will always change and it affected the coordinate x/d and y/d in the loop. After the problem was spotted, I then replaced it by for loop instead.
- **fill_loop**: As stated in the project the usage of recursive function will lead to stack overflow, so after spending several days on that function by seeking senior's advice, *memorization_method* is one of the solution to transform the recursion to iteration. But still, I can only display a 300x300 image (ocaml.ipi, flowerbed.ipi,...) not the large one. Furthermore, after I tried to implement using stack to handle the stack overflow and with the inspiration from *"Lode's Computer Graphics Tutorial"*, I managed to solve the issue by using the 4 way method of flood fill with stack.

- **fusion and cutting_layers**: I first placed the argument layer c0 and layer c1 respectively and store the calculation obtained to layer c0, but unfortunately these two functions did not work correctly; hence, I made an inversion in the argument input from *(c0, c1, w, h)* to *(c1, c0, w, h)* and stored the calculation obtained in the layer c1 instead. Beside the inversion, the parentheses were wrap in the wrong as well. To clarify, originally it was *(c1 * (c0 / 255))*, then I changed it to *((c1 * c0) / 255)*.

- **overflow of bucket color**: As the bucket in the state machine is a static array; therefore, it has a fixed size. Initially, I initialized its size to 256. As a result, it runs to segmentation fault for large images such as virus.ipi and lindenmayer.ipi. Afterward, I assigned every bucket to the biggest possible (description in the code), and it works just fine after that.

- **SDL**: after watching and grasping from a bunch of tutorials, and with the simplification from senior, I managed to deliver the graphical user interface on how the image is built. But, at first, the way on how it constructs was in a excessively slow pace. It took a fourfold increase of time comparing to the solution found. So, the key was defining *MAX_WIDTH* and *MAX_HEIGHT* of the image to a relatively small number than before, as previously, both of the max value both were 10000, then to be more optimal was to change it to 1000. (in define.h)

5 Limitations

The flaws or shortcomings of this program are:

- **The utilization of pipe symnbol ("|")**: the program is not able to perform the below execution on a one line command.

  $$./prog < test.ipi \mid display$$

  Hence, the program has to prompt the user to execute twice, first the execution of the program and display on its own afterwards.

  Usage:

  $$./prog < test.ipi$$

  OR

  $$./prog\ \ test.ipi$$

  OR

  $$./prog\ \ test.ipi\ \ image\_name$$

  Then

  $$display\ \ image\_name.ppm$$

- **Run-time**: the program takes up to **1 minutes** (case: **without graphical interface**) for the complex image such as *virus.ipi* and *lindenmayer.ipi*. Whereas, it takes up to **18 minutes** (case: **with graphical interface**) for the images mentioned above. Hence, due to the complexity of the image, the user must be patient during the execution. On top of that, as the consequence of the variety of images (the immense one), the run-time could be pretty much longer than *virus.ipi*'s run-time.

- **Optimization**: the program uses static array to assign the fixed size of the bucket of colors, doses of opacity, and the stack of layer; consequently, it does not proceed to be a superbly optimal program. It might run into certain issues comparatively **segmentation fault**.

- **User-friendly**: users have to deeply understand how the program works, each role of the character assigned, certainly, the tasks it performs. It would not be much of a burden in terms of creating a simple image, for example, a circle or a rectangle with color at the background and so on. But for much more complex pictures, it requires a strong **critical thinking skill** and **imagination** to produce a file contains letters which then eventually the program will produce the image as the users want.
