PRIM - Projet C

TAN Victor - 1A - ENSIIE

3 Janvier 2023

Table des matières

1 Introduction

Dans le cadre de l'UE intitulé **PRIM11**, un projet en langage de programmation C a été demandé. Celui-ci doit interpréter un fichier d'entrée sous format *.ipi* contenant un nombre (côté de l'image en pixel) ainsi qu'une série de caractères et doit retourner une image sous format *.ppm*.

Toutes les fonctions utilisées ont été codées ou alors vues en cours, à l'exception de **memset** de la bibliothèque standard, présentée dans l'énoncé du sujet.

À l'exécution, le comportement du programme variera selon le nombre d'arguments :

Nombre d'arguments	Entrée	Sortie	Remarque
0	stdin	stdout	-
1	1^{er} argument	stdout	-
2	1^{er} argument	$2^{ème}$ argument	-
3 ou plus	1^{er} argument	$2^{ème}$ argument	Ignorance des autres arguments

TABLE 1 – Comportement du programme selon le nombre d'arguments.

Une visionneuse d'image telle que Image Magick pourra être utilisée pour ouvrir le fichier *.ppm*. Par exemple, le fichier **clip.ipi** donné en exemple est composé de ...

```
10
rfsibfsitfovvhvvpvvvvvvlpavvvvvvlpavvvvvvlpavvvvvvlavavfiovhvpvvlpavvlp
avvlpavvlavavfje
```

... et correspond à l'image :

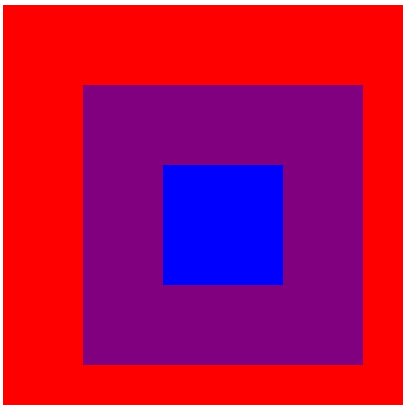

FIGURE 1 – Image obtenue avec le programme pour **clip.ipi** .

2 Composition de l'archive

Dans l'archive tar gzippée **victor_tan.tgz**, vous trouverez :

— Le fichier **rapport.pdf** ci-présent ;
— calques.c, calques.h ;
— couleur.c, couleur.h ;
— curseur.c, curseur.h ;
— etat.c, etat.h ;
— lecteur.c, lecteur.h ;
— outils.c, outils.h ;
— seaux.c, seaux.h ;
— main.c ;
— **Makefile** ;
— **README.md** ;
— L'exécutable **prog** préalablement compilé avec le Makefile.

Il serait donc possible d'exécuter le Makefile avec la commande 'make'.

2.1 Diagramme des dépendances

Nous pouvons représenter les dépendances entre chaque header .h et le main.c.

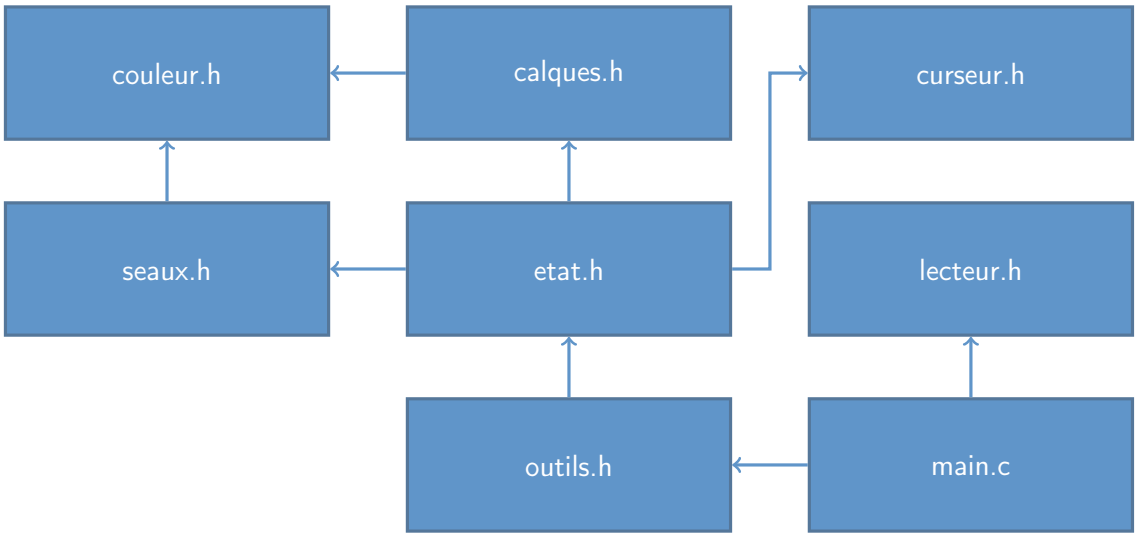

FIGURE 2 – Diagramme des dépendances du projet.

2.2 Makefile

Afin de voir toutes les dépendances entre les fichiers, les patterns **%.o**, **%.c**, ou encore **$@** et **$<** n'ont pas été utilisés, mais auraient pu l'être.

```
prog: couleur.o calques.o seaux.o curseur.o etat.o outils.o lecteur.o main.o
	gcc -Wall -Wextra -g couleur.o calques.o seaux.o curseur.o etat.o outils.o lecteur.o main.o -o prog

couleur.o: couleur.c couleur.h
	gcc -Wall -Wextra -c couleur.c

calques.o: calques.c calques.h
	gcc -Wall -Wextra -c calques.c

seaux.o: seaux.c seaux.h
	gcc -Wall -Wextra -c seaux.c

curseur.o: curseur.c curseur.h
	gcc -Wall -Wextra -c curseur.c

etat.o: etat.c etat.h
	gcc -Wall -Wextra -c etat.c

outils.o: outils.c outils.h
	gcc -Wall -Wextra -c outils.c

lecteur.o: lecteur.c lecteur.h
	gcc -Wall -Wextra -c lecteur.c

main.o: main.c lecteur.h outils.h
	gcc -Wall -Wextra -c main.c
```

FIGURE 3 – Fichier Makefile présent dans l'archive.

3 Choix envisagés

Pour la réalisation de ce projet, plusieurs implémentations différentes étaient possibles.

3.1 Construction des structures de données

Tout d'abord, une composante est un entier non signé sur 1 octet. Il était donc possible de les définir avec des **char** ou bien des **int8_t**, mais ces derniers n'ont pas été vus en cours.

Les couleurs sont composés de 3 composantes, il était donc possible d'utiliser des **char[3]** ou des **struct couleur** dont les membres auraient été des composantes. En ayant constaté que l'écriture d'un fichier PPM était possible à partir de **struct couleur**, et car la première méthode avait déjà été vue en cours et en TP, j'ai pris l'initiative de découvrir comment marche les **struct couleur** pour l'implémentation des couleurs.

Les calques sont des tableaux statiques modifiables (une fois crées, ils ne vont pas s'agrandir ou se réduire), initialisés avec des malloc, de la taille donné par le fichier d'entrée. Dans le programme, l'énoncé précisait que l'initialisation se faisait à RGBA(0, 0, 0, 0) ; il était donc possible d'utiliser des calloc plutôt, qui initialisent automatiquement à 0.

La pile de calques est une pile du type **FILO** (First In Last Out) étant donné que toutes les actions sont effectuées sur le calque le plus récent (celui au-dessus de la pile). De plus, les calques sont tous de même taille, il est donc possible de conserver la variable "taille d'un côté du calque" dans la pile plutôt que dans chaque calque.

L'énoncé nous indiquait que la taille maximale de cette pile est de 10 calques ; il est donc possible d'utiliser la syntaxe suivante :

```c
#define MAX_pile 10
```

```c
struct _pile {
    unsigned int taille; // Taille actuelle de la pile
    unsigned int cote; // Taille d'un côté de l'image en pixel
    calque calque[MAX_pile];
};
```

Une implémentation alternative est d'utiliser un pointeur vers un calque :

```c
struct _pile {
    unsigned int taille; // Taille actuelle de la pile
    unsigned int cote; // Taille d'un côté de l'image en pixel
    calque *calque;
};
```

Mais cette méthode implique l'utilisation de malloc supplémentaires, et de la fonction realloc lors de l'ajout d'un calque.

Les listes chaînées ne semblent pas être une très bonne représentation de la pile de calques ici, car nous savons qu'une pile de plus de 10 calques est impossible, et qu'il existe donc d'autres méthodes plus efficaces. Cependant, il est possible de représenter les seaux de couleurs et d'opacité par des listes chaînées, car nous devons ajouter des couleurs ou des opacités sans compter le nombre déjà présent, et car leurs tailles ne sont pas connues à l'avance.

3.2 Affichages des variables

Dans le but de débugger, plusieurs fonctions d'affichage ont été implémentées :

```c
void print_etat(etat e); // Dans etat.h
void print_pile(pile p); // Dans calques.h
void print_seau_couleur(seau_couleur c); // Dans seaux.h
void print_seau_opacite(seau_opacite o); // Dans seaux.h
```

Il aurait donc été possible de coder un mode **verbose** où le programme affichera à chaque étape le contenu de l'état de la machine, ce qui n'a pas été fait ici.

3.3 Lecture des caractères

Le contenu d'un fichier *.ipi* étant standardisé, il y a plusieurs méthodes possibles pour lire les caractères dans une **entree** (stdin ou un fichier *.ipi*).

Une lecture caractère par caractère avec

```c
int c = fgetc(entree);
while (c != EOF) {
    /*Faire la modification de l'état de la machine*/
    c = fgetc(entree.ipi);
}
```

Il est aussi possible de mettre en mémoire un grand nombre de caractères (par exemple, 512) et de les lire, avec :

```c
char buf[512] = {0}; // Ou bien utiliser memset
char *c = fgets(buf, 512, entree);
while (c != NULL) {
    /*Faire la modification de l'état de la machine*/
    memset(buf, 0, 512);
    c = fgets(buf, 512, entree);
}
```

Ici, memset sert à réinitialiser les caractères déjà présents dans le buffer. Cela permet d'avoir (c == NULL) à la fin de la lecture du fichier.

Cependant, en utilisant la commande **time ./prog ....** sur le terminal, nous constatons que pour les petits fichiers (de l'ordre du kilooctet **ko**), la méthode avec fgetc est plus rapide ; et qu'avec des fichiers de l'ordre de la centaine ou millier de **ko**, les deux méthodes sont relativement équivalentes, probablement à cause du memset ici.

3.4 Moyenne des composantes sur 1 octet

Les composantes de couleurs et d'opacité étant codées sur 1 octet, pour faire leur moyenne, il faut passer par un entier int (jusqu'à 8 octets) pour faire la somme, avant de diviser.

3.5 Rôles de c0 et c1 dans 'Fusion' et 'Découpe' des calques

L'énoncé était vague quant aux rôles des calques **c0** et **c1** pour ces deux fonctions. Nous pouvons nous demander quel calque correspond au calque situé en haut de la pile ; car les calques des fonctions ont des rôles non-symétriques.

En utilisant un logiciel d'édition d'images comme **Photoshop** ou **GIMP**, l'expérience nous montre que c'est l'opacité du calque le plus haut dans la pile qui importe. Ainsi, **c0** correspond bien au calque le plus haut dans la pile. Il suffira alors de libérer **c0**.

4 Problèmes techniques et solutions

4.1 Coordonnées cartésiennes et matricielles

Il faut faire la distinction entre les coordonnées cartésiennes et matricielles pour certaines fonctions : Par exemple, le point (0, 1) a différente position selon les coordonnées choisies.

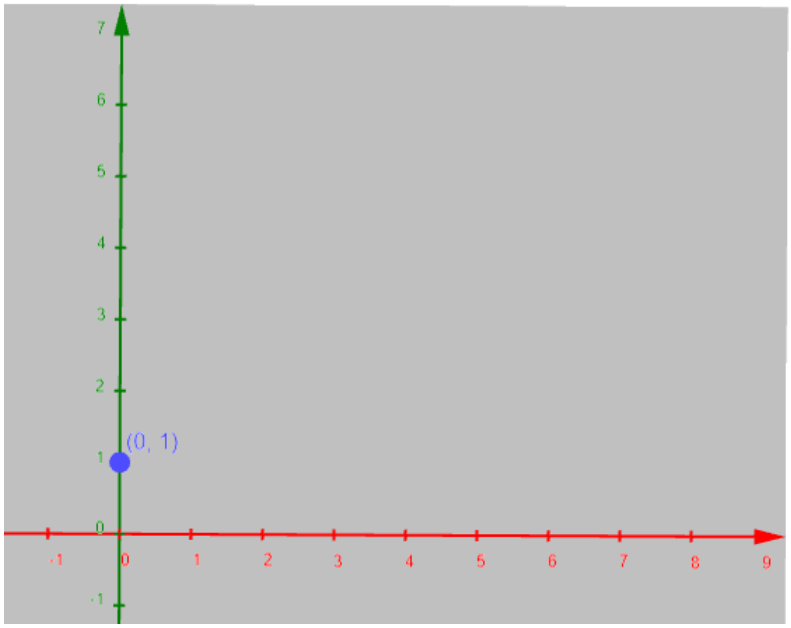

FIGURE 4 – Point (0, 1) dans un repère cartésien

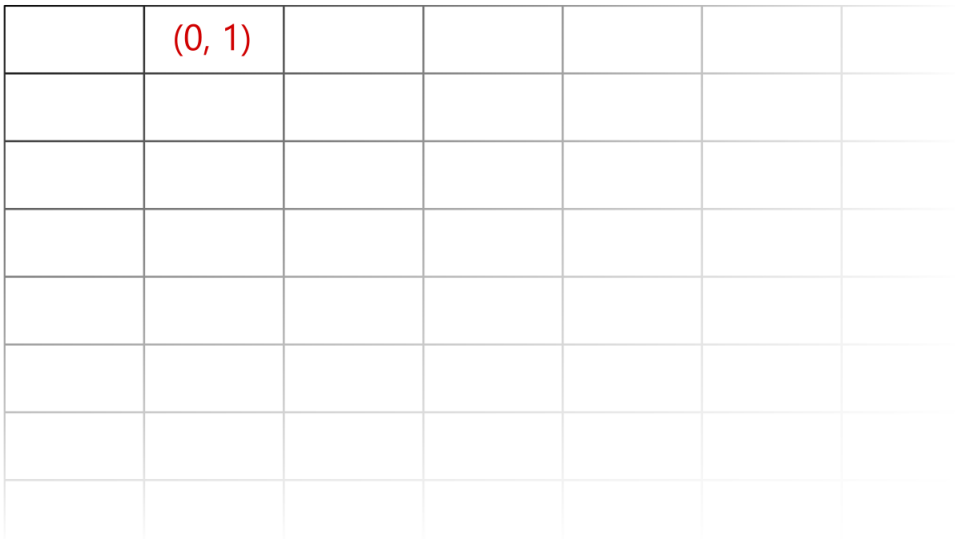

FIGURE 5 – Point (0, 1) dans un repère matriciel

Ainsi, nous prendrons un repère cartésien où l'axe des ordonnées croît quand nous avançons vers le sud. En particulier pour la fonction de remplissage, nous avons l'équivalence entre calque[x,y] de l'énoncé (c'est-à-dire la position (x, y) du calque dans le repère choisi) et calque[y][x] dans la fonction de remplissage (c'est-à-dire la position (y, x) du calque dans le repère matriciel).

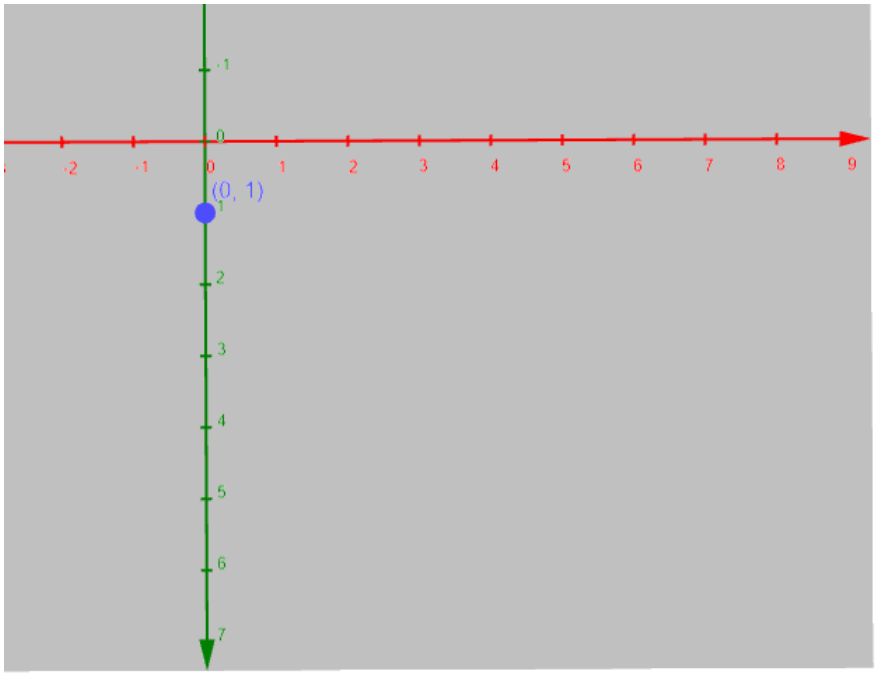

FIGURE 6 – Point (0, 1) dans un repère choisi, centré en haut à gauche de l'image

4.2 Fonction de remplissage

La fonction de remplissage par récurrence est simple à implémenter, mais reste **limitée** : Si le calque est trop grand (grossièrement supérieur à 200 pixels de côté), alors le programme rencontrera une erreur de segmentation Stack Overflow, c'est-à-dire que la pile d'appels de fonction est dépassée.

L'implémentation réalisée ici consiste à créer une **liste doublement chaînée** qu'il reste à traiter, et une **matrice de booléen** qui indique si la position a été traitée ou non. Ceci permet d'ajouter des positions tant qu'il reste de la mémoire. De plus, traiter une position (c'est-à-dire modifier sa couleur) en même temps que l'ajout de ses voisins devient possible ; ce qui repousse fortement le moment où la mémoire est saturée.

Nous pourrions montrer que le nombre d'éléments ajoutés en moyenne à chaque passage dans la boucle est compris entre $]1\,;\,4[$ (plutôt vers 2 - 3). Il serait alors possible de déterminer la taille maximale de calque supportée par cette implémentation.

Nous constatons que pour ajouter des voisins à la liste chaînée ou retirer l'élément le plus ancien, dans le cas d'une simple liste chaînée (avec seulement un membre vers le maillon suivant), il faudrait parcourir toute la liste chaînée simple à un moment. Pour éviter cela, une liste doublement chaînée (avec à la fois un membre vers le maillon suivant, mais également précédent) permet d'éviter de parcourir toute la chaîne.

La matrice évite la répétition de position dans la liste chaînée.

4.3 Optimisation

Nous remarquons que de simples problèmes d'optimisation peuvent affecter l'exécution du programme. Par exemple, lors d'un tracé d'une ligne entre les positions marquées et actuelles, si nous décidons de calculer le pixel courant (moyennes des seaux de couleur et d'opacité) dans la boucle du parcours, la production de l'image **virus.ipi** passe de 0.170 s à 2 min, à cause de la répétition de l'instruction du tracé *l*.

4.4 Affichage dynamique

L'affichage dynamique de l'image n'a pas été réalisé. Une idée consiterait à créer des fichiers *.ppm* à chaque modification majeure des calques (ou à calculer la taille totale du fichier d'entrée *.ipi* et créer des fichiers tous les x% de lecture) ; mais cela ne semble pas être l'implémentation attendue.

Une autre méthode serait d'insérer des temps d'attente **sleep** et d'utiliser l'option native de display : **display -update 1** *le_fichier_en_sortie.ppm*&, où '1' correspond au rafraîchissement du fichier toutes les secondes (valeur minimale possible) ; si nous modifions le fichier *.ppm* à chaque modification des calques.

5 Limites du programme

Il aurait été possible d'améliorer le programme :

- Utiliser des fork : Nous remarquons que la fonction **remplissage** n'utilise qu'un seul coeur de la machine. L'utilisation de plusieurs processus avec fork ou thread permettrait de gagner du temps pour le remplissage de grande surface.

- Au lieu d'imposer un buffer de lecture de 512 caractères, créer un buffer selon la taille du fichier et les ressources disponibles.

- Ecrire l'entièreté du code (notamment les noms des fonctions) en anglais pour éviter les incompatibilités avec les accents.

- Rajouter des options (–help, –verbose, ...) et un manuel **man**

- La taille maximale de l'image, donnée par la première ligne du fichier d'entrée, est limitée par la nature d'un **integer** donc sur 4 octets (0 à 2^{31} - 1). Même si peu probable, utiliser un **long integer** serait plus sûr.

- La nature de l'image *.ppm* rend impossible la possibilité d'avoir des pixels RGBA, c'est-à-dire avec potentiellement une opacité différente de 255.