Karim GOELLER

Rapport du projet de Programmation Impérative (IPI)

Interpréteur Graphique

08/01/2023

Table des matières

1 Introduction **2**

2 Construction de la structure **2**

2.1 Les données qui caractérisent l'état de la machine
2.2 L'implémentation d'une structure pour l'état de la machine
2.3 Les fonctions de dessins
2.4 lectures des différentes instructions de l'utilisateur
2.5 création de l'image finale

3 Fonctionnement Globale **5**

4 Amélioration **5**

1 Introduction

L'objectif de ce programme est d'implémenter un interpréteur graphique pour un petit langage. Dans un premier temps, l'utilisateur devra entrer un entier positif, qui sera la taille de l'image .ppm créée à la fin du programme, puis une suite de caractères qui dira comment l'image doit être créée.

Notre programme fonctionnera sur la base d'une machine à état. Dans cette machine à état, on retrouvera une position, une pile de calque, une position marquée, une direction et un sceau de couleur et d'opacité. Les instructions que l'on entrera ne feront pas exactement la même chose en fonction de l'état actuel de la machine (par exemple, si la direction est nord ou sud on n'avancera pas dans le même sens). Comme instruction, on peut trouver par exemple : ajouter un calque dans la pile, tracer une ligne entre la position et la position marquer, remplir une zone d'une couleur créée à partir des sceaux de couleur et d'opacité.

Une fois toutes nos instructions rentrées, on prend le calque situé tout en haut de la pile et on le transforme en une image au format .ppm.

2 Construction de la structure

2.1 Les données qui caractérisent l'état de la machine

Les opérations seront à effectuer sur le calque au sommet de la pile, un calque est un tableau 2D de pixel. Les pixels sont composés d'une couleur (un triplet de unsigned char nommé rgb) et d'une opacité (un entier). Pour implémenter cela, j'ai donc fait la structure suivante :

```
typedef unsigned char rgb[3];
struct pixel{
    rgb color;
    unsigned int opac;
};
typedef struct pixel pixel;
```

Un calque est donc un type Pixel** ayant comme longueur et largeur la taille de l'image que l'on veut produire. Ainsi, j'ai fait cette structure pour les piles de calque :

```
struct pill {
    int top;
    pixel **content[STACK_SIZE];
};
typedef struct pill pill;
```

J'ai opté pour une pile statique, car on connaît le nombre maximal de calques dans notre pile (ici 10) j'ai aussi implémenté les fonctions de base pour les piles, à savoir : pop, push, init. Ainsi qu'une fonction pour créer un calque avec uniquement des 0 et aussi une fonction pour supprimer toute la mémoire allouée.
Une direction est un char qui vaudra 'e' pour l'est, 'n' pour le nord etc. On peut le modifier grâce à deux fonctions qui le font tourner selon le sens horaire et antihoraire.
Pour se repérer sur les calques, on a besoin de deux doublets d'entier positif (position et position marqué) qui seront codés de cette manière :

```
int position[2];   //[0] ligne [1] les colonnes
int mark_position[2];
```

On dispose de deux fonctions pour les modifier : la fonction qui permet d'avancer qui va incrémenter ou décrémenter de 1 soit la ligne ou la colonne en fonction de la direction de l'état de la machine. Puis une autre fonction qui change la position marquée par la position actuelle.
Pour les sceaux de couleur et d'opacité. J'ai fait une structure qui contient un tableau (de rgb pour la couleur ou de int pour l'opacité) dynamique ainsi que la longueur de celle-ci (-1 si le tableau est vide). Cela donne la structure suivante :

```
struct color_bucket{
    rgb *bucket; //le tableau avec les différente couleur (rgb)
    int len;     //combien de couleur il y a dans le scean
};
typedef struct color_bucket color_bucket;

struct opac_bucket{  //idem que color bucket mais avec des int pour l'opacité
    int *bucket;
    int len;
};
typedef struct opac_bucket opac_bucket;
```

J'ai utilisé cette structure, car c'est la plus simple, et ce sceau va permettre de calculer des couleurs plus complexes, mais pour faire tout le spectre on n'a pas besoin de beaucoup de couleur et d'opacité dans les différents sceaux donc, même si la complexité pour ajouter un nouvel élément est élevée cela ne pose pas de problème en pratique car il n'y aura jamais plus d'une centaine d'éléments dans les sceaux. Pour les modifier, on a une fonction d'ajout de couleur, une fonction d'opacité et une fonction qui vide les deux sceaux.

2.2 L'implémentation d'une structure pour l'état de la machine

Afin de faciliter la conception des différentes fonctions de dessin, j'ai regroupé toutes ces données dans une structure afin que chaque fonction ne prenne qu'un ou deux arguments maximum (Il est aussi possible que l'on prenne la taille de l'image comme argument).

```
struct machine_state{  //struct où il y a toute les données de l'état de la machine
    int position[2];   //[0] ligne [1] les colonnes
    int mark_position[2];
    char direction;
    color_bucket color_b;
    opac_bucket opac_b;
    pill stack_layer;
};
```

Une fois que l'utilisateur a rentré la taille de l'image, on initialise l'état de la machine avec comme position,position marquée {0,0}, la direction par Est ('e'), les sceaux avec comme longueur -1 et aucun tableau dans bucket et avec un seul calque dans la pile contenant uniquement des 0.
J'ai fait une fonction qui sert à initialiser l'état de la machine au début du programme et à supprimer toute la mémoire allouée à la fin du programme.
Au début, j'avais aussi pensé à mettre la taille de l'image dans cette structure, toujours dans l'optique que les fonctions aient le moins d'argument mais la fonction de suppression des sceaux modifiait cette taille en un entier plus grand qu'un million ce qui causait une erreur de segmentation. Je l'ai donc retirée de cette structure.

2.3 Les fonctions de dessins

Les fonctions de dessin sont les fonctions de remplissage, de traçage de ligne, de mixage et de découpage de calques. Pour les fonctions de remplissage et traçage de ligne on aura besoin d'une couleur, cette dernière provient d'un calcuk sur les sceaux de couleur et d'opacité selon une formule écrite dans le sujet. J'ai de ce fait écrit une fonction qui calcule le pixel courant avec comme argument l'état de la machine. L'une des difficultés rencontrées dans cette fonction a été de calculer la moyenne pour chaque composante de la couleur du pixel courant car les couleurs sont codées avec des unsigned char et leur valeur maximal est 255. Je faisais les calculs sur des unsigned char et de ce fait toutes mes images étaient très sombres car je divisais 255 par de grand nombre. Alors, pour palier à ce problème, je fais tous les calculs sur des entiers puis je pose que le pixel vaut ces entiers qui seront convertis en unsigned char.

La fonction de traçage nécessite une fonction qui nous retourne la plus grande valeur absolue entre deux entiers, je l'ai codée puis j'ai suivi les instructions et elle ne m'a posé aucune difficulté.

J'ai ensuite fait la fonction de remplissage, au départ j'avais choisi de la faire de manière récursive, mais pour les grandes images cela fait une erreur de segmentation, j'ai fait une autre version avec une pile où sont stockés les positions à traiter. Comme on ne sait pas précisément combien il y aura de position à traiter (on sait seulement que cela est majoré par la taille de l'image au carré) donc j'ai fait cette pile avec une liste chaînée. Ainsi, on peut empiler des positions à l'infinie à moindre complexité temporelle.

```
typedef int pos[2];
struct cell{
   pos position;
   struct cell *next;
};
typedef struct cell* list;
```

On commence par empiler là où on se trouve et on enregistre la couleur du pixel à modifier, puis on calcule le pixel courant que l'on garde dans un pixel noté new. L'une des difficultés a été le temps d'exécution, comme j'empilais plusieurs fois les mêmes positions voisines, cela tournait en boucle. Pour éviter cela au début de la fonction je crée un tableau 1D noter test de taille size*size avec que des 0 dedans et une fois que la position {i,j} est empilée, on va mettre dans ce tableau l'élément numéro i*size+j à 1 et avant chaque empilement de voisin on regarde si le voisin est de la même couleur que old et que dans le test il a une valeur différente de 1. À la fin, on désalloue la mémoire.

Pour les deux fonctions de découpage de calque et de mixage, il suffit de dépiler les deux calques puis d'effectuer les opérations comme dit dans le sujet.

2.4 lectures des différentes instructions de l'utilisateur

Dans un premier temps on va récupérer la première ligne que l'on va stocker dans un buffer grâce à fgets, puis on va stocker la taille de l'image dans un entier size grâce à sscanf. On initialise l'état de la machine, ensuite on va récupérer un à un chaque caractère grâce à getc et tant que le fichier n'est pas finis on va lire un à un les caractères qui vont passer dans un switch pour savoir ce que l'on va faire sur l'état de la machine en fonction du caractère lu.

2.5 création de l'image finale

J'ai décidé d'écrire mes images au format P6, une fois que tous les caractères lus, je dépile le calque, et je recopie les couleurs dans une bitmap, les opacités ne sont plus utiles (la bitmap est un rgb *, et je copie chaque composante). Après cela, je réécris soit sur la sortie standard, soit dans un fichier passé en argument, d'abord le format, puis la taille, puis la valeur maximum. Ensuite, j'écris la bitmap dans le fichier grâce à fwrite tout à la suite. Une fois l'image écrite, j'utilise la bitmap.

3 Fonctionnement Globale

En fonction du nombre d'argument on ne va pas lire et écrire les caractères au même endroits, pour le main on va créer une fonction constructions qui prendra en argument un état de machine, une taille et un FILE *, et qui va lire l'entrée donner , modifier l'état de la machine et nous retourner une bitmap correspondant à l'image créée. Cette fonction sera exécutée quel que soit le nombre argument. Puis l'image sera écrits là où l'on veut, la sortis standard ou un fichier dont le nom est donné en argument

4 Amélioration

Il serait judicieux d'utiliser une liste chaînée pour les sceaux au lieu d'un tableau 1D car pour rajouter un élément cela requiert moins de temps.
Au lieu d'utiliser des tableaux 2D pour les calques, on pourrait prendre un tableau 1D de taille size*size qui ont peut-être une taille en mémoire plus petite.
Pour faire des images plus complexes, on pourrait faire une pile dynamique pour la pile de calques afin de ne pas limiter sa taille à dix grâces à une liste chainée.