Compte rendu de projet IPI 2022-2023

Léos Coutrot

January 8, 2023

Contents

1 Introduction du projet

Le projet a pour objectif d'implémenter un programme en langage C ayant pour fonction d'être un interpréteur pour un petit langage graphique.

Le programme doit lire sur l'entrée standard un fichier au format .ipi qui lui indiquera la taille de l'image à créer, puis une série d'instruction lui permettant de construire l'image. Le fichier comporte toujours sur sa première ligne un nombre indiquant la taille de l'image à créer, puis sur les autres lignes il y a des caractères qui représentent (ou non) des instructions. Ainsi le programme doit récupérer toutes les informations du fichier.

2 Problèmes et solutions trouvés durant la programmation

2.1 Les matrices

Le traitement d'image en langage C se fait à l'aide de matrices, leur usage est donc nécessaire pour le bon fonctionnement du programme. Il faut donc créer une matrice de pixels (un pixel étant une couleur et une opacité).

```
/* Creation d'une matrice*/
 matrix matrix_create(int size) {
    matrix m;
    m.size = size;
    m.cells = malloc(size * sizeof(pixel*));
    for (int i = 0; i < size; i++) {m.cells[i] = malloc(size * sizeof(pixel));}
    for (int i = 0; i < size; i++) {
        for(int j = 0; j<size; j++){
            m.cells[i][j].op  = 0;
            m.cells[i][j].c.r = 0;
            m.cells[i][j].c.g = 0;
```

```
            m.cells[i][j].c.b = 0;}}
    return m;}
```

Cependant, lors de la création de matrices, il faut allouer de la mémoire pour cette dernière et penser à la libérer à la fin pour éviter les fuites mémoires.

```
/*Liberation de l'espace alloue a la matrice*/
void matrix_free(matrix *m){
    for(int i = 0; i < m->size; i++){free(m->cells[i]);}
    free(m->cells);}
```

2.2 Seaux de couleur et d'opacité

Pour les seaux de couleur et d'opacité, je me suis orienté vers des listes chaînées notamment car elles n'ont pas de limite de taille pré-établie et on ne risque donc pas de dépassement de pile. Cependant, tous comme pour les matrices, les listes chaînées vont allouer de la mémoire à chaque fois qu'on va ajouter un élément à la liste: il faut donc la libérer pour éviter les fuites de mémoire. J'ai eu des difficultés à implémenter le programme libérant la mémoire, ma première tentative était un échec entraînant une erreur de segmentation. Après de nombreux échecs, j'ai finis par trouver la solution suivante qui me permettait de libérer la mémoire allouée tout en implémentant la fonction pour vider un seau.

```
/*Liberation de l'espace allouer a une liste chainee tout en la vidant*/
void empty_bucket(List *o){
    while(!is_empty(*o)){
        List nextnode = (*o)->next;
        free(*o);
        (*o)=nextnode;}}
```

2.3 Type engine

Afin de pouvoir regrouper toutes mes données dans un seul et même endroit, j'ai créer un type engine regroupant toutes mes données. Ainsi, tous mes fonctions "principales" auraient à priori uniquement besoin d'avoir l'engine en argument (sauf pour la fonction de remplissage, car son aspect récursif m'a fait douter de l'utilisation d'engine en argument). Ainsi, au début de mon programme, il me suffira d'utiliser une fonction d'initialisation d'engine avec pour seul argument la taille récupérée dans le fichier.

```
typedef struct engine{
    int x;                  //Abscisse du pixel courant
    int y;                  //Ordon e du pixel courant
    int xm;                 //Abscisse du pixel marqu
    int ym;                 //Ordon e du pixel marqu
    int cap;                //Direction du curseur : 0=Nord; 1=Est; 2=Sud; 3=Ouest
    List bucket_color;      //Sceau de couleurs
    List_op bucket_opacity; //Sceau d'opacit
    stack layers;           //Pile de calques
} engine;

engine initialize_engine(int size){
    engine e;
    e.cap = 1; //Definition du cap de base
    e.x = 0; e.y = 0; //On initialise la position a (0,0)
    e.xm = 0; e.ym = 0; //On initialise le pixel marque a (0,0)
    e.layers = stack_new(); //On cree la pile de calques
    stack_add_new_layer(matrix_create(size),&e.layers); //On ajoute le premier calque
    e.bucket_color = create_empty(); //On cree le sceau de couleur
    e.bucket_opacity = create_empty_op(); //On cree le sceau d'opacite
```

```
    return e;
}
```

2.4 Fonction de remplissage

Mis à part la fonction libérant la mémoire allouée, j'ai pu implémenter la plupart des fonctions sans de grandes difficultés excepté pour la fonction de remplissage. L'implémentation récursive étant donnée dans le sujet a été plutôt simple, mais dès que j'ai essayé de tester mon programme sur des images plus grandes que ducks.ipi, j'avais un dépassement de pile (comme annoncé dans le sujet). J'ai essayé comme conseillé d'introduire une pile de coordonnées déja visitées, mais j'avais gardé la forme récursive et ma tentative a échoué. Après des recherche pour une autre forme de fonction de remplissage, j'ai découvert les fonctions de remplissage dites BFS (Breath First Search). La version que j'ai trouvée utilise des queues pour l'implémentation du programme. Après avoir réadapté le programme en C, j'ai eu un problème car la queue que j'avais créée n'arrivait pas à rajouter un nouvel élément une fois qu'elle avait été vidée. J'ai donc dû trouver une autre solution. C'est alors que j'ai essayer de mélanger l'algorithme BFS avec une liste chaînée tout en m'inspirant de la précédente version récursive du programme. Après divers essais j'ai fini par réussir a implémenter le programme suivant.

```
void flood_fill(int x, int y, pixel old, pixel new,matrix *m) {
    if((old.c.r==new.c.r) && (old.c.g==new.c.g) && (old.c.b==new.c.b)
    && (old.op==new.op)){}
    else{
    List_point l = create_empty_point();
    add_point((Point){x,y},&l);
    while (!is_empty_point(l)) {
        Point p = pop_point(&l);
        int x = p.x;
        int y = p.y;
        if ( (m->cells[x][y].c.r == old.c.r) && (m->cells[x][y].c.g == old.c.g)
        && (m->cells[x][y].c.b== old.c.b) && (m->cells[x][y].op == old.op) ){
            m->cells[x][y] = new;
            if (is_valid(x + 1,y,m->size)) {add_point((Point){x + 1,y},&l);}
            if (is_valid(x - 1,y,m->size)) {add_point((Point){x - 1,y},&l);}
            if (is_valid(x,y + 1,m->size)) {add_point((Point){x,y + 1},&l);}
            if (is_valid(x,y - 1,m->size)) {add_point((Point){x,y - 1},&l);}}}}}
```

2.5 Fuites mémoires

J'ai porté une grande attention à ces dernières. Mon projet final n'en contient pas, ou tout du moins la commande valgrind –leak-check=full ./proj [nom du dossier].ipi [nom du dossier].ppm ne détecte aucune fuite mémoire. Mon programme étant assez lent pour les grandes images et puisque valgrind ralentit encore plus le processus, je n'ai pas pu tester valgrind sur des images comme best ou lindenmayer: je suis parti du principe que l'image ducks était assez complexe pour créer une fuite mémoire si mon programme en avait, et donc que valgrind le détecterait.

```
pujadas@pujadas-HP-EliteBook-850-G3:~/sauv/Projet_C/code$ valgrind ./prog ~/Projet_C/exemples_ipi_2022/ducks.ipi duckst2.ppm
==14939== Memcheck, a memory error detector
==14939== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==14939== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==14939== Command: ./prog /home/pujadas/Projet_C/exemples_ipi_2022/ducks.ipi duckst2.ppm
==14939==
==14939==
==14939== HEAP SUMMARY:
==14939==     in use at exit: 0 bytes in 0 blocks
==14939==   total heap usage: 144,032 allocs, 144,032 frees, 4,492,272 bytes allocated
==14939==
==14939== All heap blocks were freed -- no leaks are possible
==14939==
==14939== For lists of detected and suppressed errors, rerun with: -s
==14939== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 1: Résultat de la commande Valgrind ./prog ducks.ipi ducks2.ppm

2.6 Création du main dans main.c

La fonction main nécessite l'utilisation des fonctions fgets, sscanf et getc pour pouvoir récupérer les différentes données. J'ai donc voulu m'aider du conseil de fin de sujet et utiliser fgets ainsi que sscanf pour récuper la taille, mais sans faire exprès j'ai utilisé fgets dans le vide et fscanf. Après quelques recherche et quelques tests, je me suis rendu compte que le programme fonctionnait et j'ai donc décidé de rester ainsi.

```
/*Recuperation de la taille dans le fichier (ouvert avec fopen) f*/
    int size;
    fscanf(f,"%d",&size);
```

Après avoir initialisé mon type engine grâce à la taille, je dois alors récupérer tout les caractères un à un, puis d'appliquer mes programmes. J'ai donc utilisé une boucle sur getc (qui renvoie EOF à la fin du fichier) ainsi qu'un switch pour couvrir tout les cas.

```
char l=0;
while ((l = getc(f)) != EOF){
        switch(l){
            case 'n':
                add_color_black_in_the_bucket(&e.bucket_color);
                break;
            case 'r':
                add_color_red_in_the_bucket(&e.bucket_color);
                break;
            ...
            default:
                    break;

            }
    }
```

2.7 Bonus : Création d'un fichier

Ayant mal lu l'énoncé lors de ma première lecture, je pensais qu'il était necessaire d'avoir le nom du fichier en argument et de créer un fichier .ppm qui représente l'image. Cela a donc été ma première version. J'ai utilisé la fonction fopen pour créer un nouveau fichier (pour peu que le fichier n'existe pas déjà) que je pouvais ensuite remplir grâce à deux boucles et en utilisant des fprintf. Bien entendu, à chaque utilisation de fopen, on pense à fclose (sauf pour stdin et stdout).

```
if(argc==3){
fclose(f);
FILE *fp = fopen(argv[2],"ab+");
fprintf(fp,"P6\n%d %d\n255\n",size,size);
for(int i =0; i<size; i++){
    for(int j = 0; j<size; j++){
        fprintf(fp,"%c%c%c",e.layers.stack[e.layers.top].cells[i][j].c.r,...);
            }
        }
fclose(fp);}
```

3 Conclusion

Des améliorations peuvent êtres apporté à mon projet:
-une implémentation correcte de l'algorithme de remplissage BFS pour gagner en rapidité dans l'execution de mon programme.
-réécrire mes fonctions plus efficacement, comme pour mes 7 programmes qui ajoutent une couleur au sceau de couleur, un unique programme ayant la couleur à ajouter en argument pourrait me permettre

d'avoir moins de lignes de code.
-réécrire mes programmes pour qu'ils soient plus rapides car le temps d'éxécution du programme sur de grandes images est assez long (au moins 15 secondes pour certaines).

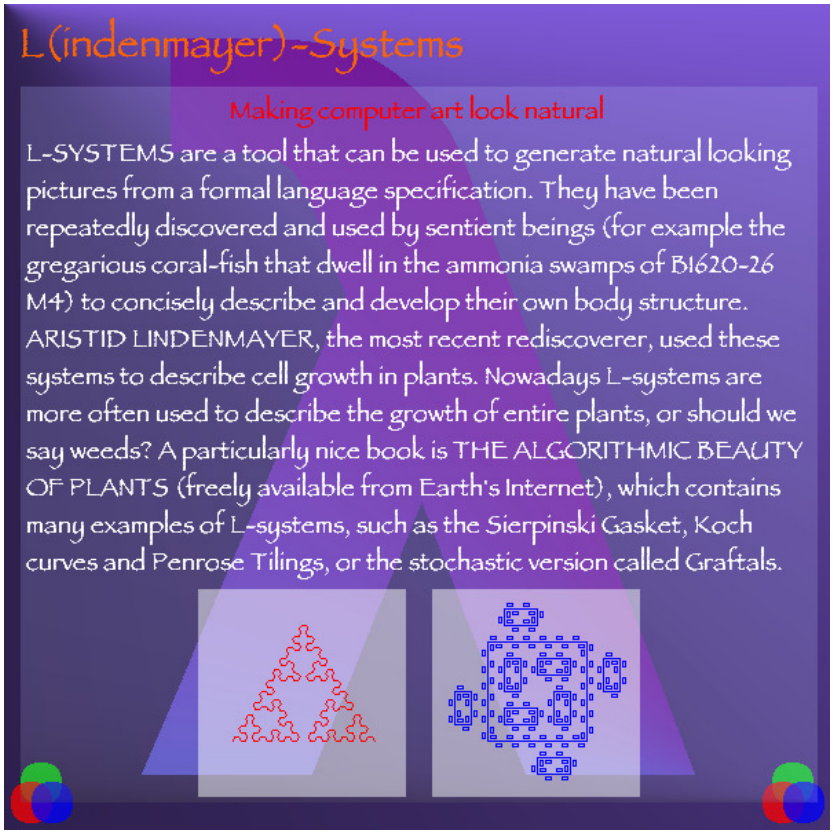

Figure 2: Exemple d'utilisation de mon programme avec la commande:
./prog lindenmayer.ipi lindenmayer.ppm