École Nationale Supérieure
d'Informatique pour l'Industrie et
l'Entreprise

PRIM
Projet 2022/2023
Rapport

Implémentation d'un interpréteur

Élève :
Colin Coërchon

Enseignant :
Guillaume Burel

Janvier 2023

Table des matières

1 Introduction

L'objectif de ce projet était d'implémenter un interpréteur pour un petit langage graphique prédéfini. Concrètement, il nous était fourni des fichiers textes comme par exemple le fichier `ok.ipi` qui, par la suite, devait être transformé par notre interpréteur par une image sous la forme d'un fichier PPM.

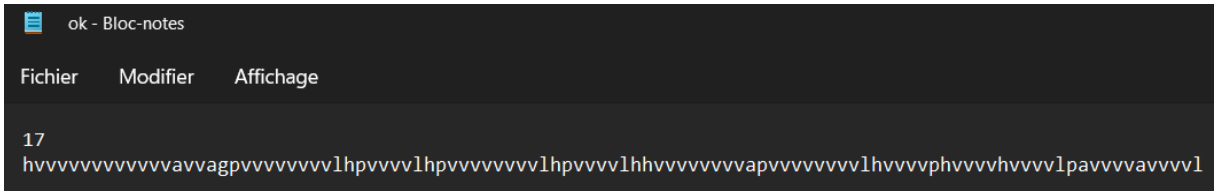

FIGURE 1 – Fichier `ok.ipi`

La première ligne correspond à la taille de l'image (carré) à produire. Le reste du fichier correspond au "langage graphique" que notre programme doit traduire en une image en format PPM.

Le projet était très guidé (ce qui n'enlève en rien de sa difficulté). Ainsi, il est expliqué que le calcul de l'image à produire se fait à l'aide d'une machine à états. Cette machine va lire caractère par caractère le code "étrange" du fichier et va, en conséquence, modifier itérativement ses états pour produire, à la fin de la lecture du fichier, un calque finale correspondant à une matrice de pixels. On peut alors construire le fameux fichier PPM attendu en sortie d'exécutable.

Un état de la machine peut alors être décrit de la manière suivante :
- Deux entiers indiquant la position courante du curseur.
- Deux entiers indiquant la dernière position marquée.
- Un élément qui indique la direction du curseur ; celle-ci peut être Nord, Est, Sud ou Ouest.
- Un multiensemble de couleurs qui représente le seau dans lequel sont mélangées des doses de couleurs pour produire la couleur courante.
- Un multiensemble d'opacités qui représente le seau dans lequel sont mélangées des doses d'opacités pour produire l'opacité courante.
- Une pile de calques de taille au plus 10, chaque calque étant une grille de pixels de la taille de l'image à produire.

Il nous est alors possible de visualiser ces images en format PPM à l'aide de différents outils comme par exemple la visionneuse d'image *Magick* (conseillé par M.Burel).

2 Implémentation des types et fonctions utiles

2.1 Découpage du programme

Pour le typage :

— Le fichier `pixel.h` comprend l'ensemble des types de première nécessité dans ce projet : les types `composante`, `opacite`, `couleur`, `pixel` et `calque` y sont définis.
— Le fichier `couleur.h` comprend la définition du type `list_c`, correspondant à l'implémentation du sceau de couleur sous forme de liste chaînée.
— Le fichier `opacite.h` comprend la définition du type `list_op`, correspondant à l'implémentation du sceau d'opacité sous forme de liste chaînée.
— Le fichier `pile.h` comprend la définition du type `pile`, correspondant à l'implémentation de la pile de calque sous forme de liste chaînée.
— Le fichier `machine.h` comprend la définition du type `pile`, correspondant à l'implémentation de la pile de calque, mais il comprend aussi la définition du type `pile_pos` qui est utile uniquement pour la version évoluée de la fonction remplissage (cf. *plus tard*).
— Et le plus important, le fichier `projet.c` comprend le `main` du projet entier. C'est ce fichier qui est compiler en dernier pour avoir notre exécutable.

2.2 Explication des choix et dépendances des fichiers

J'ai choisi de travailler avec des **listes chaînées** pour les sceaux de couleurs et d'opacités car il est expliqué dans la présentation du projet que ces sceaux peuvent stocker un nombre d'éléments qui peut grandement varier et dont l'accès à ce nombre n'est pas du tout important. Au vu du travail effectué durant les TPs durant ce premier semestre, j'ai trouvé ça plus logique de travailler avec des listes chaînées.

Pour le cas de la pile de claque de la machine, sa taille est plafonnée par 10. Il ne peut donc pas avoir plus de 10 calques dans un état de la machine. J'ai donc choisi de travailler une **structure de pile statique** pour laquelle je définie une variable globale `#define MAX` = 10.

Ainsi, au vu du typage et des différentes fonctions présentes dans les fichiers en `.h`, j'ai été forcé d' "include" le fichier `pixel.h` dans tous les fichiers en `.h` suivant : `couleur.h`, `opacite.h`, `pile.h` et `machine.h`.

Et au vu des fonctions présentent dans `machine.h`, j'ai dû "include" tous les autres fichiers en `.h` dans `machine.h` car le type `machine` prend dans sa structure quasiment tous les types définies dans les autres fichiers en `.h`.

Finalement, dans le fichier principal `projet.c`, il est seulement nécessaire de faire un "include" à `machine.h`. On peut alors correctement travaillé dans notre fichier principal avec des types abstraits bien définies dans nos fichiers `.h` précédemment.

3 Écriture du `main()` et du `Makefile`

Dans la section précédente, l'architecture des fichiers est donnée. Il en résulte alors la création de Makefile. Les dépendances sont alors écrites à côté des `.o`. Pour être vraiment sûr de n'avoir **aucun "warnings" et aucune erreur**, j'ai ajouté les drapeaux de compilations suivant `CFLAGS=-Wall -Wextra -Werror`. Mais aussi `LFLAGS=-lm`.

Pour le `main()`, il est abondamment commenté, et j'en ai profité pour rajouter la première **question BONUS**. Ainsi, il est possible de placer autant d'arguments que l'on souhaite derrière notre exécutable ! (Bien que seul le premier ou les deux premiers seront pris en compte.)

Ainsi,

— Si un premier argument est fourni, ce sera le nom d'un fichier qui sera lu en entrée à la place de l'entrée standard.
— Si un deuxième argument est fourni, ce sera le nom d'un fichier qui sera écrit en sortie à la place de la sortie standard.
— Les autres arguments seront ignorés, avec un avertissement sur la sortie d'erreur standard.

De plus, on peut résumer le main dans le cadre où 2 arguments sont donnés, puisque dans les autres cas, le seul changement est qu'on s'intéresse à l'entrée et à la sortie standard, et non aux fichiers placés aux arguments.

Ainsi, le `main(int argc, char* argv[]` effectue dans l'ordre :

— On ouvre le fichier placé en premier argument en lecture seule.
— On récupère la première ligne du fichier $f1$ dans le buffer à l'aide de la fonction `fgets`.
— On récupère l'entier écrit dans le buffer pour récupérer la taille à l'aide de `sscanf`.
— On initialise la machine (cf. `machine.h`).
— On parcourt l'ensemble des caractères de $f1$ et on modifie à chaque itération l'état de la machine `m`. Le caractère `c` est modifié à chaque itération de la boucle "while" à l'aide de la fonction `getc`. Pour cela, on fait appel à la grande fonction `modif_etat` qui fait un "switch" sur toutes les possibilités possibles en fonction du caractère `c`.
— On récupère l'image finale (qui correspond au sommet de la pile de calque).
— On ouvre le fichier placé en premier argument en écriture binaire.
— On écrit l'en-tête du fichier PPM dans le fichier $f2$.
— On écrit dans $f2$ compasante par composante de chaque pixel grâce à `fwrite`.
— On ferme les fichiers $f1$ et $f2$.

4 Résultats

En terme de résultats, cela a été mitigé. Mon programme compile très bien. La majorité des images obtenues sont bonnes mais certaines (comme par exemple degrade.ppm) ne sont vraiment pas parfaites. Pourtant des images comme best.ppm sont très bonnes, mais pas parfaites je crois...

C'est dommage, je pense que cela est dû au fait que initialement, j'ai défini mon type `composante` comme un entier non signé, et non comme un char non signé : `typedef unsigned int composante`.

FIGURE 2 – Fichier `best.ipi`