Rapport

Projet Programmation Impérative

Interpréteur pour langage graphique

ENSIIE
2022-2023

Noah Kwa Moutome

8 janvier 2023

Table des matières

1 Présentation du projet

1.1 But du projet

Le but de ce projet était de créer un programme dans le langage C ayant les fonctionnalités de base d'un outil de création d'image classique, l'image produite étant au format PPM.

Ce programme devait permettre d'utiliser les outils classiques présents dans les applications de ce genre, à savoir une fonction de remplissage d'une zone de couleur, une fonction de tracer de ligne entre deux points, la gestion de plusieurs calques (jusqu'à 10), la fusion de 2 calques, le découpage de 2 calques et ainsi que la gestion de seaux de couleur et d'opacité.

1.2 Principe de fonctionnement

Le programme avait plusieurs modes de fonctionnement, définis en fonction du nombre d'arguments qui lui était fournis. Cependant le principe général est le même pour tous ces modes :

1. Le programme lit (sur l'entrée standard *stdin* ou sur un fichier) la taille de l'image (qui sera carré) et une suite de caractères correspondant aux différentes actions effectuées par l'utilisateur.

2. A chaque lecture de ces caractères, l'état de la *machine* est mis à jour

3. Une fois la lecture finie, le programme écrit l'image obtenue ou bien dans la sortie standard *stdout* ou bien dans un fichier PPM passer en argument (seul le calque présent en haut de la machine est pris en compte).

2 Gestion du projet

Dans cette partie, nous allons voir le fonctionnement général et les différents types de variable, ainsi que les fonctions nécessaires à la réalisation de ce programme et les raisons qui ont poussé à de tels choix. Les problèmes rencontrés au cours de l'implémentation seront également mentionnés, pour une explication plus détaillée de chacune des fonctions se référer aux fichiers de code.

2.1 Les différents types et leurs implémentations

- Une variable globale *SIZE_OF_STACK* représentant la taille maximale d'une pile de calques (dans notre cas 10) ;

- Un type enumerate (car il y a un nombre fini de valeurs possibles) *direction* prenant des valeurs prédéfinies (north, east, south, west) permettant d'indiquer la direction du curseur de la machine ;

- Un type *composante* qui est simplement un entier codé sur un octet ;

- Le type *couleur* qui est un struct, composé de 3 composantes de type *composante*, qui représentent le code de la couleur en RGB. J'ai choisi d'implémenter ce type avec un struct et non un tableau de taille 3 pour être sûr de ne pas me tromper lorsque je voulais accéder à la composante particulière d'une couleur ;

- Le types *pixel* qui est un struct composé d'une *couleur* et d'une *opacité* représentée par une *composante* ;
- Le type *claque* est simplement une matrice carré de *pixel* ;
- Le type *pile_ of_ calques* est une pile de calques de taille maximum *SIZE_ OF_ STACK*. J'ai fait le choix ici d'utiliser une pile de taille maximum fixée d'avance, car on sait que l'on aura jamais plus de 10 calques. On prend certes plus de mémoire que nécessaire si l'image fait moins de 10 calques mais faire une pile dynamique aurait obliger de ré-allouer un nouveau tableau de calques en incrémentant la taille de 1 et en copiant l'ancienne pile dans la nouvelle, ce qui me semblait être une perte de temps et d'efficacité ;
- Le type *seau_ couleur* est en fait un pointeur vers une autre structure, composée d'une *couleur* et d'un *seau_ couleur* qui représente la suite du seau : c'est donc une simple liste chaînée. J'ai fait ce choix car il autorise à avoir plusieurs fois la même couleur dans le seau, et un très grand nombre de couleur ;
- Le type *seau_ opacite* reprend les mêmes principes que le type *seau_ couleur*, en remplaçant simplement la *couleur* par une *opacite* ;
- Enfin, le dernier type important : le type *machine*. J'ai choisi de créer ce type pour faciliter l'accès à l'état courant de la machine, et cela me semblait être la manière la plus intuitive de faire, cependant il aurait également été possible de créer une variable de chacun des types précédents et de les utiliser comme état courant de la machine. Ce type *machine* représente donc l'état courant de la machine, et est composé de : 2 tableaux d'entier de taille 2, représentant la position courante du curseur ainsi que la position marquée, la *direction* du curseur, le *seau_ couleur*, le *seau_ opacite*, et la *pile_ of_ calques*. L'état de la machine est donc mis à jour à chaque lecture d'une action, dont nous allons parler dans la partie suivante.

D'autres types secondaires ont également été implémentés pour permettre l'implémentation de certaines fonctions, je ne vais donc pas les détailler ici pour des raions de concisions, pour plus de détail se référer au code commenté.

2.2 Les fonctions et leur implémentation

Pareillement à la partie précédente, de nombreuses fonctions ont été implémentées pour permettre la réalisation de ce projet, j'énoncerai donc simplement les fonctions qui ne posent aucun problème particuliers et je m'attarderai plus en détails sur les fonctions un peu plus complexes.

- Couleur : une fonction **newCouleur()** qui prend en paramètre 3 composantes et retourne la couleur correspondante ;
- Pixel : un fonction **newPixel()** qui retourne un pixel noir, d'opacité 0 ;
- Calque : une fonction *newCalque()* qui retourne un calque de pixels noirs et d'opacité 0, de la taille de l'argument passé en paramètre et une fonction **freeCalque()** qui libère l'espace du calque passé par référence en argument ;
- Pile de calques : une fonction **newPile()** qui renvoie une pile de calques vide, ainsi que toutes les fonctions de bases relatives aux piles (pop, push, **popFreeCalque()** qui libère le calque enlevé de la pile) ;

- Seau de couleur (resp. opacité) : **newSeauCouleur()** (resp. **newSeauOpacite**) qui renvoie un seau de couleur (resp. opacite) vide, **isEmptySeauCouleur()** (resp. **isEmptySeauOpacite()**) qui renvoie 1 si le seau est vide, 0 sinon, une fonction **addCouleur()** (resp. **addOpacite()**) qui prend une couleur (resp. une opacité) et un seau passé par référence en argument, et ajoute cette couleur (resp. cette opacité) au seau en question, et enfin **VideSeauCouleur()** (resp. **VideSeauOpacite()**) qui vide le seau passé par référence en argument ;

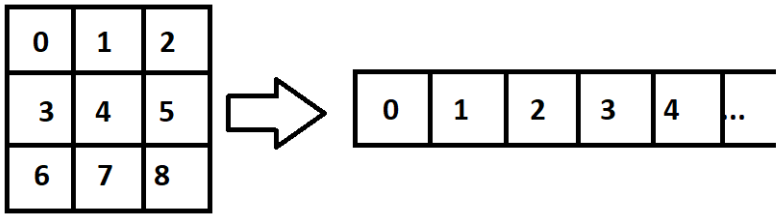
Passons maintenant aux fonctions relatives à la machine, qui sont bien plus complexes que les fonctions mentionnées précédemment. Pour la plupart de ces fonctions, j'ai décidé de passer en argument non pas la machine, mais simplement les composantes de la machine nécessaires à la fonction en question. Ceci permet d'utiliser les fonctions indépendemment de l'existence ou non d'une machine, et cela a rendu les test de ces dernières plus agréables.

1. **createMachine()** qui renvoie une machine initialisée avec une taille passée en argument ;
2. **moyOpacite()** et **moyCouleur()**, qui calculent les moyennes d'opacité et de couleur des seaux de la machine passée en argument, fonctions uniquement utilisées dans la fonction **calculPixelCourant()** qui renvoie le pixel courant de la machine passée en argument ;
3. **tracerLigne()** qui prend en argument 2 positions (ie 2 tableaux d'entier de taille 2) et un pointeur vers une machine, et va tracer une ligne de pixel identique au *pixel courant* sur le calque présent en haut de pile dans la machine ;
4. **fusionCalques()** qui prend en paramètre 2 calques et fusionne le 1er calque (qui correspond à celui du haut) sur le 2^{e} calque (qui correspond donc à celui du bas). J'ai au début eu du mal à comprendre l'ordre des calques passés en argument en lisant le sujet, sans doute car je ne suis pas du tout familier avec ce type d'outil de peinture. Cette fonction ne renvoie rien, elle ne fait que modifier un calque ;
5. **decoupageCalque()** qui prend les mêmes paramètres que la fonction précédente, et découpe le 2^{e} calque par rapport au 1er. Cette fonction ne renvoie également rien ;
6. **avanceDirection()** qui avance le curseur de 1 vers la direction courante ;
7. **rotationHoraire()** et **rotationAntiHoraire** qui, comme leur nom l'indique, effectue une rotation de la direction de la machine passée en argument dans le sens horaire ou anti-horaire ;
8. Enfin, la fonction **fill()** qui a été celle m'ayant posé le plus de problèmes. Cette fonction était à l'origine récursive et suivait le principe de remplissage par diffusion, hors sur des calques d'une taille conséquente, on avait un dépassement de pile assez important, et le programme se terminait donc sans avoir remplit la zone désirée. J'ai donc du implémenté un autre type de pile (de points) et toutes les fonctions de base associées, dans laquelle je mettais les points qu'il me restait à traiter dans la fonction *fill*. La taille de cette pile étant inconnue à l'avance, j'ai utilisé des listes chaînées. Autre problème, au vu de l'algorithme, de nombreux points étaient ajoutés plusieurs fois dans la pile, ce qui ralentissait l'execution. J'ai donc décidé d'utiliser une matrice de booléen qui m'indiquait si oui ou non une positon avait déjà été ajouté à la pile, ce qui améliore grandement l'efficacité de cette fonction.

2.3 Lecture et création de l'image

La lecture et créattextbfion de l'image se sont faites à l'aide de 3 fonctions :

- **lecture()** qui prend en paramètre une machine par référence ainsi qu'un caractère (correspondant au caractère lu) et effectue une mise de la machine en fonction de ce caractère ;
- **createBtm()** qui créer une bitmap (ie tableau 1D) de taille size*size (size étant l'entier passé en argument, correspondant à la taille des calques de la machine), contenant uniquement les *couleurs* du calque passé en argument. J'avais dans un premier temps opté pour un tableau à 2 dimensions, mais les images obtenues avaient des problèmes et je n'ai pas réussi à comprendre pourquoi. J'ai donc essayé avec un tableau 1D (l'erreur obtenue ne pouvait venir que de cette fonction à mon sens, j'ai donc envisagé chaque changement possible au sein de celle-ci) et cela a resolu le problème.



- **createPPM()** qui écrit l'entête d'un fichier PPM dans le fichier pointé par le FILE* passé en argument, et qui par la suite écrit la bitmap crée via un appel à *createBtm()* depuis le calque passé en argument. Une image au format PPM est donc produit sur le fichier ou le flux passé en argument.

Enfin la lecture des caractères passés en arguments au programme, selon les différents modes de fonctionnement de ce dernier, se fait dans la fonction *main*. On boucle sur un caractères *c* tant que ce dernier est différent de *EOF*. On fait appel à *lecture()* à chaque itération puis on crée l'image à l'aide de *createPPM()*.

3 Conclusion

Mon travail a permis d'aboutir à un programme fonctionnel, que j'ai essayé d'optimiser autant que je l'ai pu, dans la limite des idées que j'ai eu, et il parvient à passer chacun des tests proposés par le sujet. Le programme peut donc être utilisé de 3 manières différentes (se référer au manuel d'utilisation pour plus d'informations), s'execute dans un temps relativement court (de l'ordre des *4 secondes*) et produit l'image attendue par l'utilisateur.

De potentielles améliorations pourraient être implémentées pour améliorer la rapidité d'exécution (par exemple sur la fonction *fill()* que l'on pourrait modifier pour implémenter l'algorithme de remplissage par diffusion en *bouclant vers l'est et vers l'ouest*) ou encore introduire une interface graphique pour permettre à l'utilisateur de voir la construction de son image en temps réel.

4 Manuel d'utilisation

J'ai implémenté une version du programme permettant de l'utiliser de plusieurs manières. Dans un premier temps il faut évidemment effectuer la commande *make* dans un terminal, en se trouvant dans le répertoire contenant tous les fichiers du projet. L'exécutable *prog* est alors crée et plusieurs choix sont possibles :

- effectuer un **./prog < fichier.ipi**, ceci va lire le fichier *fichier.ipi* qui contient la suite d'instructions pour construire l'image et l'afficher sur la sortie standard *stdout*. Ajouter un | **display** permet d'afficher l'image à l'aide d'un logiciel. La redirection < *fichier.ipi* simule le fait que l'utilisateur rentre la liste d'instructions directement depuis le terminal ;
- effectuer un **./prog fichier.ipi**,cette fois-ci la suite d'instruction est un *paramètre* du programme, et l'image est aussi afficher sur la sortie standard. Même remarque que précédemment pour le | **display** ;
- enfin, effectuer un **./prog fichier.ipi fichier.ppm** permet d'écrire l'image correspondant aux instruction du fichier *fichier.ipi* directement dans le fichier *fichier.ppm*. Attention cependant, le fichier *fichier.ppm* doit déjà exister !

Tout autre argument supplémentaire est ignoré.