Bruno GALIBERT
Première année, 2023-2024

Rapport de Projet de PROGRAMMATION IMPÉRATIVE

IMPLÉMENTATION D'UN INTERPRÉTEUR POUR UN LANGAGE DE PROGRAMMATION DONT LES FONCTIONS SONT DES IMAGES

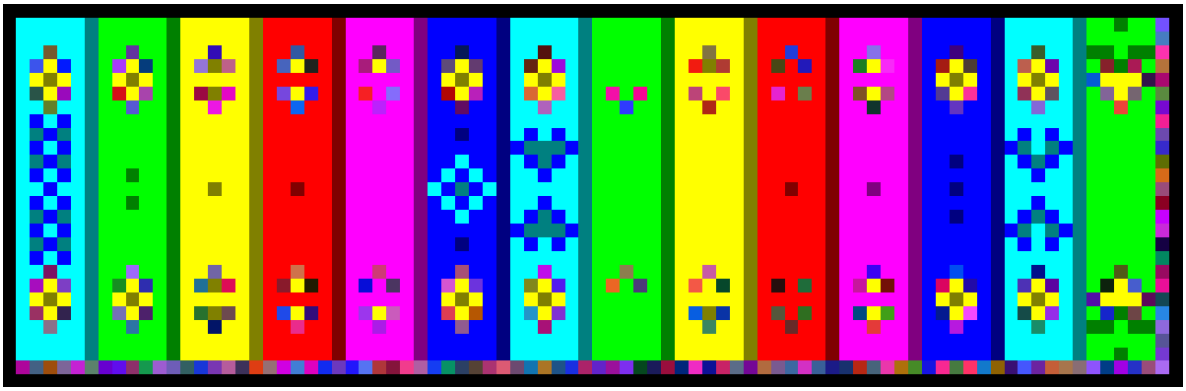

Table des matières

1 Introduction

Le but de ce projet est d'implémenter en C un interpréteur pour un langage de programmation dont les programmes sont des images : le "Cornelis".

En Cornelis (fortement inspiré du langage Piet), les programmes sont donnés sous la forme d'images en deux dimensions. L'interpreteur à implémenter doit parcourir ces images selon des règles prédéfinies, et effectuer une certaine action en cas de passage d'une couleur à une autre.

2 Élaboration du programme

2.1 Analyse des besoins

La première lecture du sujet de ce projet fut quelque peu destabilisante. Entre les blocs de couleurs, les déplacements de l'interpreteur, les actions à effectuer, les différences de couleurs... : le projet relie de nombreuses sections avec leurs problématiques propres. La première chose que j'ai faite a donc été de réfléchir à la structure globale du projet : comment le diviser en parties distinctes, ou autrement dit, quels fichiers thématiques écrire. De toute évidence, j'allais avoir besoin de gérer séparément :

- l'identification des blocs de couleurs, et les déplacements de l'interpreteur en leur sein : aller au bon pixel selon sa direction et son bord,
- la structure-même de l'interpreteur : comment implémenter ces fameux direction et bord,
- la structure de piles, puisque l'interpreteur en a une,
- les couleurs : comment définir les 18 constantes, mais surtout comment implémenter le tableau 2D de leurs différences en couleur ET en luminosité,
- les différentes actions à effectuer selon cette différence,
- bien-sûr, les étapes de l'éxécution de l'image donnée,
- sans oublier l'obtention d'une image exploitable, initiallement au format PPM.

Une fois ces différentes parties rapidement identifiées, la tâche semblait nettement plus accessible : il suffisait de les traiter une part une, en les testant au fur et à mesure.

2.2 Exploitation de l'image PPM

Pour pouvoir naviguer dans l'image, j'ai repris la manière de convertir une image PPM en un tableau de pixels que nous avions vue en cours lors du semestre. Pour épurer au maximum le fichier "main.c", j'ai consacré un fichier entier à la gestion des images : images.[h/c].

J'ai d'abord repris les structures vues pour les pixels et les images, avant d'écrire une fonction prenant en entrée le nom du fichier PPM, et qui se charge de l'ouvrir, le copier, et qui retourne un pointeur vers la mémoire allouée pour le tableau de pixels.

```c
typedef unsigned char pixel[3];

typedef struct image {
    int w;
    int h;
    pixel *mat;
} image;
```

FIGURE 1 – Définitions du pixel et de l'image.

Dans un premier temps, j'avais opté pour des tableaux en véritable deux dimensions (dont on accède à la case (i, j) par tab[i][j]), mais pour une raison que je n'ai pas réussi à identifier les pixels obtenus ne correspondaient pas à ceux de l'image. Changer pour des tableaux à une seule dimension (accès par tab[i * largeur + j]) a réglé le problème.

La seule différence notable entre la version vue en cours et celle du programme réside dans la gestion des éventuelles lignes de commentaires, puisque deux des qunze images mises à notre disposition en guise d'exemples en comprenaient.

```c
do {
    fgets(buffer, sizeof(buffer), file);
} while (buffer[0] == '#');
```

FIGURE 2 – Boucle pour ignorer les éventuelles lignes de commentaires.

Puisque le nombre de lignes de commentaire, à compter qu'il y en ait, est à priori inconnu, il faut utiliser une boucle while, et le choix de la forme do-while réside dans le fait qu'on souhaite dans tous les cas lire au moins une ligne, quitte à l'exploiter si ce n'est pas un commentaire.

Après quelques tests, en comparant les composantes RVB de plusieurs pixels à l'image ouverte dans un visionneur d'images, le tableau de pixels obtenu était bien conforme à l'image donnée.

2.3 Implémentation de l'interpreteur

Implémenter l'interpreteur n'était sûrement pas le plus urgent à ce stade, mais je l'ai fait très tôt car je savais que ce serait relativement facile, la structure de pile m'étant déjà bien familière.

2.3.1 Coordonnées, direction et bord

L'interpreteur décrit dans le sujet comprend plusieurs variables. J'ai rapidement décidé de lui dédier une structure, pour les stocker à la même adresse mémoire et simplifier l'appel aux futures fonctions.

Pour stocker la localisation de l'interpreteur, le choix était évident : deux entiers, 'i' pour la ligne et 'j' pour la colonne font parfaitement l'affaire.

Pour la direction et le bord en revanche, la solution n'était pas toute trouvée. Puisque la direction est amenée à être incrémentée de manière cyclique dans le sens horaire, j'ai d'abord envisagé d'en faire une liste chaînée circulaire. Mais j'étais persuadé qu'une solution plus simple existait, et j'ai finalement choisi de traiter les directions abstraitement, en associant un entier à chacune des quatre, et en les

incrémentant modulo 4. Néanmoins, je n'ai réalisé qu'en relisant l'ensemble de mes fichiers qu'utiliser 'enum' aurait simplifié les notations (Cf partie 4.4).

Pour le bord, j'ai choisi d'associer la valeur -1 à bâbord, et 1 à tribord, de sorte que dans les deux cas, changer de bord revienne simplement à multiplier l'actuel par -1.

Ne restait plus que la pile, mais pour clarifier les fichiers "interpreteur.[h/c]", je leur ai consacré des fichiers à part entière : "stack.[h/c]" (Cf 2.3.2). Voici donc la structure retenue pour l'interpreteur :

```
typedef struct interpreteur {
    int i;
    int j;
    int dir;
    int bord;
    stack *pS;
} interpreteur;
```

FIGURE 3 – Implémentation de l'interpréteur.

J'ai ensuite écrit des foncions pour déplacer l'interpreteur, incrémenter sa direction et changer son bord. Pour toutes les actions modifiant sa pile, j'ai d'emblée préféré les rassembler dans des fichiers à part "actions.[h/c]" (Cf 2.6).

2.3.2 Implémentation de la pile de l'interpreteur

Comme pour l'ouverture de fichiers PPM, j'ai repris l'implémentation des piles vue en TP lors du semestre.

Puisque la taille de la pile de l'interpreteur dépend de l'image donnée, il fallait prévoir une taille modifiable pour ne pas manquer de place, ni gacher d'espace mémoire en prévoyant une taille inutilement trop grande. Après avoir hésité entre les listes chaînées et les tableaux dynamiques, j'ai estimé que les deux solutions convenaient, et ai donc opté pour les tableaux par simple préférence personnelle.

Cet unique choix nécessaire fait, j'ai repris les fonctions classiques de manipulation de piles : initialisation, test de vacuité, push, pop, en écrivant une fonction de redimensionnement de la pile à utiliser dans push et pop.

```
typedef struct stack {
    int top;
    int size;
    int *tab;
} stack;

void stack_init (stack *s);
void stack_free (stack *s);
int stack_size (stack *s);
stack* stack_empty(void);
int stack_isEmpty (stack *s);
void stack_resize(stack *s, int new_size);
void stack_push (stack *s, int elem);
int stack_pop (stack *s);
void stack_print (stack *s);
```

FIGURE 4 – Structure de piles et signatures des fonctions de manipulation.

2.4 Définition des couleurs

Choisir comment gérer les couleurs, en particulier les différences de couleurs, représenta la première interrogation pour moi. Je me suis tout de suite dit qu'une couleur étant définie par ses composantes RVB, ce n'était rien d'autre qu'un 'pixel' défini plus tôt, et différencier les couleurs passantes des bloquantes était facile puisqu'il suffisait de calculer leur luminosité avec la formule classique rappelée.

Néanmoins, je ne savais comment implémenter les 18 couleurs codantes. Dans le sujet, elles sont présentées dans un tableau, en fonction de leur "couleur" mais aussi de leur luminosité. Puisque ces deux critères sont à regarder lors d'un passage d'une couleur codante à une autre, je me suis demandé s'il fallait physiquement implémenter un tel tableau, et calculer l'écart de deux couleurs en terme de lignes et colonnes. Mais d'un autre côté, les 6 couleurs comme les 3 luminosités forment des cycles, et ajoutés aux piles et à la direction de l'interpreteur, je me suis redemandé si je ne ferais pas mieux de définir une structure de listes chaînées une fois pour toute.

Finalement, comme pour la direction, j'ai pensé qu'il était moins gourmand en ressources de manipuler des entiers que des listes, et que je traiterais les deux cycles grâce à des modulos (respectivement 6 et 3). Ainsi, plus question d'un tableau 2D de couleurs, définir 18 constantes suffisait :

```
pixel rouge = {255, 0, 0};
pixel vert = {0, 255, 0};
pixel bleu = {0, 0, 255};
```

FIGURE 5 – Exemple : définitions des 3 premières couleurs.

Une fois les 18 constantes définies, j'ai écrit deux fonctions similaires, 'whichColor' et 'whichBright', prenant toutes deux en argument une couleur codante, et ignorant respectivement sa luminosité et sa "couleur". Par exemple :

— whichColor(rouge foncé) retourne 0 pour 'rouge',
— whichBright(rouge foncé) retourne 2 pour 'foncé'.

Je pouvais ainsi facilement obtenir la différence en couleur comme en luminosité entre deux couleurs codantes, grâce au modulo.

```
int diffColor (pixel arrivee, pixel depart){
    return (whichColor(arrivee) - whichColor(depart) + 6) % 6;
}
```

FIGURE 6 – Fonction 'diffColor'. 'diffBright est écrite sur le même principe.

J'obtenais donc les deux valeurs nécessaires et suffisantes pour déterminer l'action à effectuer par l'interpreteur. Mais plutôt que d'identifier chaque action par un couple (diffColor, diffBright), j'ai préféré leur donner un identifiant entier unique généré par ces deux valeurs. J'ai ainsi obté pour :

```
int codeAction (pixel colArriv, pixel colDepart) {
    return diffColor(colArriv, colDepart) * 10 + diffBright(colArriv, colDepart);
}
```

FIGURE 7 – Fonction générant un ID entier unique par action.

L'écriture des dites actions serait pour plus tard, mais j'avais résolu la question de leur identification.

2.5 Traitement du bloc courant

Lors d'une étape de l'exécution de l'image donnée, tout repose sur la connaissance du bloc de pixels de même couleur sur lequel l'interpreteur se situe présentement. Il faut donc réussir à le délimiter. Pour cela, j'ai bien-sûr repris l'algorithme récursif de type "remplissage" expliqué dans le sujet du projet : partant du point initial, on garde trace des pixels déjà traités dans un tableau de booléens, et traite ses quatre voisins si le pixel observé appartient au bloc. J'ai adapté l'algorithme selon les besoins du programme. En effet, les informations nécessaires pour le bon déroulement de l'étape sont :

— la taille du bloc (pour l'action "empile"),
— les coordonnées du pixel le plus du bord actuel, sur la frontière la plus loin dans la direction courante.

Malgré la taille, qui nécessite evidemment de parcourir le bloc dans son entièreté, j'ai d'abord réfléchi à partir dans la direction de la frontière recherchée. Néanmoins, je me suis rendu compte que peu importe la direction et le bord actuels, il était nécessaire de parcourir l'ensemble du bloc.

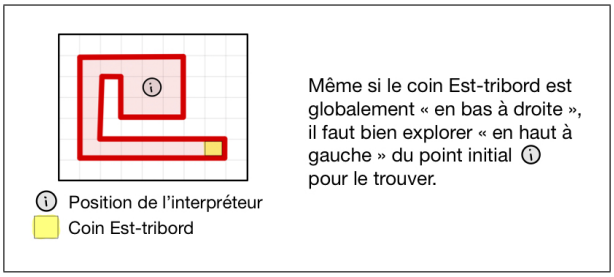

FIGURE 8 – Illustration du besoin de parcourir l'entièreté du bloc.

Ainsi, la connaissance du pixel recherché n'aide en rien sa recherche. Quitte à devoir partir dans toutes les directions, j'ai alors choisi de récupérer les 8 coins simultanément, et de remettre le choix du bon parmi les 8 à plus tard, lors de l'appel à cette fonction de recherche. De plus, j'ai d'emblée su que cette manière m'aurait permis, avec un peu plus de temps, d'optimiser le programme en évitant de reparcourir un bloc déjà connu (Cf partie 4.3).

Pour me déplacer dans l'image à la recherche des frontières du bloc, j'ai commencé par écrire quatre fonctions pour gérer la topologie de tore de l'image, et partir "de l'autre côté" en cas de dépassement :

```c
int incr_i(image *pImg, int i) {
    return (i + 1) % pImg->h;
}
```

FIGURE 9 – Fonction qui incrémente la variable de ligne.

Ensuite, après avoir établi le tableau de booléens évoqué, j'ai écrit une fonction 'traitement' parcourant le bloc et récupérant les informations recherchées, en m'inspirant de l'algorithme de recherche de max dans un tableau : on initialise un candidat avec une valeur que l'on sait plus petite que le max, et on actualise ce candidat dès que l'on rencontre une valeur plus grande. Ici, on ne cherche pas un, mais 4 "max" : les indices des lignes (respectivement colonnes) des frontières les plus au Nord et Sud (respectivement Est et Ouest) du bloc, c'est-à-dire les pixels dont le voisin dans la direction regardée n'est PAS de la même couleur (sinon, on n'est pas sur une frontière).

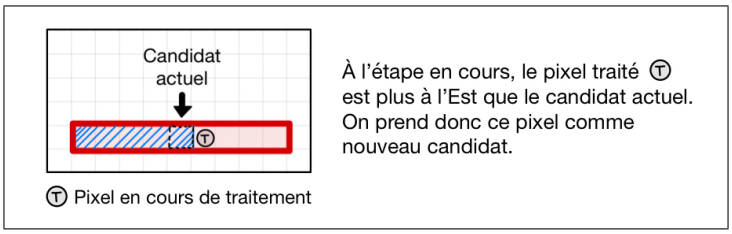

FIGURE 10 – Exemple : recherche de la frontière Est.

Une fois les quatre frontières localisées (via les quatre indices recueillis), on peut les longer afin de trouver les pixels les plus à babord et tribord de chaque. Là encore, on procède par recherche de maximum. Prenons l'exemple de la frontière Est, dont on a trouvé la colonne 'frontiereEst' :

1. on se place au "début" de la colonne, soit en : image[0][frontiereEst],
2. tant qu'on n'a pas trouvé de pixel qui convient (on discutera de ce que cela signifie juste en-dessous), on "descend" (on incrémente la ligne),
3. le premier pixel à convenir est notre pixel Est-babord. On retient sa ligne, et on initialise une variable entière 'max' à cette ligne.
4. on continue de "descendre", et dès qu'un nouveau pixel convient, on remplace la valeur précédente de 'max' par son indice de ligne.

Une fois la limite de l'image atteinte, 'max' contient bien la valeur du dernier pixel à convenir, autrement dit le coin Est-tribord.

Reste à identifier ce que "convenir" signifie. Un pixel de la colonne 'frontiereEst' appartient bel et bien à cette frontière si et seulement si :

1. il est de la couleur du bloc,
2. il a été traité (cela évite qu'il appartienne à un autre bloc de la même couleur),
3. son voisin dans la direction fixée est d'une couleur différente (sinon il n'est pas sur la frontière).

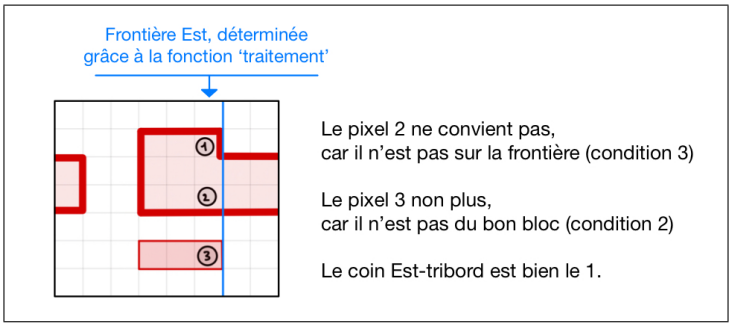

FIGURE 11 – Exemples de pixels ne respectant que 2 conditions sur 3.

On obtient donc cette condition à tester :

```
if (traites.tab[lin * traites.w + extrm.e] == 1
    && isColoredBy(img->mat[lin * img->w + extrm.e], color)
    && !isColoredBy(img->mat[lin * img->w + incr_j(img, extrm.e)], color))
```

FIGURE 12 – Conditions qu'un pixel doit remplir pour "convenir".

Ainsi, après avoir regroupé en une seule fonction 'getBloc' ces étapes (ainsi que d'autres fonctions auxiliaires ne méritant pas de figurer ici), j'étais en mesure de localiser n'importe quel coin du bloc où se trouvait l'interpreteur.

2.6 Les différentes actions possibles en cas de bloc codant

Pour ne pas surcharger les fichiers "interpreur.[h/c]", j'ai préféré consacrer un fichier "action.[h/c]" aux différentes actions possibles selon la différence de couleurs codante entres les pixels de départ et d'arrivée.

L'implémentation de la plupart des actions était toute trouvée, puisqu'elles ne nécessitaient qu'un court enchaînement de fonctions de manipulation de piles (push, pop).

J'ai simplement préféré ajouter une fonction aux fichiers "stack.[h/c]" pour calculer la taille de la pile. En effet chaque fonction d'action commence par tester s'il y a suffisamment d'éléments dans la pile pour effectuer l'action. Le test est immédiat (complexité constante), puisqu'il suffit de retourner ('top' + 1) mais j'estimais que l'on gagnait en lisibilité des fonctions en dédiant à la taille de la pile une fonction à part entière.

La seule difficulté de cette partie résidait dans la fonction "tourne". En effet, c'est la seule action qui nécessite d'ajouter un élément ailleurs qu'au sommet de la pile.

Néanmoins, ayant fréquemment manipulé des fonctions récursives en option Informatique (CPGE), j'ai rapidement reconnue une situation familière.

L'idée de "tourne" est de déplacer l'élément au sommet de la pile à une certaine 'profondeur' donnée, en n'utilisant que les fonctions de manipulation de piles de "stack.h". Mon idée était alors d'utiliser une fonction auxiliaire, qui prend comme arguments l'élément 'elem' à placer, et la 'profondeur' :

— si la 'profondeur' donnée est nulle, on place l'élément au sommet de la pile (via push),
— sinon :

1. on dépile l'élément au sommet de la pile (pop),
2. on ajoute récursivement 'elem' à la profondeur ('profondeur' - 1),
3. on rajoute l'élément dépilé à l'étape 1.

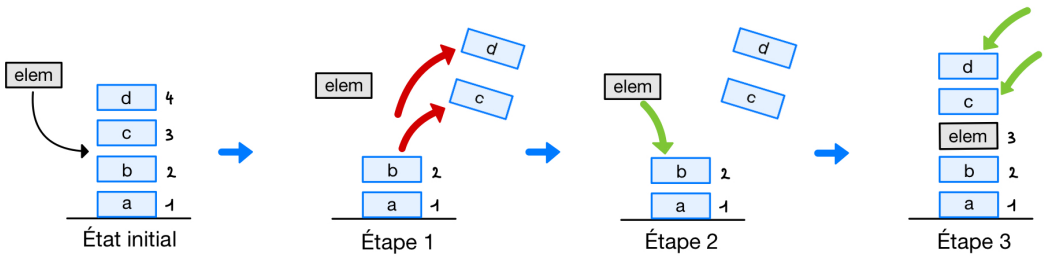

FIGURE 13 – Illustration de l'algorithme récursif.

L'algorithme termine bien, 'pronfondeur' étant fini, et puisqu'on a ré-empilé les élements dans le bon ordre, 'elem' se retrouve bien à la profondeur souhaitée.

Il ne restait plus qu'à écrire la fonction "tourne", dont la seule difficulté restante résidait dans le test du nombre d'éléments de la pile avant toute manipulation. Le soucis est que la fameuse profondeur est elle-même le deuxième élément de la pile. Si la pile contient donc au moins deux éléments, il faut dépiler ses deux premiers, puis tester qu'il reste bien au moins 'profondeur' éléments dans la pile. Sinon, on ré-empile ces deux éléments dans le bon ordre, afin de laisser la pile intacte.

2.7 Étapes de l'exécution

À ce stade, nous avons toutes les parties nécessaires à l'écriture d'une étape du programme. Mises à part quelques fonctions sans grande difficulté (comme la récupération des coordonnées du bon coin du bloc parmis les 8 relevés, par exemple), il ne restait plus qu'à :

1. traiter le cas du bloc de couleur passante,
2. BIEN lire et interpréter le sujet pour ne pas faire d'erreur dans les instructions,
3. agencer l'ensemble en une fonction 'step', qui prend tous les arguments nécessaires à la réalisation de chacun des cas de figure possibles, de sorte qu'on puisse l'appeler en boucle jusqu'à l'arrêt du programme.

2.7.1 Cas délicat : rencontre d'un bloc passant

Dans tout le programme, l'interpreteur se déplace seulement de deux façons :

1. au début de l'étape, pour aller sur le bon coin, selon ses direction et bord,
2. et une fois dessus, pour aller sur le pixel voisin, de l'autre côté de la frontière

La seule exception est lorsque l'on rencontre un bloc passant. Dans ce cas, on continue d'avancer dans la même direction, jusqu'à rencontrer un nouveau bloc. Alors seulement on peut regarder le pixel voisin, et agir en conséquence.

J'ai ainsi écrit une fonction 'moveUntilNewBloc' qui récupère les coordonnées du point de la frontière diamétralement opposée du bloc passant, en continuant d'incrémenter la ligne ou colonne (selon la direction) tant que le pixel est de couleur passante.

Pour ce qui est d'identifier la bonne action à effectuer, je m'en suis occupé au sein même de la fonction 'step', sans fonction auxiliaire.

Puisqu'en cas de rencontre avec un bloc de couleur bloquante, on doit recommencer une nouvelle étape à partir du bloc passant, j'ai décidé de déplacer l'interpreteur sur le bloc passant avant de le traverser, quitte à devoir re-bouger si l'on atteint un bloc de couleur codante de l'autre côté.

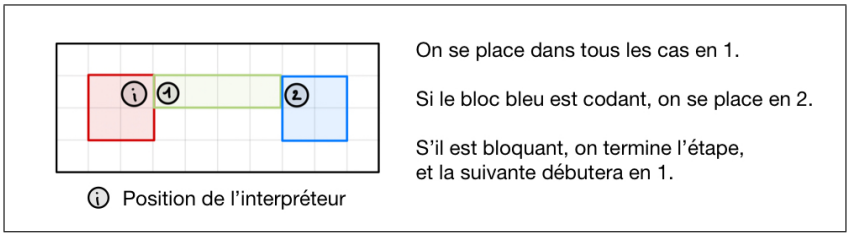

FIGURE 14 – Passage du bloc rouge au beige, passant.

Se faisant, j'ai dû veiller à vérifier que l'on se trouvait bien nous-même sur une couleur codante avant d'effectuer toute action dûe à l'arrivée sur un nouveau bloc codant, pour respecter la consigne selon laquelle on n'agit pas si l'on a traversé un bloc passant.

2.7.2 Interprétation du sujet

En lisant la description d'une étape du programme, j'ai eu un doute. Doit-on continuer d'alterner entre changer la direction ou le bord en passant d'un bloc bloquant à un autre, où doit-on toujours commencer par changer le bord, puis alterner tant qu'on reste sur le bloc courant ?

J'ai d'abord pensé que même en ayant changé de bloc, si lors de la dernière rencontre avec un bloc bloquant, on avait changé le bord, alors il fallait cette fois-ci changer la direction. Mais c'est en comparant mes résultats avec le comportement attendu d'images données en exemple que j'ai compris que non : il faut bien toujours commencer par changer le bord, puis alterner.

2.7.3 La fonction 'step'

Le programme consiste à enchaîner les étapes jusqu'à arriver à notre unique condition d'arrêt : on a rencontré 8 voisins bloquants depuis le même bloc. En réalité, le sujet nous suggérait ainsi la manière d'implémenter la condition d'arrêt intuitive : quand il n'y a nulle part où continuer, puisque chacun des 8 coins mènent à un bloc de couleur bloquante. J'ai donc passé en argument de la fameuse fonction 'step' un compteur de voisins bloquants, à remettre à 0 dès que l'on change de bloc codant, où si l'on traverse un bloc passant.

Le corps de la fonction suit la description de l'étape :

1. on parcourt le bloc courant pour localiser le "bon coin", selon la direction et le bord de l'interpreteur,
2. on regarde si le voisin est codant, bloquant ou passant :
   — si le voisin est CODANT, on s'y place, et si l'on était nous-même sur une couleur codante, on effectue l'action correspondante à la différence de couleur et luminosité entre les deux blocs.
   — si le voisin est BLOQUANT, on change de direction ou de bord, puis on incrémente le compteur de bloquants. On termine l'étape, puisqu'ayant changé soit de direction soit de bord, l'interpreteur rebougera au début de l'étape suivante.

     Pour savoir s'il faut changer la direction ou le bord, il suffit de regarder la parité du compteur de bloquants : pair, on change de bord ; impair, de direction.

   — si le voisin est PASSANT,
     (a) on y place l'interpreteur,
     (b) on continue dans la même direction jusqu'à atteindre la frontière du bloc passant, et on regarde le voisin :
        — si le voisin est bloquant, on termine l'étape. Puisqu'on s'est placé sur le bloc passant, la prochaine étape débutera depuis ce-dernier.
        — si le voisin est codant, on s'y place, mais on n'effectue aucune action.

        — REMARQUE : si le voisin est lui aussi d'une (autre) couleur passante, peu importe. En effet, il faudrait là encore continuer dans la même direction jusqu'à tomber sur une couleur codante ou bloquante, alors autant continuer notre traversée tant que la couleur est passante, sans se soucier de la couleur exacte.

     (c) Dans tous les cas, on remet le compteur de bloquantes à 0, puisque l'on a traversé une couleur passante.

Maintenant qu'on a notre étape, il ne reste plus qu'à l'effectuer en boucle, dans "main.c".

2.8 Écriture de "main.c"

Il ne restait plus qu'à tout mettre en place :

1. récupérer le nom du fichier, stocké dans argv[1] lorsqu'il est passé en argument lors de la commande :

```
$ ./exec cornelis/hw2.ppm
```

FIGURE 15 – Commande bash pour exécuter l'image hw2.ppm.

2. convertir notre image.ppm en tableau de pixels,
3. initialiser un interpreteur,
4. initialiser un compteur (de voisins bloquants) à 0, et appeler la fonction 'step' jusqu'à ce qu'il atteinge 8 :

```
int countBloquantes = 0;
while (countBloquantes < 8) {
    step(pImg, &inter, &countBloquantes);
}
```

FIGURE 16 – Boucle while exécutant le programme.

NB : on suppose que le compteur finit toujours pas atteindre 8. J'ai réfléchi à un moyen de sécuriser le programme face à des comportements infinis, mais n'ai pas eu le temps d'aboutir (Cf partie 4.2).

3 Questions bonus

3.1 Environnement de débogage

Afficher des informations à l'écran lors de l'exécution du programme, c'est déjà ce que je faisais par moi-même pour localiser les bugs. Faire un environnement de débogage à proprement parler ne demandait donc que quelques ajustements :

3.1.1 Qu'afficher, et comment

Lors de mes propres débogages, j'affichais

1. l'état intégral de l'interpréteur,
2. de sa pile,
3. les composantes RVB du pixel actuel ainsi que de son voisin dans la même direction,
4. l'action réalisée.

J'ai donc repris ces informations en leur redonnant forme. Pour se faire, j'ai dû écrire une deuxième version de certaines fonctions-clés (comme 'step' notamment), affichant une série d'informations à l'écran. Je ne détaille pas ces fonctions ici : les différences de versions ne me semblent pas primordiales, et le code est abondamment commenté.

3.1.2 Entrée dans l'environnement

Je ne voulais pas proposer à l'utilisateur.ice d'entrer dans l'environnement de débogage au moment de l'exécution, car je tenais à ce que la commande :

```
$ ./exec cornelis/hw2.ppm
```

FIGURE 17 – Commande bash pour exécuter l'image hw2.ppm.

donne bien uniquement et directement le retour attendu.

J'ai donc ajouté la lecture d'un éventuel argument "-debog" spécifié lors de l'appel du programme, qui ouvre directement l'environnement de débogage.

```
$ ./exec cornelis/hw2.ppm -debog
```

FIGURE 18 – Commande bash pour exécuter hw2.ppm dans l'environnement de débogage.

Le comportement à l'intérieur de la fonction 'main' dépend donc du nombre d'arguments passés lors de l'exécution du programme. J'ai donc adapté "main.c", le test du nombre d'arguments (via 'argc') menant à présent à deux boucles while différentes selon si l'option "-debog" avait été demandée : l'une sans affichage d'informations à l'écran, l'autre avec.

3.1.3 Demande de confirmation avant de passer à l'étape suivante

La dernière chose à faire était de trouver comment attendre confirmation de l'utilisateur.ice avant de passer à la prochaine étape.

J'ai donc pris soin d'ajouter, à l'intérieur de la boucle while réalisant les étapes, la demande (via 'scanf') d'un caractère de la part de l'utilisateur.ice : 'n' (pour "next") pour passer à l'étape suivante, ou 'q' pour "quitter" l'environnement et arrêter le programme. Afin d'éviter tout comportement innatendu de la part de l'utilisateur.ice, on continue de demander un caractère via 'scanf' jusqu'à ce que le caractère entré soit 'n' ou 'q'.

Pour prendre ce caractère en compte, j'ai ajouté comme deuxième condition possible d'arrêt de la boucle (en plus du compteur atteignant 8) que ce caractère soit 'q'.

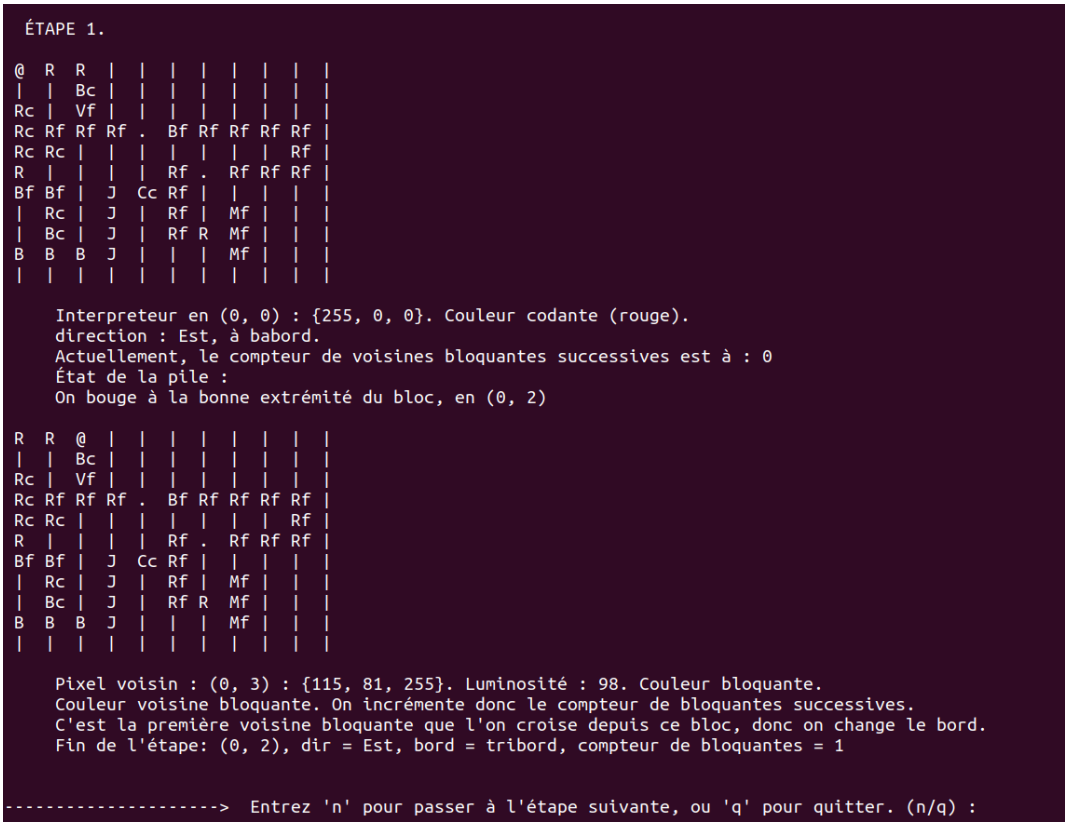

FIGURE 19 – Affichage de l'étape 1 de l'exécution de "euclide.ppm".

3.2 Traitement d'image au format JPEG

Quand le sujet proposait de permettre au programme de lire d'autres formats dd'image que PPM, j'ai tout de suite pensé au JPEG, puisque c'est le seul dont le nom m'est familier depuis longtemps.

Pour écrire la fonction de conversion d'une image JPEG en tableau de pixels équivalente à celle écrite pour le format PPM, j'ai dû me documenter sur la librairie "jpeglib.h".

C'est la partie du projet qui m'a le moins plu : comme je n'allais pas réinventer la roue, je n'ai fait que reprendre une fonction déjà existente pour exploiter une image JPEG en C. Tout ce que je pouvais faire était de comprendre le fonctionnement de la fonction, en particulier l'utilisation d'outils de décompression, mais je n'avais pas à "trouver" la solution moi-même.

J'ai testé la fonction en donnant au programme des images PPM converties (en ligne) au format JPEG, mais je n'obtenais pas les comportements attendus. En cherchant le problème, il s'est révélé que la grande majorité des pixels obtenus étaient extrèmement proches de ceux attendus, mais avec une petite différence suffisante pour fausser le programme.

Par exemple, le pixel (0, 0) de l'image euclide.ppm est rouge : {255, 0, 0}. Pourtant dans mon tableau obtenu, ce pixel est {253, 0, 2}. La différence est minime, mais suffisante pour que le pixel ne soit pas identifié comme rouge, donc comme codant, et ainsi entièrement fausser le programme. Il en est de même pour presque tous les pixels.

Mes hypothèses sont que le problème vient soit :

1. de la décompression de l'image afin d'établir le tableau de pixels,
2. de la conversion des images PPM données en exemple en JPEG, effectuée sur différents sites en ligne.

Préférant me concentrer sur les autres problèmes du projet ne nécessitant que mes connaissances en C plutôt que ma documentation sur le format JPEG, je n'ai malheureusement pas résolu ce problème.

4 Limites du programme

Je suis dans l'ensemble très satisfait de mon projet. 14 des 15 images données en exemple fonctionnent parfaitement, le résultat est obtenu immédiatement sur mon ordinateur. J'ai même pu peindre "MESS" dans "pietquest.ppm", et cela me comble de joie. Je suis de manière plus générale content de mon organisation, et j'ai le sentiment de m'y être pris méthodiquement.

Néanmoins, je reconnais quelques limites à mon programme.

4.1 Le cas "euclide.ppm"

Constat : l'image "euclide.ppm" ne donne pas le résultat attendu pour certaines valeurs entrées. Par exemple, le programme affirme que pgcd(5, 2) = 2, ou encore que pgcd(13, 9) = 4.

En analysant l'exécution du programme étape par étape, j'ai observé que tout se joue à un endroit précis, au moment d'effectuer l'action "plus_grand". En effet, selon si le reste de la division euclidienne est ou non plus grand que 1, le programme réalise une boucle de plus. C'est le même principe que lors de l'exécution de l'image "power.ppm", qui elle marche sans soucis.

Après recherches, il me semble que le problème vient de l'inégalité à tester dans l'action "plus_grand" : le second élément doit-il être STRICTEMENT supérieur au premier ? Le sujet ne précisant pas "strictement", j'ai d'abord utilisé une inégalité large :

```
stack_push(inter->pS, (second >= premier) );
```

FIGURE 20 – Test d'inégalité large.

```
stack_push(inter->pS, (second > premier) );
```

FIGURE 21 – Test d'inégalité stricte.

Dans le 1er cas, "euclide.ppm" fonctionne pour tous les couples d'entiers testés. Mais c'est alors "power.ppm" qui boucle clairement une fois de trop, affirmant par exemple que $2^3 = 16$, $5^2 = 125$, etc.

En utilisant l'inégalité stricte en revanche (2e cas), "power.ppm" fonctionne sans soucis, mais "euclide.ppm" se trompe pour certaines valeurs.

Puisque "power.ppm" bouclait systématiquement une fois de trop avec l'inégalité large, alors que "euclide.ppm" ne se trompait que pour certaines valeurs, j'ai laissé la stricte. Mais à moins que l'image ait été conçue pour une inégalité large, et soit alors incompatible avec "power.ppm", je n'ai pas réussi à expliquer ce comportement innatendu.

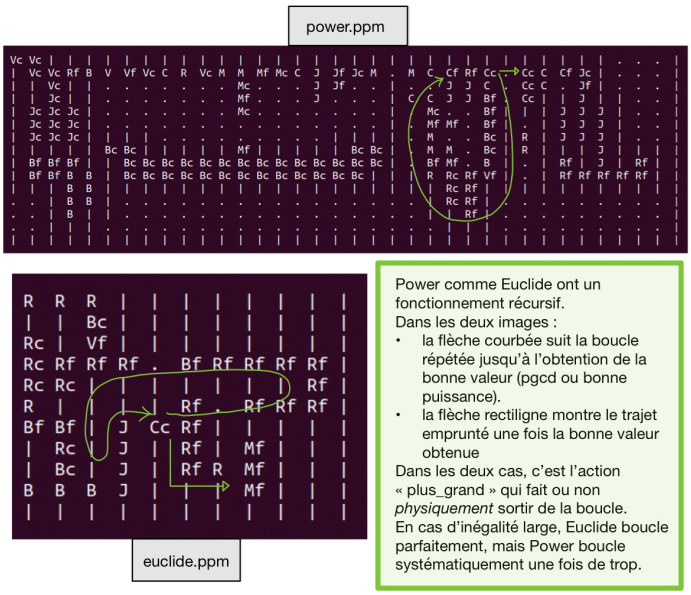

FIGURE 22 – Visualisation des boucles "while" d'euclide.ppm et ower.ppm

4.2 Sécurité anti boucle infinie

Le sujet ne demandant pas explicitement d'établir une sécurité en cas de boucle infinie (si l'on n'atteint jamais 8 couleurs bloquantes), je ne l'ai pas fait par manque de temps.

J'ai néanmoins bien réfléchi à la problématique. Je pensais d'abord qu'on pourrait sans doute vérifier que l'on ne se répète pas en gardant trace des blocs déjà traversés, et de l'état exact de l'interpréteur à leur sortie.

En effet, si l'on repasse par un bloc déjà traversé, et qu'on en sort avec la même direction, le même bord et exactement la même pile, alors on est sûr d'être dans une boucle infinie : le parcours étant déterministe, on reviendra à coup sûr encore et encore au même bloc, avec encore l'interpreteur dans le même état.

Néanmoins, avoir l'interpreteur dans l'exact même état est une condition suffisante, mais pas nécessaire pour boucler indéfiniement. En effet, on peut imaginer qu'on retourne à un bloc déjà traversé avec la même direction, le même bord, mais avec simplement un élément de plus au sommet de la pile. L'interpreteur n'est pas exactement dans le même état que lors du passage précédent par ce bloc, pourant le programme ne terminera pas.

J'ai alors imaginé qu'il fallait retenir les actions effectuées entre temps, et les éléments exacts qu'elles empilaient/dépilaient, pour vérifier qu'on ne bouclait pas. Mais je ne voyais pas de manière triviale de comparer cette liste d'actions à celles déjà effectuées entre n'importe quels blocs, et j'ai estimé que je n'allais pas avoir le temps d'aboutir à quelque chose de concluant.

4.3 Re-calcul du bloc à chaque fois

Mon programme pourrait être légèrement optimisé. En effet, actuellement, quand on achève une étape sans changer de bloc (car on s'est heurté à un voisin de couleur bloquante), on re-parcours ce bloc alors qu'on avait déjà obtenu ses informations (taille et coordonnées des 8 points). On pourrait alors gagner du temps de calcul en conservant les données des blocs, ou au moins celle du bloc de l'étape précédente. Ainsi, si l'on n'a pas changé de bloc, on peut directement accéder à ses données.

La majorité des fonctions du projet sont de complexité constante. La seule davantage gourmande est justement le parcours de bloc, de complexité quadratique en O(image.hauteur * image.largeur). Éviter un parcours inutile ferait gagner en temps, d'autant que notre parcours récupère les coordonnées des 8 coins simultanément.

4.4 Aucune utilisation de 'enum'

N'ayant jamais utilisé 'enum' auparavant, je n'ai pas eu le réflexe de l'utiliser dans ce projet. Je pense pourtant que certaines fonctions auraient gagné en lisibilité. Un 'enum' pour la direction de l'interpreteur par exemple, aurait permis d'utiliser les noms des directions (NORD, etc) plutôt que les valeurs que je leur ai attribuées.

Ceci dit, cela n'aurait rien changé en terme d'efficacité du programme, 'enum' n'étant qu'une manière de présenter différemment des entiers.

4.5 GDB

Ce n'est pas vraiment une limite du programme, mais je regrette légèrement de ne pas avoir profité de ce premier projet pour découvrir GDB. Je l'ai lancé à un moment, mais j'ai estimé qu'apprendre à m'en servir efficacement allait me prendre plus de temps que de localiser les sources de bugs en ajoutant des "printf" manuellement, simplement parce que j'étais familier de cette méthode. Il est pourtant certain que l'investissement serait bénéfique à long terme.

4.6 Un bug réglé mais resté incompris

J'ai rencontré un bug inattendu lors de la définition des 18 couleurs codantes. J'avais d'abord défini les 18 couleurs codantes dans mon fichier "couleur.h" ainsi :

```
pixel rouge = {255, 0, 0};
```

FIGURE 23 – Exemple : définition initiale de la couleur rouge.

mais j'obtenais une erreur de "définitions multiples", et ce alors même que :

1. ces 18 constantes n'étaient bien définies qu'ici,
2. je n'avais pas de problème d'inclusion infinie de fichiers headers, puisque j'avais proprement testé leur inclusion via 'ifndef' etc.

En cherchant sur Internet, j'ai lu que déclarer mes constantes dans "couleurs.h" comme 'extern', et de ne leur attribuer leur valeur que dans "couleur.c" réglerait sans doute mon problème. Cette solution proposée fonctionnait, donc je l'ai gardée. Mais je dois avouer, et c'est le seul élément de tout mon programme, que je n'ai pas compris d'où venait le problème, ni la solution.

5 Conclusion

Comme écrit plus tôt, je suis très satisfait de mon projet. À part "euclide.ppm", j'obtiens le comportement attendu pour toutes les images données en exemple, même "pietquest.ppm". J'estime également avoir achevé mon code en un temps très raisonnable.

Ce projet m'a par ailleurs beaucoup apporté. Il m'a d'une part permis de consolider mes connaissances en C, mais également de réflechir à comment diviser une vaste tâche en plusieurs petites étapes. C'était de plus l'occasion pour moi de m'entraîner à commenter du code en anglais, ce que je serai sans doute amené à faire un jour.

Sur le plan technique, je retiendrai deux idées de programmation :

1. contrairement à ce que dicte mon réflexe premier, implémenter un cycle n'implique pas forcément l'utilisation de listes chainées : un modulo convient parfaitement,
2. lors de la recherche de quelque chose dans un espace à n dimensions, il est malheureusement possible de devoir parcourir tout l'espace, malgré une idée de la direction.

Enfin, ce projet m'a beaucoup fait réfléchir à la suite pour moi : j'ai beaucoup aimé travailler sur un projet de programmation ambitieux, devoir établir une stratégie d'ensemble, et enfin trouver les solutions moi-même. J'ai trouvé confirmation néanmoins que ce que j'aime, c'est utiliser la programmation comme outil pour résoudre des problèmes concrets. La partie ouverture d'images PPM ou JPEG m'a beaucoup moins intéressé que la recherche des coins dans le bloc, ou la réalisation d'une étape du programme, par exemple.