Rapport de Projet : ImagesTaches

Nawfoel Ardjoune &Chaolei CAI

December 16, 2019

Contents

1 Introduction

Ce document constitue notre rapport sur le projet de fin de semestre L2 pour le cours d'impérative 2 de M.Bourdin.
Le projet est consultable depuis ce lien github: https://github.com/bk211/ImagesTaches

2 Dependences

Ce projet nécessite le support de l'API GL4Dummies de M.Belhadj, par extension, vous devez avoir sur votre machine les dépendances de GL4Dummies (SDL2 et OpenGL pour ne citer que les plus importants).

3 Compilation du projet

Un makefile est présent dans le répertoire de racine, pour compiler le programme, une simple commande "make" suffit à obtenir l'exécutable "exec".

Enfin, lancer le programme exec pour faire appraître la fênetre d'affichage.
Par défaut l'image affiché est l'image avant le traitement,
Pour afficher le l'image avant l'application du traitement, pressez la touche "v" de votre clavier.
Pour afficher le l'image après l'application du traitement, pressez la touche "b" de votre clavier.
Pour passer à l'image suivante, pressez la touche "n" de votre clavier.
Pour terminer le programme, vous pouvez presser la touche "ECHAP", "q" ou la croix situé dans le coin supérieur droite.

4 Objectives et consigne du Projet

Il faut sur, une image, trouver automatiquement toutes les "taches de couleur".
L'image est donnée comme un grand tableau avec un octet par couleur par "pixel" (trois couleurs, R, G et B).
C'est donc une vaste matrice de (XMAX x YMAX x 3) octets. Trouver les taches consiste à trouver et numéroter les zones connexes ayant la même couleur.
Une bonne méthode consisterait à commencer par trouver tous les pixels connexes ayant une certaine couleur, ils forment une tache.
Puis à réitérer ce processus pour tous les pixels. On a alors toutes les taches, si elles ont été listées, il est maintenant facile de leur appliquer un traitement.
Un traitement pourrait être de délimiter en noir les taches de couleur. Enfin, une tache peut être une zone dont les pixels ont à peu près la même couleur. On définira cette approximation et on fera la délimitation de ces zones comme précédemment.
Le rapport contiendra des explications sur les procédures mises en oeuvre, le mode d'emploi, le listing du programme et des traces d'utilisation.
Le programme devra tourner sur au moins une des machines en libre-service du Bocal et sera écrit en C, avec, si nécessaire, utilisation des librairies OpenGL, Glu, Glut et GL4D, seules librairies non standard autorisées.

5 Structures de données de base

```
typedef unsigned char color;
```

```
typedef struct pixel pixel;
struct pixel{
    color R;
    color G;
    color B;
};

typedef struct image image;
struct image{
    color * tab;
    int w;
    int h;
};
```

Tout d'abord, "color" est un alias que j'ai donné au type unsigned char (il occupe précisément 1 octet dans la mémoire), il a été mise en place pour quantifier la couleur dans le système RGB, en effet, dans ce dernier, les valeurs que peuvent prendre une couleur est comprise dans l'intervalle [0;255].
Par extension, on arrive à notre structure pixel, qui est constitué de 3 composants "color" RBG.
Enfin, une image seras pour nous une structure composé de 3 élements:
Le premier, "tab" est le tableau de couleur qu'il faudra alloué dynamiquement selon le besoin.
"w" et "h" sont les variables qui indique les dimensions de l'image.
J'ai choisi de prendre un tableau à simple entrée comme nous avons déja les dimensions de l'image, la manipulation des indices n'est pas fondamentalement plus dur d'un tableau à 2 entrée, ce n'est qu'une histoire de conversion entre les coordonnées x et y.
A vrai dire, dans les première versions, il s'agit plutôt d'un pointeur vers une structure "pixel" que nous avons utilisée. Mais comme dans la consigne, il est dit qu'il faut utiliser une matrice de (XMAX x YMAX x 3) octets, nous sommes revenue vers le pointeur vers "color".

6 Explications sur les fonctions utilisées

6.1 image.c

pixel build_pixel(color tab, int pos)
retourne une structure pixel à partir des couleurs qui se situent en position

"pos" du tableau de couleur donné en paramètre.

create_pixel(color R, color G, color B)
retourne une structure pixel à partir des couleurs données en paramètre.

void modify_pixel(color tab,int pos, color R, color G, color B)*
Comme son nom l'indique, il modifie les couleurs situées à la position "pos" du tableau avec les paramètres donnés.

*void affiche_pixel(color * tab,int pos)*
Affiche le trio de couleur situé à la position pos.

image create_image(int w, int h)
C'est la fonction d'intialisation d'image, il retourne une structure image avec la bonne dimension tout en allouant la mémoire qu'il faut pour le tableau. Cependant, il ne remplie pas le tableau avec une valeur de défaut.

image cpy_image(image src)
Crée une copie de l'image src et la renvoie.

image create_test_image(int i)
La fonction retourne différents images de test selon l'argument donnée en paramètre.

*int compare_pixel(pixel pi, color * tab, int pos, int option)*
Retourne le résultat de la comparaison entre le pixel et le trio de couleurs à la position pos.
"option" correspond en réalité à un macro qui se situe dans "image.h"
S'il est nul, alors la fonction fait un test strict entre les 2 couleurs.
Sinon on fait alors une comparaison "soft", c'est à dire qu'on considère identique 2 pixel s'ils ont des intensité de couleur proche. Il y a aussi 2 macro dédiés "DIFF_TOT" et "DIFF_MONO": "DIFF_MONO" est la différence maximale qu'on accepte pour un seul couleur. "DIFF_TOT" est la différence maximale qu'on accepte pour la totalité des couleurs,

void affiche_image(image img)
Permet d'afficher l'image sur l'écran du terminale, le résultat est biensûr très moche, et devient rapidement illisible pour de grande dimension, cependant,

il a été très utile pour visualiser rapidement l'image pendant la phase de developpement du projet, l'affiche via openGL n'intervient que très tardivement dans notre projet.
*void free_image(image * src)*
Fonction qui libère l'espace mémoire utilisé par l'image (plus précisement celui pointé par "tab").

6.2 propager.c

int propager(image g, int tab[], int i, int num_tache, pixel p)
Dans ce fichier, il n'y a qu'une seule fonction, c'est la fonction de test pour identifier les différentes taches de couleurs.
Néanmoins, elle est très facile à comprendre.
"tab" est un tableau constitué de "0" à l'état initial, un peu comme ce que nous avons appris dans votre cours pour les arbres, les cases non traités contiennent des "0", à l'inverse, si une case n'est pas nul, il contient forcément le numéro de tache qu'on lui a attribué. (il est instantié par la fonction supérieur qui appeleras propager).
"i" est le numéro de case que la fonction est en train de traiter.
Enfin "p" est la couleur de réference pour faire les tests de comparaison.
Au départ, il faut vérifier si la case "i" est déja traité ou non, dans le cas nul, il faut lancer le test de couleur pour voir si les 2 pixels(trio de couleurs et pixel par abus) sont identique. Si la comparaison s'avère bonne, la case "i" est marqué avec le numéro de tache donné en argument. Puis la fonction se propage récursivement dans les 4 cases en proximité (gauche, haut, droite et bas).
Supposons nous une image de 10x10, divisée par 2 taches de couleurs différentes, après la boucle de traitement via propager, nous aurons le résultat suivant:

1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2
1	1	1	1	1	2	2	2	2	2

| 1| 1| 1| 1| 1| 2| 2| 2| 2| 2
| 1| 1| 1| 1| 1| 2| 2| 2| 2| 2
| 1| 1| 1| 1| 1| 2| 2| 2| 2| 2

Les 2 taches ont bien été marqué par 2 numéro différentes.

6.3 traitement.c

void affiche_tab(image g,int tab[])
Cette fonction a pour but d'afficher le tableau "tab" cité dans la section précedente, c'est pour des besoin de visualisation lors du developpement qu'elle a été crée.

void init_tab(image g,int tab[])
Comme son nom l'indique, la fonction initialise toutes les cases du tableau "tab" avec "0", à vrai dire, il n'est pas nécessaire de donner image g en argument, nous avons juste besoin de connaitre les dimensions du tableau voir même juste sa taille dans ce cas présent. Mais bon, c'est pas une fonction que nous utilisons dans toutes les fonctions, elle a été crée car nous voulions segmenter au maximum les taches.
Et en effet, la fonction a été utilisée au moins 2 fois, ce qui justifie amplement son utilité et d'éviter de copier-coller 2 boucle for identique.

int mark_border(image g, int tab[], int nb_tache)*
Cette fonction continue son travail sur propager qui auras produit un tableau d'entier remplie avec les numéros de taches. Le but de cette fonction est de detecter les bordures entre les taches. Le résultat produit est un tableau qui contient que "0" ou "-1", "0" pour une pixel quelconque et "-1" pour indiquer qu'il s'agit d'une limite de tache.
Pour cela rien de plus simple, il suffit d'iterer sur toutes les cases du tableau de tache, si la case traité a une case en proximité qui a une numéro de tache différente d'elle, alors la case traîté est en bordure de la tache.
Voilà une exemple de sortie

| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0

| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0
| 0| 0| 0| 0|−1|−1| 0| 0| 0| 0

int is_border(int tab[], int w, int h, int i)
Cette fonction permet à mark_border de savoir si un pixel est en bordure de la tache en comparant son numéro de tâche avec ses voisins de proximité.

*void apply_borders(image * dst, int* borders, pixel color)*
Cette permet d'appliquer le traitement de délimitation tant espéré sur une image src, en se basant sur le tableau de bordure "borders" ainsi que la pixel de couleur "pi".

image traitement(image g)
Revenons sur la première fonction de ce fichier, c'est la fonction principale de traitement, qui appeleras tous les fonctions précedentes afin d'appliquer notre traitement. Biensûr, il ne modifie pas l'image d'origine mais crée une copie de cette image source, l'image traité est retourné à l'utilisateur.
Les étapes, une fois segmentées, sont très intuitif:

- Trouver toutes les taches.
- Trouver toutes les limites des taches.
- Créer la copie de l'image source et appliquer le traitement sur cette image.
- Retourner l'image traîté.

6.4 window.c

Ce fichier a été copié depuis un des exemple de cours de M.Belhadj pour le cours de programmation graphique L2.
Nous pouvons diviser les explications sur 3 parties distincts.
static void init(void)
Fonction qui initialise les paramètre OpenGL, ici, nous avons juste besoin d'initier un écran d'affichage et de mettre une couleur de fond de defaut.

static void print_image(image img)
static void draw(void)
Ces 2 fonctions sont intimement liées, elles permettent d'afficher l'image qui est pointé par le pointeur d'image global "displayed_image". Il n'y a rien d'extraodinaire, il suffit de boucler sur l'image, recupèrer le trio de couleur, et de les afficher sur le bon pixel de l'écran.

static void keydown(int keycode)
Enfin, la fonction keydown permet de gérer les évenements quand l'utilisateur presse sur une touche.
Pour nous, il s'agit simplement de modifier la cible qui est pointé par "displayed_image", il y a 2 image qui sont déclaré en globale dans ce fichier: "before_image" et "after_image".

7 Bugs et limite du programme

Le programme est très gourmande en mémoire, dependante du capacité de votre machine, il se peut que la fênetre se ferme sur un seg fault, pour ma part, je peux aller jusqu'à une résolution de 400*400.
Si jamais ce problème appraît sur votre machine, allez dans image.c, puis dans la fonction create_test_image , modifier "default_w" et "default_h" par une valeur plus petite.
A cause des derniers évenements, nous n'avons pas pu tester la dernière partie concernant l'affichage du traitement sur une machine du bocal, en "théorie", il ne devrait pas poser de problème car les machines du bocal dispose de l'API GL4Dummies.
Enfin, une des limite du programme est qu'il ne gère pas les lignes ou verticales de fine épaisseur, le traitement que nous appliquons sur la tache se situe dans la bordure interne, ce qui fait que pour une épaisseur de 1 à 2 pixel, la tache est entierement recouverte par le traitement.