Compte rendu AGUIR Chams \_ TP2 :

**Exercice1 :**

1/



2/

Analysons la structure du code :

- `ppm.h` : Fichier d'en-tête qui définit la structure de l'image PPM et ses fonctions

- `ppm.c` : Implémentation des fonctions de manipulation d'images PPM

- `main.c` : Programme de test qui crée une image simple avec des motifs

- `mandel.c` : Programme principal pour générer l'ensemble de Mandelbrot (actuellement cassé)

3/

#ifndef PPM\_H

#define PPM\_H

// Définition de la structure d'un pixel dans une image PPM (format couleur RGB).

// Chaque pixel est représenté par trois composantes : rouge (r), vert (g) et bleu (b).

struct ppm\_pixel {

unsigned char r; // Intensité du rouge (0-255)

unsigned char g; // Intensité du vert (0-255)

unsigned char b; // Intensité du bleu (0-255)

};

// Fonction inline pour définir la couleur d'un pixel.

// Cette fonction assigne les valeurs r, g et b à un objet de type `ppm\_pixel`.

static inline void ppm\_setpixel(

struct ppm\_pixel \*px,

unsigned char r, unsigned char g, unsigned char b)

{

px->r = r; // Met à jour l'intensité du rouge

px->g = g; // Met à jour l'intensité du vert

px->b = b; // Met à jour l'intensité du bleu

}

// Définition de la structure représentant une image au format PPM.

// Cette structure contient la largeur, la hauteur, et un tableau de pixels.

struct ppm\_image {

unsigned int width; // Largeur de l'image en pixels

unsigned int height; // Hauteur de l'image en pixels

struct ppm\_pixel \*px; // Pointeur vers un tableau de pixels (données de l'image)

};

// Prototypes des fonctions pour manipuler une image PPM.

// Initialise une image PPM avec une largeur et une hauteur spécifiées.

// Alloue la mémoire nécessaire pour stocker tous les pixels.

// Retourne 0 en cas de succès, une valeur négative sinon.

int ppm\_image\_init(struct ppm\_image \*im, int w, int h);

// Libère la mémoire allouée pour une image PPM.

// Retourne 0 en cas de succès, une valeur négative sinon.

int ppm\_image\_release(struct ppm\_image \*im);

// Fonction inline pour définir un pixel spécifique dans une image.

// Met à jour les valeurs r, g et b pour le pixel situé aux coordonnées (x, y).

// Les coordonnées sont basées sur une disposition en matrice (ligne, colonne).

static inline void ppm\_image\_setpixela(

struct ppm\_image \*im, int x, int y,

unsigned char r, unsigned char g, unsigned char b)

{

// Calcule l'adresse du pixel dans le tableau de données.

struct ppm\_pixel \*px = im->px + im->width \* y + x;

// Met à jour les valeurs RGB de ce pixel.

ppm\_setpixel(px, r, g, b);

}

// Sauvegarde une image PPM dans un fichier spécifié par `path`.

// Le fichier est au format texte PPM (P3) ou binaire (P6).

// Retourne 0 en cas de succès, une valeur négative sinon.

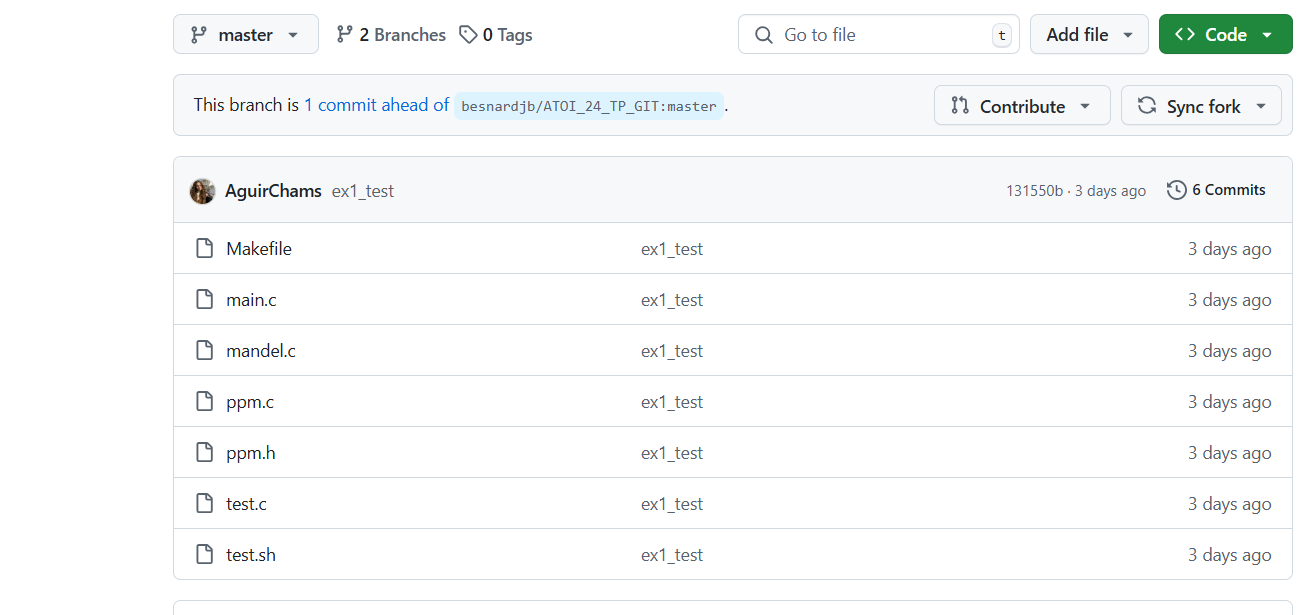
int ppm\_image\_dump(struct ppm\_image \*im, char \*path);

#endif /\* PPM\_H \*/

//création de test.sh

**Exercice2 :**

1/



2/

Explications du code par blocs :

1. Lignes 18-31 : Fonctions de conversion des coordonnées pixels en plan complexe

2. Lignes 35-36 : Initialisation de l'image PPM

3. Lignes 38-60 : Boucle principale de calcul

- Pour chaque pixel, calcule les itérations de l'ensemble de Mandelbrot

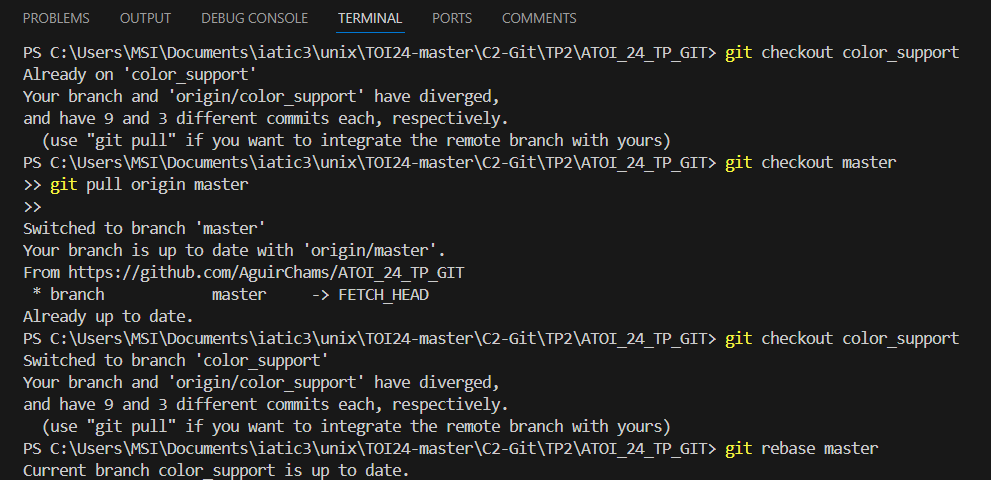
- Convertit le nombre d'itérations en niveau de gris

4. Lignes 62-63 : Sauvegarde de l'image et nettoyage

**Exercice3 :**

1/

Pour ramener la branche color\_support au-dessus de la branche master dans Git, vous pouvez utiliser l'une des deux approches suivantes : **fusion (merge)** ou **rebasage (rebase)**.



2/

Oui, dans certains cas, après un rebase, des **conflits de fusion** peuvent survenir, surtout si des modifications sur master touchent des zones similaires au code sur votre branche.

Si des conflits apparaissent :

1. **Git vous indiquera quel(s) fichier(s) sont en conflit**.
2. Vous devrez résoudre ces conflits manuellement dans les fichiers concernés. Par exemple, dans votre cas, cela pourrait être dans mandel.c, comme nous l'avons vu précédemment.
3. **Après avoir résolu les conflits**, vous devez marquer les fichiers comme résolus :

git add <fichier\_conflit>

1. **Continuez le rebase** :

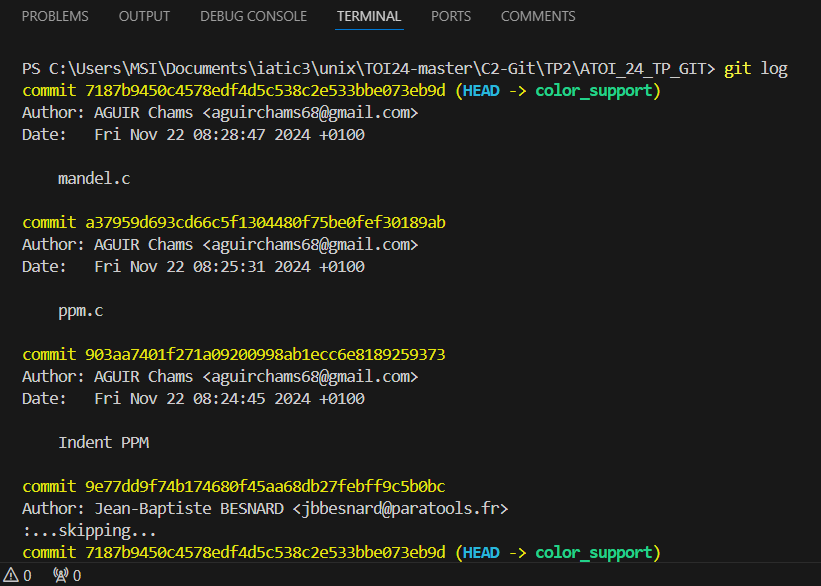
git rebase –continue



Cela va poursuivre le rebase. Si vous avez plusieurs conflits, Git vous demandera de les résoudre un à un.

1. **Terminer le rebase** : Une fois tous les conflits résolus et le rebase terminé, vous pouvez vérifier l’historique des commits avec :

git log



3/

Voici une description détaillée de la gestion de l'opération, en suivant les étapes pour ramener la branche au-dessus de master et résoudre les conflits :

**Étape 1 : Vérification et mise à jour de master**

Tout d'abord, je m'assure que la branche master contient les derniers changements depuis le dépôt distant.

git checkout master # Assurez-vous d'être sur la branche master

git pull origin master # Récupérez les dernières modifications de master

**Étape 2 : Passage sur la branche de travail (**color\_support**)**

Je m'assure ensuite que je suis sur la branche de développement color\_support celle que je souhaite ramener par-dessus master.

git checkout color\_support # Basculez sur la branche de travail

**Étape 3 : Rebase de la branche sur master**

Ensuite, j'effectue un rebase de ma branche de travail sur la branche master. Cela va rejouer mes commits par-dessus ceux de master :

git rebase master # Effectuer le rebase de color\_support sur master

**Étape 4 : Résolution des conflits**

Si des conflits de fusion surviennent, Git les signalera. Dans ce cas, par exemple, il pourrait y avoir des conflits dans le fichier mandel.c comme mentionné précédemment. Git va marquer les parties conflictuelles dans ces fichiers. Voici comment je procède pour résoudre ces conflits :

1. **Identifier les fichiers en conflit** : Git indiquera les fichiers en conflit (par exemple, mandel.c, ppm.c, etc.).
2. **Ouvrir les fichiers en conflit** : J'ouvre les fichiers marqués en conflit pour examiner les sections conflictuelles. Git insère des délimiteurs comme ceci :

<<<<<<< HEAD

(modifications dans la branche master)

=======

(modifications dans la branche color\_support)

>>>>>>> color\_support

1. **Résoudre les conflits** : Je choisis quelle version du code garder (soit celle de master, soit celle de la branche color\_support ou une combinaison des deux), puis je retire les délimiteurs.
2. **Marquer les fichiers comme résolus** : Une fois les conflits résolus dans les fichiers, je les marque comme résolus :

git add mandel.c # Ajoutez les fichiers modifiés

1. **Poursuivre le rebase** : Après avoir résolu tous les conflits et ajouté les fichiers, je continue le rebase :

git rebase --continue

1. **Répéter si nécessaire** : Si d'autres conflits apparaissent, je répète les étapes de résolution et de continuation du rebase.

**Étape 5 : Vérification du rebase**

Une fois le rebase terminé, je vérifie l'historique des commits pour m'assurer que tout s'est bien passé :

git log

Je m'assure que les commits de la branche color\_support sont maintenant au-dessus des commits de master.

**Étape 6 : Pousser les changements**

Si tout est correct, je pousse ma branche mise à jour sur le dépôt distant :

bash

Copy code

git push origin color\_support –force

4/