POLYTECHNIQUE MONTRÉAL

INF1600 - Architecture des micro-ordinateurs

Travail pratique 3

Programmation en assembleur x86 plus avancée et récursivité

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal
Automne 2025

Table des matières

Introduction

Dans ce laboratoire, vous allez explorer le traitement d'images à bas niveau en utilisant le langage assembleur x86. L'objectif principal est de vous familiariser avec des instructions assembleur plus complexes ainsi qu'avec les conventions d'appel de fonctions, la pile du programme et la récursivité.

Vous appliquerez ces concepts en manipulant directement les pixels d'une image pour effectuer des transformations simples en assembleur. Les filtres à implémenter incluent l'inversion des couleurs, la conversion en niveaux de gris et l'application d'un flou simple. Ensuite, vous pourrez explorer des filtres plus avancés, tels que le flood fill récursif, qui mettra à l'épreuve votre compréhension de la récursivité et de la gestion de la mémoire.

Modalité de remise

Remettre sur **Moodle** une archive `.zip` contenant tous les fichiers sources. Le zip peut être créé en exécutant la commande suivante dans le dossier racine du TP :

```
make remise
```

pour générer le fichier suivant qui doit être déposé sur Moodle dans la boîte de remise associée au TP :

INF1600_remise_TP3.zip

Date de remise pour tous les groupes : mercredi 5 novembre 2025, 23:59

Directives particulières

- Vous devez chercher une ou des images en format `.bmp`
- Ajouter les images dans le dossier `/bmp`
- Assurez d'indiquer la source des images dans l'entête du fichier `main()`
- Ajuster le chemin des images dans le fichier `main()`
- 🚨 Toute image offensive entraînera une note de `0`
- Il est suggéré de compléter les fonctions dans l'ordre présenté dans l'énoncé
- La signature de chaque fonction se trouve dans `include/filters.h`
- Les divisions sont entières (pas de décimales)
- Il faut en tout temps respecter les conventions d'appels des fonctions (pénalités).

Fichiers BMP

On utilise des images Bitmap, ou BMP, qui est un format de fichier d'image très simple, habituellement sans compression (quoiqu'il le supporte). Un BMP est composé d'un entête général qui a toujours la même suivi d'un **tableau de pixels**, une ligne après l'autre dont chacune est alignée sur 4 octets.

Pixel

La structure **Pixel** représente un pixel RGBa32. Néanmoins, la composante alpha est présente, mais ne sera pas considérée dans nos calculs.

```
struct Pixel {
  uint8_t b;
  uint8_t g;
  uint8_t r;
  uint8_t a;
};
```

⚠️ Noter l'ordre des couleurs : B, G, R

Image

La structure `Image` représente une image générale, c'est-à-dire avec une largeur, une hauteur et un tableau de pixels. Les pixels sont organisés par lignes, dans un tableau 2D d'objets Pixel contigus (voir figure 1).

```
struct Image {
  uint32_t largeur;
  uint32_t hauteur;
  Pixel** pixels; // voir comme un tableau 2D : Pixel[][]
}
```

Rappel : La taille des images est variable. Puisqu'en C/C++, la taille d'un tableau doit être statique à la compilation (Ex. `Pixel[100][100]`, on utilise des pointeurs pour contourner cette limitation et permettre une taille dynamique.

Un pointeur n'est rien d'autre qu'une **adresse mémoire** (une référence vers une case mémoire).
Ainsi, lorsqu'on déclare un tableau Pixel** image, il ne contient pas directement les pixels, mais **des adresses vers d'autres zones mémoire** où sont stockées les lignes de pixels.

Ainsi, on déclare un pointeur de pointeurs `Pixel** image`, qui représente un tableau dynamique à deux dimensions.

Le premier pointeur `Pixel**` pointe vers un tableau de pointeurs (adresses) `Pixel*`, chacun représentant une ligne de pixel**s** dans l'image.

Chaque ligne, à son tour, pointe vers un tableau de structures `Pixel`, représentant les colonnes (les pixels individuels) de cette ligne.

Puisque l'image est stockée **c**omme un tableau en mémoire, l'indexation suit l'ordre naturel des tableaux :
la coordonnée **Y augmente de haut en bas** (la ligne 0 correspond au haut de l'image, et la dernière ligne correspond au bas, voir figure 1).

En résumé :

- `Pixel** image` est une adresse vers un tableau de pointeurs (Pixel*) où chaque coordonnée (x, y) correspond à `image[y][x]`.
- Chaque `Pixel*` est une adresse vers un tableau de Pixel. C'est le type de `image[y]`.
- Chaque `Pixel` contient les données réelles du pixel (R, G, B, a). C'est le type de `image[y][x]`.

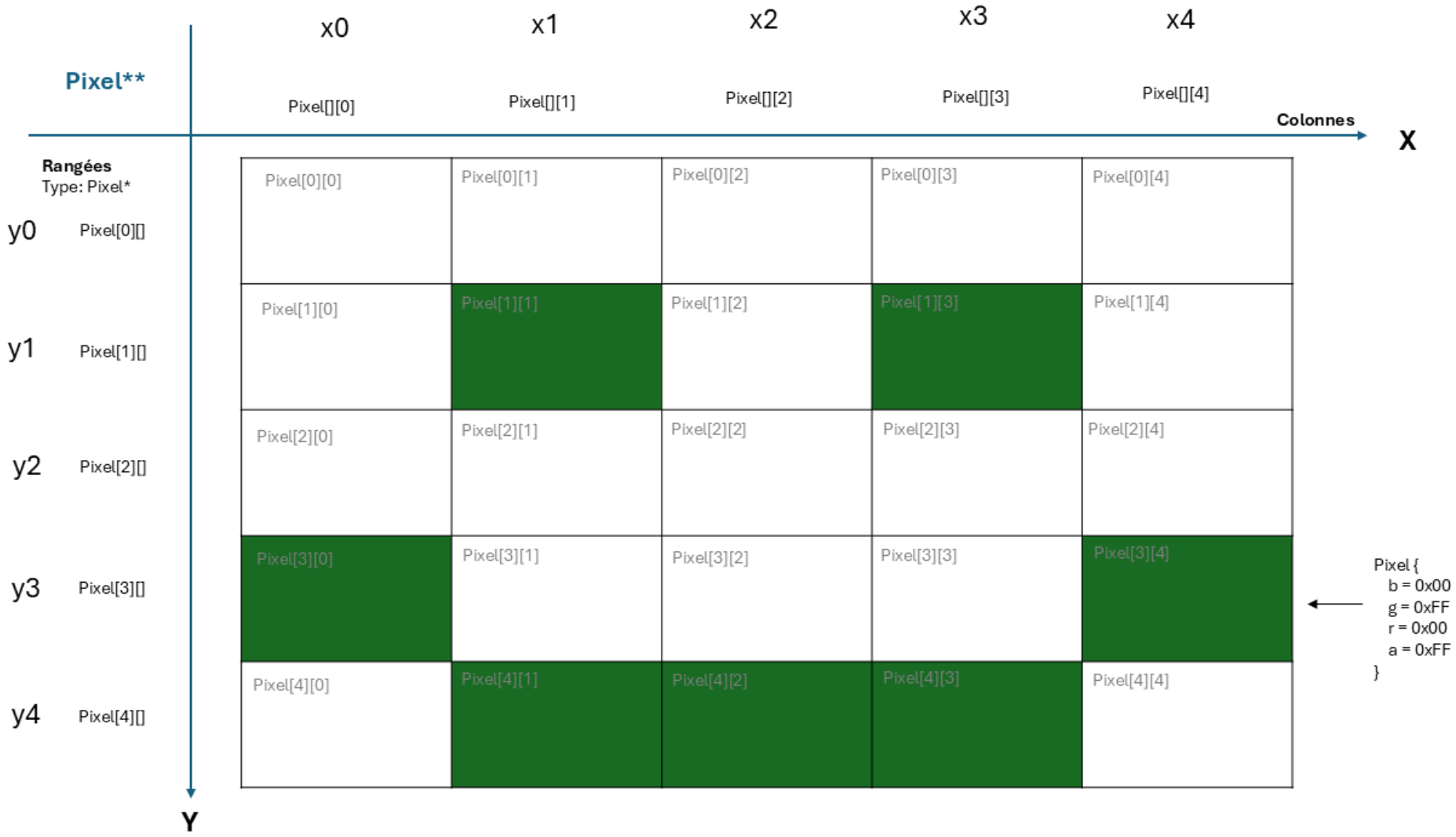


Figure 1 : Représentation du tableau Pixel** en mémoire

isSameColor

Fichier : `src_TODO/isSameColor.s`

Signature : `include/filters.h`

Fonction aidante qui retourne `1` (vrai) si 2 pixels sont égaux et `0` (faux) sinon.
Deux pixels sont égaux si leurs composantes R, G et B sont identiques.

Cette fonction sera utilisée plus tard dans la fonction `floodFill`. Il n'y donc pas de tests explicites dans le `main()`, mais n'hésitez pas à la tester vous-même!

Inversion des couleurs / Négatif

Fichier : `src_TODO/invert.s`

Signature : `include/filters.h`

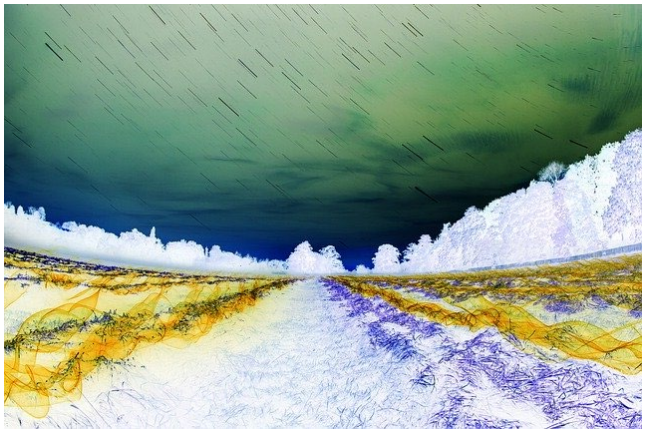
Le négatif d'une image inverse les couleurs de chaque pixel. Chaque composante RGB passe à `255 - valeur`, transformant le blanc en noir, le noir en blanc, et chaque couleur en sa couleur complémentaire.

Afin d'inverser une couleur, il suffit de parcourir le tableau de pixel et appliquer la formule suivante :

$$R = 255 - R$$
$$G = 255 - G$$
$$B = 255 - B$$

Exemple de résultat attendu

Avant



Après

Source de l'image : https://www.wallpaperflare.com/green-plants-under-shooting-star-nature-landscape-trees-depth-of-field-wallpaper-pwjgd

Nuances de gris / Grayscale

Fichier : `src_TODO/grayscale.s`

Signature : `include/filters.h`

Chaque pixel prend une seule valeur d'intensité lumineuse, généralement entre 0 (noir) et 255 (blanc), représentant des nuances de gris. On utilise la moyenne des composantes RGB pour définir l'intensité.

La moyenne des composantes RGB d'un pixel se calcule ainsi :

$$Gris = \frac{R + G + B}{3}$$

où R, G et B sont les valeurs de rouge, vert et bleu du pixel. Cette valeur peut ensuite être utilisée comme intensité pour convertir le pixel en niveau de gris :

$$R = G = B = Gris$$

Exemple de résultat attendu

Avant

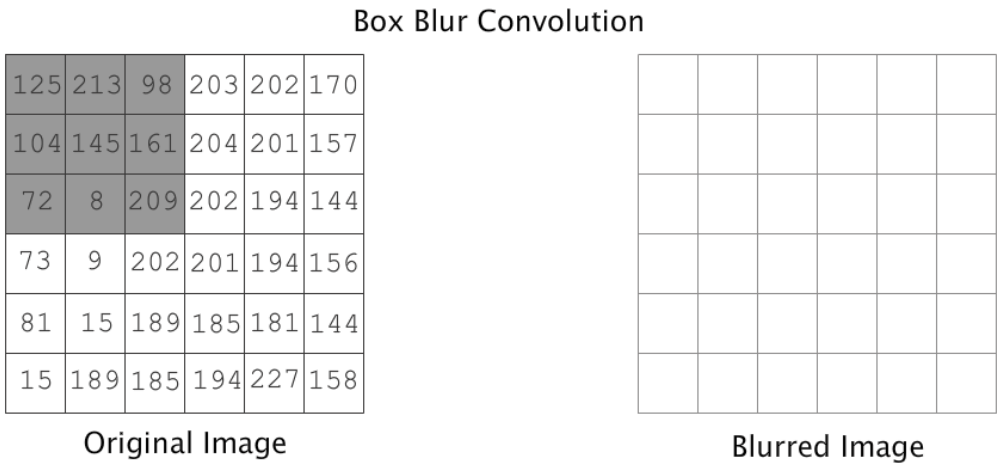
Après

Flou simple / Box blur

Fichier : `src_TODO/blur.s`

Signature : `include/filters.h`

Le *box blur* est un flou simple appliqué à une image où chaque pixel est remplacé par la moyenne des couleurs de ses voisins dans une fenêtre carrée de 3×3.



Source de l'animation : https://medium.com/hackernoon/cv-for-busy-developers-convolutions-5c984f216e8c

Puisqu'on a besoin des voisins pour la moyenne, on a besoin d'une copie de l'image qui reste inchangée durant l'application de l'algorithme. La fonction `blur` prend donc 2 paramètres, la référence vers l'image qui sera floue (`img`), ainsi que la référence vers la copie de l'image (`imgCopy`). Garder la copie de l'image intact et l'utiliser pour le calcul de la moyenne.

✅ INDICE : On doit faire la moyenne des voisins dans toutes les directions, sans dépasser les bornes du tableau.
On itère donc
de **`x = 1 à x < largeur - 1`**
et **`y = 1 à y < largeur - 1`**.

Remarquez que les pixels en bordure demeurent inchangés.

Exemple de résultat attendu

Avant

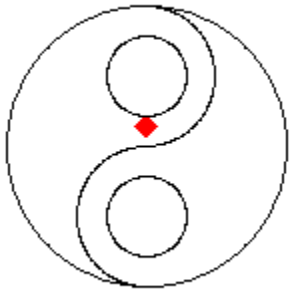
Après

Remplissage par propagation / Floodfill

Fichier : `src_TODO/floodFill.s`

Signature : `include/filters.h`

Le *flood fill* est un algorithme récursif qui remplit une zone connectée d'une image avec une couleur donnée, en partant d'un pixel initial et en étendant le remplissage à tous les pixels voisins ayant la même couleur de départ.



Voici un exemple de code C++ de cet algorithme. (L'implémentation de l'algorithme est à votre guise)

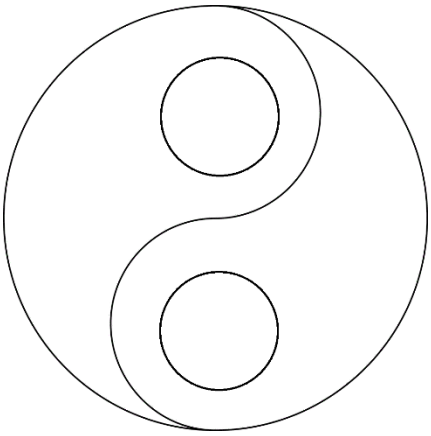
```
void floodFill(Image& img, int x, int y, Pixel targetColor,
               Pixel newColor) {
  // vérifier les bordures
  if (x < 0 || y < 0 || x >= (int)img.largeur || y >= (int)img.hauteur)
    return;

  Pixel& current = img.pixels[y][x];

  // arrêter si le Pixel courant n'est pas de la même couleur que targetColor
  // ou s'il est déjà de la même couleur que newColor
  if (!isSameColor(current, targetColor) || isSameColor(current, newColor))
    return;

  // remplir le Pixel courrant
  current = newColor;

  // appels récursifs dans la direction des 4 points cardinaux
  floodFill(img, x + 1, y, targetColor, newColor);
  floodFill(img, x - 1, y, targetColor, newColor);
  floodFill(img, x, y + 1, targetColor, newColor);
  floodFill(img, x, y - 1, targetColor, newColor);
}
```

⚠️ Une implémentation itérative de la fonction `floodFill` entraînera une note de `0` pour cette fonction.

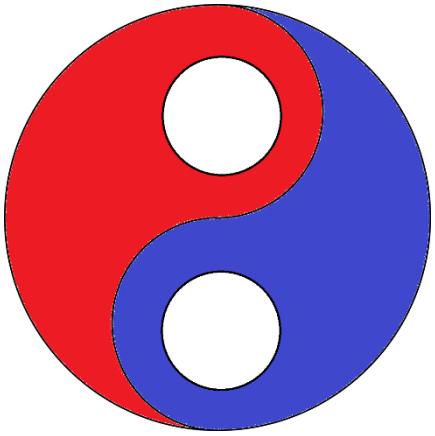
Étant une fonction récursive, `floodFill` a besoin d'un point d'entrée. Voici l'implémentation (**déjà codée**) du point d'entrée : `applyFloodFill`. C'est cette dernière qui est appelée dans le `main()`.

```
void applyFloodFill(Image *img, int startX, int startY, Pixel newColor) {
  if (startX < 0 ||
      startY < 0 ||
      startX >= (int)img->largeur ||
      startY >= (int)img->hauteur)
  {
      return;
  }

  Pixel targetColor = img->pixels[startY][startX];

  if (isSameColor(targetColor, newColor))
  {
    return;
  }

  floodFill(img, startX, startY, targetColor, newColor);
}
```

Exemple de résultat attendu



Avant



Après

Source de l'animation : https://en.wikipedia.org/wiki/Flood_fill