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
Hiver 2026

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. Le thème cette année est : rétro. Vous serez portés à implémenter un filtre de type CRT. Ensuite, le traçage de la fractale de Sierpinski 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 : dimanche 15 mars 2026, 23:00

Directives particulières

- Vous devez chercher une ou des images en format `.bmp .png, .jpg ou .jpeg`
- Ajouter les images dans le dossier `/images`
- Assurez d'indiquer la source des images dans l'entête de la fonction `main()`
- Ajuster le chemin des images dans la fonction `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).

Commandes disponibles :

Le projet fournit plusieurs commandes Make pour faciliter la compilation, l'exécution, les tests et la génération du livrable :

- `make` : compile le projet en générant l'exécutable principal.
- `make run` : compile le projet (si nécessaire) puis exécute l'application.
- `make test` : lance la suite de tests prévue pour vérifier le bon fonctionnement des fonctions et filtres implémentés.
- `make remise` : crée un fichier zip contenant l'ensemble des fichiers nécessaires pour la remise du projet, prêt à être soumis.

Instruction x86-IA32 intéressante pour ce TP (facultative) :

`movzx` : copier le contenu du registre de départ vers le registre de destination de plus grande taille, est étendre le reste avec des 0.

Ex :

```
movb $0xCF, %al # %al est de taille 1 octet et vaut 0xCF
movzx %al, %ebx # %ebx est de taille 4 octets et vaut 0x000000CF
```

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 r;
  uint8_t g;
  uint8_t b;
  uint8_t a;
};
```

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. Les pixels ne sont donc pas tous contigus en mémoire.

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]`.
- Les pixels ne sont pas tous contigus en mémoire.
- Les **adresses** des rangées sont contiguës en mémoire.
- Pour une rangée donnée, les pixels sont contigus en mémoire.

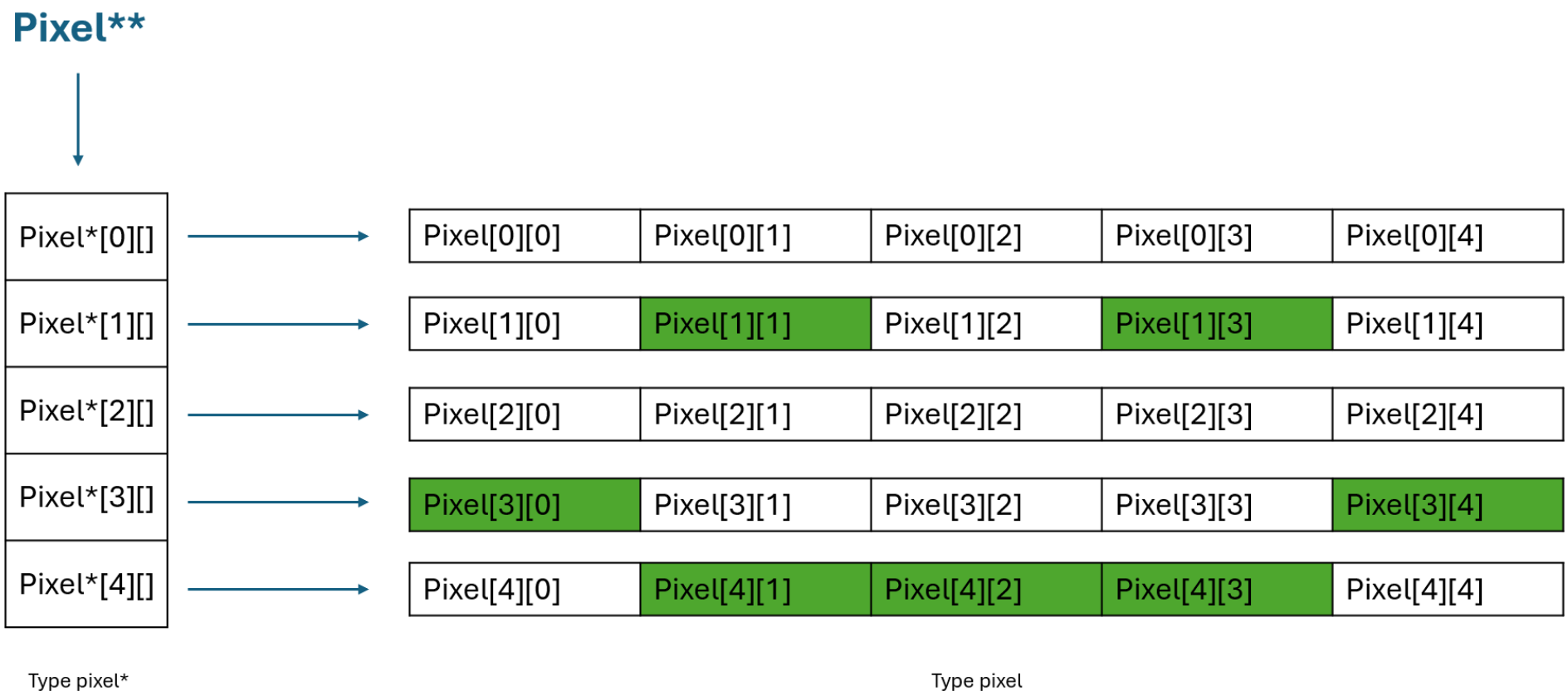


Figure 1 : Représentation du tableau Pixel** en mémoire

Filtre CRT

applyScanline

Fichier : `src/crt/apply_scanline.s`

Signature :

```
void applyScanline(Pixel& p, int percent)
```

Paramètres :

`p` : la référence vers le pixel à modifier (sur place)

`percent` : facteur d'assombrissement (entier entre 0 et 100)

Description : Cette fonction applique un facteur d'assombrissement à un pixel en **multipliant chacune de ses composantes RGB par un pourcentage donné.** Le canal alpha est ignoré.

$$nouvelle\_valeur = valeur\_originale \times percent / 100$$

Cette fonction sera utilisée plus tard dans la fonction `crtFilter`.

applyPhosphor

Fichier : `src/crt/apply_phosphor.s`

Signature :

```
void applyPhosphor(Pixel& p, int subpixel)
```

Paramètres :

`p` : la référence vers le pixel à modifier (sur place)

`subpixel` : indice du pixel dominant

Description : Le paramètre subpixel détermine quelle composante reste dominante :

- si `subpixel == 0` → le rouge est conservé, le vert et le bleu sont réduits à 70 % de leur valeur initiale.
- si `subpixel == 1`→ le vert est conservé, le rouge et le bleu sont réduits à 70 % de leur valeur initiale.
- sinon → le bleu est conservé, le rouge et le vert sont réduits à 70 % de leur valeur initiale.

Encore une fois, puisqu'on travaille avec des divisions entières, la réduction se fait avec la formule suivante :

$$nouvelle\_valeur = valeur\_originale \times 70 / 100$$

Cette fonction sera utilisée plus tard dans la fonction `crtFilter`.

crtFilter

Fichier : `src/crt/crt_filter.s`

Signature :

```
void crtFilter(Image& img, int scanlineSpacing)
```

Paramètres :

`img` : la référence vers l'image à modifier (sur place)

`scanlineSpacing` : espacement entre les lignes que l'on va dessiner sur l'image pour l'effet CRT

Description : Cette fonction applique un filtre global à une image afin de reproduire l'apparence d'un ancien écran CRT. Elle combine les deux fonctions précédentes.

Il faut parcourir TOUS les pixels et appliquer les traitements suivants:

1. Appeler `applyScanline()`
    - Si la ligne `y` est un multiple de `scanlineSpacing` on applique un assombrissement de **`60 %`**.

2. Appler `applyPhosphor()`
    - Le paramètre `subpixel` est déterminé par la position horizontale du pixel : **`x % 3`**

Exemple de résultat attendu

Avant

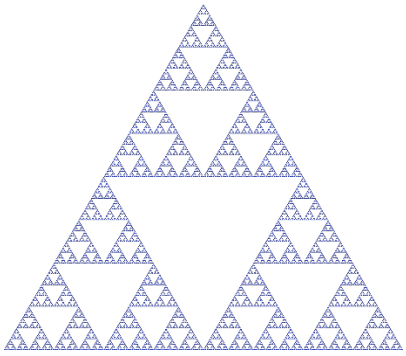
Après (avec un scanlineSpacing de `2`)

Source de l'image : https://pixelatedarcade.s3.us-east-005.dream.io/screenshots/2506/2624/screenshot_e50-the-legend-of-zelda-exploring-hyrule.png

Triangle de Sierpiński

Fichier : `src/sierpinski/sierpinski.s`

Cette fonction dessine un triangle de Sierpinski directement dans une image en utilisant une approche **récursive**. Le triangle de Sierpinski est une figure fractale caractérisée par un motif répétitif de triangles imbriqués.



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

```
void sierpinskiImage(uint32_t x, uint32_t y, uint32_t size, Image& img, Pixel
color) {
    // vérifier les bornes
    if (x >= img.largeur || y >= img.hauteur) return;

    // Cas de base: dessiner un seul pixel
    if (size == 1) {
        img.pixels[y][x] = color;
        return;
    }

    uint32_t half = size / 2;

    // Triangle en bas à gauche
    sierpinskiImage(x, y + half, half, img, color);
    // Triangle en bas à droite
    sierpinskiImage(x + half, y + half, half, img, color);
    // Triangle du haut
    sierpinskiImage(x + half / 2, y, half, img, color);
}
```

Source de l'image : https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle#/media/File:Sierpinski_triangle.svg

L’algorithme fonctionne mieux avec des tailles **puissances de 2.**

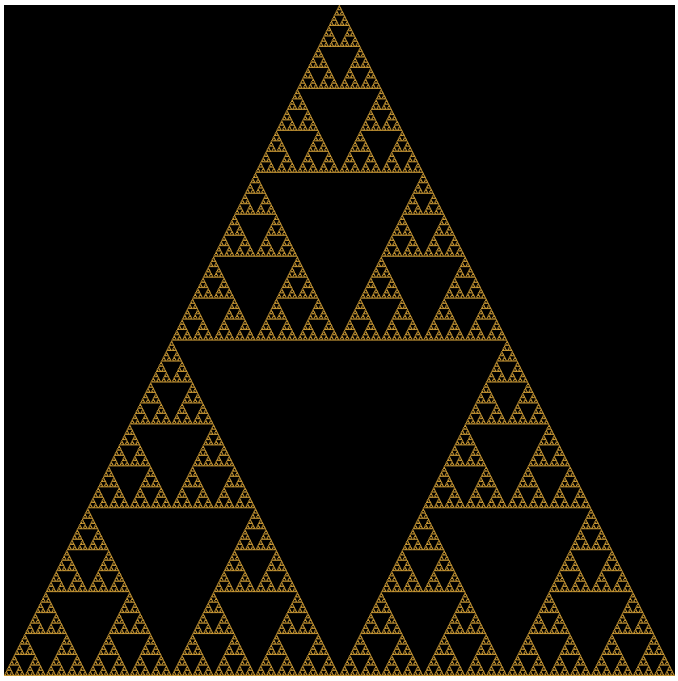
L’appel de la fonction dans le `main` sera ainsi :

```
sierpinskiImage(0, 0, 1024, img, color);
```

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

Exemple de résultat attendu

Exemple avec une taille de 1024.



main

Fichier : `src/main.s`

1. Filtre CRT

- **Étapes :**
    1. Charger l'image source `inputCrt` avec `loadImage()`.
        - L'image doit être une variable locale allouée sur la pile.
        - `loadImage()` prend une référence (adresse) vers l'image et une chaine de caractère (pointeur).
    2. Appliquer le filtre `crtFilter()` sur l'image chargée.
        - Le paramètre scanlineSpacing est représente l'espacement entre les lignes qu'on va dessiner.
    3. Sauvegarder l'image résultante dans `outputCrt` avec `saveImage()`.
        - `saveImage()` prend une référence (adresse) vers l'image et une chaine de caractère (pointeur).
    4. Librérer la mémoire de l'image avec `freeImage()`.
        - `freeImage()` prend une référence (adresse) vers l'image.

---

2. Triangle de Sierpinski

- **Étapes :**
    1. Créer une nouvelle image vide avec `createImage()`.
        - La taille de l'image doit être une **puissance de 2** (ex. 1024) pour que la récursion fonctionne correctement.
        - Puisque `createImage()` retourne une struct Image, il faut d'abord allouer de l'espace sur la pile pour l'image, puit push l'adresse de cet espace comme 3e paramètre avant de call la fonction.
    2. Dessiner le triangle de Sierpinski en appelant `sierpinskiImage()` de manière récursive sur cette image.
        - `Ex : sierpinskiImage(0, 0, 1024, img, {227, 171, 59, 255});`
    3. Sauvegarder l'image finale dans `outputSierpinski` avec `saveImage()`.
    4. Librérer la mémoire de l'image avec `freeImage()`.
        - `freeImage()` prend une référence (adresse) vers l'image.