Ditherpunk

Membre du groupe

- Lucidor Léo
- Blandeau Erwan
- Pilet Colin

Partie - 1

Question 2 - Pour ouvrir une image depuis un fichier, on utilise … On obtient un DynamicImage, à quoi correspond ce type?

- Le type DynamicImage est une structure qui peut contenir différentes - représentations d'une image en fonction de son format (par exemple, RGB, RGBA, etc.).
- Pour obtenir une image en mode rgb8, il faut utiliser la méthode .to_rgb8() qui convertit l'image en une image avec 3 canaux (R, G, B), chacun étant un u8.

Pour utiliser le mode seuil avec une image d'entrée et une sortie spécifiée, il faut exécuter cette commande :

```
cargo run -- img/IUT.jpg img/IUT_OUT.png seuil
```

On peux voir ci-dessous que la sortie des deux image est identique.

IUT.jpg	**IUT_OUT.png**
	

Question 3 - Sauver l’image obtenue au format png. Que se passe-t-il si l’image de départ avait un canal alpha?

- Si l'image d'entrée a un canal "alpha", une erreur survien lors du traitement : **Error: Decoding(DecodingError { format: Exact(Png), underlying: Some(Format(FormatError { inner: InvalidSignature })) })**
- Cette erreur se déclenche à cause de ce canal "alpha" au moment d'executer la **fonction "to_rgb8()" lors du traitement**. On peut l'expliquer par le fait que cette fonction "to_rgb8()" essaie de convertir une image RGBA, en RGB8. Le RGB8 ne contenant pas de canal alpha, le format de lecture est en erreur.

Question 4 - Afficher dans le terminal la couleur du pixel (32, 52) de l'image de votre choix

- Pour afficher la couleur du pixel (32, 52) d'une image. Il nous faut utiliser la fonction "get_pixel()" avec les coordonnées "32, 52".
- Il nous faut ensuite exectuter "println!("Pixel (32, 52) : {:?}", pixel);" pour afficher la couleur en RGB du Pixel.

commande : `cargo run -- img/IUT.jpg img/IUT_OUT.png seuil`

```
![](./rapport/q4/Capture%20d'écran_2025-01-20_18-13-28.png)
```

Question 5 - Passer un pixel sur deux d'une image en blanc. Est-ce que l'image obtenue est reconnaissable?

Passer un pixel sur deux d'une image en blanc : Est-ce que l'image obtenue est reconnaissable ?

Lorsqu'on passe un pixel sur deux d'une image en blanc, l'image résultante reste souvent reconnaissable, bien que son apparence soit altérée. La perception humaine est particulièrement douée pour interpréter des motifs et reconstruire des formes même lorsqu'une partie de l'information visuelle est absente ou modifiée.

Cependant, plusieurs facteurs influencent la reconnaissance :

Passer un pixel sur deux d'une image en blanc : Est-ce que l'image obtenue est reconnaissable ?

- Pour modifier l'image, j'ai utilisé un simple algorithme qui parcourt chaque pixel et passe un pixel sur deux en blanc (`rgb(255, 255, 255)`). Voici le code utilisé :

```
let mut pixelblanc = false;

for (x, y, pixel) in rgb_image.enumerate_pixels_mut() {
    if pixelblanc {
        // passer le pixel en blanc en utilisant le rgb(255, 255, 255)
        pixel.0[0] = 255;
        pixel.0[1] = 255;
        pixel.0[2] = 255;
        pixelblanc = false;
    } else {
        pixelblanc = true;
    }
}
```

- Grâce à la haute résolution de l'image utilisé, l'image obtenue reste bien reconnaissable malgré cette transformation.

IUT.jpg	IUT_OUT.png
	

Question 6 - Comment récupérer la luminosité d'un pixel?

- La luminosité d'un pixel peut être estimée en appliquant une formule pondérée, qui tient compte de la sensibilité humaine aux différentes couleurs : **Luminosité=0.299×R+0.587×G+0.114×B**
- Cette formule donne un nombre à virgule flottante représentant la luminosité.

Voici la fonction crée afin de récupérer la luminosité d'un pixel :

```
fn luminosity_of_pixel(pixel: Rgb<u8>) -> f32 {
    let (r, g, b) = (pixel[0], pixel[1], pixel[2]);
    0.299 * r as f32 + 0.587 * g as f32 + 0.114 * b as f32
}
```

- pixel[0], pixel[1], et pixel[2] représentent les composantes Rouge, Vert et Bleu du pixel.
- Chaque composante est convertie en f32 pour effectuer le calcul avec les coefficients pondérés.
- La valeur de la luminosité renvoyer sera :
    - Valeur minimale : 0.00.0 (luminosité d'un pixel complètement noir).
    - Valeur maximale : 255.0255.0 (luminosité d'un pixel complètement blanc).

Question 7 - Implémenter le traitement

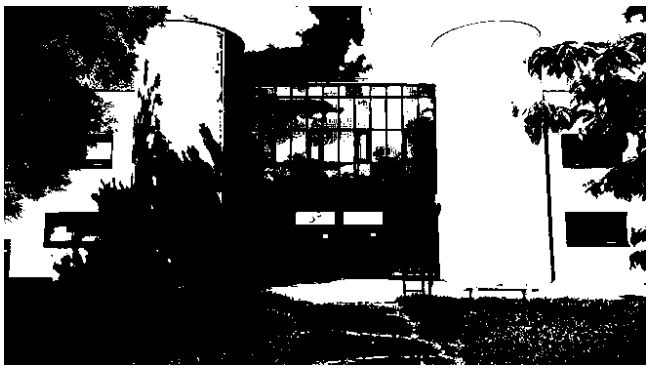
- Si la luminosité dépasse 50% de son maximum (127.5 sur une échelle de 0 à 255), le pixel sera remplacé par blanc → R=G=B=255R=G=B=255 → Rgb([255, 255, 255])
- Sinon, il sera remplacé par noir → R=G=B=0R=G=B=0 → Rgb([0, 0, 0])

Voici le code de la fonction crée passer une image en monochrome :

```
fn to_monochrome(image: &mut RgbImage) {
    for y in 0..image.height() {
        for x in 0..image.width() {
            let pixel = image.get_pixel(x, y);
            let luminosity = luminosity_of_pixel(*pixel);
```

```
            // Remplacement par blanc ou noir en fonction de la luminosité
            if luminosity > 127.5 {
                image.put_pixel(x, y, WHITE);
            } else {
                image.put_pixel(x, y, BLACK);
            }
        }
    }
}
```

- La fonction **to_monochrome** convertit une image couleur en une image monochrome (noir et blanc) en remplaçant chaque pixel par du blanc ou du noir en fonction de sa luminosité.

IUT.jpg	IUT_OUT.png
	

Question 8 - Permettre à l'utilisateurice de remplacer "noir" et "blanc" par une paire de couleurs au choix

- Pour permettre à l'utilisateur ou à l'utilisatrice de remplacer le "noir" et le "blanc" par une paire de couleurs personnalisées, nous avons ajoutée deux paramètres couleurs à l'appel de la fonction.

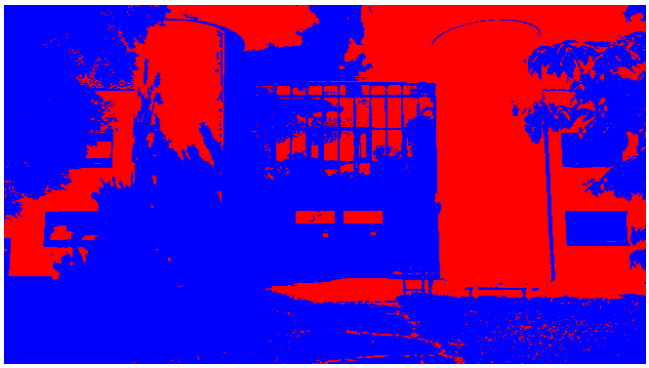
Voici le code la fonction crée :

```
fn to_pair_colors(image: &mut RgbImage, color_low: Rgb<u8>, color_high:
Rgb<u8>) {
    for y in 0..image.height() {
        for x in 0..image.width() {
            let pixel = image.get_pixel(x, y);
            let luminosity = luminosity_of_pixel(*pixel);

            // Remplacement par `color_high` ou `color_low` en fonction de
la luminosité
            if luminosity > 127.5 {
                image.put_pixel(x, y, color_high);
            } else {
                image.put_pixel(x, y, color_low);
            }
        }
    }
```

```
}
```

- Cette fonction remplace les pixels de l'image par une paire de couleurs en fonction de la luminosité des pixels. Les utilisateurs peuvent choisir librement les couleurs pour les zones sombres et claires.

IUT.jpg	IUT_OUT.png
	

Question 9 - Comment calculer la distance entre deux couleurs? Indiquer dans le README la méthode de calcul choisie

- Pourquoi choisir la distance euclidienne ?
    - Simplicité : La distance euclidienne est facile à comprendre et à implémenter.
    - Performance : Calculer cette distance est rapide et suffisant pour la plupart des applications en RGB.
    - Applications courantes : Elle est souvent utilisée dans des algorithmes de clustering (comme K-means) et dans les comparaisons simples de couleurs.

Voici la fonction crée avec le choix de la distance euclidienne :

```
fn color_distance(c1: Rgb<u8>, c2: Rgb<u8>) -> f32 {
    let r_diff = c1[0] as f32 - c2[0] as f32;
    let g_diff = c1[1] as f32 - c2[1] as f32;
    let b_diff = c1[2] as f32 - c2[2] as f32;

    // Calcul de la distance euclidienne
    ((r_diff.powi(2) + g_diff.powi(2) + b_diff.powi(2)).sqrt())
}
```

- La méthode utilisée pour calculer la distance entre deux couleurs est basée sur la distance euclidienne dans l'espace RGB. Voici la formule employée : distance= √{(R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2}

Distance minimale (0) :

```
La distance entre deux couleurs est 0 lorsque les couleurs sont identiques.
Cela signifie que les composantes RGB de chaque couleur sont identiques,
donc R1=R2R1=R2, G1=G2G1=G2, et B1=B2B1=B2.
Par exemple, la distance entre rouge (255,0,0)(255,0,0) et rouge (255,0,0)
(255,0,0) est 0.
```

Distance maximale ($\sqrt{(255^2 + 255^2 + 255^2)}$) :

```
La distance maximale se produit lorsque les couleurs sont complètement
opposées dans l'espace RGB. Cela se produit, par exemple, lorsque l'une des
couleurs est complètement noire (0,0,0)(0,0,0) et l'autre complètement
blanche (255,255,255)(255,255,255), ou si les composantes de l'une des
couleurs sont maximales (255) et minimales (0) dans toutes les composantes
RGB.
Pour calculer la distance maximale, on suppose que chaque composante RR,
GG, et BB varie de 0 à 255. La distance maximale entre deux couleurs sera
donc :

distance maximale=(255−0)2+(255−0)2+(255−0)2
distance maximale=2552+2552+2552=3×2552≈3×65025≈195075≈441.67
```

Question 10 - Implémenter le traitement

1. Définition de la palette

La palette utilisée dans ce cas est un ensemble de couleurs prédéfinies. Chaque couleur est représentée par une valeur RGB (Rouge, Vert, Bleu). Voici un exemple de palette avec 9 couleurs :

```rust
const PALETTE: [Rgb<u8>; 8] = [
    BLACK, WHITE, BLUE, RED, GREEN, YELLOW, MAGENTA, CYAN
];
```

2. Calculer la distance entre un pixel et chaque couleur de la palette

```
Pour déterminer quelle couleur de la palette est la plus proche d’un pixel,
nous utilisons la distance euclidienne dans l’espace RGB. Nous utilisons
donc la fonction **"color_distance"** précèdement crée.
```

3. Remplacer le pixel par la couleur la plus proche dans la palette

```
Chaque pixel de l’image est comparé à toutes les couleurs de la palette
pour déterminer laquelle est la plus proche en termes de distance. Une fois
cette couleur identifiée, elle remplace la couleur originale du pixel.
```

Voici les fonctions créent pour appliquer cette transformation sur une image :

```rust
fn to_palette(image: &mut RgbImage, palette: &[Rgb<u8>]) {
    for y in 0..image.height() {
        for x in 0..image.width() {
            let pixel = image.get_pixel(x, y);
            let closest_color = find_closest_color(*pixel, palette);
            image.put_pixel(x, y, closest_color);
        }
    }
}

fn find_closest_color(pixel: Rgb<u8>, palette: &[Rgb<u8>]) -> Rgb<u8> {
    let mut min_distance = f32::MAX;
    let mut closest_color = palette[0];

    for &color in palette {
        let distance = color_distance(pixel, color);
        if distance < min_distance {
            min_distance = distance;
            closest_color = color;
        }
    }

    closest_color
}
```

Lorsqu'on applique cette méthode, chaque pixel de l’image originale est remplacé par la couleur de la palette qui lui est la plus proche. Par exemple :

```
Un gris clair dans l’image peut être remplacé par le GREY (127,127,127)
(127,127,127).
Une teinte bleu ciel peut être remplacée par le CYAN (0,255,255)
(0,255,255).
```

IUT.jpg	IUT_OUT.png

IUT.jpg	IUT_OUT.png
	

Question 11 - Votre application doit se comporter correctement si on donne une palette vide. Vous expliquerez dans votre README le choix que vous avez fait dans ce cas

Lorsque l'application reçoit une palette vide, elle ne modifie pas l'image et affiche un message d'information dans la console :

```
La palette est vide. Aucun traitement n'est appliqué.
```

Cela garantit que l'application reste stable et n'essaie pas de comparer des pixels à une palette inexistante.

Justification du choix

1. **Robustesse :** En ne modifiant rien, nous évitons des erreurs ou des comportements inattendus (comme une tentative d'accès à un élément inexistant dans la palette).
2. **Clarté pour l'utilisateur :** L'utilisateur est informé que la palette est vide grâce à un message clair.
3. **Non-destruction :** Il est préférable de ne rien faire que de produire des résultats inattendus (par exemple, remplir l'image avec une couleur par défaut).

Question 12 - Implémenter le tramage aléatoire des images

1. Principe du tramage aléatoire :

```
Pour chaque pixel, un seuil aléatoire est généré entre 0 et 255.
La luminosité du pixel est comparée à ce seuil :
    Si la luminosité >> seuil aléatoire, le pixel devient blanc.
    Sinon, le pixel devient noir.
```

2. Utilisation de rand::Rng:

```
Le générateur aléatoire rand::thread_rng() est utilisé pour produire un
seuil différent pour chaque pixel.
```

```
La plage [0.0,255.0][0.0,255.0] garantit que le seuil aléatoire est
comparable à la luminosité calculée, qui est également dans cet intervalle.
```

3. Luminosité d'un pixel :

```
La fonction luminosity_of_pixel(pixel) (définie précédemment) calcule la
luminosité pondérée selon les composantes RGB.
```

Avant tramage

```
L'image est composée de couleurs ou de niveaux de gris.
```

Après tramage aléatoire

```
L'image est composée uniquement de noir et de blanc, mais avec des motifs
aléatoires qui ajoutent une impression de nuances.
```

Voici la fonction crée pour implémenter le tramage aléatoire des images :

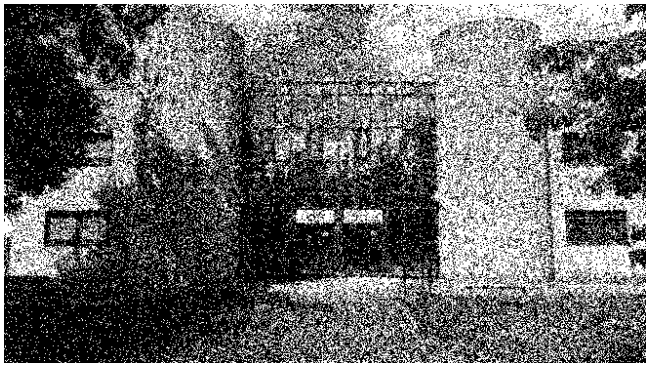
```rust
fn random_dithering(image: &mut RgbImage) {
    let mut rng = rand::thread_rng(); // Générateur de nombres aléatoires

    for y in 0..image.height() {
        for x in 0..image.width() {
            let pixel = image.get_pixel(x, y);
            let luminosity = luminosity_of_pixel(*pixel);

            // Générer un seuil aléatoire entre 0 et 255
            let random_threshold = rng.gen_range(0.0..255.0);

            // Déterminer la nouvelle couleur en fonction de la luminosité
et du seuil
            if luminosity > random_threshold {
                image.put_pixel(x, y, WHITE);
            } else {
                image.put_pixel(x, y, BLACK);
            }
        }
    }
}
```

IUT.jpg	IUT_OUT.png

IUT.jpg	IUT_OUT.png
	

Question 13 - Déterminer $B3$

Définition de la matrice de Bayer

```
La matrice de Bayer d'ordre 0 est donnée comme :

B0=[0]

La matrice de Bayer d'ordre n+1 est obtenue par :
Bn+1=(1/4)*[4Bn     4Bn+3Un
            4Bn+2Un 4Bn+Un]


où Un est une matrice de taille 2n×2n dont tous les éléments valent 1
```

Calcul pas à pas

La matrice B3 peut être calculée de manière récursive à partir de B2. Voici les étapes :

```
B2=(1/16)*[0   8  2 10
           12  4 14  6
           3  11  1  9
           15  7 13  5]
```

U2 est une matrice 4×4 de 1 :

```
U2=[1 1 1 1
    1 1 1 1
    1 1 1 1
    1 1 1 1]
```

Calcul de 4B2,4B2+3U2,4B2+2U2,4B2+U2

Chaque élément de B2 est multiplié par 4, puis on ajoute 3,2, ou 1 selon le cas pour les blocs.

```
4B2 :

4B2=[0  32  8 40
     48 16 56 24
     12 44  4 36
     60 28 52 20]

4B2+3U2 :

4B2+3U2=[3  35 11 43
         51 19 59 27
         15 47  7 39
         63 31 55 23]

4B2+2U2 :

4B2+2U2=[2  34 10 42
         50 18 58 26
         14 46  6 38
         62 30 54 22]

4B2+1U2 :

4B2+U2=[1  33  9 41
        49 17 57 25
        13 45 5  37
        61 29 53 21]
```

Assemblage de B3

On assemble les blocs pour obtenir B3 :

```
B3=(1/64)*[4B2      4B2+3U2
           4B2+2U2  4B2+U2]
```

Ce qui donne directement :

```
B3=[0  32  8 40  2 34 10 42
    48 16 56 24 50 18 58 26
    12 44  4 36 14 46  6 38
    60 28 52 20 62 30 54 22
     3 35 11 43  1 33  9 41
    51 19 59 27 49 17 57 25
    15 47  7 39 13 45  5 37
    63 31 55 23 61 29 53 21]
```

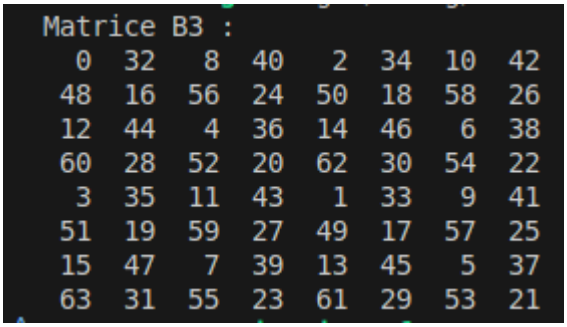
Nous avons pu créée la fonction generate_bayer_matrix génère récursivement une matrice de Bayer d'ordre nn, utilisée pour le "ordered dithering".

Elle commence avec une matrice de base B0=[[0]]. À chaque ordre supérieur, la matrice est agrandie en divisant l'espace en 4 quadrants, chacun calculé selon la formule donnée :

```
4×Bn4×Bn
4×Bn+2×Un
4×Bn+3×Un
4×Bn+Un
```

où Un est une matrice remplie de 1.

La fonction combine ces quadrants dans une nouvelle matrice de taille 2n×2n, en suivant une approche récursive. Cela permet de construire des matrices d'ordre arbitraire de manière efficace, tout en respectant la définition mathématique.



Question 14 - Quel type de données utiliser pour représenter la matrice de Bayer? Comment créer une matrice de Bayer d’ordre arbitraire?

Pour représenter la matrice de Bayer, un tableau 2D (comme un vecteur de vecteurs de 32bits en Rust) est adapté. Voici pourquoi :

```
Structure simple : La matrice est carrée (2n×2n) et ses éléments sont des
entiers.
Facilité d'accès : Un tableau 2D permet un accès direct à M[i][j]
```

Pour générer une matrice d'ordre arbitraire, une fonction récursive est idéale. Voici l'algorithme :

```
Cas de base : B0=[[0]]
Construction :
    Calculer 4Bn, 4Bn+3, 4Bn+2, 4Bn+1.
    Assembler les blocs dans une nouvelle matrice
```

Question 15 - Implémenter le tramage par matrice de Bayer

```
fn bayer_dithering(image: &mut RgbImage, bayer_matrix: &[Vec<u32>]) {
    let matrix_size = bayer_matrix.len() as u32;

    for y in 0..image.height() {
        for x in 0..image.width() {
            let pixel = image.get_pixel(x, y);
            let luminosity = luminosity_of_pixel(*pixel);

            // Récupérer le seuil de la matrice (en répétant la matrice)
            let threshold = bayer_matrix[(y % matrix_size) as usize][(x %
matrix_size) as usize] as f32;

            // Appliquer le seuil (normalisé à 255)
            if luminosity > (threshold / (matrix_size * matrix_size) as
f32) * 255.0 {
                image.put_pixel(x, y, WHITE);
            } else {
                image.put_pixel(x, y, BLACK);
            }
        }
    }
}
```

La fonction "bayer_dithering" permet d'appliquer un tramage ordonné à une image en utilisant une matrice de Bayer pour déterminer les seuils de conversion de chaque pixel en noir ou blanc. Voici les étapes principales :

Taille de la matrice de Bayer :

```
let matrix_size = bayer_matrix.len() as u32;
```

```
Cette variable récupère la taille de la matrice de Bayer (assumée carrée).
Elle est utilisée pour gérer la répétition de la matrice sur l'image.
```

Parcours de l'image pixel par pixel :

```
for y in 0..image.height() {
    for x in 0..image.width() {
```

La fonction parcourt chaque pixel de l'image pour appliquer la transformation.

Calcul de la luminosité du pixel :

```
let luminosity = luminosity_of_pixel(*pixel);
```

La luminosité est extraite pour déterminer si le pixel doit être remplacé par du noir ou du blanc.

Répétition de la matrice de Bayer :

```
let threshold = bayer_matrix[(y % matrix_size) as usize][(x % matrix_size) as usize] as f32;
```

La matrice de Bayer est répétée sur l'image à l'aide des opérations modulo (%). Cela permet de couvrir une image de taille arbitraire en "mosaïquant" la matrice de Bayer.

Normalisation du seuil :

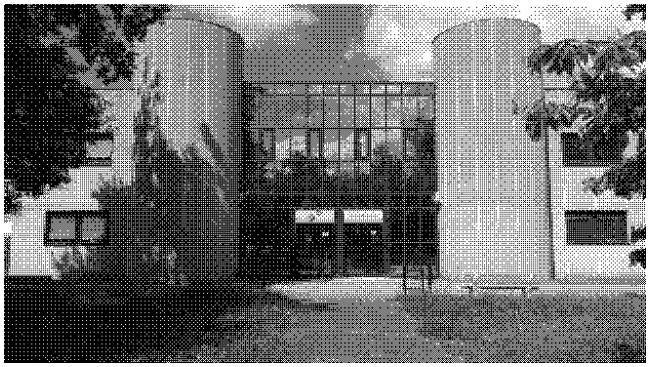
```
if luminosity > (threshold / (matrix_size * matrix_size) as f32) * 255.0 {
```

Les valeurs dans la matrice de Bayer sont normalisées par rapport à la plage de valeurs des pixels (0 à 255). Cela garantit que les seuils de Bayer, initialement entre 0 et 2n−1, correspondent à des seuils dans l'échelle de la luminosité.

Application du tramage :

```
if luminosity > normalized_threshold {
    image.put_pixel(x, y, WHITE);
} else {
    image.put_pixel(x, y, BLACK);
}
```

```
Le pixel est remplacé par du blanc si sa luminosité dépasse le seuil
correspondant de la matrice de Bayer, sinon par du noir. Cela crée un effet
de tramage visuellement ordonné.
```

IUT.jpg	IUT_OUT.png
	

Question 16 - Implémenter un mécanisme de diffusion d’erreur

Cette fonction applique un tramage en noir et blanc avec diffusion d'erreur selon une matrice définie. Voici les étapes avec des extraits de code correspondants :

```rust
fn error_diffusion(image: &mut RgbImage) {
    let width = image.width() as i32;
    let height = image.height() as i32;

    // Convertir l'image en niveaux de gris
    let mut grayscale_image: Vec<Vec<f32>> = vec![vec![0.0; width as
usize]; height as usize];
    for y in 0..height {
        for x in 0..width {
            let pixel = image.get_pixel(x as u32, y as u32);
            grayscale_image[y as usize][x as usize] =
luminosity_of_pixel(*pixel) / 255.0; // Normaliser à [0, 1]
        }
    }

    for y in 0..height {
        for x in 0..width {
            // Récupérer la valeur actuelle
            let old_value = grayscale_image[y as usize][x as usize];

            // Quantification : remplace par noir (0.0) ou blanc (1.0)
            let new_value = if old_value > 0.5 { 1.0 } else { 0.0 };

            // Appliquer la nouvelle valeur au pixel
            let color = if new_value == 1.0 { WHITE } else { BLACK };
            image.put_pixel(x as u32, y as u32, color);
```

```
            // Calculer l'erreur
            let error = old_value - new_value;

            // Diffuser l'erreur aux pixels voisins
            if x + 1 < width {
                grayscale_image[y as usize][(x + 1) as usize] += error *
0.5; // Pixel à droite
            }
            if y + 1 < height {
                grayscale_image[(y + 1) as usize][x as usize] += error *
0.5; // Pixel en dessous
            }
        }
    }
}
```

Étape 1 : Conversion en niveaux de gris

Avant de commencer le traitement, l'image est convertie en niveaux de gris. Pour cela, chaque pixel est analysé, et sa luminosité (valeur entre 0 et 1) est calculée et stockée dans une matrice 2D grayscale_image

```
let mut grayscale_image: Vec<Vec<f32>> = vec![vec![0.0; width as usize];
height as usize];
for y in 0..height {
    for x in 0..width {
        let pixel = image.get_pixel(x, y);
        grayscale_image[y as usize][x as usize] =
luminosity_of_pixel(*pixel) / 255.0;
    }
}
```

Étape 2 : Traitement des pixels un par un

Chaque pixel est parcouru dans un ordre spécifique (ligne par ligne). La valeur de luminosité du pixel est extraite et utilisée pour déterminer s'il sera transformé en noir ou blanc, selon un seuil de 0.5

```
let old_value = grayscale_image[y as usize][x as usize];
let new_value = if old_value > 0.5 { 1.0 } else { 0.0 };
```

Étape 3 : Calcul de l'erreur de quantification

```
L'erreur entre la luminosité originale et la valeur quantifiée (noir ou
blanc) est calculée. Cette erreur sera ensuite diffusée aux pixels voisins
```

```
let error = old_value - new_value;
```

Étape 4 : Diffusion de l'erreur aux voisins

Selon la matrice donnée (* 0.5 / 0.5 0), l'erreur est répartie sur les pixels adjacents :

```
- 50% de l'erreur est ajoutée au pixel à droite
- 50% de l'erreur est ajoutée au pixel en dessous
- Des vérifications assurent que ces pixels voisins sont dans les limites
de l'image
```

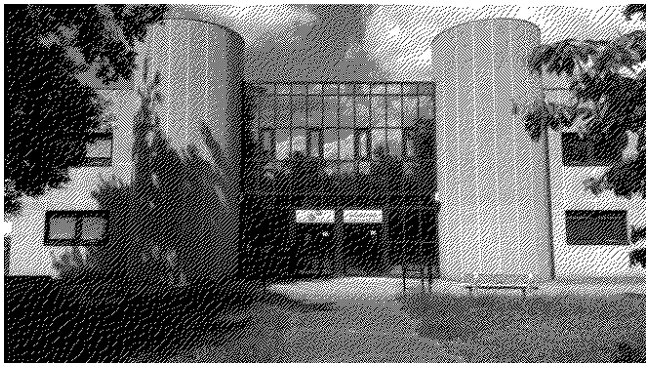
```
if x + 1 < width {
    grayscale_image[y as usize][(x + 1) as usize] += 0.5 * error;
}
if y + 1 < height {
    grayscale_image[(y + 1) as usize][x as usize] += 0.5 * error;
}
```

Étape 5 : Mise à jour de l'image

```
Le pixel est finalement remplacé par du noir (BLACK) ou du blanc (WHITE)
dans l'image de sortie
```

```
if new_value == 1.0 {
    image.put_pixel(x, y, WHITE);
} else {
    image.put_pixel(x, y, BLACK);
}
```

IUT.jpg	IUT_OUT.png

IUT.jpg	IUT_OUT.png
	

Question 17 - Pour une palette de couleurs comme dans la partie 3, expliquer dans votre README comment vous représentez l'erreur commise à chaque pixel, comment vous la diffusez

Lorsque nous utilisons une palette de couleurs au lieu d'une simple conversion en noir et blanc, la gestion de l'erreur devient plus complexe. L'erreur commise lors de l'approximation de la couleur d'un pixel est alors représentée par un vecteur contenant les différences pour chacune des trois composantes de couleur : rouge (R), vert (G) et bleu (B).

Étape 1 : Représentation de l'erreur

L'erreur pour un pixel est définie comme la différence entre sa couleur d'origine et la couleur choisie dans la palette. Cela signifie qu'au lieu d'une simple valeur de luminosité, l'erreur est un vecteur contenant trois valeurs : l'erreur pour le rouge, le vert et le bleu. Ces erreurs traduisent combien la couleur finale du pixel s'éloigne de la couleur réelle.

Étape 2 : Diffusion de l'erreur

Une fois l'erreur calculée, elle est répartie entre les pixels voisins non traités. Cette diffusion est effectuée à l'aide d'une matrice qui détermine quelle proportion de l'erreur est attribuée à chaque voisin. Par

exemple, si on utilise une matrice simple :

```
[ *  0.50]
[0.5 0.50]
```

Cela signifie que 50 % de l'erreur est transférée au pixel situé à droite et 50 % au pixel situé en dessous. Chaque composante de l'erreur (R, G, B) est propagée de manière indépendante selon cette matrice.

Étape 3 : Correction des pixels

Avant de quantifier la couleur d'un pixel suivant la palette, on prend en compte l'erreur accumulée provenant des pixels précédents. Cela permet d'ajuster la couleur du pixel pour compenser les approximations faites plus tôt. Cette méthode assure une meilleure continuité entre les couleurs et réduit les artefacts visuels.

Question 18 - Implémenter la diffusion d'erreur pour la palettisation d'images

Nous avons implémentaté la fonction de diffusion d'erreur pour la palettisation d'images, en tenant compte des trois composantes de couleur (R, G, B). La fonction utilise une matrice simple pour répartir l'erreur entre les pixels voisins :

```
fn error_diffusion_palette(image: &mut RgbaImage, palette: &[Rgba<u8>]) {
    let width = image.width() as i32;
    let height = image.height() as i32;

    // Parcours des pixels
    for y in 0..height {
        for x in 0..width {
            let pixel = image.get_pixel(x as u32, y as u32);
            let original_color = *pixel;

            // Trouver la couleur la plus proche dans la palette
            let closest_color = palette.iter().min_by(|&c1, &c2| {
                color_distance(&original_color, c1)
                    .partial_cmp(&color_distance(&original_color, c2))
                    .unwrap()
            }).unwrap();

            // Appliquer la couleur la plus proche au pixel
            image.put_pixel(x as u32, y as u32, *closest_color);

            // Calculer l'erreur (différence entre l'original et la couleur
choisie)
            let error = [
                original_color[0] as i32 - closest_color[0] as i32,
                original_color[1] as i32 - closest_color[1] as i32,
                original_color[2] as i32 - closest_color[2] as i32,
            ];

            // Diffuser l'erreur aux pixels voisins
            if x + 1 < width {
                let right_pixel = image.get_pixel(x as u32 + 1, y as u32);
                let new_right_pixel = [
                    (right_pixel[0] as i32 + error[0] * 7 / 16).clamp(0,
255) as u8,
                    (right_pixel[1] as i32 + error[1] * 7 / 16).clamp(0,
255) as u8,
                    (right_pixel[2] as i32 + error[2] * 7 / 16).clamp(0,
255) as u8,
                    255,
                ];
                image.put_pixel(x as u32 + 1, y as u32,
Rgba(new_right_pixel));
            }

            if y + 1 < height {
                if x > 0 {
```

```
                    let bottom_left_pixel = image.get_pixel(x as u32 - 1, y
as u32 + 1);
                    let new_bottom_left_pixel = [
                        (bottom_left_pixel[0] as i32 + error[0] * 3 /
16).clamp(0, 255) as u8,
                        (bottom_left_pixel[1] as i32 + error[1] * 3 /
16).clamp(0, 255) as u8,
                        (bottom_left_pixel[2] as i32 + error[2] * 3 /
16).clamp(0, 255) as u8,
                        255,
                    ];
                    image.put_pixel(x as u32 - 1, y as u32 + 1,
Rgba(new_bottom_left_pixel));
                }

                let bottom_pixel = image.get_pixel(x as u32, y as u32 + 1);
                let new_bottom_pixel = [
                    (bottom_pixel[0] as i32 + error[0] * 5 / 16).clamp(0,
255) as u8,
                    (bottom_pixel[1] as i32 + error[1] * 5 / 16).clamp(0,
255) as u8,
                    (bottom_pixel[2] as i32 + error[2] * 5 / 16).clamp(0,
255) as u8,
                    255,
                ];
                image.put_pixel(x as u32, y as u32 + 1,
Rgba(new_bottom_pixel));

                if x + 1 < width {
                    let bottom_right_pixel = image.get_pixel(x as u32 + 1,
y as u32 + 1);
                    let new_bottom_right_pixel = [
                        (bottom_right_pixel[0] as i32 + error[0] * 1 /
16).clamp(0, 255) as u8,
                        (bottom_right_pixel[1] as i32 + error[1] * 1 /
16).clamp(0, 255) as u8,
                        (bottom_right_pixel[2] as i32 + error[2] * 1 /
16).clamp(0, 255) as u8,
                        255,
                    ];
                    image.put_pixel(x as u32 + 1, y as u32 + 1,
Rgba(new_bottom_right_pixel));
                }
            }
        }
    }
}
```

Palette :

```
La fonction accepte une liste de couleurs représentant la palette. Chaque
pixel sera quantifié à la couleur la plus proche de cette palette.
```

Erreur :

```
L'erreur est calculée comme la différence entre les composantes de la
couleur originale et celles de la couleur la plus proche.
```

Diffusion :

```
L'erreur est distribuée aux pixels voisins selon les coefficients de
diffusion. Dans cet exemple, 50 % de l'erreur est envoyée au pixel de
droite et 50 % au pixel en dessous.
```

Question 19 - Implémenter la diffusion d'erreur pour la matrice de Floyd-Steinberg

Nous avons implémenté la fonction de diffusion d'erreur pour la palettisation d'images, en prenant en compte les trois composantes de couleur (R, G, B). La fonction utilise la matrice de Floyd-Steinberg pour répartir l'erreur de manière proportionnelle entre les pixels voisins :

```rust
fn error_diffusion_matrice_floyd_steinberg(image: &mut RgbaImage, palette:
&[Rgba<u8>]) {
    let width = image.width() as i32;
    let height = image.height() as i32;

    // Parcours des pixels
    for y in 0..height {
        for x in 0..width {
            let pixel = image.get_pixel(x as u32, y as u32);
            let original_color = *pixel;

            // Trouver la couleur la plus proche dans la palette
            let closest_color = palette.iter().min_by(|&c1, &c2| {
                color_distance(&original_color, c1)
                    .partial_cmp(&color_distance(&original_color, c2))
                    .unwrap()
            }).unwrap();

            // Appliquer la couleur la plus proche au pixel
            image.put_pixel(x as u32, y as u32, *closest_color);

            // Calculer l'erreur (différence entre l'original et la couleur
choisie)
            let error = [
                original_color[0] as i32 - closest_color[0] as i32,
                original_color[1] as i32 - closest_color[1] as i32,
                original_color[2] as i32 - closest_color[2] as i32,
            ];
```

```
            // Diffuser l'erreur aux pixels voisins selon Floyd-Steinberg
            if x + 1 < width {
                let right_pixel = image.get_pixel(x as u32 + 1, y as u32);
                let new_right_pixel = [
                    (right_pixel[0] as i32 + (error[0] * 7 / 16)).clamp(0,
255) as u8,
                    (right_pixel[1] as i32 + (error[1] * 7 / 16)).clamp(0,
255) as u8,
                    (right_pixel[2] as i32 + (error[2] * 7 / 16)).clamp(0,
255) as u8,
                    255,
                ];
                image.put_pixel(x as u32 + 1, y as u32,
Rgba(new_right_pixel));
            }

            if y + 1 < height {
                if x > 0 {
                    let bottom_left_pixel = image.get_pixel(x as u32 - 1, y
as u32 + 1);
                    let new_bottom_left_pixel = [
                        (bottom_left_pixel[0] as i32 + (error[0] * 3 /
16)).clamp(0, 255) as u8,
                        (bottom_left_pixel[1] as i32 + (error[1] * 3 /
16)).clamp(0, 255) as u8,
                        (bottom_left_pixel[2] as i32 + (error[2] * 3 /
16)).clamp(0, 255) as u8,
                        255,
                    ];
                    image.put_pixel(x as u32 - 1, y as u32 + 1,
Rgba(new_bottom_left_pixel));
                }

                let bottom_pixel = image.get_pixel(x as u32, y as u32 + 1);
                let new_bottom_pixel = [
                    (bottom_pixel[0] as i32 + (error[0] * 5 / 16)).clamp(0,
255) as u8,
                    (bottom_pixel[1] as i32 + (error[1] * 5 / 16)).clamp(0,
255) as u8,
                    (bottom_pixel[2] as i32 + (error[2] * 5 / 16)).clamp(0,
255) as u8,
                    255,
                ];
                image.put_pixel(x as u32, y as u32 + 1,
Rgba(new_bottom_pixel));

                if x + 1 < width {
                    let bottom_right_pixel = image.get_pixel(x as u32 + 1,
y as u32 + 1);
                    let new_bottom_right_pixel = [
                        (bottom_right_pixel[0] as i32 + (error[0] * 1 /
16)).clamp(0, 255) as u8,
                        (bottom_right_pixel[1] as i32 + (error[1] * 1 /
```

```
16)).clamp(0, 255) as u8,
                        (bottom_right_pixel[2] as i32 + (error[2] * 1 /
16)).clamp(0, 255) as u8,
                        255,
                    ];
                    image.put_pixel(x as u32 + 1, y as u32 + 1,
Rgba(new_bottom_right_pixel));
                }
            }
        }
    }
}
```

Question 20 - Comment représenter une matrice de diffusion d'erreur arbitraire? Permettre de changer de matrice de diffusion d'erreurs, et tester les matrices de diffusion de Jarvis-Judice-Ninke

---

Question 21 - Donner une spécification de votre interface sous forme d'un projet d'écran d'aide, tel que celui qui sera obtenu par cargo run -- --help

Pour permettre l'affichage des différentes fonction de l'interface avec la commande `cargo run -- --help` il a fallut ajouter ces fonction avec la bibliotheque argh.

```
#[derive(Debug, Clone, PartialEq, FromArgs)]
/// Rendu de l'image en monochrome par dithering
#[argh(subcommand, name="dithering")]
struct OptsDithering {}

#[derive(Debug, Clone, PartialEq, FromArgs)]
/// Rendu de l'image en monochrome en utilisant la matrice de Bayer
#[argh(subcommand, name="Bayer")]
struct OptsBayer {}
```

Question 22 - Déterminer le type Rust correspondant à une sélection d'options fournies par l'utilisateur

En Rust, pour déterminer le type correspondant à une sélection d'options fournies par l'utilisateur, on pourrait utiliser une énumération (enum) avec différentes variantes, puis associer ces variantes à des options spécifiques. Cela permet de définir un type qui peut prendre plusieurs valeurs, chacune correspondant à une option.

Voici un exemple de code en Rust pour illustrer ce principe :

```
use std::io;

enum OptionType {
```

```
    OptionA,
    OptionB,
    OptionC,
}

fn main() {
    println!("Sélectionnez une option:");
    println!("1. Option A");
    println!("2. Option B");
    println!("3. Option C");

    // Lire l'entrée de l'utilisateur
    let mut choix = String::new();
    io::stdin().read_line(&mut choix).expect("Échec de la lecture de la
ligne");

    let choix: u32 = match choix.trim().parse() {
        Ok(num) => num,
        Err(_) => {
            println!("Entrée invalide");
            return;
        }
    };

    // Associer l'entrée à un type OptionType
    let option = match choix {
        1 => OptionType::OptionA,
        2 => OptionType::OptionB,
        3 => OptionType::OptionC,
        _ => {
            println!("Option invalide");
            return;
        }
    };

    // Utiliser le type en fonction de la sélection
    match option {
        OptionType::OptionA => println!("Vous avez sélectionné Option A."),
        OptionType::OptionB => println!("Vous avez sélectionné Option B."),
        OptionType::OptionC => println!("Vous avez sélectionné Option C."),
    }
}
```

Question 23 - Implémenter votre interface en ligne de commande à l'aide de la directive #[derive(FromArgs)] sur votre type, suivant la documentation à https://docs.rs/argh/0.1.13/argh/