Création et manipulations d'images

Qualité de Développement - R5.A.08

Le thème de ce TP est la création et la modification d'images, un air de déjà vu…

Les considérations de qualité de développement doivent orienter votre travail.

Ce qui est fourni

Une bibliothèque qui permet de :

- Créer des images simples, monochromes;
- Sauvegarder une image dans un fichier;
- Charger une image depuis un fichier;
- Appliquer des modifications simples à une image :
- Modification de la luminosité
- L'application de filtres :
- Flou
- Détection de contours

Le format d'image utilisé est le format PPM.

Rappel sur le format

Le format **PPM** permet de stocker simplement les valeurs des différents pixels directement sous forme matricielle.

Par exemple, l'image avec 6 pixels suivante :

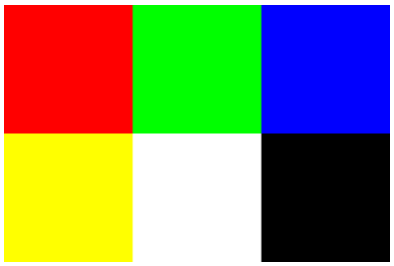

Figure 1. Simple 6 pixels

Correspond au fichier PPM suivant :

```
P3                                  ①
3 2                                 ②
255                                 ③
255 0   0  0   255 0    0 0 255     ④
255 255 0  255 255 255  0 0 0
```

(1) “P3”" pour le format ASCII, RGB.
(2) 3 pixel de large, 2 pixels de haut
(3) Valeur maximal possible pour les composants (rouge, vert ou bleu)
(4) Les valeurs des composantes sont ensuite mise les unes à la suite des autres

Les retours à la ligne n’ont pas besoin de correspondre au lignes de l’image.

Les lignes ne doivent normalement pas dépasser 74 caractères (mais cela semble fonctionner quand même dans la plupart des logiciels qui affichent des images). Tous les caractères d’espacement (espace, tabulation, `\n`) sont acceptés pour séparer les valeurs.

Il est possible d’ajouter des commentaires dans l’en-tête en commençant la ligne par `#`.

Ce qui est demandé

Tests

La bibliothèque fournie n’a pas été correctement testée et il existe des bugs dans les fonctions fournies. Avant d’ajouter une gestion des erreurs propre et d’ajouter de nouvelle fonctionnalités, il faudra donc :

- vérifier que les tests existants sont pertinents et compléter ceux qui sont incomplet,
- corriger les bugs pour que les tests existants passent,
- ajouter des tests pour valider toutes les fonctionnalités existantes.

Gestion des erreurs

Afin de rendre la bibliothèque plus robuste, vous allez mettre en place une gestion rigoureuse des erreurs.

Dans sa version actuelle, la bibliothèque a une fâcheuse tendance à "paniquer" lorsque quelque chose ne se passe pas comme il faut (fichier tronqué, valeurs invalides);

En utilisant les types dédiés à la gestion des erreurs et des résultats dans Rust, c’est-à-dire `Option` et `Result`, faites en sorte que les cas d’erreur soient pris en compte de manière adéquate :

- pas de "panic",
- des messages d’erreur clairs,
- des comportements par défaut conservateurs (on écrase pas un fichier existant implicitement).

Gestion des erreurs de chargement depuis un fichier

Commencez par la fonction qui charge une image depuis un fichier : `load_from_file`.

En l’état cette fonction retourne une image en cas de succès et panique si quelque chose ne se passe pas bien. Ce comportement ne correspond pas un un code de qualité. Pour autant il est tout à fait probable que le chargement d’un fichier échoue (mauvais nom de fichier, fichier tronqué, etc.).

Vous allez donc utiliser les outils de Rust pour gérer correctement les cas d'erreur.

Erreur de lecture dans le fichier

Faites en sorte de traiter le résultat de la fonction `read_to_string` correctement à l'aide d'un `match` :

```
let file_content = match std::fs::read_to_string(path) {
    Ok(s) => s,
    Err(e) => {
        eprintln!("Failed to read file.");
        return Err(e);
    }
};
```

Pour que cela compile, vous devrez adapter votre code :

- Changez le type de retour de la fonction vers `Result<Image, io::Error>`
- Retournez un `Ok` de l'image chargée en cas de succès.

Le premier changement vous obligera à modifier le code qui utilise la fonction `load_from_file`. Le compilateur vous le dira.

Le fait de retourner un `Result` plutôt que directement une Image va permettre de :

- traiter proprement les cas où une image ne peut pas être créée,
- obliger l'utilisateur de la bibliothèque à gérer ces cas.

Fichier tronqué

Le deuxième type d'erreur est celle produite par un fichier qui ne contient pas assez d'information pour générer une image : fichier vide, en-tête incomplet, valeurs de pixels manquantes.

Ces erreurs seront détecter lors de l'appel à `next()` sur le résultat du `split_whitespace`. Ce `next` retourne une `Option<&str>` qui correspond à une chaîne de caractère ou à la valeur None suivant si des éléments sont encore présents.

Dans le code fourni, il existe déjà un `match` qui traite la première ligne de l'en-tête. Il serait souhaitable que ce `match` ne panique pas et retourne une erreur. Le problème est que retourner une erreur du type `io::Error` n'est pas approprié puisqu'il ne s'agit pas ici d'une erreur d'entrées-sorties.

Pour résoudre ce problème il existe au moins deux options :

1. Retourner `Box<dyn Error>`. Cela permet de retourner n'importe quel type d'erreur en l'enrobant dans une "Box". Les inconvénients associés à cette solution sont que l'on propage différents types d'erreur, de manière implicite, et que l'on doit "Boxer" toutes les erreurs.
2. Retourner un type d'erreur spécifique à la bibliothèque qui représente explicitement les différents type d'erreur rencontrées lors du fonctionnement de la bibliothèque. L'inconvénient est la nécessité de mettre à jour le type si un nouveau type d'erreur doit être traité.

Vous allez implémenter la deuxième solution.

Création d'un type d'erreur dédié

Pour pouvoir retourner une erreur du même type dans les deux cas d'erreur précédents (erreur de lecture dans un fichier et fichier tronqué) vous allez créer un nouveau type d'erreur, dédié à la bibliothèque.

Pour cela, ajoutez dans votre projet un `enum` "**PPMImageError**" qui contiendra les différentes valeurs d'erreur associées aux erreurs de la bibliothèque.

```
pub enum PPMImageError {
    IOError,
    TruncatedFile,
    // Other relevant error types
}
```

Maintenant que vous avez un type d'erreur spécialisé pour votre bibliothèque, faites en sorte que le type de retour de la fonction de chargement d'une image à partir d'un fichier soit :

```
pub fn load_from_file(path: &Path) -> Result<Image, PPMImageError> {...}
```

Pour que le code compile, vous devrez :

- Adapter les sorties anticipées de la fonction (`return`) qui correspondent aux cas d'erreur, pour que le bon type soit retourné,
- Adapter les utilisations de cette fonction à ce nouveau type de retour pour que le code compile. Notamment les tests.

Pour le premier point, le compilateur vous expliquera que vous avez besoin de la fonction `ok_or` du type `Option` pour transformer un "Ok" en valeur et un "None" en erreur.

Utilisation de l'opérateur "?"

La construction suivante qui gère les erreurs de lecture dans un fichier est typique de la gestion des erreurs en Rust :

```
let file_content = match std::fs::read_to_string(path) {
    Ok(s) => s,
    Err(e) => Err(e),
};
```

En résumé cela consiste à affecter une valeur à une variable en cas de succès et à sortir de la fonction en cas d'erreur.

Cette construction est tellement typique et fréquente, que le langage possède un raccourci pour

l'exprimer :

```
let file_content = std::fs::read_to_string(path)?;
```

Beaucoup plus pratique, non ?

Remplacez, dans la fonction `load_from_file`, le `match` qui affecte une valeur à la variable `file_content` par une utilisation de l'opérateur "**?**".

Il y a un problème de compilation. En effet, dans le cas présent, l'opérateur "?" sort de la fonction `load_from_file` en en renvoyant l'erreur produite par la fonction `read_to_string`, c'est-à-dire `std::io::Error`. Il faut donc fournir un moyen au compilateur de produire une erreur du type `PPMImageError` à partir d'une erreur standard. Pour cela vous devrez implémenter le trait `From<io::Error>` pour le type `PPMImageError` :

```
impl From<io::Error> for PPMImageError {
    fn from(_e: io::Error) -> Self {
        PPMImageError::IOError
    }
}
```

Affichage des erreurs

Maintenant que vous avez un type d'erreur dédié à la bibliothèque, il pourrait être utile que le message associé à une erreur soit pertinent. Pour cela, Implémentez les traits "Display" et "Debug" pour le type `PPMImageError` :

```
impl Debug for PPMImageError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        match self {
            PPMImageError::IOError => write!(f, "IO Error"),
            PPMImageError::TruncatedFile => write!(f, "truncated file"),
            // Other relevant errors
        }
    }
}
```

Maintenant que vous savez gérer une erreur spécifique à la bibliothèque, ajoutez les autres types d'erreur potentiellement rencontrées (erreur d'écriture, tentative d'écraser un fichier, valeur invalide pour un pixel dans la lecture d'un fichier, etc.) aux différentes fonctions de la bibliothèque.

Il ne doit plus rester de `expect`, de `unwrap` ou autres fonctions qui paniquent.

Fonctionnalités supplémentaires

- Copie d'une image

- Création de motifs simples :
  - Gradient
  - Damier
- Recadrage de l'image
- Modification de la résolution de l'image
- Zoom blur
- Enregistrement au format binaire (P6).
- Gestion des commentaires dans l'en-tête.

Logging (bonus)

Afin de rendre les messages d'erreur et d'information plus pertinents et flexibles, vous utiliserez le logger "env_logger".

Amélioration des erreurs (bonus)

Afin de rendre plus fine la gestion des erreurs, vous ajouterez de l'information aux différents types d'erreurs du type `PPMImageError`. Comme premier exercice, ajoutez l'erreur initiale à la variante `IOError` de ce type :

```
pub enum PPMImageError {
    // ...
    IOError(std::io::Error),
    // ...
}
```

Il vous faudra modifier l'implémentation du trait `From<io::Error>` afin de construire correctement l'erreur.

Cela vous permettra de traiter plus finement des erreurs ainsi que d'afficher un message plus pertinent :

```
impl Debug for PPMImageError {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        match self {
            // ...
            PPMImageError::IOError(e) => write!(f, "IO Error: {}", e),
            // ...
        }
    }
}
```