Compte rendu Mini-projet 2

Etape 1 :

L'objectif de cette première étape est de découvrir comment étendre un programme en langage interprété (Python) par une bibliothèque de code natif (écrite en Rust ici). Cela permet de tirer parti de la performance des langages compilés pour exécuter des traitements coûteux, tout en bénéficiant de la flexibilité des langages interprétés.

Réalisation

Création de la Bibliothèque Rust

1. **Initialisation de la crate Rust** :

   `cargo new mini_python_rust –lib`

   Cela crée une structure de projet Rust standard avec un fichier `Cargo.toml` et un répertoire `src/`.

2. **Renommer et modifier les fichiers** :

- Le fichier `src/main.rs` est renommé en `src/lib.rs`.
- Le contenu de `src/lib.rs` est modifié comme suit :

```
#[no_mangle]
pub extern "C" fn say_hello() {
 println!("Hello from Rust!");
}
```

La directive `#[no_mangle]` empêche la modification du nom de la fonction lors de la compilation, ce qui garantit qu'elle pourra être retrouvée par son nom original.

3. **Modification de `Cargo.toml`** : La section suivante est ajoutée :

```
[lib]
crate-type = ["cdylib"]
```

Cela spécifie que la crate doit être compilée sous forme de bibliothèque dynamique compatible avec les systèmes d'exploitation.

4. **Compilation** : La commande suivante compile la crate :

   `cargo build`

   Sous Linux/WSL, le fichier généré est `target/debug/libmini_python_rust.so`.

- Sous macOS, c'est `target/debug/libmini_python_rust.dylib`.
- Sous Windows, c'est `target/debug/mini_python_rust.dll`.

Personnellement,j’ai chois d’exécuter le programme sous Windows.

Création du Programme Python

1. **Fichier Python** : Un fichier `mini_python_rust.py` est créé à la racine du projet avec le contenu suivant :

```
import sys
import ctypes

print()
native_lib_name = sys.argv[1]
print('• loading', native_lib_name)
native_lib = ctypes.CDLL(native_lib_name)

print()
function_1_name = 'say_hello'
print('• accessing', function_1_name)
function_1 = native_lib[function_1_name]
print('• calling', function_1_name)
function_1()
```

   Ce script :

   - Charge une bibliothèque native via `ctypes.CDLL()`.
   - Accède à la fonction `say_hello` par son nom.
   - Appelle la fonction, ce qui produit une sortie sur la console.

2. **Exécution** : Le programme est exécuté avec :

```
python3 mini_python_rust.py target/debug/mini_python_rust.dll
```

Resultats

```
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\mini_python_rust> cargo build
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\mini_python_rust> python3 mini_python_rust.py target/deb
ug/mini_python_rust.dll

• loading target/debug/mini_python_rust.dll

• accessing say_hello
• calling say_hello
Hello from Rust!
```

- **Conclusion :**
  - La bibliothèque native est correctement chargée par Python.
  - La fonction `say_hello` est retrouvée et invoquée sans erreur.
  - Le message "Hello from Rust!" produit par Rust est affiché sur la console.

Etape 2 :

L'objectif de cette partie du projet est d'intégrer Rust et Python en utilisant des appels inter-langages via des fonctions exportées en Rust. Ces fonctions permettent de manipuler des valeurs numériques, des chaînes de caractères, et des tableaux de manière transparente entre les deux langages.

Code Rust

1. Fonctionnement `compute`

La fonction `compute` effectue des calculs mathématiques entre deux nombres flottants (de type `c_double`) en fonction d'une opération donnée sous forme de chaîne de caractères.

```rust
use std::ffi::c_double;

#[no_mangle]
fn say_hello() {
    println!("Hello from Rust!");
}

#[no_mangle]
pub extern "C" fn compute(
    a: c_double,
    b: c_double,
    c_operation: *const std::ffi::c_char,
) -> c_double {
    let operation =
        unsafe { std::ffi::CStr::from_ptr(c_operation) }.to_string_lossy();
    // Effectuer l'opération demandée
    match operation.as_ref() {
        "add" => a + b,
        "sub" => a - b,
        "mul" => a * b,
        "div" => {
            if b != 0.0 {
                a / b
            } else {
                0.0 // Retourne 0.0 en cas de division par zéro
            }
        }
        _ => 0.0, // Retourne 0.0 pour les opérations inconnues
    }
}
```

- Décorateur `#[no_mangle]`et `extern "C"` : Ces attributs indiquent que la fonction est exportée de manière compatible avec le langage C (et donc avec Python via `ctypes`).
    - **Paramètres** :
        - `a` et `b` sont des réels de type `c_double` (équivalent du type `double` en C).
        - `c_operation` est un pointeur vers une chaîne de caractères C (`*const c_char`), représentant l'opération à réaliser (addition, soustraction, multiplication, division).
    - : La fonction utilise un `match` pour déterminer l'opération à effectuer en fonction de la valeur de la chaîne `operation`. Si l'opération est valide (`"add"`, `"sub"`, `"mul"`, `"div"`), elle effectue l'opération correspondante. En

cas de division par zéro, elle renvoie `0.0`. Si l'opération est inconnue, elle renvoie également `0.`

Intégration avec Python

L'intégration avec Python se fait via le module `ctypes` pour appeler ces fonctions Rust en respectant la convention de passage de paramètres du langage C.

1. Passage de valeurs numériques

La fonction `compute` est appelée depuis Python avec deux nombres flottants et une chaîne représentant l'opération.

mini_python_rust > mini_python_rust.py > ...

```python
import sys
import ctypes

# function1
print()
native_lib_name=sys.argv[1]
print('• loading', native_lib_name)
native_lib=ctypes.CDLL(native_lib_name)

print()
function_1_name='say_hello'
print('• accessing', function_1_name)
function_1=native_lib[function_1_name]
print('• calling', function_1_name)
function_1()

#function2

print()
function_2_name='compute'
print('• accessing', function_2_name)
function_2=native_lib[function_2_name]
print('• calling', function_2_name)
function_2.argtypes=[ctypes.c_double, ctypes.c_double]
function_2.restype=ctypes.c_double
a=1
b=2
operation = ['add', 'sub', 'mul', 'div', 'unkknown']
for i in operation:
    op_str = i.encode()
    result = function_2(a, b, op_str)
    print(f'Compute, a={a},b={b},operation={i} :', result)
```

Explications

1. **Chargement de la bibliothèque Rust** :
    - La bibliothèque Rust est chargée dynamique en utilisant `ctypes.CDLL(native_lib_name)`. Le nom de la bibliothèque native (par exemple, `libmylibrary.so` ou `mylibrary.dll`) est passé en argument lors de l'exécution du script Python.

```
native_lib_name = sys.argv[1]
native_lib = ctypes.CDLL(native_lib_name)
```

2. **Appel de la fonction** `say_hello` :

- Cette fonction est simplement appelée pour afficher un message de confirmation que l'intégration fonctionne correctement.

```
function_1_name = 'say_hello'
function_1 = native_lib[function_1_name]
function_1()
```

3. **Appel de la fonction** `compute` :

- La fonction `compute` est appelée avec deux arguments de type `c_double` (des réels) et un troisième argument de type chaîne de caractères (représentée par `ctypes.c_char_p`).
- Avant d'appeler la fonction, nousssons les types d'arguments (`argtypes`) et le type de retour (`restype`).

```
function_2.argtypes = [ctypes.c_double, ctypes.c_double, ctypes.c_char_p]
function_2.restype = ctypes.c_double
```

4. **Test des opérations** :

- Nous définissons une liste d'un test : ajout `'add'`), soustraction (`'sub'`), multiplication (`'mul'`), divis `'div'`), et une opération inconnue (`'unkknown'`).
- Chaque opération est convertie en chaîne d'octets via `.encode('utf-8')` pour être compatible avec le type `ctypes.c_char_p` attendu par la fonction Rust.

```
operation = ['add', 'sub', 'mul', 'div', 'unkknown']
for i in operation:
    op_str = i.encode()
    result = function_2(a, b, op_str)
    print(f'Compute, a={a},b={b},operation={i} :', result)
```

Resultat

Lorsque l’on exécute le programme python avec la commande : `python3 mini_python_rust.py target/debug/mini_python_rust.dll`, on obtient le résultat suivant :

```
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\mini_python_rust> cargo build
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.02s
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\mini_python_rust> python3 mini_python_rust.py target/deb
ug/mini_python_rust.dll

• loading target/debug/mini_python_rust.dll

• accessing say_hello
• calling say_hello
Hello from Rust!

• accessing compute
• calling compute
a = 1
b= 2
a+b= 3.0
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\mini_python_rust>
```

- Le code Rust est correctement compilé en une bibliothèque partagée (`.so`).
- le nom de la bibliothèque Rust compilée en argument lors de l'exécution du script Python

L’étape 3 :

L’étape 3 était l’explication du squelette du code qui va nous servir plus tard pour construire notre client.

L’étape 4 :

L'objectif de cette étape est de créer une fonction `load_image` qui charge une image **PPM/P3**, un format très simple basé sur du texte, dans une structure `Image`. La fonction devait être capable de traiter le contenu du fichier image, d'extraire ses dimensions et ses pixels, et de gérer les erreurs qui pourraient survenir lors de la lecture ou du parsing des données.

Fonctionnement de la fonction `load_image`

1. **Vérification de l'existence du fichier**
   Avant de commencer le traitement, la fonction vérifie que le brevet breveté par le chemin `path`aire de bien. Si le fichier est introuvable, une erreur descriptive est renvoyée.

```
if !std::path::Path::new(path).exists() {
    return Err(format!("Le fichier '{}' est introuvable.", path).into());
```

2. **Conférence du fichier et préparation de l'itérateur sur les mots** :

   Une fois le fichier ouvert, son contenu est lu dans un tableau d'octets, puis converti en une chaîne de caractères. Un itérateur est ensuite créé sur les mots du fichier, en ignorant les lignes de commentaires (délimitées par un #) et en extrayant les mots non vides.

```rust
let content = fs::read(path)?;

// Construire un itérateur
let mut words = std::str::from_utf8(&content)?
    .lines()
    .map(|l| l.find('#').map_or(l, |pos| &l[0..pos]))
    .flat_map(|l| l.split_whitespace())
    .filter(|w| !w.is_empty());
```

3. **Vérification du marqueur de format** :
   Le premier mot du fichier doit être `"P3"`, indiquant que le fichier est au format texte PPM/P3. Si ce n'est pas le cas, une erreur est retournée.

```rust
// Vérifier le marqueur de format
match words.next() {
    Some("P3") => (),
    _ => return Err("Invalid format marker (expected P3)".into()),
}
```

4. **Extraction des dimensions de l'image** : La largeur et la hauteur de l'image sont extraites des mots suivants. Si ces valeurs sont absentes ou mal formatées, une erreur est renvoyée.

```rust
// Extraire la largeur et la hauteur
let width = words.next().ok_or("Missing width")?.parse::<usize>()?;
let height = words.next().ok_or("Missing height")?.parse::<usize>()?;
```

5. **Vérification de la valeur maximale des couleurs** :
   Le fichier PPM/P3 doit spécifier la valeur maximale pour chaque composante de couleur (généralement 255). Si ce n'est pas le cas, une erreur est renvoyée.

```rust
    // Vérifier la valeur maximale (doit être 255)
    match words.next() {
        Some("255") => (),
        _ => return Err("Invalid max value (expected 255)".into()),
    }
```

6. **Chargement des pixels :**

La fonction lit les triplets de valeurs représentant les couleurs des pixels. Chaque triplet est composé de trois nombres représentant les composantes rouge, verte et bleue. Ces valeurs sont converties en `u8` et stockées dans un vecteur de type `Vec<Color>`, , , , `Color` est une structure définissant les trois composantes de couleur.

```rust
// Charger les pixels
let mut pixels = Vec::with_capacity(width * height);
while let (Some(r), Some(g), Some(b)) =
    (words.next(), words.next(), words.next())
{
    let color = Color {
        r: r.parse::<u8>()?,
        g: g.parse::<u8>()?,
        b: b.parse::<u8>()?,
    };
    pixels.push(color);
}
```

7. **Vérification du nombre de pixels** :
   Après avoir extrait les triplets de couleurs, la fonction vérifie que le nombre de pixels correspond bien à la largeur multipliée par la hauteur. Si ce n'est pas le cas, une erreur est renvoyée.

```rust
// Vérifier que tous les pixels ont été chargés
if pixels.len() != width * height {
    return Err("Pixel count does not match width x height".into());
}
```

8. **Retour de l'image** :
   Si tout se passe bien, la fonction retourne une structure `Image` contenant la largeur, la hauteur et le vecteur des pixels.

```rust
    // Retourne l'image
    Ok(Image {
        width,
        height,
        pixels,
    })
}
```

Etape 5 :

L'objectif de cette étape est d'intégrer une image dans la structure `Application` et de la dessiner sur l'écran en utilisant la fonction `draw_image`. Cette fonction doit être capable de gérer

l'affichage de l'image à une position spécifique sur l'écran tout en respectant les limites de celui-ci. La fonction doit également permettre de dessiner uniquement la portion de l'image qui tient sur l'écran.

Mise à jour de la structure

Structure de la structure `Application` doit maintenant contenir deux nouveaux membres :

1. : une instance de la str`image`: une ins`Image`, représentant l'image à afficher.
2. `position` : un `Point` représentant la position de l'image sur l'écran (initialisée à `(0, 0)` dans `init_application()`).

Fonctionnement

La fonction `draw_image` est responsable du dessin de l'image à l'écran. Elle prend en entrée trois paramètres :

- : une référence mutable à l'écran (`Screen`) surmanoue l'image dessi
- `image`: une référence partagée à l'image (`Image`) à une
- `position`: : `Point` indiquant où placer l'image sur l'écran.

La fonction commence par le calcul des points de départ (`p0`) et`p1`) de l'image sur l'écran, en s'assurant que l'image ne dépasse pas les bords de l'écran. Cette étape est réalisée en "clampant" les coordonnées pour qu'elles restent dans les limites de l'écran.

```
let p0 = Point {
    x: position.x.clamp(0, screen.width as i32),
    y: position.y.clamp(0, screen.height as i32),
};
let p1 = Point {
    x: (position.x + image.width as i32).clamp(0, screen.width as i32),
    y: (position.y + image.height as i32).clamp(0, screen.height as i32),
};
```

Calcul des indices des pixels

Ensuite, la fonction calcule les indices de départ pour l'image (`i_idx`) et l'écran (`s_idx`) en tenant compte des décalages (`dx` et zdy`dy`) afin de dessiner seulement la portion visible de l'image sur l'écran. Ces indices permettent de naviguer dans les pixels de l'image et de l'écran.

```
let dx = 0.max(p0.x - position.x);
let dy = 0.max(p0.y - position.y);
let mut image_idx = dy as usize * image.width + dx as usize;
let mut screen_idx = p0.y as usize * screen.width + p0.x as usize;
let w = 0.max(p1.x - p0.x) as usize;
```

Dessin de l'image

La fonction parcourt ensuite chaque ligne de l'image à dessiner (de `p0.y` à `p1.y`). Pour chaque ligne, elle copie les pixels correspondants de l'image (`src`) vers `dst`. Les indices `image_idx` et `screen_idx` sont utilisés pour accéder aux pixels de l'image et de l'écran, respectivement.

```
for _ in p0.y..p1.y {
    let src = &image.pixels[image_idx..image_idx + w];
    let dst = &mut screen.pixels[screen_idx..screen_idx + w];
```

À chaque itération de la boucle, une portion de ligne horizontale de l'image est copiée dans l'écran, à la position spécifiée. Le calcul des indices assure que l'image est dessinée correctement même si elle est partiellement affichée sur l'écran.

Fonctionnement `redraw_if_needed`

La fonction`redraw_if_needed()`, une fonction qui permet de rafraîchir l'écran en cas de besoin. Elle sera appelée après la boucle qui couvre l'écran d'une couleur uniforme. Les paramètres `screen` et `app` de `redraw_if_needed()` sont utilisés pour appeler la fonction `draw_image` et afficher l'image choisie.

Etape 6 :

L'objectif de cette étape est d'inconnie la notion de transparence dans l'affichage des images. Pour cela, une couleur spécifique (le vert vif, avec les DRB)

Mise à jour de la fonction

La fonction `Option<Color>` a été ajouté. Ce paramètre peut contenir une couleur spécifique qui indique la couleur transparente à ne pas dessiner.

- Si ce paramètre est
- Si ce paramètre contenu dans une couleur (`Some(tr)`), alors la fonction vérifie chaque pixel de l'image. Si un pixel correspond à la couleur de transparence, il n'est pas copié sur l'écran. Autrement dit, si un pixel de l'image est de couleur `tr`, l'écran garde sa couleur existante à cet endroit. Sinon, la couleur du pixel de l'image est copiée sur l'écran.

Voici la modification apportée dans la fonction `draw_image` :

```
388         match transparent_color {
389             None => {
390                 dst.copy_from_slice(src);
391             }
392             Some(tr) => {
393                 for (src_pixel, dst_pixel) in src.iter().zip(dst.iter_mut()) {
394                     if *src_pixel != tr {
395                         *dst_pixel = *src_pixel;
396                     }
397                 }
398             }
399         }
400
```

Utilisation de la transparence

La couleur de fond de l'image est la couleur verte vive, avec les composantes RGB `(0, 255, 0)`. Lors du dessin de l'image, cette couleur ne sera pas copiée sur l'écran si elle est spécifiée comme couleur transparente. Le code de la fonction `draw_image` s'assure que cette couleur n'apparaît pas sur l'écran et laisse l'arrière-plan de l'écran inchangé.

Modifications dans la fonction `redraw_if_needed`

Pour le testeur l'effet de la transparence, il faut faut fauter un paramètre à la fonction `draw_image` depuis la fon`redraw_if_needed`. La couleur verte vive `(0, 255, 0)` sera fournie à la fonction sous la forme `Some(Color { r: 0, g: 255, b: 0 })`.

```
252  fn redraw_if_needed(
253      app: &Application,
254      screen: &mut Screen,
255  ) {
256      if let UpdateStatus::Redraw = app.status {
257          for c in screen.pixels.iter_mut() {
258              let (r, g, b) =
259                  (c.r as u32 + 10, c.g as u32 + 25, c.b as u32 + 35);
260              c.r = r as u8;
261              c.g = g as u8;
262              c.b = b as u8;
263          }
264          let position = Point {
265              x: app.position.x,
266              y: app.position.y,
267          };
268          let transparent_color = Some(Color { r: 0, g: 255, b: 0 });
269          draw_image(screen, &app.image, position, transparent_color);
270      }
271  }
```

Etape 7 :

L'objectif principal de cette étape est d'ajouter la capacité de déplacer une image dans l'application graphique en utilisant les flèches du clavier. De plus, cette étape implique l'amélioration du rendu visuel pour éviter une traînée colorée indésirable lors du déplacement de l'image.

Analyse de la fonctionnement général de `update_application()`

1. La fonction `update_application()` est responsable de la mise à jour de l'état de l'application en fonction des événements reçus, ainsi que du redessin de l'interface graphique. Elle prend plusieurs paramètres :
    - `evt` : L'événement (par exemple, pression de touche ou mouvement de souris).
    - `key` : La clé associée à l'événement.
    - `btn` : Un identifiant pour le bouton cliqué.
    - `point` : La position de la souris ou du curseur à un moment donné.
    - `screen` : L'écran sur lequel l'image est affichée.
    - `app` : La structure représentant l'état actuel de l'application.
2. **Logique de la fonction**

    La fonction commence par une série de vérifications et de traitements avant de se concentrer sur la gestion des événements. Voici les étapes clés :

    - **Vérification de l'événement (`evt`)** : Si l'événement n'est pas un "T" (qui pourrait être un code d'événement spécifique), un message de débogage est affiché avec les informations liées à l'événement, la touche pressée, et la position de la souris.
    - **Modification de l'état de l'application (`app.status`)** : Par défaut, l'état de l'application est défini sur `UpdateStatus::GoOn`, ce qui permet à l'application de continuer son exécution.
    - **Réinitialisation du fond de l'écran :** Tous les pixels de l'écran sont mis à zéro pour obtenir une couleur de fond noire :

```
191
192     // Remplacer la couleur de fond
193     for c in screen.pixels.iter_mut() {
194         c.r = 0;
195         c.g = 0;
196         c.b = 0;
197     }
198
```

3. **Gestion des événements et déplacement de l'image**

- La fonction `handle_event()` est appelée pour analyser l'événement et déterminer s'il s'agit d'un mouvement. Elle renvoie un `Option<Motion>` (où `Motion` est probablement une structure représentant un déplacement, sous la forme d'un vecteur ou d'un `Point`).

- Si un mouvement est détecté (`if let Some(motion) = handle_event(app, evt, key)`), l'état de l'image est mis à jour en modifiant les coordonnées de `app.position` avec les valeurs de `motion` :

  ```
  app.position.x += motion.x;
  ```

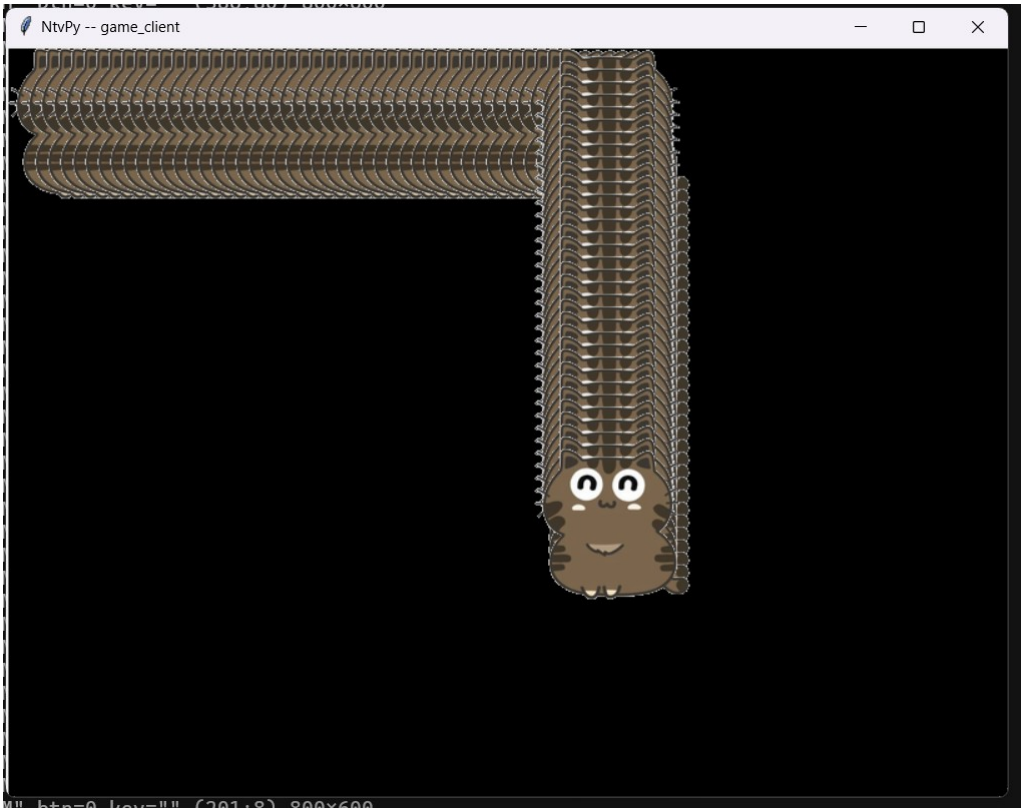
  ```
  app.position.y += motion.y;
  ```

Cette mise à jour déplace l'image à la nouvelle position en fonction de la direction et de la distance indiquées par le `motion`.

- **Demande de redessin** : Pour que l'application redessine l'image après le déplacement, l'état de l'application est mis à jour avec `UpdateStatus::Redraw`, indiquant qu'un redessin est nécessaire

Gestion du fond d'écran

- À chaque redessin, la couleur de fond de l'écran était initialement changée, créant une traînée colorée derrière l'image en mouvement. Cela a été modifié pour utiliser une couleur de fond fixe, comme le gris, afin de rendre l'interface plus propre et d'éviter cet effet visuel perturbant :



Pour corriger cela on fixe une couleur de fond. J’ai choisi la couleur noir dans la fonction `update_application()` :

```rust
// Remplacer la couleur de fond
for c in screen.pixels.iter_mut() {
    c.r = 0;
    c.g = 0;
    c.b = 0;
}
```

Resultat dans le terminal :

Etape 8 :

La fonction `init_application` a pour but d'initialiser l'application en chargeant l'image et en établissant une connexion TCP avec le serveur, tout en configurant les paramètres de la fenêtre et de la mise à jour de l'application. Elle prépare ainsi l'application pour la gestion des événements et l'interaction avec le serveur.

1. **Initialisation des paramètres de la fenêtre**
    - La fonction commence par initialiser les dimensions de la fenêtre de l'application (800x600 pixels) et la fréquence de mise à jour (30 FPS, représentée par `dt`).

```rust
) -> Result<Application, Box<dyn std::error::Error>> {
    println!("args: {:?}", args);
    *width = 800;
    *height = 600;
    *dt = 1.0 / 30.0;
```

2. **Chargement de l'image :**

    L'image à afficher dans l'application est chargée à partir d'un chemin spécifié dans les arguments de ligne de commande. L'argument correspondant est récupéré dans `args[2]`. Si l'image est correctement chargée, elle est retournée et stockée dans la structure de l'application. Sinon, une erreur est renvoyée.

```rust
    let position = Point { x: 0, y: 0 };
    let image = if let Some(image_path) = args.get(2) {
        println!("Loading image from: {}", image_path);
        match load_image(image_path) {
            Ok(image) => {
                //println!("Image loaded successfully: {:?}", image);
                image
            }
            Err(e) => {
                eprintln!("Failed to load image: {}", e);
                return Err(e);
            }
        }
    } else {
        return Err("No image path provided in arguments".into()); // Erreur si aucun chemin d'image n'est fourni
    };

```

3. **Initialisation de la connexion au serveur :**

   la fonction init_server récupère le nom du serveur et le port depuis les arguments `args[3]` et `args[4]`. Ces informations sont nécessaires pour établir une connexion avec le serveur via TCP.

```rust
fn init_server(
    args: &[&str]
) -> Result<(TcpStream, TcpStream), Box<dyn std::error::Error>> {
    let server_name =
        args.get(3).ok_or("Server name not provided in arguments")?;
    let port = args.get(4).ok_or("Port not provided in arguments")?;
    let server_address = format!("{}:{}", server_name, port);

    // Connexion au serveur
    println!("Connecting to server at {}", server_address);
    let stream = TcpStream::connect(&server_address).map_err(|e| {
        eprintln!("Failed to connect to server: {}", e);
        e
    })?;
    println!("Connected to server at {}", server_address);

    // Duplication du stream pour input/output
    let output = stream.try_clone()?;
    let input = stream;
    Ok((output, input))
}
```

Etape 9

L’étape 9 consiste à implémenter un échange de données entre un client et un serveur via une connexion TCP. L'objectif principal est de transmettre des informations concernant le mouvement du client afin que le serveur calcule la nouvelle position et la renvoie au client.

1. Flux de travail côté client

Lorsque le mouvement est détecté, le client doit envoyer un message de type `motion x y` au serveur. La fonction `update_application()` du client est modifiée pour envoyer ce message au serveur à chaque mouvement détecté :

```
let msg = format!("motion {} {}\n", motion.x, motion.y);
```

2. Gestion des messages côté serveur

Le serveur, une fois la connexion établie, est chargé de recevoir les messages du client et de calculer la nouvelle position du client en fonction du mouvement. La gestion des messages du client est organisée comme suit :

2.1 Lecture des lignes non bloquantes

Une nouvelle fonction, `read_lines_nonblocking()`, est ajoutée pour lire les messages entrants sans bloquer le serveur. Elle utilise la méthode `set_nonblocking` sur le `TcpStream` pour éviter que le serveur ne soit bloqué si aucune donnée n’est disponible. Si aucune ligne n'est reçue, la fonction renverra un vecteur vide, indiquant qu'il n'y a pas de nouvelles informations. Si la connexion est fermée, la fonction renverra une ligne vide pour signaler la déconnexion du client.

2.2 Traitement des messages reçus

La fonction `handle_messages()` du serveur est responsable de traiter les lignes lues par `read_lines_nonblocking()`. Elle prend en charge l’analyse de chaque ligne de texte reçue :

- Si la ligne est vide, cela indique la fin de la connexion, et le serveur doit se préparer à quitter.
- Si la ligne contient des données de type `position x y`, elle met à jour les coordonnées du client.
- Pour d'autres types de messages, le serveur peut les afficher sans traitement supplémentaire.

Exemple d’analyse d'une ligne `position` :

```
if let Some(data) = line.strip_prefix("position ") {
let mut words = data.split_whitespace();
let x = words.next().ok_or("missing x")?.parse::<i32>()?;
let y = words.next().ok_or("missing y")?.parse::<i32>()?;
```

}

Le serveur répond ensuite avec la ligne :

```
let reply = format!("position {} {}\n", x, y);
 output.write_all(reply.as_bytes())?;
output.flush()?;
```

2.4 Limites du jeu

Bien que ce système permette d'échanger les positions du client, il est possible de l'étendre pour appliquer des règles de jeu. Le serveur impose des limites à la position du client avec la fonction map certains mouvements pour introduire de la contrainte dans le jeu.

Resultats

Les résultats dans le terminal seront présenté dans l'étape car ce sont les mêmes.

Étape 10 :

L'étape 10 vise à simplifier l'échange d'informations complexes entre le client et le serveur en utilisant des mécanismes de sérialisation et de désérialisation. Cette approche permet de transformer des structures de données en flux d'octets (sérialisation) et de reconstruire ces structures à partir de ces flux (désérialisation). L'utilisation de la bibliothèque `serde` en Rust permet d'automatiser cette transformation pour simplifier la gestion des messages dans l'application.

1. Sérialisation et Désérialisation

1.1 Modification de la structure `Point`

Dans cette étape, il est nécessaire de qualifier la structure `Point` avec les traits `Serialize` et `Deserialize` de la bibliothèque `serde`. Cela permet de sérialiser la structure en un format JSON et de désérialiser des messages JSON en une structure `Point`.

Ces traits sont utilisés pour gérer automatiquement la conversion de la structure en une chaîne JSON (sérialisation) et la conversion inverse (désérialisation).

1.2 Sérialisation du message dans le client

Dans la fonction `update_application()`, la ligne envoyée au serveur ne contiendra plus les coordonnées `x` et `y` explicitement. Au lieu de cela, le client utilise `serde_json::to_string(&motion)` pour sérialiser la structure `Point` en une chaîne JSON et l'envoyer au serveur :

Cela permet de gérer plus facilement les messages complexes, sans avoir besoin de gérer manuellement chaque valeur ou champ de manière individuelle.

```rust
    if let Some(output) = app.output.as_mut() {
        // Sérialiser motion
        match serde_json::to_string(&motion) {
            Ok(json_motion) => {
                println!("json_serialised_motion {:?}", json_motion);
                let msg = format!("motion {}\n", json_motion);
                output.write_all(msg.as_bytes())?;
                output.flush()?;
                app.status = UpdateStatus::Redraw;
            }
            Err(e) => {
                eprintln!("Erreur lors de la sérialisation JSON : {}", e);
            }
        }
    } else {
        println!("Aucun flux de sortie disponible pour envoyer la demande au serveur.");
    }
}
```

1.3 Désérialisation du message dans le serveur

Lors de la réception d'un message côté serveur, la structure `Point` peut être reconstruite directement à partir de la chaîne JSON envoyée par le client. Dans la fonction `handle_messages()`, au lieu de parser manuellement les coordonnées, on utilise `serde_json::from_str::<Point>(data)` pour désérialiser les données :

```rust
if let Some(input) = request.strip_prefix("motion ") {
    // Désérialiser motion
    match serde_json::from_str::<Point>(input) {
        Ok(motion) => {
            x += motion.x;
            y += motion.y;
            println!(
                "Mouvement reçu : dx={} dy={} -> Nouvelle position : {:?}, {:?}",
                motion.x, motion.y, x, y )
        }
        Err(e) => {
            eprintln!(
                "Erreur de désérialisation JSON dans '{}': {}",
                input, e
            );
        }
    }
```

Cela simplifie considérablement le traitement des messages, car le serveur peut désormais directement travailler avec des structures de données plutôt que de devoir analyser chaque champ du message.

1.4 Envoi de la position au client

De même, lorsqu'un message de type `position` doit être envoyé au client, le serveur peut sérialiser directement la structure `Point` et envoyer le résultat au client sous forme de message JSON. Au lieu de composer manuellement la chaîne avec les coordonnées, le serveur utilise `serde_json::to_string` pour sérialiser la structure `Point` :

```
// Sérialisation
let position = Point { x, y };
match serde_json::to_string(&position) {
    Ok(json_position) => {
        let reply =
            format!("position {}\n", json_position);
        println!(
            "Réponse au client : {:?}",
            reply.trim()
        );
        output.write_all(reply.as_bytes())?;
        output.flush()?;
    }
    Err(e) => eprintln!(
        "Erreur lors de la sérialisation JSON : {}",
        e
    ),
}
```

Cela garantit une transmission propre et structurée des données, en permettant au client de recevoir un message clair et directement utilisable.

L’utilisation de la sérialisation et de la désérialisation avec `serde` présente plusieurs avantages majeurs :

- **Simplicité :** La gestion des données devient beaucoup plus simple. Il n'est plus nécessaire d'extraire et d'analyser manuellement chaque champ des messages entrants ou sortants.
- **Extensibilité :** Si d'autres types de messages doivent être ajoutés à l'application, il suffira de définir de nouvelles structures et de les qualifier avec `Serialize` et `Deserialize`. Les conversions entre ces structures et les flux JSON se feront automatiquement.
- **Réduction des erreurs :** En utilisant un format standard (JSON) et une bibliothèque fiable (serde), on réduit le risque d'erreurs de manipulation des données, comme les problèmes de format ou d'incohérences entre les différents composants du système.

Resultat dans le terminal :

game_client :

```
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\game_client_backup>
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\game_client_backup> ./run_client.bat data/cat01.ppm localhost 5555
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.08s
args: ["NtvPy.py", "game_client", "data/cat01.ppm", "localhost", "5555"]
Loading image from: data/cat01.ppm
Chargement de l'image à partir de : data/cat01.ppm
Connecting to server at localhost:5555
Connected to server at localhost:5555
800×600@0.033
evt="C" btn=0 key="" (535;254) 800×600
evt="KP" btn=0 key="Right" (535;254) 800×600
motion: Point { x: 10, y: 0 }
json_serialised_motion "{\"x\":10,\"y\":0}"
Message brut reçu (line): position {"x":10,"y":0}

Données position reçues : Point { x: 10, y: 0 }
evt="KR" btn=0 key="Right" (535;254) 800×600
evt="KP" btn=0 key="Escape" (535;254) 800×600
PS C:\Users\ewann\Documents\Cours\S3\programmation\mini_projet2\game_client_backup>
```

```
waiting for request from client...
obtained "motion {\"x\":10,\"y\":0}" from client
input :BufReader { reader: TcpStream { addr: 127.0.0.1:5555, peer: 127.0.0.1:60416, socket: 264 }, buffer: 0/8192 }
Mouvement reçu : dx=10 dy=0 -> Nouvelle position : 10, 0
Réponse au client : "position {\"x\":10,\"y\":0}"
Déconnexion du client
```

game_server :

Etape 11 :

L’objectif est de centraliser la gestion des positions et des images des clients sur le serveur, de manière à permettre à chaque client d’avoir une vue à jour des autres participants et d’interagir avec eux en temps réel. Le serveur gère désormais la communication des informations entre les clients, centralise les données et envoie ces informations à tous les clients connectés pour maintenir une vue synchronisée de l’état du jeu.

1. **Centralisation des données sur le serveur** :

   Le serveur maintient maintenant une structure `ServerState` qui contient toutes les informations nécessaires sur les clients :

   - `clients`: un `HashMap<u32, ClientInfo>`, où chaque identifiant client est associé à des informations sur la position et l'image du client.
   - `positions`: un `HashMap<u32, Point>`, pour gérer les positions des clients.
   - `images`: un `HashMap<u32, Vec<u8>>`, pour stocker les images des clients sous forme binaire.

   Cette centralisation remplace le calcul des déplacements précédemment effectué localement par le client, et le serveur est responsable de maintenir et diffuser les informations actualisées.

2. **Gestion des connexions des clients** :

- Lorsqu’un client se connecte, le serveur lui attribue un identifiant unique (`client_id`) et une position initiale, puis lui envoie cette information sous forme de réponse.
- Le serveur envoie également à tous les autres clients la paire `(identifiant, image, position)` de ce nouveau client afin que les autres clients puissent visualiser le nouveau participant.

  Cela est géré dans le code par la partie suivante :

```
// Enregistrement initial : génération d'un identifiant unique
let client_id;
{
    let mut state = state.lock().unwrap();

    client_id = state.next_id;
    state.next_id += 1;

    // Ajouter le client à l'état global avec des champs vides pour l'image et la position
    state.clients.insert(
        client_id,
        ClientInfo {
            position: Point { x: 0, y: 0 },
            image: Image {
                width: 0,
                height: 0,
                pixels: Vec::new(),
            },
            stream: output.try_clone()?,
        },
    );
}

println!("Client {} connecté, en attente de données.", client_id);
```

  Le serveur attribue un identifiant au client et lui envoie sa position et son image à son arrivée.

3. **Diffusion des informations aux autres clients** :

- Lorsqu'un client envoie un message de déplacement (`motion`), le serveur met à jour la position du client et diffuse cette nouvelle position à tous les autres clients connectés avec la fonction handle_motion.

```rust
    let mut state = state.lock().unwrap();

    if let Some(client) = state.clients.get_mut(&client_id) {
        client.position.x += delta.x;
        client.position.y += delta.y;

        // Limiter la position aux dimensions autorisées (par exemple, 800x600)
        client.position.x = client.position.x.clamp(0, 800);
        client.position.y = client.position.y.clamp(0, 600);

        // Nouvelle position à envoyer
        let new_position = client.position;

        // Envoyer la nouvelle position au client
        let reply = serde_json::to_string(&(client_id, new_position))?;
        client
            .stream
            .write_all(format!("position {}\n", reply).as_bytes())?;
        client.stream.flush()?;

        // Envoyer la nouvelle position à tous les autres clients
        let position_update =
            serde_json::to_string(&(client_id, new_position))?;

        for (&other_id, other_client) in &mut state.clients {
            if other_id != client_id {
                if let Err(e) = other_client.stream.write_all(
                    format!("position {}\n", position_update).as_bytes(),
                ) {
                    eprintln!(
                        "Erreur lors de l'envoi de la mise à jour au client {} : {}",
                        other_id, e
                    );
                }
            }
        }
    }

```

Cette portion de code montre comment le serveur met à jour la position du client et envoie cette mise à jour à tous les autres clients connectés.

4. **Fin de la communication et déconnexion des clients** :

- Lorsqu'un client se déconnecte , le serveur enlève ce client de l'état global (`clients`, `positions`, `images`) et envoie un message de déconnexion à tous les autres clients dans la fonction handle_disconnect.
- Lorsque le client se déconnecte, le serveur notifie tous les autres clients de son départ en envoyant un message de type `client_left`.

  Le code de gestion de la déconnexion :

```rust
fn handle_disconnect(
    client_id: u32,
    state: &SharedServerState,
) -> Result<(), Box<dyn std::error::Error>> {
    let mut state = state.lock().unwrap();

    if state.clients.remove(&client_id).is_some() {
        state.positions.remove(&client_id);
        state.images.remove(&client_id);

        // Diffuser l'identifiant du client partant
        let message = format!("client_left {}\n", client_id);
        for (&other_id, other_client) in &mut state.clients {
            if let Err(e) = other_client.stream.write_all(message.as_bytes())
            {
                eprintln!("Erreur d'envoi au client {} : {}", other_id, e);
            }
        }

        println!("Client {} déconnecté et supprimé.", client_id);
    }

    Ok(())
}
```

Lorsque le client se déconnecte, le serveur notifie tous les autres clients de son départ en envoyant un message de type `client_left`.

Interaction entre les clients et le serveur

1 - Connexion du client :

- Lors de la connexion, le client envoie son image au serveur. Le serveur attribue un identifiant et une position au client, et lui renvoie ces informations. Le serveur envoie également les informations sur ce client à tous les autres clients déjà connectés.

Dans le code, cela se passe dans la fonction `handle_connection`, où les informations sont envoyées après avoir ajouté le client à la liste :

```rust
// Boucle principale : surveiller les messages du client
loop {
    let mut request = String::new();
    let r = input.read_line(&mut request)?;
    if r == 0 {
        // Fin de communication
        handle_disconnect(client_id, &state)?;
        break;
    }

    if let Some(input) = request.strip_prefix("motion ") {
        // Demande de déplacement
        match serde_json::from_str::<Point>(input.trim()) {
            Ok(delta) => {
                let new_position =
                    handle_motion(client_id, delta, &state)?;
                println!(
                    "Client {} déplacé vers la nouvelle position {:?}",
                    client_id, new_position
                );
            }
            Err(e) => {
                eprintln!(
                    "Message de déplacement mal formaté {} : {}",
                    input, e
                );
            }
        }
    }
}
```

2 - **Déplacement du client** :

- Lorsqu’un client demande un déplacement, il envoie la nouvelle position au serveur, qui met à jour sa position et l’envoie à tous les autres clients connectés. Cela est géré dans la fonction `handle_motion.`

3- **Mise à jour des clients avec les informations des autres** :

- Lorsqu'un client reçoit des informations sur un autre client, il doit mettre à jour son état local et redessiner l’écran en conséquence. Les clients mémorisent ces informations dans des structures comme `client_info` et redessinent leur interface graphique lorsque nécessaire.

4- **Déconnexion** :

- Lorsqu’un client se déconnecte, le serveur notifie les autres clients de cette déconnexion, ce qui est géré par la fonction `handle_disconnect` :

1. Gestion des threads pour chaque client

Chaque client est géré dans un thread distinct. Lorsqu'un client se connecte, un nouveau thread est créé pour traiter la communication avec ce client. Cela permet au serveur de gérer simultanément plusieurs connexions sans que l'une d'elles bloque les autres.

2. Partage des données avec les Mutex

Le serveur maintient un état global qui inclut les informations de tous les clients : leur identifiant, image, et position. Cet état est une ressource partagée entre tous les threads. Afin d'éviter les conditions de concurrence (race conditions) lors de l’accès ou de la modification de cet état, le serveur utilise un `Mutex` pour protéger les accès à cette ressource.

3. Synchronisation des accès avec le Mutex

Le Mutex est utilisé principalement dans les endroits où les données partagées doivent être mises à jour par plusieurs threads. Par exemple :

- Lorsqu'un client se déplace, le serveur met à jour la position du client dans `state.positions`. Le Mutex empêche plusieurs threads d'écrire simultanément dans cette structure, ce qui pourrait entraîner des incohérences.
- Lorsqu'un client se connecte ou se déconnecte, les informations sur le client sont ajoutées ou supprimées de `state.clients`, et un Mutex assure que seul un thread à la fois puisse modifier cette structure.

Résultat dans le terminal :

Lorsque l'on exécute deux fichiers game_client (en faisant simplement une copie du premier game_client) avec le fichier game_serveur, on obtient bien les deux images des deux clients sur chaque fenêtre des clients :