Compte rendu

Traitement harmonique des images

Partie 1 : État de l'art

L'histoire de l'harmonisation des couleurs en numérique peut se découper en cinq grandes étapes.

Tout commence au XXe siècle avec des bases théoriques : des chercheurs comme Moon et Spencer, ou encore Granville et Jacobson, cherchent à comprendre pourquoi certaines combinaisons de couleurs nous paraissent plus agréables que d'autres. Ils construisent des modèles mathématiques basés sur la perception humaine et la disposition des teintes sur un cercle chromatique. Ces travaux poseront alors les fondations des premières expérimentations en informatique graphique.

Dans les années 2000, avec l'essor du numérique, on passe à la pratique avec les premiers algorithmes capables d'harmoniser automatiquement les couleurs d'une image. L'idée est d'ajuster les teintes pour qu'elles respectent certaines règles harmoniques prédéfinies. Par exemple, en 2006, Cohen-Or et son équipe proposent un système basé sur les histogrammes de teinte, qui réaligne les couleurs en fonction de modèles d'harmonie. Si cette méthode fonctionne bien pour des images fixes, elle montre vite ses limites dès qu'il s'agit de traiter des vidéos, où il faut maintenir une cohérence d'une image à l'autre (Sawant et Mitra, 2008).

Dans les années 2010, la recherche s'oriente vers des approches plus fines. Plutôt que de transformer uniformément les couleurs, on cherche à analyser l'image en profondeur pour tenir compte de sa structure et de son contexte. On intègre alors la notion de saillance visuelle (Baveye et al., 2013), c'est-à-dire l'importance de certaines couleurs pour l'œil humain. De nouvelles techniques permettent aussi de lisser les transitions entre les couleurs ajustées, afin d'éviter les effets artificiels et la perte de détails (Chamaret et al., 2014 ; Li et al., 2015).

Une avancée majeure arrive ensuite avec les modèles basés sur les palettes de couleurs. Plutôt que de modifier directement les pixels, ces approches extraient une palette représentative de l'image et travaillent sur celle-ci. Cela permet d'avoir des ajustements plus cohérents et de mieux préserver l'esthétique originale. En 2016, Tan et son équipe développent une méthode géométrique pour extraire automatiquement des palettes et décomposer une image en plusieurs couches translucides. D'autres travaux, comme ceux d'Aksoy et Mellado en 2017, affinent encore ces méthodes en intégrant des modèles de mélange des couleurs, rendant les transitions encore plus naturelles.

Enfin, les recherches les plus récentes se concentrent sur la géométrie des espaces colorimétriques. En utilisant des représentations plus sophistiquées comme les espaces LCh ou RGBXY, on atteint des résultats encore plus précis. En 2018, Tan et son équipe présentent une approche permettant de modifier une image en temps réel, tout en respectant les principes d'harmonie des couleurs.

Partie 2 : Librairie d'images

Nous avons décidé de créer notre propre base de code au lieu d'utiliser une librairie d'images.

Il y a deux classes principales. La classe **Vec3** est un vecteur de flottant. Elles contient toutes les méthodes usuelles pour manipuler des vecteurs, ainsi que des fonctions de conversions entre les différents formats de couleur.

```
class Vec3 {
private:
    std::array<float, 3> mVals;
public:

    ... // méthodes et accesseurs

};
```

Et la classe **ImageRGB**, qui représente une image. L'énumération **PixelType** permet de retenir le format actuel des pixel de l'image (RGB, YCrCb, HSV, etc...).

Toutes les opérations sur les images fonctionnent par copie, pour éviter les effets de bords et faciliter la manipulation des images.

```
class ImageRGB
{
protected:
    vector< Vec3 > data;

    PixelType type = PixelType::RGB;

public:
    int w, h;

    ... // méthodes et accesseurs
```

Voici un exemple d'utilisation:

- D'abord, on charge une image. Pour l'instant, seul les formats ppm/pgm sont supportés. C'est prévu d'ajouter les formats classiques (png, jpg, exr, ...).
- Ensuite, on convertit l'image en HSV.
- Ensuite, on utilise la fonction **ImageRGB::apply()**, qui prend en entrée une fonction et l'applique sur chaque pixel de l'image.

  Ici, on augmente juste la teinte de 0.5 (la teinte est dans $\frac{\mathbb{R}}{\mathbb{Z}}$, donc cela équivaut à un décalage de 180°)
- On reconvertit enfin en RGB puis on exporte l'image.

```
int main(void){

    const string input_path = "./img/edgar.ppm";
    const string output_path = "./img_out/";

    // small example
    ImageRGB base = ImageRGB::fromPPM(input_path);

    ImageRGB base_hsv = base.convertTo(PixelType::HSV);

    base_hsv = base_hsv.apply(
        [](const Vec3 & col) -> Vec3 {
            return Vec3(col[0]+0.5, col[1], col[2]);
        }
    );
    // reconversion en rgb
    base_hsv.convertTo(PixelType::RGB).saveAs(output_path + "test1.ppm");

    return 0;
}
```

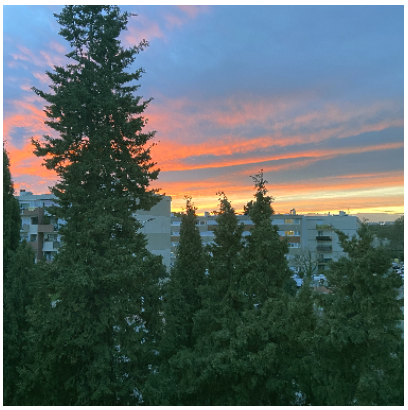

Image de base

Image résultante

La classe image contient par ailleurs des fonctions pour générer les histogrammes et fonctions de répartition, ainsi que pour effectuer des convolutions (par le biais d'une classe Kernel).

```
int main(void){

    const string input_path = "./img/edgar.ppm";
    const string output_path = "./img_out/";

    // small example
    ImageRGB base = ImageRGB::fromPPM(input_path);

    // génère un noyau gaussien de rayon 15 et dont la variance est fonction de 1.0
    Kernel gaussian = KernelUtilities::gaussian(15, 1.0);
    ImageRGB result = base.convolve(gaussian);

    result.saveAs(output_path + "test2.ppm");
    base.saveHistogramData(output_path + "base.dat");

    return 0;
}
```

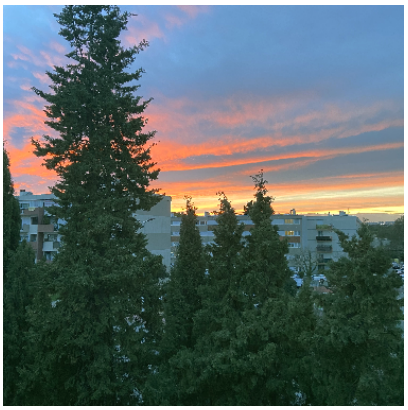

Image de base

Image floutée

En construisant sur cette base, nous pourront mettre en oeuvre les algorithmes d'harminisation que nous avons l'objectif d'implémenter.