Introduction à la prog. multimédia

Chapitre 1 : Les couleurs

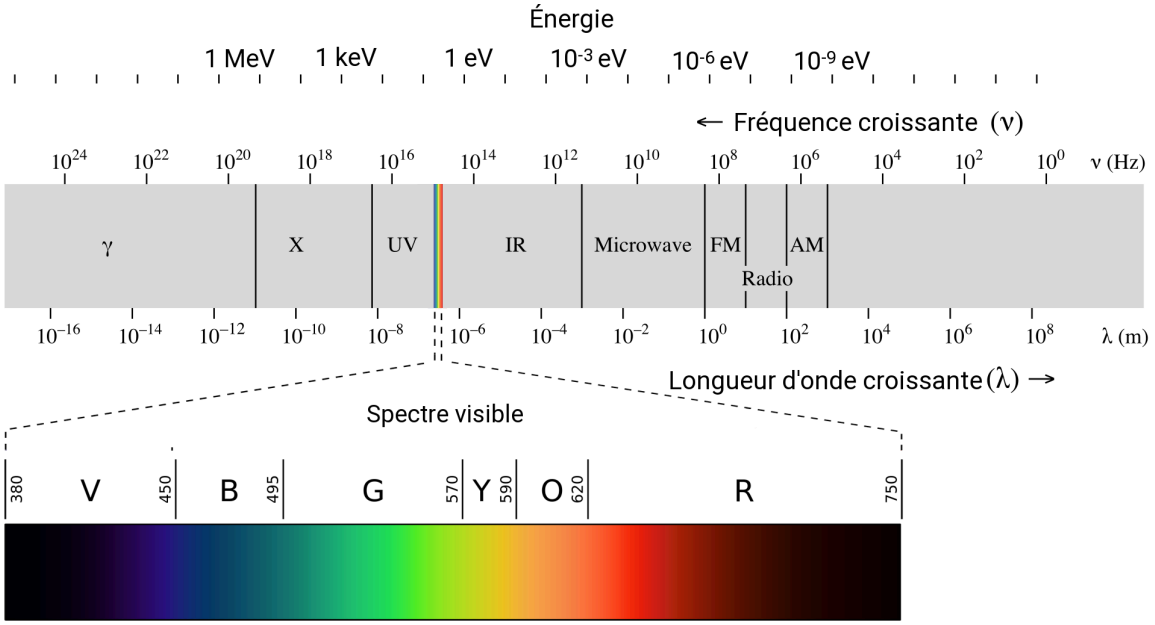
Les couleurs : idées vraies et idées fausses

Un peu de théorie

Les couleurs sont basées sur l'excitation d'un certain nombre de cellules qui tapissent la rétine au fond de l'œil. L'excitation de ces cellules est transmise par voie nerveuse et forme une image mentale dans le cerveau. C'est cette image mentale qui sert de référence.

Une partie de la théorie des couleurs est donc liée à la physique, et une autre partie est liée aux neurosciences (et à la biologie en général).

Ainsi, les couleurs liées à une émission de photos avec une longueur d'onde donnée excitent les trois types de cellules présentes chez l'humain d'une certaine façon, qui donne ce qu'on appelle le spectre visible.

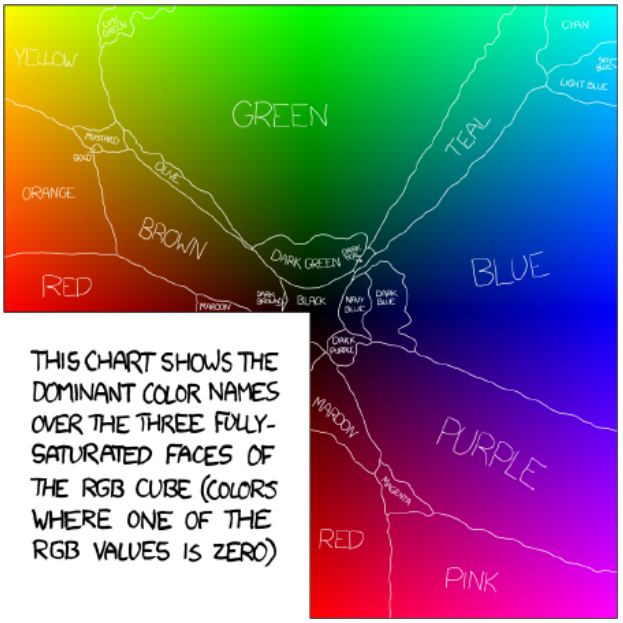


Il y a bien des couleurs, mais pas toutes les couleurs possibles. Toutes les couleurs ne correspondent pas à une seule longueur d'onde. On distingue les couleurs spectrales (émises par le spectre) des couleurs non-spectrales qui sont des sensations composites liées à plusieurs couleurs (donc sensation qui ne peut pas être créée par des lumières monochromatiques : par exemple le violet chez l'humain).

À l'autre bout de la perception, on a des mécanismes biologiques fondés sur des cônes et des bâtonnets (cones & rods). Les bâtonnets servent à la vision en faible luminosité, et ne perçoivent pas les couleurs. Ce sont les cônes qui vont nous intéresser ici. Il y a habituellement trois types de cônes chez les mammifères (et quatre chez les oiseaux, les reptiles, les poissons). Voir Cônes. (NB: il est possible que ce soit quatre chez une partie des humains)

L'ensemble des couleurs possibles dépend évidemment de la vision de la personne qui reçoit l'information. Pour les défauts de perception des couleurs, on peut regarder par exemple https://lesyeuxdudaltonisme.fr/les-types-de-daltonisme/. Si vous n'avez jamais été testé, sachez qu'environ 8% des hommes et 0,5% des femmes ont des défauts de perception des couleurs (le reste de la vision étant normale par ailleurs): essayez https://lesyeuxdudaltonisme.fr/le-test-d-ishihara/ au calme chez vous.

Nommage des couleurs



Le nommage des couleurs est une affaire compliquée depuis longtemps. Agrippa d'Aubigné (auteur du XVIIe siècle) observait qu'il avait plus à voir avec la science du discours que de la physique. C'est aussi une notion très dépendante de la langue. Certaines couleurs ont été nommées tardivement : ainsi dans l'Odyssée, la mer est décrite comme une sorte de rouge sombre, plutôt que bleu. Une tribu namibienne n'a pas de mots pour le bleu, mais plusieurs mots pour le vert. Le mot rose n'a commencé à désigner une couleur qu'au Moyen-Âge pour finir dans les usages au XVIIIe siècle ; le mot *pink* en anglais désignait une couleur laquée (brillante) et existait en vert comme en rouge. En vietnamien, le mot bleu et le mot vert dérivent du même mot (xanh dương = vert océan, xanh lá cây = vert de feuille). Le mot rouge est apparu avant le mot bleu dans toutes les langues. Le mot cyan désigne une nuance de turquoise qui est elle-même une couleur située entre le bleu et le vert. Magenta était un nom de marque avant d'être celui d'une couleur, et certains noms de couleurs restent des marques déposées (le *bleu Klein*, par exemple). Une couleur est plutôt un champ chromatique (imprécis) qu'une référence précise (jusqu'aux normes AFNOR X08-100). Voir par exemple le travail de Randall Munroe sur les noms spontanément attribués à des couleurs par des individus anglophones (on montre des couleurs à un grand nombre d'individus, et on demande leur nom).

Le cerveau nous ment

Même dans le cas où les couleurs sont perçues normalement, le cerveau corrige énormément les choses. C'est le cas notamment de la loi du contraste simultané des couleurs qui dit que:

> *Le ton de deux plages de couleur paraît plus différent lorsqu'on les observe juxtaposées que lorsqu'on les observe séparément, sur un fond neutre commun.*

On distingue plusieurs variantes de cette loi (en luminance ou en teinte), mais ça donne lieu à des illusions d'optiques comme l'image ci-dessous :

Voir aussi Échiquier_d'Adelson.

D’autres surcorrections sont liées au contexte:

http://www.psy.ritsumei.ac.jp/~akitaoka/colorconstancy8e.html

https://www.roman.co.uk/thedress?colour=Royal-Blue

Synthèse additive, synthèse soustractive

Lorsqu’on voit une couleur, on voit en fait un spectre continu du visible dont une partie seulement porte de l’énergie.

La **synthèse additive** est le mélange des couleurs tel que constaté sur des sources de lumières. Lorsqu’on additionne deux lumières, leur énergie s’additionne.

La **synthèse soustractive** est le mélange des couleurs tel que constaté par de la réflexion d’une lumière blanche sur une surface colorée. Parce que l’opération est en fait une réduction d’énergie sur des parties choisies du spectre (donc une multiplication par un nombre inférieur à 1), la synthèse soustractive n’est pas l’exact opposé de la synthèse additive. Toutefois, pour les couleurs primaires et secondaires, les opérations se font en complément.

On utilise traditionnellement 3 couleurs de base en synthèse additive : rouge, vert, bleu. Ces couleurs de base, en se combinant, déterminent un triangle de couleurs que l’on peut obtenir. Avec ces trois couleurs, on ne peut pas obtenir tout le spectre possible, mais comme l’œil humain ne réagit que sur trois longueurs d’onde, on peut assez bien s’en approcher.

En synthèse soustractive, on utilise des couleurs qui vont bloquer une partie du spectre. Il existe trois couleurs fondamentales qui bloquent le rouge, le vert et le bleu. Mais si on bloque intégralement le bleu, on obtient une couleur qui visuellement est définie comme “jaune”. De même, le blocage du rouge est dit “cyan” et le blocage du vert est dit “magenta”.

Pour faire du rouge en synthèse soustractive, il faut bloquer dans la lumière blanche à la fois le bleu et le vert. Mais les encres ou pigments utilisés ne sont pas des pigments idéaux : on ne peut donc pas obtenir des couleurs parfaitement bloquantes. On ne sait même pas bloquer totalement la

lumière (pour faire du noir). Les essais les plus concluants descendent à 0,035% de lumière réfléchie (Vantablack).

Lorsqu'on définit une couleur, on peut donc la définir en synthèse additive, ou négative. Comme nous utiliserons plutôt des écrans, nous nous concentrerons sur la synthèse additive.

Gamut

Le Gamut est l'ensemble des couleurs qui peuvent être produites par un dispositif (synthèse additive ou soustractive). Par exemple, un écran RGB.

https://fr.wikipedia.org/wiki/Gamut

Pour établir le gamut, il faut un appareillage capable de mesure spectrométriques. Un vidéo-projecteur a sûrement un gamut inférieur à un écran. Aucun dispositif n'a un gamut complet.

La standard sRGB est capable de travailler sur une base de trois couleurs avec les fréquences et saturations suivantes :

- Rouge: 612 nm @92 %
- Vert: 547 nm @74 %
- Bleu: 464,5 nm @93 %

Ça dessine dans l'espace des couleurs possibles un triangle, et toutes les combinaisons de couleurs qu'on peut obtenir avec sont à l'intérieur de ce triangle (dans la bonne représentation). Les imprimantes (de bonne qualité) ont normalement un gamut plus élevé.

NB: il existe un format AdobeRGB qui permet un gamut nettement plus étendu, mais son emploi reste fragile : le gamut étendu est en fait transformé dans le gamut sRGB habituel et si on oublie une conversion, les images perdent encore plus de possibilités de couleurs. Vous voulez en savoir plus ? https://www.kenrockwell.com/tech/adobe-rgb.htm

Décomposition des couleurs

Composantes

Les couleurs sont souvent codées par le **modèle** *RGB* dans le domaine du traitement numérique de la couleur. Ce modèle, qui correspond à la synthèse additive décrite ci-dessus, n'est toutefois pas le seul modèle basé sur la synthèse additive de quantité non-négative de lumière (ou soustractive de superposition de pigments bloquants).

Le système de synthèse soustractive correspondant au modèle RGB est le modèle CMY, qui est peu utilisé : on lui préfère le système CMYK (Cyan Magenta Yellow blacK, CMJN en français) qui rajoute la couleur noire (le noir idéal bloquant toute réflexion) qui est donc un peu redondant (on peut fabriquer du noir à partir des trois autres couleurs) ; mais le prix de l'encre noir est tellement inférieur et sa qualité meilleure que par mélange des trois couleurs (le mélange n'est pas idéal, et un noir constitué des trois couleurs primaires sera gris foncé) que le CMYK est plus utilisé. C'est le système qu'on appelle aussi l'impression en **quadrichromie**.

Il est possible d'avoir plus de composantes couleurs, ce qui change alors le gamut généré : d'un triangle, on passe à un quadrilatère avec quatre couleurs (si la quatrième est en dehors du triangle formé par les trois premières), et des polygones avec encore plus de côtés au fur et à mesure qu'on augmente le nombre de couleurs de base. Ces couleurs de base s'appellent des **composantes**.

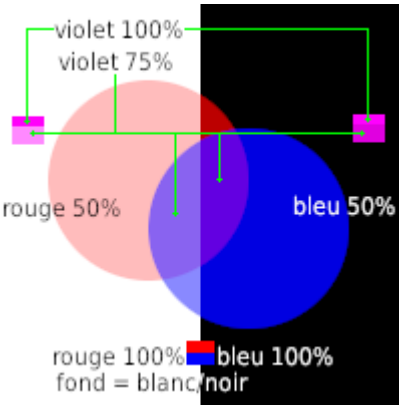
On peut étendre le nombre de composantes notamment dans le domaine de l'impression de façon à se passer le plus possible de l'obtention de couleurs par mélange. Cela permet une grande stabilité de la couleur finale de l'image (notamment dans le cas de l'impression : chaque composante a alors son pigment). C'est ce qui se passe quand on utilise des gammes de couleur comme Pantone® ou Focoltone®, deux marques qui commercialisent des nuanciers (et des encres) pour l'impression professionnelle.

Les valeurs des composantes peuvent être exprimées de plusieurs façons : par une valeur décimale entre 0 et 1, un pourcentage, ou une valeur (souvent contrainte à être entière) entre 0 et une valeur maximale. La plupart des applications utilisent un codage sur 8 bits par composante, ce qui se traduit par des valeurs entre 0 et 255 (c'est le cas en CSS qui utilise une notation connue #RRGGBB avec des chiffres hexadécimaux).

Il est possible d'utiliser une composante dite **d'opacité** ou de **transparence**. C'est plutôt la première, puisque quand cette composante à une valeur maximale, cela représente une couleur complètement opaque.

Mélange de couleurs

Dans les opérations normales, le mélange de couleur se fait par superposition des deux couleurs : si on met du bleu par dessus du rouge, le résultat sera bleu. Cette opération change dès lors que l'on a affaire à des couleurs partiellement transparentes.



Ainsi, dans l'image ci-contre, un cercle rouge 50% (couleur A) et un cercle bleu 50% (couleur B) ont été superposés (le bleu au dessus du rouge, mais ça ne change rien avec la valeur de 50%) et vont donner une couleur violette (50%,0,50%)=#800080 avec une transparence finale de 75% (pour montrer la transparence des couleurs obtenues, on a mis un fond blanc à gauche et noir à droite). La partie violette à 100% a été mise pour montrer la différence entre la couleur pure et la couleur sans transparence.

Les équations de superposition (on dit aussi ***a-compositing*** ou ***a-blending***) sont décrites par https://fr.wikipedia.org/wiki/Alpha_blending. On abrège souvent cette composante A (comme alpha) d'où le modèle RGBA.

Palette

Une façon de gérer les couleurs consiste à avoir une table d'indexation (typiquement un tableau indexé par des nombres) qui contient les couleurs et les pixels référencent non pas la couleur, mais directement le numéro dans la table. La palette peut ainsi être changée facilement.

Pendant longtemps, le matériel a fonctionné de cette façon, et sans changer une image, il est possible d'obtenir des effets avec uniquement le changement de palette. Voici un exemple de cette technique, librement (et mal) repris de http://www.effectgames.com/demos/canvascycle/ (qui fonctionne en Javascript).

Les avantages de l'utilisation d'une palette de couleurs sont :

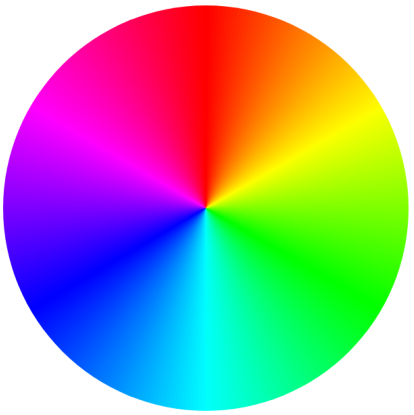
- une économie en quantité d'information : chaque couleur utilisée est codée une fois avec toute l'étendue de ses composantes, puis après chaque pixel (qu'on suppose beaucoup plus nombreux que le nombre de couleurs de la palette) n'est codé qu'avec un numéro qui ne dépasse pas la taille de la palette (exemple typique : couleurs 24 bits, palette 8 bits, chaque pixel occupe trois fois moins de place)
- une nécessité liée au matériau de transfert (par exemple: impression 3D où chaque élément de la palette correspond à une bobine différente de composite, réalisation en crochet, tricot ou tapisserie où les couleurs de la palette représentent des bobines de fil différentes)
- possibilité de nommer une couleur *transparente*, qui sert ainsi à délimiter les contours de l'image réel et de s'affranchir de la forme rectangulaire

- changement de la nature du codage des couleurs (nombre de composantes, type de synthèse, modèle de couleur)

Les inconvénients sont :

- nécessité d'une indirection (lecture dans une table) à chaque lecture ou écriture d'un pixel pour obtenir les composantes de la couleur, ce qui ralentit les opérations de traitement
- les opérations de mélange de deux images, assez courante (opacité partielle avec superposition d'images) génèrent des couleurs supplémentaires qui peuvent être hors de la palette et qui peuvent en faire exploser la taille

Teinte, saturation, valeur



On définit aussi souvent les couleurs par leur teinte, saturation, et valeur.

La teinte provient d'une répartition des couleurs sur la roue des couleurs qui fait alterner couleurs primaires et couleurs secondaires (une couleur primaire est à l'opposé de sa couleur bloquante).

La saturation correspond à la pureté de la teinte. La saturation maximale donne les couleurs primaires ou secondaires, la saturation minimale donne du gris.

Il existe deux systèmes concurrents basés sur la même roue des couleurs. Les différences sont très bien expliquées ici (en particulier le schéma avec le cône et le double cône): https://en.wikipedia.org/wiki/HSL_and_HSV

La traduction française ajoute à la confusion puisque des mots proches en anglais (luminosity, brightness par exemple) ont été traduits par le même mot en français (luminosité). La traduction actuelle est TSL et TSV.

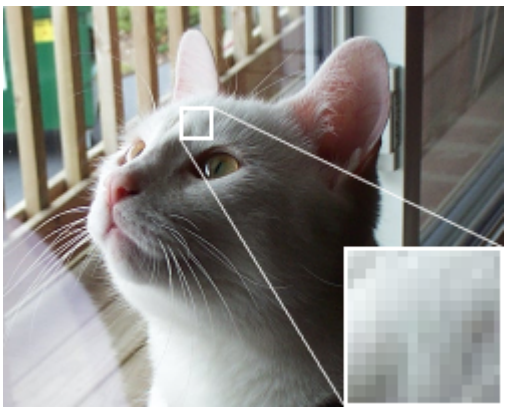
La valeur correspond à la lumière émise (après correction pour être mis sur une échelle linéraire, ce n'est donc pas l'énergie). En effet l'énergie en augmentant linéairement ne change pas la perception de la lumière linéairement : deux fois plus d'énergie ne donnera pas une image deux fois plus lumineuse. Le passage d'une valeur de couleur à une valeur utilisable en électronique repose donc sur des modèles perceptuels de la vision très complexes comme expliqué dans https://fr.wikipedia.org/wiki/Luminosit%C3%A9_(colorim%C3%A9trie). Il existe des formules de conversion (qui ne sont pas à apprendre par cœur), mais qui permettent de travailler en HSL ou HSV même si les dispositifs physiques sous-jacents sont en RGB.

La roue des couleurs est à connaître par cœur.

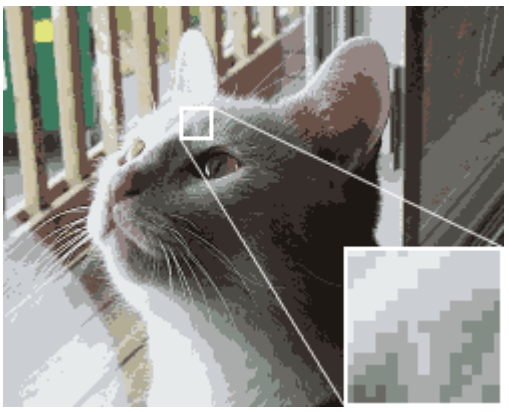
Réduction d'information

Quantization/Thresholding (seuillage)

Lorsque le nombre de couleurs doit être réduit, on peut utiliser plusieurs algorithmes. L'algorithme le plus simple est la quantification. On prend la couleur la plus proche. On peut choisir des couleur qui sont disposés avec une certaine régularité ou être fixée à l'avance (par exemple, la palette basique du web de 16 couleurs ou la palette des couleurs prédéfinies en SVG, voir la page https://fr.wikipedia.org/wiki/Couleur_du_Web). Cette approche présente beaucoup de défauts visuels.

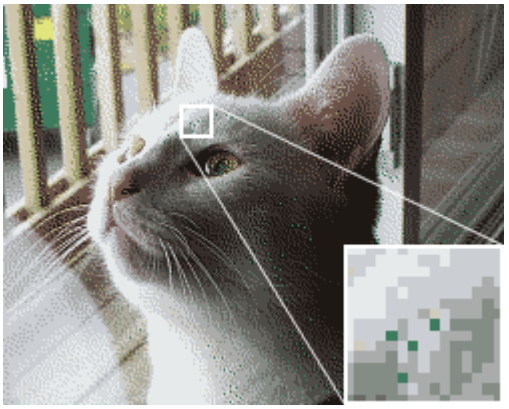


wikimedia.org/wikipedia/commons/e/e3/Dithering_example_undithered.png



.../2/2d/Dithering_example_undithered_16color.png

Dithering (diffusion d'erreur)



Le dithering est une technique utilisée en informatique graphique pour atténuer les effets de la réduction de la profondeur de couleur ou de la résolution d'une image. Lorsqu'une image est convertie en une palette de couleurs réduite ou affichée sur un écran avec une résolution inférieure, des variations subtiles de couleur peuvent être perdues, entraînant une apparence dégradée.

Le dithering résout ce problème en introduisant délibérément du bruit dans l'image. Plutôt que de représenter une couleur pure, chaque pixel peut contenir une combinaison de couleurs de la palette avoisinante. Cela crée une illusion optique qui permet de reproduire des teintes et des nuances qui seraient autrement perdues dans une représentation plus simplifiée.

Il existe différentes méthodes de dithering, mais l'objectif commun est de préserver autant que possible l'apparence générale de l'image tout en réduisant la quantité d'informations nécessaires pour la représenter. Le dithering est couramment utilisé dans la conversion d'images pour des affichages à faible résolution, la création de GIF animés et d'autres contextes où la fidélité des couleurs doit être préservée malgré des limitations techniques.

L'algorithme de Floyd-Steinberg est l'un des plus simples : on prend séquentiellement tous les pixels et on applique à chacun l'algorithme suivant :

- On prend la couleur la plus proche
- On mesure la différence de couleurs entre l'original et la nouvelle valeur
- On reporte 7/16 de la différence sur le pixel de droite
- On reporte 3/16 de la différence sur le pixel en dessous à gauche
- On reporte 5/16 de la différence sur le pixel en dessous
- On reporte 1/16 de la différence sur le pixel en dessous à droite

Sur les bords, la différence peut être perdue, ou bien on utilise un masque différent pour les bords.

Mise en pratique : codage des couleurs en python

On va voir comment les couleurs sont gérées dans pygame, pillow et colormath. Dans les deux premières bibliothèques, les couleurs sont simplement des couleurs RGB ou RGBA et sont données sous la forme d'un triplet ou quadruplet d'entiers entre 0 et 255, par exemple ci-dessous avec pygame.

```
import pygame

# Initialisation de Pygame
pygame.init()

# Création d'une fenêtre
screen = pygame.display.set_mode((800, 600))

# Définition des couleurs
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
TRANSPARENT_RED = (255, 0, 0, 128)

# Remplissage de l'écran avec une couleur
screen.fill(WHITE)

# Dessin de formes avec des couleurs RGB
pygame.draw.rect(screen, RED, (50, 50, 100, 100))  # Un rectangle rouge
pygame.draw.circle(screen, GREEN, (200, 200), 50)  # Un cercle vert
pygame.draw.line(screen, BLUE, (300, 300), (400, 400), 5)  # Une ligne bleue

# Création d'une surface avec une couleur RGBA (transparente)
surface = pygame.Surface((100, 100), pygame.SRCALPHA)
surface.fill(TRANSPARENT_RED)
screen.blit(surface, (500, 50))  # Dessin de la surface transparente sur l'écran

# Mise à jour de l'affichage
pygame.display.flip()

# Boucle principale pour garder la fenêtre ouverte
running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

# Quitter Pygame
pygame.quit()
```

La bibliothèque colormath permet de définir des couleurs dans un modèle et de faire une conversion de tous les modèles vers tous les autres (ou presque) : RGB, RGBA, sRGB, AdobeRGB,

HSL, HSV, spectre lumineux… Attention, par défaut, les valeurs sont entre 0 et 1, sauf à utiliser les méthodes avec *upscaled* dans leur nom ou en paramètre. On peut en trouver la documentation ici : https://python-colormath.readthedocs.io/en/latest/color_objects.html

```
from colormath.color_objects import sRGBColor, LabColor
from colormath.color_conversions import convert_color
from colormath.color_diff import delta_e_cie2000

rgb_color1 = sRGBColor(255, 0, 0)
hsv_color = convert_color(rgb_color1, HSVColor)
print(hsv_color)
hsl_color = HSLColor(0, 1, 0.5)  # Rouge pur

rgb_color2 = convert_color(hsl_color, sRGBColor)
print(rgb_color2)
print(rgb_color2.get_upscaled_value_tuple())

lab_color1 = convert_color(rgb_color1, LabColor)
lab_color2 = convert_color(rgb_color2, LabColor)
# Calculer la distance de couleur Delta E
delta_e = delta_e_cie2000(lab_color1, lab_color2)
print(f"Delta E entre les deux couleurs : {delta_e}")
```

Objectifs testables

Les élèves doivent savoir à l'issue de ce cours :

- Exprimer les 8 couleurs de base dans les espaces RGB (pourcentage, valeur entière ou codage hexadécimal), CMY ou CMYK, HSV
- Assigner des couleurs correspondant à un nom de couleur donné textuellement, ou inversement (notamment identifier la couleur primaire la plus proche d'une couleur donnée)
- Repérer les défauts de perception des couleurs (daltonisme, loi de contraste simultané des couleurs, limitation du gamut par exemple) et si possible les compenser
- Identifier les effets du mélange de couleurs, y compris avec les couleurs transparentes
- Expliquer l'effet de la réduction du nombre de couleurs sur la qualité de l'image et le principe d'une palette.
- Vérifier la qualité d'un algorithme de diffusion d'erreur (conservation de la couleur moyenne sur plusieurs pixels)

Chapitre 2 : les images bitmap

Les image point à point

Les **images bitmap** (on traduit parfois par *point à point*) sont des images qui permettent de manipuler des collections de points de couleurs. Elles sont généralement arrangées en grilles rectangulaires régulières de **pixels** carrés, mais la forme n'est pas une obligation (ni le carré, ni la grille rectangulaire).

Chaque image est définie par sa taille (sous forme d'une largeur et d'une hauteur), sa profondeur (liée au modèle de couleur), et parfois son espace de couleur.

La plupart des images sont accompagnées de **métadonnées** qui permettent de véhiculer d'autres informations, comme la résolution d'intention (à quelle taille physique chaque pixel est censé correspondre), les droits attachés (auteur, licence), les conditions de prise de vue.

pygame comme pillow (bibliothèque de manipulation d'image invoquée sous le nom `PIL`) sont deux bibliothèques python qui permettent de manipuler des images, mais qui n'utilisent pas le même système de codage. Toutefois, il est simple de transformer une image PIL en une image pygame et réciproquement.

Construction des images

Les images bitmap utilisent des systèmes de coordonnées à deux dimensions appelées couramment *abscisse et ordonnée* ou *largeur et hauteur*. L'abscisse est en général le premier axe et définit l'horizontale ; l'ordonnée est le second axe et définit la verticale. Les images bitmap que nous allons évoquer seront toujours rectangulaires. Le système de coordonnées est presque toujours le *système de coordonnées écran*, qui va de gauche à droite pour l'abscisse et de haut en bas pour l'ordonnée, avec le point de coordonnées (0,0) qui est donc en haut à gauche.

Stockage des images

En mémoire, le stockage des images se fait en utilisant l'abstraction d'un tableau bidimensionnel qui est donc accessible par des méthodes de type `image[x][y]` ou `image.get(x,y)` même si le stockage réel peut être différent. En plus de cela, il faut stocker plusieurs autres informations : au minimum, la largeur et la hauteur de l'image, mais également les éventuelles métadonnées.

Le stockage sur disque se fait à travers un format de fichier bitmap. Il en existe des dizaines, mais les principaux sont les formats JPG, GIF, PNG, BMP, PBM/PNM (portable bitmap)

Le format PBM/PNM

Ce format est le plus simple possible. L'entête du fichier (les premiers octets) doivent être le codage ASCII de "P1" à "P6" selon la variante utilisée (noir et blanc, niveaux de gris, , puis ensuite quelques informations en ASCII (largeur, hauteur, valeur maximale des composantes couleurs), puis une suite de nombres codés soit en binaire ou en ASCII (selon la variante). Il n'y a pas de métadonnées structurées. Par exemple, le fichier contenant uniquement la chaîne "`P1 3 3 010111010`" est un fichier valide (qui décrit le symbole + dans une grille 3x3).

Le format BMP

À ne pas confondre avec le précédent, c'est un format utilisé par l'écosystème Microsoft uniquement. Il est similaire au format P6, mais peut contenir plus de métadonnées pour un meilleur rendu : résolution d'intention, profil colorimétrique, plusieurs variantes pour accommoder diverses résolutions de couleurs (palette, noir et blanc, niveaux de gris, couleur 8/composante, etc.) Il ne supporte que difficilement la transparence partielle ou les formes non rectangulaires (masque). Il ne supporte pas l'animation. Il supporte des algorithmes simples de compression sans perte.

Le format PNG

Ce format supporte à peu près tout ce qu'on veut à part la compression avec perte d'information. Transparence partielle, compression, métadonnées riches, profils colorimétriques, tout peut s'y retrouver. C'est le format à utiliser préférentiellement pour toutes les images à échanger qui n'utilisent pas de compression avec perte : icônes, vignettes, dessins au trait (même si on verra que les images vectorielles sont meilleures pour ce dernier usage).

Le format GIF

C'est un format ancien, qui permettait de faire des choses que le format PNG permet maintenant de faire mieux : animation, transparence. Les métadonnées sont peu nombreuses. Longtemps, il a été contraint à des couleurs par palette limitée à 256 couleurs. Il a également été gêné dans son développement par l'utilisation d'un algorithme breveté pour le stockage des pixels (qui a poussé au développement du format concurrent PNG).

Le format JPEG

La compression JPEG fonctionne en divisant l'image en blocs de 8x8 pixels. Chaque bloc est transformé par une transformation en cosinus discrète (DCT), qui convertit les valeurs des pixels en une série de coefficients de fréquence. Ces coefficients sont ensuite quantifiés, c'est-à-dire qu'ils sont arrondis à des valeurs plus simples, ce qui réduit la quantité de données nécessaires mais introduit une perte d'information. Les blocs quantifiés sont ensuite compressés à l'aide de la méthode de codage entropique (comme le codage de Huffman).

Le principal avantage des fichiers JPEG est la taille occupée par les images photographiques. Cette taille est en plus paramétrable (par un facteur de qualité qui va de 0 à 100) selon les usages nécessaires. La norme JPEG 2000 n'a pas gagné beaucoup de popularité, et les images photographiques circulent encore essentiellement en JPEG classique ou en format RAW (données brutes des capteurs des appareils photographiques, pas réellement un format d'image exploitable).

Les métadonnées

Les métadonnées sont des données qui accompagnent l'image, mais ne sont pas strictement nécessaires à son interprétation (bien qu'elles puissent la préciser). Elles contiennent souvent des informations telles que la date et l'heure de la prise de vue, les paramètres de l'appareil photo (comme l'exposition, la sensibilité ISO, l'ouverture), parfois les coordonnées GPS du lieu où la photo a été prise, ainsi que des informations sur le copyright et les droits d'utilisation. Les métadonnées peuvent également contenir des commentaires, des descriptions, et des mots-clés facilitant l'organisation et la recherche des images.

Les standards reconnus sont EXIF (Exchangeable Image File Format), IPTC (International Press Telecommunications Council), et XMP (eXtensible Metadata Platform) sont couramment utilisés pour structurer et stocker ces informations. Les métadonnées jouent un rôle crucial dans la gestion et l'archivage des images, améliorant leur traçabilité et leur utilité dans divers contextes professionnels et personnels. Le standard EXIF est le plus courant et un certain nombre de champs sont automatiquement remplis par les appareils photos. Attention, toute photo qui garde des métadonnées est susceptible de contenir des données sensibles ou personnelles (notamment la position GPS au moment de la prise de vue), même si celle-ci peut être facilement modifiée ou supprimée (il n'y a pas de signature fiable des EXIF).

DÉCOUPAGES ET COLLAGES

Toutes les bibliothèques permettent de découper un morceau rectangulaire (aligné horizontalement et verticalement) d'une image (en anglais *crop*). Elles permettent aussi de coller une image dans une autre. Combinée à la création d'images vides, cela permet de découper selon des rectangles n'importe quelle image.

Attention, les coordonnées des rectangles avec Pillow sont des coordonnées de type (x1,y1,x2,y2) avec (x1,y1) le coin supérieur gauche et (x2,y2) le coin inférieur droit non inclus. Mais pour les

opérations de dessin, le coin inférieur droit est inclus. Avec pygame, les coordonnées sont de type Rect((x1,y1),(w,h)).

```
from PIL import Image
image = Image.open('test.png')
width, height = image.size
midpoint = width // 2
left_part = image.crop((0, 0, midpoint, height))
right_part = image.crop((midpoint, 0, width, height))
new_image = Image.new('RGB', (width, height))
new_image.paste(right_part, (0, 0))
new_image.paste(left_part, (midpoint, 0))
new_image.save('swapped_test.png')
new_image.show()
```

Décomposition en composantes de base

Les images manipulées peuvent être facilement décomposées en leurs composantes de base : une image “RGB” possède 3 composantes de base (“R”, “G” et “B”), une image “RGBA” en possède 4. Les composantes sont des images elle-même, de type “L” (niveaux de gris). Les opérations qui permettent de passer de l’un à l’autre sont img.split() => liste de composantes et Image.merge(“RGB”,composante1,composante2,composante3) pour le mouvement inverse.

Chaque composante peut être donc traitée indépendamment des autres par n’importe quel procédé, y compris fabriqué à partir de d’un traitement point par point :

```
zeroed_band = original.point(lambda _: 0)
```

L’exemple ci-dessus transforme une composante en une nouvelle composante de même taille où chaque valeur de point vaut 0 au lieu de sa valeur initiale (notez la syntaxe de fonction `lambda _ : f(_)` qui est utilisée ici).

Formes élémentaires

On peut faire des tracés élémentaires : rectangles, ellipses ou cercles, lignes et texte. C’est le domaine du module ImageDraw de Pillow. Un exemple qui utilise ces possibiilités :

```
from PIL import Image, ImageDraw, ImageFont
text = "test.png"
image = Image.open(text)
width, height = image.size
draw = ImageDraw.Draw(image)
font = ImageFont.load_default()
text_x = width // 2
text_y = height - 20
text_bbox = draw.textbbox((text_x, text_y), text, font=font, anchor="mm")
draw.rectangle(text_bbox, outline="red", fill="blue")
draw.text((text_x, text_y), text, font=font, fill="white", anchor="mm")
image.save('test_with_text_and_textbbox.png')
```

Transformations affines

Il y a deux types de transformations : celles qui préservent totalement l'information, et qui sont donc réversibles, et celles qui perdent de l'information. Les transformations réversibles sont peu nombreuses : symétrie verticale, symétrie horizontale, rotations autour du centre de l'image à 90, 180 et 270 degrés. Elles se font via une méthode spécifique `transpose`. Toute autre transformation perdra de l'information sauf, à la rigueur, les agrandissements via des nombres entiers (mais pas les contractions). Une fois que l'information a été mélangée, il n'est plus possible de la récupérer intégralement (et notamment, on ne peut pas "améliorer" une image en zoomant sur une partie spécifique ; quand l'information n'est pas là, on ne peut que l'inventer).

Les autres opérations affines se font donc à partir des méthodes rotate et resize (ou la version plus générale, qui permet de faire d'autres transformations comme la transformation quadrilatère, qui permet d'envoyer un quadrilatère quelconque sur l'image).

```
image = Image.open('test.png')
image = image.resize((image.width // 3, image.height // 3))
image = image.rotate(45, expand=True)
image.show()
image = image.rotate(-45, expand=True)
image = image.resize((image.width * 3, image.height * 3))
image.show()
```

Opérations mathématiques sur les images

Additions et soustractions

Puisqu'une image peut être considérée comme un tableau bidimensionnel de tuples (ou de valeurs simples dans le cas d'une seule composante), il est facile d'importer une image dans numpy (bibliothèque spécialisée de calcul) et de la réimporter comme image ensuite.

Ce cours ne se veut pas être un cours d'utilisation de numpy, mais certains usages basiques sont simples :

```
from PIL import Image
import numpy as np

image1 = Image.open('test1.png')
image2 = Image.open('test2.png')

arr1 = np.array(image1)
arr2 = np.array(image2)

diff = np.abs(arr1 - arr2)
mask = np.any(diff > 5, axis=-1)

highlighted_diff = np.zeros_like(arr1)
highlighted_diff[mask] = [255, 0, 0]

result = Image.fromarray(highlighted_diff)
result.show()
```

Ce script produit une image qui est constituée de zéros sauf dans les endroits très différents entre deux images.

Attention : les valeurs lors de la reconversion en image doivent être dans la gamme [0,255] pour chaque composante. Les valeurs négatives ou trop grandes sont étêtées.

Utilisation de ces images : la soustraction de deux images permet de percevoir les différences, par exemple. De façon plus générale, on peut par exemple essayer de voir la façon dont Porter et Duff ont définies les façons de faire interagir deux images entre elles (opérations reprises par Photoshop et tous les produits Adobe, ainsi que Gimp) : https://ssp.impulsetrain.com/porterduff.html

Quelle est l'opération qui permet de faire cette opération d'image ?

Réponse : ici, on a gardé la couleur rouge présente partout dans la forme 1, et on a additionné la couleur de la forme 2 là où la forme 1 était présente. Il s'agit donc ici d'une addition des canaux couleurs là où la forme 1 est opaque. Cette opération est appelée "Exclusion" dans Gimp.

Opérateurs locaux

Une opération que l'on utilise beaucoup est de construire une nouvelle image en prenant une opération qui permette de construire chaque point d'une nouvelle image à partir des points voisins.

Par exemple, si on construit chaque point comme étant la moyenne de lui-même et des huit points environnants, on va gommer une partie des différences entre chaque point et créer un effet de flou (c'est ce qu'on appelle le BoxBlur, de rayon 1 ici). Il existe d'autres styles d'opérateurs que l'on peut appliquer : flou gaussien par exemple.

Les règles qu'il faut retenir est que si la somme des coefficients est égale à 1, la valeur globale est préservée ; si la somme n'est pas 1, on a d'autres effets. Par exemple, les opérateurs de Sobel ou de Prewitt peuvent être utilisés pour faire de la détection de bords, ou pour faire de l'amélioration d'image (sharpening) comme mentionné dans cet article de Wikipédia : https://en.wikipedia.org/wiki/Unsharp_masking

Objectifs testables

Les élèves doivent savoir à l'issue de ce cours :

- Charger en mémoire des fichiers images ainsi que sauvegarder une image
- Fabriquer une image bitmap comportant des éléments géométriques simples
- Lire et interpréter des métadonnées d'images photographiques
- Faire des modifications élémentaires des images : découpage, collage, transformations isométriques
- Maîtriser les changements d'échelle et leurs effets
- Utiliser la décomposition et recomposition des images bitmaps en composantes couleurs
- Faire des opérations mathématiques sur les images : soustractions, opérateurs locaux (flou gaussien, par exemple)

Chapitre 3 : les images vectorielles

Usage des images vectorielles

Les images vectorielles s'appuient sur une description mathématique des images, et leur représentation se fait au moment de la visualisation pour être adaptée au mieux à l'outil de représentation (*device*) (la plupart du temps un écran, mais ça peut aussi être une imprimante, alors appelée *traceuse*). Les images vectorielles sont donc à favoriser pour les images synthétiques.

Le passage des formules mathématiques à une image adaptée au support final s'appelle le rendu (*rendering*). La durée du rendu peut être un obstacle à l'utilisation d'images vectorielles mais les cas simples sont calculées très rapidement.

Attention : ce qu'on appelle les images de synthèse sont fabriquées selon des formules mathématiques aussi, mais sont souvent fabriquées à l'avance. En effet, les méthodes de calcul de luminosité, de diffusion et de réfraction de la lumière dans des matériaux (semi-)transparents ou réfléchissants (verre, miroirs, nuages) entraînent des temps de calcul très longs.

Nous étudierons dans le cadre de ce cours uniquement les images 2D qui utiliseront essentiellement un seul format de description : le format SVG. Pour la partie expérimentale, nous utiliserons DrawSVG.py.

Construction arborescente

Les éléments qui servent à fabriquer une image peuvent être vus comme disposés de façon arborescente : on a un objet principal qui est un groupe, qui représente une série d'éléments, et ces éléments peuvent être des primitives de base (ligne, arc, etc.) ou bien un groupe, qui lui-même pourra contenir d'autres primitives, etc. Les groupes peuvent avoir des propriétés par défaut, qui sont alors transmises à tous les éléments qui les composent ; et chaque élément a un certain nombre de propriétés qui permettent de le définir (par exemple, un cercle va avoir un centre, un rayon, une épaisseur de trait, une couleur de remplissage, une couleur de tracé…).

Cette construction arborescente se prête très bien à utiliser un format de la famille de HTML. Ce format s'appelle SVG (*Scalar Vector Graphics*).

Les premières versions des images vectorielles étaient plus linéaires (Postscript, PDF) mais avaient déjà cette propriété de pouvoir empiler des contextes qui étaient utilisés un certain temps, puis détruits (cela correspond au parcours en profondeur d'un arbre).

Les repères

Les images vectorielles se basent sur des coordonnées qui s'expriment dans un repère. Le premier repère (et donc le premier système de coordonnées) est celui du support de rendu (*system coordinates, base coordinates, device coordinates*). Mais il est possible de prendre une partie d'une image (un groupe), et de changer son système de coordonnées temporairement (qui n'a plus besoin d'être orthogonal).

Les composants d'une image vectorielle

Les éléments

Les éléments sont communs à presque toutes les formes d'images vectorielles :

- Les cercles et les ellipses

- Les lignes
- Les rectangles
- Les polygones et les polylignes (polygone qui n'est pas fermé)
- Les chemins complexes
- Les groupes

Il en existe d'autres, mais on ne les verra pas.

Chaque élément a des attributs qui permettent de le caractériser. Par exemple

```
<circle stroke="red" stroke-width="5" cx="30" cy="30" r="20" fill="white"/>
```

est un cercle qui est tracé en rouge de 5 px d'épaisseur, rempli en blanc, centré sur (30,30) et de rayon 20.

Les chemins complexes sont des suites de petits éléments (qu'on va appeler *directives*), qui permettent de construire un chemin graphique qui partagera les mêmes caractéristiques tout du long.

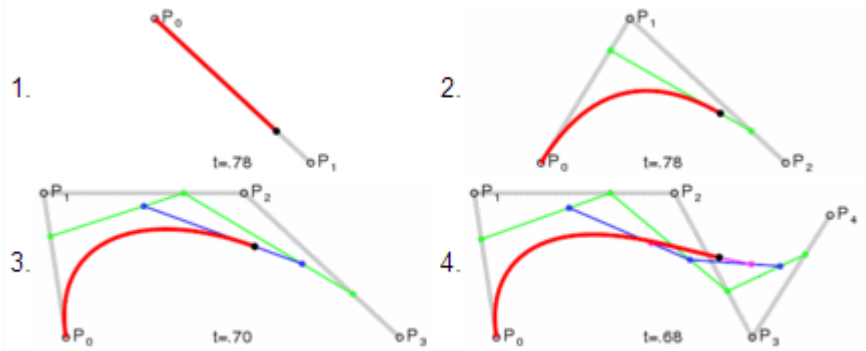
Ces directives sont : des déplacements, des lignes, des arcs de cercle ou d'ellipse (déterminés non par centre et rayons, mais par deux points et rayons). Et des courbes de Bézier quadratiques ou cubiques.

Les courbes de Bézier

$$P(t) = \sum_{i=0}^{n} B_i^n(t) \cdot \mathbf{P}_i$$

$$B_i^m(u) = \binom{m}{i} u^i (1-u)^{m-i}$$

Les courbes de Bézier ont été conçues par Pierre Bézier pour lui permettre de dessiner les profils des voitures sur lesquelles il travaillait (chez Renault). Ce sont des courbes paramétriques construites à partir de n+1 points pour une courbe de degré n. Leur construction fait qu'au temps t=0, la courbe est au point P0, et au temps t=1, elle est au point Pn, et entre les deux, elle se dirige vers P1, puis P2, etc. en étant plus ou moins tiré vers chaque point (les illustrations ci-dessous sont tirées de Wikipédia). Les points intermédiaires sont appelés points de contrôle, et nous serons concernés essentiellement par le cas 2 et 3 (courbes quadratiques et cubiques). Ces courbes présentent de très bonnes propriétés de continuité et dérivabilité et dans le cas de courbes de degré 2 et 3, il est facile d'interpréter le ou les points intermédiaires comme donnant les "directions" de la courbe à partir du départ ou de l'arrivée. C'est donc un outil intuitif pour faire des courbes harmonieuses. Elles sont de plus faciles à tracer du point de vue informatique.

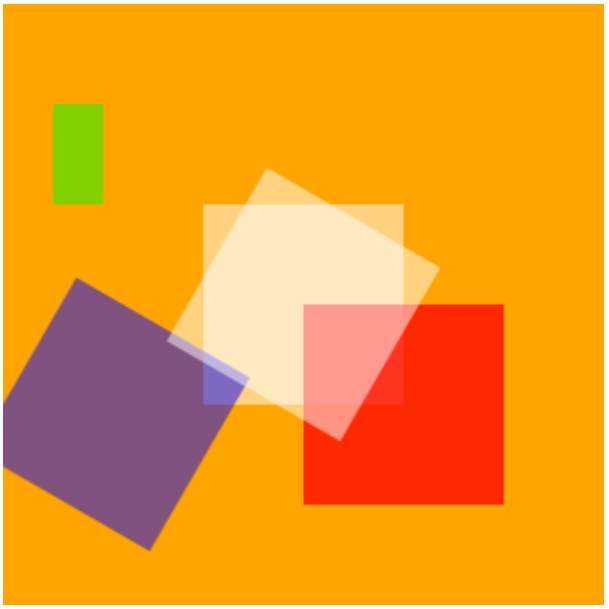


Les caractères

Un cours entier serait possible sur les caractères et les polices de caractères. Le minimum à savoir est : une police de caractère est une collection de glyphes, qui sont des chemins composés de

séries de droites et de courbes de Bézier (cubiques pour les polices Postscript, quadratiques pour les polices TrueType). Les glyphes sont sélectionnés en fonction des caractères qui forment le texte à écrire, et peuvent être répartis en ligne droite ou sur un chemin complexe. Il est possible de faire appel à des polices de caractères externes, mais pour des raisons légales il est toujours mieux de les héberger soi-même. SVG utilise les mêmes distinctions de style de police que CSS, c'est-à-dire decoration, weight, style. Pour voir plus, lire la documentation : https://cduck.github.io/drawsvg/

Les transformations



Il existe trois transformations basiques : agrandissement, rotation, translation. À ces trois là s'ajoutent la transformation quelconque (matrice de transformation), et les déformations latérales (skewX et skewY). On ne verra pas les deux dernières.

Lorsque plusieurs transformations sont spécifiées, elles sont enchaînées séquentiellement : on fait la première transformation, puis la deuxième, puis la troisième. Mais attention : comme le repère est transformé, une translation (par exemple translate(100,0)) après une rotation (rotate(45)) sera une translation dans le repère tourné, pas dans le repère d'origine.

Les transformations par matrice utilisent des coordonnées homogènes, c'est-à-dire qu'un point est représenté par le vecteur (x,y,1) et une matrice est avec six chiffres seulement, tels que

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \times \begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} ax+by+c \\ dx+ey+f \\ 1 \end{pmatrix}$$

Ces matrices permettent de coder toute composition des trois éléments de base. Dans ce cours, il est recommandé de toujours décomposer en les trois éléments de base (rotation, translation, mise à l'échelle).

Attention : les rotations sont dans le sens trigonométrique, mais comme le vecteur des ordonnées est orienté vers le bas, un angle positif correspond à une rotation dans le sens des aiguilles d'une montre (ce qui est le contraire de ce qui est enseigné au niveau lycée).

Attention : les rotations se font à partir des coordonnées (0,0), ce qui peut être un problème. Il existe une propriété `transform-origin` (`transform_origin` en drawsvg) avec deux nombres séparés par des espaces qui peuvent modifier le point de départ des transformations.

```
import drawsvg as draw
d = draw.Drawing(300,300, origin=(0,0))
d.append(draw.Rectangle(0, 0, 300,300, fill='orange'))
d.append(draw.Rectangle(100, 100, 100,100,fill='rgba(255,255,255,.5)'))
d.append(draw.Rectangle(100, 100,
100,100,fill='rgba(255,0,0,.5)',transform='translate(50,50)'))
d.append(draw.Rectangle(100, 100,
100,100,fill='rgba(255,0,0,.5)',transform='translate(50,50)'))
d.append(draw.Rectangle(100, 100,
```

```
100,100,fill='rgba(0,255,0,.5)',transform='scale(.25,0.5)'))
d.append(draw.Rectangle(100, 100,
100,100,fill='rgba(0,0,255,.5)',transform='rotate(30)'))
d.append(draw.Rectangle(100, 100,
100,100,fill='rgba(255,255,255,.5)',transform_origin="150
150",transform='rotate(30)'))
display(d)
```

Intégration dans du HTML

Par nature, SVG est compatible avec HTML. Il existe (au moins) six façons d'intégrer du SVG dans un fichier HTML.

L'usage de `embed`, `frame` ou `iframe` n'est pas recommandé car ce n'est pas standardisé par le W3C (pour `embed`), ou complique l'interaction avec Javascript (dépend du navigateur, cette phrase peut donc devenir facilement obsolète, ou redevenir vraie de façon embarrassante dans des contextes particuliers).

L'usage de `object` permet d'utiliser un fichier SVG externe, tout en autorisant l'accès au DOM pour par exemple manipuler le SVG avec Javascript. C'est la meilleure solution. En plus, le contenu interne de la balise `object` peut être utilisé au cas où le navigateur ne comprend pas le format SVG.

```
<object type="image/svg+xml" data="image.svg" width="300" height="200">
  <img src="image.png" alt="Schéma décrivant X" />
</object>
```

L'usage de la balise `img` est très suffisant pour les images SVG statiques (logos, par exemple) :

```
<img src="logo.svg" alt="Stark Company" />
```

Et enfin, il est tout à fait possible de mettre du SVG directement dans un fichier HTML :

```
<p>Regardez ce schéma :</p>
<svg width="100" height="100" xmlns="http://www.w3.org/2000/svg">
  <circle cx="50" cy="50" r="40" stroke="black" stroke-width="3" fill="red" />
</svg>
```

Objectifs testables

Les élèves doivent savoir à l'issue de ce cours :

- Connecter une description mathématique d'une image et l'image elle-même
- Sélectionner le meilleur format d'image pour une utilisation donnée
- Utiliser les formes élémentaires : rectangles, lignes, arcs, courbes ainsi que le tracé ou le remplissage des formes
- Faire des motifs réutilisables et intégrer des éléments de programmation à leurs tracés
- Utiliser les multiples systèmes de coordonnées : absolues, relatives, changement de repère
- Exporter leurs productions au format SVG et les intégrer dans une page web

Chapitre 4 : Animations

Objectifs testables

Les élèves doivent savoir à l'issue de ce cours :

- Comprendre le principe de l'animation image par image.
- Intégrer la gestion d'événements à une animation sous forme de mini-jeux.
- Construire un élément utilisant son propre comportement dans un ensemble.
- Assigner des sprites correspondant au déroulement d'un comportement.
- Faire le lien entre un modèle physique et la simulation informatique en prenant en compte les modèles à temps variable entre les itérations.