Rapport technique

Ambilight

Léa Brito

Clara Soncin

Salma El Miz

Alexandre Bebon

G2.01

2021-2022

Table des matières

Introduction :

Un système Ambilight désigne un type d'écran qui vise à procurer une expérience immersive à l'utilisateur. Pour cela, des LEDs placées derrière l'écran permettent de prolonger les couleurs du bord de l'image affichée. Ce système est vendu intégré à certains modèles de téléviseurs (Sony, Phillips…).
Nous proposons de mettre en place un système semblable dans le cadre des projets IéTI, et ce document fait état de synthèse de notre projet.

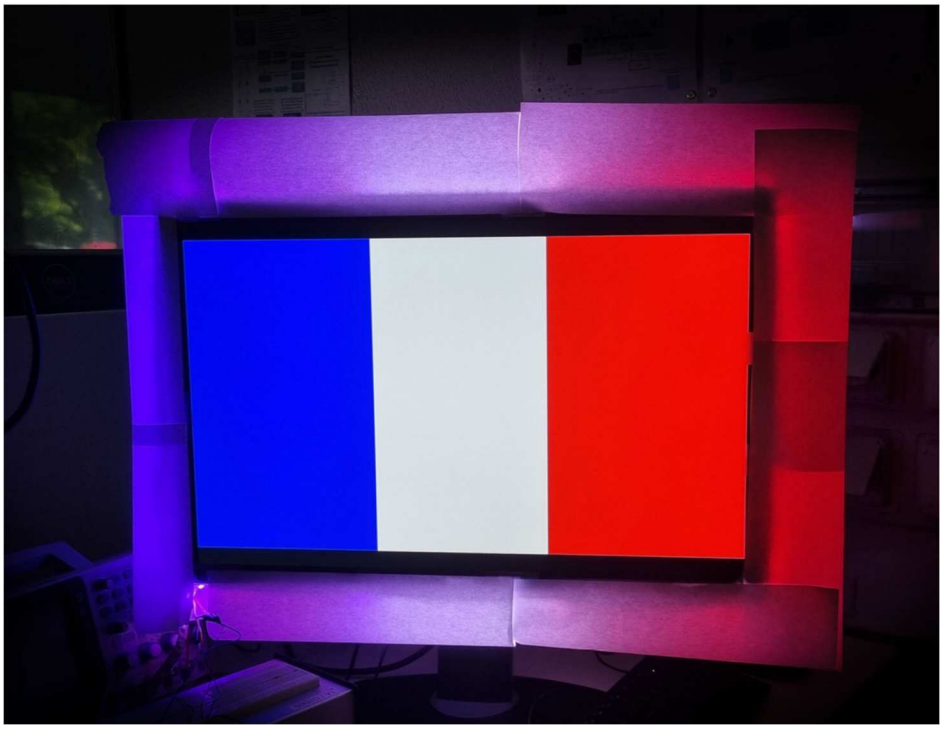

Figure 1: Notre Ambilight en salle de TP

Découpage fonctionnel

A. Cahier des charges

Le système Ambilight doit être adaptable à n'importe quel écran. [Réalisé]

Il doit pouvoir être utilisable par n'importe quel utilisateur de plus de 10 ans. Pour cela il doit être installable facilement sans connaissance électronique préalable. [Réalisé]

Il doit être branché à une prise secteur classique. [Réalisé]

La webcam doit être suffisamment discrète pour qu'elle ne gêne pas l'utilisateur. [Réalisé]

Le système doit assurer une immersion optimale : marche avec des images diversifiées, avec un temps de réponse le plus court possible. [Réalisé, optimal pour un des deux codes]

Améliorations possibles :
-adaptation colorimétrique de la caméra [Réalisé mais pas optimal]
-l'utilisateur peut régler la luminosité globale des LEDs grâce à un potentiomètre. [Non réalisé]

À priori, ce système est universel et adaptable à tout écran rectangulaire qu'il soit un ordinateur, une télévision ou même un téléphone portable. Donc il suffit à l'utilisateur de brancher le système à une prise secteur classique avec n'importe quelle Webcam à disposition pour qu'il puisse profiter d'une immersion totale. Et plus les couleurs sont vives et le temps de réponse est court c'est-à-dire la lumière suit exactement la dynamique des images affichées sur l'écran, plus l'expérience est immersive.

B. Schéma fonctionnel

La mise en œuvre du cahier des charges revient à suivre et appliquer le schéma suivant :

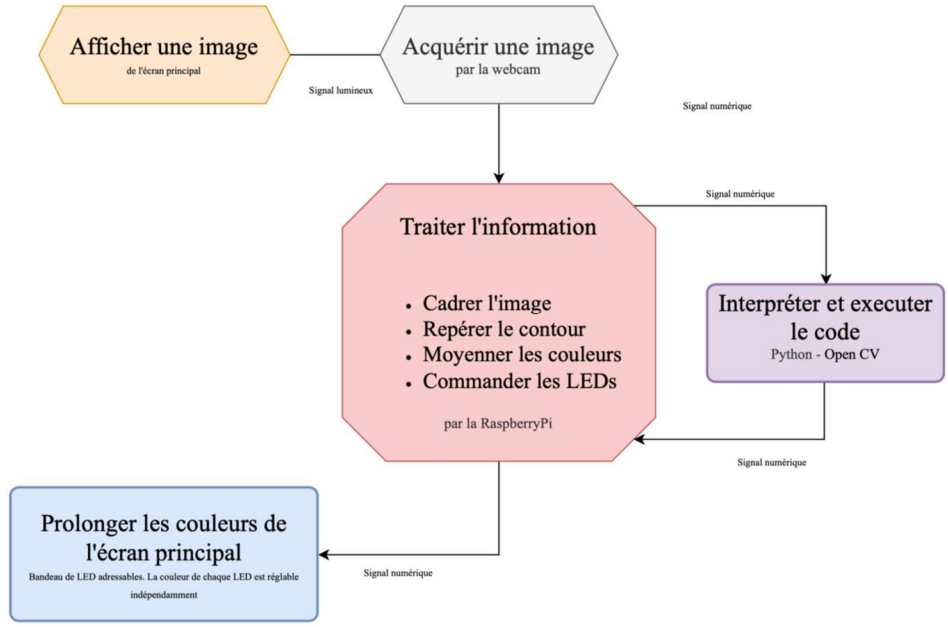

Figure 2: Schéma technique

Après avoir affiché l'image ou la vidéo sur l'écran, une caméra la capte et la transforme en matrice de pixel ; chaque pixel est représenté par trois couleurs : Rouge, Vert et Bleu. Cette matrice est ensuite traitée pour avoir un vecteur contenant les informations pour prolonger les couleurs de l'écran. Le traitement de l'image passe par une première étape de cadrage et repérage de l'écran. Puis le moyennage sur des petites zones du contour. Enfin ces données sont utilisées pour commander des LEDs RGB. Toute la partie programmation est réalisée par langage python avec la bibliothèque « OpenCV ».

Description technique par bloc

A. Schéma du montage

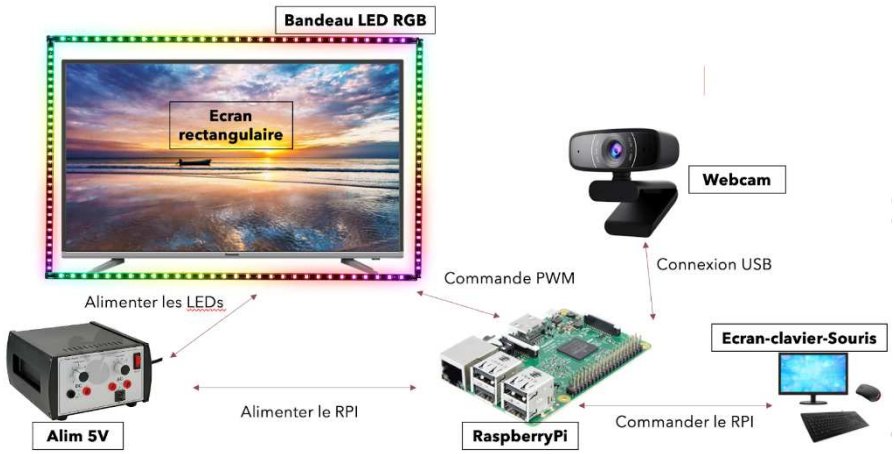

Figure 3: Composants utilisés et montage

L'image est affichée sur un écran rectangulaire, de taille quelconque. Une webcam, connectée en USB à la carte RaspberryPi acquiert des images de cet écran, et après des manipulations en Python, les LEDs sont commandées pour prolonger les couleurs de l'écran. L'interface utilisateur est un ensemble écran, clavier, souris branchés sur la RaspberryPi. Le tout est alimenté avec une alimentation 5V. Une puissance de 15W est nécessaire lorsque toutes les LEDs sont allumées en blanc.

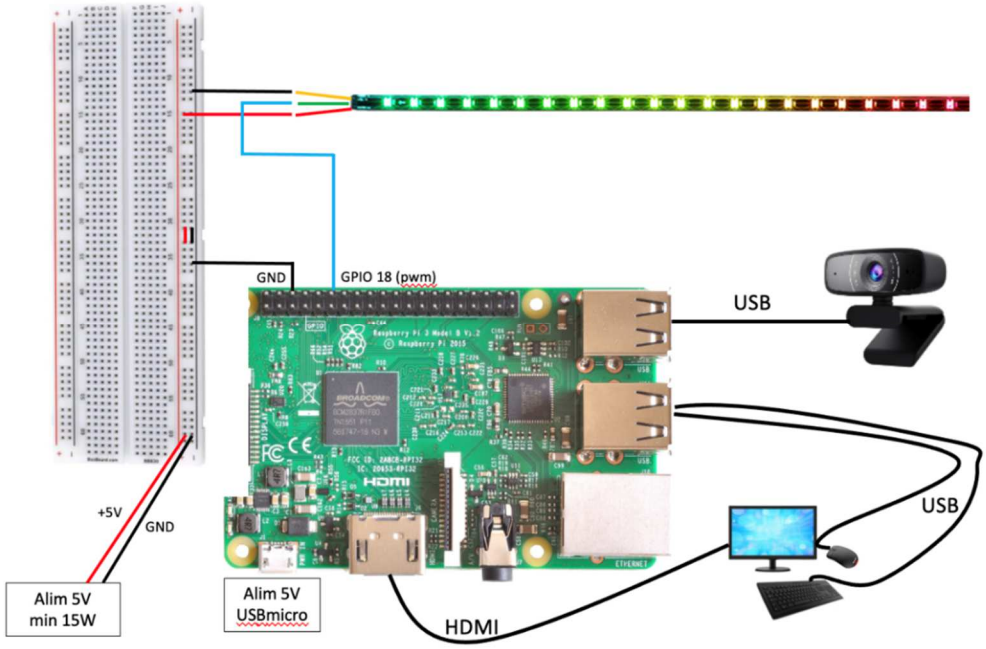

Figure 4: Schéma détaillé du montage

Nous allons maintenant détailler chaque fonction du système.

B. Acquérir une image

La première étape du système est l'acquisition de l'image affichée sur l'écran. Pour cela, on utilise une WebCam, ce qui est facile à utiliser. On commande cette caméra en utilisant la Raspberrypi. L'enchaînement normal était d'ouvrir la caméra prendre l'image, l'enregistrer la traiter et puis prendre une seconde image et ainsi de suite. Cependant, cette méthode est très lente vue le temps et le stockage nécessaire pour enregistrer la photo chaque fois. On se retrouve avec une vitesse de presque 0.5 image par seconde. La solution à ce problème est le « ***Multithreading*** ». Cette méthode est basée sur l'idée que et la caméra et le programme de traitement se sont exécutés en parallèle. La caméra capte donc une vidéo en continu et on extrait une image directement exploitable par OpenCV. On gagne beaucoup de temps avec la méthode du ***Multithreading*** et on obtient jusqu'à 30 images par seconde.

C. Cadrer l'écran

Le repérage et le cadrage de l'écran interviennent dans la phase d'initialisation du dispositif et servent à fournir de précieux renseignements pour la suite du code. En effet, l'image que fournit la caméra n'est pas parfaitement cadrée sur l'écran : Il y a toujours des bords et il peut également y avoir un angle d'inclinaison entre l'écran et la webcam.
L'ensemble des étapes décrites ci-dessous sont regroupées dans la fonction **initialisation** (*Annexe A*), lancée au début du programme principal.

La première étape consiste à repérer l'écran dans l'image renvoyée par la caméra. Pour cela, nous utilisons la fonction **reconnaissance_rect** qui permet de renvoyer l'image rectangulaire redimensionnée autour de l'écran crop (*en pointillés rouges sur la figure 5 ci-dessous*), la hauteur h et la largeur w du rectangle encadrant l'écran, une liste de points définissant le contour de l'écran contour_rect, ainsi que les coordonnées du point de départ de cette liste x0, y0, définie pour n'importe quelle configuration de l'écran comme le coin le plus haut du contour. Cette fonction utilise principalement les fonctions OpenCV **threshold**, **findcontours** et **boundingrect**.
Ensuite, à partir des coordonnées du contour de l'écran, on détermine l'angle d'inclinaison α de celui-ci par rapport à la caméra. Cela se fait grâce à la fonction **angle** (*Annexe A*) qui calcule la pente du côté le plus haut du rectangle en utilisant la fonction arc-tangente.

On incline ensuite l'image d'un angle -α, avec la fonction **subimage**, afin d'avoir l'écran droit sur une nouvelle image. On utilise à nouveau **reconnaissance_rect** sur cette image, afin de déterminer la hauteur h2 et la largeur w2 de l'écran lui-même.

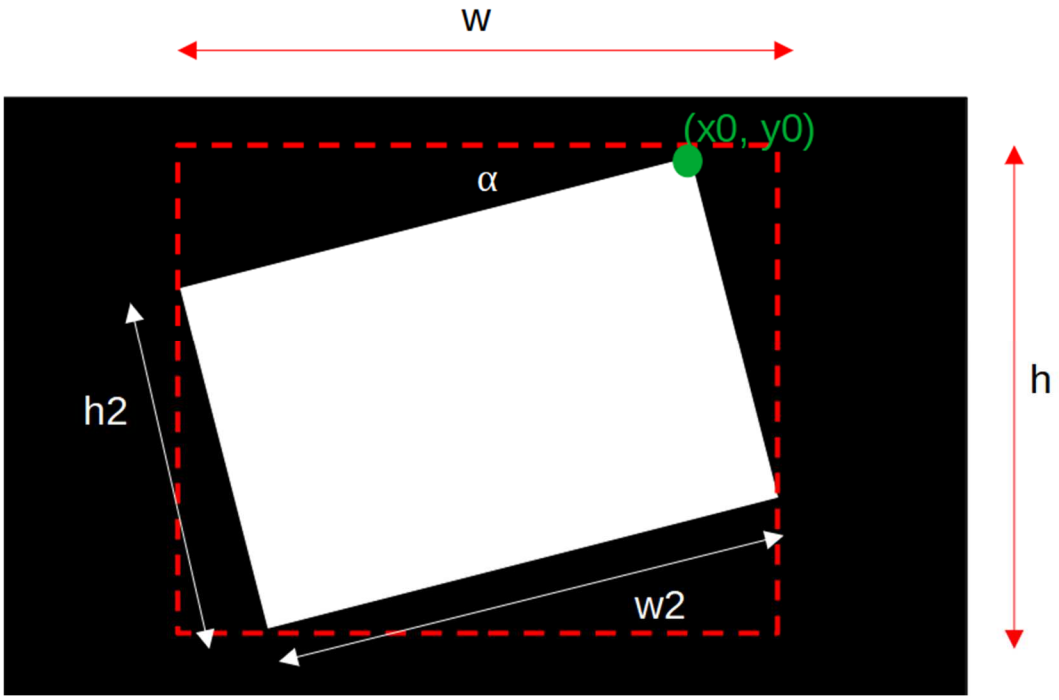

Figure 5: Etapes cadrage de l'écran

La fonction **initialisation** renvoie x0, y0, les coordonnées du point le plus haut de l'écran sur les images de la caméra, h2, w2, la hauteur et la largeur de l'écran, h et w les hauteurs et largeurs du plus petit rectangle encadrant l'écran et α, l'angle d'inclinaison de l'écran par rapport à la caméra.
Ces grandeurs ne sont pas modifiées lors de la phase d'utilisation du dispositif car la caméra reste fixe. Elles peuvent donc servir lors des étapes suivantes.

D. Moyenner les couleurs

Une fois que l'initialisation est faite, on connaît l'angle d'inclinaison de la caméra (α) et les dimensions du rectangle correspondant à l'écran (w2,h2). A partir de ces données, on souhaite dresser une liste de 110 tuples (x,y), un pour chaque LED, tout autour du contour de l'écran (*rectangle tireté orange figure 6*). C'est la fonction **bordure** qui réalise cette liste (*Annexe A*)

Une fois cette liste créée, elle est utilisée durant tout le code pour moyenner la couleur sur chacun des rectangles (*rectangles bleus clairs figure 4*) dont le coin supérieur gauche (*disques verts figure 6*) est un point de cette liste. On obtient alors une liste de 110 listes de 3 éléments [G,R,B]. C'est la fonction **moyennage** qui réalise ce calcul. (*Annexe A*)

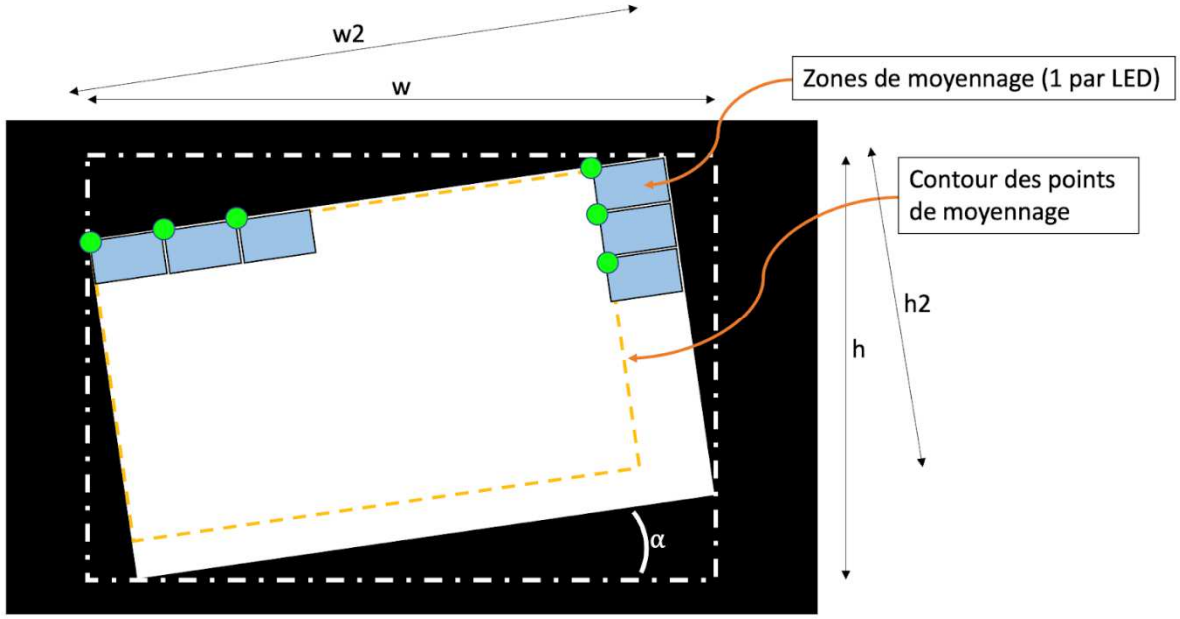

Figure 6: Etapes moyennage couleurs

Une fois la liste de couleur des 110 LEDs établie, on passe à la commande des LEDs.

E. Contrôler les LEDs

La liste déterminée par la fonction **moyennage** (*Annexe A*) permet alors de contrôler la couleur des LEDs. En effet nous avons pour ce projet utilisé un bandeau de 110 LEDs du type WS2801. Ces dernières sont très facilement manipulables avec une RaspberryPI grâce à la bibliothèque rpi_ws281x. On peut alors contrôler les couleurs de chaque LED indépendamment en lui associant une liste [G,R,B] où G,R,B sont respectivement les composantes vert, rouge et bleu de la couleur voulue (nombres entre 0 et 255). Cet ordre est essentiel car c'est celui défini par la bibliothèque utilisée. La fonction **changement_couleur** (*Annexe A*) permet alors de changer la couleur des LEDs à partir d'une liste de 100 listes de type [G,R,B]. On modifie ainsi les LEDs comme voulu en appliquant cette fonction à la liste trouvée par le moyennage.

Les LEDS sont branchées comme indiqué sur la *figure 4* : on branche les LEDs a une alimentation externe de 5V, on les relie à la RaspberryPI et on relie les masses des différents composants électroniques. On prend cette précaution afin de ne pas surchauffer la RaspberryPI: en effet si elle peut théoriquement alimenter des LEDs directement en 5V, nous avons ici utilisé un trop grand nombre de LEDs pour qu'elle puisse le supporter.

F. Réglages colorimétriques

Toutes les fonctions précédentes permettent déjà d'avoir un système fonctionnel et très rapide, comme décrit dans la partie Test et Validation. Cependant certaines couleurs n'apparaissent pas sur les LEDs (le marron ou le violet par exemple). Nous avons donc essayé d'apporter des améliorations colorimétriques.

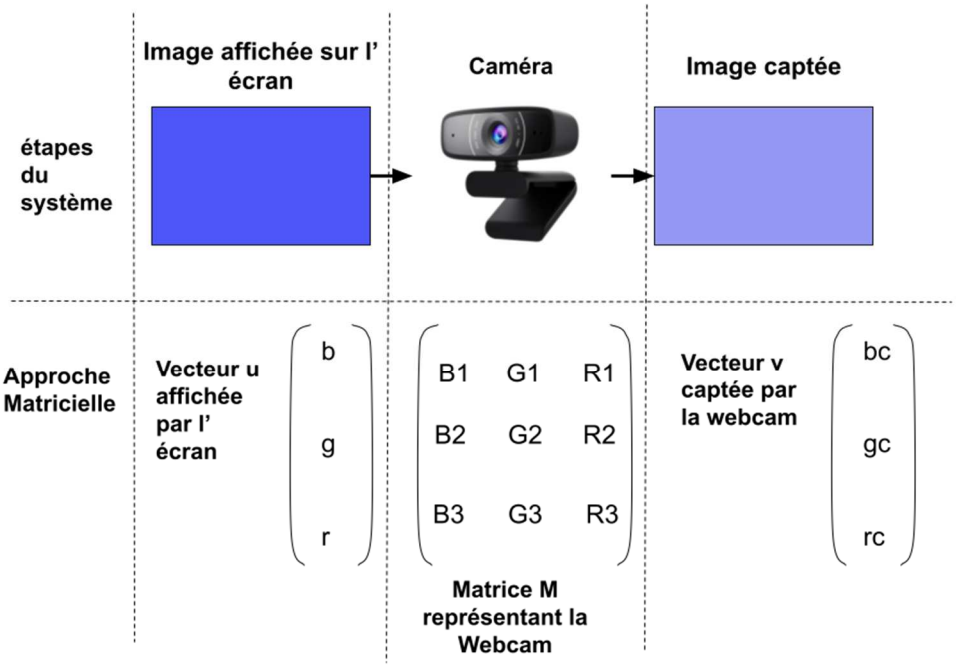

Figure 7: Visualisation matrice de colorimétrie

Pour ce faire, nous avons utilisé une approche matricielle, c'est-à-dire qu'on modélise, comme explicité sur la *figure 7* la webcam comme une matrice, et un pixel comme un vecteur colonne de type [B G R]. Notons dès à présent que l'ordre des composantes diffère de celui du contrôle des LEDs: c'est ici l'ordre donné par la webcam. Le pixel vu par la caméra correspond alors à l'opération : $v = M\,u$. Ainsi, dès que l'on connaît la matrice de passage M, on peut facilement remonter à la couleur originelle du pixel affichée grâce à la formule : $u = M^{-1}v$. Ainsi, si l'on détermine la matrice M, on pourra à partir de l'image captée obtenir les véritables couleurs que doivent prendre les LEDs.

On détermine M dans l'étape d'initialisation. Pour ce faire, on procède par colonne.
-On affiche sur l'écran un couleur "parfaite", par exemple un bleu (255,0,0)
-On obtient alors le vecteur $v_{couleur}$ lui correspondant : $v_{couleur}=(b_{couleur},v_{couleur},r_{couleur})$. Ce vecteur correspond à une colonne de la matrice (celle de la couleur utilisée sur l'écran). En effet, si on modélise le bleu parfait comme u=[255, 0,0], v correspond bien à la première colonne de la matrice (par produit matriciel). Et de même pour le vert

[0.255,0] et le rouge [0,0,255] correspondent ainsi respectivement à la deuxième et la troisième colonne de la matrice.
La matrice M mesurée correspond alors à : $M = \frac{1}{255}(\ v_{BLEU} \quad v_{VERT} \quad v_{Rouge})$. On divise ici par 255 pour normaliser notre matrice et la rendre utilisable pour n'importe quelle couleur.

Pour utiliser cette méthode dans le deuxième mode de fonctionnement (*Annexe B*), on ajoute, pendant l'initialisation, une étape de colorimétrie pour avoir la matrice : après avoir fait l'initialisation de la caméra avec un écran blanc, on indique à l'utilisateur qu'il doit passer à des images fournies : un écran vert puis rouge puis bleu. On va alors obtenir la matrice en faisant la moyenne sur plusieurs pixels.

A présent on va toujours utiliser cette matrice pour avoir les bonnes couleurs pour les LEDs. Nous avons décidé d'appliquer la matrice aux vecteurs captés dans la fonction **moyennage** (*Annexe B*) avant de l'ajouter à la liste des couleurs des LEDs.

Tests et validation

Nous avons pour ce projet réalisé deux modes de fonctionnement possibles : une version sans et une avec colorimétrie. Leur différence est donc la prise ou non en compte de l'étape Réglages colorimétriques mentionnée précédemment. Celle-ci va donner à chacun des deux codes des avantages différents. C'est pour cette raison que nous avons décidé de garder les deux. Cette partie a donc pour but de comparer ces deux codes.

Le premier code est la version de base du système. Cette version a un grand avantage : elle est très rapide. En effet les LEDs changent presque instantanément de couleur lors d'un changement d'image. Cependant comme mentionné plus tôt, il y a certains défauts colorimétriques : certaines couleurs sont mal retranscrites sur les LEDs (notamment violet, marron, jaune). Pendant l'utilisation, on remarque pourtant que lors d'un changement de couleur, les LEDs prennent souvent une couleur très proche de celle affichée, et c'est pendant la stabilisation et donc un certain temps d'exposition que celles-ci vont prendre une couleur différente. Par exemple, si l'on met une image rouge, les LEDs vont d'abord s'allumer en rouge vif comme l'image, puis avec le temps elles vont devenir plus claires et légèrement rosées. C'est pourquoi cette version est idéale pour des changements d'image, c'est-à-dire des vidéos (On aura seulement de légers retards).

L'avantage de la version colorimétrique est donc l'ajout de correction qui permettent d'avoir des LEDs qui s'affichent avec des couleurs fidèles à celles affichées sur l'écran. Cependant, l'approche utilisée est limitée et pose des problèmes. En effet, si notre code fonctionne dans le cas d'image unie (surtout pour les couleurs primaires), certaines erreurs apparaissent sur des images plus complexes, notamment au niveau des transitions entre les couleurs. On pourrait améliorer cet aspect en étudiant davantage les paramètres colorimétriques du système. En outre, cette version est plus lente que celle sans correction. Les LEDs peuvent ainsi mettre plusieurs secondes à changer de couleur. On pourrait, pour une future amélioration, réduire ce temps de réaction en modifiant le code : on utilise ici directement la matrice M pour chaque élément de la liste calculée par la fonction **moyennage** (*Annexe B*). Cela fait donc 110 calculs. A la place, on pourrait réaliser un unique calcul en créant une unique matrice, par laquelle on multiplierait la liste calculée. Cette version est donc à privilégier dans le cas d'image, pour avoir une meilleure immersion.

Travail de groupe

Ce projet, en plus de suivre une démarche d'ingénieur d'un point de vue technique, nous a aussi permis de travailler en groupe. Dans cette partie, nous allons détailler notre organisation, ainsi que ce que nous a apporté cette expérience de ce point de vue.

Planning

Nous avons d'abord dressé un planning prévisionnel. Par la suite, nous avons eu quelques imprévus, et quelques bonnes surprises, ce qui nous a mené à quelques changements. Voici donc le planning que nous avons effectivement suivi :

<table>
<tr><th>Séances</th><th colspan="2">Salma</th><th>Léa</th><th>Clara</th><th>Alexandre</th></tr>
<tr><td>1</td><td colspan="5">Séance d'introduction</td></tr>
<tr><td rowspan="2">2</td><td colspan="5">Réflexion générale
Choix des composants (bandeau LED adressable)
Cadrage du cahier des charges
Répartition des tâches
Début du codage</td></tr>
<tr><td colspan="3">Caméra
en mode image par image</td><td colspan="2">Cadrage image
via OpenCV</td></tr>
<tr><td>3</td><td colspan="3">Caméra
en mode image par image</td><td colspan="2">Cadrage image
via OpenCV</td></tr>
<tr><td>4</td><td colspan="5">Branchement des LEDs
Mise en commun du code
Débuggage</td></tr>
<tr><td>5</td><td colspan="5">Débuggage/ Moyennage des coleurs
Résolution partielle des problèmes
(effet miroir, coin de référence du rectangle, repères différents)</td></tr>
<tr><td>6</td><td colspan="5">Débuggage/ Moyennage des coleurs
Résolution totale des problèmes
(effet miroir, coin de référence du rectangle, repères différents)
Premier code totalement fonctionnel</td></tr>
<tr><td>7</td><td colspan="5">Optimisation code/accélération
Réglage colorimétrique</td></tr>
<tr><td>8</td><td colspan="5">Séance de présentation</td></tr>
</table>

Difficultés rencontrées

Pour résumer ce qui a déjà été présenté dans la partie II, nous pouvons dire que les difficultés que nous avons rencontrées sont surtout sur le plan technique. Les différences de repères dans les images suivant les bibliothèques, les problématiques de rapidité rencontrées ainsi que la qualité des câblages nous ont donné du fil à retordre, mais nous avons tout de même réussi à résoudre ces problèmes. En effet, les documentations d'OpenCV sont complètes et compréhensibles, et nous avons décidé de souder les câbles pour ne pas avoir de faux-contacts.

Analyse du travail d'équipe

Tous les membres du groupe ont pu apporter leurs compétences et les mettre à profit. Ce type de projet, essentiellement basé sur du code, se prête bien à une répartition du travail par fonction. Nous nous sommes réparti les fonctions à coder par binôme, et nous avons trouvé le mode de fonctionnement adéquat. Nous sommes contents de notre fonctionnement en tant que groupe et nous avons acquis des compétences non seulement techniques mais aussi organisationnelles et relationnelles dans un travail de groupe.

Annexe A : Code version rapide

```
import numpy as np
import cv2
import time
import matplotlib.pyplot as plt
from threading import Thread
import math as ma
from rpi_ws281x import *

#Définition des variables globales (utilisables dans les fonctions sans les mettre en
entrée)
global image
global deltaYmontee
global deltaXmontee
global deltaYdroite
global deltaXdroite

def cadrage(cap): #fonction qui affiche un retour video live jusqu'à l'appui de la
touche "q"
    if (cap.isOpened()== False):
      print("Error opening video  file")
    while(cap.isOpened()):
      ret, frame = cap.read()
      if ret == True:

        cv2.imshow('Frame', frame)
        if cv2.waitKey(25) & 0xFF == ord('q'):
    #Ferme la fenêtre quand on appuie sur q
          break
      else:
        break
    cv2.destroyAllWindows()

def initialisation(img):
    #executée une seule fois au début pour repérer l'écran dans l'image

    crop, contour_rect, w, h,x,y,x0,y0= reconnaissance_rect(img,'export1.jpg')
    alpha=angle(contour_rect,x,y,w,h)
    image_red=subimage(crop,(h/2,w/2),alpha,w,h)
    cv2.imwrite('export2.jpg',image_red)

crop2,contour_rect2,w2,h2,x2,y2,x02,y02=reconnaissance_rect(image_red,'finale.jpg
')
    return(x0,y0,h2,w2,alpha,w,h)
```

```
def reconnaissance_rect(crop, chemin_output):
    #reconnaît les rectangles dans une image et renvoie le contour du plus grand rectangle, ainsi que d'autres mesures

    img = crop
    img = cv2.medianBlur(img,5)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    _,thresh = cv2.threshold(gray,120,255,cv2.THRESH_BINARY)  #pixels<150 = noirs, pixels>150=255
    contours,hierarchy = cv2.findContours(thresh,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)

    aire_max=0
    indice_max=0

    a=len(contours) #contours contient les contours de tous les rectangles contenus dans l'image

    for i in range (a):  #on cherche celui avec l'aire la plus grande
        cnt=contours[i]
        x,y,w,h = cv2.boundingRect(cnt)
        if w*h>aire_max:
            aire_max=w*h
            indice_max=i

    cnt=contours[indice_max]
    (x0,y0)=(cnt[0][0][1],cnt[0][0][0])
    x,y,w,h = cv2.boundingRect(cnt) #coordonnées et dimensions
    crop=img[ y:y+h , x:x+w ]  #on recadre l'image
    cv2.imwrite(chemin_output,crop) #on exporte
    return crop, contours[indice_max], w, h,x,y,x0,y0

def angle(contour_rect,x,y,w,h):
    #calcule l'angle en degré de l'inclinaison du rectangle à partir de la liste de son contour
    #creation liste pour tracer le bord superieur du rectangle
    liste_xa=[]
    liste_ya=[]

    #on selectionne le côté supérieur du rectangle (critères de position à x% de la hauteur depuis le haut...pour que ca marche)
    for k in (contour_rect):
        if (k[0][0]>(x+w*0.2)) and (k[0][0]<(x+w*0.8)) and (k[0][1]>(y+h-h*0.25)) and (k[0][1]<y+h):
            liste_xa.append(k[0][0])
            liste_ya.append(k[0][1])

    #on calcule alpha en prenant le côté supérieur du rectangle et en calculant son inclinaison par trigo
    dx=liste_xa[-1]-liste_xa[0]
```

```
    dy=liste_ya[-1]-liste_ya[0]
    alpha=np.arctan(dy/dx)
    alpha_deg=alpha*180/np.pi

    return alpha_deg

def subimage(image, center, theta, width, height):
    #incline une image d'un angle alpha et recadre pour que cela ait la même
dimension

   shape = image.shape[:2]
   a=shape[0]
   b=shape[1]
   shape=(b,a)

   matrix = cv2.getRotationMatrix2D( center=center, angle=theta, scale=1 )
   image = cv2.warpAffine( src=image, M=matrix, dsize=shape )

   return image

def moyennage(deltaXm,deltaYm,deltaXd,deltaYd,h2,w2,alpha,listeBordure):
    #moyenne les couleurs des rectangls de coin supérieur gauche
    #appartenant à "listeBordure" et renvoie la "listeCouleur" correspondante
    #deltaXm (X ou Y pour l'axe, m ou d pour "monter", "droite")
    #le mot "monter" devrait finalement être remplacé par "descendre" (vestige d'un
ancien code...)
    listeCouleur=[]
    for k in listeBordure: #pour chaqun des 110 points de la bordure
        x0=k[0]
        y0=k[1]
        somme=[0,0,0]
        facteur=2 #prendre 1 pixel sur facteur (1px/2 fonctionne bien) pour accélerer le
moyennage

        for j in range (int(w2/(facteur*33))):#sur la largeur du rectangle de moyennage
            x0=x0+2*j*deltaXd  #on se décale à droite d'une incrémentation
            y0=y0+2*j*deltaYd
            for i in range (int(h2/(facteur*22))): #sur la hauteur du rectangle de
moyennage

somme[0]+=image[int(x0+facteur*i*deltaXm)][int(y0+facteur*deltaYm*i)][0] #on se
décale vers le bas et on ajoute la couleur à la somme

somme[1]+=image[int(x0+facteur*i*deltaXm)][int(y0+facteur*deltaYm*i)][1]

somme[2]+=image[int(x0+facteur*i*deltaXm)][int(y0+facteur*deltaYm*i)][2]
        Npx=(h2//(facteur*22))*(w2//(facteur*33))
        moyenne=[int(somme[0]/Npx) , int(somme[1]/Npx) , int(somme[2]/Npx)] #on
divise par le nombre de pixels pour faire la moyenne
        listeCouleur.append(moyenne) #on ajoute à la liste des couleurs finales
```

```
    return(listeCouleur)

def bordure(alpha,h2,w2,x,y,h,w):
    #renvoie une liste de 110 couples (x,y), coordonnées des points autour du
rectangle à partir desquels les moyennes seront calculées.
    listeContourFinale=[]

    if alpha>0:#alors le coin haut gauche est en x,y
        yBG=y-ma.sin(alpha)*h2
        xBG=x+ma.cos(alpha)*h2

    elif alpha<0:#alors le coin haut droit est en x,y
        yBG=y-w2*ma.cos(alpha) - h2*ma.sin(alpha)
        xBG=x+ma.cos(alpha)*h2-ma.sin(alpha)*w2

    #on appelle les variables globales
    global deltaYmontee
    global deltaXmontee
    global deltaYdroite
    global deltaXdroite

    #décalages qui servent à parcourir le rectangle de périphérie
    #(déplacement verticaux et horizontaux inclinés de l'angle alpha)
    deltaYmontee=ma.sin(alpha)
    deltaXmontee=-ma.cos(alpha)
    deltaYdroite=ma.cos(alpha)
    deltaXdroite=ma.sin(alpha)

    imageinit[int(xBG)][int(yBG)]=(0,255,0)

    #coté vertical gauche
    for z in range(21):
        xu=xBG+z*(h2/22)*deltaXmontee
        yu=yBG+z*(h2/22)*deltaYmontee
        xi=int(xu)
        yi=int(yu)
        listeContourFinale.append([xi,yi])
    xBG=xu
    yBG=yu
    listeContourFinale.append(listeContourFinale[-1]) #on duplique le pixel du coin

    #coté horizontal haut
    for z in range(32):
        xu=xBG+z*(w2/33)*deltaXdroite
        yu=yBG+z*(w2/33)*deltaYdroite
        xi=int(xu)
        yi=int(yu)
        listeContourFinale.append([xi,yi])
    xBG=xu
```

```
    yBG=yu
    listeContourFinale.append(listeContourFinale[-1])#on duplique le pixel du coin

    #coté vertical droit
    for z in range(21):
        xu=xBG-z*(h2/22)*deltaXmontee
        yu=yBG-z*(h2/22)*deltaYmontee
        xi=int(xu)
        yi=int(yu)
        listeContourFinale.append([xi,yi])
    xBG=xu
    yBG=yu
    listeContourFinale.append(listeContourFinale[-1])#on duplique le pixel du coin

    #coté horizontal bas
    for z in range(32):
        xu=xBG-z*(w2/33)*deltaXdroite
        yu=yBG-z*(w2/33)*deltaYdroite
        xi=int(xu)
        yi=int(yu)
        listeContourFinale.append([xi,yi])
    xBG=xu
    yBG=yu
    listeContourFinale.append(listeContourFinale[-1])#on duplique le pixel du coin

    return(listeContourFinale)

def changement_couleur(liste): #liste=liste de 110 listes de 3 éléments
correspondant aux couleurs v,r,b
    #change les 110 leds suivant le contenu de liste, et les affiche
    for i in range (110):
        strip.setPixelColor(i, Color(liste[i][1], liste[i][2], liste[i][0]))
    strip.show()

#THREAD
#cette classe est lancée en parallèle. Tout ce qu'il s'y passe n'influence pas les
calculs du code principal.
#cela permet notamment d'accélérer considérablement le temps de réponse du
système
class CalculCam:
    def __init__(self):
        self._running=True

    def terminate(self):
        self._running=False

    def run(self):
        if (cap.isOpened()== False):  # si il y a un probleme dans l'ouverture de la
camera
            print("Error opening video  file")
```

```
        while(cap.isOpened()): # tant que la camera est bien lue
            ret, frame = cap.read()  # on lit frame by frame

            if ret == True:
                global image #variable globale pour la réutiliser hors de la fonction
                image = frame.copy() # fonction pour copier l'image
                if cv2.waitKey(25) & 0xFF == ord('b'): #Ferme la fenêtre quand on appuie
sur q
                    break
            else:
                break

#Configuration du baldeau de LED
LED_COUNT      = 110     # Number of LED pixels.
LED_PIN        = 18      # GPIO pin connected to the pixels (must support PWM!).
LED_FREQ_HZ    = 800000  # LED signal frequency in hertz (usually 800khz)
LED_DMA        = 10      # DMA channel to use for generating signal (try 10)
LED_BRIGHTNESS = 255     # Set to 0 for darkest and 255 for brightest
LED_INVERT     = False   # True to invert the signal (when using NPN transistor level
shift)
LED_CHANNEL    = 0
LED_STRIP      = ws.SK6812_STRIP_RGBW

if __name__ == '__main__':
    # Create NeoPixel object with appropriate configuration.
    strip = Adafruit_NeoPixel(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA,
LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL, LED_STRIP)
    # Intialize the library (must be called once before other functions).
    strip.begin()


cap = cv2.VideoCapture(0) #damarrage de la caméra
cadrage(cap) #cadrage en retour video live
ret1, image = cap.read() #prise d'une image d'initialisation
imageinit=image.copy()
x0,y0,h2,w2,alpha,w,h=initialisation(image) #enfin, on initialise le repérage de l'écran
dans l'image

print("initialisation finie")

#création d'un thread avec la méthode run de la classe CalculCam
#qui réalise la prise d'image en continu (30fps)
Cam=CalculCam()
CamThread=Thread(target=Cam.run)
CamThread.start()

alpha=(ma.pi/180)*alpha #conversion en degrés pour la suite
```

```
time.sleep(1) #pour être sur que le thread est bien lancé, et que tout tourne
correctement

a=1
listeBordure=bordure(alpha,h2,w2,x0,y0,h,w)

while True:

moyenne=moyennage(deltaXmontee,deltaYmontee,deltaXdroite,deltaYdroite,h2,w2,
alpha,listeBordure)
    changement_couleur(moyenne)
```

Annexe B : Code avec améliorations colorimétriques

Attention : Cette annexe n'inclus pas l'ensemble du code mais uniquement les parties du code qui sont modifiées par rapport à la version précédente et les nouvelles fonctions.

```
def moyennage(deltaXm,deltaYm,deltaXd,deltaYd,h2,w2,alpha,listeBordure,invMat):
    listeCouleur=[]
    for k in listeBordure:
        x0=k[0]
        y0=k[1]
        somme=[0,0,0]
        for j in range (int(w2/(2*33))):
            x0=x0+2*j*deltaXd
            y0=y0+2*j*deltaYd
            for i in range (int(h2/(2*22))):
                    somme[0]+=image[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][0]
                    somme[1]+=image[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][1]
                    somme[2]+=image[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][2]
        Npx=(h2//44)*(w2//66)
        moyenne=np.array(( [(somme[0]/Npx)] , [(somme[1]/Npx)] , [(somme[2]/Npx)] ))
        #print("moyenne")
        #print(moyenne)
        L=np.dot(invMat,moyenne)
        #print("L")
        #print(L)
        L2=[int(L[0][0]),int(L[1][0]),int(L[2][0])]
        if L2[0]<0:
            L2[0]=0
        if L2[1]<0:
            L2[1]=0
        if L2[2]<0:
            L2[2]=0
        #print("L2")
        #print(L2)
        listeCouleur.append(L2)

    return(listeCouleur)

def
moyennageColo(deltaXm,deltaYm,deltaXd,deltaYd,h2,w2,alpha,listeBordure,img):
    listeCouleur=[]
    for k in listeBordure:
        x0=k[0]
        y0=k[1]
        somme=[0,0,0]
        for j in range (int(w2/(2*33))):
```

```
            x0=x0+2*j*deltaXd
            y0=y0+2*j*deltaYd
            for i in range (int(h2/(2*22))):
                somme[0]+=img[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][0]
                somme[1]+=img[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][1]
                somme[2]+=img[int(x0+2*i*deltaXm)][int(y0+2*deltaYm*i)][2]
        Npx=(h2//44)*(w2//66)
        moyenne=[int(somme[0]/Npx) , int(somme[1]/Npx) , int(somme[2]/Npx)]
        listeCouleur.append(moyenne)
    return(listeCouleur)



# LED strip configuration:
LED_COUNT      = 110     # Number of LED pixels.
LED_PIN        = 18      # GPIO pin connected to the pixels (must support PWM!).
LED_FREQ_HZ    = 800000  # LED signal frequency in hertz (usually 800khz)
LED_DMA        = 10      # DMA channel to use for generating signal (try 10)
LED_BRIGHTNESS = 255     # Set to 0 for darkest and 255 for brightest
LED_INVERT     = False   # True to invert the signal (when using NPN transistor level
shift)
LED_CHANNEL    = 0
LED_STRIP      = ws.SK6812_STRIP_RGBW

if __name__ == '__main__':
    # Create NeoPixel object with appropriate configuration.
    strip = Adafruit_NeoPixel(LED_COUNT, LED_PIN, LED_FREQ_HZ, LED_DMA,
LED_INVERT, LED_BRIGHTNESS, LED_CHANNEL, LED_STRIP)
    # Intialize the library (must be called once before other functions).
    strip.begin()
    #strip.setPixelColor(3, Color(0, 255, 255))
    #strip.show()




cap = cv2.VideoCapture(0)
cadrage(cap)
ret1, image = cap.read() #Initialiser l'image
imageinit=image.copy()
x0,y0,h2,w2,alpha,w,h=initialisation(imageinit) #initialise le repérage de l'écran dans
l'image
cap.release()

alpha=(ma.pi/180)*alpha

listeBordure=bordure(alpha,h2,w2,x0,y0,h,w)

cap = cv2.VideoCapture(0)
```

```
print("Mettre l'image de calibration G")
for i in range (110):
    strip.setPixelColor(i, Color(100,0,0))
strip.show()
time.sleep(1)
for i in range (110):
    strip.setPixelColor(i, Color(0,0,0))
strip.show()
time.sleep(5)
ret1, imageG = cap.read() #Initialiser l'image
time.sleep(5)
#cv2.imshow("G",imageG)
cap.release()

moyenne=moyennageColo(deltaXmontee,deltaYmontee,deltaXdroite,deltaYdroite,h2
,w2,alpha,listeBordure,imageG)
#print(moyenne)
#print(len(moyenne))
G1,G2,G3=0,0,0
for k in range (110):
    G1+=moyenne[k][0]
    G2+=moyenne[k][1]
    G3+=moyenne[k][2]
G1=G1/110
G2=G2/110
G3=G3/110

cap = cv2.VideoCapture(0)
print("Mettre l'image de calibration R")
for i in range (110):
    strip.setPixelColor(i, Color(0,100,0))
strip.show()
time.sleep(1)
for i in range (110):
    strip.setPixelColor(i, Color(0,0,0))
strip.show()
time.sleep(5)
ret1, imageR = cap.read() #Initialiser l'image
time.sleep(5)
#cv2.imshow("R",imageR)
cap.release()

moyenne=moyennageColo(deltaXmontee,deltaYmontee,deltaXdroite,deltaYdroite,h2
,w2,alpha,listeBordure,imageR)
R1,R2,R3=0,0,0
for k in range (110):
    R1+=moyenne[k][0]
    R2+=moyenne[k][1]
    R3+=moyenne[k][2]
R1=R1/110
```

```
R2=R2/110
R3=R3/110

cap = cv2.VideoCapture(0)
print("Mettre l'image de calibration B")
for i in range (110):
    strip.setPixelColor(i, Color(0,0,100))
strip.show()
time.sleep(1)
for i in range (110):
    strip.setPixelColor(i, Color(0,0,0))
strip.show()
time.sleep(5)
ret1, imageB = cap.read() #Initialiser l'image
time.sleep(5)
#cv2.imshow("B",imageB)
cap.release()

moyenne=moyennageColo(deltaXmontee,deltaYmontee,deltaXdroite,deltaYdroite,h2
,w2,alpha,listeBordure,imageB)
B1,B2,B3=0,0,0
for k in range (110):
    B1+=moyenne[k][0]
    B2+=moyenne[k][1]
    B3+=moyenne[k][2]
B1=B1/110
B2=B2/110
B3=B3/110

matrice=np.array(([B1,G1,R1],[B2,G2,R2],[B3,G3,R3]))
#rint(matrice)
invMat=np.linalg.inv(matrice/255)
#print("InvMat")
#print(invMat)

#print(B1,B2,B3)
#print(np.dot(invMat,np.array(([G1],[G2],[G3]))))


#print(np.dot(invMat,np.array(([R1],[R2],[R3]))))


#print(np.dot(invMat,np.array(([B1],[B2],[B3]))))

print("initialisation finie")

#création d'un thread avec la méthode run de la classe CalculCam
#qui réalise les calculs assez lourds de traitement d'image.
cap = cv2.VideoCapture(0)
Cam=CalculCam()
```

```
CamThread=Thread(target=Cam.run)
CamThread.start()



alpha=(ma.pi/180)*alpha

'''
largeur2,hauteur2=h2,w2
xNous,yNous=x,y
largeur,hauteur=w,h

print("largeur 2 =",largeur2)
print("hauteur2 =",hauteur2)
print("xNous,yNous =",xNous,yNous)
print("largeur=",largeur)
print("hauteur=",hauteur)
'''


#print(image.shape)
time.sleep(1)
#print(alpha)
#print(len(moyenne))
#print(moyenne)
a=1

while True:
    #start=time.time()

moyenne=moyennage(deltaXmontee,deltaYmontee,deltaXdroite,deltaYdroite,h2,w2,
alpha,listeBordure,invMat)
    #print(time.time()-start)
    changement_couleur(moyenne)


cap.release()
```

Annexe C : Notice d'utilisation

- Préparer un diaporama à projeter sur l'écran d'ordinateur avec les diapositives suivantes
  - Blanc (255,255,255)
  - Vert (0,255,0)
  - Rouge (255,0,0)
  - Bleu (0,0,255)
  - Attention: l'ordre des couleurs est très important, notamment pour la colorimétrie

- Brancher le montage
  - Le câble rouge du bandeau LEDs (soudé à un câble rouge) doit être relié à une alimentation 5V continue extérieure.
  - Le câble jaune du bandeau LEDs (soudé à un câble noir) doit être relié à la masse de l'alimentation 5V et l'ensemble doit également être relié à la masse du RaspberryPi3(pin #6)
  - Le câble vert/bleu du bandeau LEDs (soudé à un câble bleu) est relié au RaspberryPi3 à la borne GPIO18 (pin #12)
  - Attention: le code est prévu pour que les LEDs soient reliées à cette borne et uniquement cette borne!
  - Le RaspberryPi3 est relié à son alimentation
  - La caméra doit être branchée sur l'un des ports USB du RaspberryPi

- Allumer le Raspberry et l'alimentation des LEDs
  - Dans la fenêtre de commande du Raspberry, entrer la commande *sudo Thonny* . Dans la suite, l'ensemble des codes seront ouvert sous Thonny

- Choisir le code utilisé
  - Le code *VF_rapide* correspond à la version la plus rapide du code, sans les améliorations colorimétriques.
  - Le code *VF_colo* correspond à la version plus lente avec les améliorations colorimétriques.

- Lancer le code en appuyant sur Run
  - La caméra est lancée et un retour vidéo apparaît sur l'écran du Raspberry. Positionner la caméra afin de voir l'ensemble de l'écran. Il ne faut pas être trop près de l'écran ni trop loin. Il peut y avoir un angle entre l'écran et la caméra.
  - Appuyer sur la touche *q* du clavier afin de lancer la suite du programme et de fermer le retour vidéo.
  - Attention: Ne plus toucher à la position de la caméra!

- Si le code **rapide** est utilisé
  - La phase d'initialisation est presque instantanée, attendre quand même de voir s'afficher à l'écran "Initialisation terminée" et de voir les LEDs s'allumer en blanc
  - Vous pouvez changer d'image ou lancer une vidéo

- Si le code **colorimétrique** est utilisé
  - La phase d'initialisation continue
  - Lorsque les LEDs deviennent vertes, mettez un fond vert à l'écran
  - Lorsque les LEDs deviennent rouges, mettez un fond rouge à l'écran
  - Lorsque les LEDs deviennent bleues, mettez un fond bleu à l'écran
  - Attention: les LEDs ne restent allumées qu'une seconde, il reste ensuite 5 secondes pour changer la couleur de l'écran
  - Attention: Les couleurs des écrans doivent être pures ex: (255,0,0)
  - Lorsque les LEDs s'allument en bleu, vous pouvez changer d'écran, le temps de réaction est plus long (plusieurs secondes)

- Arrêter le système : appuyer sur Stop sur l'écran du Raspberry

- Eteindre et rallumer l'alimentation des LEDs avant de faire fonctionner à nouveau le code

Annexe D : Sources

lense.institutoptique.fr

L’ensemble des images utilisées sont issues de sources les ayant classées comme libre de droit d’usage. Certaines figures ont également été réalisées par nos soins.