

Rapport Projet 903

Traitement et Analyse d'images

Julien Ducrey
M2 CMI MATH

Année Universitaire: 2023-2024

Table des matières

1	Introduction	2
2	Partie I : Set-up initial avec mini-application de capture vidéo	2
3	Partie II : Distribution ou Histogramme de couleurs	3
3.1	Représentation d'un histogramme de couleurs	3
3.2	Distance entre deux histogrammes de couleurs	4
3.3	Vérification	5
4	Partie III : Comparaison Fond et Objet	6
4.1	Distribution de couleurs du fond	6
4.2	Distribution de couleurs d'un objet	7
4.3	Vérification	7
5	Partie IV : Mode Reconnaissance	8
5.1	Vérification	9
6	Partie V : Autres développements possibles	11
6.1	Ajouter d'autres objets	11
6.2	Vérification	12
6.3	Optimiser la vitesse d'exécution	14
6.4	Suppression du dernier objet mémorisé	15
7	Conclusion	17



1 Introduction

Ce mini-projet a pour but de construire progressivement une application qui segmente un flux vidéo en temps-réel et reconnaît des "objets", que vous lui avez montré.

- I) Le programme "apprend" les distributions de couleur du fond.
- II) Le programme "apprend" 2 distributions de couleurs, une pour le fond, une pour le visage ou un objet.
- III) Le programme "apprend" plusieurs distributions de couleurs sur des objets différents.

2 Partie I : Set-up initial avec mini-application de capture vidéo

On nous donne le code en C++, utilisant la bibliothèque OpenCV ci-dessous. Ce code ouvre votre caméra et récupère une image, toutes les 50 ms.

```
#include <iostream>
#include <algorithm>
#include <opencv2/core/utility.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>

using namespace cv;
using namespace std;

int main( int argc, char** argv )
{
    Mat img_input, img_seg, img_d_bgr, img_d_hsv, img_d_lab;
    VideoCapture* pCap = nullptr;
    const int width = 640;
    const int height= 480;
    const int size = 50;
    // Ouvre la camera
    pCap = new VideoCapture( 0 );
    if( ! pCap->isOpened() ) {
        cout << "Couldn't open image / camera ";
        return 1;
    }
    // Force une camera 640x480 (pas trop grande).
    pCap->set( CAP_PROP_FRAME_WIDTH, 640 );
    pCap->set( CAP_PROP_FRAME_HEIGHT, 480 );
    (*pCap) >> img_input;
    if( img_input.empty() ) return 1; // probleme avec la camera
    Point pt1( width/2-size/2, height/2-size/2 );
    Point pt2( width/2+size/2, height/2+size/2 );
    namedWindow( "input", 1 );
    imshow( "input", img_input );
    bool freeze = false;
    while ( true )
    {
        char c = (char)waitKey(50); // attend 50ms -> 20 images/s
        if ( pCap != nullptr && ! freeze )
            (*pCap) >> img_input;      // récupère l'image de la caméra
        if ( c == 27 || c == 'q' ) // permet de quitter l'application
            break;
        if ( c == 'f' ) // permet de geler l'image
```

```

        freeze = ! freeze;
        cv::rectangle( img_input, pt1, pt2, Scalar( { 255.0, 255.0, 255.0 } ), 1 );
        imshow( "input", img_input ); // affiche le flux video
    }
    return 0;
}

```

Notez où on récupère l'image courante (ligne 39) et comment on récupère facilement sur quelle touche on presse (ligne 37), par exemple si on presse 'f', on peut geler la caméra.

3 Partie II : Distribution ou Histogramme de couleurs

De la même façon que l'on peut définir la distribution des niveaux de gris d'une zone dans une image N & B (c'est l'histogramme), on peut définir une distribution des couleurs dans une zone d'une image couleur. On peut aussi parler d'histogramme couleur.

3.1 Représentation d'un histogramme de couleurs

Le problème est que a priori notre histogramme couleur aurait une taille 256x256x256 pour l'espace RGB usuel. C'est beaucoup trop coûteux à stocker, mais surtout énormément de ces cases seraient vides.

Du coup, on va résumer l'histogramme dans beaucoup moins de cases, typiquement 8x8x8, donc 512 cases (on peut faire plus genre 16x16x16 mais votre ordinateur va ramer).

La première étape du développement de notre application, a été la gestion de l'objet *ColorDistribution*, représentant les histogrammes des distributions de couleurs. Au travers d'une structure, il a fallu écrire les méthodes membres, permettant de gérer ces objets.

Le code suivant présente l'initialisation, l'actualisation, la suppression et la normalisation des histogrammes de couleurs, ainsi que le calcul de la distance entre deux histogrammes :

```

struct ColorDistribution {
    float data[8][8][8]; // l'histogramme
    int nb = 0;           // le nombre d'échantillons

    ColorDistribution() { reset(); }

    ColorDistribution& operator=(const ColorDistribution& other) = default;
    // Met à zéro l'histogramme
    void reset() {
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                for (int k = 0; k < 8; k++) {
                    data[i][j][k] = 0;
                }
            }
        }
    }

    // Ajoute l'échantillon color à l'histogramme:
    // met +1 dans la bonne case de l'histogramme et augmente le nb d'échantillons
    void add(Vec3b color) {
        // Vec3b contient 3 entiers entre 0 et 255 la couleur du pixel courant, dans le carré blanc de
        // On souhaite convertir ces valeurs entre 0 et 8, pour incrémenter de 1 la case correspondante
        data[int(floor(color[0] / 32))][int(floor(color[1] / 32))][int(floor(color[2] / 32))]++;
        nb++;
    }

    // Indique qu'on a fini de mettre les échantillons:
    // divise chaque valeur du tableau par le nombre d'échantillons
    // pour que case représente la proportion des pixels qui ont cette couleur.
}

```

```

void finished() {
    //normalisation des valeurs des histogrammes, en divisant par le nombre d'échantillons:
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            for (int k = 0; k < 8; k++) {
                data[i][j][k] = data[i][j][k] / nb;
            }
        }
    }
};

// Retourne la distance entre cet histogramme et l'histogramme other
float distance(const ColorDistribution& other) const {
    float resultat = 0;
    // Calcul de la distance du CHI-2 entre les histogrammes:
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            for (int k = 0; k < 8; k++) {
                if ((data[i][j][k] + other.data[i][j][k]) > 0) {
                    resultat = resultat + ((data[i][j][k] - other.data[i][j][k]) * (data[i][j][k] - other.data[i][j][k]));
                }
            }
        }
    }
    return resultat;
};

```

On nous a également fournit le code suivant, utilisant les méthodes membres de la structure *ColorDistribution*, afin de nous aider à les écrire, ainsi que de voir comment les utiliser, pour récupérer les distributions de couleurs, des images capturées par la caméra :

```

ColorDistribution getColorDistribution( Mat input, Point pt1, Point pt2 ) {
    ColorDistribution cd;
    for ( int y = pt1.y; y < pt2.y; y++ )
        for ( int x = pt1.x; x < pt2.x; x++ )
            cd.add( input.at<Vec3b>( y, x ) );
    cd.finished();
    return cd;
}

```

3.2 Distance entre deux histogrammes de couleurs

On se rappelle que l'on peut voir un histogramme comme une densité de probabilité et qu'on peut appliquer toute l'artillerie usuelle des statistiques.

La distance du chi2 a été choisie, pour mesurer l'écart entre deux histogrammes de couleurs. Il a été demandé d'écrire une méthode membre pour la structure *ColorDistribution*, permettant de calculer cette distance.

Le code suivant présente cette méthode, les valeurs de l'histogramme ont bien été normalisées, en divisant par le nombre d'échantillons :

```

// Retourne la distance entre cet histogramme et l'histogramme other
float distance(const ColorDistribution& other) const {
    float resultat = 0;
    // Calcul de la distance du CHI-2 entre les histogrammes:
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            for (int k = 0; k < 8; k++) {

```

```

        if ((data[i][j][k] + other.data[i][j][k]) > 0) {
            resultat = resultat + ((data[i][j][k] - other.data[i][j][k]) * (data[i][j][k] -
        }
    }
}
return resultat;
};

```

3.3 Vérification

Afin de vérifier que le calcul de la distance entre deux histogrammes de couleurs se fait correctement, on ajoute une touche "v", permettant de réaliser un petit test.

Ce test consiste à enregistrer les 2 histogrammes des distributions de couleurs, issues des moitiés gauche et droite de l'image capturée par la caméra, à calculer ensuite la distance entre ces deux histogrammes et à l'afficher.

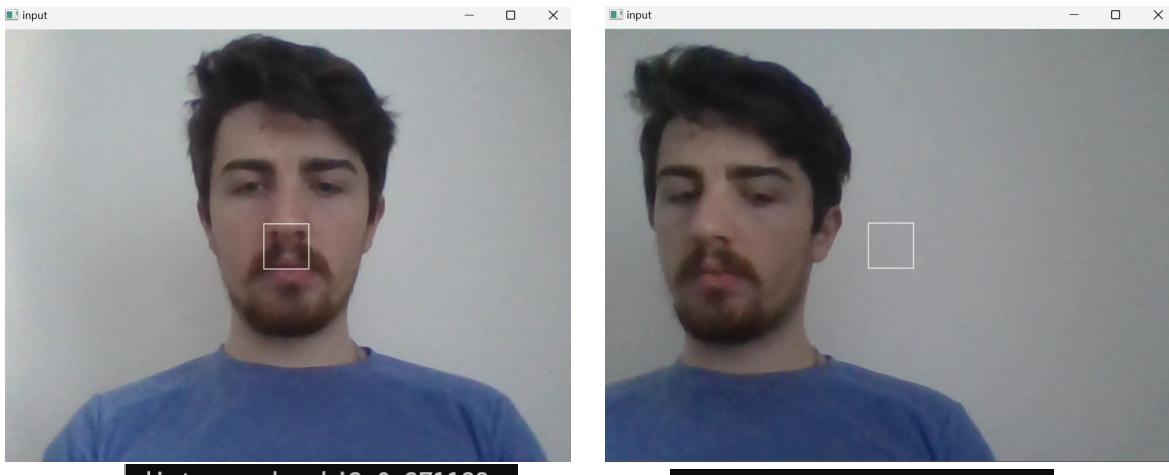
Voici le code de la touche, permettant de réaliser ce petit test :

```

if (c == 'v' && !freeze) { // test le calcule de la distance chi2
    Point p1(0, 0);
    Point p2(width / 2, height - 1);
    Point p3(width / 2, 0);
    Point p4(width - 1, height - 1);
    ColorDistribution cd1 = getColorDistribution(img_input, p1, p2);
    ColorDistribution cd2 = getColorDistribution(img_input, p3, p4);
    //ICI ajouter la suite pour calculer la distance entre les histogrammes.
    float res = cd1.distance(cd2);
    cout << "distance du chi2:" << res << endl;
}

```

À l'utilisation, on observe que le code fonctionne correctement :



Le code semble bien fonctionné, car la première image obtient effectivement une distance plus petite, que la seconde. Étant placé au milieu de la première image, je la rends plus symétrique. Ainsi les deux histogrammes des distributions de couleurs, des parties gauche et droite de cette image, sont donc plus proches.

Pour la seconde image, où je suis placé plus sur la partie gauche, la distance entre les deux histogrammes de distributions de couleurs, est comme attendu plus grande.

4 Partie III : Comparaison Fond et Objet

Afin de faire de la reconnaissance d'objet, on a besoin de distinguer le fond de l'objet à reconnaître. Pour cela, on va voir dans un premier temps, une manière de mémoriser le fond, puis dans un second temps, une manière de mémoriser l'objet, que l'on souhaite plus tard pouvoir reconnaître et distinguer du fond.

4.1 Distribution de couleurs du fond

Comme le fond peut être assez varié (suivant votre décor), qu'éventuellement vous voudriez faire partie du fond, on ne va pas mémoriser une seule distribution pour le fond mais plusieurs. On va découper le fond en blocs 128x128 (sauf aux bords), du coup ça fera environ 20 distributions de couleurs qui décrivent le fond.

Les structures de données utilisées, pour représenter le fond et l'objet, sont les suivantes :

```
std::vector<ColorDistribution> col_hists; // histogrammes du fond  
std::vector<ColorDistribution> col_hists_object; // histogrammes de l'objet
```

Ce sont des vecteurs, dans lesquels on peut enregistrer plusieurs distributions de couleurs, qui serviront à reconnaître la même entité, que ce soit le fond ou un objet à reconnaître.

À cette étape du développement, il a été demander de rajouter une touche 'b', qui calcule ces histogrammes de couleurs du fond, sur les différentes parties de l'image, et qui les mémorise dans le tableau *col_hists*.

Pour cela, on nous a fournit le pseudo code suivant :

```
const int bbloc = 128;  
for (y=0; y <= height-bbloc; y += bbloc )  
    for (x=0; x <= width-bbloc; x += bbloc )  
    {  
        calcule ColorDistribution sur le bloc (x,y) -> (x+bbloc, y+bbloc)  
        le mémorise dans col_hists  
    }  
int nb_hists_background = col_hists.size();
```

Sur la base de ce pseudo code, j'ai écrit le code suivant :

```
if (c == 'b' && !freeze) { //Calcule des histogrammes de couleurs, sur les différentes parties de l'image  
    const int bbloc = 128;  
    for (int y = 0; y <= height - bbloc; y += bbloc) {  
        for (int x = 0; x <= width - bbloc; x += bbloc) {  
            //calcule ColorDistribution sur le bloc(x, y) -> (x + bbloc, y + bbloc):  
            Point ptHautGauche(x, y);  
            Point ptBasDroite(x + bbloc, y + bbloc);  
            ColorDistribution cdBloc = getColorDistribution(img_input, ptHautGauche, ptBasDroite);  
            //le mémorise dans col_hists:  
            col_hists.push_back(cdBloc);  
        }  
    }  
    cout << "Histogrammes de fond, taille: " << col_hists.size() << endl;  
    int nb_hists_background = col_hists.size();  
}
```

En le testant, on observe qu'il fonctionne correctement :

```
Histogrammes de fond, taille: 15
```

On peut également ajouter une nouvelle série d'histogrammes de couleurs, représentant le fond, en appuyant à nouveau sur la touche "b" :

```
Histogrammes de fond, taille: 30
```

4.2 Distribution de couleurs d'un objet

Maintenant que l'on dispose des histogrammes des distributions de couleurs du fond, on souhaite également obtenir ceux de l'objet à reconnaître.

On nous demande donc de rajouter une touche "a", qui calcule l'histogramme de couleur de la partie, qui est matérialisé par le rectangle blanc, et qui le rajoute, à l'autre tableau d'histogrammes de couleurs *col_hists_object*.

Ainsi, l'utilisateur pourra présenter plusieurs parties du même objet. Cela facilitera sa reconnaissance après.

Voici une première version du code produit, pour la gestion de la touche "a", ce code évoluera par la suite, pour prendre en compte les besoins des nouvelles fonctionnalités de l'application :

```
if (c == 'a' && !freeze) {  
    //calcule ColorDistribution sur le rectangle blanc:  
    ColorDistribution cdRectangleBlanc = getColorDistribution(img_input, pt1, pt2);  
    //le mémorise dans col_hists:  
    col_hists_object.push_back(cdRectangleBlanc);  
    cout << "Histogramme de l'objet, taille: " << col_hists_object.size()<< endl;  
}
```

4.3 Vérification

On effectue une rapide vérification, du bon fonctionnement de la touche "a" :

Histogramme de l'objet, taille: 1

Si on appuie de nouveau, sur la touche "a" :

Histogramme de l'objet, taille: 2

On peut alors enregistrer plusieurs histogrammes des distributions de couleurs, de l'objet que l'on souhaite reconnaître.

5 Partie IV : Mode Reconnaissance

Dès lors que les deux tableaux *col_hists* et *col_hists_objects* ne sont pas vides, l'utilisateur peut demander à basculer en mode "reconnaissance", en appuyant sur la touche "r".

Dans ce mode, on découpe l'image en blocs 8x8 ou 16x16 (comme vous voulez). Par bloc, on calcule son histogramme de couleurs *h* et on cherche l'histogramme dans *col_hists* et dans *col_hists_objects* qui a le plus petite distance avec *h*.

Si le plus proche est dans *col_hists*, alors le bloc sera étiqueté "fond", sinon il sera donc étiqueté "objet".

Afin de mettre en place ce mode reconnaissance, on a besoin d'écrire une fonction qui retourne la plus petite distance, entre *h* et les histogrammes de couleurs de *hists*.

Voici le code développé, de cette fonction :

```
float minDistance(const ColorDistribution& h, const std::vector< ColorDistribution >& hists) {
    float dMin = h.distance(hists[0]);
    for (int i=1;i<hists.size();i++) {
        float newD=h.distance(hists[i]);
        if (newD<dMin) {
            dMin = newD;
        }
    }
    return dMin;
};
```

Ensute, on a besoin d'écrire une autre fonction, qui fabrique une nouvelle image, où chaque bloc est coloré selon qu'il est "fond" ou "objet".

Voici le code développé de cette seconde fonction :

```
Mat recoObject(Mat input,
    const std::vector< ColorDistribution >& col_hists, /*< les distributions de couleurs du fond */
    const std::vector< ColorDistribution >& col_hists_object, /*< les distributions de couleurs de l'obj */
    const std::vector< Vec3b >& colors, /*< les couleurs pour fond/objet */
    const int bloc_taille /*< taille de chaque bloc, 16 si 16x16 */) {
    Mat img_output= Mat::zeros(input.size(), input.type());
    for (int y = 0; y <= input.rows-bloc_taille; y += bloc_taille) { // On met -bloc_taille, afin de s'assurer que la derniere ligne soit entierement traitée
        for (int x = 0; x <= input.cols-bloc_taille; x += bloc_taille) {
            //calcule ColorDistribution sur le bloc(x, y) -> (x + bbloc_taille, y + bloc_taille):
            Rect bloc(x, y, bloc_taille, bloc_taille);
            ColorDistribution cdBloc = getColorDistribution(input, Point(x, y), Point(x+bloc_taille,y+bloc_taille));
            //On calcule la distance minimale pour chaque distributions, du fond et de l'objet, par rapport à la distribution du bloc
            float dFond = minDistance(cdBloc, col_hists);
            float dObjet = minDistance(cdBloc, col_hists_object);
            //on range la couleur correspondant à l'étiquetage du bloc dans le vecteur de triplet RGB color
            if (dObjet < dFond) {
                //cout<< "rouge" << endl;
                rectangle(img_output, bloc, colors[1], FILLED);
            }
            else {
                rectangle(img_output, bloc, colors[0], FILLED);
            }
        }
    }
    return img_output;
};
```

Le tableau *colors* contient 2 couleurs, mettons noir et rouge, pour désigner la couleur du fond et la couleur de l'objet.

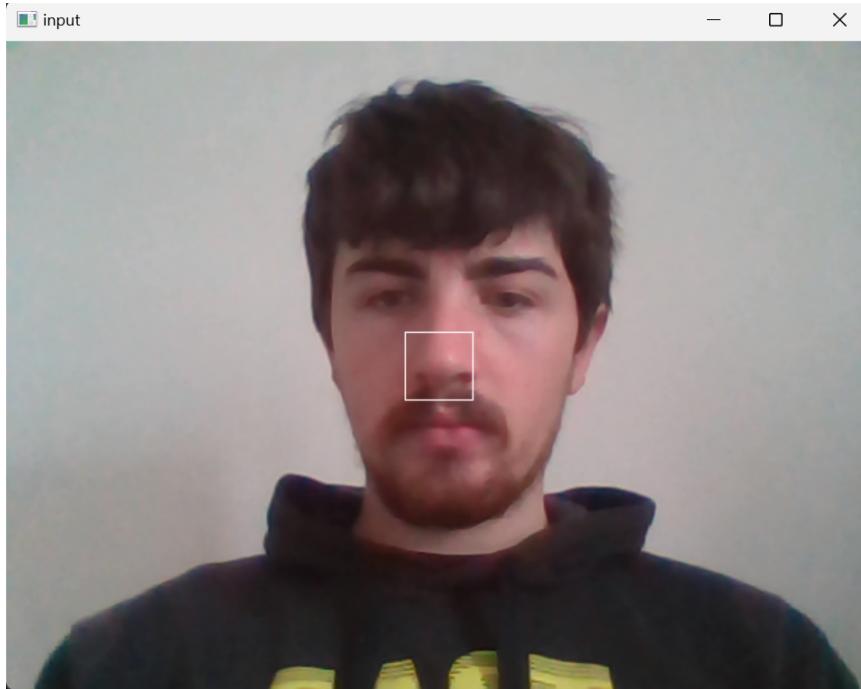
Pour faire un affichage, où on distingue la classification des blocs en temps réel, on peut utiliser le code suivant :

```
Mat output = img_input;
if (c == 'r' && !freeze && col_hists.size()>0 && col_hists_object.size()>0) {
    reconnaissance=!reconnaissance;
}
if (reconnaissance==true) { // mode reconnaissance
    Mat gray;
    cvtColor(img_input, gray, COLOR_BGR2GRAY);
    Mat reco = recoObject(img_input, col_hists, col_hists_object, colors, 16);
    cvtColor(gray, img_input, COLOR_GRAY2BGR);
    output = 0.5 * reco + 0.5 * img_input; // mélange reco + caméra
}
else
    cv::rectangle(img_input, pt1, pt2, Scalar({ 255.0, 255.0, 255.0 }), 1);
imshow("input", output);
```

5.1 Vérification

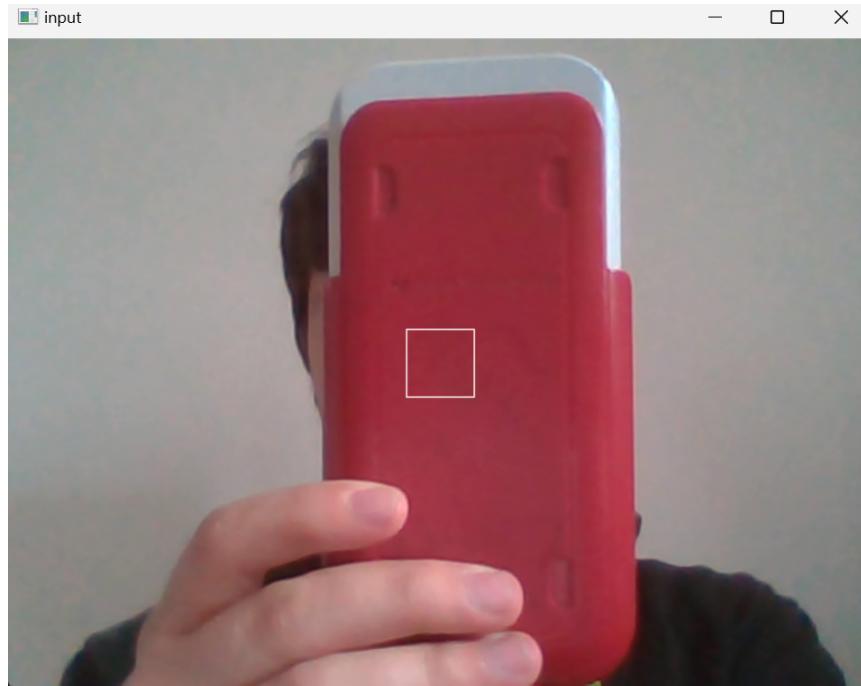
On réalise maintenant un petit test, afin de vérifier que le mode reconnaissance et tout les traitements qu'il entraîne, se déroulent correctement.

On commence par lancer l'application, puis on appuie tout d'abord sur la touche "b", afin de mémoriser les histogrammes des distributions de couleurs du fond :



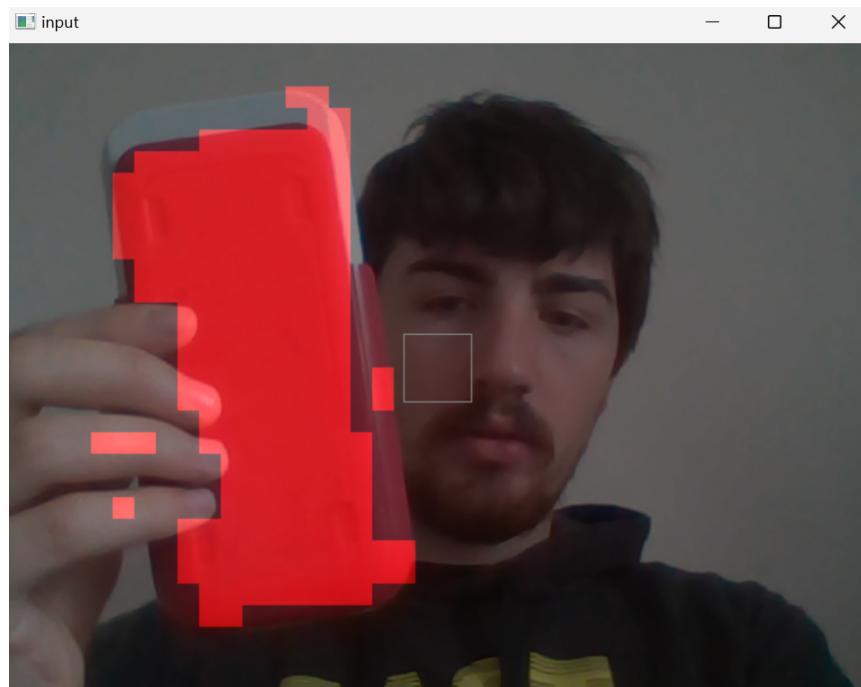
Ensuite, on présente la partie de l'objet à reconnaître, dont on souhaite mémoriser l'histogramme de la distribution de couleurs, dans le petit carré blanc, au centre de l'image et on presse la touche "a".

Afin d'être sûr de mémoriser correctement et entièrement l'objet, on peut appuyer plusieurs fois sur la touche "a", en présentant des parties différentes de l'objet, dans le carré blanc.



Notre objet est maintenant mémorisé, on peut alors tester, si le mode reconnaissance fonctionne correctement.

Pour cela, il suffit d'appuyer sur la touche "r" :



On voit alors les cases, que l'algorithme a reconnu comme appartenant à l'objet, se colorer bien en rouge, comme défini dans le vecteur *colors*.

6 Partie V : Autres développements possibles

Afin d'améliorer notre application, on peut effectuer plusieurs modifications, permettant de rendre plus intéressante et pratique son utilisation.

6.1 Ajouter d'autres objets

La première modification, est la possibilité de mémoriser et de reconnaître plusieurs objets différents, en même temps.

Pour cela, on va pouvoir mémoriser plusieurs distributions de couleurs, pour plusieurs objets, en utilisant un vecteur de vecteur de *ColorDistributions*, dans notre fonction *recoObject*.

On va également modifier le traitement de nos touche "a" et "r", ainsi qu'ajouter une touche "n", permettant d'ajouter le nouvel objet que l'on souhaite reconnaître, dans le vecteur des objets mémorisés.

Ainsi, désormais la touche "a" servira à ajouter un nouvel histogramme, au vecteur des histogrammes de couleurs de l'objet courant, qui n'est pas encore mémorisé et reconnaissable dans le mode reconnaissance.

La touche "n", permettra d'ajouter le vecteur des histogrammes de couleurs de l'objet courant, au vecteur de vecteur d'histogrammes de couleurs, représentant la mémoire des objets mémorisés pour reconnaissance.

Il se chargera ensuite de réinitialiser, le vecteur des histogrammes de couleurs de l'objet courant, afin de pouvoir mémoriser un nouvelle objet, en appuyant une ou plusieurs fois sur la touche "a", puis à nouveau la touche "n".

L'application est capable de mémoriser autant d'objets différents, qu'il y a de couleurs enregistrés, dans le vecteur *colors*. Une fois qu'un objet est ajouté à la mémoire, par la touche "n", on ne peut plus le retirer, il faut relancer l'application et tout recommencer.

Une fois qu'un histogramme est ajouté, au vecteur des histogrammes de couleurs de l'objet courant, il ne peut plus non plus être retirer, il faudra également relancer l'application et tout recommencer.

La touche "r" servira toujours à activer et désactiver le mode reconnaissance.

Voici le nouveau code, pour le traitement de la fonction *recoObjet*, prenant en compte que maintenant, on doit distinguer les blocs découpés de l'image, parmi le fond et plusieurs séries d'histogrammes de couleurs, de plusieurs objets différents :

```
Mat recoObject(Mat input,
    const std::vector< ColorDistribution >& col_hists, /*< les distributions de couleurs du fond */
    const std::vector<std::vector< ColorDistribution >>& all_col_hists_objects, /*< les distributions des objets */
    const std::vector< Vec3b >& colors, /*< les couleurs pour fond/objet */
    const int bloc_taille /*< taille de chaque bloc, 16 si 16x16 */ {
    Mat img_output = Mat::zeros(input.size(), input.type());
    for (int y = 0; y <= input.rows - bloc_taille; y += bloc_taille) { // On met -bloc_taille, afin de ne pas dépasser
        for (int x = 0; x <= input.cols - bloc_taille; x += bloc_taille) {
            //calcule ColorDistribution sur le bloc(x, y) -> (x + bbloc_taille, y + bloc_taille):
            Rect bloc(x, y, bloc_taille, bloc_taille);
            ColorDistribution cdBloc = getColorDistribution(input, Point(x, y), Point(x + bloc_taille, y + bloc_taille));
            //On calcule la distance minimale pour chaque distributions, du fond et des objets, par rapport à la distribution du bloc
            float dFond = minDistance(cdBloc, col_hists);
            int IndexClosestObject = -1;
            float distMin = dFond;
            for (int i = 0; i < all_col_hists_objects.size(); i++) {
                float dist = minDistance(cdBloc, all_col_hists_objects[i]);
                if (dist < distMin) {
                    distMin = dist;
                    IndexClosestObject = i;
                }
            }
        }
    }
}
```

```

    }
    //on range la couleur correspondant à l'étiquetage du bloc dans le vecteur de triplet RGB c
    if (IndexClosestObject == -1) {
        //cout<< "rouge" << endl;
        rectangle(img_output, bloc, colors[0], FILLED);
    }
    else {
        rectangle(img_output, bloc, colors[IndexClosestObject + 1], FILLED);
    }
}
return img_output;
};

```

Voici le nouveau code de la touche "a", qui ajoute l'histogramme de couleurs du petit carré blanc, seulement au vecteur des histogrammes de couleurs, de l'objet courant :

```

if (c == 'a' && !freeze) {
    //calcule ColorDistribution sur le rectangle blanc et on le mémorise dans newObjectHist:
    newObjectHist.push_back(getColorDistribution(img_input, pt1, pt2));
    cout << "Histogramme de l'objet courant, taille: " << newObjectHist.size() << endl;
}

```

Voici le code de la nouvelle touche "n", qui permet de stocker le vecteur des histogrammes de couleurs de l'objet courant, dans la mémoire et de mémoriser un nouvel objet, en réinitialisé ce même vecteur :

```

if (c == 'n') { // Touche pour ajouter un nouvel histogramme d'objet
    // S'il n'est pas vide, on ajoute le nouveau vecteur de distribution de couleurs, correspondant au :
    if (newObjectHist.size() > 0) {
        all_col_hists_objects.push_back(newObjectHist);
        cout << "Ajout du nouvel objet" << endl;
        cout << "Nombres d'objets memorises :" << all_col_hists_objects.size() << endl;
    }
    // On vide l'objet newObjectHist:
    newObjectHist.clear();
    cout << "contenu de newObjectHist :" << newObjectHist.size() << endl;
}

```

Enfin voici le nouveau code de la touche "r", pour le traitement du mode reconnaissance. J'ai supprimé quelques étapes inutiles et qui demandait des ressources, pour fluidifier l'affichage :

```

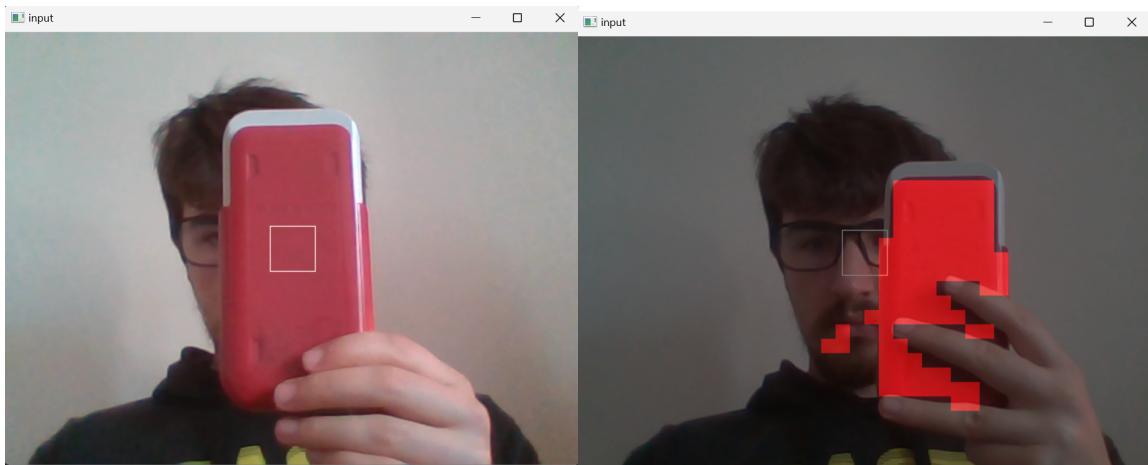
Mat output = img_input;
if (c == 'r' && !freeze) {
    reconnaissance = !reconnaissance;
}
if (reconnaissance == true) { // mode reconnaissance
    Mat reco = recoObject(img_input, col_hists, all_col_hists_objects, colors, 16);
    output = 0.5 * reco + 0.5 * img_input; // mélange reco + caméra
    imshow("input", output);
}
else {
    cv::rectangle(img_input, pt1, pt2, Scalar({ 255.0, 255.0, 255.0 }), 1);
    imshow("input", output);
}

```

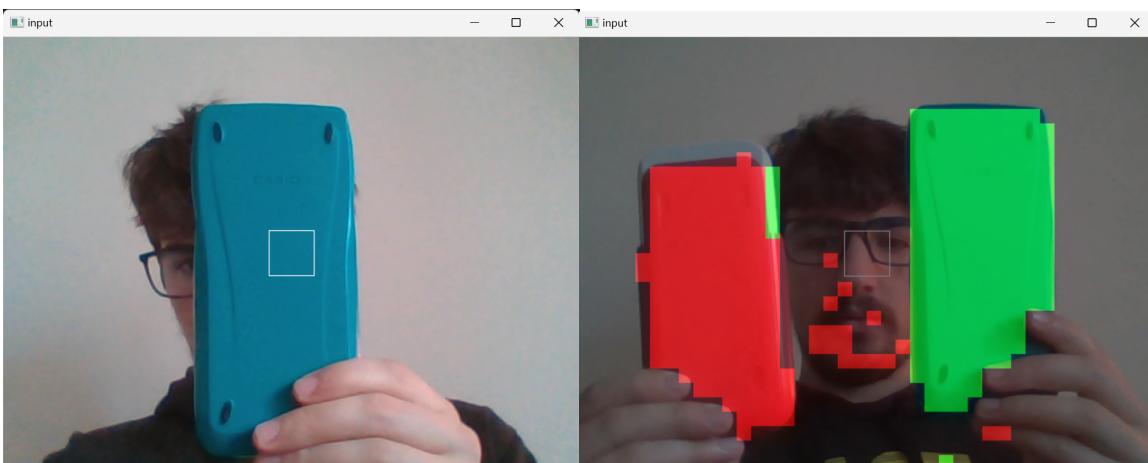
6.2 Vérification

À l'aide d'un nouveau test, on vérifie que l'application peut désormais mémoriser et reconnaître, les distributions de couleurs de plusieurs objets différents en même temps.

On commence par mémoriser le fond, par l'emploi de la touche "b".
 Ensuite, on mémorise un premier objet rouge, avec la combinaison des touches "a" et "n", puis on teste la reconnaissance de ce premier objet, dans le mode reconnaissance avec la touche "r" :

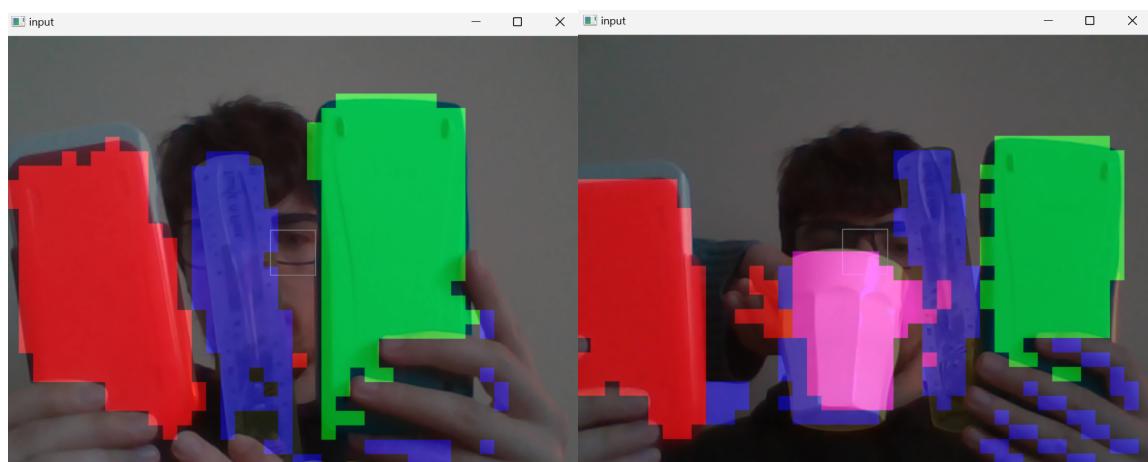


On observe que l'algorithme reconnaît bien ce premier objet. On peut donc en mémoriser un deuxième, avec la même combinaison des touches "a" et "n" et on vérifie de nouveau dans le mode reconnaissance, avec la touche "r" :



On observe que le code fonctionne correctement et que les deux objets sont bien reconnus et distingués, en même temps par l'application.

Si on ajoute de nouvelles couleurs différentes au vecteur *colors*, on peut ensuite mémoriser encore d'autres objets, comme testé ci-dessous, même si l'augmentation du nombre d'objets réduit quelque peu la précision de la reconnaissance :



6.3 Optimiser la vitesse d'exécution

Un gros problème de l'approche est son temps de calcul proportionnel au nombre de distributions apprises. Or, certaines distributions apprises ne sont pas très utiles, car elles sont très proches d'autres distributions. On peut donc simplifier les distributions de couleurs en éliminant les distributions qui ressemblent trop aux autres déjà apprises, c'est-à-dire celles dont la distance aux autres distributions déjà apprises est très petite.

Une façon de faire, est de ne mémoriser une distribution de couleur que si sa distance minimale, à toutes les autres distributions est supérieure, à un seuil donné.

Pour apporter cette modification au traitement de l'application, il a d'abord fallu écrire une nouvelle fonction, qui renvoie un booléen indiquant si la nouvelle distribution de couleurs, que l'on souhaite mémoriser, n'est pas trop proche d'une des distribution de couleurs, déjà mémorisé pour l'objet courant.

voici le code de cette nouvelle fonction :

```
bool isDistinctInObject(const ColorDistribution& newHist, const vector<ColorDistribution>& currentObjectHists) {
    for (const auto& hist : currentObjectHists) {
        if (newHist.distance(hist) < threshold) {
            // La nouvelle distribution est trop proche d'une distribution existante de l'objet courant
            return false;
        }
    }
    // La nouvelle distribution est suffisamment distincte des distributions de l'objet courant
    return true;
};
```

Ensuite, il a fallu également écrire une autre nouvelle fonction, qui renvoie un booléen indiquant que la nouvelle distribution de couleurs que l'on souhaite mémoriser, n'est pas trop des distributions de couleurs, des autres objets déjà mémorisés.

Voici le code de cette nouvelle fonction :

```
// Vérifie si la distribution de couleur du nouvel objet est suffisamment distincte de celles des autres objets
bool isDistinct(const ColorDistribution& newHist, const vector<ColorDistribution>& existingObjectHists,
               const float threshold) {
    for (const auto& hist : existingObjectHists) {
        if (newHist.distance(hist) < threshold) {
            // La nouvelle distribution est trop proche d'une distribution existante
            return false;
        }
    }
    // La nouvelle distribution est suffisamment distincte
    return true;
};
```

Dans la méthode *main*, on ajoute une variable définissant le seuil de distance minimale, permettant de définir si on accepte la nouvelle distribution de couleur, où si elle est trop proche d'une autre déjà en mémoire :

```
const float threshold = 0.05;
```

Étant donné que c'est avec la touche "a", que mon application ajoute une nouvelle distribution de couleurs, à la mémoire de l'objet courant. Il m'a paru plus pratique de ne modifier uniquement le traitement de la touche "a", pour ajouter la vérification du respect de la distance seuil, lors de la mémorisation de nouvelles distributions de couleurs et laisser le traitement associé à la touche "n" intact.

J'ai donc modifié le code de la touche "a", pour ajouter deux étapes de tests. La première étape vérifie que la nouvelle distribution de couleurs, n'est pas trop proche de l'une des distributions, des autres objets précédemment mémorisés.

La seconde étape vérifie la même chose, mais pour les distributions déjà mémorisés, pour l'objet courant. Après chaque test, un message indiquant le résultat, est imprimé afin de savoir, si l'ajout a correctement été effectué et sinon pourquoi.

Voici donc le nouveau code de la touche "a" :

```
if (c == 'a' && !freeze) { // Touche pour ajouter une distribution à l'objet courant
    // création d'une nouvelle distribution de couleur pour l'objet courant
    ColorDistribution newHist = getColorDistribution(img_input, pt1, pt2); // Supposons que vous avez d

    // Première étape : vérifier par rapport aux autres objets mémorisés
    bool isDistinctFromOthers = true;
    for (const auto& otherObjectHists : all_col_hists_objects) {
        if (!isDistinct(newHist, otherObjectHists, threshold)) {
            isDistinctFromOthers = false;
            break; // La nouvelle distribution est trop proche d'une des distributions des autres objets
        }
    }

    if (isDistinctFromOthers) {
        // Seconde étape : vérifier par rapport aux distributions de l'objet courant
        if (isDistinctInObject(newHist, newObjectHist, threshold)) {
            // Si la distribution est distincte, on l'ajoute à l'objet courant
            newObjectHist.push_back(newHist);
            cout << "Distribution ajoutée à l'objet courant." << endl;
        }
        else {
            cout << "La distribution est trop similaire à une existante dans l'objet courant. Pas d'ajout."
        }
    }
    else {
        cout << "La distribution est trop similaire à une distribution d'un autre objet. Pas d'ajout."
    }
}
```

On peut vérifier rapidement que le code fonctionne :

```
Histogrammes de fond, taille: 15
Distribution ajoutée à l'objet courant.
La distribution est trop similaire à une existante dans l'objet courant. Pas d'ajout.
La distribution est trop similaire à une existante dans l'objet courant. Pas d'ajout.
```

Durant cette étape, j'ai également modifié un peu la fonction *recoObject*, pour que l'application ne crache pas, lorsque l'utilisateur essaye de mémoriser plus d'objets, que le nombre de couleurs enregistrés, dans le vecteur *colors* et qu'il obtienne seulement, l'impression d'un message d'alerte à la place.

6.4 Suppression du dernier objet mémorisé

En testant l'application, je me suis aperçu que lorsque je faisais une erreur dans l'enregistrement des objets, je ne pouvais pas revenir en arrière. Il fallait quitter l'application et tout fermer, avant de relancer et de devoir enregistrer de nouveau tous les objets, un par un.

Pour pallier à cet inconvénient et rendre l'application plus pratique à utiliser, j'ai donc décidé d'ajouter un bouton "d", qui permet de supprimer de la mémoire des objets, c'est à dire le vecteur de vecteur de *ColorDistribution*, le dernier objet qui a été mémorisé.

Ceci, dans le but de pouvoir recommencer l'enregistrement d'un objet plus simplement et directement, en cas d'erreur, sans avoir à relancer l'application et tout recommencer.

Il est à noter que si la mémoire des objets est vide, le bouton "d" n'aura aucun effet sur l'application.

Voici le code permettant d'ajouter rapidement et simplement cette fonctionnalité :

```

if (c == 'd') { // Touche pour supprimer le dernier objet enregistre.
    // Check s'il y a au moins un objet memorise
    if (all_col_hists_objects.size() > 0) {
        // Supprime le dernier objet memorise
        all_col_hists_objects.pop_back();
        cout << "Dernier objet supprime." << endl;
    }
    else {
        cout << "Aucun objet à supprimer." << endl;
    }
}

```

On teste facilement, que ce code fonctionne correctement :

```

Ajust du nouvel objet
Nombres d'objets memorises :3
contenu de newObjectHist :0
Dernier objet supprime.
Distribution ajoutee a l'objet courant.
La distribution est trop similaire a une existante dans l'objet courant.
Ajust du nouvel objet
Nombres d'objets memorises :3
contenu de newObjectHist :0
Dernier objet supprime.
Dernier objet supprime.
Distribution ajoutee a l'objet courant.
Ajust du nouvel objet
Nombres d'objets memorises :2
contenu de newObjectHist :0

```

On observe que le nombre d'objets mémorisé est bien décrémenté lorsque l'on presse la touche "d" et que le mode reconnaissance n'identifie plus les distributions de couleur de les objets oubliés.

Un autre modification qui aurait pu être ajouté, aurait consisté à permettre de supprimer, le dernier histogramme de distribution de couleurs enregistré, dans le vecteur de l'objet courant.

Afin de ne pas devoir recommencé à mémoriser, toutes les distributions de l'objet courant, en cas d'erreur sur la dernière enregistré.

Mais ce problème est moins contraignant, depuis l'ajout de la touche "d", car il suffit simplement de presser successivement les touches "n" et "d", pour repartir au début de l'enregistrement de l'objet courant, sans plus avoir besoin de relancer l'application et tout le processus de mémorisation des objets, depuis le début.

7 Conclusion

À la fin de mon développement du projet, voici le code de mon application :

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <opencv2/core/utility.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>

using namespace cv;
using namespace std;

struct ColorDistribution {
    float data[8][8][8]; // l'histogramme
    int nb = 0;           // le nombre d'échantillons

    ColorDistribution() { reset(); }
    ColorDistribution& operator=(const ColorDistribution& other) = default;
    // Met à zéro l'histogramme
    void reset() {
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                for (int k = 0; k < 8; k++) {
                    data[i][j][k] = 0;
                }
            }
        }
    }
};

// Ajoute l'échantillon color à l'histogramme:
// met +1 dans la bonne case de l'histogramme et augmente le nb d'échantillons
void add(Vec3b color) {
    // Vec3b contient 3 entiers entre 0 et 255 la couleur du pixel courant, dans le carré blanc de
    // On souhaite convertir ces valeurs entre 0 et 8, pour incrémenter de 1 la case correspondante
    data[int(floor(color[0] / 32))] [int(floor(color[1] / 32))] [int(floor(color[2] / 32))]++;
    nb++;
};

// Indique qu'on a fini de mettre les échantillons:
// divise chaque valeur du tableau par le nombre d'échantillons
// pour que case représente la proportion des pixels qui ont cette couleur.
void finished() {
    //normalisation des valeurs des histogrammes, en divisant par le nombre d'échantillons:
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            for (int k = 0; k < 8; k++) {
                data[i][j][k] = data[i][j][k] / nb;
            }
        }
    }
};

// Retourne la distance entre cet histogramme et l'histogramme other
float distance(const ColorDistribution& other) const {
    float resultat = 0;
    // Calcul de la distance du CHI-2 entre les histogrammes:
    for (int i = 0; i < 8; i++) {
        for (int j = 0; j < 8; j++) {
            for (int k = 0; k < 8; k++) {
```

```

                if ((data[i][j][k] + other.data[i][j][k]) > 0) {
                    resultat = resultat + ((data[i][j][k] - other.data[i][j][k]) * (data[i][j][k] -
                }
            }
        }
    }
    return resultat;
};

ColorDistribution getColorDistribution(Mat input, Point pt1, Point pt2) {
    ColorDistribution cd;
    for (int y = pt1.y; y < pt2.y; y++)
        for (int x = pt1.x; x < pt2.x; x++)
            cd.add(input.at<Vec3b>(y, x));
    cd.finished();
    return cd;
};

float minDistance(const ColorDistribution& h, const std::vector< ColorDistribution >& hists) {
    float dMin = h.distance(hists[0]);
    for (int i = 1; i < hists.size(); i++) {
        float newD = h.distance(hists[i]);
        if (newD < dMin) {
            dMin = newD;
        }
    }
    return dMin;
};

Mat recoObject(Mat input,
    const std::vector< ColorDistribution >& col_hists, /*< les distributions de couleurs du fond */
    const std::vector<std::vector< ColorDistribution >>& all_col_hists_objects, /*< les distributions des objets */
    const std::vector< Vec3b >& colors, /*< les couleurs pour fond/objet */
    const int bloc_taille /*< taille de chaque bloc, 16 si 16x16 */ {
    Mat img_output = Mat::zeros(input.size(), input.type());
    for (int y = 0; y <= input.rows - bloc_taille; y += bloc_taille) { // On met -bloc_taille, afin de ne pas
        for (int x = 0; x <= input.cols - bloc_taille; x += bloc_taille) {
            //calcule ColorDistribution sur le bloc(x, y) -> (x + bbloc_taille, y + bloc_taille):
            Rect bloc(x, y, bloc_taille, bloc_taille);
            ColorDistribution cdBloc = getColorDistribution(input, Point(x, y), Point(x + bloc_taille, y + bloc_taille));

            //On calcule la distance minimale pour chaque distributions, du fond et des objets, par rapport à ce bloc
            float dFond = minDistance(cdBloc, col_hists);
            int IndexClosestObject = -1;
            float distMin = dFond;
            for (int i = 0; i < all_col_hists_objects.size(); i++) {
                float dist = minDistance(cdBloc, all_col_hists_objects[i]);
                if (dist < distMin) {
                    distMin = dist;
                    IndexClosestObject = i;
                }
            }
            //on range la couleur correspondant à l'étiquetage du bloc dans le vecteur de triplet RGB color
            if (IndexClosestObject == -1) {
                //cout<< "rouge" << endl;
                rectangle(img_output, bloc, colors[0], FILLED);
            }
        }
    }
}

```

```

        else {
            if (all_col_hists_objects.size() <= colors.size() + 1) {
                rectangle(img_output, bloc, colors[IndexClosestObject + 1], FILLED);
            }
            else {
                cout << "Pas assez de couleurs enregistres, par rapport au nombre d'objets en memoire";
            }
        }
    }
    return img_output;
};

// Vérifie si la distribution de couleur du nouvel objet est suffisamment distincte de celles des autres
bool isDistinct(const ColorDistribution& newHist, const vector<ColorDistribution>& existingObjectHists,
    for (const auto& hist : existingObjectHists) {
        if (newHist.distance(hist) < threshold) {
            // La nouvelle distribution est trop proche d'une distribution existante
            return false;
        }
    }
    // La nouvelle distribution est suffisamment distincte
    return true;
};

bool isDistinctInObject(const ColorDistribution& newHist, const vector<ColorDistribution>& currentObjectHists,
    for (const auto& hist : currentObjectHists) {
        if (newHist.distance(hist) < threshold) {
            // La nouvelle distribution est trop proche d'une distribution existante de l'objet courant
            return false;
        }
    }
    // La nouvelle distribution est suffisamment distincte des distributions de l'objet courant
    return true;
};

int main(int argc, char** argv)
{
    Mat img_input, img_seg, img_d_bgr, img_d_hsv, img_d_lab;
    VideoCapture* pCap = nullptr;
    const int width = 640;
    const int height = 480;
    const int size = 50;
    // Ouvre la camera
    pCap = new VideoCapture(0);
    if (!pCap->isOpened()) {
        cout << "Couldn't open image / camera ";
        return 1;
    }
    // Force une camera 640x480 (pas trop grande).
    pCap->set(CAP_PROP_FRAME_WIDTH, 640);
    pCap->set(CAP_PROP_FRAME_HEIGHT, 480);
    (*pCap) >> img_input;
    if (img_input.empty()) return 1; // probleme avec la camera
    Point pt1(width / 2 - size / 2, height / 2 - size / 2);
    Point pt2(width / 2 + size / 2, height / 2 + size / 2);
    std::vector<ColorDistribution> col_hists; // histogrammes du fond
    std::vector<std::vector<ColorDistribution>> all_col_hists_objects; // ensemble des histogrammes des

```

```

std::vector<ColorDistribution> newObjectHist; // Histogramme de l'objet courant.
// Pour le mode reconnaissance:
std::vector< Vec3b > colors;
colors.push_back(Vec3b(0, 0, 0)); // Noir pour le Fond.
colors.push_back(Vec3b(0, 0, 255)); // Rouge pour l'Objet 1.
colors.push_back(Vec3b(0, 255, 0)); // Vert pour l'Objet 2.
colors.push_back(Vec3b(255, 0, 0)); // Bleu pour l'Objet 3.
colors.push_back(Vec3b(255, 0, 255)); // Violet pour l'Objet 4.
const float threshold = 0.05;
namedWindow("input", 1);
imshow("input", img_input);
bool freeze = false;
bool reconnaissance = false;
while (true)
{
    char c = (char)waitKey(50); // attend 50ms -> 20 images/s
    if (pCap != nullptr && !freeze)
        (*pCap) >> img_input; // récupère l'image de la caméra
    if (c == 27 || c == 'q') // permet de quitter l'application
        break;
    if (c == 'f') // permet de geler l'image
        freeze = !freeze;
    cv::rectangle(img_input, pt1, pt2, Scalar({ 255.0, 255.0, 255.0 }), 1);
    imshow("input", img_input); // affiche le flux vidéo
    if (c == 'v' && !freeze) { // test le calcul de la distance chi2
        Point p1(0, 0);
        Point p2(width / 2, height - 1);
        Point p3(width / 2, 0);
        Point p4(width - 1, height - 1);
        ColorDistribution cd1 = getColorDistribution(img_input, p1, p2);
        ColorDistribution cd2 = getColorDistribution(img_input, p3, p4);
        //ICI ajouter la suite pour calculer la distance entre les histogrammes.
        float res = cd1.distance(cd2);
        cout << "distance du chi2:" << res << endl;
    }
    if (c == 'b' && !freeze) { //Calcule des histogrammes de couleurs, sur les différentes parties du fond
        const int bbloc = 128;
        for (int y = 0; y <= height - bbloc; y += bbloc) {
            for (int x = 0; x <= width - bbloc; x += bbloc) {
                //calcule ColorDistribution sur le bloc(x, y) -> (x + bbloc, y + bbloc):
                Point ptHautGauche(x, y);
                Point ptBasDroite(x + bbloc, y + bbloc);
                ColorDistribution cdBloc = getColorDistribution(img_input, ptHautGauche, ptBasDroite);
                //le mémorise dans col_hists:
                col_hists.push_back(cdBloc);
            }
        }
        cout << "Histogrammes de fond, taille: " << col_hists.size() << endl;
        int nb_hists_background = col_hists.size();
    }
    if (c == 'a' && !freeze) { // Touche pour ajouter une distribution à l'objet courant
        // création d'une nouvelle distribution de couleur pour l'objet courant
        ColorDistribution newHist = getColorDistribution(img_input, pt1, pt2); // Supposons que vous
        // Première étape : vérifier par rapport aux autres objets mémorisés
        bool isDistinctFromOthers = true;
        for (const auto& otherObjectHists : all_col_hists_objects) {
            if (!isDistinct(newHist, otherObjectHists, threshold)) {

```

```

        isDistinctFromOthers = false;
        break; // La nouvelle distribution est trop proche d'une des distributions des autres
    }
}

if (isDistinctFromOthers) {
    // Seconde étape : vérifier par rapport aux distributions de l'objet courant
    if (isDistinctInObject(newHist, newObjectHist, threshold)) {
        // Si la distribution est distincte, on l'ajoute à l'objet courant
        newObjectHist.push_back(newHist);
        cout << "Distribution ajoutée à l'objet courant." << endl;
    }
    else {
        cout << "La distribution est trop similaire à une existante dans l'objet courant. Pas d'ajout." << endl;
    }
}
else {
    cout << "La distribution est trop similaire à une distribution d'un autre objet. Pas d'ajout." << endl;
}

Mat output = img_input;
if (c == 'r' && !freeze) {
    reconnaissance = !reconnaissance;
}
if (c == 'n') { // Touche pour ajouter un nouvel histogramme d'objet
    // S'il n'est pas vide, on ajoute le nouveau vecteur de distribution de couleurs, correspondant à l'objet
    if (newObjectHist.size() > 0) {
        all_col_hists_objects.push_back(newObjectHist);
        cout << "Ajout du nouvel objet" << endl;
        cout << "Nombres d'objets memorisés :" << all_col_hists_objects.size() << endl;
    }
    // On vide l'objet newObjectHist:
    newObjectHist.clear();
    cout << "contenu de newObjectHist :" << newObjectHist.size() << endl;
}
if (c == 'd') { // Touche pour supprimer le dernier objet enregistré.
    // Check s'il y a au moins un objet memorisé
    if (all_col_hists_objects.size() > 0) {
        // Supprime le dernier objet mémorisé
        all_col_hists_objects.pop_back();
        cout << "Dernier objet supprimé." << endl;
    }
    else {
        cout << "Aucun objet à supprimer." << endl;
    }
}
if (reconnaissance == true) { // mode reconnaissance
    Mat reco = recoObject(img_input, col_hists, all_col_hists_objects, colors, 16);
    output = 0.5 * reco + 0.5 * img_input; // mélange reco + caméra
    imshow("input", output);
}
else {
    cv::rectangle(img_input, pt1, pt2, Scalar({ 255.0, 255.0, 255.0 }), 1);
    imshow("input", output);
}
}

return 0;
}

```

Avec ce code, on obtient une application avec les fonctionnalités suivantes :

- une touche "b", qui permet de mémoriser les distributions de couleurs du fond.
- une touche "a", qui permet de mémoriser une nouvelle distributions de couleurs, pour l'objet courant.
- une touche "n", qui permet de mémoriser l'ensemble des distributions de couleurs de l'objet courant, dans l'ensemble des objets mémorisés, permettant ainsi de mémoriser un nouvel objet, avec la touche "a" et de voir si le mode reconnaissance peut le reconnaître, avec la touche "r".
- une touche "d", qui permet de supprimer l'ensemble des distributions de couleurs, du dernier objet mémorisé, dans l'ensemble des objets mémorisés par l'application.
- une touche "r", qui permet d'activer et de désactiver le mode reconnaissance, dans lequel les images sont traités, pour reconnaître les distributions de couleurs, des objets mémorisés préalablement avec les touches "a" et "n", en les distinguant des distributions de couleurs du fond de l'image. Des pixels de couleurs apparaissent sur les parties de l'objet, que l'algorithme identifie comme appartenant à l'objet mémorisé. Une couleur est associé à chaque objet mémorisé. L'application peut en mémoriser jusqu'à 4 différents, mais on peut encore ajouter d'autres couleurs au besoin.
- une touche "f", qui permet de geler l'affichage du flux des images capturées, par la caméra.
- une touche "v", qui permet de calculer la distance entre la distributions de couleurs, de la moitié gauche de l'écran et celle de la moitié droite. Cela permet d'estimer un peu, les valeurs retournées par la distance du chi 2 et ainsi de pouvoir choisir une valeur de threshold, pour limiter la mémorisation de distributions de couleurs trop proches.