CHPS0807

Travail d'Etudes

et

de Recherche

Par

Karim KHATER

Introduction :

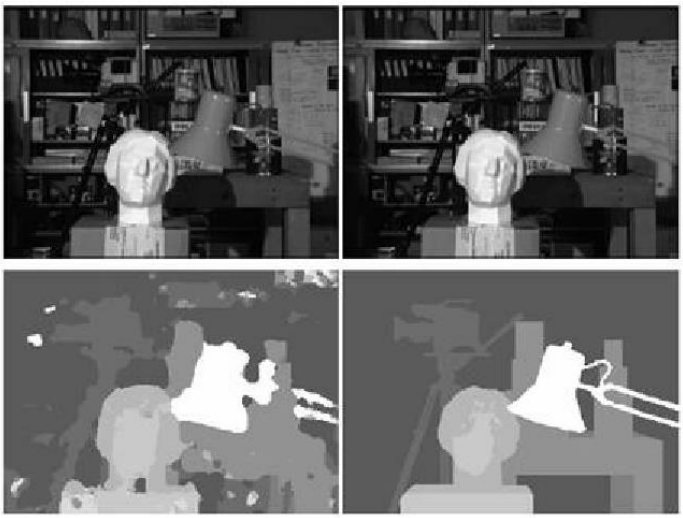
Mon sujet pour le Travail d'Etudes et de Recherche était la Reprojection de cartes de disparité qui est un procédé utilisé dans une variété de domaines comme l'industrie créative, la préservation du patrimoine ou la conduite autonome .

Les approches de reconstruction varient selon le nombre et la disposition des caméras. Sous un certain nombre d'hypothèses de configuration, de disposition et d'alignement d'une grille de caméras. Cette reconstruction est équivalente à une estimation de la disparité.

Formellement, étant donné deux caméras d'une grille de caméras respectivement aux positions (i,j) et (i',j'), si un même point de l'espace est vu par ces deux caméras respectivement aux pixels (x,y) et (x',y'), alors les cartes de disparité associées aux caméras $\Delta i, j$ et $\Delta i', j'$ suivent cette relation :

$$(x', y') = (x + (i - i').\delta, y + (j - j').\delta) \quad \text{(eq 1)}$$

Où δ est la disparité du point dans l'espace et $\delta = \Delta i, j\ (x, y) = \Delta i', j'\ (x', y')$.



Implémentations:

Fonction **EXR_2_PNG()** :

J'ai commencé par comprendre l'équation et comment je peux l'appliquer à un ensemble d'images et de cartes de disparité.

J'ai ensuite programmé une méthode qui prend un chemin vers un dossier contenant la carte de disparité d'un ensemble d'images, puis crée un tableau de 2 dimensions égal au nombre d'images de l'ensemble.

et retourne le tableau contenant le chemin de chaque carte de disparité avec sa position dans la grille des caméras.

Il convertit également le fichier .exr en .png pour visualiser les cartes de disparité.

```
def EXR_2_PNG(path,n):
    size = int(math.sqrt(n))    # to get the size of rows and columns which
has to be equal to each other
    disp_array = np.zeros((size,size)).astype(str)  # array of strings(the
paths to the disparity maps) shape is 2D (4x4)
    index = 0   # for the name of the disparity map
    for i in range(size):
        for j in range(size):
            count = path + "Disp_" + str(index) + ".exr"    # path of each
disparity map
            disp_array[i,j] = count


            # for saving the disparity maps as png to see them
            im_exr = cv2.imread(count,  cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)

            countEXR = "Disp_2_png/Disp_" + str(index) + ".png"
            index = index + 1
            cv2.imwrite(countEXR,im_exr)

    return disp_array
```

Fonction **is_power_of_two():**

Ensuite, je crée une fonction juste pour voir si le nombre d'images fournies est une puissance de deux pour avoir une grille de caméra carrée (4x4).

```
def is_power_of_two(n):
    return (n != 0) and (n & (n-1) == 0)
```

Fonction **cam_pos() :**

Après cela, j'ai une fonction qui est responsable de faire la même chose que EXR_2_PNG mais pour les images. Je prends le chemin du dossier des images et le nombre d'images qu'il contient, puis je renvoie un tableau 2D contenant le chemin de chaque image

```
def cam_pos(path,n):
    if(is_power_of_two(n)):
        size = int(math.sqrt(n)) # to get the number of columns and rows
        imgs_array = np.zeros((size,size)).astype(str)
        index = 0
        for i in range(size):
            for j in range(size):
                imgs_array[i,j] = str(path) + "Image_" + str(index) + ".png" #
path of each image
                index = index +1
    return imgs_array
```

Fonction **project_img()** :

La fonction suivante, son but est de projeter les images donc elle prend comme paramètres la source d'image et l'image qui sera projetée et la carte de disparité de chacune d'elles, et enfin la position de l'image à projeter dans la grille de caméras.

Ensuite je boucle à l'intérieur pour appliquer l'équation à chaque pixel puis je vérifie si la valeur de la position du pixel projeté est hors limite et ensuite je vérifie si la valeur de la disparité à cette position est supérieure ou inférieure à celle de l'image source car si un objet est caché derrière un autre dans l'image en prenant la valeur de disparité la plus élevée renverra l'objet à l'avant.

```
def project_img(img_source,img_res,disp1,disp2,i1,j1,i2,j2):
    rows,columns,_ = img_source.shape
    for x1 in trange(rows):
        for y1 in range(columns):
            x2 = x1 + ((i1 - i2) * disp1[x1,y1])    # x2  = x1 + (i1-i2).δ
            y2 = y1 + ((j1 - j2) * disp1[x1,y1])    # y2  = y2 + (j1-j2).δ

            if(x2 > rows-1 or y2 > columns-1 or x2 < 0 or y2 <0 ) :  # to
check if x2 or y2 are out of range of the rows and columns
                continue

            if(np.sum(disp1[x1,y1]) >= np.sum(disp2[x2,y2])):   # If the value
of disparity of the source is closer we will use it else we will keep the
current value in the pixel
                img_res[x2,y2] = img_source[x1,y1]

    return img_res
```

Fonction **project_disp()** :

Ici, j'applique le même algorithme que project_img(), la seule différence est qu'au lieu de vérifier les valeurs de la disparité, j'applique la valeur la plus élevée à la disparité à projeter au lieu de l'image.

```
def project_disp(disp_source,disp_res,i1,j1,i2,j2):
    rows , columns = disp_source.shape
    print(disp_source.shape)
    for x1 in trange(rows):
        for y1 in range(columns):
            x2 = x1 + ((i1 - i2) * disp_source[x1,y1])  # x2  = x1 + (i1-i2).δ
            y2 = y1 + ((j1 - j2) * disp_source[x1,y1])  # y2  = y2 + (j1-j2).δ

            if(x2 > rows-1 or y2 > columns-1 or x2 < 0 or y2 <0 ) :   # to
check if x2 or y2 are out of range of the rows and columns
                continue
            if(np.sum(disp_source[x1,y1]) >= np.sum(disp_res[x2,y2])): # If
the value of disparity of the source is closer we will use it else we will
keep the current value in the pixel
                disp_res[x2,y2] = disp_source[x1,y1]

    return disp_res
```

Fonction **project()**:

La fonction project est responsable de la mono-projection. Elle prend comme paramètres le chemin vers l'image source, le chemin vers l'image qui sera projetée comme référence, le chemin vers la carte de disparité des deux images, et les positions des deux images.

Il convertit ces entrées en un tableau en utilisant cv2 et NumPy puis il crée une carte de disparité complètement noire et une image à utiliser comme point de départ puis il appelle la fonction disp_projected() pour la disparité et project_img() pour les images et après avoir fini de projeter, je prends la sortie et la compare avec la référence pour voir les différences entre elles.

```
def project(img1,img2,disp1,disp2,x1,y1,x2,y2):

    #read image path
    img1 = cv2.imread(img1)     #[x1,y1]
    img2 = cv2.imread(img2)     #[x2,y2]
    disp1 = cv2.imread(disp1,  cv2.IMREAD_ANYCOLOR | cv2.IMREAD_ANYDEPTH)
#[x1,y1]
    disp2 = cv2.imread(disp2,  cv2.IMREAD_ANYCOLOR | cv2.IMREAD_ANYDEPTH)
#[x2,y2]
    #transform to array
    disp1 = np.asarray(disp1).astype(int)
    disp2 = np.asarray(disp2).astype(int)

    # OBJECTIF 1 PROJECTION OF DISPARITY MAP ONTO ANOTHER POINT OF VIEW

    disp_projected = np.zeros_like(disp1)
    print(" START DISPARITY PROJECTION ")
    disp_projected = project_disp(disp1,disp_projected,x1,y1,x2,y2)
    diff_disp = disp_projected - disp2
    cv2.imwrite("Diff_Disparity_projected_once.png",diff_disp)
    cv2.imwrite("Disp_Projected_once.png",disp_projected)

    # OBJECTIF 2 PROJECTION OF IMAGE
    img_projected = np.zeros_like(img1)
    print(img_projected.shape)
    img_projected = project_img(img1,img_projected,disp1,disp2,x1,y1,x2,y2)
    #print(img_projected)
    diff_img = img_projected - img2
    print("START IMAGE PROJECTION")
    cv2.imwrite("Diff_projected_Image_once.png",diff_img)
    cv2.imwrite("Image_Projected_once.png",img_projected)
```

Fonction **neighbours()** :

Cette fonction calcule les positions des caméras voisines dans la grille utilisée pour les projections multiples. Elle prend en entrée le nombre total d'images qui devrait être de 3x3 ou 4x4 ou plus grand et la position de la caméra qui sera projetée.

Si le nombre de caméras est supérieur à 4, je vérifie si elle est dans un coin ou dans une ligne ou une colonne de début ou de fin ou si elle est au milieu, puis elle crée un tableau 1D de la taille du nombre de voisins multiplié par 2 car il est unidimensionnel et il doit stocker la position des caméras voisines donc [x0,y0,x1,y1,...,xn,yn].

```
def neighbours(nb_total,i,j):
    #To determine the size of the array and to check corners and conditions
    if (nb_total > 4):
        size = int(math.sqrt(nb_total))
        if((i==0 and (j ==0 or j==size-1))or (i==size-1  and (j == size -1 or
j==0))):
            neighbors = 3
        elif(i==0 or j==0 or i==size-1 or j==size-1):
            neighbors = 5
        else:
            neighbors = 8

    array_pos = np.zeros((neighbors*2))
    count = 0
    for x in range(size):
        for y in range(size):
            if(x == i and y == j ):
                continue
            if((x < i+2 and y <j+2)and (x> i-2 and y > j-2 )):
                array_pos[count] = x
                count = count + 1
                array_pos[count] = y
                count = count + 1
                if(count == neighbors*2):
                    break
        if(count == neighbors*2):
                break
    return array_pos
```

Fonction **multiple_projection_img() :**

Cette fonction prend comme paramètres un tableau 2D de cartes de disparité et un tableau 2D d'images et la position de l'image à projeter.

Je commence par appeler la fonction neighbors () pour obtenir la position des caméras voisines.

Ensuite je prends les chemins et les convertit en tableaux puis je boucle, chaque voisin deviendra l'image source ou la disparité source de l'image et il appellera la fonction disp_projected() pour obtenir la disparité qui sera utilisée comme référence dans project_img() que nous appelons après puis après avoir fini de boucler je calcule l'erreur entre l'image projetée et son équivalent dans les images fournies

```
def multiple_projection_img(disps,imgs,i,j):

    neighbours_cams = neighbours(imgs.size,i,j)
    print(neighbours_cams.size)

    img_init = cv2.imread(imgs[i,j])
    img_projected = np.zeros_like(img_init)
    img_projected = np.asarray(img_projected).astype(int)

    disp_init = cv2.imread(disps[i,j],  cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)
    disp_projected = np.zeros_like(disp_init)
    disp_projected = np.asarray(disp_projected).astype(int)
    for x in range(0,len(neighbours_cams),2):
        # Position of a neighbours camera
        pos_x1 = int(neighbours_cams[x])
        pos_y1 = int(neighbours_cams[x+1])

        # Init of source Image
        img_source = cv2.imread(imgs[pos_x1,pos_y1])
        img_source = np.asarray(img_source).astype(int)

        # Init of Disparity of Source Image
        disp_source = cv2.imread(disps[pos_x1,pos_y1],  cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)
        disp_source = np.asarray(disp_source).astype(int)

        # Init of Disparity of Projected Image
        disp_projected =
project_disp(disp_source,disp_projected,pos_x1,pos_y1,i,j)

        img_projected =
project_img(img_source,img_projected,disp_source,disp_projected,pos_x1,pos_y1,
i,j)
 cv2.imwrite("imgs_Projected_multiples.png",img_projected)

    diff_img = img_projected - img_init
    cv2.imwrite("Diff_imgs_multiples.png",diff_img)
```

Fonction **multiple_projection_disp**() :

Ici j'applique le même algorithme que multiple_projection_images() mais avec moins d'entrées car je n'ai pas besoin des images ici

```
def multiple_projection_disp(disps,i,j):
    neighbours_cams = neighbours(disps.size,i,j)
    disp_init = cv2.imread(disps[i,j],  cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)
    disp_projected = np.zeros_like(disp_init)
    disp_projected = np.asarray(disp_projected).astype(int)
    # To apply the function project_disp multiple times depending on the
projected disparity neighbours
    for x in range(0,len(neighbours_cams),2):
        # Position of a neighbours camera
        pos_x1 = int(neighbours_cams[x])
        pos_y1 = int(neighbours_cams[x+1])

        # Init of Disparity map of the image source
        disp_source = cv2.imread(disps[pos_x1,pos_y1],  cv2.IMREAD_ANYCOLOR |
cv2.IMREAD_ANYDEPTH)
        disp_source = np.asarray(disp_source).astype(int)

        disp_projected =
project_disp(disp_source,disp_projected,pos_x1,pos_y1,i,j)

    cv2.imwrite("Disps_Projected_multiples.png",disp_projected)

    diff_disp = disp_projected - disp_init
    cv2.imwrite("Diff_Disparity_disp_multiples.png",diff_disp)
```

problèmes rencontrés :

Tout d'abord, c'était la compréhension des objectifs et de l'équation car au début j'ai compris les objectifs mais j'ai pensé que c'était trop simple ou que je manquais quelque chose donc j'ai décidé de compliquer les choses pendant un mois d'affilée en essayant de créer une carte de disparité moi-même et en l'utilisant avec l'équation qui n'était pas l'objectif.

Le deuxième problème que j'ai rencontré après avoir compris la fonctionnalité et l'objectif était avec les projections multiples car avec une seule projection je ne pouvais pas avoir tous les résultats possibles donc j'ai raté quelque chose j'ai délimité la valeur maximale pour la projection mais j'ai oublié d'ajouter la valeur minimale qui est zéro donc certains de mes résultats avaient des problèmes dans le côté droit de l'image/de la disparité car quand une valeur est négative dans un index de tableau il commence de droite à gauche.

Le troisième problème concernait les projections multiples de l'image. Lorsque j'ai codé, j'ai utilisé la carte de disparité qui m'a été fournie comme référence dans la comparaison entre les deux disparités, donc lorsque je l'ai comparée avec une autre disparité, j'ai obtenu un résultat erroné car la moyenne de la valeur des pixels dans la disparité fournie est plus élevée qu'elle ne devrait l'être, du moins au début alors j’ai utilisé la fonction project_disp() pour obtenir la disparité de l’itération d’image projet et l’utilisé dans project_imgs().

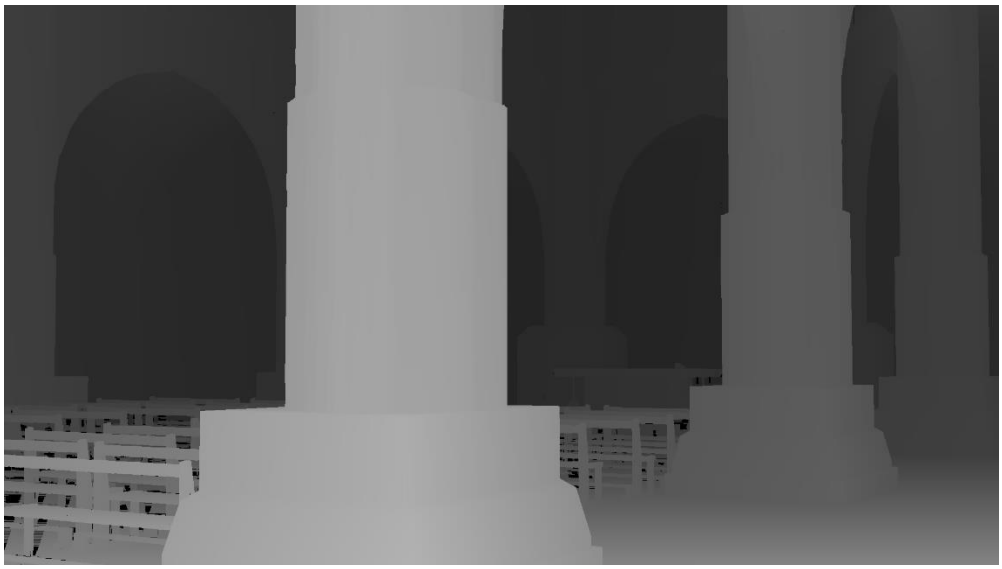
Comparaison des résultats :

Répertoire /Validation/Church :

Taille :4x4

Camera position : (3x3)

Mon disparité projété :



Disp_11.exr

Répertoire /Test/VanGhogh/ :

Mon image géneré :

Image_11.png