ISTIA 2013-2014

Projet tuteuré EI4 Réalité Augmentée

Rapport final

Projet réalisé par :

CASSARD Benjamin
IAZ Zakaria
MASMOUDI Nassim
CATEAU Sébastien

Projet encadré par :

FASQUEL Jean-Baptiste

Remerciements

Dans un premier temps, nous tenons à remercier toute l'équipe pédagogique de l'Institut des Sciences et Techniques de l'Ingénieur d'Angers (ISTIA) et les intervenants responsables de la formation Automatique et Génie Informatique (AGI).

Nous tenons plus particulièrement à remercier notre tuteur responsable de notre projet, Mr Jean-Baptiste FASQUEL, pour nous avoir consacré une partie de son temps afin de nous guider dans la réalisation de ce projet.

Table des matières

Remerciements 1
Introduction (Formation du groupe et contexte) 3
Partie I : Présentation 4
A) La réalité augmentée 4
1) Principe 4
2) Domaines d'application 4
B) Gestion de projet 7
1) Présentation du projet : Cahier des charges 7
2) Planning et étapes 9
3) Répartition des tâches 10
Partie II : Outils et méthodes 11
A) Notions du projet 11
1) Traitement d'image 11
2) Homographie 14
3) Projection et recalage 15
B) Logiciels et librairies 17
1) Computer Vision de J.E Solem 17
2) Python 17
Partie III : Résultats et perspectives 19
1) Les tests finaux 19
2) Difficultés rencontrées 34
3) Perspectives 34
Bibliographie 35
Annexes 36
Fichier Functions.py 36
Fichier Test1.py 40
Fichier Test2.py 43
Fichier Test3.py 47
Fichier Test4.py 49
Fichier Test5.py 51
Fichier Test6.py 57

Introduction (Formation du groupe et contexte)

Nous sommes 4 étudiants de l'ISTIA et dans le cadre de notre projet tuteuré de quatrième année, nous avons réalisé une application de réalité augmentée.

Ce projet nous a été proposé par M. Fasquel, enseignant/chercheur en traitement d'images et développement logiciel.

Nous n'avions aucune expérience en réalité augmentée mais nous avions des notions en traitement d'images qui nous ont été très utiles durant ce projet.

De plus nous ne doutions pas de notre capacité à travailler ensemble. La formation de ce groupe nous a donc parue cohérente et naturelle.

Partie I : Présentation

A) La réalité augmentée

1) Principe

Il faut discerner la réalité augmentée de la réalité virtuelle. En effet ce sont deux choses différentes. Dans l'une il y a une quantité importante de réel, alors qu'en réalité virtuelle, comme son nom l'indique, il n'y a que du virtuel.

Nous pouvons résumer le principe de la réalité augmentée comme étant la superposition en temps réel d'une image virtuelle en deux ou trois dimensions sur les éléments de notre réalité. Ce concept vise à compléter notre perception du monde réel en y ajoutant des éléments fictifs. En d'autres termes, c'est insérer, dans une prise de vue réelle, des éléments virtuels. Cela offre à l'utilisateur la possibilité d'être immergé dans cet environnement mixte.

Deux objectifs peuvent être visés :

->Le premier est d'enrichir la perception du réel avec des images virtuelles.

->Le second est de donner l'impression que la couche virtuelle intègre la réalité.

2) Domaines d'application

Dans quelles applications utilisons-nous la réalité augmentée ?

Les domaines d'application sont très vastes : Shopping, urbanisme, industrie, marketing, immobilier, secteur des jeux vidéo, tourisme et transport... Les applications de la réalité augmentée sur smartphone et sur PC entrent dans notre utilisation quotidienne… c'est la possibilité de voir dans un environnement réel des objets virtuels comme un immeuble, un train, un nouvel aménagement ou l'ancien décor d'un vieux château en offrant à l'utilisateur la possibilité d'être immergé dans cet environnement mixte.

Voici quelques exemples d'application où l'on retrouve de la réalité augmentée :

Exploration urbaine : Découvrir le monde qui nous entoure avec ses charmes et ses animations. L'agence de design lilloise Axone Design a créé un concept de bornes qui diffusent vers les smartphones des informations sur leur environnement immédiat : données thématiques autour de l'histoire, de l'architecture, des arts... La partie "réalité augmentée" est le cœur de la solution mais elle est enrichie d'autres contenus : des textes et des vidéos spécialement créés pour un parcours bien précis. La faisabilité technique du concept va être expérimentée durant six mois dans le parc Montsouris dans le cadre de l'appel à projet Mobilier Urbain Intelligent de la Ville de Paris

Culture :

Grâce à une technologie appelée Héritage 3D, le Château de Vincennes (ancienne demeure des rois de France) a permis à ses visiteurs de découvrir le cabinet de travail de Charles V à l'occasion de Futur en Seine, la première fête du numérique en Ile-de-France.
Equipé d'un moniteur doté d'une double caméra, «l'une orientée vers l'utilisateur, l'autre vers la pièce», le visiteur voit apparaître sur l'écran le cabinet tel qu'il était à l'époque de Charles le Sage**.** La pièce actuelle a d'abord été entièrement numérisée, et tout a été recréé dans les proportions exactes grâce à de longues recherches.
Une quarantaine de marqueurs ont ainsi été positionnés sur les murs de la pièce, permettant à la console de savoir où elle se situait et ce qu'elle devait montrer.

Autre application, celle de CultureClic, éditée par l'agence i-Marginal, entraîne les visiteurs des monuments et musées de l'Hexagone dans un véritable voyage au cœur temps grâce à la réalité augmentée : découverte du Louvre à travers cinq siècles, premières photos au monde de Notre-Dame et de Montmartre en 1841, construction du pont Neuf, Place Royale de Bordeaux au XVIIIe siècle, port de Marseille en 1900... CultureClic est basée sur le catalogue de la Réunion des musées nationaux et les archives Gallica de la Bibliothèque nationale de France.

Shopping : S’habiller virtuellement

Une Agence de Communication vient de présenter une application mixant réalité augmentée et capture de mouvements. La cible est les clients de boutiques de vente de vêtements en ligne. L’idée est de leur offrir la possibilité d’essayer virtuellement les produits**.**
Le fonctionnement est assez simple, on choisit un vêtement, on imprime le code, le vêtement s’ajuste à notre morphologie et reste « fixé » sur nous**.**

Autre exemple, Atol propose l'essayage de lunettes en 3D sans réglages sur son site web. Avec la webcam de son ordinateur, l'internaute peut essayer les nouvelles montures, en changeant leur aspect, grâce à la réalité virtuelle.

Urbanisme :

Dans les rue de Paris et d'Ile-de-France, il suffit de lancer le navigateur de réalité augmentée Layar et de pointer la caméra de son smartphone sur un immeuble résidentiel pour voir son prix au m² et son adresse s'afficher instantanément à l'écran par-dessus l'image de la caméra. L'application est éditée par MeilleursAgents, courtier en agences immobilières. La société accompagne, via son site web meilleursagents.com, les propriétaires d'immobilier résidentiel souhaitant vendre leur bien rapidement et au juste prix du marché.

Marketing :

S’il y a bien une marque de jouet qui ne vieillit pas ce sont les LEGO. Pour cela la célèbre marque de briques se renouvèle constamment et a dernièrement misé sur l’intégration de la réalité augmenté pour un effet très apprécié chez les plus jeunes. Ils proposent aux enfants en magasin de passer la boîte de jeu devant une caméra afin de visualiser dans leurs mains les constructions possibles grâce à la boîte en question.

B) Gestion de projet

1) Présentation du projet : Cahier des charges

L'objectif de notre projet est de faire interagir une plateforme réelle et un objet virtuel de notre choix.

Principe :

Une caméra capte les images du monde réel dans lequel se trouve notre objet de référence. Notre programme est chargé de récupérer les images du monde réel, de créer un monde virtuel à partir de ses images. Pour cela le programme est chargé de reconnaître des points caractéristiques de l'objet de référence pour détecter ses déplacements et de calculer la matrice qui va permettre de les appliquer à notre objet virtuel. Il est ensuite chargé de générer des images en 3 dimensions en temps réel qui seront affichées sur le monde réel. On peut donc synthétiser notre programme en 3 actions majeures :

- Faire le lien entre les points-clés de la nouvelle image et ceux de l'image de référence
- En déduire la matrice qui correspond aux transformations de l'image
- Recaler l'objet virtuel en fonction de cette matrice

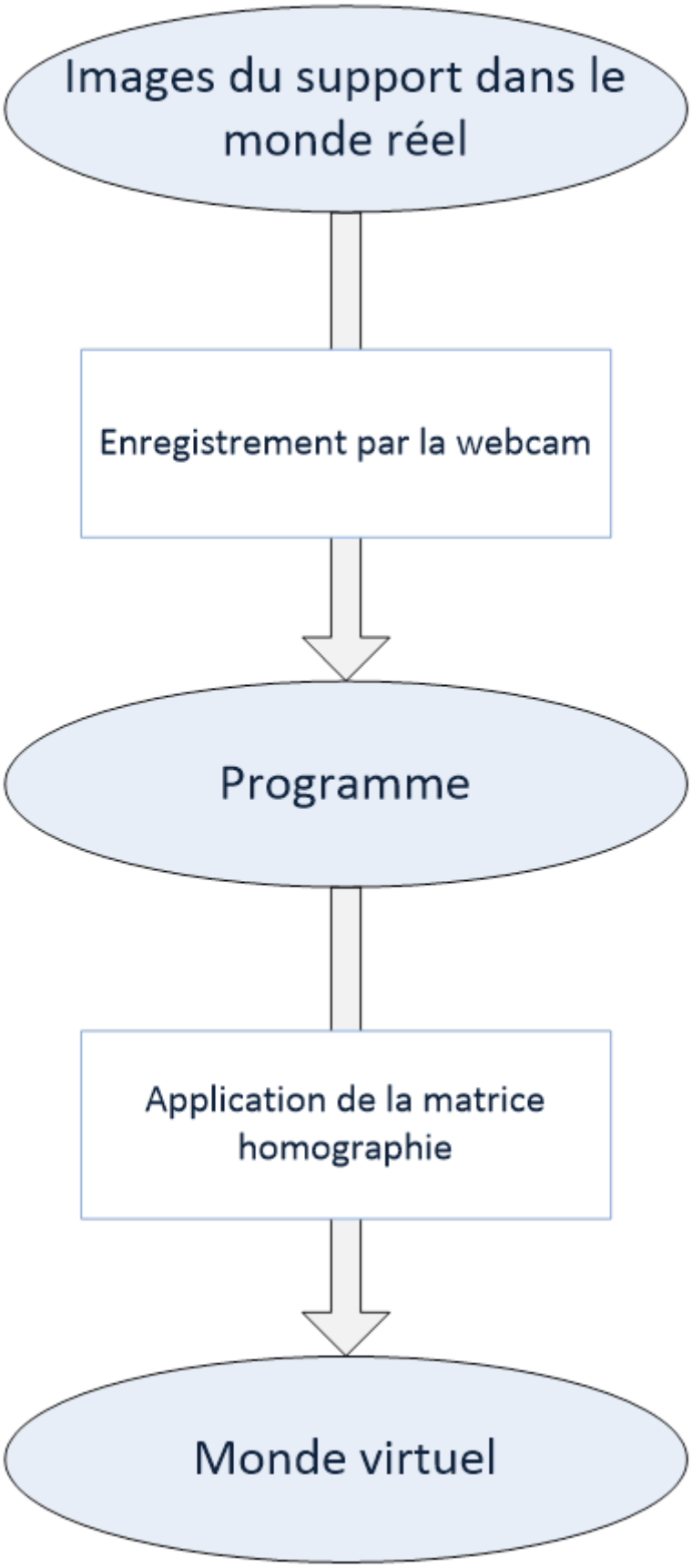


Figure 1 _ Schéma de principe projet

2) Planning et étapes

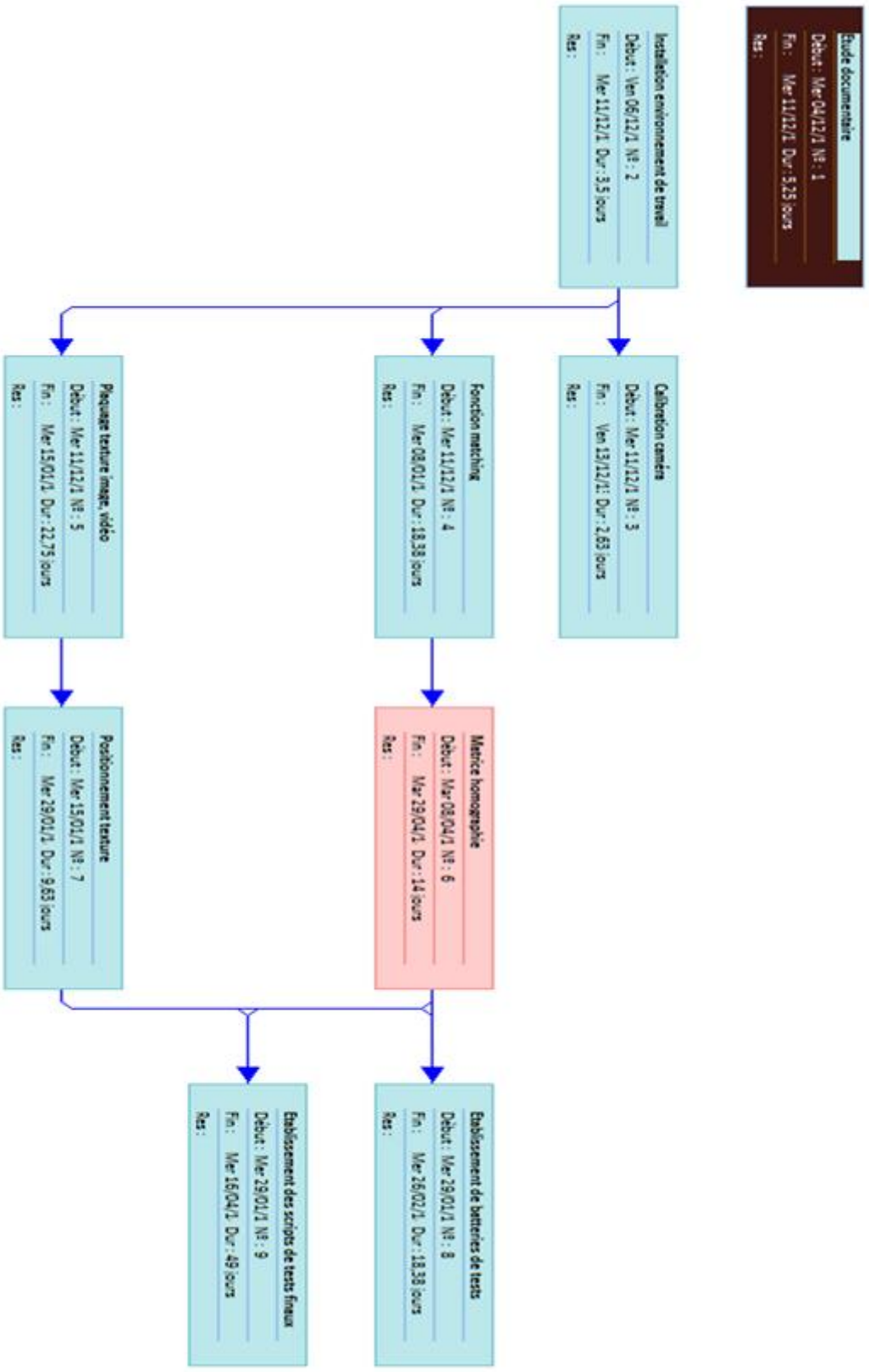


Figure 2 _ Diagramme de PERT

3) Répartition des tâches

Voici un diagramme qui met en lumière le temps utilisé par chaque tâche pour mener ce projet.

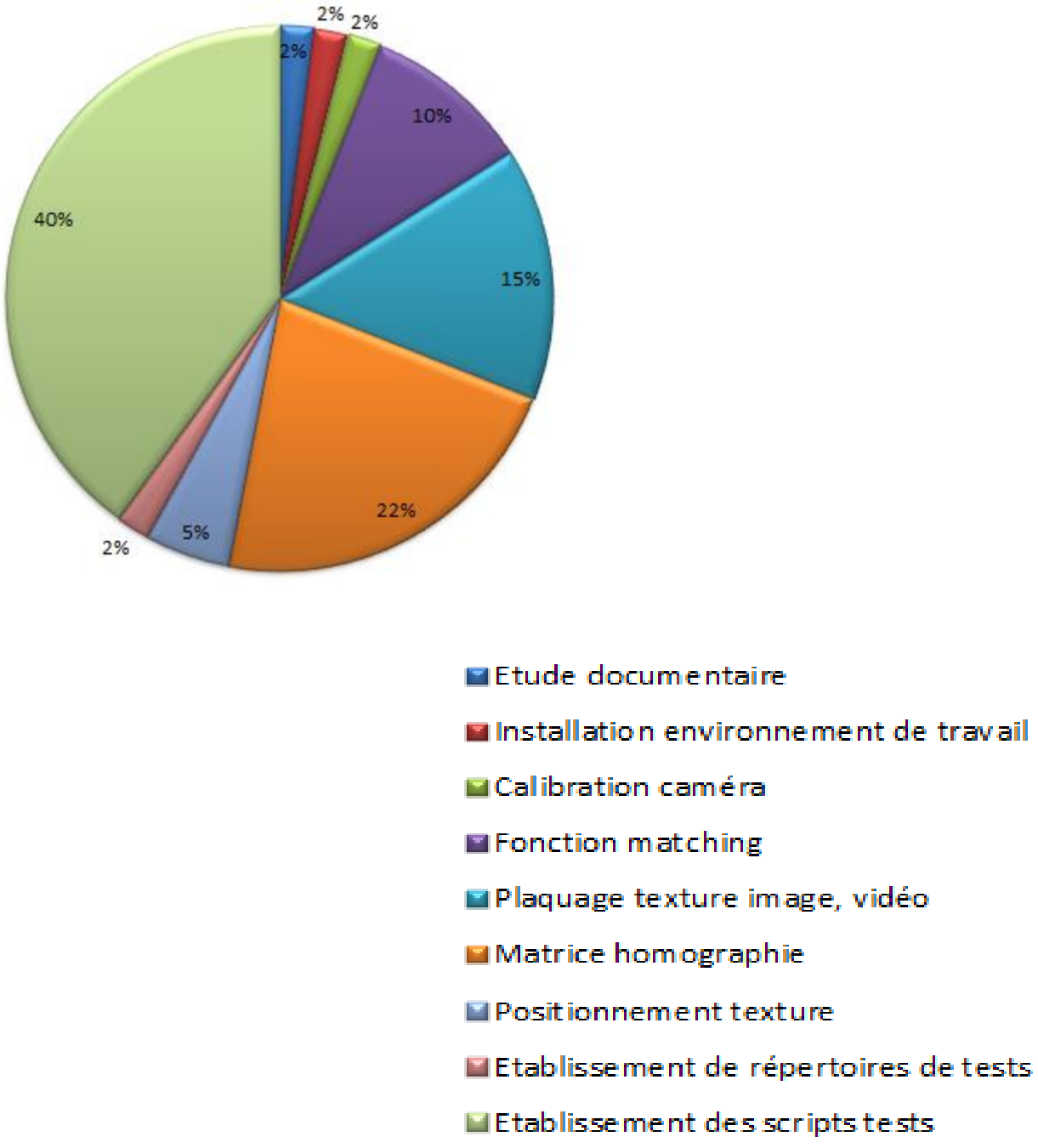


Figure 3 _ Diagramme de répartition des tâches

Partie II : Outils et méthodes

A) Notions du projet

1) Traitement d'image

Le traitement d'image est une partie très importante du projet, les images sont enregistrées puis traitées afin de tirer au mieux les informations qui nous sont utiles. Certaines étapes de nos programmes ont nécessité une conversion de l'image en niveaux de gris afin d'optimiser les temps de calcul mais la partie la plus conséquente du traitement d'image dans notre projet se matérialise par la fonction de reconnaissance de points (matching). On utilisera précisément la fonction *match_twosided* tirée de la librairie SIFT. C'est cette fonction qui permet de faire le lien entre l'image de base de l'objet de référence et la nouvelle image. Elle est conçue de façon à reconnaître les points-clés de l'objet de référence (le support) et de les retrouver dans n'importe qu'elle image afin de les associer. Pour chaque point, le programme va calculer un angle (grâce à fonction arcos) en faisant des produits scalaires entre deux vecteurs. La correspondance est déterminée selon la valeur de cet angle. En effet, si cet angle vaut 0, c'est que les vecteurs sont alignés, si l'angle est très faible cela indique une correspondance entre les vecteurs sinon si l'angle est élevé c'est qu'il n'y a pas correspondance. Les calculs de produit scalaires nécessitent que les vecteurs soient normalisés (valeur module entre 0 et 1) pour que la matrice de l'angle ne soient composées que de valeur comprises entre -1 et 1. La reconnaissance peut être perturbée à cause des points « bruits » de l'image, c'est-à-dire ceux qui ne font pas partie de la forme à repérer mais que le programme détecte. Nous avons utilisé un filtre qui permet de les isoler. Ce filtre utilise l'algorithme RANSAC. C'est un algorithme qui permet de créer une estimation d'un modèle mathématique à partir d'un ensemble de données. Il a l'avantage par rapport à la méthode des moindres carrés de ne pas être perturbé par des valeurs aberrantes, c'est exactement ce qu'on recherche dans notre cas.

On peut voir un exemple d'estimation d'un modèle sur la figure () ci-dessous. On voit que malgré un nombre important de points aberrants, RANSAC

parvient à estimer un modèle mathématique plus précis qu'une estimation linéaire et même relativement proche du modèle réel.

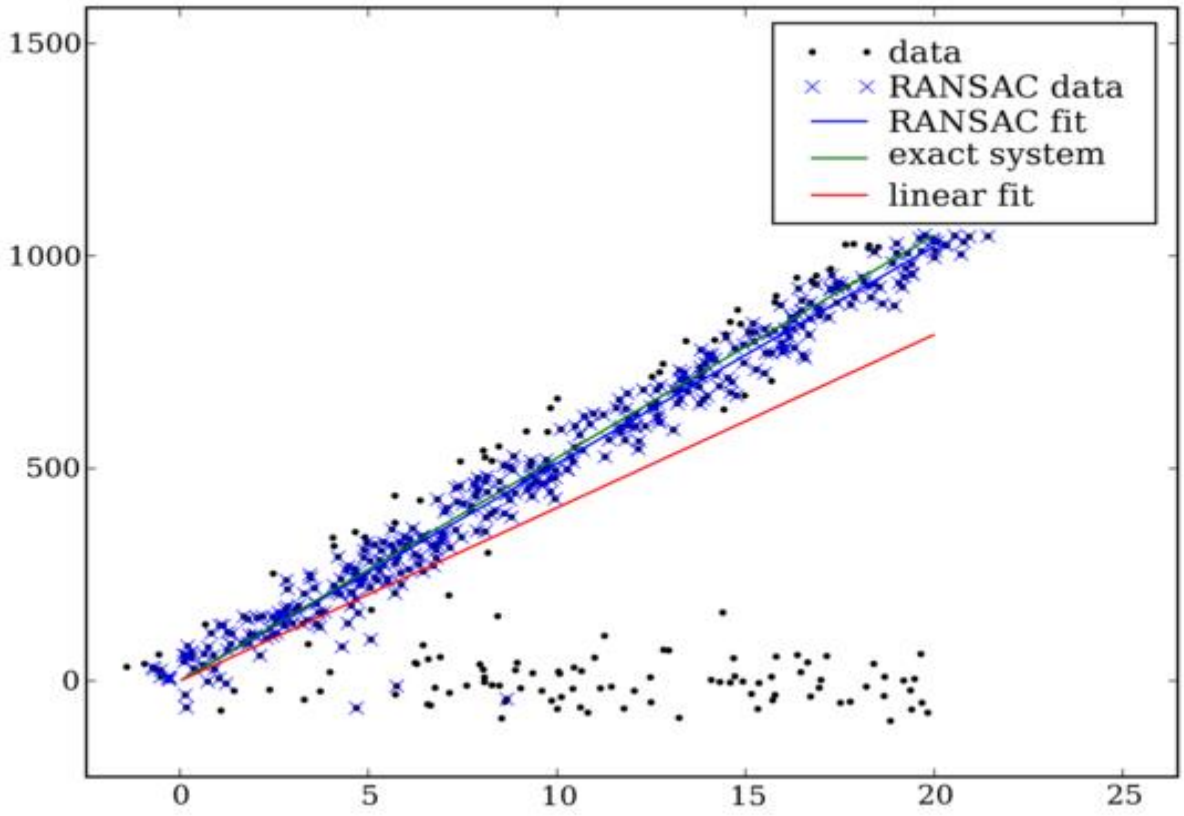


Figure 4 _ estimation d'après RANSAC
Source dans la bibliographie

Voici ci-dessous un exemple de matching. L'image de référence est une carte, la dame de pique, le but est de la retrouver dans la seconde image parmi d'autres objets. Ce que fait très bien l'algorithme, on voit notamment la correspondance entre les points-clés de la carte tels les points qui forment le pique ou encore ceux qui forment la dame.

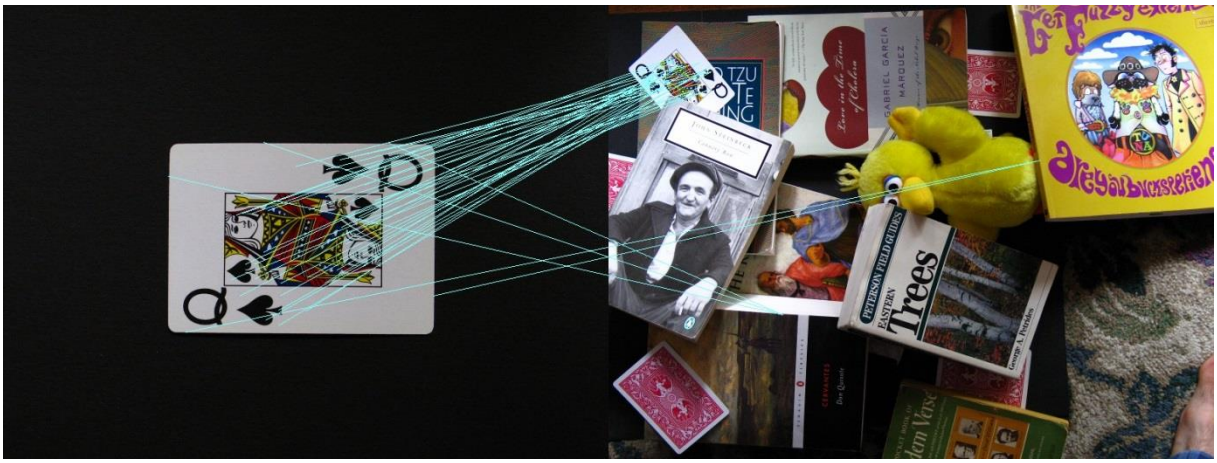


Figure 5 _ Matching grâce à SIFT
Source dans la bibliographie

Voyons ce que cela donne avec notre programme. Dans ce cas-ci, le support, c'est-à-dire l'image que l'on souhaite reconnaître, est le livret de la webcam, on effectue une simple translation de ce support vers la gauche. Cette opération est effectuée sur un fond qui peut nous paraître unie, hors pour l'algorithme de matching il ne l'est pas du tout vu sa sensibilité. C'est la raison pour laquelle des point aberrants sont trouvés et mis en correspondance avec d'autres points de la seconde image qui n'ont aucun lien avec les points initiaux.

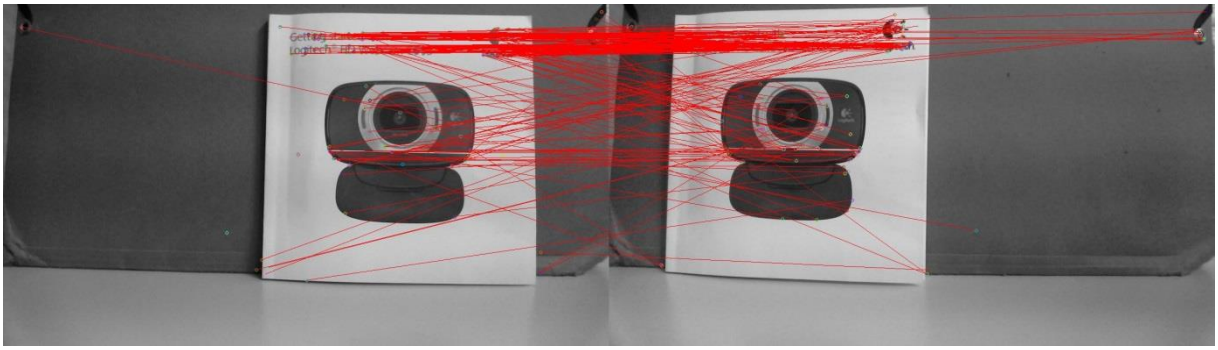


Figure 6_ Association des keypoints

2) Homographie

Nous avons précédemment mis en correspondance les points de deux images grâce au *match_twosided*. Nous avons donc une relation d'un plan à l'autre telle *p α H q*, le but est de pouvoir en tirer l'homographie *H* pour ensuite l'appliquer à notre objet virtuel.

Rappelons qu'une homographie est une transformation projective 2D qui fait correspondre les points dans un plan à l'autre. Dans notre cas, les plans sont des images ou des surfaces planes en 3D. L'homographie peut être utilisée dans différents cas tels que l'enregistrement des images, la rectification des images, de la texture de déformation, la création de panoramas.

Soit x' = H x, voici la représentation classique de cette homographie :

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

3) Projection et recalage

Une notion à bien comprendre dans notre projet est celle des différents mondes par lesquels nous passons. Un point d'un objet réel possède des coordonnées dans un monde en 3 dimensions. Hors la caméra enregistre des images en 2 dimensions, le programme récupère donc les coordonnées de l'objet en 2D. Les points sont ensuite projetés dans un environnement virtuel qui lui est en 3D. Ce monde virtuel est ensuite reprojeté en 2D pour être affiché à l'écran. Ce sont ces deux dernières projections qui sont délicates.

Ci-dessous un schéma qui correspond à la matrice homographie précédente :

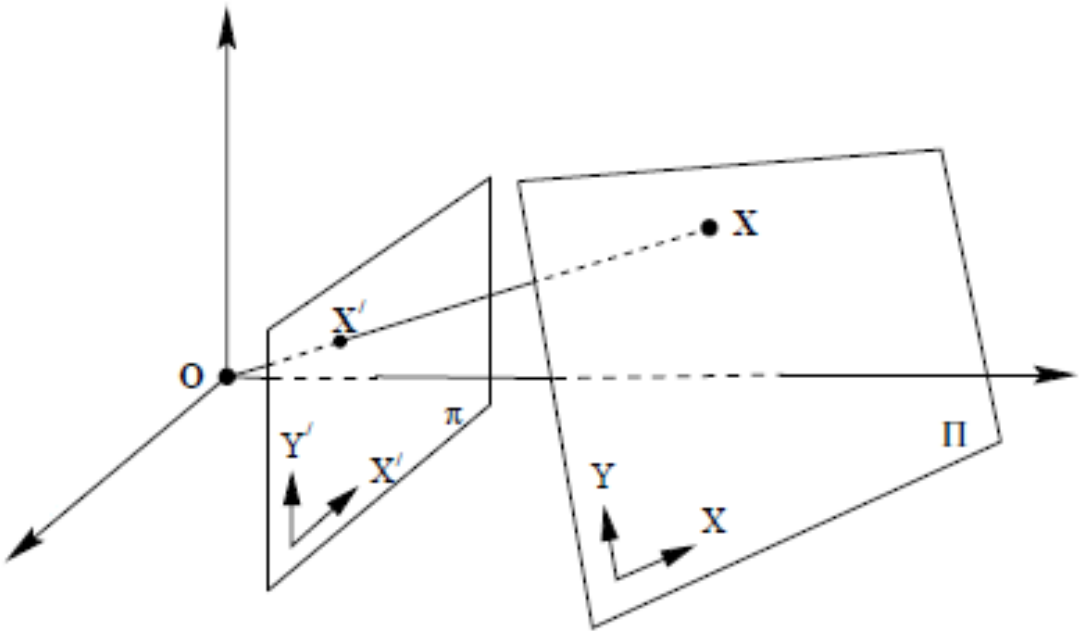


Figure 7 _ Représentation d'un point dans plusieurs mondes

Pour passer d'un point en 3D du monde réel à sa projection dans notre environnement virtuel on a donc une première projection, puis on applique la matrice homographie. Les 2 schémas ci-dessous résument ce principe :

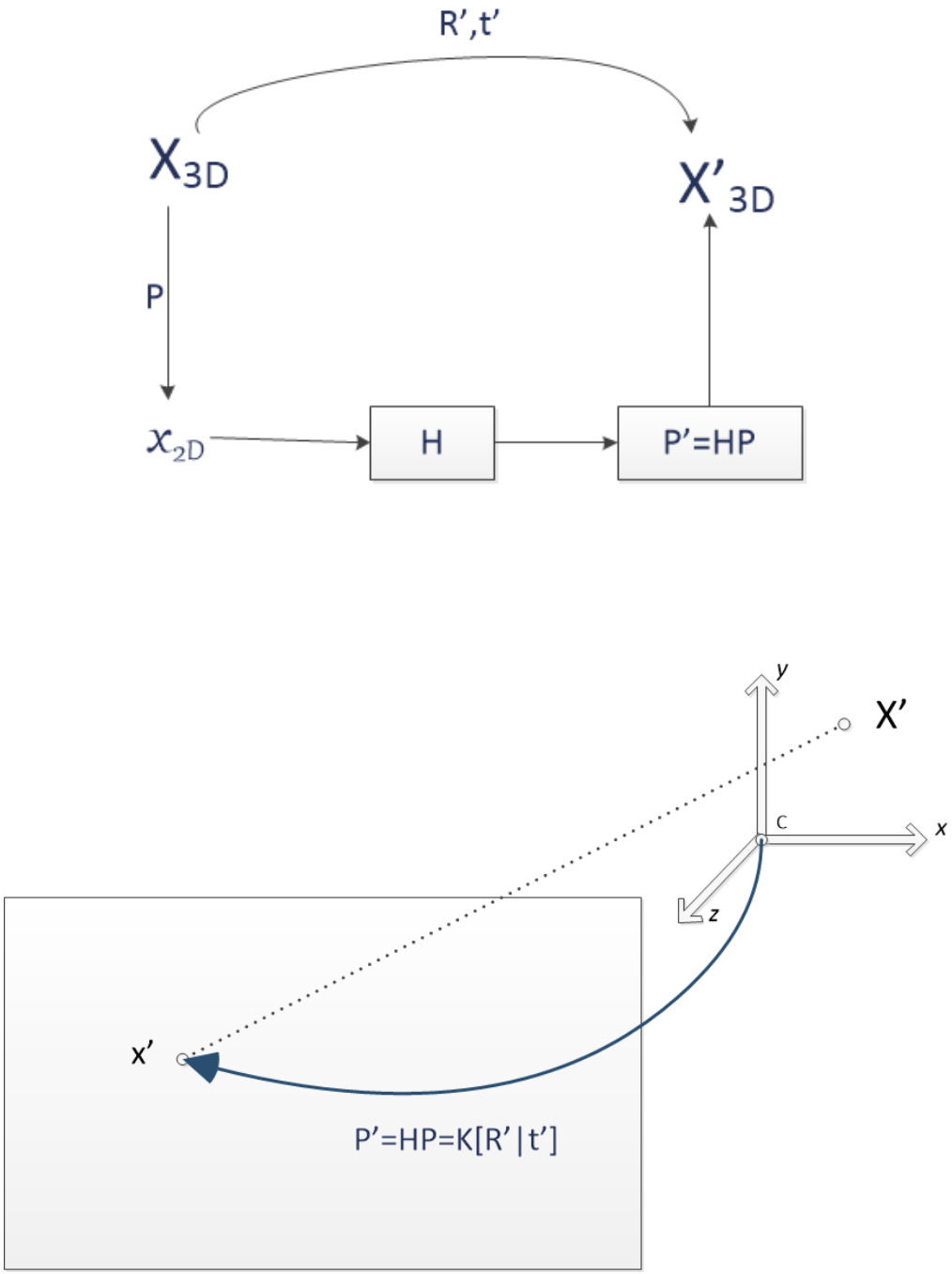
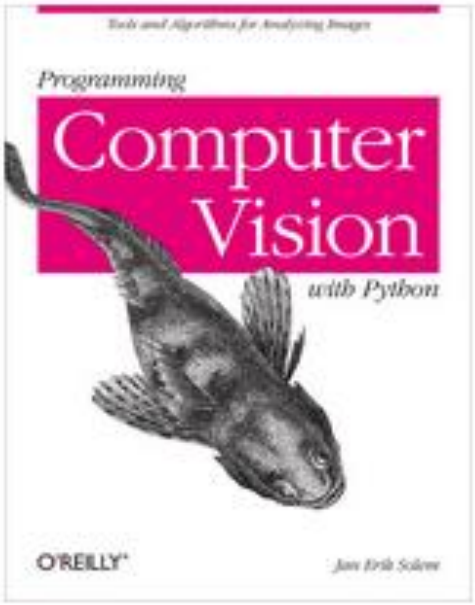


Figure 8 _ Résumé principe projection

B) Logiciels et librairies

1) Computer Vision de J.E Solem

Tous nos travaux sont inspirés du livre « Computer Vision » de Jan Erik Solem, un professeur suédois spécialisé dans l'analyse d'images. Il a été publié en 2012. Ce livre permet une compréhension de base de la théorie et des algorithmes appliqués à la vision par ordinateur. On y apprend des techniques pour la reconnaissance d'objets, reconstruction 3D, l'image stéréo, la réalité augmentée, et d'autres applications de vision par ordinateur. Notamment grâce à des exemples de code complets avec des explications sur la façon de reproduire et de construire sur chaque exemple. Tous ces exemples sont écrits en Python.



2) Python

Le langage Python est un langage de programmation orienté objet mais également multi-plateformes. Il est apparu en 1990, créé par Guido van Rossum un développeur néerlandais. Les principaux avantages du Python sont qu'il possède un typage dynamique fort qui permet à son utilisateur de commettre très peu d'erreur lors de manipulation des variables, il possède également une gestion de la mémoire automatique et un système de gestion des exceptions permettant au développeur de déboguer son programme aisément.

De nos jours, le langage python est utilisé dans beaucoup de domaine. On le retrouve notamment en développement Web et internet comme le framework Django. Il est également utilisé dans le domaine scientifique et numérique grâce à ses librairies Scipy, Numpy et bien d'autres encore.

Nous utilisons donc le langage Python pour réaliser notre application car c'est un langage puissant et simple d'utilisation pour des utilisateurs peu expérimentés. Il est énormément utilisé en modélisation 3D.

Figure 9 - Langage Python

Partie III : Résultats et perspectives

1) Les tests finaux

a) Test1.py : Plaquage d'un objet virtuel sur image

Pour ce projet nous avons utilisé deux technologies pour l'affichage, Pygame et OpenGL. Le premier permet de créer des interfaces graphiques et le deuxième est une librairie graphique d'affichage en trois dimensions.

Pour effectuer les premiers tests, il faut auparavant faire une calibration en fonction de l'objectif. On récupère une matrice qui va ensuite être utilisé pour la projection.

gluPerspective(fovy,aspect,near,far)

Le premier test est une application des scripts donnés dans le livre de Solem. Cela a été une première approche et nous a permis de mettre en place les principales techniques pour ce projet. Le principe est simple, on charge une image que l'on plaque comme texture puis on vient placer un objet virtuel.

Le script se décompose en quatre fonctions. La première permet de calibrer la matrice de projection (*GL_PROJECTION*) en fonction de notre matrice de calibration. C'est ici que l'on définit le mode de projection (perspective ou orthogonale). Dans ce cas ça sera le mode perspective.

Ensuite, il faut s'occuper de la matrice qui permet de positionner les objets sur la scène ta(*GL_MODELVIEW*). C'est grâce à cette matrice que l'on pourra faire bouger notre objet virtuel. C'est une matrice 4X4 de la forme :

$\begin{bmatrix} R & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$, où R est la matrice de rotation et t le vecteur de translation.

Dans un espace à trois dimensions, la matrice de rotation est de la forme :

$$R_{\mathbf{x}}(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix}, \quad R_{\mathbf{y}}(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix}, \quad R_{\mathbf{z}}(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

, où θ est l'angle de rotation.

La troisième étape consiste à afficher l'image de fond sur laquelle on veut afficher l'objet virtuel. Ici, on utilise toujours OpenGL pour créer un *quad* de la taille de l'écran dont les coordonnées vont de -1 à 1 pour les trois axes. L'image est chargée grâce à Pygame et convertie en texture OpenGL et placée dans le *quad*.

Il ne nous reste plus qu'à dessiner l'objet virtuel, on utilise pour cela la librairie OpenGL. Elle contient des objets utilisables facilement, on utilisera ici une théière (*SolidTeapot*) que l'on affiche au centre de notre écran.

Le *main* de cette première partie est de cette forme-là, avec toutes les étapes décrites ci-dessous :

```
# main method
def main():
    #load camera
    with open('ar_cameraHD.pkl','r') as f:
        K = pickle.load(f)
        Rt = pickle.load(f)
    # initialisation de Pygame
    setup()

    draw_background('image.jpg')
    set_projection_from_camera(K)
    set_modelview_from_camera(Rt)
    draw_teapot(0.05)

    # Prise en compte des modifications et affichage
    pygame.display.flip()
```

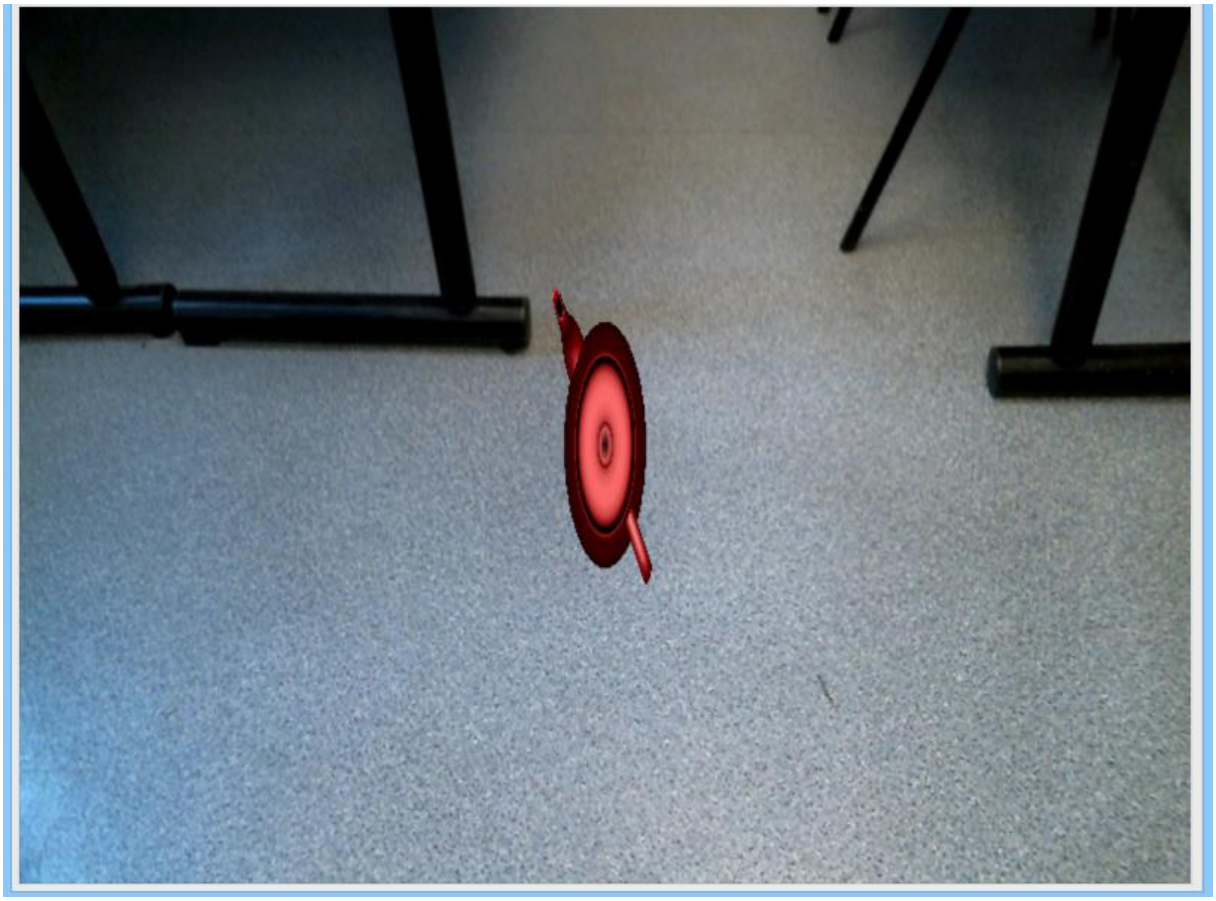


Figure 10 - Rendu du premier test

b) Test2.py : Plaquage de l'objet sur un flux vidéo

Une fois le principe compris, il est simple de remplacer l'image statique par plusieurs images à la suite. Nous avons créé une boucle et à chaque itération, on rafraichi l'affichage par des images provenant d'une webcam. Une librairie python permet de récupérer les images provenant de la webcam.

Dans le premier test l'image était enregistrée sur le disque dur puis traitée, mais cela pouvait entrainer des pertes de performances. C'est pour cela que l'on a choisi de convertir l'image pour qu'elle soit plaquée en tant que texture sans être enregistrée. Cela permet un gain en performance.

On récupère l'image provenant de la webcam que l'on envoie directement dans la fonction d'affichage de l'image :

image = cam.getImage()
draw_background(image)

La fonction est, elle aussi, modifiée pour permettre de traiter cette nouvelle variable qui n'est plus le nom de l'image mais une image de type PIL.

image_surface = pygame.image.frombuffer(image.tostring(), image.size, image.mode)
bg_image = image_surface.convert()

Contrairement à la version précédente, le type de projection a été changé. Il est maintenant orthogonal. Nous avons fait ce choix, car nous avions des difficultés lors de la projection, en effet il y avait un léger décalage lors du déplacement de l'objet virtuel par rapport à la souris.

glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);

La différence entre les deux méthodes de projection est que chaque projection a une approche différente des dimensions.

L'objet virtuel est toujours affiché et peut même être déplacé en fonction des mouvements de la souris. Pour mettre en mouvement notre objet virtuel nous utilisons la matrice vue précédemment. La rotation reste inchangée mais le vecteur de translation prend les valeurs x et y de la souris. Ces coordonnées sont adaptées pour correspondre au bon monde dans lequel la matrice les applique. Comme dit précédemment, on travaille avec la libraire Pygame et OpenGL et chacun a son repère.

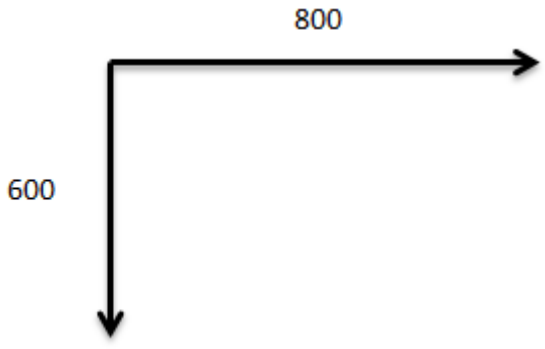


Figure 11 - Monde Pygame

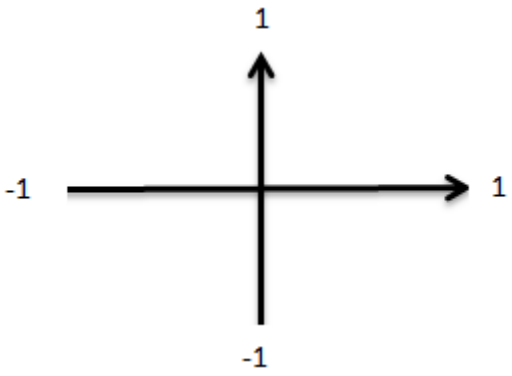


Figure 12 - Monde OpenGL

Il faut donc faire la correspondance entre les deux repères, la formule est simple :

xmin,xmax,ymin,ymax = -1.0,1.0,-1.0,1.0
currentPositionX = x * (xmax-xmin) / width + xmin
y = height - y
currentPositionY = y * (ymax-ymin) / height + ymin

c) Test3.py

L'objectif de ce test est de charger deux images pour ainsi calculer les descripteurs. Par la suite il nous faut détecter les correspondances et afficher les 2 images côte à côte avec des traits liant les points et descripteurs appareillés.

Nous devons tout d'abord commencer par initialiser l'algorithme SIFT à l'aide d'OpenCV *cv2.SIFT()*. La valeur passée en paramètre n'a pas d'incidence par la suite.

La prochaine étape consiste à lire deux images à l'aide de la fonction *imread()* de la librairie OpenCV. Elle prend en paramètre deux informations. Le premier est le chemin de l'image, le deuxième consiste à convertir l'image couleur en

noir et blanc. La deuxième image représente l'objet de la première image mais cette fois-ci avec une transformation qui peut être une translation et/ou une rotation selon les axes x,y. Ces images ont été créées à partir de photos prises.

L'étape suivante consiste à détecter les keypoints et les descripteurs de l'image. Les keypoints contiennent les coordonnées x et y du point et les descripteurs contiennent une taille ainsi qu'un angle qui spécifie l'orientation du descripteur. Cette fonction prend seulement en paramètre l'image (d'autre paramètres sont possibles mais nous ne les utilisons pas ici) et retourne, dans un premier temps, une liste de Keypoints puis, dans un second temps, un tableau de descripteurs. Les keypoints et les descripteurs sont associés. (Le premier keypoint correspond au premier descripteur)

l0, d0 = mysift.detectAndCompute(img0,None)

l1, d1 = mysift.detectAndCompute(img1,None)

Nous devons ensuite dessiner ces keypoints sur deux autres images afin de pouvoir dessiner les lignes ultérieurement. Nous utilisons la fonction *drawKeypoints* qui prend en paramètre l'image sur laquelle nous allons effectuer le traitement et ses keypoints associés. Cette fonction nous retourne l'image avec les keypoints dessinés.

Nous devons par la suite convertir les keypoints(l0,l1) en tableau car nous avons besoin d'un tableau d'integer et non un tuple pour construire l'homographie. Cela dit avant cela, nous conservons ce tuple dans une autre variable afin d'éviter de multiples conversions. Nous utiliserons cette nouvelle variable pour la création des lignes liant les points des deux images.

Par la suite nous devons chercher dans les deux descripteurs, qui correspond respectivement à l'image une et deux, les éléments qui correspondent. La

fonction *match_twosided()* retourne un tableau d'éléments correspondant aux points similaires entre les deux images.

matches = sift.match_twosided(d0,d1)

Une fois les correspondances trouvées, nous devons convertir ces données en coordonnées homogènes afin de pouvoir calculer l'homographie. Nous utilisons la fonction *make_homog* de la classe [Homography.py] provenant du livre de Jan Erik Solem “Programming computer vision with python”. Rappelons que l'homographie est une matrice composée de la translation ainsi que de la rotation en x et y entre les deux images. La fonction prend en paramètres les coordonnées x, y des points de correspondances. Elle retourne un tableau avec chaque coordonnées des points(x,y,z) définis par une colonne.

fp = homography.make_homog(l0[ndx,:2].T)

tp = homography.make_homog(l1[ndx2,:2].T)

Figure 13 _ Résultat fp

Lorsque nous avons récupéré tous les points, il ne nous reste plus qu'à calculer l'homographie via la méthode *H_from_ransac* issue de la classe

[Homographie.py]. Mais avant cela nous devons initialiser le model RANSAC grâce à la fonction *RansacModel()* de la classe [Homography.py]. La fonction *RansacModel()* nous renvoie le model ainsi créé. Par la suite nous appelons la fonctions *H_From_Ransac()* en prenant comme paramètres les coordonnées homogènes des points (fp,tp) et le model créé précédemment. La fonction *H_From_Ransac()* retourne la matrice homographie H et également un tableau contenant les meilleures correspondances.

Il ne nous reste plus qu'à créer une nouvelle image contenant les deux images juxtaposées. Il faut par la suite tracer les lignes liant les points et descripteurs associés avec la fonction *cv2.line* qui prend en paramètres notre nouvelle image précédemment créée, les deux listes de points des deux images et la couleur des lignes.

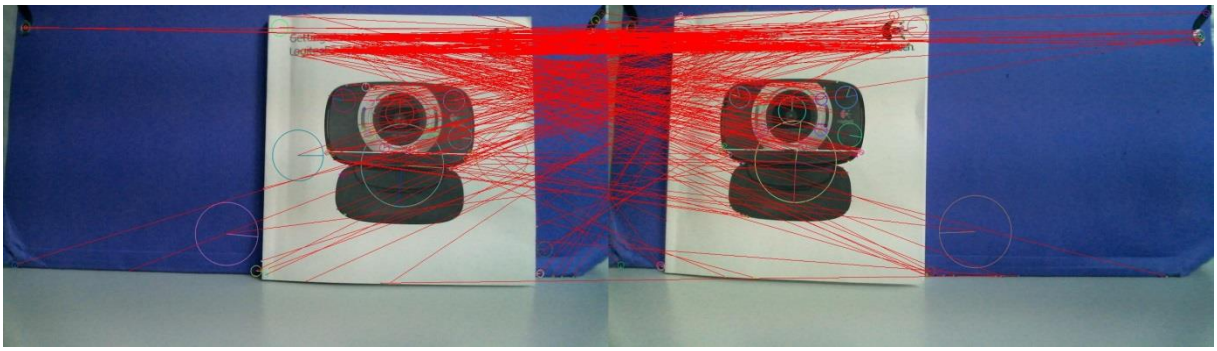


Figure 14 _ Résultat du Test3

d) Test4.py

L'objectif de ce test est de charger deux images, calculer les descripteurs SIFT puis détecter les correspondances pour enfin afficher la première image avec l'objet virtuel puis la seconde avec l'objet virtuel recalé et enfin la dernière image avec l'objet virtuel recalé en 3D.

Dans ce test nous réutilisons le code généré dans le Test3 pour charger les deux images, calculer les descripteurs SIFT et détecter les correspondances.

Commençons par initialiser l'algorithme SIFT. Ensuite nous chargeons les deux images (la deuxième étant décalée de la première). Ensuite nous calculons les descripteurs et les keypoints pour pouvoir ainsi calculer les correspondances entre les deux images. Puis nous initialisons le model RANSAC. Par la suite, nous utilisons les résultats précédents pour obtenir la matrice homographie *H* (fonction *H_From_Ransac()* de la classe [Homography.py]).

Une fois la matrice *H* récupérée, il ne nous reste plus qu'à ajouter l'objet virtuel sur les images. Nous prenons des points fixes pour ajouter l'objet virtuel à la première image, cet objet sera par la suite recalé par rapport à la position définie sur la première image.

Commençons par calibrer la camera avec la fonction *my_calibration* qui provient du livre de Jan Erik Solem. Cette fonction prend en paramètre la résolution de l'image. Les paramètres internes de la fonction de calibration ont été calculés par rapport à la caméra utilisée pour prendre les photos.

Ensuite nous créons l'objet virtuel qui sera dans notre cas un cube. La fonction qui nous a permis de créer le cube a également été fourni par Jan Erik Solem.

Puis nous créons l'objet *cam1* qui initialise la camera avec la calibration effectuée précédemment. Cette fonction nommée "*Camera"* est donnée par la classe [Camera] fournit par Jan Erik Solem. Après cela, nous projetons le cube avec la méthode *project* de la classe [Camera] :

box_cam1 = cam1.project(homography.make_homog(box[:,:5]))

Nous pouvons également créer la projection du cube recalé pour la deuxième image en multipliant la matrice H et la première projection (*box_cam1*).

box_trans = homography.normalize(dot(H,box_cam1))

Pour afficher l'objet virtuel en 3D sur la troisième image, nous devons recréer un nouvel objet *cam2* qui est la multiplication de *cam1* et la matrice *H*.

cam2 = camera.Camera(dot(H,cam1.P))

On modifie par la suite le bloc 3x3 de la matrice de la camera2 pour obtenir une projection 3D.

Puis nous projetons le cube à travers la camera2.

box_cam2 = cam2.project(homography.make_homog(box))

Pour finir, il ne nous reste plus qu'à afficher l'objet virtuel sur l'image à l'aide de la fonction *plot* de la librairie Matplotlib.

plot(box_cam1[0,:],box_cam1[1,:],linewidth=3)

plot(box_trans[0,:],box_trans[1,:],linewidth=3)

plot(box_cam2[0,:],box_cam2[1,:],linewidth=3)

Figure 15 _ Projection d'un carré virtuel en 2D

Figure 16 _ Projection 2D avec H

Figure 17 _ Reconstruction 3D sur la deuxième image

e) Test5.py

Le cinquième test est une première combinaison de tous les tests déjà effectués. La différence est que l'on n'utilise plus Matplotlib pour l'affiche mais la combinaison pygame/OpenGL. On commence par charger deux images auxquelles on applique l'algorithme SIFT. Ensuite celles-ci sont affichées tour à tour avec l'objet virtuel suivant l'élément de référence.

box_cam1,box_cam2,H = algorithme_sift(K)

On récupère la matrice homographie H et deux tableaux de points correspondant aux rectangles dessinés dans la version précédente. Pour dessiner notre objet virtuel, nous avons décidé de le dessiner au centre du rectangle.

x = (box_cam1[0,0]+box_cam1[0,2])/2
y = (box_cam1[1,0]+box_cam1[1,2])/2

La matrice H contient les informations de translations et de rotations. Une « teapot » est affichée avec la première image puis celle-ci est affichée avec la deuxième image au même endroit que l'image de référence.

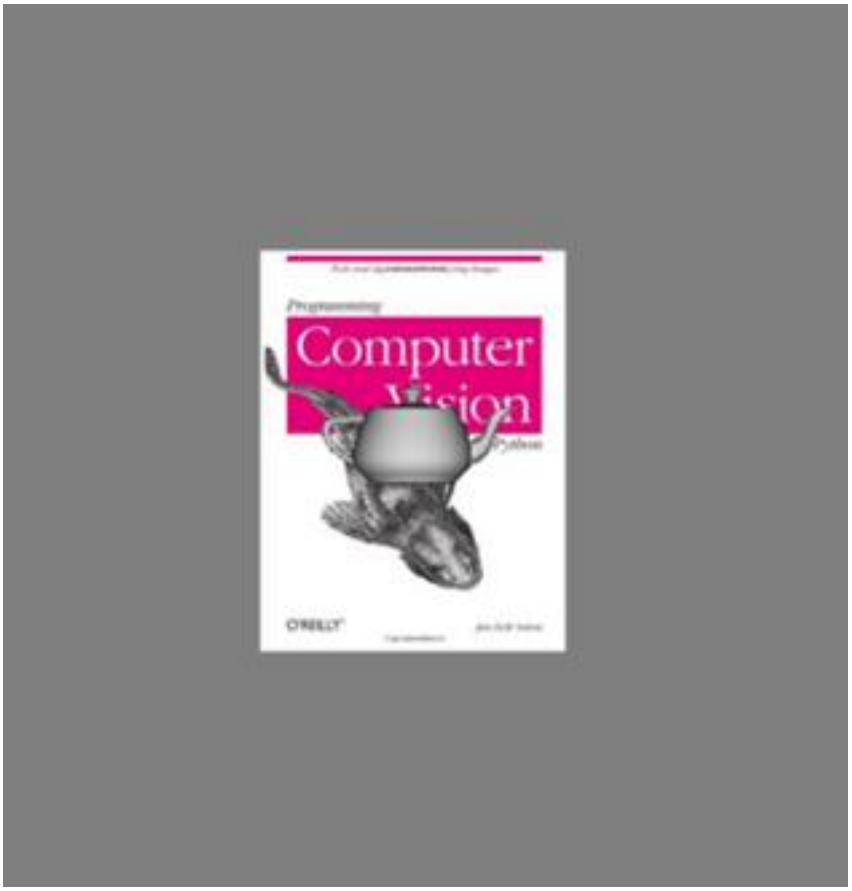


Figure 18 - Première image

Figure 19 - Deuxième image avec la *teapot* toujours sur l'élément de référence

f) Test6.py

L'objectif de ce test est de lire un flux vidéo via une webcam ou bien un fichier enregistré afin de placer l'objet virtuel sur chaque image de cette vidéo. Pour des raisons de communication nous générerons un fichier vidéo de sortie comprenant la vidéo d'origine avec l'ajout de l'objet virtuel recalé pour chaque image.

Dans un premier temps nous devons ouvrir le flux vidéo. Le paramètre *filepath* étant le chemin de la vidéo à lire.

video_reader=skimage.io.Video(filepath)

On crée une variable *previous* qui contient la première image de la vidéo, puis on la convertit en noir et blanc.

previous=video_reader.get()

previous=skimage.color.rgb2gray(previous)

On crée la vidéo de sortie qui sera construite image par image après avoir ajouté l'objet virtuel.

writer = cv.CreateVideoWriter("testVideoOut.avi",cv.CV_FOURCC('X', 'V', 'I', 'D'),20,(previous.shape[1],previous.shape[0]),is_color=1)

Par la suite nous devons parcourir le flux vidéo d'entrée image par image afin de pouvoir effectuer le traitement (Ajout de l'objet virtuel) sur chaque image.

for i in range(0,np.uint(video_reader.frame_count()-1)):

Dans cette boucle nous créons une variable *current_rgb* qui récupère l'image en cours et nous effectuons une conversion d'image du type numpy vers OpenCV.

current_rgb=video_reader.get()

cv_rgb=op.numpyrgb2opencvrgb(current_rgb)

On affiche la vidéo avec la fonction *ShowImage* de la librairie OpenCV. Puis on effectue le traitement qui permet d'ajouter l'objet virtuel à l'image.

im = op.operation(np.array(previous),np.array(current_rgb))

Les fonctions *numpyrgb2opencvrgb()*, *my_calibration()* et *operation()* sont implémentées dans la classe [functions.py].

La fonction *operation()* comprend presque le même code que le Test4.py car nous devons retourner une image, ce qui va nous permettre de l'enregistrer dans notre buffer vidéo nommé *writer*.

Il faut également redessiner à la main avec la fonction *Rectangle* de la librairie OpenCV car dans le Test4.py nous dessinions directement sur la fenêtre avec Matplotlib tandis qu'ici nous sommes obligés de dessiner dans l'image afin de pouvoir l'enregistrer aisément sans perte de résolution.

Cette ligne permet de dessiner le cube à l'aide de la projection du cube.

cv.Rectangle(cv_rgb, (int(box_trans[0,0]),int(box_trans[1,0])),(int(box_trans[0,2]),int(box_trans[1,2])), cv.RGB(255,0,0), thickness=5)

Nous utilisons la fonction *WriteFrame()* de la librairie OpenCV pour écrire dans le buffer vidéo "*writer*" qui prend en paramètre le buffer et l'image.

cv.WriteFrame(writer,im)

Cependant ce script n'est à l'heure actuelle non fonctionnel en raison d'un problème d'écriture dans le fichier vidéo de sortie. L'erreur provient soit de la fonction *CreateVideoWriter* soit de la fonction *WriteFrame* car aucun fichier *videoOut* n'est généré.

2) Difficultés rencontrées

Tout d'abord, contrairement à d'autres languages et à d'autre IDE, avec PythonXY tous les libraires ne sont pas intégrées dans le logiciel. Nous avons donc eu souvent des erreurs concernant le manque d'un package ou d'un autre.

Une mauvaise calibration de la caméra donnait des problèmes de projection lorsqu'on travallait avec pygame et opengl.

Une grande difficulté était de jongler entre les différents formats des libraires. Par exemple, la librairie numpy n'a pas la même façon de déclarer une image que la libraie opencv ou la libraire PIL. Il fallait donc faire des convertions assez régulièrement entre les différents types de données.

3) Perspectives

Bien sûr il y a possibilité d'optimiser nos scripts notamment pour gagner en fluidité. On peut par exemple diviser notre application en deux *thread*. C'est-à-dire qu'il y aurait deux tâches en parallèle, la première s'occuperait de l'affichage de la texture et l'autre de l'algorithme SIFT. Cela permettrait d'afficher l'objet virtuel uniquement quand le calcul est terminé.

Nous sommes actuellement obligés d'effectuer une saisie de deux images avec un intervalle de 5 secondes notamment dans le test 5. L'idéal serait de diminuer le plus possible ce délai de saisie de l'image afin de s'approcher d'une véritable application en temps réel.

L'un des objectifs possible est aussi d'étendre l'application utilisable sur Galaxy Tab. Nous n'avons pas pu aborder cet aspect. Mais il n'existe notamment une application, QPython, qui permet d'exécuter des scripts écris en Python sur un smartphone Android.

Bibliographie

http://www.loria.fr/~gsimon/ra/

https://code.google.com/p/pythonxy/wiki/StandardPlugins#Extensions_Python

http://lentreprise.lexpress.fr/publicite-et-communication/realite-augmentee-15-exemples-d-applications-pratiques_30360.html?p=1#content

http://jodul.developpez.com/tutoriels/android/utiliser-sift/

image SIFT :

http://robwhess.github.io/opensift/images/sift_matches.jpg

image RANSAC :

http://wiki.scipy.org/Cookbook/RANSAC

Référence de travail :

http://stackoverflow.com/

http://docs.opencv.org/index.html

http://www.janeriksolem.net/

Annexes

Fichier Functions.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 15 09:59:59 2014

@author: ben
"""

from numpy import *
import numpy as np
import sift
import cv2
import cv
from pylab import *
import skimage
from PCV.geometry import homography, camera
from PCV.localdescriptors import sift

def numpyrgb2opencvrgb(numpyrgb):
    cvbgr=cv.fromarray(numpyrgb.copy())
    cvrgb = cv.CreateImage(cv.GetSize(cvbgr),8,3)
    cv.CvtColor(cvbgr, cvrgb, cv.CV_RGB2BGR)
    return cvrgb

def cube_points(c,wid):
    """ Creates a list of points for plotting
        a cube with plot. (the first 5 points are
        the bottom square, some sides repeated). """
    p = []
    # bottom
    p.append([c[0]-wid,c[1]-wid,c[2]-wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])
    p.append([c[0]-wid,c[1]-wid,c[2]-wid]) #same as first to close plot
    # top
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]-wid,c[2]+wid]) #same as first to close plot
    # vertical sides
```

```
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])
    return array(p).T

def my_calibration(sz):
    """
    Calibration function for the camera (iPhone4) used in this example.
    """
    row,col = sz
    fx = 2555*col/2592
    fy = 2586*row/1936
    K = diag([fx,fy,1])
    K[0,2] = 0.5*col
    K[1,2] = 0.5*row
    return K

def operation(img0,img1):

    mysift = cv2.SIFT(400)
    l0, d0 = mysift.detectAndCompute(img0,None)
    l0=np.asarray([ [i.pt[0],i.pt[1]] for i in l0]) #convert to array([[x0,y0],[x1,y1],...])
    l1, d1 = mysift.detectAndCompute(img1,None)
    l1=np.asarray([ [i.pt[0],i.pt[1]] for i in l1]) #convert to array([[x0,y0],[x1,y1],...])

    #match features and estimate homography
    matches = sift.match_twosided(d0,d1)
    ndx = matches.nonzero()[0]
    fp = homography.make_homog(l0[ndx,:2].T)
    ndx2 = [int(matches[i]) for i in ndx]
    tp = homography.make_homog(l1[ndx2,:2].T)

    model = homography.RansacModel()
    H, inliers = homography.H_from_ransac(fp,tp,model)
    # camera calibration
    height=img0.shape[0]
    width=img0.shape[1]
    K = my_calibration((height,width))
    # 3D points at plane z=0 with sides of length 0.2
    box = cube_points([0,0.1,0],0.1)
```

```
    # project bottom square in first image
    cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )
    # first points are the bottom square
    box_cam1 = cam1.project(homography.make_homog(box[:,:5]))


    # use H to transfer points to the second image
    box_trans = homography.normalize(dot(H,box_cam1))

    # compute second camera matrix from cam1 and H
    cam2 = camera.Camera(dot(H,cam1.P))
    A = dot(linalg.inv(K),cam2.P[:,:3])
    A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
    cam2.P[:,:3] = dot(K,A)

    # project with the second camera
    box_cam2 = cam2.project(homography.make_homog(box))

    # plotting
    im0 = array(img0)
    im1 = array(img1)

    cv_rgb=numpyrgb2opencvrgb(img1)
    cv.Rectangle(cv_rgb,
(int(box_trans[0,0]),int(box_trans[1,0])),(int(box_trans[0,2]),int(box_trans[1,2])), cv.RGB(255,0,0),
thickness=5)

    # Convert cv image to array
    im3 = np.asarray( cv_rgb[:,:] )
    return cv_rgb

def calculateHomography(image, template):
    l0,l1,d0,d1 = matchPointFromImage(image, template)

    # match features and estimate homography
    matches = sift.match_twosided(d0,d1)
    ndx = matches.nonzero()[0]

  # initialize an array with x in the first column and y in the
    # second column of the first image
    src_points = []
    for i in ndx:
        src_points.append(l0[i].pt)

    src_points = array(src_points)
```

```
ndx2 = [int(matches[i]) for i in ndx]

# initialize an array with x in the first column and y in the
# second column of the second image
target_points = []
for j in ndx2:
   target_points.append(l1[j].pt)

target_points = array(target_points)
# Mise en forme selon page 66: version optimise
f_src_points=src_points.flatten()
A=zeros((len(f_src_points),4))
#Colonne 0
A[::2,0]=f_src_points[::2] #les x: tous les pairs
A[1::2,0]=f_src_points[1::2] #les y : tous les impairs
#Colonne 1
A[::2,1]=-f_src_points[1::2] #les -y : tous les impairs
A[1::2,1]=f_src_points[::2] #les x : tous les pairs
#Colonne 2 & 3: des 1 sur lignes pairs/impaires
A[::2,2]=1.0
A[1::2,3]=1.0
B=target_points.flatten()

#Resultat
a,b,tx,ty = linalg.lstsq(A,B)[0]

########################################
#EXPLOITATION: mise en forme M, T et H + affichage
########################################
M=array([[a,-b],
        [b,a]])
print "\nMatrice rotation-homothetie:\n", M
T=array([tx,ty])
print "\nMatrice translation:\n", T
H=array([[a,-b,tx],
        [b,a,ty],
        [0,0,1]])
print "\nHomography:\n", H

return H
```

Fichier Test1.py

```
# -*- coding: utf-8 -*-

"""Created on Wed Mar 19 15:02:21 2014

PyOpenGl texture statique: affichage d'une texture stockée dans un numpy array + objet virtuel (e.g. Teapot) La texture pourra être créée manuellement (e.g. np.array([[1,0,1],[0,1,0]]) ), et/ou par chargement d'une image quelconque

"""

from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
from OpenGL.GLUT import *
import pickle
import math
from numpy import *

width,height = 800,600

def setup():
    pygame.init()
    pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
    pygame.display.set_caption('Test 1')

def set_modelview_from_camera(Rt):
    """ Set the model view matrix from camera pose. """
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    # rotate teapot 90 deg around x-axis so that z-axis is up
    Rx = array([[1,0,0],[0,0,-1],[0,1,0]])
    # set rotation to best approximation
    R = Rt[:,:3]
    U,S,V = linalg.svd(R)
    R = dot(U,V)
    R[0,:] = -R[0,:] # change sign of x-axis
    # set translation
    t = Rt[:,3]
    # setup 4*4 model view matrix
    M = eye(4)
    M[:3,:3] = dot(R,Rx)
    M[:3,3] = t
    # transpose and flatten to get column order
    M = M.T
```

```
    m = M.flatten()
    # replace model view with the new matrix
    glLoadMatrixf(m)


def set_projection_from_camera(K):
    """ Set view from a camera calibration matrix. """
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    fx = K[0,0]
    fy = K[1,1]

    fovy = 2*math.atan(0.5*height/fy)*180/math.pi
    aspect = (width*fy)/(height*fx)
# define the near and far clipping planes
    near = 0.1
    far = 100.0
# set perspective
    gluPerspective(fovy,aspect,near,far)
    glViewport(0,0,width,height)


def draw_background(imname):
    """ Draw background image using a quad. """
    # load background image (should be .bmp) to OpenGL texture
    bg_image = pygame.image.load(imname).convert()
    bg_data = pygame.image.tostring(bg_image,"RGBX",1)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    # bind the texture
    glEnable(GL_TEXTURE_2D)
    glBindTexture(GL_TEXTURE_2D,glGenTextures(1))

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED_BYTE,bg_data)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)
    # create quad to fill the whole window
    glBegin(GL_QUADS)
    glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
    glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
    glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
    glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
    glEnd()
    # clear the texture
```

```
    glDeleteTextures(1)

def draw_teapot(size):
    """ Draw a red teapot at the origin. """
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)
    glEnable(GL_DEPTH_TEST)
    glClear(GL_DEPTH_BUFFER_BIT)
    # draw red teapot
    glMaterialfv(GL_FRONT,GL_AMBIENT,[0,0,0,0])
    glMaterialfv(GL_FRONT,GL_DIFFUSE,[0.5,0.0,0.0,0.0])
    glMaterialfv(GL_FRONT,GL_SPECULAR,[0.7,0.6,0.6,0.0])
    glMaterialf(GL_FRONT,GL_SHININESS,0.25*128.0)
    glutInit()

    glPushMatrix()
    glRotate(45,0,1,0)
    glutSolidTeapot(size)
    glPopMatrix()

#main method
def main():
    #load camera
    with open('ar_cameraHD.pkl','r') as f:
        K = pickle.load(f)
        Rt = pickle.load(f)
     # initialisation de Pygame
    setup()
    draw_background('image.jpg')
    set_projection_from_camera(K)
    set_modelview_from_camera(Rt)
    draw_teapot(0.05)
    # Prise en compte des modifications et affichage
    pygame.display.flip()

main()
```

Fichier Test2.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 19 16:06:35 2014

@author: Nassim
PyOpenGl texture webcam : idem que test1.py mais l'image stockée dans le numpy array vient de la webcam.
"""

from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
from OpenGL.GLUT import *
import pickle
import math
from numpy import *
from VideoCapture import Device


width,height = 1280,720

def setup():
    pygame.init()
    pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
    pygame.display.set_caption('Test 2')

def set_modelview_from_camera(x,y,Rt):
    """ Set the model view matrix from camera pose. """
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    # rotate teapot 90 deg around x-axis so that z-axis is up
    Rx = array([[1,0,0],[0,0,-1],[0,1,0]])
    # set rotation to best approximation
    R = Rt[:,:3]
    U,S,V = linalg.svd(R)
    R = dot(U,V)
    R[0,:] = -R[0,:] # change sign of x-axis
    # set translation
    t = Rt[:,3]
    t[0] = x
    t[1] = y
    # setup 4*4 model view matrix
    M = eye(4)
```

```
    M[:3,:3] = dot(R,Rx)
    M[:3,3] = t
    # transpose and flatten to get column order
    M = M.T
    m = M.flatten()
    # replace model view with the new matrix
    glLoadMatrixf(m)


def set_projection_from_camera(K):
    """ Set view from a camera calibration matrix. """
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    # set perspective
    #gluPerspective(fovy,aspect,near,far)
    # ou
    glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    glViewport(0,0,width,height)


def draw_background(imname):
    """ Draw background image using a quad. """
    # load background image (should be .bmp) to OpenGL texture
    image_surface = pygame.image.frombuffer(imname.tostring(), imname.size, imname.mode)
    bg_image = image_surface.convert()
    bg_data = pygame.image.tostring(bg_image,"RGBX",1)

    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    # bind the texture
    glEnable(GL_TEXTURE_2D)
    glBindTexture(GL_TEXTURE_2D,glGenTextures(1))

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED_BYTE,bg_data)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)
    # create quad to fill the whole window
    glBegin(GL_QUADS)
    glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
    glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
    glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
    glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
    glEnd()
    # clear the texture
```

```
    glDeleteTextures(1)


def draw_cube(size,x,y,angle,Rt):
    xmin,xmax,ymin,ymax = -1.0,1.0,-1.0,1.0
    currentPositionX = x * (xmax-xmin) / width + xmin
    y = height - y
    currentPositionY = y * (ymax-ymin) / height + ymin

    glutInit()
    set_modelview_from_camera(currentPositionX,currentPositionY,Rt)
    glPushMatrix()
    glRotate(angle,0,0,1)
    glRotate(angle,0,1,0)
    glColor3f (1.0, 1.0, 1.0);
    # dessin du cube
    glutWireCube(size)
    glPopMatrix()

def reshape(w, h):
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity()
   glViewport(0,0,w,h);

#main method
def main():
    #load webcam
    cam = Device()
    # résolution de la caméra
    cam.setResolution(1280,720)
    angle = 1
    #load camera
    with open('ar_cameraHD.pkl','r') as f:
        K = pickle.load(f)
        Rt = pickle.load(f)
    setup()

    while True:
        # on récupère une image de la webcam
        image = cam.getImage()
        draw_background(image)
        set_projection_from_camera(K)
        # position x et y de la souris
        x,y = pygame.mouse.get_pos()
        # affichage du cube
```

```
        draw_cube(0.1,x,y,angle,Rt)
        # modification de l'angle
        angle = (angle + 1)%90
        reshape(width,height)
        event = pygame.event.poll()
        if event.type in (QUIT,KEYDOWN):
            break

        pygame.display.flip()

main()
```

Fichier Test3.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 15 10:52:10 2014
@author: ben
"""

from pylab import *
from numpy import *
from PIL import Image
import cv2
from matplotlib import pyplot as PLT
import StringIO
import PIL

from PCV.geometry import homography, camera
from PCV.localdescriptors import sift

"""
This is the augmented reality and pose estimation cube example from Section 4.3.
"""

mysift = cv2.SIFT(400)
img0 = cv2.imread('images/imageTest1.1.jpg',cv2.COLOR_BGR2GRAY)
l0, d0 = mysift.detectAndCompute(img0,None)
im = cv2.drawKeypoints(img0,l0,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
cv2.imwrite("ble.jpg",im)
kp = l0
l0=np.asarray([ [i.pt[0],i.pt[1]] for i in l0]) #convert to array([[x0,y0],[x1,y1],...])
img1 = cv2.imread('images/imageTest1.2.jpg',cv2.COLOR_BGR2GRAY)
l1, d1 = mysift.detectAndCompute(img1,None)
temp = cv2.drawKeypoints(img1,l1,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
kp2 = l1
l1=np.asarray([ [i.pt[0],i.pt[1]] for i in l1]) #convert to array([[x0,y0],[x1,y1],...])


#match features and estimate homography
matches = sift.match_twosided(d0,d1)
ndx = matches.nonzero()[0]
fp = homography.make_homog(l0[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
tp = homography.make_homog(l1[ndx2,:2].T)

model = homography.RansacModel()
H, inliers = homography.H_from_ransac(fp,tp,model)
```

```
M=H[:2,:2]
T=H[:2,2]
#print Homography
print "\nMatrice rotation-homothetie:\n",M
print "\nMatrice translation:\n", T
print "\nMatrice homography",H

# Draw Keypoints:
h1, w1 = im.shape[:2]
h2, w2 = temp.shape[:2]
nWidth = w1+w2
nHeight = max(h1, h2)
hdif = (h1-h2)/2
newimg = zeros((nHeight, nWidth, 3), uint8)
newimg[hdif:hdif+h2, :w2] = temp
newimg[:h1, w2:w1+w2] = im


# draw lines joining matched points.
for i in range(min(len(kp2), len(kp))):
    pt_a = (int(kp2[i].pt[0]), int(kp2[i].pt[1]+hdif))
    pt_b = (int(kp[i].pt[0]+w2), int(kp[i].pt[1]))
    cv2.line(newimg, pt_a, pt_b, (255, 0, 0))


cv2.imshow("image", newimg)
cv2.waitKey(0)
```

Fichier Test4.py

```
# -*- coding: utf-8 -*-
""" Created on Tue Apr 15 10:52:10 2014
@author: ben
"""
from pylab import *
from numpy import *
from PIL import Image
import cv2
from matplotlib import pyplot as PLT
import StringIO
import PIL
import functions

from PCV.geometry import homography, camera
from PCV.localdescriptors import sift

""" This is the augmented reality and pose estimation cube example from Section 4.3.
"""
mysift = cv2.SIFT(400)
img0 = cv2.imread('images/imageTest1.1.jpg',cv2.COLOR_BGR2GRAY)
l0, d0 = mysift.detectAndCompute(img0,None)
l0=np.asarray([ [i.pt[0],i.pt[1]] for i in l0]) #convert to array([[x0,y0],[x1,y1],...])

img1 = cv2.imread('images/imageTest1.2.jpg',cv2.COLOR_BGR2GRAY)
l1, d1 = mysift.detectAndCompute(img1,None)
l1=np.asarray([ [i.pt[0],i.pt[1]] for i in l1]) #convert to array([[x0,y0],[x1,y1],...])

#match features and estimate homography
matches = sift.match_twosided(d0,d1)
ndx = matches.nonzero()[0]
fp = homography.make_homog(l0[ndx,:2].T)
ndx2 = [int(matches[i]) for i in ndx]
tp = homography.make_homog(l1[ndx2,:2].T)

model = homography.RansacModel()
H, inliers = homography.H_from_ransac(fp,tp,model)
# camera calibration
K = functions.my_calibration((747,1000))

# 3D points at plane z=0 with sides of length 0.2
box = functions.cube_points([0,0,0.1],0.1)
# project bottom square in first image
cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )
```

```
# first points are the bottom square
box_cam1 = cam1.project(homography.make_homog(box[:,:5]))

# use H to transfer points to the second image
box_trans = homography.normalize(dot(H,box_cam1))
# compute second camera matrix from cam1 and H
cam2 = camera.Camera(dot(H,cam1.P))
A = dot(linalg.inv(K),cam2.P[:,:3])
A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
cam2.P[:,:3] = dot(K,A)

# project with the second camera
box_cam2 = cam2.project(homography.make_homog(box))

# plotting
im0 = array(img0)
im1 = array(img1)
print H

figure()
imshow(im0)
plot(box_cam1[0,:],box_cam1[1,:],linewidth=3)
title('2D projection of bottom square')
axis('off')
figure()
imshow(im1)
plot(box_trans[0,:],box_trans[1,:],linewidth=3)
title('2D projection transfered with H')
axis('off')

figure()
imshow(im1)
plot(box_cam2[0,:],box_cam2[1,:],linewidth=3)
title('3D points projected in second image')
axis('off')

show()
```

Fichier Test5.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 26 14:06:37 2014
@author: Nassim
"""
from OpenGL.GL import *
from OpenGL.GLU import *
import pygame, pygame.image
from pygame.locals import *
from OpenGL.GLUT import *
import pickle
import math
from numpy import *
import numpy as np

from PIL import Image
import cv2

from PCV.geometry import homography, camera
from PCV.localdescriptors import sift

# taille des images
width,height = 600,600
currentPositionX = 0
currentPositionY = 0

#choix des deux images
image1 = "image1.png"
image2 = "image7.png"

def setup():
    pygame.init()
    pygame.display.set_mode((width,height),OPENGL | DOUBLEBUF)
    pygame.display.set_caption('Test 5')


def my_calibration(sz):
    row,col = sz
    fx = 2555*col/2592
    fy = 2586*row/1936
    K = diag([fx,fy,1])
    K[0,2] = 0.5*col
    K[1,2] = 0.5*row
    return K
```

```
def set_modelview_from_camera(Rt,x,y,rot):
    """ Set the model view matrix from camera pose. """
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()

    # set rotation to best approximation
    R = Rt[:,:3]
    U,S,V = linalg.svd(R)
    R = dot(U,V)
    R[0,:] = -R[0,:] # change sign of x-axis
    # set translation
    t = Rt[:,3]
    t[0] = x
    t[1] = y
    # setup 4*4 model view matrix
    M = eye(4)
    M[:3,:3] = rot
    M[:3,3]= t
    #print M
    # transpose and flatten to get column order
    M = M.T
    m = M.flatten()
    # replace model view with the new matrix
    glLoadMatrixf(m)

def set_projection_from_camera():
    """ Set view from a camera calibration matrix. """
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(-1.0,1.0,-1.0,1.0,-1.0,1.0);
    glViewport(0,0,width,height)

def cube_points(c,wid):
    """ Creates a list of points for plotting
        a cube with plot. (the first 5 points are
        the bottom square, some sides repeated). """
    p = []
    # bottom
    p.append([c[0]-wid,c[1]-wid,c[2]-wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])
    p.append([c[0]-wid,c[1]-wid,c[2]-wid]) #same as first to close plot
    # top
```

```
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]-wid,c[2]+wid]) #same as first to close plot
    # vertical sides
    p.append([c[0]-wid,c[1]-wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]+wid])
    p.append([c[0]-wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]-wid])
    p.append([c[0]+wid,c[1]+wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]+wid])
    p.append([c[0]+wid,c[1]-wid,c[2]-wid])

    return array(p).T

def draw_background(im):
    """ Draw background image using a quad. """
    image_surface = pygame.image.frombuffer(im.tostring(), im.size, im.mode)
    bg_image = image_surface.convert()
    bg_data = pygame.image.tostring(bg_image,"RGBA",1)
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    # bind the texture
    glEnable(GL_TEXTURE_2D)
    glBindTexture(GL_TEXTURE_2D,glGenTextures(1))

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,width,height,0,GL_RGBA,GL_UNSIGNED_BYTE,bg_data)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST)
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST)
    # create quad to fill the whole window
    glBegin(GL_QUADS)
    glTexCoord2f(0.0,0.0); glVertex3f(-1.0,-1.0,-1.0)
    glTexCoord2f(1.0,0.0); glVertex3f( 1.0,-1.0,-1.0)
    glTexCoord2f(1.0,1.0); glVertex3f( 1.0, 1.0,-1.0)
    glTexCoord2f(0.0,1.0); glVertex3f(-1.0, 1.0,-1.0)
    glEnd()
    # clear the texture
    glDeleteTextures(1)

def draw_teapot(size,x,y,rot,Rt):
    """ Draw a red teapot at the origin. """
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)
```

```
    glEnable(GL_DEPTH_TEST)
    glClear(GL_DEPTH_BUFFER_BIT)
    # draw red teapot
    glutInit()
    xmin,xmax,ymin,ymax = -1.0,1.0,-1.0,1.0
    currentPositionX = x * (xmax-xmin) / width + xmin
    y = height - y
    currentPositionY = y * (ymax-ymin) / height + ymin
    set_modelview_from_camera(Rt,currentPositionX,currentPositionY,rot)
    glPushMatrix()
    glutSolidTeapot(size)
    glPopMatrix()

def algorithme_sift():
    #calcul
    mysift = cv2.SIFT(400)
    img0 = cv2.imread(image1,cv2.COLOR_BGR2GRAY)
    l0, d0 = mysift.detectAndCompute(img0,None)
    l0=np.asarray([ [i.pt[0],i.pt[1]] for i in l0])
    img1 = cv2.imread(image2,cv2.COLOR_BGR2GRAY)
    l1, d1 = mysift.detectAndCompute(img1,None)
    l1=np.asarray([ [i.pt[0],i.pt[1]] for i in l1])


    # match features and estimate homography
    matches = sift.match_twosided(d0,d1)
    ndx = matches.nonzero()[0]
    fp = homography.make_homog(l0[ndx,:2].T)
    ndx2 = [int(matches[i]) for i in ndx]
    tp = homography.make_homog(l1[ndx2,:2].T)

    model = homography.RansacModel()
    H, inliers = homography.H_from_ransac(fp,tp,model)
    K = my_calibration((height,width))

    # 3D points at plane z=0 with sides of length 0.2
    box = cube_points([0,0,0.1],0.1)

    # project bottom square in first image
    cam1 = camera.Camera( hstack((K,dot(K,array([[0],[0],[-1]])) )) )
    # first points are the bottom square
    box_cam1 = cam1.project(homography.make_homog(box[:,:5]))


    # use H to transfer points to the second image
```

```
    box_trans = homography.normalize(dot(H,box_cam1))

    # compute second camera matrix from cam1 and H
    cam2 = camera.Camera(dot(H,cam1.P))
    A = dot(linalg.inv(K),cam2.P[:,:3])
    A = array([A[:,0],A[:,1],cross(A[:,0],A[:,1])]).T
    cam2.P[:,:3] = dot(K,A)

    # project with the second camera
    box_cam2 = cam2.project(homography.make_homog(box))
    #print H

    return box_cam1,box_cam2,H

def reshape(w, h):
  glViewport(0,0,w,h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity()

def main():
    with open('ar_cameraHD.pkl','r') as f:
        K = pickle.load(f)
        Rt = pickle.load(f)

    # initialisation de pygame
    setup()
    im1 = Image.open(image1)
    im2 = Image.open(image2)
    # application de l'algorithme sift et récupération de la matrice H
    box_cam1,box_cam2,H = algorithme_sift()
    print H

    # on dessine la première image de fond
    draw_background(im1)
    # projection from camera
    set_projection_from_camera()
    x = (box_cam1[0,0]+box_cam1[0,2])/2
    y = (box_cam1[1,0]+box_cam1[1,2])/2

    # on dessine la teapot au milieu de l'écran
    rot_init = array([[1,0,0],[0,1,0],[0,0,1]])
    draw_teapot(0.1,x,y,rot_init,Rt)
    pygame.display.flip()
    # on attend 5 secondes
    pygame.time.wait(5000)
```

```
    reshape(width,height)
    # on dessine la deuxième image de fond
    draw_background(im2)
     # projection from camera
    set_projection_from_camera()
     #calculer le centre
    x = (box_cam2[0,0]+box_cam2[0,2])/2
    y = (box_cam2[1,0]+box_cam2[1,2])/2
    # on créé la matrice de rotation grâce à H
    rot = array([[H[0,0],H[0,1],0],[H[1,0],H[1,1],0],[H[2,0],H[2,1],H[2,2]]])
    draw_teapot(0.1,x,y,rot,Rt)
    pygame.display.flip()

    while True:
        event = pygame.event.poll()
        if event.type in (QUIT,KEYDOWN):
            break



main()
```

Fichier Test6.py

```
# -*- coding: utf-8 -*-
""" Created on Wed Mar 19 14:54:52 2014
@author: ben
"""
import cv
import numpy as np
from pylab import *
import skimage
from skimage import io
import functions as op

filepath = "/home/ben/Documents/ProjetPython/TestingFiles/images/video2.MP4"

#Création du lecteur
video_reader=skimage.io.Video(filepath)
previous=video_reader.get()
previous=skimage.color.rgb2gray(previous)

#Création du writer
writer = cv.CreateVideoWriter("testVideoOut.avi",cv.CV_FOURCC('X', 'V', 'I',
'D'),20,(previous.shape[1],previous.shape[0]),is_color=1)

for i in range(0,np.uint(video_reader.frame_count()-1)):
   #Numpy image
   current_rgb=video_reader.get()

   if(i==0):
      previous=current_rgb

   #Image opencv
   cv_rgb=op.numpyrgb2opencvrgb(current_rgb)

   #Affichage
   cv.ShowImage("Video 1", cv_rgb) #copy to avoid non contiguous data
   im = op.operation(np.array(previous),np.array(current_rgb))

    #Ecriture
   cv.WriteFrame(writer,im)
   #Attente 16 ms
   key=cv.WaitKey(16)

#writer.close()
print "finished"
```

Résumé

Aujourd'hui la réalité augmentée est de plus en plus présente dans notre vie quotidienne. La réalité augmentée enrichit notre perception du monde réel en y ajoutant des informations virtuelles. Le projet est inspiré du livre "Computer Vision", écrit par JE Solem. Ce livre fournit des méthodes et des algorithmes particulièrement utiles pour la reconnaissance de points de l'image traitée.

Ce projet de quatrième année permet d'aborder ce domaine tant prometteur. Il s'agit de développer une application de réalité augmentée sur PC. Le but de cette application est d'interagir avec un objet placé dans notre espace virtuel au moyen d'un support. Notre espace virtuel est visible sur l'écran du PC et le support est un objet réel, il sert de référence pour la webcam et ses mouvements permettent d'agir sur l'objet virtuel.

L'utilisation du langage de programmation Python était nécessaire pour mener à bien le projet, c'est en effet un langage assez souple et qui donne de nombreuses fonctionnalités pour fournir les résultats attendus.

Grâce à ce projet, il est possible d'ajouter des informations sur l'environnement de l'utilisateur ce qui permet de donner un visuel interactif. Ce projet peut être amélioré de plusieurs façons en fonction des besoins.

Abstract

Today augmented reality is increasingly present in our daily lives. Augmented reality enriches our perception of reality by adding virtual information. The project is inspired by the book "Computer Vision" written by JE Solem. This book provides methods and algorithms particularly useful for the recognition of points of the processed image.

This fourth year's project permits to discover this growing field. It involves developing an augmented reality application on PC. The purpose of this application is to interact with an object placed in our virtual space through a support. Our virtual space is visible on the PC screen and the support is a real object, it is used as a reference for the webcam and his movements allow interacting with the virtual object.

Using the Python's programming language was necessary to carry out the project that gives flexibility and many features to provide expected results.

Thanks to this project, it is possible to add information in a user's environment which permits to give an interactive visual. This project can be improved in many ways depending on the needs.