Université Mouloud MAMMERI Tizi Ouzou
Faculté de génie électrique et d'informatique
Département d'Informatique

Tracking 3D de marqueurs en temps réel

Application à la réalité augmentée et à l'asservissement d'une tourelle.

Par
Mohand TINEDGHAR

Tizi Ouzou
Année académique
2013-2014

Encadreur
Mohammed Ourabah SOUALAH

Présentation du mémoire en vue d'obtenir le diplôme de Master en informatique

INTRODUCTION..... 1

NOTIONS DE BASES SUR LES SYSTÈMES D'ACQUISITION 3

PREAMBULE..... 4

I-1 LES LENTILLES.....4

I-2 LA DISTANCE FOCALE 5

I-3 CHAMPS ET ANGLES DE VUE (CDV ET ADV) 6

I-4 DISTORSION OPTIQUE.....7

I-4-1 MODELISATION DE LA DISTORSION RADIALE..... 9

I-4-2 CORRECTION DE LA DISTORSION OPTIQUE..... 10

DISCUSSION 11

TRACKING 3D 13

PREAMBULE..... 14

II-1 TRACKING 3D..... 16

II-1-1 LE MATCH MOVING..... 16

II-1-2 DÉTECTION DE POINTS D'INTÉRÊT 18

II-1-2 CALIBRATION..... 35

II-2 THÉORIE DE NOTRE APPROCHE 39

II-2-1 METHODE D'OTSU POUR LE SEUILLAGE 39

II-2-2 DÉTECTION DE CONTOURS..... 40

II-2-3 CALIBRATION DE LA CAMERA 41

II-2-4 RECONSTRUCTION 3D..... 43

DISCUSSION 45

DEVELOPPEMENT DE L'APPLICATION 46

III-1 RÉALITÉ AUGMENTÉE BASÉE SUR LES MARQUEURS 47

III-2 LES OUTILS NÉCESSAIRES 48

III-2-1 Softwares 48
III-2-2 HARDWARE 49
III-3 ARCHITECTURE DE L'APPLICATION 50
III-4 DESCRIPTION DÉTAILLÉE DU PROGRAMME 52
III-4-1 capture vidÉo 52
III-4-2 DÉtection du marqueur 53
III-4-3 Identification d'un marqueur 54
III-4-4 conversion en niveau de gris 55
III-4-5 Binarisation de l'image 55
III-4-6 DÉtection des contours 57
III-4-7 Recherche du candidat 58
III-4-8 RECONNAISANCE DU CODE 61
III-4-9 lecture du code du marqueur 61
III-4-10 rafFinement de la position du marqueur 65
III-4-11 Estimation de la position du marqueur 66
III-5 CALIBRATION DE LA CAMÉRA 70
III-6 RENDU DE L'OBJET VIRTUEL 3D 72
III-6-1 Rendu de la scÈne en realitÉ augmentÉe 73
III-6-2 Mise en place de l'arriÈre plan 74
III-6-3 mise en place de la perspective 75
III-6-4 Incrustation de l'objet 3D 76
III-7 ASSERVISSEMENT DU ROBOT 77
III-7-1 Dans le microContrÔleur 78
III-7-2 Dans le programme principal 79
III-8 FONCTIONNALITÉS SUPPLÉMENTAIRES 80
II-8-1 l'interface 81
III-8-2 Multi marqueurs 83
III-8-3 Chargement automatique des marqueurs connus 84
III-8-4 importation d'objets 3D 86
III-8-5 Aller plus loin 90
III-9 FUTURE DE LA RÉALITÉ AUGMENTÉE 91
III-9-1 Viseur tÊte haute 92
III-9-2 Dans la medecine 94
III-9-3 Dans l'Éducation 95
III-10 RÉSULTATS 96

DISCUSSION .....
98

BIBLIOGRAPHIE .....
102

TABLE DES ILLUSTRATIONS.....
104

REMERCIEME

INTRODUCTION

Avec la montée en puissance de la technologie de nos jours, le domaine de vision par ordinateur se voit de plus en plus développé avec l'évolution des systèmes d'acquisitions et les unités de calculs. Les solutions développées en traitement d'image n'arrêtent pas de voir le jour, dans les appareils les plus sophistiqués mais aussi dans les plus simples, qu'on utilise tous les jours. Ainsi le fait de pouvoir interagir avec les objets qui nous entourent via une interface virtuelle est devenu une branche de recherche alléchante pour les scientifiques, et c'est là que le tracking 3d nous permet justement d'obtenir certaines informations cruciales qui sont d'ordre spatiale, et ce afin de déterminer, par exemple, la position d'un objet (proche ou loin), mais aussi de déterminer sa forme ou même de le reconstituer virtuellement.

Le tracking passe par une étape cruciale qu'est l'extraction de caractéristiques visuelles (ou visual features extraction en anglais) et qui consiste en des transformations mathématiques calculées sur les pixels d'une image numérique, le but de l'extraction étant de récupérer un maximum d'information d'une image en utilisant un minimum de donnée. Cette technologie fait objet aussi d'une grande recherche pour améliorer les algorithmes et les rendre plus performants.

Les caractéristiques visuelles permettent généralement de mieux rendre compte de certaines propriétés visuelles de l'image, utilisées pour des traitement ultérieurs entrant dans le cadre d'applications telles que la détection d'objets présents dans l'image, ou comme dans notre cas la réalisation d'application de réalité augmentée, mais aussi, ils servent aussi à la détection des visages ou des personnes. Ces descripteurs sont aussi utilisés dans la recherche d'image par le contenu, d'une manière générale, ce sont des technologies de base qui servent dans de nombreux domaines ou la vision par ordinateur intervient, soit : robotique, vidéosurveillance, vision industrielle, etc.

Dans notre projet, nous allons construire une application de réalité augmentée basée sur des marqueurs, nous allons incruster des objets virtuels dans le monde réel capturé par une caméra, et ce en utilisant un procédé astucieux qui nous permettra de rendre l'application en temps réel. On se penchera sur la détection des contours et la segmentation de l'image, qui sont des opérations plus légères néanmoins très efficace.

En effet, nous avons considéré ce procédé comme le plus adapté à nos besoins car il propose des résultats en un temps record. Le domaine de tracking est particulièrement vaste du fait qu'il existe un grand nombre d'approches possibles pour résoudre le même problème, il existe

plusieurs algorithme d'extraction de caractéristiques, et de suivi d'objet, mais à chacun ses points forts et points faibles, notamment dans la stabilité et les temps de calculs. Certains algorithmes tirent des informations très stables mais en dépit du temps de calcul car cela nécessite des calculs mathématiques complexes et souvent très long.

La vision artificielle a l'avantage de produire des solutions non encombrantes, précises et moins couteuses. Néanmoins, celle-ci exigent un grand investissement en effort dans la création d'algorithmes robustes.

Dans le premier chapitre, nous parlerons, dans un premier temps, du fonctionnement de base des systèmes d'acquisitions, ce qui nous mènera à la compréhension des différents concepts liés au monde des appareils photos et caméras. Dans un second temps, nous présenterons une solution pour la correction de la distorsion optique causée par le dispositif optique sur les images.

Dans le second chapitre, nous exposerons, en premier lieu, ce qu'est le match moving, car ce dernier a eu un impact sans précédent sur le monde du 7ème art, on a jugé bon de le présenter. Dans un second lieu, on parlera de certaines méthodes d'extraction de caractéristiques connues puis après cela de calibration, étape primordiale pour pouvoir estimer la position d'un objet dans l'espace. Enfin dans le même chapitre nous donnerons les définitions nécessaires des différents algorithmes que nous avons utilisé dans notre approche pour la détection et le suivi du marqueur.

Le troisième et dernier chapitre sera consacré au développement de notre application, mettant en œuvre l'approche des contours et de l'approximation polygonale pour la détection de marqueurs, puis de l'estimation de position en utilisant les fonction de calibration, enfin la librairie d'openGL pour la création d'objets virtuels

CHAPITRE I

NOTIONS DE BASES SUR LES SYSTÈMES D'ACQUISITION

PREAMBULE

Les cameras sont conçues pour enregistrer des séquences d'images. Au cœur de ce processus se trouvent des lentilles servant à focaliser la lumière sur un support d'acquisition (film pour analogique et CCDs pour numérique).

Les séquences d'images étant capturées dans un support d'enregistrement, celui-ci reconstituera la scène lorsqu'il sera déroulé à la même vitesse d'enregistrement. La majorité des caméras analogiques enregistrent à une vitesse de 24 images par secondes. Ces vitesses varient selon le standard considéré (30 pour le format NTSC et 25 pour le format PAL). Le choix du standard est lié aux différentes utilisations de la séquence d'images. Dans ce chapitre nous présenterons les éléments essentiels de ce type de capteurs.

I-1 LES LENTILLES

Les lentilles utilisées dans la réalisation de caméras sont conçues avec différents degrés de précisions. 'L'objectif' de la caméra comprend une multitude de lentilles appelées éléments (figure I-1). On distingue les lentilles principales qui ont pour but de focaliser la lumière avec précision sur le plan du film et les lentilles auxiliaires qui permettent de corriger les imperfections telles que la distorsion et les aberrations chromatiques. Dans la plupart des cas, ces éléments sont enduits avec une fine couche d'un film qui réfracte ou filtre la lumière afin que la qualité de l'image soit meilleure. Outre ces éléments, l'objectif contient aussi un iris ou un diaphragme qui contrôle la quantité de lumière projetée sur le film (1).

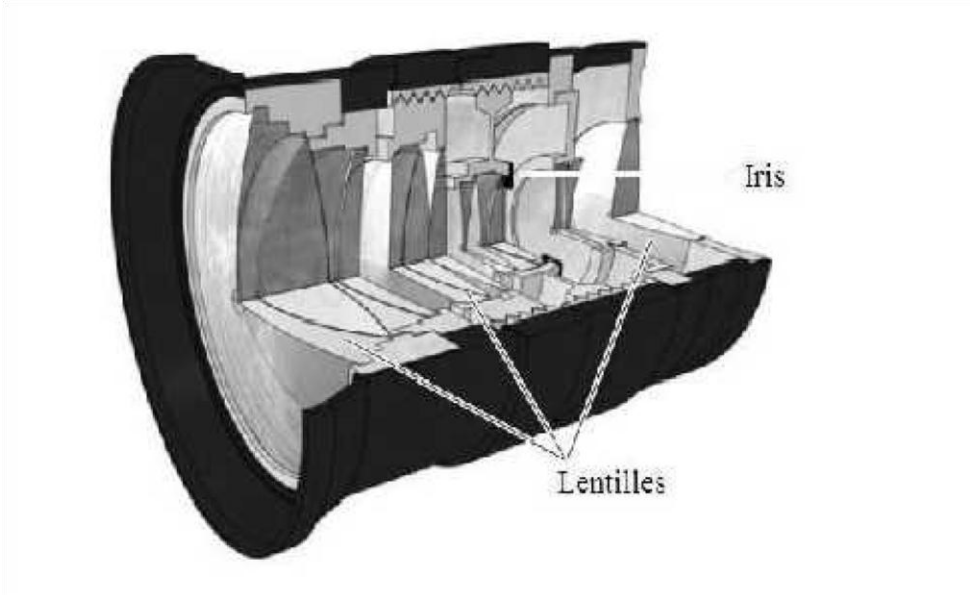


Figure I-1- Les composantes d'un objectif (1).

I-2 LA DISTANCE FOCALE

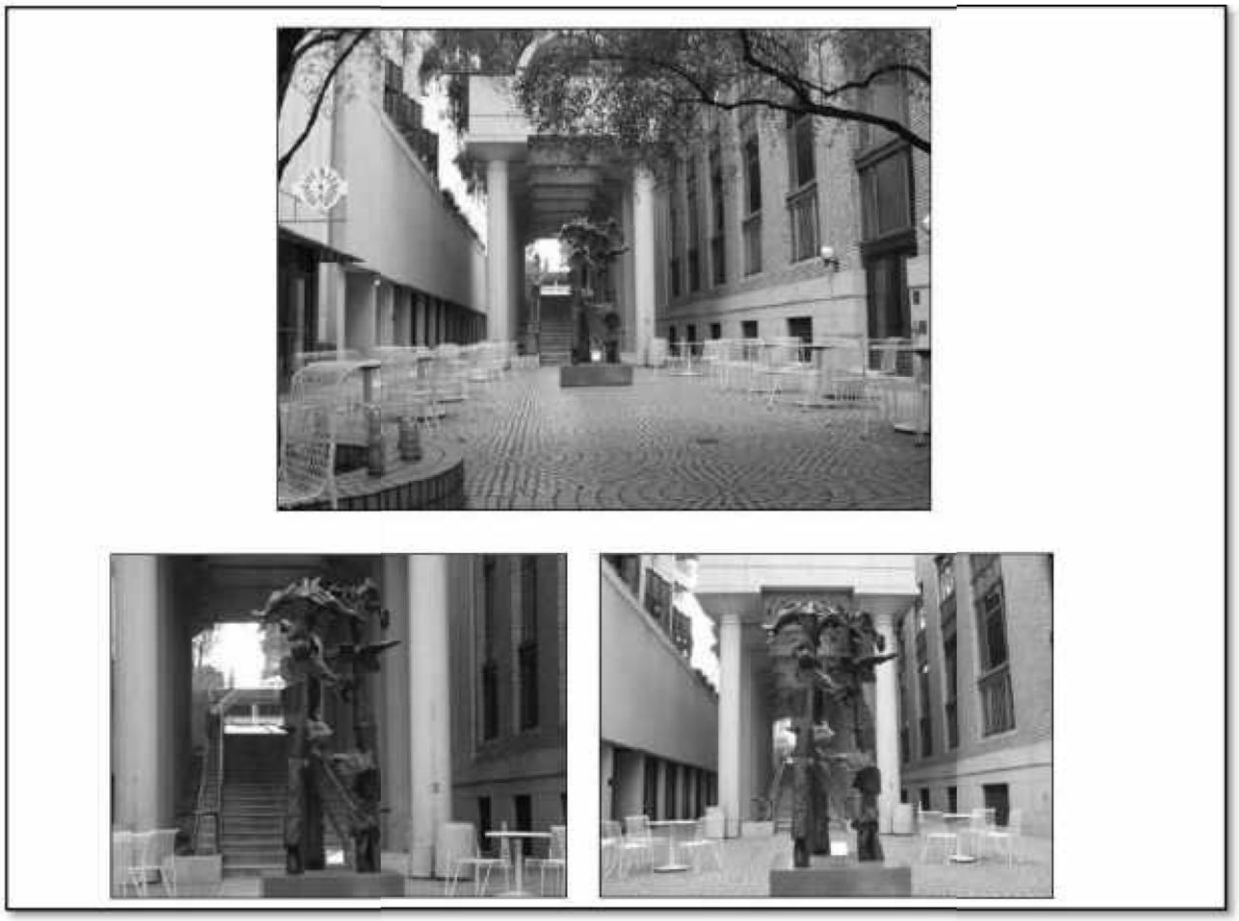
La caractéristique de la lentille qui a le plus d'incidence sur l'image finale est sans doute la distance focale. Elle est définie comme étant la distance entre le centre optique (centre de la lentille principale) et le plan de projection. Une courte distance focale produit un effet de perspective plus prononcé, un large sens de profondeur et un nombre élevé d'objets visibles de la scène (voir figure I-2-a).

Une distance focale plus longue produit des images montrant un aspect plus plat de la scène avec un sens moins évident de perspective et aussi moins d'objets visibles de la scène (voir figure I-2-b) (2)

(a) (b)

Figure I-2- différence entre courte et longue distance focale(2).

Certaines lentilles sont appelées lentilles fixes, car leur distance focale reste inchangée. On distingue aussi les lentilles variables (ou lentilles de zoom), appelées ainsi en raison du caractère variable de leur distance focale. Le zoom d'une lentille variable est possible grâce à un ensemble d'éléments sis à l'intérieur de l'objectif qui se meuvent le long du cylindre, ceci a pour conséquence l'augmentation ou la diminution de la distance focale. Lorsque la lentille est en zoom (on part d'une distance focale donnée à une autre plus longue), les objets de la scène paraissent de plus en plus proches et remplissent davantage le cadre de l'image. Cela s'apparente à un déplacement d'une camera avec une lentille à distance focale fixe le long de l'axe principal en direction du sujet, mais il subsiste une différence subtile entre l'action de zoomer et une translation en profondeur. La figure 1.3 illustre bien cette différence, en effet on constate clairement que les relations des arrières et avant plans des éléments changent radicalement entre les deux.



Zoom **Translation en profondeur**

Figure I-3- Différence entre un zoom et un déplacement en profondeur.

Lorsqu'on analyse une image pour déterminer si la camera est en zoom ou en translation en profondeur, il suffit de vérifier la relation entre l'avant et l'arrière plan. S'ils ne se déplacent pas indépendamment, alors il y a de fortes chances que ce soit un zoom.

I-3 CHAMPS ET ANGLES D E VUE (CDV ET ADV)

Comprendre comment le plan de projection et la distance focale sont reliés requiert un peu d'exercice mental. Si on étend des lignes imaginaires à partir des quatre coins du film vers les bords de la lentille, on obtient une forme pyramidale qui s'élargit de plus en plus (connue sous le nom technique de Frustum) définissant un champ de vue pour une configuration particulière des lentilles et du plan de projection (Figure I-4).

Le CDV est la mesure de ce que l'on peut actuellement voir dans la scène. Si on change l'une ou l'autre composantes en question (plan de projection et/ou distance focale), l'angle de vue (ADV) sera automatiquement affecté.

Il est aussi important de savoir que le CDV et l'ADV sont souvent utilisés de façon interchangeable. Cependant, les deux notions sont différentes (Figure I-5)**.** ADV mesure la portion du cercle visible par la caméra.

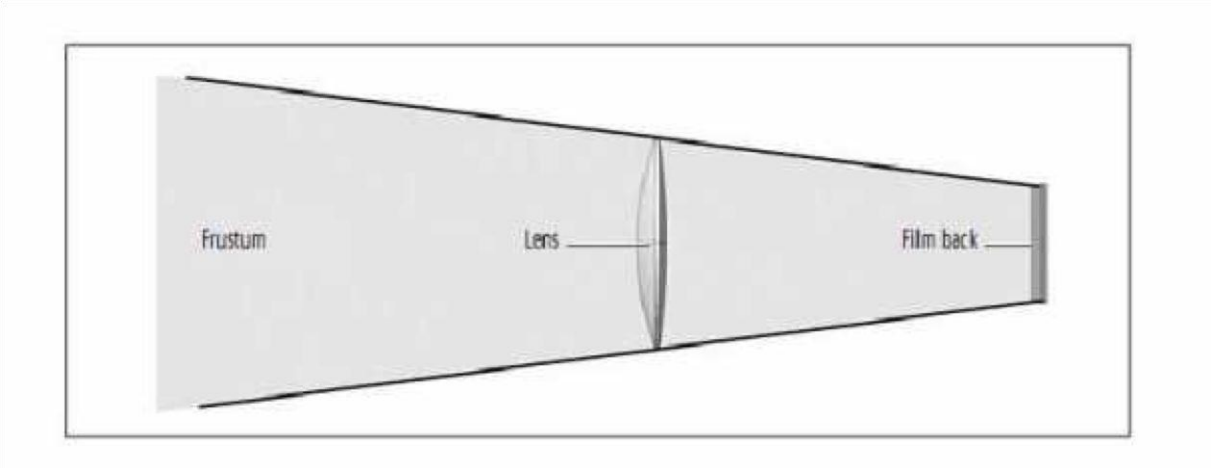


Figure I-4- Frustum.

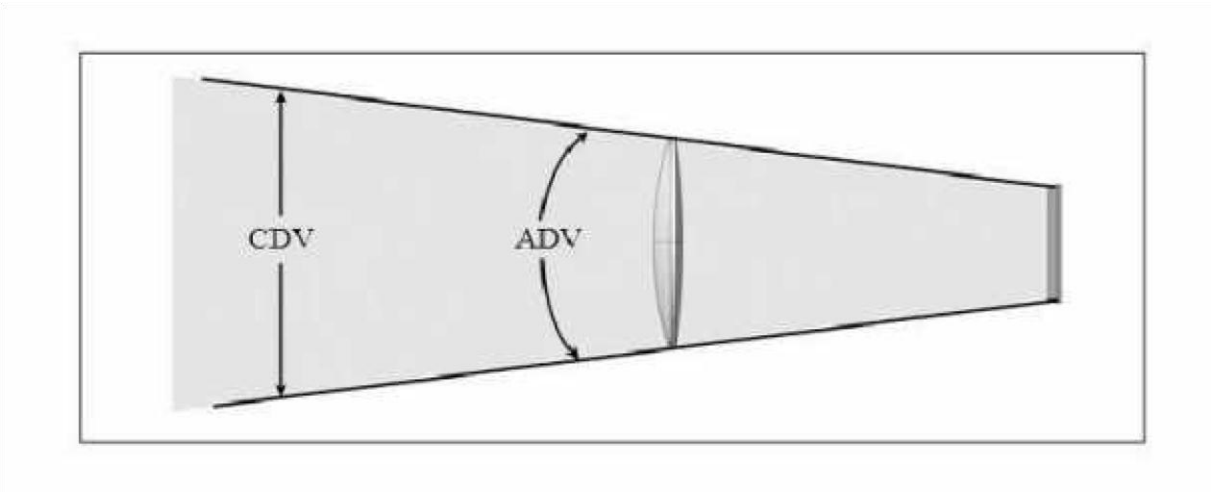


Figure I-5- Angle et champ de vue.

ADV est usuellement mesuré en degrés, par exemple, si la lentille a un ADV de 90°, elle peut couvrir approximativement un tiers de la scène de gauche a droite. Il y a une relation inverse entre la distance focale de la lentille et son ADV. Plus la distance focale est grande, moins est l'ADV et vice versa. Et c'est pourquoi les petits objectifs sont généralement connus pour leur large ADV vu la petitesse de leur distance focale. Par exemple, l'utilisation d'un objectif long signifie que le centre optique est loin du plan de projection, ce qui produit un frustum étroit et un ADV plus petit. Utiliser un plan de projection plus grand en laissant la distance focale comme telle aura le même effet (1).

I-4 DISTORSION OPTIQUE

La distorsion est une altération de la prise de vue d'un dispositif d'acquisition d'images tel qu'un appareil photo ou une camera. La distorsion est due à la position du diaphragme dans le système optique : plus le diaphragme est loin du centre optique, plus la distorsion est forte. Il existe deux formes de distorsions : radiale et tangentielle. La première a tendance à arrondir les bords de l'image. Cette distorsion est souvent plus flagrante sur

des appareils à courte focale (grands angles).La seconde applique une sorte de rotation autour du centre de l'image. L'amplitude de cette rotation est variable suivant la position du point traité sur l'image. Cette distorsion est toutefois négligeable et nous ne la traiterons pas dans ce sujet (3).

Le diaphragme (tel un iris dans une lentille) limite l'accès de la lumière à l'intérieur de l'objectif. Si ce diaphragme est assez proche de la lentille principale, alors l'image capturée à travers cette lentille est relativement intacte (sans distorsion). Néanmoins, si le diaphragme est à une distance significative ou derrière la lentille, les choses seront différentes. Considérant la lumière qui traverse la lentille, par définition le rayon qui passe par le centre du diaphragme doit toujours parvenir au plan de l'image au centre du plan de projection (Figure I-6) . Mais que dire de la lumière qui passe a travers les bords du diaphragme? Si le diaphragme est devant la lentille alors ces rayons de lumière peuvent atteindre la lentille avec un angle oblique causant des courbures vers le centre de l'image et dont il résulte distorsion en barillet. Vice versa si le diaphragme est derrière la lentille, dans ce cas les seuls rayons autorisés à passer a travers le diaphragme sont ceux autour des bords, ces rayons tendent à se courber vers l'extérieur et sont déformés en croissant.

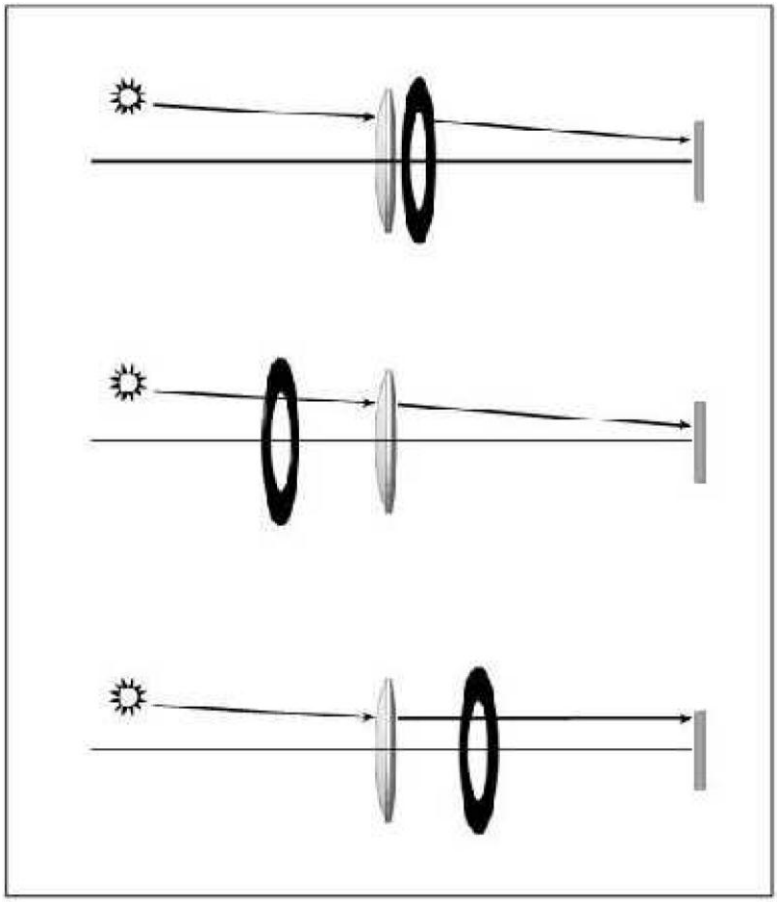


Figure I-6- Les différentes positions du diaphragme

La distorsion est plus visible avec des lentilles à grands angles et/ou de zoom, et peut être très difficile et compliquée de la corriger dans de larges traitements séquentiels. On doit souvent corriger les distorsions des images trop flagrantes avant leurs utilisations dans le tracking .

Heureusement, la distorsion optique peut être modélisée par une transformation 2-D de l'image, donc pouvant être corrigée de manière algorithmique. Elle se présente en deux formes, radiale et tangentielle. Toutefois, cette dernière est négligeable vu qu'elle n'a pas d'effet néfaste sur l'image originale.

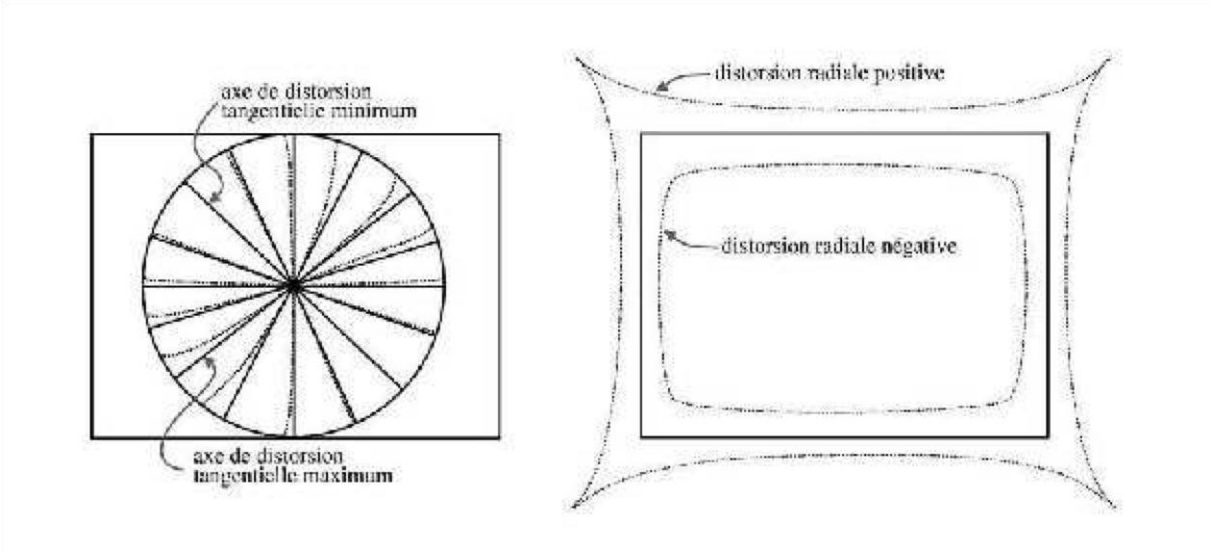


Figure I-7- Distorsion tangentielle et radiale.

I-4-1 MODELISATION DE LA DISTORSION RADIALE

Comme son nom l'indique, la distorsion radiale affecte différemment les points selon leur distance par rapport au centre de distorsion. La distorsion radiale révèle un caractère non linéaire ce qui complique légèrement sa modélisation. Elle peut-être modélisée par un polynôme vérifiant certaines contraintes (4).

Soit (x, y) le centre de distorsion, il s'agit en général du point principal qui correspondra à l'intersection de l'axe optique avec limage. Il arrive que le centre de distorsion soit décentré par rapport à l'image dans le cas d'ass s optiques de mauvaise qualité.

Ce centre de distorsion peut toutefois être assimilé au centre de l'image sans trop d'effets néfastes sur le modèle. Soit (x_i, y_i) semble de points sur l'image distordue et ( ) les points correspondants sur l'image rectifiée (Figure I-8).

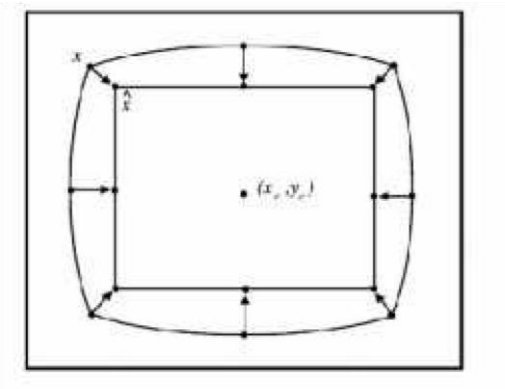


Figure I-8- Modélisation de la distorsion radiale.

La relation entre ces deux ensembles de points peut s'écrire de la façon suivante :

$$\begin{cases} \hat{x} = x + \Delta x \\ \hat{y} = y + \Delta y \end{cases}$$

Avec

$$\begin{cases} \Delta x = (x_i - x_c)(k_1 r^2 + k_2 r^4) \\ \Delta y = (y_i - y_c)(k_1 r^2 + k_2 r^4) \end{cases}$$

Nous utilisons ici un polynôme de degré 4 en ne gardant que les puissances paires, ce qui est réputé pour être suffisant, il est toutefois possible d'utiliser un polynôme de degré 6, toujours en ne gardant que les puissances paires. La correction de la distorsion radiale est donc intimement liée a la recherche des coefficients k1, k2 , ...kn .

I-4-2 CORRECTION DE LA DISTORSION OPTIQUE

Avant de s'attaquer au problème de distorsion, il faut préalablement préparer les données à l'aide de méthodes de traitement d'images et de géométrie.

- **Algorithme de correction**

La méthode proposée se décompose en 6 étapes. Certaines d'entres-elles sont indépendantes mais il est conseillé de les traiter dans l'ordre. Pour pouvoir effectuer la correction automatiquement, nous proposons l'emploi d'une mire (feuille en papier) composée de disques noirs alignés selon un quadrillage défini, sur fond blanc. Connaissant la position des disques sur la mire leur position altérée sur l'image, il est alors possible de paramétrer le modèle de distorsion et finalement de corriger cette distorsion (5).

Les 6 étapes sont:

a. La détection des disques sur la mire par seuillage.
b. L'identification des 4 disques situés sur les coins du quadrillage.

c. Le calcul d'une homographie représentant la projection idéale des points de la mire sur une image sans distorsion.

d. L'identification des autres disques et leur mise en correspondance avec les

points projetés (de façon idéale) à l'aide de l'homographie calculée dans l'étape 3.

e. Le calcul des paramètres de notre modèle de distorsion radiale.

f. La correction de distorsion sur l'image distordue.

- **Définition d'une homographie :**

Une homographie est une transformation linéaire permettant de décrire la projection d'un plan sur un autre plan. Une telle transformation peut s'écrire sous forme matricielle de la façon suivante :

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ w1 \end{bmatrix}$$

$X=(x, y, 1)^T$, $X' = (x', y', w')^T = \left(\frac{x'}{w'}, \frac{y'}{w'}, 1\right)^T$

Image distordue **Image corrigée**

Figure I-9- Image distordue, image corrigée

DISCUSSION

Ce chapitre a été consacré à la présentation de notions de base nécessaires à la compréhension du fonctionnement interne des capteurs visuels. Les différentes définitions de la distance focale, du champ et angle de vue nous permettront de modéliser dans le prochain chapitre leurs comportements. En effet, notre travail consiste à reproduire le processus inverse de l'acquisition, autrement dit l'extraction des informations géométriques de la camera et ainsi celle de la scène.

Parfois, il se trouve que les images acquises présentent certaines altérations liées aux lentilles de la camera plus particulièrement celles à courte distance focale. Ces imperfections présentent des inconvénients qui peuvent fausser les calculs. Pour y remédier, nous avons présenté les étapes d'un algorithme permettant d'identifier les paramètres de correction.

Vu que l'appareil utilisé dans le cadre de ce projet ne présente pas d'effet de distorsion gênant, raison pour laquelle nous n'avons pas détaillé les étapes de l'algorithme de correction.

CHAPITRE II

TRACKING 3D

PREAMBULE

Extraire des informations spatiales relatives à l'environnement à partir d'une ou d'un lot d'images planes a été et est actuellement le but de nombreux chercheurs, en effet, il est possible d'acquérir des informations sur l'espace 3d telle que l'orientation, la distance ou le mouvement d'un objet par rapport au point de vue, en utilisant une ou plusieurs caméras qui créent un flux d'images continu, d'où il est possible, suivant des algorithmes d'estimations et de tracking, de déduire de tels informations.

Une telle solution pourrait remédier à beaucoup de problèmes rencontrés dans de nombreux domaines, tel que le domaine du cinéma, plus précisément des effets spéciaux, où il est question d'intégrer et de superposer des éléments et des objets virtuels qui interagissent avec l'environnement capturé, comme par exemple un robot qui détruit une voiture. Un autre domaine ou le tracking 3 d est convoité est le domaine militaire, en effet, étant donné que beaucoup de systèmes d'armements et de tirs actuels sont soumis aux lois newtoniennes et aux règles de la projection physique. Dans ce cas connaitre la position de la cible pourrait s'avérer être une information qui tendrait la probabilité d'atteindre la cible à 1, car connaitre la distance c'est connaitre l'angle de propulsion ou de projection. Une autre utilisation dans le même domaine pourrait être les missiles dotés de têtes chercheuses, en intégrant un système de visé optique, le missile pourrait suivre une cible à courte porté d'une manière très réactive et précise.

Il y a beaucoup d'autres domaines qui pourrait faire usage de cette technologie, tel que l'éducation ou la santé, ceux ci montrent à quel point l'évolution de cette technologie pour rait affecter d'une manière drastique la vie de tous les jours.

Dans le chapitre nous allons présenter les différentes connaissances théoriques relatives au traitement d'image, plus précisément le tracking. Nous allons définir les différents procédés suivis lors de la construction de notre application.

PROBLÉMATIQUE

L'être humain a la possibilité de voir l'espace qui l'entoure en 3d, des opérations telles que la détermination de la taille d'un objet, de la profondeur de l'environnement ou la distance d'un objet par rapport à un autre ou par rapport à celui qui voit sont faites d'une

manière automatique par le cerveau, la question est : qu'est ce qui nous permet d'acquérir de telles informations ? la réponse en ce qui concerne l'être humain est tout simplement l'œil, plus exactement sa paire d'yeux, c'est ce qui lui permet de sentir la profondeur de son environnement, en effet les yeux perçoivent deux images décalées et le cerveau analyse et interprète ce décalage, bien évidement tout ceci se passe en arrière plan, l'humain ne s'en rend pas compte.

En traitement d'image cette opération peut aussi être réalisée grâce à deux caméras, on pourrait facilement approximer la profondeur de la scène. Malheureusement cette opération nécessite obligatoirement la présence des deux caméras, condition qui n'est pas forcément respectée lorsqu'on a un ordinateur de bureau ou un ordinateur portable, on ajouterait que les caméras doivent être à une distance bien précise et doivent être le plus horizontal possible. C'est pour cela que nous allons, dans ce projet, expliquer et utiliser les méthodes de tracking 3d en utilisant une seule caméra.

De telles opérations d'estimation 3d sur un flux vidéo complet coûterait énormément de temps, ce qui nous pousse à cibler un objet précis pour réduire la taille de la zone de traitement d'image, ça nous permettra d'atteindre le niveau temps réel pour notre application, condition à respecter à tout prix vu qu'on doit asservir un robot qui doit lui aussi répondre en temps réel.

En résumé notre problème est : comment effectuer un tracking 3d en temps réel et en utilisant qu'une seule caméra ?

Chapitre II

II-1 TRACKING 3D

La première étape consiste à identifier et à tracker des cibles. Une cible est un point spécifique de l'image qu'un algorithme de tracking peut verrouiller (dans le sens militaire) et suivre sur plusieurs images. Le choix de ces cibles dépend de l'algorithme de tracking, mais ce sont souvent des endroits lumineux/sombres, des arêtes ou des coins. L'important est que chaque cible représente un point spécifique de la surface d'un objet réel. Lorsqu'elle est trackée, une cible devient une suite de coordonnées bidimensionnelles représentant la position de la cible à travers la séquence d'image. Cette suite est appelée track. Une fois que ces tracks ont été calculées, elles peuvent soit être utilisées immédiatement pour faire du match moving 2D, soit être utilisées pour calculer les informations 3D (6).

La seconde étape nécessite une résolution pour obtenir le mouvement 3D. Le but est de déduire le mouvement de la caméra en résolvant une projection inverse des chemins 2D pour la position de la caméra. Ce processus est appelé Calibrage et vient après le tracking.

II-1-1 LE MATCH MO VING

Le match moving est une technique utilisée pour injecter des images générées par ordinateur dans un flux vidéo réel. Le résultat est une illusion réaliste, qui entraine ceux qui regardent dans une confusion totale lorsque celui ci est bien exécuté. En effet l'objet apparait faisant partie de l'environnement de la vidéo, alors qu'il n'en ai rien, il a été rajouté par au niveau de la postproduction. Typiquement, le but du match moving est de faire correspondre le mouvement de la caméra réelle avec celui de la caméra virtuelle, de telle manière à ce que la perspective de la caméra virtuelle soit équivalente à celle de la caméra réelle lorsqu'elles sont fusionnées (7). Voici un exemple pour illustrer cette définition :

Figure II-10- Flux de la caméra réelle

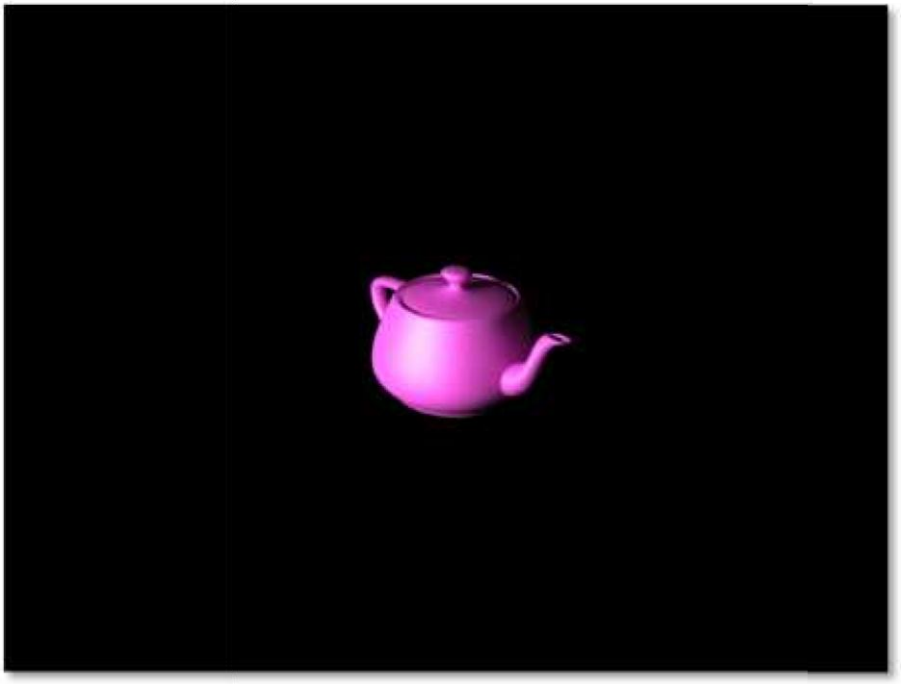


Figure II-11- Flux de la caméra virtuelle

Figure II-12- Les deux flux fusionnés

La figure (II-1) montre l'image réelle d'une scène dans une chambre avec une armoire, une décente de lit, et un trépied. Nous considérons cette image étant le flux de la caméra réelle.

La figure (II-2) montre l'objet 3D d'une théière, prise par une caméra positionné dans l'espace virtuelle 3d et contenant les mêmes attributs que la caméra réelle, afin d'atteindre la même perspective.

La figure (II-3) nous montre l'intégration de l'objet 3d dans la vrai scène (étape appelée compositing) on voit que l'objet fait partie intégrante de l'image, et c'est comme si qu'il été présent lors de la prise de l'image réelle.

Le match moving existe en deux types, le premier étant le match moving 2D, celui ci, traque les mouvements de la caméra en 2d pour intégrer des objets sans volumes ou des caractèristiques 2d comme le flou de mouvement. Ce type de match moving est suffisant lorsque l'effet spécial et le flux réel n'altère pas trop la perspective. Des logiciels comme After Effects contiennent ce genre d'outils, qui donnent à l'utilisateur le choix de traquer un certain point, de créer une trajectoire bidimensionnelle, l'effet spécial est inséré et suit la trajectoire donnée (7). Un exemple illustrant cela peut être visualisé ici:

http://www.youtube.com/watch?v=Hon6ao5v4ms

Le match moving 3d est utilisé lorsqu'on veut recréer l'espace du flux vidéo réelle virtuellement, et traquer le mouvement de la caméra. En utilisant le flux récupéré et le mouvement recréé virtuellement, on pourrait insérer des objets 3d avec la même perspective que dans le flux vidéo, comme dans les figure (II-1/2/3). En résumé, les outils du match moving tridimensionnels extrapolent les informations tridimensionnelles à partir de séquences d'image bidimensionnelles.

Le match moving est utilisé surtout dans le cinéma, la premier film à avoir utilisé cette technique est le film **Jurassic Park**. Les réalisateurs ont placé des balles de tennis colorées dans la scène comme marqueurs. Ils ont ensuite utilisé ces marqueurs pour traquer le mouvement de la caméra durant les différentes scènes. Ceci a permit à de nombreux objets virtuels. tels que des dinosaures en image de synthèses, d'être ajouté à des scènes ayant des mouvement de caméra complexes voire des caméras-épaule. Les balles de tennis ont été par la suite peintes numériquement afin de les exclure du montage final.

Le match moving est d'ores et déjà un outil reconnu dans le milieu des effets spéciaux (6).

II-1-2 DÉTECTION DE POINTS D'INTÉRÊT

Les procédures de suivi sont souvent au cœur des algorithmes de vision répondant en temps réel. Il existe une grande variété de *points d'intérêt* qui peuvent être suivis : contours, coins, repères de sous-images ou de taches, etc. La sélection de tels points est une étape importante parce qu'elle a une influence directe sur la performance du suivi. Dans la plupart des applications, sans se poser beaucoup de questions, ces points d'intérêt proviennent des détecteurs de coins ou de zones texturées. Cependant, tels bons points d'intérêt peuvent

s'avérer catastrophiques en entrée de tel algorithme de suivi au point de dire a priori : "à chaque algorithme de suivi, ses points d'intérêt !". Nous pouvons aussi distinguer les procédures de présélection hors ligne de celles qui évaluent et mettent à jour continuellement l'ensemble de points utilisé pendant le suivi.

Cette partie du chapitre présente quelque algorithme de détection de points d'intérêt utilisés aujourd'hui dans des procédures de suivi d'objets plans indéformables.

II-1-2-1 Le point d'intérêt

Dans de nombreuses applications (mise en correspondance d'images stéréoscopiques, détection ou reconnaissance d'objets, suivi....), la comparaison systématique de tous les pixels entre deux images n'est généralement pas envisageable car elle nécessite un temps de calcul très important. Le processus peut être simplifié en réduisant le domaine de recherche aux zones de l'image localement intéressantes. Les points référençant ces zones sont dénommés *points d'intérêt*.

Informellement, un point d'intérêt est associé à une discontinuité des niveaux de gris (voire des couleurs), de la texture, de la géométrie, etc. de l'image. Il est souvent assimilé à un coin. Les images de la (Figure II-4) donnent une panoplie de types de coins qui se situent à des jonctions de type "**L**", "**V**", "**T**", "**Y**", "**X**" et "**damier**" (8).

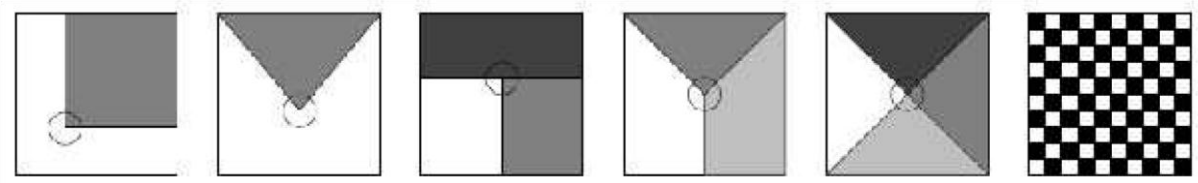


Figure II-13- Différents types de coins : (a) jonction en "L", (b) jonction en "V", (c) jonction en "T", (d) jonction en "Y", (e) jonction en "X" et (f) jonction en "damier"(8).

II-1-2-2 Les points d'intérêt et leurs détecteurs au fil du temps

Cette section présente de façon chronologique (Figure II-5) un catalogue de détecteurs de points d'intérêt publiés depuis 1976.

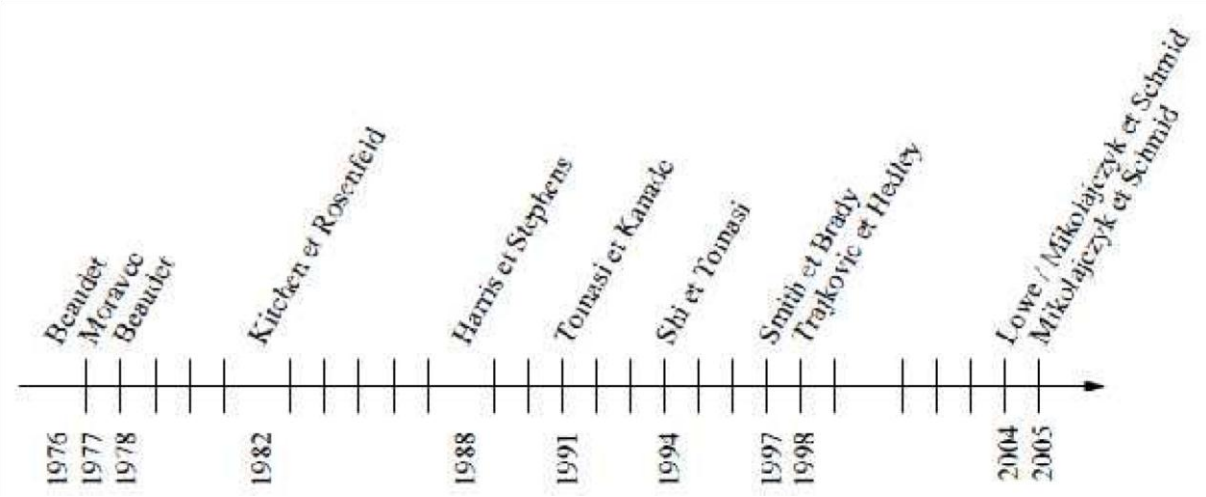


Figure II-14- Frise des détecteurs de points(9).

Dans ce qui suit nous étudierons deux méthodes parmi ces algorithmes dont *SIFT* (Scale Invariant Feature Transform) connu pour sa robustesse et son invariance à toutes sortes de changements, nous étudierons également FAST (Feature from Accelerated Segment Test) qui est connu pour sa rapidité de calcul.

II-1-2-3 Scale-Invariant Feature Transform

Scale-invariant feature transform (SIFT), que l'on peut traduire par "transformation de caractéristiques visuelles invariante à l'échelle", est un algorithme utilisé dans le domaine de la vision par ordinateur pour détecter et identifier les éléments similaires entre différentes images numériques (éléments de paysages, objets, personnes, etc.). Il a été développé en 1999 par le chercheur David Lowe (9) (10).

L'étape fondamentale de la méthode proposée par Lowe consiste à calculer ce que l'on appelle les « descripteurs SIFT » des images à étudier. Il s'agit d'informations numériques dérivées de l'analyse locale d'une image et qui caractérisent le contenu visuel de cette image de la façon la plus indépendante possible de l'échelle (« zoom » et résolution du capteur), du cadrage, de l'angle d'observation et de l'exposition (luminosité). Ainsi, deux photographies de la tour Eiffel auront toutes les chances d'avoir des descripteurs SIFT similaires, et ceci d'autant plus si les instants de prise de vue et les angles de vue sont proches. D'un autre côté, deux photographies de sujets très différents produiront selon toute vraisemblance des descripteurs SIFT très différents eux aussi (pouvoir discriminant). Cette robustesse, vérifiée dans la pratique, est une exigence fondamentale de la plupart des applications et explique en grande partie la popularité de la méthode SIFT.
Les applications de la méthode sont nombreuses et ne cessent de s'étendre ; elles couvrent au début du *XXI*^{e} siècle des domaines tels que la détection d'objet, la cartographie et la

navigation, l'assemblage de photos, la modélisation 3D, la recherche d'image par le contenu, le *tracking video* ou le match *moving*.

Cet algorithme est protégé aux États-Unis par un brevet détenu par l'université de la Colombie-Britannique.

La méthode proposée par Lowe comprend deux parties (chacune ayant fait l'objet de recherches plus ou moins indépendantes par la suite) :

- un algorithme de détection de caractéristiques et de calcul de descripteurs ;
- un algorithme de mise en correspondance proprement dit.

II-1-2-3-1 Détection des points d'intérêt

La première étape de l'algorithme est la détection des points d'intérêt, dits points-clés. Un point-clé (*x, y, σ*) est défini d'une part par ses coordonnées sur l'image (*x et y*) et d'autre part par son facteur d'échelle caractéristique (*σ*). En toute rigueur, il s'agit d'une zone d'intérêt circulaire, le rayon de la zone étant proportionnel au facteur d'échelle. Il s'ensuit une étape de reconvergence et de filtrage qui permet d'améliorer la précision sur la localisation des pointsclés et d'en éliminer un certain nombre jugés non pertinents. Chaque point-clé restant est ensuite associé à une orientation intrinsèque, c'est-à-dire ne dépendant que du contenu local de l'image autour du point clé, au facteur d'échelle considéré. Elle permet d'assurer l'invariance de la méthode à la rotation et est utilisée comme référence dans le calcul du descripteur, qui constitue la dernière étape de ce processus.

Détection d'extremums dans l'espace des échelles

La détection des points d'intérêt s'effectue dans un espace discret que l'on appelle espace des échelles (*Scale space)* qui comporte trois dimensions : les coordonnées cartésiennes *x* et *y* et le facteur d'échelle *σ* (9) (10). Il le *σ* (noté *L*) le résultat de l'échelle *σ* (noté *L*) le résultat de la convolution de l'image pa[illegible] paramètre *σ*, soit : p (, ,) =*σ*, soit :(, ,) ∗ (,)

$$L(x,y,\sigma) = G(x,y,\sigma) \quad I(x,y)$$

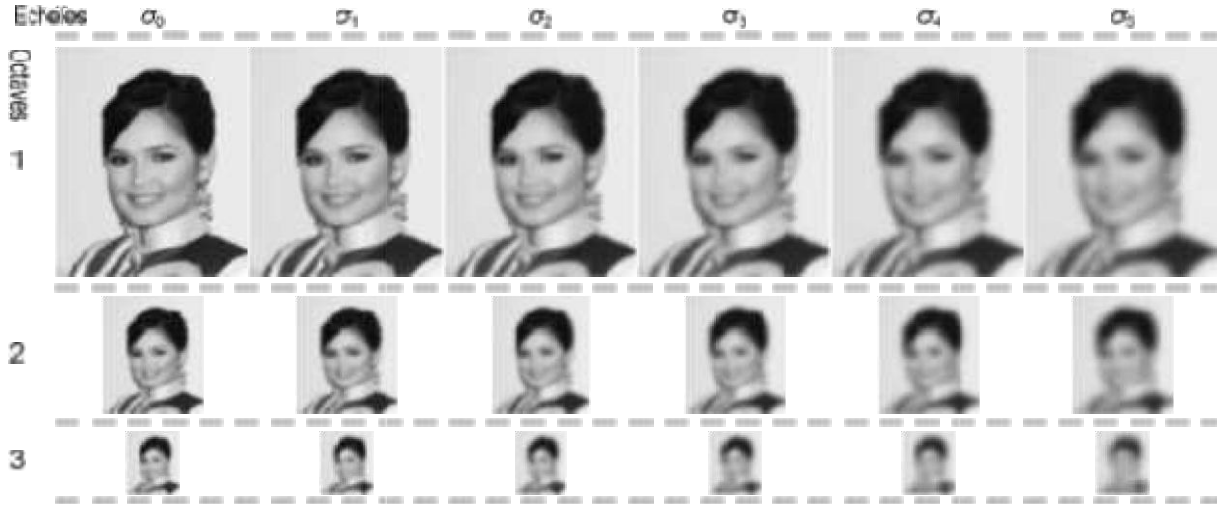


Figure II-15- Pyramide de gradients : 3 octaves de 5 gradients (9).

Cette étape fait lisser l'image originale pour enlever les petits détails et crée une pyramide avec plusieurs niveaux d'échelles de l'image originale ce qui rend la détection invariante aux changements d'échelles, chaque niveau d'échelle contient un certain nombre de niveaux de gradient. La détection des objets de dimension approximativement égale à σ se fait en étudiant l'image appelée différences de gaussiennes (en anglais différence of gaussians, DoG) définie comme suit : $D(x,y,\sigma) = L(x,y,k\sigma) - L(x,y,\sigma)$

Où *k* est un paramètre fixe choisi lors de la discrétisation de l'espace des échelles voulue.

Il ne reste plus dans l'image que les objets visibles dans des facteurs d'échelle qui varient entre σ et $k\sigma$. De ce fait, un point-clé candidat (x, y, σ) est défini comme un point où un extremum du DoG est atteint par rapport à ses voisins immédiats, c'est-à-dire sur l'ensemble contenant 26 autres points.

L'utilisation d'une pyramide est conseillée pour optimiser le temps de calcul des images floutées à un grand nombre d'échelles différentes, cette pyramide contient des *Octaves* (Octave par analogie avec la musique) qui sont obtenus a partir de la base de la pyramide qui est en général l'image originale et un niveau donné en divisant la résolution de l'image par 2. Au sein d'une même octave, le nombre de convolées à calculer est constant.

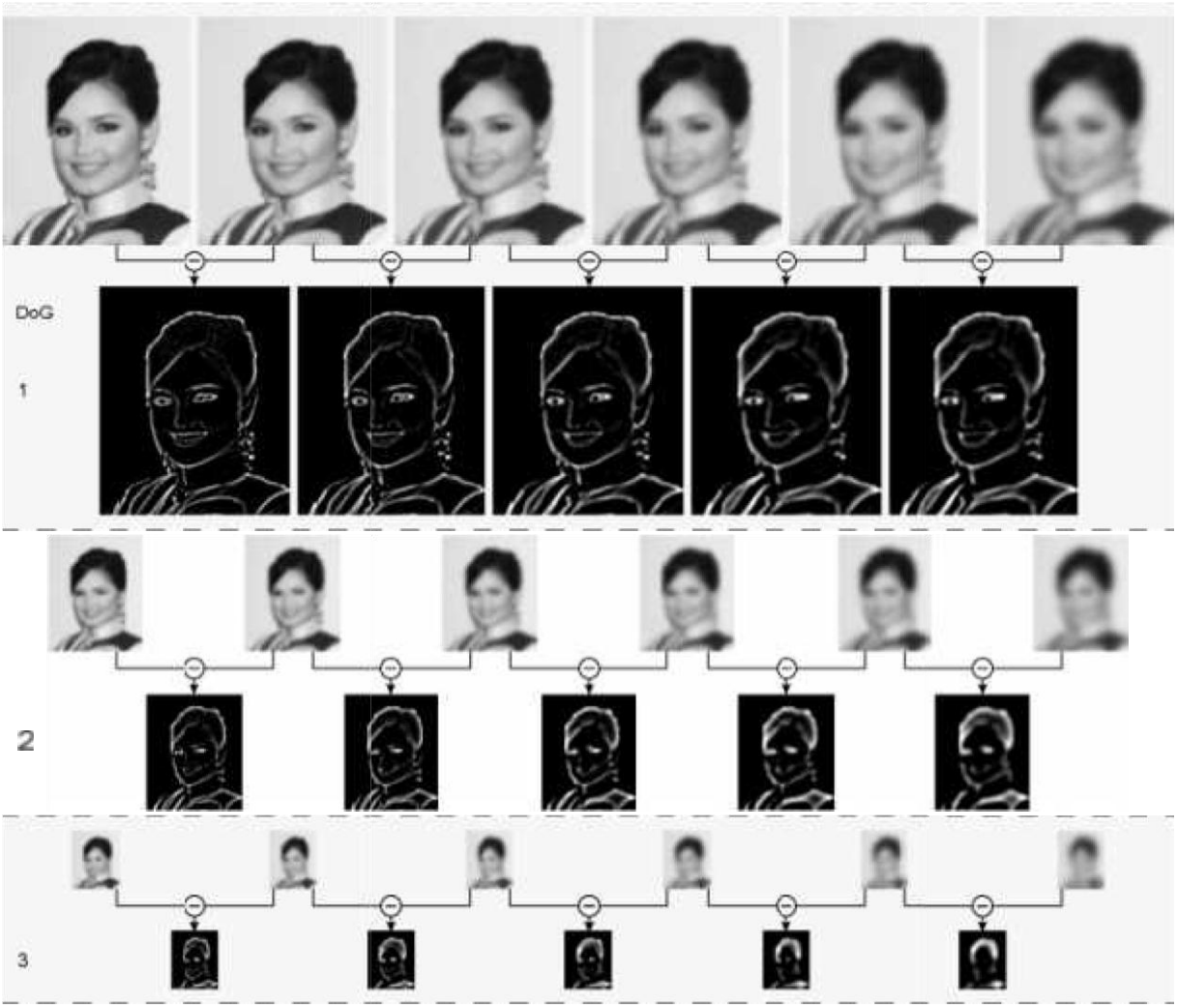


Figure II-16- Construction de la pyramide de différences de gaussiens (DoG) à partir de la pyramide de gradients (9).

Localisation précise de points clés et améliorations de localisation

L'étape de détection d'extremums produit en général un grand nombre de points-clés candidats, dont certains sont instables ; de plus, leur localisation, en particulier aux échelles les plus grandes (autrement dit dans les octaves supérieures de la pyramide où la résolution est plus faible) reste approximative. De ce fait, des traitements supplémentaires sont appliqués, pour un objectif double : d'une part, reconverger la position des points pour améliorer la précision sur x, y et σ ; d'autre part, éliminer les points de faible contraste ou situés sur des arêtes de contour à faible courbure et donc susceptibles de « glisser » facilement (9) (10).

La première amélioration est l'amélioration de la précision par interpolation des coordonnées, en visant à augmenter la stabilité et la qualité de la mise en correspondance, cette étape, qui est une amélioration de l'algorithme original, s'effectue dans l'espace des échelles à trois dimensions, où $D(x, y, \sigma)$, qui n'est pas connu que pour des valeurs discrètes de x, y et σ, doit

é canc

ion c

être interpolé. Cette interpolation est obtenue par un développement de Taylor à l'ordre 2 de la fonction différence de gaussiennes $D(x, y, \sigma)$, en prenant comme origine les coordonnées du point-clé candidat.

La deuxième amélioration consiste en l'élimination des points-clés faible contraste, la valeur de (X) aux coordonnées précises du point-clé peut être

calculée à partir du développement de Taylor de cette fonction, et constitue donc un extremum local. Un seuillage absolu sur cette valeur permet d'éliminer les points instables, à faible contraste.

Une troisième amélioration serait l'élimination des points situés sur les arêtes, les points situés sur les arêtes (ou contours) doivent être éliminés car la fonction DoG y prend des valeurs élevées, ce qui peut donner naissance à des extremums locaux instable, très sensibles au bruit : si l'image devait subir un changement numérique même imperceptible, de tels points-clés peuvent se retrouver déplacés ailleurs sur la même arête, ou même simplement disparaitre. Un point candidat à éliminer, si l'on considère les deux directions principales à sa position, est caractérisé par le fait que sa courbure principale le long du contour sur lequel il est positionné est très élevée par rapport à sa courbure dans la direction orthogonale. Cette méthode d'amélioration est inspirée de la technique de détection de points d'intérêt par l'opérateur Harris.

a ***b*** ***c***

Figure II-17- (a) détection initiale, (b) élimination des points a faible contraste, (c) élimination des points situés sur les arêtes (10).

Détermination de l'orientation

L'étape d'assignation d’orientation consiste à attribuer à chaque point-clé une ou plusieurs orientations déterminées localement sur l'image à partir de la

direction des gradients dans un voisinage autour du point. Dans la mesure où les descripteurs sont calculés relativement à ces orientations, cette étape est essentielle pour garantir l'invariance de ceux-ci à la rotation : les mêmes descripteurs doivent pouvoir être obtenus à partir d'une même image, quelle qu'en soit l'orientation.

Pour un point-clé donné (x_0, y_0, σ_0), le calcul s'effectue sur L(x, y, σ_0) , à savoir le gradient de la pyramide dont le paramètre est le plus proche du facteur d'échelle du point. De cette façon, le calcul est également invariant à l'échelle. À chaque position dans un voisinage du point-clé, on estime les différences symétriques, puis son amplitude (c.-à-d. la norme) m(x, y) , et son orientation θ(x, y) :

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = tan^{-1}\left(\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}\right)$$

Un histogramme des orientations sur le voisinage est réalisé avec 36 intervalles, couvrant chacun 10 degrés d'angle. L'histogramme est doublement pondéré : d'une part, par une fenêtre circulaire gaussienne de paramètre égal à 1,5 fois le facteur d'échelle du point-clé σ_0 ; d'autre part, par l'amplitude de chaque point.

Les pics dans cet histogramme correspondent aux orientations dominantes. Toutes les orientations dominantes permettant d'atteindre au moins 80 % de la valeur maximale sont prises en considération, ce qui provoque si nécessaire la création de points-clés supplémentaires ne différant que par leur orientation principale.

À l'issue de cette étape, un point-clé est donc défini par quatre paramètres *(x, y, σ, ϑ)*. Il est à noter qu'il est parfaitement possible qu'il y ait sur une même image plusieurs points-clés qui ne diffèrent que par un seul de ces quatre paramètres (le facteur d'échelle ou l'orientation, par exemple).

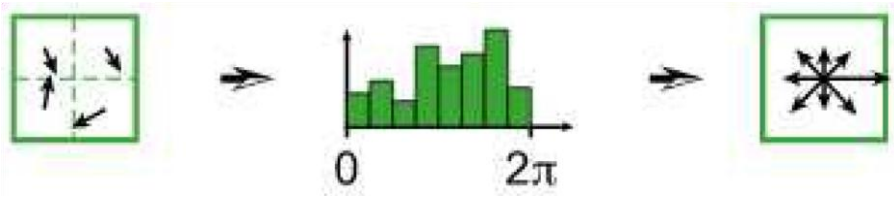


Figure II-18- Construction de l'histogramme des orientations.

II-1-2-3-2 Calcul du descripteur SIFT

Une fois les points-clés, associés à des facteurs d'échelles et à des orientations, détectées et leur invariance aux changements d'échelles et aux rotations assurée, arrive l'étape de calcul des vecteurs descripteurs, traduisant numériquement chacun de ces points-

clés. À cette occasion, des traitements supplémentaires vont permettre d'assurer un surcroît de pouvoir discriminant en rendant les descripteurs invariants à d'autres transformations telles que la luminosité, le changement de point de vue 3D, etc. Cette étape est réalisée sur l'image lissée avec le paramètre de facteur d'échelle le plus proche de celui du point-clé considéré.

Autour de ce point, on commence par modifier le système de coordonnées local pour garantir l'invariance à la rotation, en utilisant une rotation d'angle égal à l'orientation du point-clé, mais de sens opposé. On considère ensuite, toujours autour du point-clé, une région de 16 × 16 pixels, subdivisée en 4 × 4 zones de 4 × 4 pixels chacune. Sur chaque zone est calculé un histogramme des orientations comportant 8 intervalles. En chaque point de la zone, l'orientation et la magnitude du gradient sont calculés comme précédemment. L'orientation détermine l'intervalle à incrémenter dans l'histogramme, ce qui se fait avec une double pondération – par l'amplitude et par une fenêtre gaussienne centrée sur le point clé, de paramètre égal à 1,5 fois le facteur d'échelle du point-clé.

Ensuite, les 16 histogrammes à 8 intervalles chacun sont concaténés et normalisés. Dans le but de diminuer la sensibilité du descripteur aux changements de luminosité, les valeurs supérieures à 0,2 sont remplacées par 0,2 et l'histogramme est de nouveau normalisé, pour finalement fournir le descripteur SIFT du point-clé, de dimension 128.

Cette dimension peut paraître bien élevée, mais la plupart des descripteurs de dimension inférieure proposés dans la littérature présentent de moins bonnes performances dans les tâches de mise en correspondance pour un gain en coût de calculs bien modéré, en particulier quand la technique Best-Bin-First (BBF) est utilisée pour trouver le plus proche voisin. Par ailleurs, des descripteurs de plus grande dimension permettraient probablement d'améliorer les résultats, mais les gains escomptés seraient dans les faits assez limités, alors qu'à l'inverse augmenterait sensiblement le risque de sensibilité à la distorsion ou à l'occlusion. Il a également été démontré que la précision de recherche de correspondance de points dépasse 50 % dans les cas de changement de point de vue supérieur à 50 degrés, ce qui permet d'affirmer que les descripteurs SIFT sont invariants aux transformations affines modérées. Le pouvoir discriminant des descripteurs SIFT a pour sa part été évalué sur différentes tailles de bases de données de points-clés ; il en ressort que la précision de mise en correspondance est très marginalement impactée par l'augmentation de la taille de la base de données, ce qui constitue une bonne confirmation du pouvoir discriminant des descripteurs SIFT.

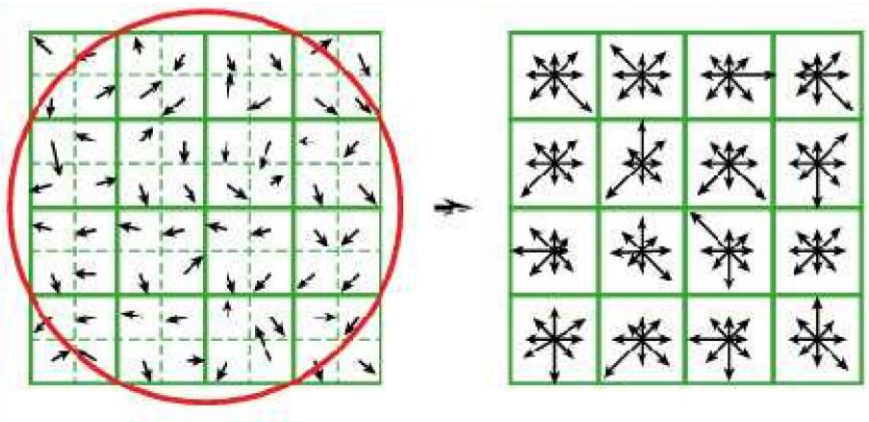


Figure II-19- Construction d'un descripteur SIFT

II-1-2-3-3 Mise en correspondance

La problématique de base pour laquelle la méthode SIFT a été conçue est la suivante : peut-on trouver dans une image donnée (dite image question ou image suspecte), des objets déjà présents dans une collection d'images de référence préétablie ?

Dans la méthode originale de David Lowe, les points-clés et les descripteurs SIFT sont tout d'abord extraits des images de référence et stockés dans une sorte de base de données. Un objet est identifié dans l'image question en effectuant une comparaison de ses descripteurs à ceux des images de référence disponibles en base de données, fondée simplement sur la distance euclidienne. Parmi toutes les correspondances ainsi établies, des sous-ensembles (clusters) sont identifiés, au sein desquels la mise en correspondance est cohérente du point de vue des positions des points, des facteurs d'échelle et des orientations. Les clusters contenant au moins trois correspondances ponctuelles sont conservés. Dans chacun d'eux, on modélise la transformation permettant de passer de l'image question à l'image de référence, et on élimine les correspondances aberrantes par simple vérification de ce modèle. Enfin, Lowe applique un modèle probabiliste pour confirmer que la détection d'une correspondance d'objets entre l'image question et l'une des images de référence n'est pas due au hasard, basé sur l'idée que si de nombreux points n'ont pas pu être mis en correspondance c'est que l'on a peut-être affaire à un faux positif.

Indexation des descripteurs et recherche de correspondances

L'indexation est l'opération de stockage des descripteurs SIFT des images de référence, d'une manière qui facilite l'identification des descripteurs correspondants de l'image question. Lowe utilise un arbre *kd* pour indexer les descripteurs puis une méthode de recherche dans cet arbre modifiée par rapport à l'approche classique, appelée *Best bin first*. Cette dernière est capable de trouver les plus proches voisins d'un descripteur question avec une bonne probabilité de façon très économe en temps de calcul. Cet algorithme se fonde sur

deux astuces. Tout d'abord, le nombre de boîtes (feuilles de l'arbre *kd*) à explorer pour trouver le plus proche voisin d'un descripteur question donné est limité à une valeur maximale fixée, par exemple à 200. Ensuite, les nœuds de l'arbre *kd* sont explorés dans l'ordre de leur distance au descripteur question, grâce à l'utilisation d'une file de priorité basée sur un tas binaire. Par distance d'un nœud à un descripteur, on entend distance euclidienne de la boîte englobante des feuilles sous-jacentes à ce descripteur. Dans la plupart des cas, ce procédé fournit la meilleure correspondance, et, dans les cas restants, un descripteur très proche de celle-ci.

De façon à déconsidérer les descripteurs faiblement discriminants (parce qu'ils sont trop souvent présents dans la base de référence, comme par exemple ceux qui sont extraits de motifs d'arrière-plan souvent répétés dans les images), Lowe recherche à la fois le plus proche voisin et le second plus proche voisin de chaque descripteur question. Lorsque le rapport des distances est supérieur à 0,8, la correspondance est éliminée, car considérée comme ambiguës (« bruit »). Par ce critère, Lowe parvient à éliminer 90 % des fausses correspondances en perdant moins de 5 % de correspondances correctes.

Identification de clusters par la transformée de Hough

On appelle pose d'un objet le point de vue sous lequel il est photographié, c'est-à-dire sa position dans l'espace par rapport à un repère absolu. En général, la base des images de référence contient différentes poses d'objets identiques.

Chaque correspondance individuelle entre points-clés obtenue à l'étape précédente constitue une hypothèse quant à la pose de l'objet sur l'image question par rapport à l'image de référence concernée. Il s'agit donc maintenant de grouper les hypothèses cohérentes entre elles de façon à mettre en correspondance les objets des images et non plus seulement des points isolés. La transformée de Hough est utilisée pour cela. Elle identifie des groupes de correspondances, que l'on appelle clusters, par un système de vote.

Alors qu'un objet rigide possède en général six degrés de liberté dans l'espace (trois rotations, trois translations), une correspondance de points-clés entre deux images ne permet de relier que quatre paramètres (deux paramètres de position, l'angle et l'échelle) et constitue donc une approximation parmi toutes les poses possibles. Le sous-espace des poses ainsi paramétré est segmenté en boîtes de façon assez grossière : Lowe utilise des intervalles de 30 degrés pour l'orientation, un facteur de 2 pour l'échelle et 0,25 fois la dimension maximale de l'image de référence concernée (compte tenu du rapport d'échelles) pour la position. Chaque correspondance vote pour la boîte associée. Les correspondances concernant les facteurs d'échelles les plus élevés, considérées comme plus pertinentes, se voient attribuer un poids double de celles concernant les facteurs les plus bas. En outre, pour éviter les effets de bord lors de l'assignation des boîtes, chaque correspondance vote pour les 2 plus proches intervalles dans chaque dimension, c'est-à-dire qu'elle vote 16 fois.

Lors de l'implémentation, Lowe recommande l'utilisation d'une table de hachage pour éviter de représenter toutes les boîtes possibles en mémoire dont seulement un petit nombre risque d'être non vide.

Lorsque plusieurs correspondances votent pour une même boîte, c'est-à-dire pour la même pose d'un objet, la probabilité que la correspondance soit correcte est largement supérieure à ce que pourrait donner une correspondance isolée. Ainsi, les boîtes contenant au moins trois entrées sont considérées comme des clusters fiables, et retenus pour la suite de l'analyse.

Vérification du modèle par la méthode des moindres carrés

Les clusters obtenus sont soumis à une procédure de vérification par l'application de la méthode des moindres carrés aux paramètres de la transformation affine reliant le modèle (image de référence) à l'image question. La transformation affine d'un point modèle $[x\ y]^T$ en un point image $[u\ v]^T$ peut s'écrire ainsi :

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Où $[t_x\ t_y]^T$ est la translation du modèle, et où les paramètres m_1, m_2, m_3 et m_4 modélisent la transformation vectorielle (qui peut être une rotation à l'origine, une similitude, une distorsion, etc.) Pour obtenir les paramètres de la transformation, l'équation ci-dessus est réécrite de manière à regrouper les inconnues dans un vecteur-colonne :

$$\begin{bmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} u \\ v \\ \vdots \end{bmatrix}$$

Cette équation représente une seule correspondance. Pour obtenir les autres correspondances, il suffit de rajouter pour chacune d'elles deux lignes à la matrice et deux coefficients au second membre. Trois correspondances au minimum sont requises pour espérer une solution. Le système linéaire c rit alors : $\approx b$

$$Ax \approx b$$

Où A est une matrice m x n connue (vérifia ement $m > n$), X un vecteur inconnu de paramètres de dimension n et enfin b un vecteur connu de mesures de dimension m.

Ce système n'a en général pas de solution exacte puisqu'il est surdimensionné ; mais ce qui nous intéresse est qu'il est possible d'[illegible] carrés, c'est-pseudol'erreur$\| X\|$, et qui est aussi solution de l'«X = b équation normale »

$$A^{\mathrm{T}}A\widehat{\mathrm{X}} = A^{\mathrm{T}}\mathrm{b}$$

solution au sens des moindres carrés, [illegible] ns linéaires [illegible] e de Moore-

$$\widehat{\mathrm{X}} \quad A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\mathrm{b}$$

La solution du système d'équations linéaires, qui s'exprime grâce à la matrice$(A^TA)^{-1}A^T$, dite pseudo-inverse de MooreX = ( ) b -Penrose de *A*, est donnée par :

Elle minimise la somme des carrés des distances entre les positions des pointsclés calculées à partir de la transformée des positions du modèle, et les positions correspondantes sur l'image cible.

Les données aberrantes peuvent ensuite être écartées en vérifiant la concordance de chaque correspondance avec la solution des moindres carrés. Sont éliminées les correspondances qui se situent au-delà de la moitié de l'intervalle d'erreur utilisé pour la discrétisation de l'espace des poses dans la transformée de Hough. Une nouvelle solution par la méthode des moindres carrés est calculée à partir des points restants, et le processus est itéré plusieurs fois. Il est également possible grâce à la transformation estimée par la méthode des moindres carrés de récupérer des correspondances qui avaient été manquées précédemment. À la fin du processus, s'il reste moins de 3 points après suppression des points aberrants, alors la correspondance est dite infructueuse et rejetée.

Validation finale par inférence bayésienne

La décision finale d'admission ou de rejet d'une mise en correspondance d'objets (hypothèse) se fait sur la base d'un modèle probabiliste. Cette méthode détermine en premier lieu le nombre attendu de fausses correspondances, connaissant la taille estimée du modèle, le nombre de points-clés dans la région et la précision de la correspondance. Une analyse par inférence bayésienne donne ensuite la probabilité que l'objet trouvé sur l'image de référence soit effectivement présent dans l'image question à partir du nombre de vraies correspondances trouvées. Une hypothèse est acceptée si cette probabilité est supérieure à 0,98. À l'exception d'images présentant de grands écarts de luminosité ou des objets ayant

subi des transformations sévères ou non rigides, la détection d'objet au moyen de la méthode SIFT parvient ainsi à d'excellents résultats.

II-1-2-4 Features from Accelerated Segment-Test

II-1-2-4-1 L'Algorithme Segment-Test

Features from accelerated segment test, que l'on peut traduire par "Caractéristiques issues de testes de segments accéléré", appartient a la classe des algorithmes qui examinent un petite partie autours d'un point candidat pour voir s'il ressemble a un coin, il utilise la définition intuitive de ce qui peut caractériser un coin, soit, une arrête qui est une frontière entre deux régions, un coin se créé lorsque l'arrête change subitement ou dévie de sa trajectoire initiale. Un point est un coin s'il y a suffisamment de pixels autour de lui qui sont dans une région différente du pixel candidat. Une autre interprétation est qu'un point est un coin si ce dernier est entouré par des pixels différents comme c'est le cas avec le détecteur USAN (11).

L'algorithme du test segment implémente cette idée en considérant un cercle autour du point candidat *p*. Il cherche le plus grand arc où les intensités de tous les points dans cet arc sont supérieur a l'intensité du pixel candidat *p* ( ) par rapport à un seuil donné noté *t*, ou alors l'intensité de tous les points dans l'arc sont inférieur a l'intensité du point candidat par *t*. le point est un coin si , où θ est égle de l'arc θ est un seuil.

Dans la pratique, l'image est discrète, donc les mesures se feront dans un cercle de Bresenham de rayon *r* autour du point candidat. Une approximation de est faite en testant juste les pixels qui sont sur le cercle de Bresenham. Ceci est utilisé pour définir le critère du test segment :

p est un point-clé si, dans le cercle de Bresenham de rayon r autour de p, il y a au moins n pixel contigus qui sont plus brillant que I_p par un plus sombre que I_p par :

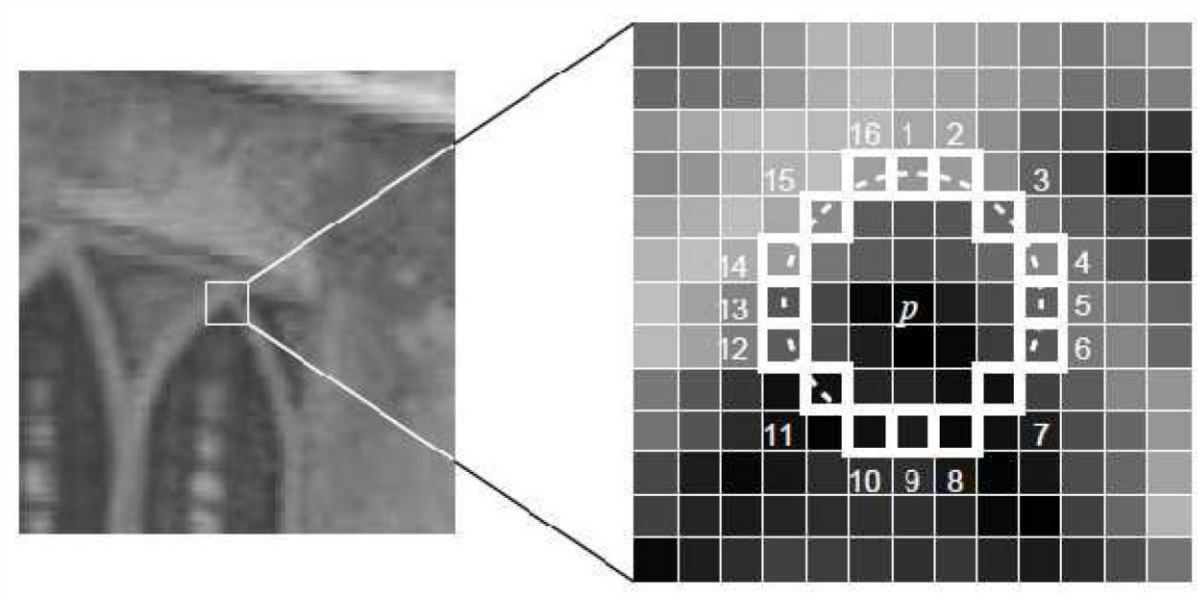


Figure II-20- 12 points de détection dans une partie de l'image (11).

Dans la (Figure II-11) les carrés en blanc sont les pixels de détection, le pixel p est le centre du coin candidat, l'arc $_{els}$ défini par les traits discontinus qui cas, $\theta = 3\pi/2$ et $\theta = 3\pi/2$ alors le point est un coin. $_{t} =$

traversent 12 pixels contigus qui sont plus brillant que le point *p* par *t*. Dans ce

Le critère donné ci dessus décrit une famille entière de détecteurs de coin du moment que le rayon du cercle sur le quel le test est effectué et le nombre de pixels contigus qu'il faut, peuvent tous les deux varier. Dans cette section nous ne considérerons que les détecteurs où r=3 pixels. Le détecteur sera utilisé sur quelques images pour une petite démonstration (Figure II-12).

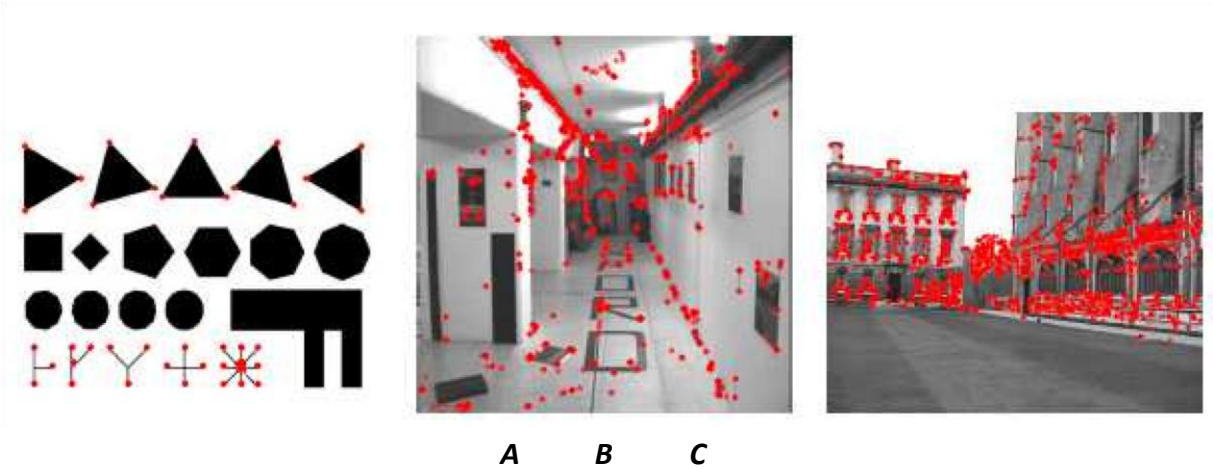
A ***B*** ***C***

Figure II-21- (A) image test contenant des polygones avec des lignes qui se joignent a des angles différents. (B) La première image de la séquence de sous sol de l'université d'Oxford (512x512). (C) une graphie du King's College, Cambridge (768x576)

Les notations suivantes seront ées dans le reste de cette section.

l'image, p est le pix nsité du pixel p. Pour chaque pixel du cercle x Є {1...16} (Figure II-11), le pixe sition x relative à p est dénoté p->x et son intensité par I_{p-x}.

Une compréhension intuitive du ge du point détecté peut être trouvé en comparant le détecteur Test Segment au détecte oG. Considérons le détecteur test segment avec $\theta = 2\pi$, un coin serait détecté s'il appar comme un point brillant entouré par des pixels sombre ou noir, ou encore un point noir entouré par des pixels brillant, le résultat est un détecteur qui est en quelque s similaire a un rapprochement quantifié du détecteur LoG/DoG. Le détecteur est invariar la rotation donc le segment de taille θ peut apparaitre avec n'importe quel angle. Les résult de l'application de cet algorithme son illustrés dans la (Figure II-13) :



A ***B*** ***C***

Figure II-22- Application du détecteur segment-test avec r=3 et n=12. (A) 318 points. (B) 1482 point (t=17). (C) 2594 points (t=50). Les points-clés sont indiqués avec des points rouges (11).

II-1-2-4-2 FAST : Accélération du test segment

Un des bénéfices clés de l'algorithme est que pour le cas de *r=3* et *n=12* le détecteur peut être écris d'une manière à être très efficace en parlant en terme de calcul, on peut le faire en utilisant l'algorithme FAST (Features from Accelerated Segment Test), si *n* pixels contigus sont exigés, alors un minimum de *n* tests sont requis pour déterminer si *p* est un coin. Cependant, si *p* n'est pas un coin, alors cela peut être souvent vérifié avec beaucoup moins de test (11). Considérons maintenant deux pixels dans des cotés opposés l'un a l'autre sur le

cercle, comme par exemple les pixels 1 et 9 dans la (Figure II-11), si ces deux pixels sont proches en terme d'intensité a celle de *p,* alors l'anneau de pixels contigus le plus large dont l'intensité est brillante ou sombre ne dépasserait pas 7 pixels de longueur, si c'est le cas, alors p n'est pas un point-clés, et ceci a était déterminé avec seulement 2 tests. Pour le cas de *n=12,* il est facile de déduire un bon ordonnancement de tests, l'algorithme complet pour *n=12* et *r=3* est présenté ci dessous, cette algorithme a était optimisé de tel façon a ce qu'il fasse le moins de tests possible :

Examiner pixels 1 et 9
Si ($\rightarrow - \leq$ ET $\rightarrow - \leq$)
p ne peut pas être un point-clé
L'algorithme se termine normalement ici
Examiner pixel 13
Si (I_p par t)
ants I_p par :)
qu
Examiner pixels 1 et 9
$|I_{p\to1} - I_p| \leq t$ ET $|I_{p\to9} - I_p| \leq t)$ *brillants que*
I_p par t)
Examiner le pixel 5
Si (3 pixels examinés sont plus brillants que
Tester p en utilisant le critère complet du test segment
Ceci est très lent, mais rarement requis.
Sinon
p n'est pas un point-clés.
Sinon Si (2 pixels examinés ou plus sont plus que I_p par :) *sombres que*
Si (seulement 2 pixels sont plus sombres I_p par :) *que*
Examiner le pixel 5
Si (3 pixels examinés sont plus sombres I_p par :)
que
Tester p en utilisant le critère complet du test segment
Sinon
p n'est pas un point-clé
Sinon
p n'est pas un point-clé

II-1-2-4-3 Marquage et Filtrage

L'algorithme du test segment a tendance a détecter plusieurs coins adjacents, pour remédier a ce problème, les coins doivent êtres marqués et les coins avec une marque qui ne sont pas des maximums locaux sont supprimés, la plus parts des algorithmes opèrent en calculant un *C* autours de chaque pixel dans l'image, si cela est fait alors la suppression non-maximal peut être effectué simplement en testant la force d'un coin candidat a tous les coins détectés qui se rapprochent de ce dernier pour voir s'ils sont des maximum locaux, ceci est fait généralement en considérant un rectangle de *3x3* (11).

Puisque le test segment ne calcule pas la fonction de réponse du coin, alors la suppression non maximale ne peut être appliquée directement sur le point-clé, par conséquence, une fonction de marquage *V* doit être calculée pour chaque coin détecté, et la suppression non-maximale doit être appliquée à *V* pour supprimer les coins qui ont (dans le carré *3x3* centré sur *p*) un coin adjacent avec un *V* plus grand.

Il y a plusieurs définitions pour *V* dont voila quelques unes :

- La valeur maximal de *n* pour la quelle *p* reste détecté comme étant un coin.
- La valeur maximale de *t* pour laquelle *p* reste détecté comme étant un coin.
- La somme de différences absolues en intensité entre les pixels contigus de l'arc et le pixel au centre.

La première et la seconde définition sont des mesures très fortement quantifiées : beaucoup de pixels auront la même valeur de marquage. Afin d'augmenter la rapidité de calcul, une version modifié de la [illegible] défini par :

être utilisée, *V* est déf[illegible]

$$V = max(\sum_{x \in B} |I_{p \to x} - I_p| - t, \sum_{x \in S} |I_p - I_{p \to x}| - t)$$

Où

$B = \{x | I_{p \to x} \geq I_p + t$ (Pixels Brillant)

$S = \{x | I_{p \to x} \leq I_p - t$ (Pixels Sombres)

Où

$=$ (Pixels Sombres)

Soustraire *t* dans l'équation précédente permet de favoriser les coins les plus forts (ceux avec une large différence d'intensité).

II-1-2 CALIBRATION

La seconde étape nécessite une résolution pour obtenir le mouvement 3D. Le but est de déduire le mouvement de la caméra en résolvant une projection inverse des chemins 2D pour la position de la caméra. Ce processus est appelé calibrage.

Plus précisément : quand un point de la surface d'un objet tridimensionnel est photographié, sa position dans l'image 2D peut être calculée par une fonction de type projection 3D. On peut considérer qu'une *caméra* est une abstraction qui contient tous les paramètres nécessaire à la modélisation d'une caméra dans un univers réel ou virtuel. Ainsi, une caméra est un vecteur qui contient comme éléments : la position de la caméra, son orientation, sa focale, et d'autres paramètres possible qui définissent comment la caméra focalise la lumière sur la pellicule. La manière dont est construit ce vecteur importe peu tant qu'il existe une fonction de projection *P* compatible.

La fonction de projection *P* prend comme entrée un vecteur de caméra (noté *camera*) et un autre vecteur représentant la position d'un point 3D dans l'espace (noté *xyz*) , et retourne un point 2D qui est le projeté du point 3D sur un plan dit image et situé devant la caméra (noté *XY*) On a alors l'expression suivante :

$$XY = P(camera, xyz)$$

La fonction de projection transforme un point 3D notamment en supprimant la composante de profondeur. Sans connaitre la profondeur, une projection inverse peut seulement retourner un ensemble de points 3D solutions. Cet ensemble est une droite partant du centre optique de la caméra et passant par le point 2D projeté. On peut exprimer la projection inverse par :

$$xyz \in P'(camera, XY) \text{ ou}$$

$$xyz : P(camera, xyz) = XY$$

Supposons que nous sommes dans le cas où les cibles que nous sommes en train de traquer sont sur la surface d'un objet rigide, par exemple un bâtiment. Comme nous savons que le point réel *xyz* restera au même endroit dans l'espace (sauf si le bâtiment se déforme) d'une image sur l'autre, on peut contraindre ce point à être constant quand bien même on ne connait pas sa position. Donc :

$$xyz_i = xyz_j$$

où les indices *i* et *j* sont des numéros arbitraires d'images de la scène que nous sommes en train d'analyser. Cela nous permet d'affirmer que :

$$P'(camera_i, XY_i) \cap P'(camera_j, XY_j) \neq \{\}$$

Du fait que la valeur XY_i a été déterminée pour toutes les images où la cible a été traquée par le programme, on peut résoudre la projection inverse entre deux images tant que $P'(camera_i, XY_i) \cap P'(camera_j, XY_j)$ est un ensemble restreint. L'ensemble des vecteurs*camera* possible qui sont solutions de l'équation aux instants i et j (noté C_{ij}).

$$C_{ij} = \{(camera_i, camera_j) : P'(camera_i, XY_i) \cap P'(camera_j, XY_j) \neq \{\}\}$$

Il y a donc un ensemble de pairs de vecteurs caméra C_{ij} pour lesquels l'intersection de la projection inverse de deux points XY_i et XY_j est non-vide, de préférence petit, et est centré autour du point théoriquement stationnaire *xyz* .
En d'autres termes, imaginez un point noir flottant dans un espace blanc et une caméra. Pour chaque position de l'espace où on place la caméra, il y a un ensemble de paramètres correspondants (orientation, focale, etc) qui vont photographier ce point exactement de la même manière. Comme C a un nombre de membres infinis, un seul point est insuffisant pour déterminer la position actuelle de la caméra.

En augmentant le nombre de points ciblés, on peut restreindre l'ensemble des positions possibles pour la caméra. Par exemple, si on dispose d'un ensemble de points $\{xyz_{i,0},...,xyz_{i,n}\}$ et $\{xyz_{j,0},...,xyz_{j,n}\}$ i et j étant toujours des indices d'image et n est un indice représentant chacune des cibles. On peut alors obtenir un ensemble de paires de vecteur-caméra

$\{C_{i,j,0},...,C_{i,j,n}\}$.

De cette manière, on restreint l'ensemble des paramètres possibles de caméra. L'ensemble des paramètres possibles qui conviennent à la caméra, F, est l'intersection de tous les ensembles :

$$F = C_{i,j,0} \cap ... \cap C_{i,j,n}$$

Plus petit est cet ensemble, plus il est facile d'approcher le vecteur-caméra solution. Cependant en pratique, des erreurs introduites par la phase de suivi impose une approche statistique pour déterminer la solution, des algorithmes d'optimisation sont souvent utilisés.. Malheureusement, il y a tellement de paramètres dans un vecteur-caméra que lorsque chacun de ces paramètres est indépendant des autres, on peut être incapable de restreindre F à une unique possibilité peu importe le nombre de points que l'on essaie de suivre. Plus le nombre

de paramètres que l'on peut restreindre lors d'une prise est grand (notamment la focale), plus il est facile de déterminer la solution.

On appelle le traitement consistant à restreindre le nombre de solutions possible du mouvement de la caméra afin d'atteindre une seule possibilité qui conviennent à la phase de compositing : phase de résolution 3D.

Projection du nuage de point

Une fois que la position de la caméra a été déterminée pour chaque image, il devient alors possible d'estimer la position de chaque cible dans l'espace réel par projection inverse. L'ensemble de points résultant est souvent nommé nuage de point du fait de son apparence nébuleuse. Comme le nuage de points révèle souvent une partie de la forme de la scène 3D, il peut être utilisé comme référence pour placer des objets en image de synthèse ou, à l'aide d'un programme de reconstruction, créer une version virtuelle de la scène réelle.

Voici une illustration d'un nuage de point créé à partir du tracking d'une caméra dans le logiciel Boujou :

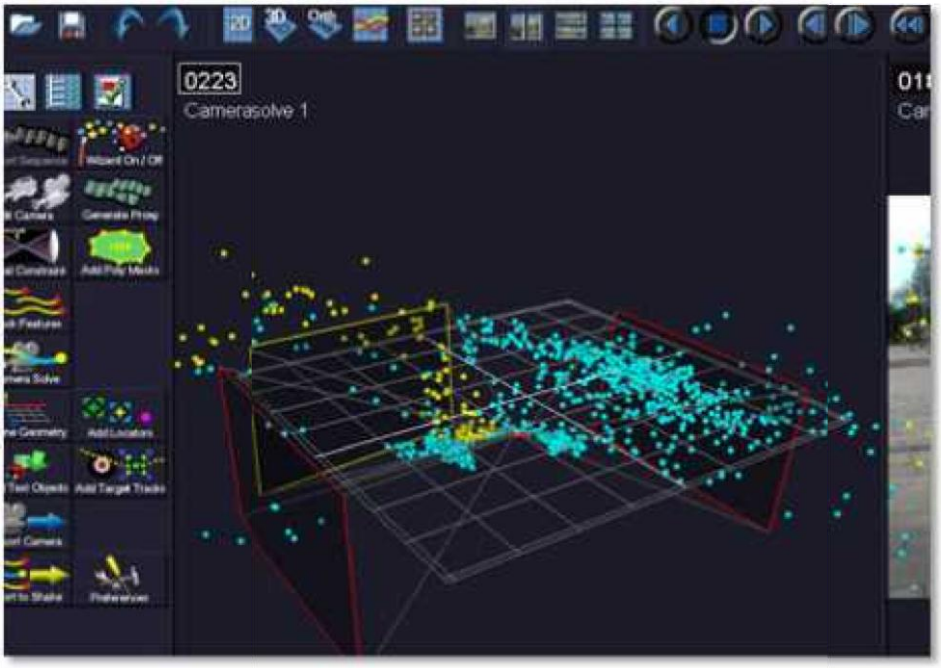


Figure II-23- Projection du nuage de point dans une scène 3D

Détermination du plan représentant le sol

La caméra et le nuage de point nécessitent d'être orientés dans l'espace. Ainsi, une fois le calibrage terminé, il est nécessaire de définir le plan représentant le sol. Normalement, il y a un plan unitaire qui détermine l'échelle, l'orientation et l'origine de l'espace projeté. Certains programmes essaient de la faire automatiquement cependant, le plus souvent, c'est l'utilisateur qui définit ce plan. Comme de la modification de ce plan ne résulte qu'une simple transformation sur tous les points, la position d'un tel plan n'est vraiment qu'une question de convention.

Reconstruction

La reconstruction est le processus interactif qui consiste à recréer un objet photographié en utilisant les données de tracking. Cette technique est liée à la **photogrammétrie**. Dans ce cas particulier, il s'agit d'utiliser le logiciel de match moving dans le but de reconstruire la scène depuis une prise adéquate.

Un programme de reconstruction peut créer des objets tridimensionnels représentants les vrais objets de la scène photographiée. En utilisant les données du nuage de point et l'estimation de l'utilisateur, le programme peut créer un objet virtuel et extraire une texture depuis la vidéo qui sera projetée sur l'objet virtuel comme texture surfacique.

II-2 THÉORIE DE NOTRE APPROCHE

Le tracking 3d est une opération très lourde en terme de calculs, en général il ne cible pas un objet voulu ou précis, il s'agit d'une estimation à partir d'un flux vidéo, une seule image contient une énorme quantité d'information, on peut trouver tellement de points d'intérêt dans une image, qu'en serait-il de toute une séquence, les standards actuels stipulent 25 images par seconde sachant que l'opération d'estimation s'effectuerait sur chacun des points trouvés, il serait presque impossible d'obtenir des résultats en direct. Dans notre application, le facteur temps réel doit être indéniablement présent, en effet notre application devra boucler, au moins, chaque 60 millisecondes, et ce pour avoir un résultat fluide. Avec tous les calculs engendrés par l'algorithme de tracking 3d, le facteur temps réel ne peut être respecté, c'est pour cela que nous allons cibler un objet, l'objet en question doit être de forme reconnaissable, et démuni de profondeur, nous opterons pour la forme carré ou rectangulaire imprimée sur un support plat, plus exactement une feuille. Cet objet nous permettrait de concentrer le calcul de tracking uniquement sur lui et d'ignorer le reste de l'image, à partir de ce moment et avec toutes ces contraintes notre choix se portera sur la réalisation d'une application de réalité augmentée.

La réalité augmentée utilise un objet cible à traquer, plus communément appelé marqueur, le processus de tracking se fera aussi d'une manière adapté à l'application, on se penchera sur les contours plutôt que sur les points d'intérêts afin de satisfaire notre contrainte principale, qu'est le temps réel.

Dans cette partie du chapitre nous exposerons théoriquement les différentes techniques et algorithmes utilisés pour la construction de l'application.

II-2-1 METHODE D'OTSU POUR LE SEUILLAGE

Dans la vision par ordinateur et le traitement d'image, la méthode d'OTSU est utilisé pour effectuer un seuillage d'une image en niveau de gris en utilisant l'histogramme de cette dernière (12), ou, la réduction d'une image en niveau de gris en binaire. L'algorithme suggère que l'image contient deux classes de pixels suivant un histogramme bi-modal, pixel de l'avant plan et pixel de l'arrière plan, il calcul ensuite le seuillage optimal qui sépare les deux classes afin que la variance intra-classe soit minimale (13). L'extension de la méthode originale vers la méthode de seuillage multi-niveau est mentionnée comme la méthode **Multi Ostu** (14). Cette méthode est nommé d'après le nom de **Nobuyuki Otsu**.

Dans cette méthode, on cherche exhaustivement le seuillage qui minimise la variance se), défi ie comme le poid de la somn

intraclasse (variance dans la é me des variances de deux

classes : ( ) ( ) + ( ) ( ) $\theta_\omega^2(t) = \omega_1(t)\theta_1^2(t) + \omega_2(t)\theta_2^2(t)$

Les poids s es probabilités des deux classes séparé par un seuillage t et variances θ_i^2 les de ces classes.

Otsu montre que minimiser la variance intra-classe et pareil que maximiser la variance

interclasse : θ (t) = θ (t) ω ωt) (t) μ (Qui e imée en

$^2_b(t)$ ^{2} − $^2_\omega(t)$ = $_1(t$ $_2(t)[\mu_1(t$ $_2(t$ ^{2} μ

terme de probabilités de cl es et les moyennes des esses La probabilité de la i

classe ω (t) est calculée de l'histog me n s l'i t : ω (t)

La moyenne de la classe μ est : = p(i

μ (t) p(x( ) /

Où x(i) est la valeur au tre du (t) n istogr 'une manière similaire nous pouvons calculer et sur la tie d e de is ramm é pour les valeurs supérieur à *t.*

$$\left[\sum_{0}^{t} \right]/\omega_1$$

Les probabilités de classe les moyennes peuvent être calculés d'une manière itérative, ce qui nous mène à l'algorithme très efficace d'Otsu.

II-2-2 DÉTECTION DE CONTOURS

Le but de la détection de contours est de repérer les points d'une image numérique qui correspondent à un changement brutal de l'intensité lumineuse. Ces changements de propriétés de l'image traduisent en général des événements importants ou des changements dans les propriétés du monde. Ils incluent des discontinuités dans la profondeur, dans l'orientation d'une surface, dans les propriétés d'un matériau et dans l'éclairage d'une scène. La détection de contour est un champ de la recherche qui appartient au traitement d'image et à la vision par ordinateur, particulièrement dans le domaine de l'extraction de caractéristiques.

La détection des contours d'une image réduit de manière significative la quantité de données et élimine les informations qu'on peut juger moins pertinentes, tout en préservant les propriétés structurelles importantes de l'image, ce qui nous aidera grandement dans la réalisation de l'application qui devra fonctionner en temps réel. Il existe un grand nombre de

méthodes de détection de l'image mais la plupart d'entre elles peuvent être regroupées en deux catégories. La première recherche les extremums de la dérivée première, en général les maximums locaux de l'intensité du gradient. La seconde recherche les annulations de la dérivée seconde, en général les annulations du laplacien ou d'une expression différentielle non-linéaire.

Dérivée première

L'opérateur de Roberts décrit dans une image rectangulaire, ayant un contour rampe, les dérivées premières elles-mêmes sont d'un intérêt limité car la pente maximale a peu de chances de se trouver sur l'une des deux directions considérées. Ce qui importe c'est la longueur du vecteur gradient donnages: au pri cor un alc tes. Cette longueur se calcule en principe par le théorème de Pyt ore x d' tière : ul sur les réels et on l'accélère considérablement en utilisant une ro en ière :|

Indépendamment de la précision du calcul, il faut transformer ce résultat en un filtre numérique. La notion physique de filtre correspond à la notion mathématique de convolution. Lorsqu'il s'agit de données numérisées comme dans le cas du traitement d'images, la relation entre les valeurs des pixels de sortie et celle des pixels d'entrée est décrite par un tableau de nombres appelé matrice de convolution.

Dérivée seconde

A priori, l'utilisation des dérivées secondes est plus commode que celle des dérivées premières : au lieu de chercher les maximums de l'intensité du gradient, on cherche l'annulation du laplacien, somme des dérivées secondes dans deux directions, soit N + S + E + W - 4C. La contrepartie de la simplicité est l'amplification du bruit par la seconde dérivation. Ce bruit est atténué par l'adjonction d'un filtre gaussien mais les méthodes du second ordre ne semblent pas compétitives avec celles du premier ordre évoquées précédemment.

Après la binarisation de l'image nous utiliserons l'implémentation de l'algorithme d'extraction de contours de suzuki (15). Celui ci génère une collection de contours externes.

II-2-3 CALIBRATION DE LA CAMERA

Les caméras sont présentes dans la vie de tous les jours depuis un certain moment, et avec l'évolution de la technologie ces systèmes sont devenus de plus en plus petits et les prix sont de moins en moins élevés, ce qui fait qu'ils sont très disponibles. Cette disponibilité n'est

pas sans prix, en effet, elle vient avec une distorsion signifiante. Par chance, les distorsions des caméras actuelles sont constantes c'est pour cela qu'avec une calibration et un remappage on peut corriger ces effets, et par la suite estimer la position d'un objet voulu et connu par rapport à la caméra.

Pour la distorsion, en ce qui concerne Opencv, ce dernier prend en compte le facteur radial et tangentiel, pour le facteu r radial il utilise cette formule : D pour un ancien point pixel dont la coordonnée est (x, y) dans l'i lont la coordonnée est (x, y) dans l'i

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + {}_3 r^6$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + {}_3 r^6$$

$$= {}_3 r^6 \quad yx((11 ++ kk\,rr ++ kk\,rr ++ kk\,rr))$$

$x_{corrected}, y_{corrected}$). La présence de littéralement effet "œil de poisson"

l'image de sortie corrigée sera (x , y). La présence de distorsion radiale se manifeste en l'effet de "fish-eye" littéralement effet "œil de poisson".

La distorsion tangentielle arrive à cause des objectifs qui prennent l'image car ils ne sont pas parfaitement parallèles au plan de l'image. Elle peut être cor e a plan de l'image. Elle peut être corrigée avec ces formules :

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{corrected} = y + [2p_2 xy + p_1(r^2 + 2y^2)]$$

Donc nous avons 5 paramètres de distorsion, les quels dans opencv sont présentés comme une matrice à une ligne et 5 colonnes :Distorsion $= (k \quad k \quad p \quad p \quad k)$ is : $_{coefficients} = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$

Pour la conversion d'unités on utilise cette formule : se cette formule :

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Ici la présence de *w* est expliquée par l'utilisation d'un système de coordonnée d'homographie et $w = Z$. Les paramètres inconnus sont qui sont les longueurs focales de la caméra de l qui sont les centres optiques exprimés en coordonnées de pixels. Si po ur les deux axes une longueur focale commune est utilisée avec un aspect ratio donné a qui qui est la plupart du temps égale à 1, alorsobtiendrons $= *$ et dans la formule citée en est expliqué l'un système f_x *et* f_y qui optiques exprimés (c_x, c_y) qui so mule est utili $f_y = f_x * a$ et f . La m

dessus une seule longueur focale . La matrice contenant ces 4 paramètres est appelée la matrice de la caméra ou la caméra intrinsèque. Tant que les coefficients de distorsion sont les même quel que soit les résolutions de la caméra utilisées, ces derniers doivent être mis à l'échelle avec la résolution actuelle calibrée.

Le processus de détermination de ces deux matrices est appelé Calibration. Le calcul de ces paramètres est fait à l'aide d'équations géométriques basiques. Les équations utilisées dépendent des objets de calibration. Actuellement opencv supporte 3 types d'objets de calibration :

- Un échiquier classique en noir et blanc (celui qu'on utilisera)
- Un schéma de cercles symétriques
- Un schéma de cercles asymétriques.

De base, nous aurons besoin de prendre des prises instantanées avec notre caméra et laisser opencv les trouver. Chaque schéma trouvé résulte en une nouvelle équation. Afin de résoudre l'équation on aura besoin d'un certain nombre déterminé de prise de schéma pour former un système d'équation correct et fiable. Ce nombre est plus grand en utilisant le schéma de l'échiquier et l'est moins avec les schémas de cercles. Par exemple, en théorie l'échiquier nécessite au moins deux prises, mais en pratique nous auront beaucoup de bruit dans l'image d'entré, donc pour de meilleurs résultats nous aurons probablement besoin d'une dizaine de bonne prise de l'image d'entré et avec de différentes positions (16).

II-2-4 RECONSTRUCTION 3D

Après la calibration de la caméra vient l'étape de l'estimation de la position de l'objet cible en 3d, ceci à pour but de déterminer son orientation et sa distance (cas d'un objet dont les proportions sont connus). Donc dans notre cas une recréation de la scène virtuellement nous permettra de visualiser l'objet dans l'espace 3D, ce en projetant les points en 3d de l'objet sur le plan de l'image en utilisant une transformation de perspective (17) :

$$sm' = A[R|t]M'$$

ou bien,

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

où :

- (X, Y, Z) sont les coordonnées d'un point 3D dans l'espace de coordonnée du monde.
- (u, v) sont les coordonnées de projections du point exprimées en pixels.
- A est la matrice intrinsèque déterminée lors de la calibration. (c_x, c_y) c'est le point principal qui d'habitude est au centre de l'image.
- f_x, f_y sont la distance focale

- sont les longueurs focales exprimées en pixels.

Ainsi, si une image issue d'une caméra est soumise à un facteur d'échelle, tous ces paramètres doivent êtres soumis au même facteur d'échelle. Les paramètres de la matrice intrinsèque ne dépendent pas de la scène perçue. Donc, une fois estimés, ils peuvent êtres réutilisés tant que la longueur focale est fixe (dans le cas des objectifs avec zoom). La fusion des matrices de rotation et de translation [R|t] est appelée la matrice des paramètres extrinsèques. Cette dernière est utilisée pour décrire les mouvements de la caméra autour de la scène statique, ou à l'inverse, les mouvements d'un objet par rapport à une caméra fixe. À savoir, [R|t]

déplace les coordonnées d'un point (X, Y, Z) à un système de coordonnée fixée en respect à la caméra. La transformation citée en haut est équivalente à celle qui suit lorsque z≠0 :

voudra dire que k_3 les coefficients de distorsion ne dépendent pas de la scène visualisée, ainsi, ils appartiennent aux paramètres de caméra intrinsèque, et ils restent les mêmes peut

ine image avec une
cients de distorsion
f_x, f_y, c_x et c_y doiv

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

x'=x/z

y'=y/z

u= $f_x * x' + c_x$

v= $f_y * y' + c_y$

Les objectifs ont tendance à toujours avoir une certaine distorsion aussi petite soit elle, principalement une distorsion radiale et un peu de distorsion tangentielle, donc le model décrit en dessus est étendu comme ceci :

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

x'=x/z

y'=y/z

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2 (r^2 + 2x'^2)$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_2 x' y' + p_1 (r^2 + 2y'^2)$$

où $r^2 = x'^2 + y'^2$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

Les opérateurs k_1, k_2, k_3, k_4, k_5 et k_6 sont les coefficients de distorsion. p_1 et p_2 sont les coefficients de distorsion tangentielle. Les coefficients d'ordre supérieur ne sont pas considérés, dans les formules en haut, les coefficients sont passés ou retournés en tant que vecteur $(k_1, k_2, p_1, p_2, [k_3 [k_4, k_5, k_6]])$. À savoir, si le vecteur contient 4 éléments, ça

importe la résolution de l'image capturée. Si par exemple une caméra est calibrée sur une image avec une résolution de 320 pixels sur 240 pixels, on peut utiliser les pendant que , , doivent êtres mis à l'échelle appropriée.

même coefficients de distorsion sur une image de 640 x 480 issue de la même caméra

La fonction qu'on utilisera dans le programme pourrait utiliser le model décrit en haut pour accomplir ce qui suit :

- Projeter les points 3D sur le plan de l'image avec les paramètres intrinsèque et extrinsèque donnés.
- Calculer les paramètres extrinsèque, avec la matrice de paramètres intrinsèque, les points 3D et leurs projections donnés.
- Estimer les paramètres intrinsèques et extrinsèques de la caméra depuis plusieurs vues d'un objet de calibration connu.

DISCUSSION

Dans ce chapitre nous avons présenté la théorie derrière les principales étapes par les quelles un système de tracking passe, la première étant la détection de points d'intérêts ou plus généralement l'extraction de caractéristiques qui nous permet de pouvoir détecter les zones de l'image les plus stables et les plus visibles que l'on peut suivre, et ce, afin de ne pas avoir de résultats erronés lors de la seconde étape qui, consiste en la calibration qui nous permet de pouvoir soit, reproduire la scène visualisée par la caméra dans un monde virtuel ou de pouvoir suivre un objet précis afin d'estimer sa position dans le monde réel, non seulement la position, mais également son orientation.

Nous avons également défini notre approche, qui nous permettra de construire une application de réalité augmentée et de pouvoir asservir par la suite un robot en temps réel. Notre choix des algorithmes et traitements a été influencé par le facteur temps, qui est dans notre application un paramètre principal, en effet si l'application n'était pas fluide, l'orientation du robot ne se serait pas mise à jour suffisamment à temps pour pouvoir suivre l'objet, ce qui nous aurait amputé cette partie Hardware.

CHAPITRE III

DEVELOPPEMENT DE L'APPLICATION

III-1 RÉALITÉ AUGMENTÉE BASÉE SUR LES MARQUEURS

Réalité Augmentée (AR) est une vue en directe de l'environnement du monde réel dont les éléments sont augmentés avec des graphismes générés par ordinateur. La technologie fonctionne en augmentant la perception courante de la réalité de celui qui l'utilise. L'augmentation, par convention, doit se faire en temps réel, et dans le contexte sémantique des éléments environnementaux. Avec l'aide de la technologie avancé des **AR** (par exemple, en ajoutant la vision par ordinateurs et la reconnaissance d'objets) l'information sur le monde réel qui entoure l'utilisateur devient interactive et peut être digitalement manipulée. Les informations artificielles à propos de l'environnement et de ses objets peuvent être superposées sur le monde réel (18).

Dans ce chapitre nous allons créer une application **AR** pour la version bureau de Windows qui manipulera, pour l'exemple, un robot doté de deux moteurs pas à pas prélevés d'anciennes imprimantes en panne. À partir de zéro, nous allons créer un programme qui va utiliser des marqueurs pour dessiner des objets artificiels sur les images acquises d'une seule caméra. Nous allons utiliser pour le traitement des images la bibliothèque très connue de **Intel** qu'est **OPENCV (Opensource Computer Vision)**, **OPENGL (Open Graphics Library)** pour l'insertion d'objets artificiels ou virtuels et **ARDUINO** pour la programmation du microcontrôleur. Nous allons décrire en détail l'architecture d'une application de réalité augmentée.

Nous expliquerons le processus complet de la détection de marqueurs. Nous estimerons la position 3d de ce dernier par rapport à la caméra, son orientation et la distance de celui ci, nous utiliserons ces transformations afin de visualiser l'objet 3d et pour asservir aussi la tourelle. Nous parcourrons dans ce chapitre ce qui va suivre :

- Les outils nécessaires
- Architecture de l'application
- Description détaillée du programme
- Calibration de la caméra
- Rendu de l'objet 3D
- Asservissement du Robot
- Fonctionnalités supplémentaires

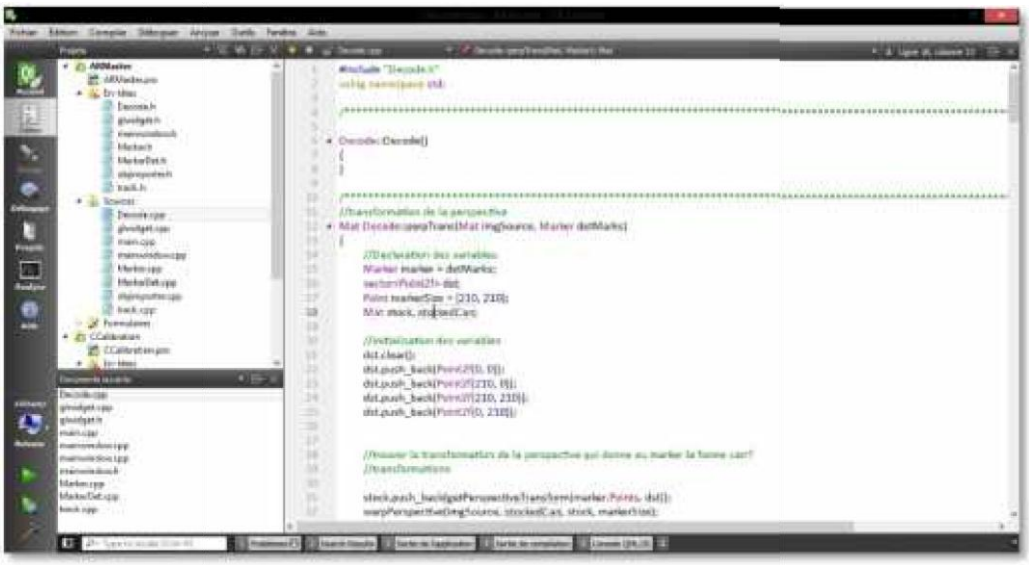
III-2 LES OUTILS NÉCESS AIRES

Nous avons besoin de décrire deux type d'outils, les logiciels ou Softwares et le matériel ou Hardware

III-2-1 SOFTWARES

- **QTCreator5.3:**

  Qt est une interface de programmation orientée objet et nous servira en tant qu'environnement de développement de notre application, nous utiliserons ses composants qui nous serviront à créer l'interface et différents autres modules que nous verrons par la suite.



- **OpenCV2.4.9:**

  OpenCV (pour Open Computer Vision) est une bibliothèque graphique libre, initialement développée par Intel, spécialisée dans le traitement d'images en temps réel (19) nous utiliserons ses fonctions pour faire tous les traitements de l'image nécessaires à notre application.

- **OpenGL4.5:**

  OpenGL (Open Graphics Library) est un ensemble normalisé de fonctions de calcul d'images 2D ou 3D (20), cette bibliothèque nous permettra de créer nos objets virtuels (3D) que nous

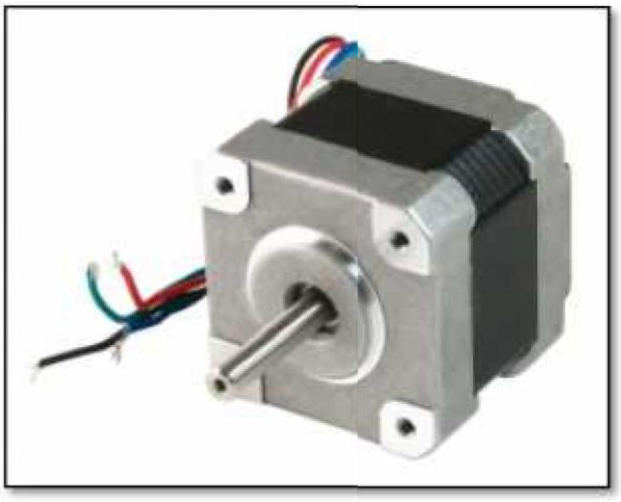
superposerons à l'image initialement envoyé par la caméra.

- **ArduinoC:**

  Interface de développement pour les microcontrôleurs **ATMEL** qui nous servira pour programmer notre microcontrôleur.

III-2-2 HARDWARE

- **MoteursPÀP:**



Nous aurons besoin de deux moteurs pas à pas, dont le pas est approximativement égale à 1.8°, ce qui sera suffisant pour avoir la précision requise pour ce genre de système de suivi de cible. Le premier moteur couvrira l'axe Z du monde réel (axe qui va du sol vers le haut) et ce sur 360°, le second moteur contrôlera la rotation autour de l'axe Y sur 360°, le résultat sera un contrôle total sur une zone sphérique, le robot n'aura aucun angle mort. Les moteurs sont Unipolaires, c'est à dire qu'on aura besoin d'alimenter 6 fils, deux vers la batterie, et 4 vers le driver ULN2003 .

- **DriversULN2003:**

L'ULN2003 est un tableau de transistor Darlington Courant-Haut et Voltage-Haut, nous en utiliserons deux, un pour chaque moteur (21), leur but et de limiter l'utilisation du courant et éviter le retour de courant, ils nous éviteront toute panne électrique.

- **ArduinoUno:**

Arduino est un circuit imprimé en matériel libre sur lequel se trouve un microcontrôleur qui peut être

programmé pour analyser et produire des signaux électriques (22), celui ci nous servira à produire les signaux qui nous aideront à diriger nos moteurs.

- **LogitechC270:**

  Camera HD (720 p) de marque Logitech, qu'on utilisera pour acquérir les images à analyser.

Maintenant que nous avons étalé tous les outils matériels et logiques nécessaires à notre application nous allons procéder à l'explication en détails de la partie software.

III-3 ARCHITECTURE DE L' APPLICATION

Notre application est composée de l'interface principale qui est **MainWindow**, nous pouvons dire que nous gérons tout à partir de cette interface, des événements de visualisation en passant par la gestion de la logique de l'application jusqu'à l'appel du moteur

de rendu 3d. l'appel à la capture de la caméra se fait à partir de cette interface.

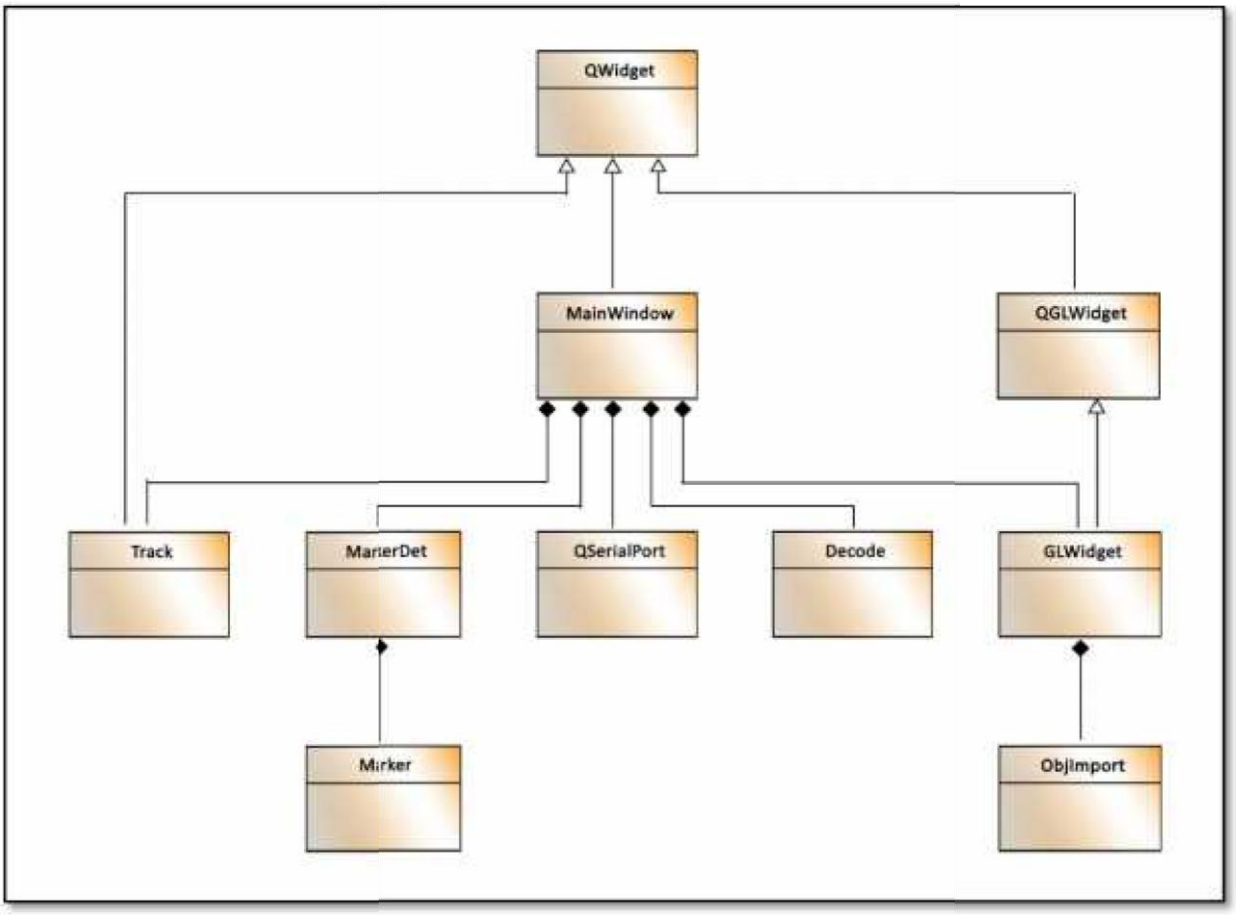


Figure III-24- Schéma des classes du programme

L'interface principale fait appel à quatre autres classes, dont **MarkerDet** qui encapsule toutes les méthodes de traitement de l'image source nécessaires à la détection d'un marqueur potentiel qu'il soit connu ou inconnu de l'application, les marqueurs détectés seront instanciés avec la classe **Marker**.

MainWindow instancie aussi la classe **Decode** qui contient toutes les méthodes qui nous servent au décodage du marqueur détecté et détermine s'il est connu de l'application ou pas.

Nous avons écris le code de ces classes de manière à ce qu'elles soient réutilisables sur n'importe quelle autre plateforme ou compilateur aussi, cette cohérence du code nous donne la liberté de changer ses composants sans réécrire le reste du code, il suffira donc de cibler la méthode que l'on veut modifier. Par exemple nous pouvons utiliser la classe **MarkerDet** facilement pour développer une application mobile sur Android ou IOS sans changer le code. On peut aussi décider d'utiliser d'autre moteur de rendu que OpenGL, tel que Direct3D, dans ce cas nous devrions juste réécrire la classe **GLWidget**, les autres parties du code resteront les mêmes.

La période du processus principale dans l'interface **MainWindow** commence lors de la réception de la nouvelle image acquise de la caméra, cette dernière envoie l'image à la classe **MarkerDet** qui détecte les marqueurs potentiels, ces marqueurs sont ensuite envoyés à la classe **Decode** ou les marqueurs sont décodés, et l'estimation de la position du marqueur par rapport à la caméra est établie. Les marqueurs reçus de cette classe sont des marqueurs valables et fiables. Le code généré sera associer à l'objet virtuel qu'on ajoutera sur l'image. Elle génère une matrice de transformation par marqueur déterminé.

Examinons de plus près ce procédé.

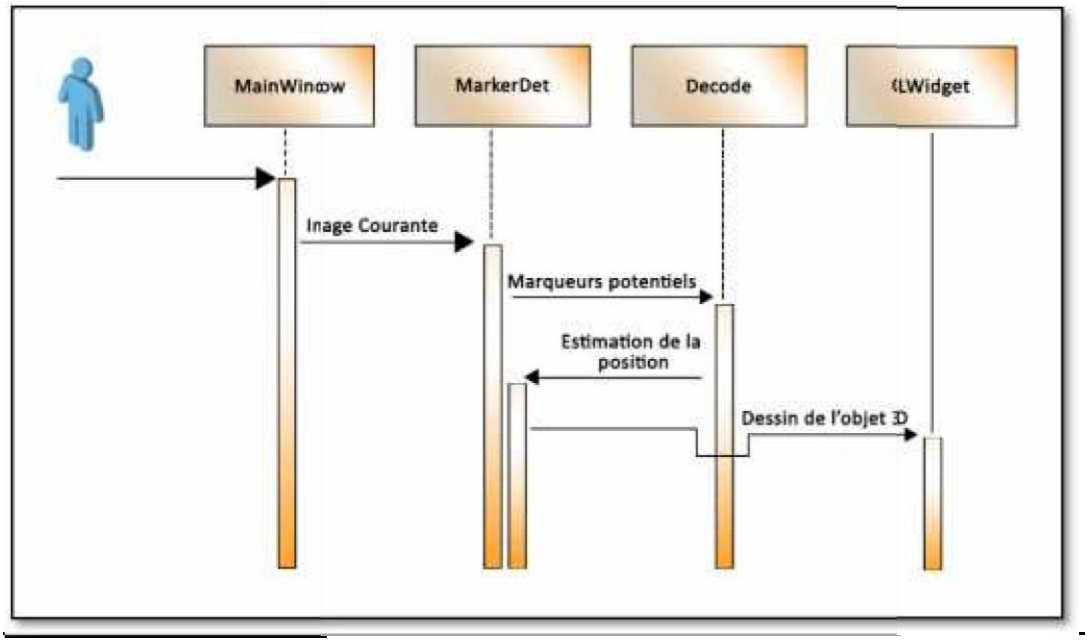


Figure III-25- Schéma du fonctionnement du programme

Nous commençons le développement de notre application par l'envoi de l'image courante à la classe **MarkerDet**, après cela nous développerons l'algorithme de détection de marqueurs. Ce processus de détection est le cœur de notre application, dans cette partie de notre programme nous utiliserons beaucoup de fonctions d'OpenCV pour traiter les images, détecter les contours en elles, trouver les marqueurs rectangulaires et estimer leurs positions. Après cela nous nous focaliserons sur la visualisation du résultat de l'estimation de la position en utilisant la réalité augmentée. Par la suite, nous asservirons le robot suivant la position du marqueurs sur l'image. À la fin nous aurons fini la construction de notre application.

III-4 DESCRIPTION DÉTAILLÉE DU PROGRAMME

III-4-1 CAPTURE VIDÉO

Les applications de réalité augmentée sont impossible à créer sans deux étapes majeures : la capture de la vidéo et la visualisation de l'objet 3D. L'étape de la capture vidéo consiste en la réception d'images acquises de la caméra, l'exécution des conversions de couleurs nécessaires, et l'envoi de celle ci à la chaine de traitement. Puisque le temps de traitement d'une seule image est critique à une application RA, le processus de capture doit être le plus efficace possible. La meilleur manière pour atteindre un maximum de performance est d'avoir un accès directe à l'image reçue de la caméra.

Pour cette raison, nous injecterons le code de récupération d'images de la caméra directement dans la classe **MainWindow** dans la méthode principale. L'ouverture de la caméra se fera dans le constructeur de la même classe, la récupération se fera exactement dans la méthode mainFct() dont l'appel se fait dans le constructeur dans une connexion de QT, nous initialiserons un QTimer pour pouvoir boucler au moins chaque 30 millisecondes.

```
//Constructeur
MainWindow::MainWindow(QWidget *parent): QMainWindow(parent),ui(new UI::MainWindow)
{
        ui-> setupUi(this);
        cap.open(0)       //ouverture de la capture

        //connexion avec la fonction principale mainFct()
        tmr = new QTimer(this);
```

```
        //Boucle à chaque signal (timeout() ) émit par le timer
        connect(tmr, SIGNAL(timeout()), this, SLOT(mainFct()))
        ; tmr-> start(30);
}

//récupération de l'image courante
Void MainWindow::mainFct()
{
        cv::Mat imgOriginale;
        cap>> imgOriginale;

        ...
        //traitements nécessaires
        ...
}
```

III-4-2 DÉTECTION D U MARQUEUR

Un marqueur est souvent représenté par une image rectangulaire contenant des zones noires et blanches à l'intérieur.

Tout d'abord nous avons besoin de trouver les contours fermés dans l'image, ensuite pour chaque contours nous étalerons son contenu sur un carré, ensuite tester si c'est un marqueur valable ou non, si oui, est il connu ou pas. exemple de marqueur 5 x 5 qu'on utilisera avec notre application:

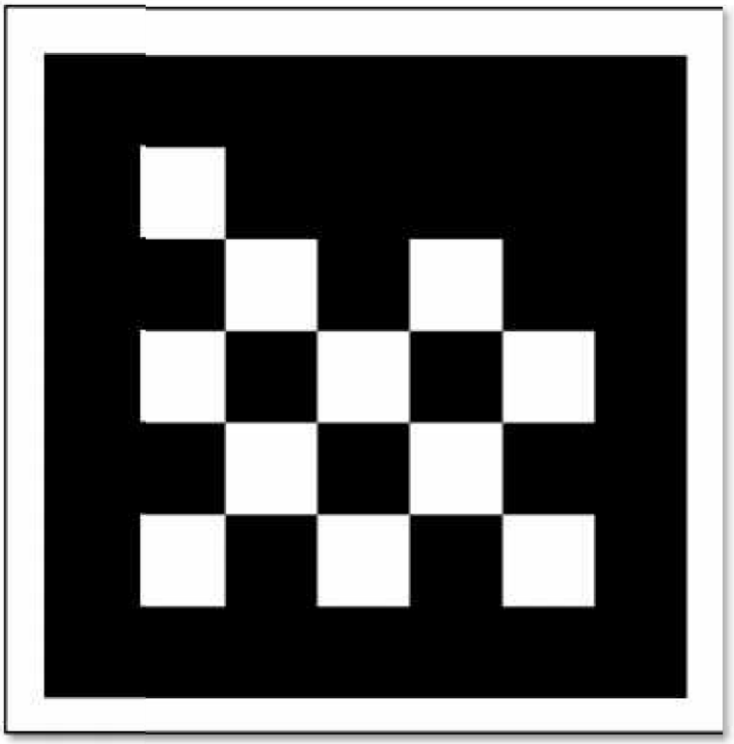


Figure III-26- Exemple de marqueur 7x7

voici l'entête de la classe **MarkerDet :**

```
Class MarkerDet
{
public :
        MarkerDet();
        Mat prepareImage(Mat imgOrig, Mat graySc); Mat performThresh(Mat& graySc, Mat&
        bin); void findCont(const Mat& bin, vector<vector<Point>>& cont, int
        minContPointsAllow); void findMarkerCand(vector<vector<Point>>& conts,
        vector<Marker>& detectedMarks);
            Mat_<float> posEstim(vector<Marker> markerDetected, vector<Point3f> obj, Mat_<float>
intrinsic, Mat distCoef, vector<Mat_<float>& glViewMatrix);
        ~MarkerDet();
}
```

III-4-3 IDENTIFICATI ON D'UN MARQUEUR

Voici le workflow du procédé de détection d'un marqueur:

- Conversion de l'image originale en niveau de gris.

- Procéder à la binarisation de l'image en niveau de gris.
- Détection des contours.
- Chercher un marqueur possible.
- Détecter un marqueur et le décoder.
- Estimer sa position en 3D.

Voici une image échantillon sur la quelle nous travaillerons:

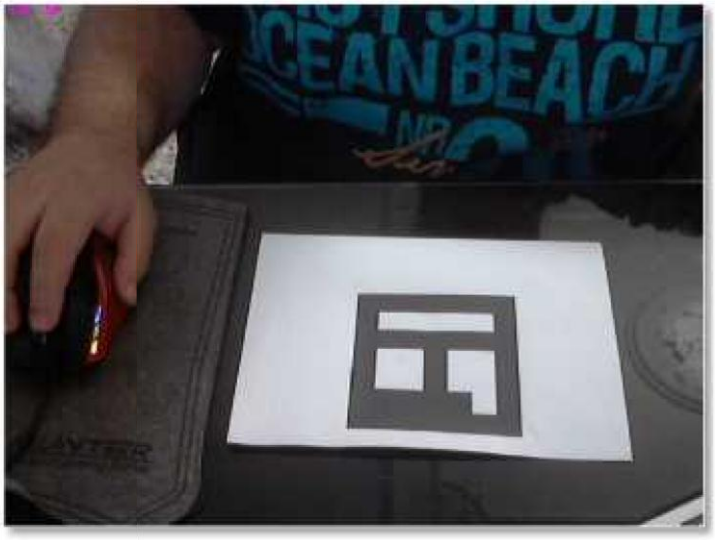


Figure III-27- Caméra capturant un marqueur

III-4-4 CONVERSION EN NIVEAU DE GRIS

La conversion en niveau de gris est nécessaire car les marqueurs, habituellement, contiennent uniquement des bloques noires ou blancs, et il est plus facile d'opérer sur eux dans une image en niveau de gris.

```
Mat MarkerDet::prepareImage(Mat imgOrig, Mat graySc)
{
        cvtColor(imgOrig, graySc, CV_BGRA2GRAY);
        return graySc;
}
```

cette fonction va convertir l'image originale RGB reçu de la caméra en image de gris soit sur un canal de 8 bits seulement, elle nous retourne l'image convertit. Tous les traitements qui suivront se feront sur l'image grise.

Figure III-28- Conversion en niveau de gris de l'image source

III-4-5 BINARISATION DE L'IMAGE

L'opération de binarisation va transformer chaque pixel de notre image en noir (intensité nulle) ou blanc (intensité maximale). Cette étape est requise pour trouver les contours. Il y a beaucoup de méthode de segmentation; chaque unes ses avantages et inconvénients. La méthode la plus facile et la plus rapide est la segmentation absolue. Le résultat de cette méthode dépend de l'intensité du pixel courant et la valeur de seuillage. Si l'intensité du pixel courant est supérieur à la valeur de seuillage, le résultat sera blanc "255"; sinon le résultat sera noir "0".

Cette méthode comporte un inconvénient, elle dépend des conditions d'éclairage et les variations subtiles d'intensité. Cela dit elle nous permet de garantir la binarisation même sur une grande distance, ce qui va nous permettre de gérer les marqueurs lointains.

Cette parcelle de code nous montre la méthode de binarisation:

```
Mat MarkerDet::performThresh(Mat& graySc, Mat& bin)
{
        threshold(graySc, bin, 125, 255, THRESH_BINARY_INV);
        return bin;
}
```

Après l'application du seuillage à l'image en niveau de gris, le résultat sur l'image est similaire à l'image qui suit:

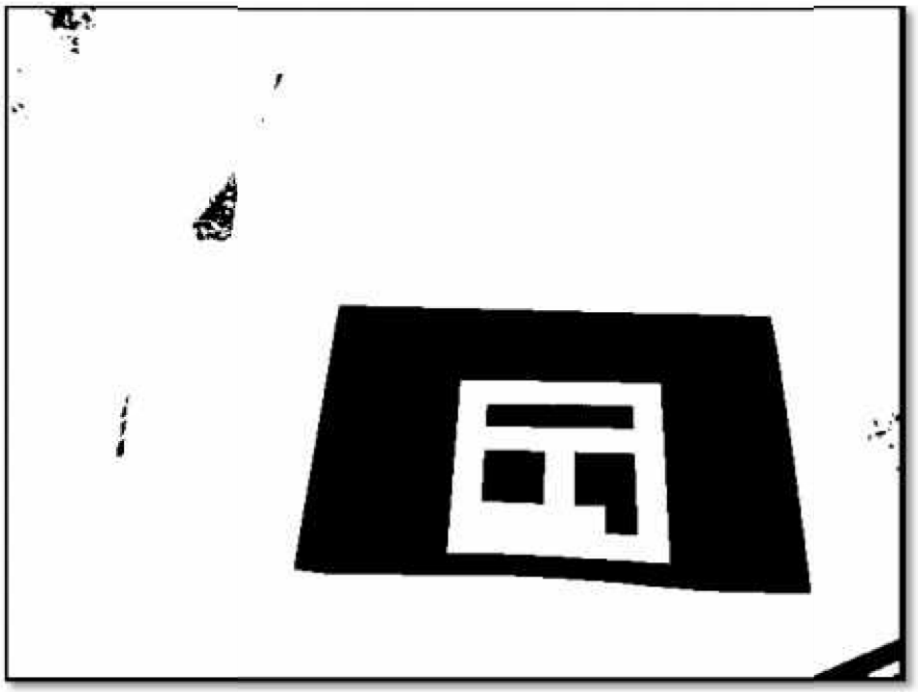


Figure III-29- Binarisation de l'image en niveau de gris

Chaque marqueur, généralement, apparait comme une figure carrée avec des zones noires et blanches à l'intérieur, donc la meilleur façon de localiser un marqueur est de trouver les contours fermés et de les approximer avec un polygone à quatre points.

III-4-6 DÉTECTION DE S CONTOURS

La fonction *findContours()* va détecter les contours dans l'image binaire en entrée :

```
void MarkerDet::findCont(const Mat& bin, vector<vector<Point>>& cont, int minContPointsAllow)
{
        vector<vector<Point> allCont; vector<Vec4i> hierarchy; findContours(bin,
        allCont, hierarchy, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);

        cont.clear(); for(size_t i=0;  i <
        allCont.size(); i++)

        {
                int contSize = allCont[i].size(); if
                (contSize > minContPointsAllow)

                {
                        cont.push_back(allCont[i]);
```

```
                }
            }
        }
```

La valeur retournée par cette fonction est une liste de polygones, chaque polygone représente un contour unique. La fonction ignore les contours avec un périmètre en valeur de pixel supérieur à la valeur de *minContoursAllow*, et ce, parce que nous ne somme pas intéressés par les petits contours.

Figure III-30- Affichage des contours extraits de l'image binaire

III-4-7 RECHERCHE DU CANDIDAT

Après avoir trouvé les contours, l'étape d'approximation du polygone est atteinte. Ceci est fait pour diminuer le nombre de points qui décrivent la forme d'un contour. C'est un bon moyen pour filtrer les contours qui ne sont pas un marqueur, parce qu'un marqueur doit toujours être représenté par un polygone à quatre sommets, si l'approximation polygonale a plus ou moins que 4 points ou sommets elle n'est définitivement plus un marqueur potentiel. La parcelle de code suivante implémente cette idée :

```
void MarkerDet::findMarkerCand(vector<vector<Point>>& conts, vector<Marker>& detectedMarks)
{
        for( site_t i=0; i< conts.size(); i++)
        {
                double eps = conts[i].size() * 0.05;
                approxPolyDP(conts[i], approxCurve, eps, true);

                //nous filtrons les polygones à 4 points et ceux qui sont
                convexes if (approxCurve.size() != 4) continue; if
                (isContourConvex(approxCurve)) continue;

                //s'assurer que la distance entre les  points consécutifs soit assez large
                float minDist = numeric_limits<float>::max();
                for( int i=0; i<4 ; i++)
                {
                        Point side = approxCurve[i] - approxCurve[(i+1)%4];
                        float squaredSideLength = side.dot(side);
                       minDist = min(minDist, squaredSideLength);
                }

                //si tous les testes sont ok alors on sauvegarde le candidat
                Marker m ;
                for(int i =0; i<4; i++)
                m.Points.push_back(Point2f(approxCurve[i].x, approxCurve[i].y));
```

```
...
// retourner les candidats
detectedMarks.clear(); for(size_t i=0; i<
possibleMarks.size(); i++)
{
```



```
        detectedMarks.push_back(possibleMarkers[i]);
    }
}
```

Maintenant nous avons obtenu une liste de parallélépipèdes qui seront avec un peu de chance des marqueurs. Pour vérifier s'ils sont des marqueurs ou pas, on a besoin de trois étapes :

1. Premièrement, nous devons supprimer la projection en perspective pour obtenir ainsi une vue de face carrée.
2. Ensuite on procède à la binarisation de cette image en utilisant l'algorithme d'**Otsu**. Cet algorithme procède selon une distribution bimodale, il trouve ainsi la valeur de seuillage qui maximise la variance extra-class, et minimise la variance intra-class; en d'autres termes il arrive à segmenter l'image d'une manière autonome, tout en enlevant le bruit qui risque d'être présent.
3. À la fin nous procédons à l'identification du marqueur. Si c'est un marqueur alors il doit contenir un code interne. Le marqueur est divisé sur une grille de 7 x7, dans la quelle les 5x5 cellules contiennent l'information de l'ID du marqueur, le reste correspond à la bordure noire du marqueur. Au début nous analysons la présence de la bordure noire, si elle est présente alors nous lisons le code interne et nous testons si le code est existant ou pas. Sachant qu'un code pourrait avoir quatre rotations possibles, ce phénomène pourrait nous fausser le calcule, une invariance à la rotation est nécessaire.

Pour avoir une image carrée du marqueur, nous devons étaler celle ci en utilisant les transformation de perspective. Cette matrice peut être calculée avec l'aide de la fonction *getPerspectiveTransform.* Elle trouve la transformation perspective à partir de quatre paires de points correspondants au marqueur. Le premier argument contient les coordonnées du marqueur sur l'image, le second correspond aux coordonnées de l'image carrée du marqueur. La transformation estimée, transformera le marqueur en une image carrée du marqueur. Voici un exemple du code:

```
Mat Decode::perpTrans(Mat imgSource, Marker detMarks)
```

```
{
        //l'image sur la quelle on étalera le marqueur en
        perspective vector<Point2f> dst; dst.clear();

        dst.push_back(Point2f(0,0));
        dst.push_back(Point2f(210,0));
        dst.push_back(Point2f(210,210));
        dst.push_back(Point2f(0,210)); //Procedure d'étalage
        stock.push_back(getPerspectiveTransform(marker.Points,
        dst)); warpPerspective(imgSource, stockedCan, Stock,
        markerSize);


        return stockedCan;
}
```

Voici le résultat obtenu :

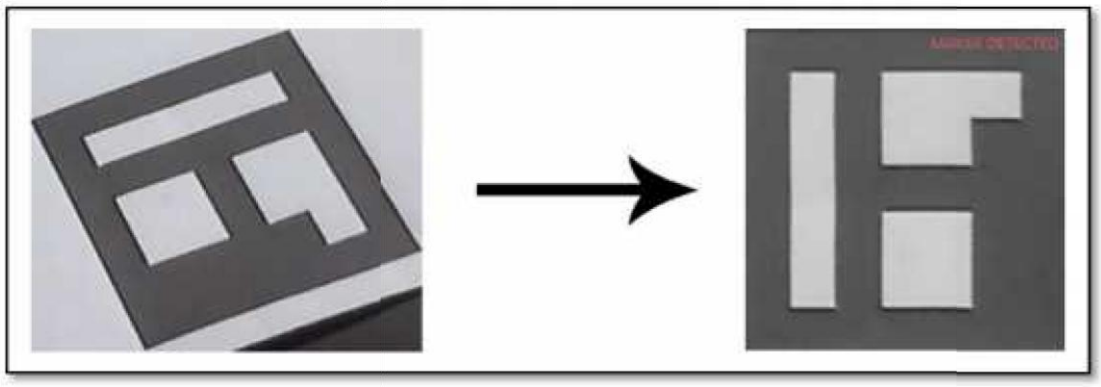


Figure III-31- Suppression de la perspective

Maintenant nous testons l'image pour vérifier si c'est un marqueur valide. Ensuite nous extrayons le code en binaire. Comme notre marqueur est sensé avoir uniquement des bloques de couleur noir ou blanc, nous pouvons procéder à la binarisation en utilisant l'algorithme d'**Otsu** pour enlever les pixels gris en laissant uniquement les pixels noirs et blancs:

```
//Seuillage de l'image
threshold(markerImage, markerImage, 125, 255, THRESH_BINARY | THRESH_OTSU);
```

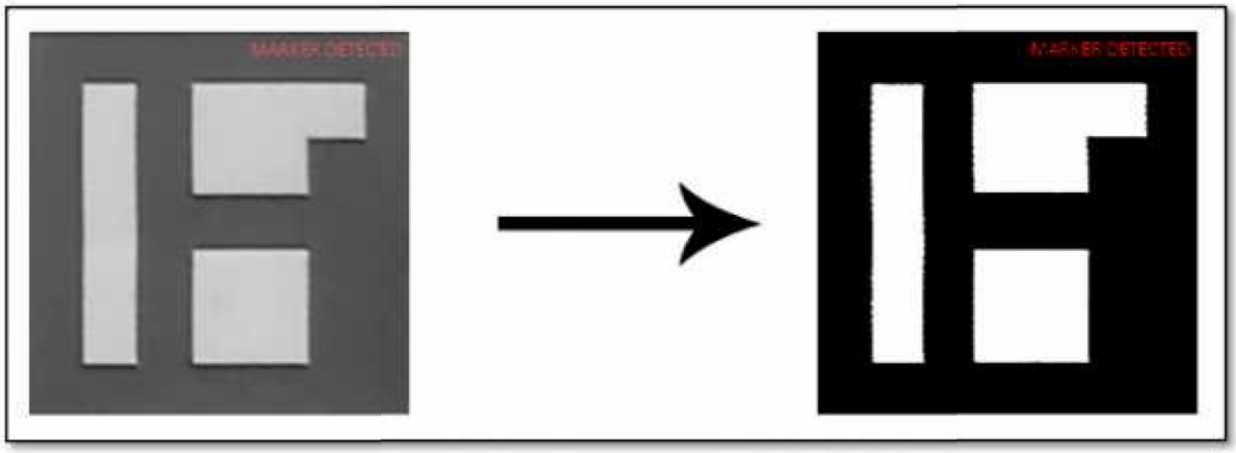
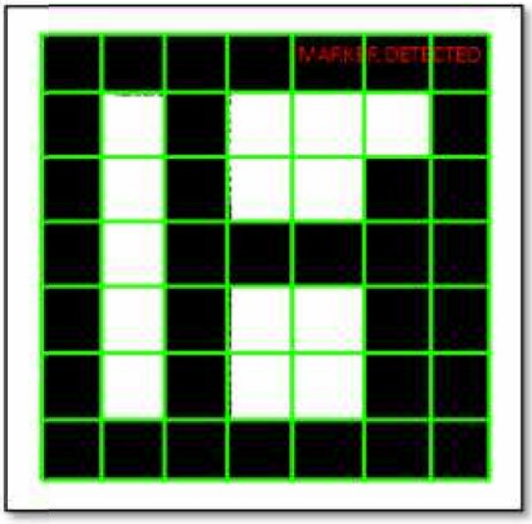


Figure III-32- Binarisation du marqueur

donn
otatic
20 —



III-4-8 RECONNAISA NCE DU CODE

Chaque marqueur a un code interne donné par 5 mots de 5 bits chacun. La codification. Le premier mot est réservé à la rotation, les quatre autres sont utiliser à la codification, en tout on peut avoir jusqu'à $2 - 2$ codes de marqueurs utilisable soit 1048574 marqueur possibles.

Figure III-33- Partitionnement du marqueur

Compter le nombre de pixels noirs et blancs pour chaque cellule nous donne un masque binaire de 5x5 représentant le code du marqueur. Pour compter le nombre de non-zero pixels (pixels non noirs) dans une image, on utilise la fonction countNonZero(), la fonction compte les pixels qui ne sont pas noirs dans une matrice. Dans notre cas, on peut obtenir une portion de l'image, qui représentera la cellule, c'est un genre de sous image, une image qui contient la portion d'une image plus grande. dans notre application une image de 210 x 210 pixels est utilisé, on pourra donc tirer 7 x 7 cellule de cette image de 30 x 30 pixels.

III-4-9 LECTURE DU C ODE DU MARQUEUR

En utilisant la technique qui suit, nous pouvons facilement trouver les cellules noires et blanches du marqueur:

```
vector<vector<int>> Decode::getCode(Mat& img)
{
    vector<vector<int>> temp;
    vector<int> temp1 ;

    temp.clear();
    temp1.clear(); for(int y = 0
    ; y<7; y++)
    {
        temp1.clear(); for(int
        x =  0; x< 7; x ++)
        {
            int cellX = x* 30;
            int cellY = y*30;

            Mat cell = img(Rect(cellX, cellY, 30,
            30)); int nZ = countNonZero(cell); if(nZ
            < (30*30)/2) temp1.push_back(1);

            else temp1.push_back(0);
        }
        if (temp1.size() != 0) temp.push_back(temp1);
    }
```

```
        return temp;
    }
```

En regardant la figure qui suit, le même marqueur peut avoir quatre représentations possibles, dépendamment du point de vue de la caméra:

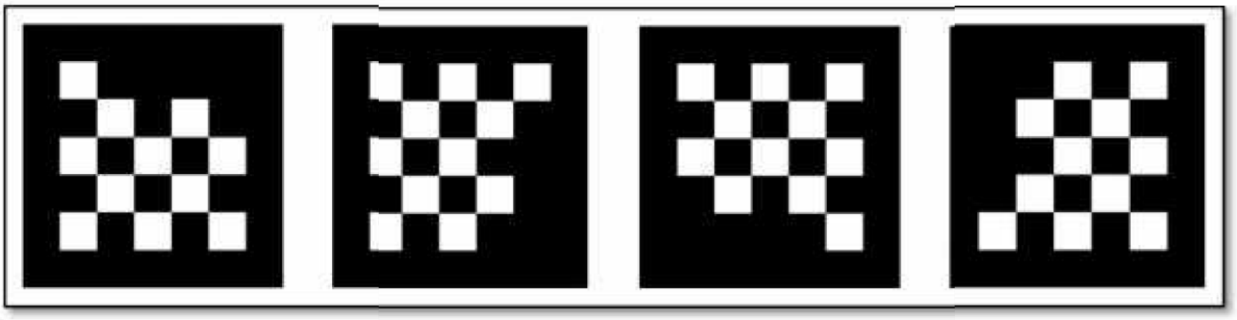


Figure III-34- 4 représentations possibles d'un seul marqueur

Du moment qu'il y a quatre orientations possible du marqueur, nous devons trouver la position correcte de ce dernier.

Pour cela nous allons assigner le premier mot du code du marqueur exclusivement à la rotation, il suffit donc de retrouver le code assigné à la rotation dans le code du marqueur pour pouvoir déduire l'orientation de celui ci. Le code en question est, en pixels, (noir, blanc, noir, noir, noir, noir, noir), en binaire, (0,1,0,0,0,0,0).

Ce code de rotation, en plus de nous offrir la position du marqueur, il nous aide aussi à déduire si le marqueur est bien un marqueur ou pas, il nous donne donc plus de précision. En effet avant d'analyser et de récupérer tout le code du marqueur (opération qui pourrait

être couteuse en terme de calculs), on pourrait limiter les erreurs dues aux faux positifs et ainsi les ignorer avant l'analyse complète du code.

Il suffit pour cela de parcourir d'abord de la première ligne du code de gauche à droite en même temps parcourir la dernière, et les comparer au code de rotation, si la première d'entre elles s'avère être égale au code de rotation on renvoi l'index 1 et on renvoi 3 dans le cas de la dernière ligne, sinon, nous parcourrons la première colonne et la dernière, et nous comparons celles ci au code, si la première colonne égale au code de rotation alors on renvoi 4, si c'est la dernière qui est égale au code on renvoi 2. Sinon notre marqueur n'est tout simplement pas un marqueur valable, et est un faux positif.

Ainsi nous pouvons indexer le sens de la rotation à effectuer, en même temps nous gagnons un temps de calculs précieux. Voici un bout de code implémentant cette idée :

```
// Indexation de la rotation int
Decode::rotation (vector<vector<int>> vec)
{
        vector<int> rot{0,1,0,0,0,0,0};
        vector<int> temp;
        temp.clear();
        for(int i = 0; i< 7; i++)
        {
                temp.push_back(vec[1][i]);
        }
        if (cmpVec(temp, rot) == true) return 0;
        else
        {
                temp.clear();
                for (int i = 0; i<7; i++)
                {
                        temp.push_back(vec[i][5]);
                }
                if (cmpVec(temp,rot) == true) return  1;
                else
                {
                        temp.clear();
                        for(int i = 6; i>=0; i --)
                        {
                                temp.push_back(vec[5][i]);
                        }
                        if (cmpVec(temp, rot) == true) return 2;
                        else
                        {
                                        temp.clear();
                                for (int i = 6; i> =0; i--)
                                {
                                        temp.push_back(vec[5][i]);
```

```
                                    }
                                    if (cmpVec(temp, rot) == true) return 3;
                                        else return 0;
                                }
                            }
                        }
    }
// annulation des faux positifs bool
Decode::falsePositive(vector<vector<int>> vec)

{ vector <int > add; int
        cmp =0;

        for (int i =0; i< vec.size(); i++)
        { add = vec[i]; for (int j =  0; j<
                add.size(); j ++)

                cmp += add[i];
        } if (cmp ==0) return
        true;

        if (cmp !=) return false;
}
```

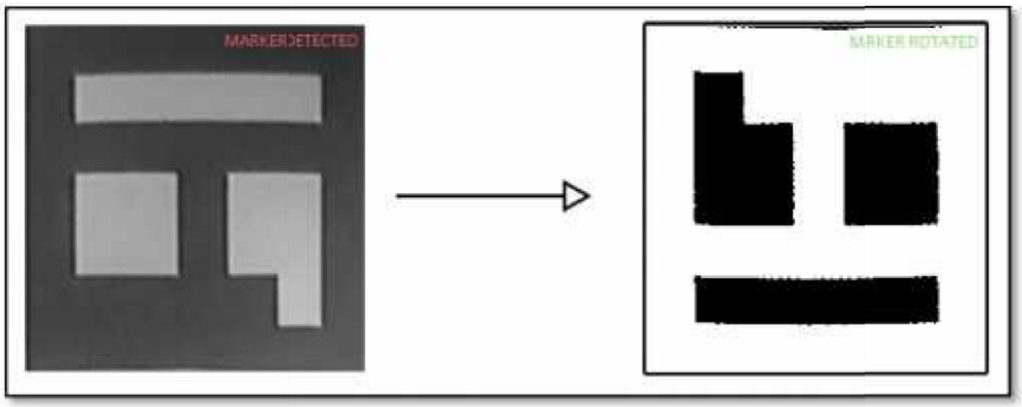


Figure III-35- Rotation du marqueur pour reconnaissance Développement de l'application

III-4-10 RAFFINEMENT DE LA POSITION DU MARQUEUR

Après avoir trouvé l'orientation correcte du marqueur, nous allons effectuer une rotation aux sommets du marqueur conformément à leur ordre:

```
for (int i =0; i < markerCandidats.size(); i++)
{
        rotate(markerCandidats[i].Points.begin(), markerCandidats[i].Points.begin() + rotVal,
markerCandidats[i].Points.end());
}
```

Maintenant que nous avons détecté les marqueurs et décodé leurs identifiants, nous allons raffiner la position de leurs sommets, cette opération va nous aider lors de la prochaine étape, lors de l'estimation de la position par rapport à la caméra.

Pour trouver la position des sommets avec un niveau de précision de sous pixels, la fonction *cornerSubPix()* sera utilisée:

```
void Decode::subP(Mat gray, vector<Marker>& mark)
{
        vector<Point2f> precCorn(4 * mark.size());
        for (size_t i=0; i<mark.size(); i++)
        {
                Marker& marker = mark[i];
                for (int c = 0; c< 4; c++) precCorn[i*4+c] = marker.Points[c];
        }
        cornerSubPix(gray, precCorn, cvSize(11, 11), cvSize(-1,-1), cvTermCriteria(CV_TERMCRIT_ITER,
30 , 0.1));

        for(size_t  i=0; i <mark.size(); i++)
        {
                Marker& marker = mark[i];
                for (int c =0; c< 4; c++)
                {
                        marker.Points[c] = precCorn[i*4+c];
                mark[i] = marker;
                }
        }
}
```

La première étape est de préparer les donnée à entrer dans la fonction, on copie la liste des sommets dans la matrice d'entré, nous appelons par la suite la fonction *cornerSubPix(),* en

passant en argument, l'image actuelle, la liste des points, et certains paramètres qui affectent la qualité et la performance du raffinement de la localisation. Quand ça sera fait, nous recopierons les positions raffinées en écrasant les sommets du marqueur. Voici le résultat :

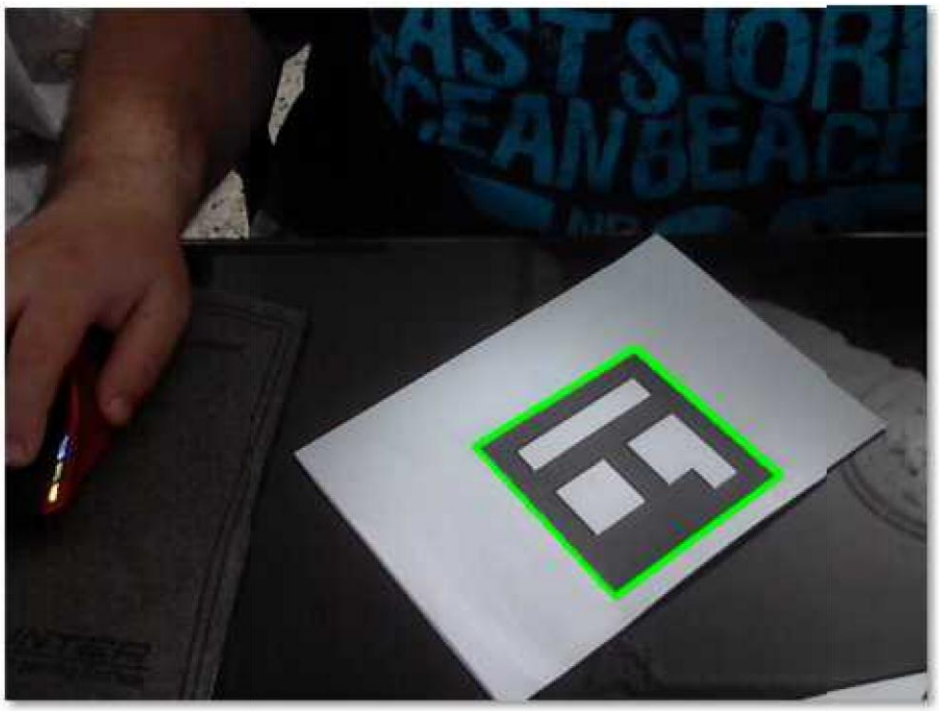


Figure III-36- Position du marqueur améliorée

Nous n'avons pas utilisé la fonction *cornerSubPix* dans les étapes précédentes de détection des marqueurs car celle ci est très complexe, et il est très chers d'appeler cette fonction pour un nombre élevé de points (en terme de temps de calcul). C'est pour cela que nous l'avons fait uniquement pour les marqueurs valides.

La prochaine étape que nous allons détailler concerne l'estimation de la position du marqueur par rapport à la caméra, cela dit pour cette étape nous aurons besoin des paramètres intrinsèques de la caméra, comme nous l'avons dit plus haut, nous avons voulu rendre l'application utilisable peut importe la caméra utilisée, pour cela nous avons codé une application en parallèle qui fait la calibration de la caméra, cette dernière nous donne en sortie un fichier contenant, les matrices intrinsèques de la caméra ainsi que le coefficient de distorsion. Nous en parlerons après l'estimation.

III-4-11 ESTIMATIO N DE LA POSITION DU MARQUEUR

Avec les positions précises des sommets des marqueurs, nous pouvons estimer la transformation entre la caméra et la position du marqueur dans l'espace 3 D. Cette opération

est connu en tant qu'estimation de la position à partir de correspondances 2D-3D. Le processus d'estimation de la position trouve les transformations Euclidiennes (qui consistent uniquement en les composants de rotation et de translation) entre la caméra et l'objet.

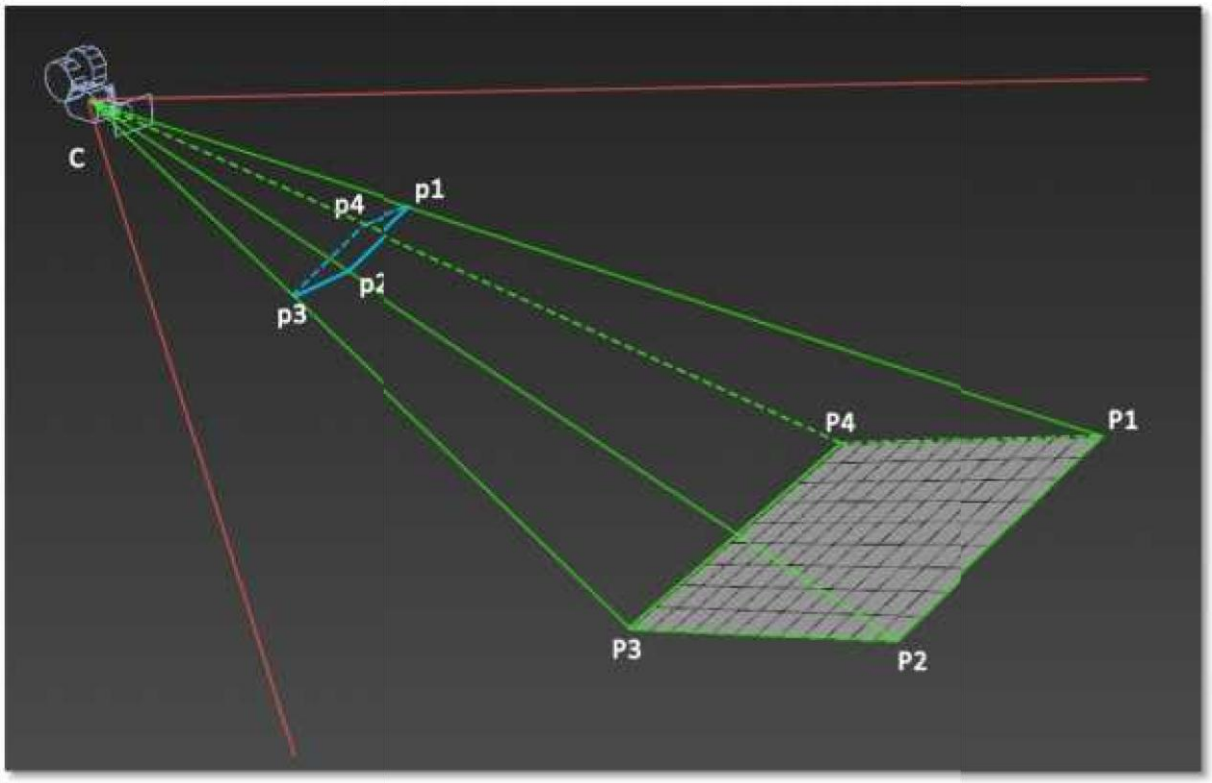


Figure III-37- Représentation du marqueur en 3d et sa projection sur le plan de l'image

Le **C** est utilisé pour dénoter le centre de la caméra. Les points de **P1** à **P4** sont les points 3D dans le système de coordonnées de l'espace, les points **p1** à **p4** sont leurs projections le plan de l'image de la caméra. Notre but est de trouver la transformation relative entre un marqueur dont la position est connue dans le monde 3D (**P1-P4**) et la caméra **C** en utilisant une matrice intrinsèque et des points de projetés sur le plan de l'image connus (**p1-p4**) . La question à se poser est d'où ramener les coordonnées de la position du marqueur dans l'espace 3D ? la réponse est simple, on les imagine. Puisque notre marqueur a toujours une forme carrée et du moment que tous ses sommets reposent sur le même plan (Z =0), on pourrait les définir de cette manière :

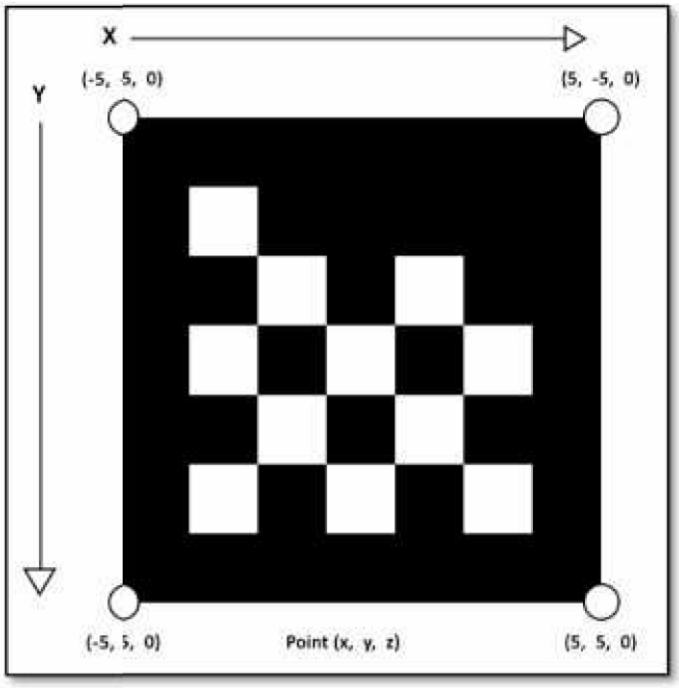


Figure III-38- Coordonnées 3D du marqueur

Nous disposons notre marqueur sur les plans X et Y, Z étant nul, et le centre du marqueur correspondent à l'origine (0, 0, 0). Pour trouver la position de la caméra avec les coordonnées 2D et 3D connues, nous utiliserons la fonction *solvePnP()* :

void solvePnP(const Mat& objectPoints, const Mat& imagePoints, const Mat& cameraMatrix, const Mat& distCoeffs, Mat& rvec, Mat& tvec, bool useExtrinsicGuess= false);

Le paramètre *objectPoints* est le tableau des points de l'objet dans l'espace de coordonnée 3D, nous lui passerons une liste de coordonnées de marqueurs dans l'espace 3D (un vecteur de 4 points).

Le paramètre *imagePoints* est un tableau qui correspond au points du marqueur sur l'image (la projection sur le plan de l'image), nous passerons dans celui ci une liste de coordonnées du marqueur en 2D.

- *cameraMatrix* : c'est la matrice 3x3 des paramètres intrinsèques.
- *distCoeffs* : c'est le vecteur qui contient les coefficients de distorsion.
- *rvec* : c'est le vecteur de rotation en sortie.
- *tvec* : c'est le vecteur de translation en sortie.

- *useExtrinsicGuess* : si il est vrai, alors la fonction va utiliser les vecteurs rvec et tvec fournis en tant qu'approximation initiale, et va les améliorer.

Les sorties de la fonction *solvePnP()* ne sont pas directement utilisables, il faut modifier certaines matrices et vecteurs, nous devons avoir en sortie une seule matrice qui contient toutes les informations nécessaires à la translation et la rotation, qu'on utilisera directement dans OpenGL, nous détaillerons dans ce qui suit sa construction :

```
Mat_<float> MarkerDet::posEstim(vector<Marker> markerDetected, vector<Point3f> obj,
    Mat_<float> intrinsic, Mat distCoef, vector<Mat_<float>>& glViewMatrix)
    {

        //Les déclarations
        Mat rvec, tvec;
        Mat Rvec;
        Mat_<float> Tvec;
        Marker tempMarker;
        Mat_<float> viewMatrix(4,4, CV_64F);
        vector<Mat> glViewMatrixTemp;

        Mat_<float> rodMat(3,3);
        Mat_<float> cvToGl =  Mat::zeros(4,4, CV_64F);
```

OpenGl ne gère pas les axes de la même manière qu'OpenCv, en effet les axes Y et Z sont inversés dans OpenGl, nous devons donc y remédier directement comme ceci :

```
        cvToGl.at<float>(0,0) = 1.0f;
        cvToGl.at<float>(1,1) = -1.0f;      //inversion de l'axe Y
        cvToGl.at<float>(2,2) = -1.0f;      //inversion de l'axe Z
        cvToGl.at<float>(3,3) = 1.0f;
        viewMatrix.at<float>(3,3) = 1.0f;
        glViewMatrixTemp.clear();


        for (int i =0; i< markerDetected.size(); i++)
        {
            tempMarker = markerDetected[i];
```

```
solvePnP(obj, tempMarker.Points, intrinsic, distCoef, rvec, tvec);
```

Afin d'obtenir une matrice de rotation de 3x3 à partir du vecteur de rotation, la fonction *Rodrigues()* sera utilisée. Cette fonction convertit la rotation représentée par un vecteur, et retourne sa matrice de rotation équivalente.

```
rvec.convertTo(Rvec, CV_32F);
tvec.convertTo(Tvec, CV_32F);
Rodrigues(Rvec, rodMat);
```

La matrice qu'on utilisera avec OpenGl devra contenir, comme cité plus haut, les information sur la rotation et la translation, pour cela nous allons joindre la matrice de rotation *rodMat* avec le vecteur de translation Tvec:

```
for (int row = 0; row <rodMat.rows; row++)
{
        for(int col = 0; col < rodMat.cols; col++)
        {
                viewMatrix.at<float>(row, col) = rodMat.at<float>( row, col);
        }
        viewMatrix.at<float>(row, 3) = Tvec.at<float>(row, 0);
}
```

Les matrices sont gérées différemment dans openGl par rapport à openCv, l'indexation est inversée, sur Opencv c'est les colonnes qui priment sur les lignes, contrairement à OpenGl, où c'est les lignes qui priment, nous devrons donc transposer la matrice :

```
Mat_<flaot> glViewTemp;
viewMatrix = cvToGl * viewMatrix;

transpose(viewMatrix, glViewTemp);
```

Il nous reste plus qu'à stocker la matrice dans un vecteur, ceci pour avoir chaque matrice de transformation reliée à son marqueur. N'oublions pas que l'application doit être multimarqueurs :

```
                glViewMatrix.push_back(glViewTemp);
        }
}
```

III-5 CALIBRATION DE LA CAMÉRA

Pendant la programmation de notre application, arrivé un moment (après la détection des marqueurs et décodage) lors de l'estimation de la position où nous avions besoin des paramètres internes de la caméra (matrice intrinsèque, et coefficients de distorsion), ceci nous a obligé à construire nous même notre propre programme de calibration de la caméra et pour ainsi permettre à l'application principale de fonctionner peu importe le système d'acquisition en place, ce programme nous donne en sortie un fichier XML ou sont stockées

les matrices de paramètres intrinsèques et les coefficients de distorsion, que nous importerons par la suite manuellement dans notre application principale.

Par convention, la calibration se fait à l'aide d'un échiquier (image imprimée, composée de carrés noirs et blancs, disposé à la manière d'un échiquier) dont la taille peut varier, selon le niveau de précision que l'on veut atteindre. Le programme cherche un échiquier sur l'image présentée, la taille de l'échiquier doit être initialement connue par le programme, dans notre cas nous allons utiliser un échiquier 9 x 6, précisons que cette taille représente le nombre de coins (vertical, et horizontal) présent sur l'échiquier.

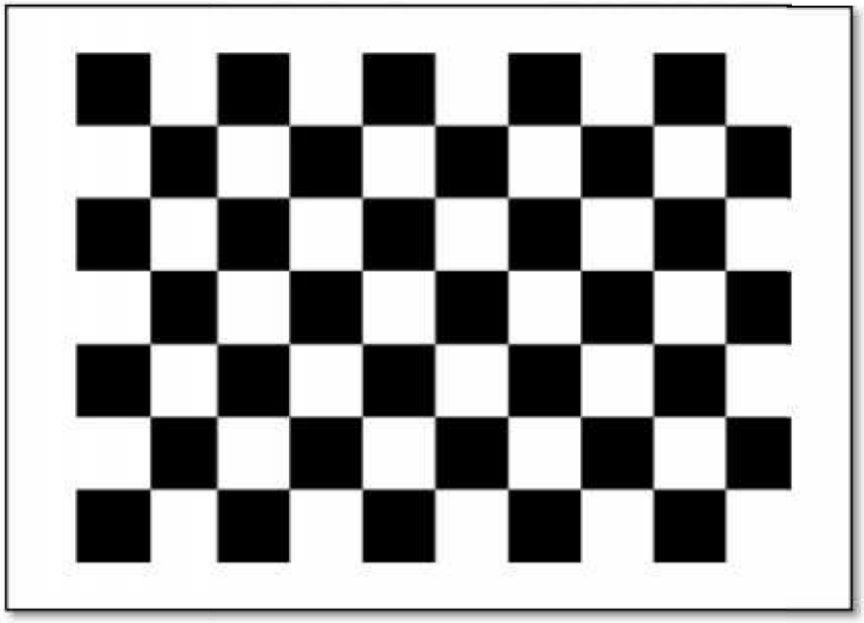


Figure III-39- Damier 9x6 utilisé pour la calibration

D'abord nous allons chercher l'échiquier sur l'image, dans notre cas nous utiliserons une caméra, la calibration se fera donc en temps réel. La fonction *findChessBoardCorners* va trouver le damier sur l'image :

```
//initialisation de la taille de l'échiquier
numH = 9;
numV = 6;

//conversion en niveau de gris
cvtColor(image, gray, CV_BGR2GRAY);

if (store ==false)
```

```
{
        found = findChessBoardCorners(image, Size(numH, numV), pointBuff,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FAST_CHECK | CV_CALIB_CB_NORMALIZE_IMAGE);
```

Par la suite nous elevons la précision de détection au niveau sous pixel en utilisant la fonction *cornerSubPix* comme ceci :

```
        if (found)
        {
                cornerSubPix(gray, pointBuff, Size(11,11), Size(-1,-1) , TermCriteria( CV_TERMCRIT_EPS
+ CV_TERMCRIT_ITER, 30, 0.1));
                //dessiner les points trouvé sur l'image
                drawChessBoardCorners(image, Size(numH, numV), pointBuff, found);
```

Après cela devons effectuer la calibration grâce à la fonction *calibrateCamera()*, pour cela nous avons associer un bouton à cette fonction :

```
Mat intrinsic = Mat(3, 3, CV_32FC1); Mat distCoeffs; vector<Mat> rvecs; vector<Mat>
tvecs; vector<vector<Point2f>> imagePoints; vector<vector<Point3f>> objectPoints;
//coordonnées 3d des coins de l'echiquier for( int i =0; i<boardSize; i++)
objectPoints.push_back(Point3f(i/numH, i%numH, 0.0f));

//Calibration
calibrateCamera(objectPoints, imagePoints, image.size(), intrinsic, distCoeffs, rvecs, tvecs);
```

puis nous stockons tout dans un fichier xml :

```
FileStorage fs(tempS, FileStorage::WRITE);
fs << "intrinsic" << intrinsic; fs
<< "distCoeffs<< distCoeffs; fs
<< "rvecs" << rvecs; fs <<
"tvecs" << tvecs;

fs.release();
```

Le fichier est prêt, il ne reste plus qu'à le charger dans l'application principale. Dans les prochains titres, nous nous intéresserons à la superposition de objets virtuels dans le contexte sémantique de l'environnement réel.

III-6 RENDU DE L'OBJET VIRTUEL 3D

Maintenant que nous avons trouvé les marqueurs sur l'image, ignoré les faux positifs, et que nous avons extrait leurs positions et orientations dans l'espace relativement à la caméra, il est temps de superposer des objets 3D. Comme cité plus haut, pour rendre la scène nous allons utiliser les fonctions d'OpenGL. La visualisation 3D est une partie importante de la réalité augmentée, OpenGL nous offrira toutes les bases nécessaires à la création d'un rendu de haute qualité.

Il y a beaucoup de moteurs de rendu 3D, commerciaux ou open source, tel que Unity, Unreal Engine, Ogre etc. Mais tous ces moteurs utilisent soit OpenGL ou DirectX pour passer leurs commandes à la carte graphique. DirectX est une API propriétaire, elle est supporté uniquement sur les plateformes Windows, c'est pour cette raison que nous utiliserons OpenGL qui contrairement à DirectX est compatible avec toutes les plateformes.

III-6-1 RENDU DE LA SCÈNE EN REALITÉ AUGMENTÉE

Pour pouvoir utiliser les fonctions d'OpenGL dans notre application, nous avons crée une classe **GLWidget.** Cette dernière hérite de la classe **QGLWidget** et ainsi toutes les méthodes présentes dans cette classe peuvent être utilisées avec la nouvelle (**GLWidget**) , nous allons cependant surcharger quelque méthodes de celle ci. L'entête de la classe GLWidget est le suivant :

```
class GLWidget : public QGLWidget
{
        Q_OBJECT
        public :
                explicit GLWidget (QWidget *parent = 0);
                void initializeGL();
                void paintGL();

                GLuint matToGl(Mat& mat, GLenum minFilter, GLenum magFilter,
GLenumwrapFilter);
                void drawAr();
                void drawBackGround();
}
```

La fonction paintGL() exécute le rendu de la scène AR sur l'objet GLWidget, elle effectue les étapes suivantes :

1. Nettoyer la scène.
2. Met en place la projection orthographique pour dessiner l'arrière plan.
3. Dessiner la dernière image reçue de la caméra sur l'arrière plan.
4. Met en place la projection en perspective en utilisant les paramètres intrinsèques de la caméra.

5. Pour chaque marqueur détecté, elle déplace le système de coordonnées vers la position du marqueur.
6. Rend l'objet 3D.
7. Affiche tout.

Cette fonction est appelé chaque fois qu'une image est prête à l'utilisation, cela arrive lorsqu'une nouvelle image est chargée dans la mémoire de la carte graphique et que toutes les étapes de détection du marqueur sont achevées.

III-6-2 MISE EN PLACE DE L'ARRIÈRE PLAN

Le dessin de l'arrière plan est une étape qui nous permet de créer l'effet de superposition de l'objet 3D avec l'image courante. D'abord on met au point la projection orthographique, ensuite on convertit l'image courante reçue d'OpenCV en texture OpenGL puis on la dessine sur un plan liée à la projection orthographique :

```
void GLWidget::drawBackGround()
{
        texture = matToGl(img, GL_NEAREST, GL_NEAREST, GL_CLAMP);
        GLfloat w = 640; GLfloat h = 480; const GLfloat squareVertices[] =
                //le plan qui va contenir la texture {

                0, 0 ,
                w, 0, 0
                , h,

                w, h
        };
        const GLfloat textureVertices[] =   //Coordonnées de la texture
        {

                1 , 0,
                1 , 1,
                0 , 0,
                0 ,  1
        };
        static const GLfloat proj[] =          //Paramètres de projection orthographique
```

```
{
        0, -2.0 f /w, 0, 0,
        -2.0 f/h, 0, 0, 0,
        0, 0 , 1, 0,
        1 , 1, 0,  1
};
//projection orthographique
glMatrixMode(GL_PROJECTION); glLoadMatrixf(proj);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Dessin du plan, et de la texture :

```
glDisable(GL_COLOR_MATERIAL);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textureID);

//dessin du plan glVertexPointer(2, GL_FLOAT, 0,
squareVertices);
glEnableClientState(GL_VERTEX_ARRAY);
//coordonnée de texture glTexCoordPointer(2,
GL_FLOAT, 0, textureVertices);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
}
```

III-6-3 MISE EN PLACE DE LA PERSPECTIVE

Nous allons régler la perspective de la scène 3d de façon à ce qu'elle coïncide avec la vue de notre caméra. D'abord nous devons effectuer quelques ajustements dans la matrice de projection dans OpenGL en utilisant la matrice intrinsèque (celle de la calibration). Sans ces ajustements nous obtiendrons une projection de la perspective erronée.

Une perspective erronée rend le dessin de l'objet d'une façon non naturelle, il apparait comme s'il flottait dans l'air, il ne fait pas parti du monde réel, une projection de la perspective doit être obligatoirement correcte pour obtenir un résultat correct.

```
float far = 100.0f; float near =
0.01f; float pi =
3.14159265358979323846;

float fx = intrinsic.at<float> (0, 0); float
fy = intrinsic.at<float>(1, 1); float fovy =
2 * atan(0.5 * h/ fy) * 180/ pi; float
aspect = (w * fy)/( h * fx);

glMatrixMode (GL_PREJECTION)
glLoadIndentity();
gluPerspective(fovy, aspect, near,
far);
```

Maintenant que nous avons chargé la matrice de perspective nous allons nous intéresser aux fonctions de dessin des objet3D

III-6-4 INCRUSTATION DE L'OBJET 3D

Dans notre application nous nous somme permit d'ajouter la fonctionnalité qui nous permet d'importer un objet 3D modélisé dans un tiers logiciel de création d'images de synthèse tel que, 3D Studio Max, Maya, Zbrush etc... nous en discuterons plus en détails dans le titre consacré aux fonctionnalités ajoutées.

Voici le code qui nous permet d'ajouter un objet standard (cube, boite, sphère, cône etc...) :

```
void GLWidget::drawAr()
{
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glPushMatrix;
        Mat_<float> matTemp = Mat::zeros(4, 4, CV_64F);
        for (int i =0; i<glViewMatrix.size(); i++)
```

```
        {
            if (codeIndex.Size() != 0)
            {
                matTemp = glViewMatrix[i];
                glMatrixMode(GL_MODELVIEW);
                glLoadIdentity();
                glLoadMatrixf(&matTemp.at<float>(0, 0));
                glColor3f(1, 1, 1);
                gluSolideTeapot(0.6);
            }
        }
        glPopMatrix();
    }
```

Le résultat obtenu avec le module d'importation d'objet 3D est le suivant, l'objet en question est une chaise :

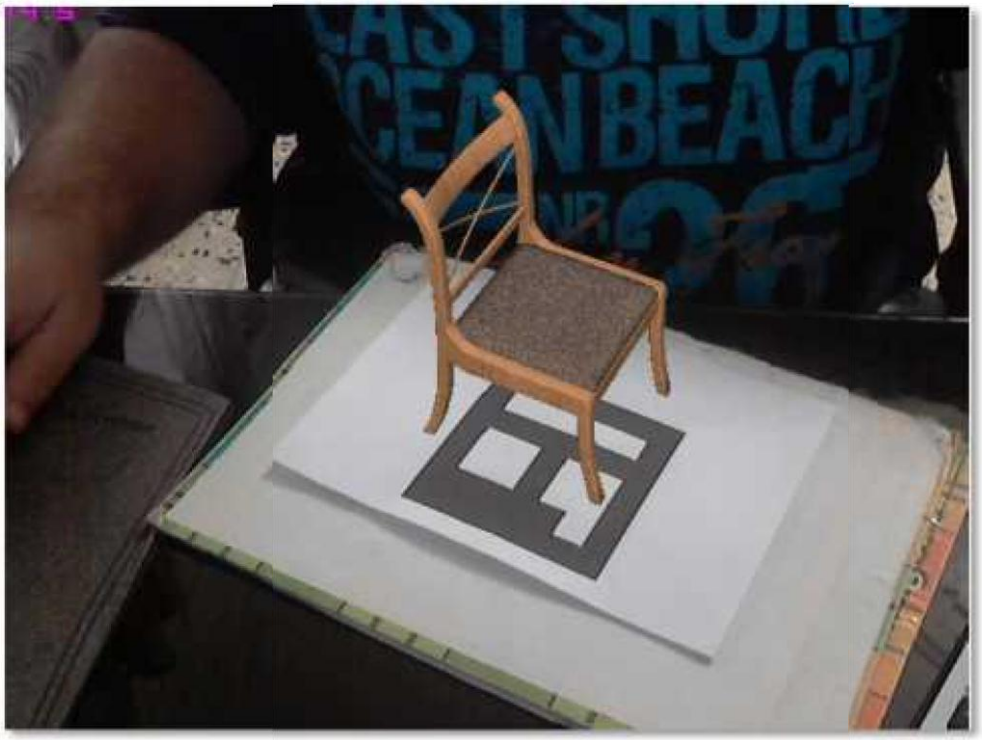


Figure III-40- Objet 3d inséré dans la scène réelle (la chaise)

III-7 ASSERVISSEMENT D U ROBOT

Dans cette partie, nous allons parler de l'asservissement du robot. D'abord nous programmerons le microcontrôleur à l'aide de l'interface de programmation incluses dans les drivers de la carte **Arduino** cette programmation nécessite une certaine base en langage C, le fait que la programmation se fasse avec le langage C va nous simplifier d'une manière drastique la programmation du microcontrôleur.

Figure III-41- Prise instantanée de l'environnement de développement pour Arduino

III-7-1 DANS LE MICROCONTRÔLEUR

L'algorithme introduit dans le microcontrôleur est plutôt simple, du moment que nous avons deux moteur, nous déclarerons donc deux variable *Stepper* puis nous initialiserons la liaison série dans la fonction *setup()* :

```
//Les 4 dernier arguments des déclaration, sont les sorties du microcontrôleur
Stepper myStepper1(stepsPerRevolution, 10, 11, 12, 13);
Stepper myStepper2( stepsPerRevolution, 6, 7, 8, 9);
// Fonction setup()
void setup()

{
        Serial.begin(9600);
}
```

Nous passon à la fonction principale qui agit comme une boucle, elle s'appelle d'ailleurs *loop()* qui signifie boucle en français. La réception des donnée et l'interprétation se passe à l'intérieur de cette fonction :

```
void loop()
{
        if (Serial.available() >0)
        {
                incom = Serial.read();
                switch (incom)

                {
                        case 'r' : myStepper1.step(1); break;
                        case 'l' : myStepper1.step(-1) ; break;
                        case 'u' : myStepper2.step(1); break;

                           case 'd' : myStepper2.step(-1) ; break;
                }
        }
}
```

le microcontrôleur attend un caractère envoyé à travers le port série, il y a quatre caractères, si le caractère est '**r**' qui représente (**RIGHT**) alors il faut faire un pas vers la droite, si le caractère est égale à '**l**' pour (**LEFT**) le moteur fera un pas vers la gauche, le même principe est appliqué au second moteur, si le caractère envoyé est '**u**' pour (**UP**) alors faire un pas vers le haut, enfin si le caractère est '**d**' (**DOWN**) alors faire un pas vers le bas. L'application fonctionnera à environ 15 image par seconde, donc les moteurs effectueront un pas 15 fois par secondes, ce qui nous garantira une bonne fluidité de mouvement.

III-7-2 DANS LE PROGRAMME PRINCIPAL

Le principe de l'asservissement est simple, tout d'abord, le branchement de la **Arduino** établi automatiquement une connexion en série, nous allons donc paramétrer une liaison série dans notre programme, par la suite, lorsqu'un marqueur est détecté, on récupère les coordonnées de son centre sur l'image, et on soumet ces coordonnées à des testes. D'abord l'initialisation :

```
serial.setPortName("COM5");
serial.setBaudRate(QSerialPort::Baud9600);
serial.setDataBits(QSerialPort::Data8);
serial.setParity(QSerialPort::NoParity);
serial.setStopBits(QSerialPort::OneStop);
serial.setFlowControl(QSerialPort::NoFlowControl);

serial.open(QIODevice::WriteOnly);
```

Puis dans la fonction principale du programme *mainFct()* nous testons le centre du marqueur, si celui ci est dans la moitié gauche de l'image, alors écrire '**l**' sur le port série, s'il est dans la moitié droite alors écrire '**r**', en même temps si le centre est dans la moitié supérieure on écrit '**u**' sinon on écrit '**d**'**.** Il ne faut pas oublier de laisser une marge au centre de l'image ou les moteurs ne doivent pas bouger, on peut dire que dans cette zone le marqueur est bien ciblé ! Le code qui illustre cette idée est le suivant :

```
if (markerEstablished.size() != 0)
{
        Marker mWrite = markerEstablished[0];
        vector<float> tempWrite =
```

```
writeMot(mWrite); float x = tempWrite[0];
float y = tempWrite[1];

serial.clear(); if (x< 280)
serial.write("r"); else if
(x>360) serial.write("l"); if
(y<220) serial.write("u");

if (y>260) serial.write("d");
}
```

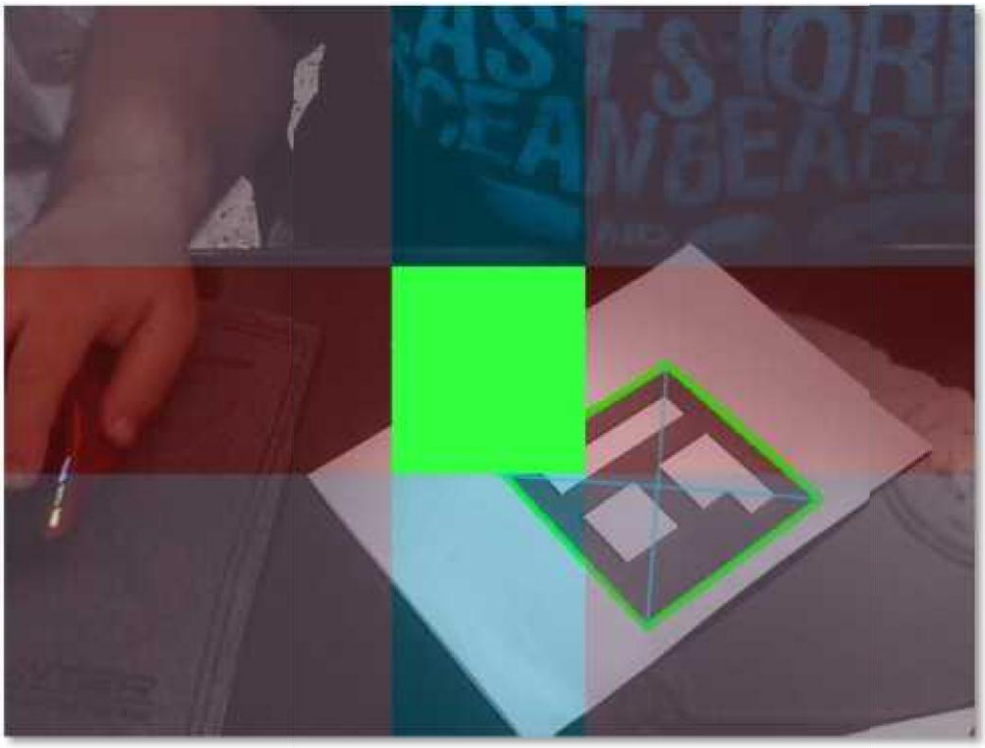


Figure III-42- Position du marqueur sur l'image

Le rectangle vert sur l'image représente la marge d'inactivité, où les moteurs ne bougent plus, nous appellerons cette zones "Cible Acquise".

III-8 FONCTIONNALITÉS S UPPLÉMENTAIRES

Notre application a beaucoup de fonctionnalités supplémentaires, nous discuterons ici les plus importante d'entre elles :

- L'interface utilisateur.
- Multi marqueurs.
- Chargement automatique des marqueurs connus.
- Importation d'objets 3D.
- Aller plus loin.
- Future de la Réalité augmentée.

II-8-1 L'INTERFACE

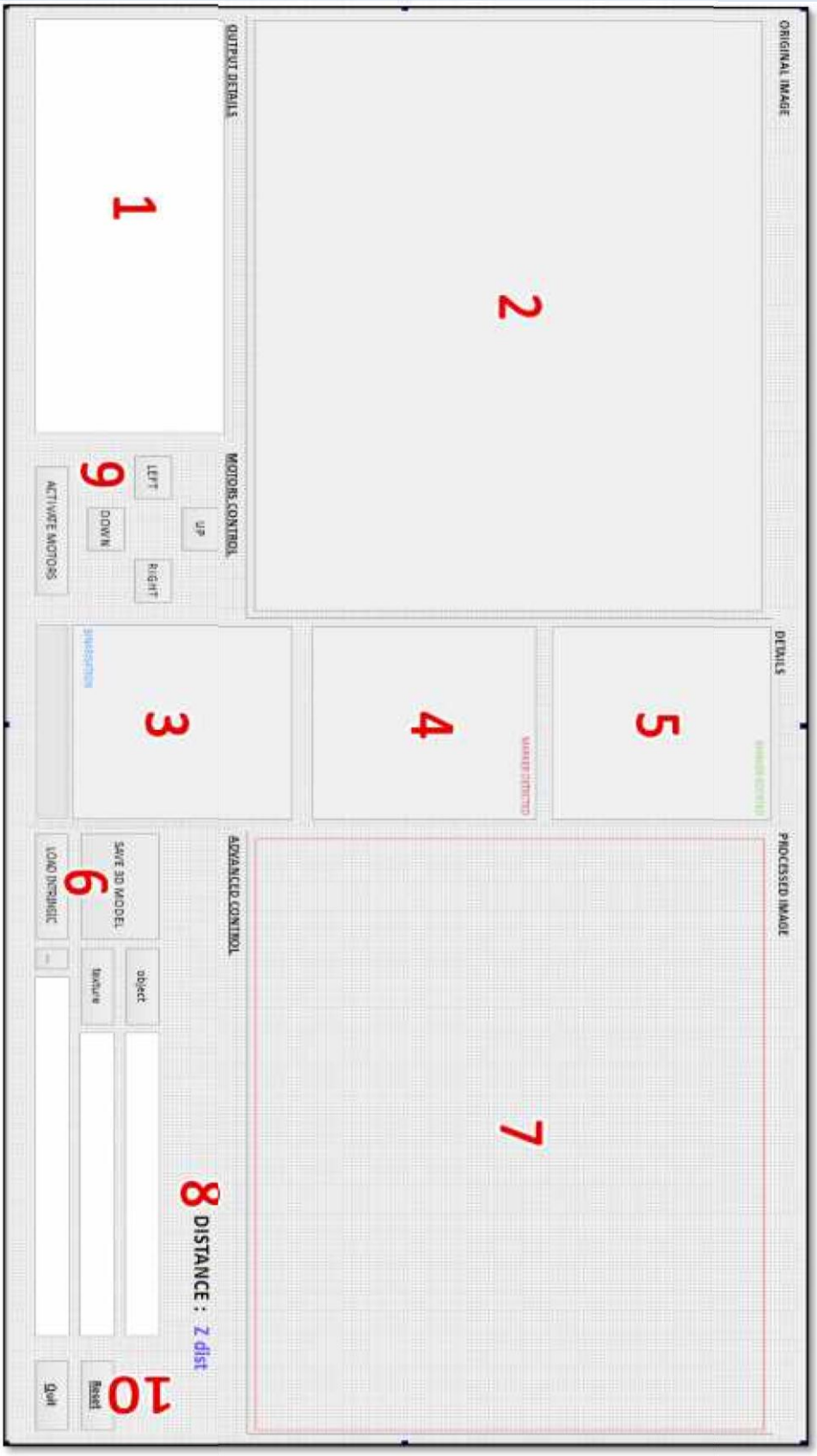


Figure III-43- Interface du programme

1. **Output details :** c'est à travers cette zone de texte que l'application communiquera avec l'utilisateur. Elle indique s'il y a un marqueur présent sur l'image, si oui le quel, et toute sorte d'autres informations.
2. **Original Image :** c'est l'image courante originale reçu de la caméra, les contours des marqueur y seront dessinés.
3. **Binarisation :** nous montre l'image courante miniaturisée et binarisée.
4. **Marker detected :** nous montre l'étalage de l'image du marqueur, perspective annulée.
5. **Marker Rotated :** ici on verra le marqueur détecté dans (4) soumit à la rotation et binarisé.
6. **Save 3d Model / Load Intrinsic :** le premier bouton est l'une des fonctionnalité supplémentaire de notre application elle permet, lorsqu'on a un marqueur inconnu, de le sauvegarder en lui associant un model 3d construit avec une application tierce, on en reparlera plus bas. Le deuxième bouton concerne le chargement du fichier XML de calibration (chargement des paramètres intrinsèques et les coefficients de distorsion).
7. **Processed Image :** la scène OpenGl sera dessinée sur cette fenêtre, elle contiendra l'image courante plus la superposition de l'objet 3D ou virtuel.
8. **Distance :** comme son nom l'indique, le texte en bleu va nous indiquer la distance à la quelle se trouve notre cible.
9. **Motors Control :** Le bouton *Activate Motors* activera la liaison série avec la carte Arduino qui nous permettra d'activer les moteurs, en d'autre terme le bouton active le tracking de l'objet par le robot. Les autres boutons de direction nous permette de contrôler manuellement le robot, peut servir en cas de perte de la cible.
10. **Reset / Quit :** comme leurs noms l'indique, le bouton reset nous permet de relancer l'application, ainsi lorsque l'on associera un objet 3d à un marqueur inconnu, il nous faudra relancer l'application. Le bouton Quit quand lui ne fait que quitter

III-8-2 MULTI MARQUEURS

Une fois le but du tracking atteint, il faut penser à rendre l'application multi marqueurs qui veut dire capable de détecter et interpréter autant de marqueurs qu'on présente à la caméra. Pour faire cela la solution est simple, stocker chaque élément relié au marqueur dans un vecteur propre. Ainsi on initialise une variable **vector** suivit du type associé pour chaque composant, ainsi :

- on initialisera deux vecteur reservés au marqueur connus et inconnus :

```
vector<Marker> establishedMarkers;
vector<Marker> unknownMarkers;
```

- On estimera la position de chaque marqueur détecté (establishedMarker), dans une boucle, puis on stockera la matrice résultante de l'estimation dans un vecteur qui les contiendra :

```
vector<Mat_<float>>& glViewMatrixList;
Mat_<float> glViewMatrix;

for (size_t i = 0; i< establishedMarkers.size(); i++)
{
        glViewMatrix = // Estimation de la position posEstim();
        glViewMatrixList.push_back(glViewMatrix);
}
```

- Dans OpenGL, en utilisant une boucle, on pourra afficher chaque objet sur le marqueur assigné et puis charger la matrice pour chaque objet créé, de cette façon :

```
//La perspective reste la même peut importe le marqueur
// C'est la vue qui change
Mat_<float> matTemp = Mat::zeros(4, 4, CV_64F);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); glPushMatrix()
;

for(int i =0; i< glViewMatrix.size(); i++)
```

```
{
        matTemp = glViewMatrix[i];
        glMatrixMode(GL_MODELVIEW);

        glLoadIdentity();
        glLoadMatrixf(&matTemp.at<float>(0,0));

        drawObj();
}
```


Après ces quelques modifications, nous obtenons cela :

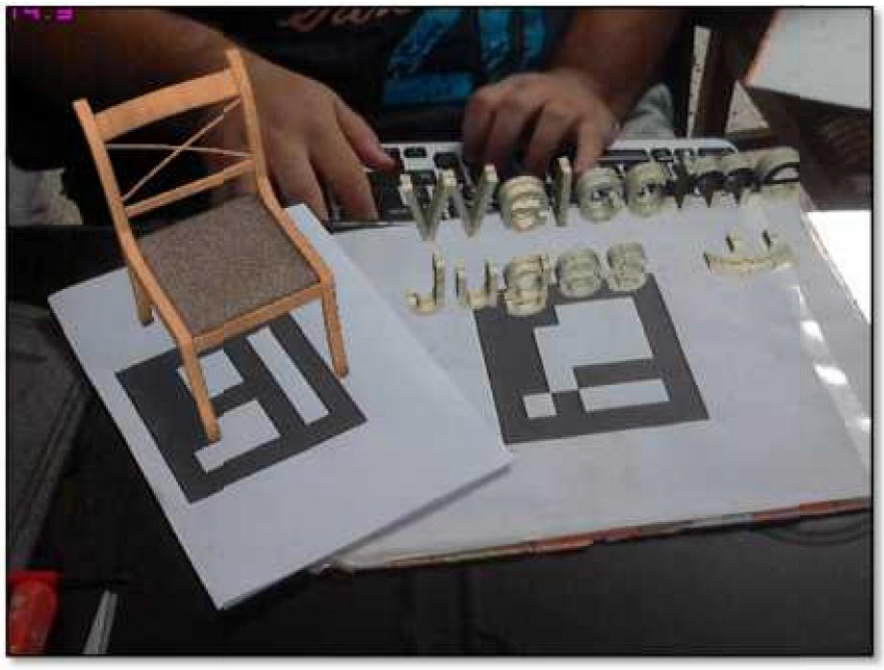


Figure III-44- Plusieurs marqueurs sur la même image

Ici nous voyons qu'il y a deux marqueurs et deux objets superposés au dessus de chacun d'eux, le premier, une chaise, le second un texte en 3d "welcome juges :)" .

III-8-3 CHARGEMENT AUTOMATIQUE DES MARQUEURS CONNU S

Le principe encore une fois ici est simple, nous stockons des marqueurs préalablement reconnus (on peut le faire manuellement aussi). On créer d'abord une structure qu'on appelera *markStr* définie de cette manière :

```
struct markStr
{
        int index ;
```

```
        vector<vector<int>>
        code; string objPath;
        string texPath; bool
        texture;

        bool object;
}
```

Cette structure contient, un indexe qui nous servira d'identifiant pour marqueurs, d'une matrice d'entier, qui représentera le code du marqueur d'identifiant "index", deux chaines de caractères, la première pour l'adresse du fichier 3d à charger avec l'objet, la seconde pour

l'adresse de la texture à associer à l'objet, les deux booléennes détermine s'il y a texture et objet ou l'un sans l'autre.

Les valeurs de cette structure pour chaque marqueur est stocké dans un fichier, qui est chargé au lancement de l'application dans un vecteur. Nous n'avons plus qu'à comparer les codes des marqueurs présent sur l'image avec ceux qui sont dans la mémoire (chargés à partir du fichier) :

```
//Fonction pour la lecture du fichier au démarrage de l'application.
void MainWindow::readFile(string path, vector<markStr>& list)

{       //....
    file.seekg(0, std::ios::beg); cout <<
    strVec.size()<< endl; for (int j =0;
    j< strVec.size()-1 ; j ++)

    {
            markerRead.code.clear(); markerRead.objPath="";
            markerRead.texPath=""; file >> markerRead.index>>
            markerRead.object >> markerRead.texture ;

            for (int i = 0; i< 7; i++)
            {
                vector<int> temp ;
                for(int j = 0; j< 7; j++)
                {
```

```
                int m;
                file >>
                m;
                temp.push_back(m);
            }
            markerRead.code.push_back(temp);
        }
        if (markerRead.object == true)
            file >> markerRead.objPath;
        if (markerRead.texture == true)
            file >>  markerRead.texPath ;
        list.push_back(markerRead);
    }
    file.close();
}
```


Ainsi les marqueurs connus de notre application, sont chargé automatiquement au chargement des paramètres intrinsèques de la caméra. Nous sommes permis aussi d'ajouter un autre module qui permet de lier un marqueurs inconnu à un objet 3D et une texture et enregistrer le tout dans le même fichier, ainsi lors du prochain démarrage de l'application le marqueur qui était avant inconnu sera connu (d'où le bouton **RESET**) et le model 3d lié à lui sera chargé dans la scène.

III-8-4 IMPORTATION D'OBJETS 3 D

On ne peut pas arriver à ce stade de l'application sans vouloir y intégrer des objet 3d modélisé par nous même, en effet cela rendrait l'application sans grand intérêt par la suite. C'est pour cela que nous avons créer la classe **ObjImporter** qui s'occupe de charger un fichier 3d avec le format standard "***.obj***" . Avant de décrire la classe **ObjImporter** nous allons décortiquer la structure d'un fichier ".obj" . Voici une parcelle du contenu d'un fichier ".obj" :

```
# 3ds Max Wavefront OBJ Exporter v0.97b - ( c)2007 guruware
# File Created: 25.10.2014 20:21:01

#
```

 # object Text001

#

v -0.7251 -0.1000 0.9432 v

-0.7925 -0.1000 0.6991 v -

0.7925 0.0000 0.6991 v -

0.7251 -0.0000 0.9432 v -

0.7152 -0.1000 0.9809 v -

0.7152 -0.0000 0.9809

...

vn 0.9639 0.0000 -0.2662

vn 0.9678 0.0000 -0.2519

vn -0.9641 0.0000 -0.2654

vn 0.0000 0.0000 -1.0000

vn 0.9649 0.0000 -0.2625

vn 0.0000 -0.0000 1.0000

...

vt 0.9820 0.4000 0.7844

vt 0.9969 0.6000 0.7919

vt 0.9969 0.4000 0.7919

vt 1.0110 0.6000 0.8005

vt 1.0110 0.4000 0.8005

vt 1.0242 0.6000 0.8100

...

f 1/1/1 2/2/1 3/3/1 f

3/3/1 4/4/1 1/1/1 f

5/5/2 1/1/2 4/4/2 f

4/4/2 6/6/2 5/5/2 f

7/7/3 5/8/3 6/9/3 f

6/9/3 8/10/3 7/7/3

...

Nous remarquons qu'il y a 5 types de début de ligne, en fait c'est des identificateurs. le symbole "**#**" dénote un commentaire, nous allons donc ignorer ces lignes. Le second identificateur est le "**v**" qui dénote quand à lui les coordonnées des sommets dans l'espace 3D, chaque ligne représente un sommet. L'identificateur "**vn**" représente la valeur de normale des sommets, la normale étant l'orientation du sommet, chaque sommet est orienté, sans cela on aurait un résultat anormal de la forme de l'objet. Le "**vt**" représente quant à lui les coordonnées de textures de l'objet, en d'autre terme comment la texture est appliquée sur l'objet. Le dernier identificateur symbolise la face, en effet, étant la face triangulaire, la ligne est composée de trois rangés séparées par des espaces "**f a/b/c d/e/f g/h/i**" dans la première rangé il y a l'index de la coordonnée du premier sommet "**a**" suivit d'un slash "**/**" puis "**b"** qui est l'index de la coordonnée de texture à utiliser, suivit d'un autre "**/**" puis de l'index de la normale à utiliser "**c**". Même chose pour la seconde rangé, sauf qu'il s'agit du deuxième sommet constituant cette face, jusqu'à la troisième rangé qui de la même manière représente le troisième sommet de la face.

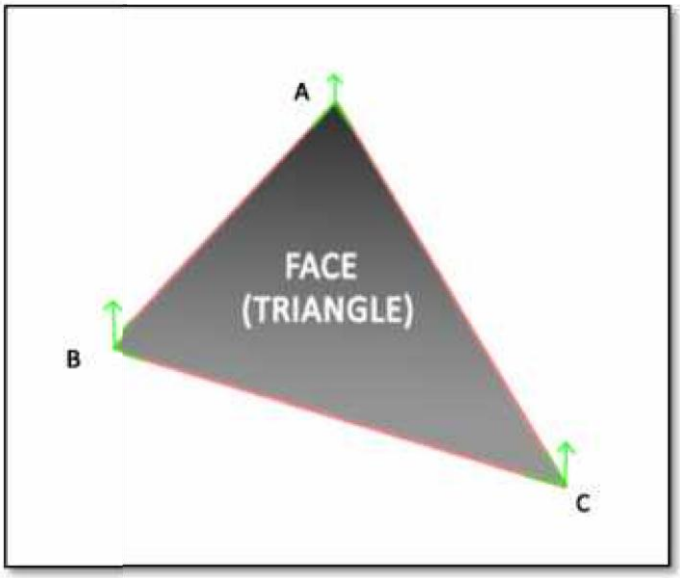


Figure III-45- Une face en 3d

Ici "**A**" représente le premier sommet constituant la face ou le triangle, il est constitué de ses coordonnée par rapport à l'origine de l'espace 3d, aussi des coordonnées de textures, et en vert de la normale du sommet ou sa direction, même chose pour le second sommet "**B**" et le troisième "**C**", c'est cela que représente les lignes commençantes avec la lettre "**f**". L'entête de la class **ObjImporter** est la suivante :

```
class ObjImporter
{
public:
```

```
    ObjImporter();
    GLuint readFile(string path, string
    imgPath); void readFile(string path); int
    displayObject();
    int displayObject(GLuint textu);

private:
    std::vector<Vertex>  vertices;
    std::vector<Vertex> normals;
    std::vector<Vertex> textures;
    std::vector<Faces>  face;
    std::vector<string> coord;
} ;
```

Les vecteurs déclarés ci dessus, contiendront toutes les données de chaque ligne, pas à pas pendant la lecture du fichier :

```
ifstream file;
file.open(path);
for(int i = 0; i < coord.size(); i++)
{
        stri = coord[i]; if(stri[0] == "#")
        continue; else if ( (stri[0] == 'v' ) &&
        (stri[1] ==' '))
        {       //....
                sscanf_s(stri.c_str(), "v %f %f %f", &tmpx, &tmpy, &tmpz);
                vertices.push_back(Vertex(tmpx, tmpy, tmpz));
        } else if (// le début de la chaine égale "vn" ){
                sscanf_s(stri.c_str(), "vn %f %f %f", &tmpx, &tmpy, &tmpz);
                normals.push_back(Vertex(tmpx, tmpy, tmpz));
        } else if (//le début de la chaine égale "vt")  {
```



```
                sscanf_s(stri.c_str(), "vt %f %f %f", &tmpx, &tmpy, &tmpz);
                textures.push_back(Vertex(tmpx, tmpy, tmpz));
```

```
        } else if (//le début de la chaine égale "f " )
        {
                int fa1, fa2, fa3, fn1, fn2, fn3, tx1, tx2, tx3;
                sscanf_s(stri.c_str(), "f %d/%d/%d %d/%d/%d %d/%d/%d", &fa1, &tx1, &fn1, &fa2,
&tx2, &fn2, &fa3, &tx3, &fn3) ;
                f = Faces(fa1, tx1, fn1, fa2, tx2, fn2, fa3, tx3, fn3);
                face.push_back(f);
        }
}
```

De cette façon nous arrivons à stocker toutes les coordonnées et tous les indexes dans la mémoire, l'objet est prêt à être dessiné, l'affichage se fera dans une fonction appelé *displayObject()* comme ceci :

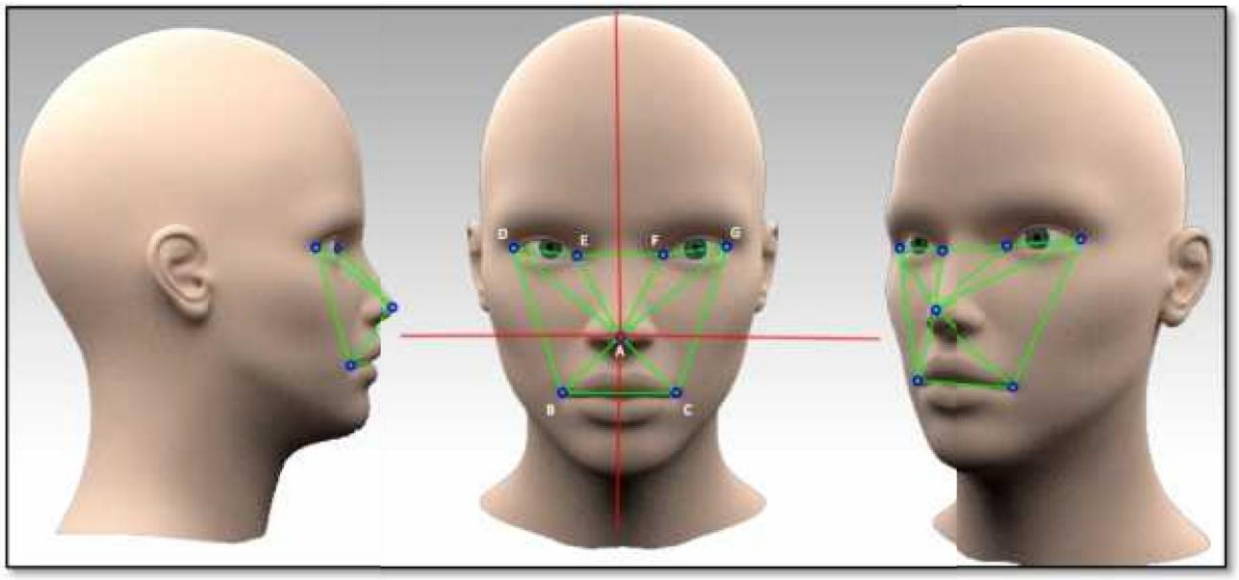
```
int ObjImporter::displayObject(GLuint textu)
{
        int num = glGenLists(1);
glNewList(num, GL_COMPILE);
for(int i =0; i< face.size(); i++)
{
        faceTemp= face[i]; vertexTemp1 = vertices[faceTemp.face[0] - 1];       // indexation
        commence de 1 donc (-1) vertexTemp2 = vertices[faceTemp.face[1]-1] ; vertexTemp3 =
        vertices[faceTemp.face[2]-1] ; normalTemp1 =normals[faceTemp.facenum[0] - 1] ;

        ...
        textTemp1 = textures[faceTemp.txt[0]-1] ;
        ...
        glBegin(GL_TRIANGLES);
        {
                glNormal3f(normalTemp1.x, normalTemp1.y, normalTemp1.z);
                glTexCoord3f(textTemp1.x, textTemp1.y, textTemp1.z);
                glVertex3f(vertexTemp1.x, vertexTemp1.y, vertexTemp1.z);

                //... même chose pour les deux autres sommets
        }
        glEnd();
} glEndList();
```

III-8-5 ALLER PLUS L OIN

l'image qui suit :



Nous allons discuter ici d'une tout autre manière d'utiliser notre application, nous allons utiliser le workflow décrit plus haut pour essayer d'estimer la position et l'orientation du visage, évidement la taille d'un visage à un autre peut varier, nous allons considérer la moyenne de la taille d'un visage d'être humain. Bien entendu cette partie sera purement théorique, et sa fonctionnalité n'est point présente dans notre application, ce qui n'aurait pas été bien compliqué à mettre en œuvre. Le résultat obtenu serait bien évidemment approximatif (du à la taille du visage qui varie).

Dans le cas d'une application RA standard, la cible est le marqueur, qui est un polygone à 4 sommets. Dans le cas d'un visage, c'est un peu plus complexe, en effet, la forme d'un visage n'est pas définit, c'est une forme organique variable. Heureusement il contient assez de composantes connues pour pouvoir y détecter des points. On pourrait détecter les yeux, le nez, et la bouche, ces composantes son facilement détectables à l'aide des algorithmes basés sur la classification tel que les cascades de Haar. On pourrait peaufiner la détection en considérant les coins des yeux, et les coins de la bouche avec la position du nez, comme dans

Figure III-46- Détection des points clé d'un visage et leurs coordonnées

Les cercles bleus représentent les coins de notre formes, on se réfèrent aux marqueurs, cela représenterais leurs sommets. On peut dire que la forme générale constitue une sorte de pyramide avec une base à 6 cotés. Les traits rouges, représentent l'origine (0 , 0, 0), et donc on peut assigner des coordonnées à chaque sommet en calculant la distance moyenne entre chaque point d'un visage. Ainsi on obtient ceci :

A. (0, 0, 0).

B. (-3, -3, -2).
C. (3, -3, -2).
D. (-4, 5, -3.5, -3.5).
E. (-2.5, 3.8, -3).
F. (2.5 , 3.8, -3).
G. (4, 5, -3.5,-3.5).

Maintenant il ne reste plus qu'à soumettre les points détectés à l'algorithme d'Estimation de la position, pour ainsi obtenir la distance du visage, et bien entendu son orientation. De la même manière que dans notre programme nous pourrions y superposer des objets 3d tels que des lunettes par exemple, ou des chapeau, ce qui nous permettrais de créer un système d'essayage en ligne, si nous compilions l'application pour un site web.

Figure III-47- Incrustation d'objet 3 D sur le visage

III-9 FUTURE DE LA RÉALI TÉ AUGMENTÉE

Avec le développement de la technologie embarquée et les systèmes d'acquisitions qui se font en miniature, on pourrait inclure des système de réalités augmentées dans des gadgets tel que les google glasses, on pourrait alors définir et répertorier chaque chose qu'on voit à

travers ces lunettes, cela rendrais l'immersion des objets 3 D complète, nous n'aurions plus besoin d'écran et de marqueurs pour visualiser le monde augmenté.
La réalité augmentée intervient dans beaucoup de domaines, si ce n'est qu'elle pourrait intervenir dans tous les domaines. Il y a tout d'abord le domaine militaire, le spatial, médical, éducatif, commercial et plein d'autres encore.

III-9-1 VISEUR TÊTE HAUTE

Les viseurs tête haute VTH ou HUD pour Head-Up Display en anglais, est tout système d'affichage sur un support transparent, qui présente des données sans que l'utilisateur ne voit ailleurs que son point de vue habituel. Initialement développé pour l'aviation militaire, les VTHs sont maintenant utilisés dans les avions de ligne, les automobiles, les jeux vidéo et d'autres applications.

Aviation

Dans les systèmes de visée des avions, ce dernier reçoit les informations émit par les capteurs de l'appareil (navigation, gyroscope..), il affiche la vitesse du vent, l'altitude, la ligne d'horizon, ceci dans la majorité des avions qu'il soit civiles ou militaires. En ce qui concerne les systèmes d'avions militaires voici ce que le VTH affiche (23) :

- Désignation de cible (TD pour target designator en anglais) - qui place une queue sur une cible aérienne ou au sol (qui est dérivé du radar ou des données du système de navigation).
- - Fermeture de vitesse cible (en anlgais closing velocity with target).
- Portée (Range) - Distance de la cible ou du point de route.
- Région d'acceptation de lancement (LAR) - Montre quand un missile air-air ou air-sol peut être lancé pour atteindre la cible avec succès.
- Capteur de ligne de mire - Montre vers où un capteur pointe.
- État de l'armement - Inclue le type et le nombre d'armes restantes sélectionnées, ou armées.

Figure III-48- VTH d'un avion de chasse (24)

Le VTH est équipé sur le casque du pilote, qui est réfléchit par la suite sur un miroir transparent qui augmente l'affichage des données par rapport à la vue, ceci à pour but de ne pas fatiguer les yeux du pilote, car ce dernier focalise la plupart du temps sa vue sur l'horizon ou l'infini, tandis que le viseur est à 20 centimètre de lui :

Figure III-49- Fixation du viseur sur le casque (25)

Automobile

En 2012 Pioneer Corporation a réussi à introduire un système de navigation qui projette un VTH sur le pare brise qui présente l'animation des conditions d'en face de la

voiture (26). Voila à quoi ressemblerais le futur de cette technologie en image :

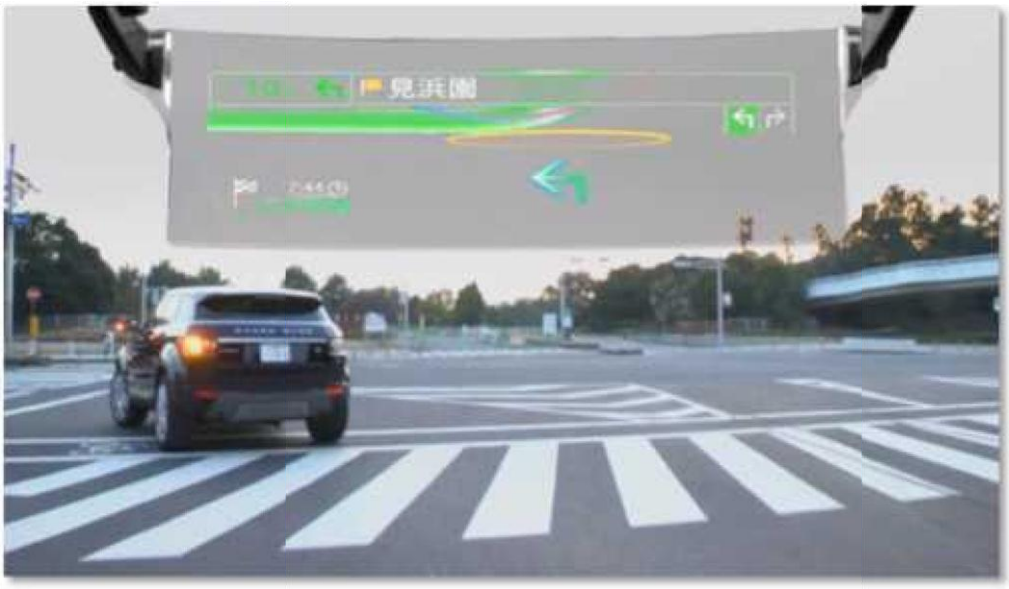


Figure III-50- Réalité augmentée sur le pare brise d'une voiture (26)

On pourrait afficher les informations de navigation telle que la direction à prendre ou la vitesse actuelle, ou des information beaucoup plus cruciales comme la distance de la voiture d'en face, cela nous permettrait d'éviter des accidents de la route liés au freinage par exemple.

III-9-2 DANS LA MED ECINE

Dans le domaine médicale, la RA a déjà permis la superposition des données d'imagerie ultrasons directement sur le corps du patient. Le médecin peut ainsi visualiser dans un même espace ses actions et leurs conséquences. En médecine, des rayons infrarouges sont projetés très proche de la peau pour qu'ils y soient absorbés, c'est là qu'intervient une caméra vidéo afin de capturer la lumière n'ayant pas été absorbée pour qu'un ordinateur déduise les positions des veines. Ces positions sont indiquées grâce à un rétroprojecteur (toutes les veines allant jusqu'à un centimètre de profondeur de la peau

sont visibles). Cette technologie baptisée VeinViewer a été développée par LuminetX (27).

Figure III-51- Réalité augementée dans la médecine (27)

III-9-3 DANS L'ÉDUC ATION

Les vertus pédagogiques de la RA ne sont plus à prouver, et c'est au tour des étudiants en médecine d'en profiter. C'est derniers n'ont pas toujours les moyens d'apprendre à traiter des patients en situation réelle de blessures graves. En effet, l'éthique médicale ne leur permet pas d'examiner notamment les victimes de crime, celles-ci devant impérativement être prises en charge par des médecins titulaires. Ainsi PRLI MedAppLab, groupe de recherche allemand spécialisé dans les appareils mobiles et applications en contexte médical, mené par Urs Albrecht, a développé l'application mARble. Les étudiants peuvent s'exercer en situation réelle à l'aide de leurs smartphones, visualiser des contenus

multimédia et répondre à des tests pratiques.

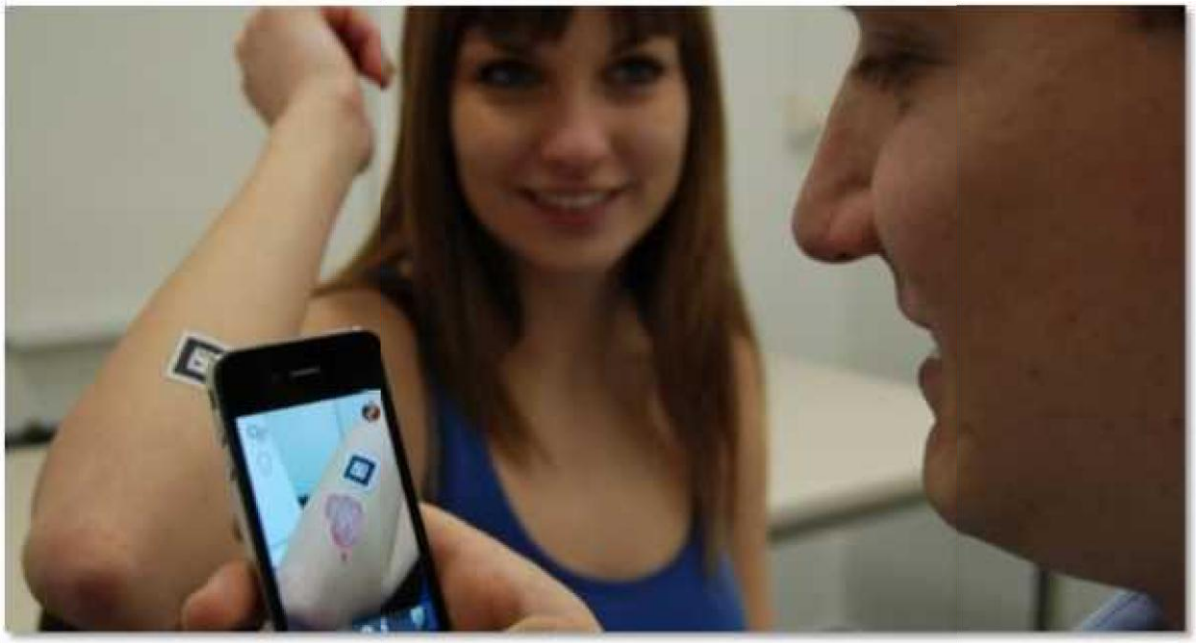


Figure III-52- Réalité augmentée dans l'éducation (28)

Les étudiants munis de leurs smartphones, prennent en photo le patient préalablement équipé d'une étiquette (marqueur) et visualisent la blessure via l'écran de leur téléphone. Ils peuvent ainsi examiner le patient directement en salle de classe. L'application intègre également du contenu interactif et multimédia sous forme de textes, ou de quizz pour tester les connaissances acquises durant le traitement du patient fictif.

La réalité augmentée a l'avantage de rendre l'expérience des étudiants plus interactive. Urs Albrecht a souhaité créer un environnement d'apprentissage pour les étudiants qui n'allait pas à l'encontre de l'éthique. Il a d'ailleurs mené une étude pour tester l'efficacité de cette application sur un petit échantillon de dix étudiants en troisième année. Un groupe de six étudiants était équipé de l'application sur un iPad et l'autre groupe étudiait sur des livres scolaires. Après trente minutes d'étude, Urs Albrecht en a conclu que les étudiants munis de l'application mobile avaient une meilleure connaissance du contenu, en se basant sur leur état émotionnel et les connaissances acquises (28).

III-10 RÉSULTATS

Après avoir fini de développer l'application nous passons aux testes, la seule partie qui a été sujette à des fluctuations dans notre application est bien la partie de détection du marqueur, en effet cette dernière est vulnérable à l'absence des bonnes conditions d'éclairage et la variation de celui ci. Aussi, et dépendamment de la résolution de la caméra, nous avons remarqué que nous perdions la détection du marqueur proportionnellement à sa distance mais aussi à son orientation, plus le marqueur est parallèle à l'axe de l'image plus il est mal détecté, c'est à dire que plus le marqueur était loin plus sa détection était difficile. Nous allons

illustrer dans cette dernière partie du dernier chapitre quelques testes effectués et les rapporter sous forme de graphes.

Le système sur le quel nous avons essayé l'application est Windows 8.1 x64 pro, la configuration de l'ordinateur est la suivante :

- Processeur i7 2600k 8 mb de cache L3.
- 4x4GB (16 Gb) de Ram 1600mhz.
- Carte graphique AMD R9 280X tri-X 3GB.

Voici le graphe représentant la variation de la fréquence d'affichage de l'objet par seconde, la zone est éclairée par une lampe de 70 watts, la résolution de la caméra est réglé sur VGA soit 640x480, qui est une résolution très basse, néanmoins standard :

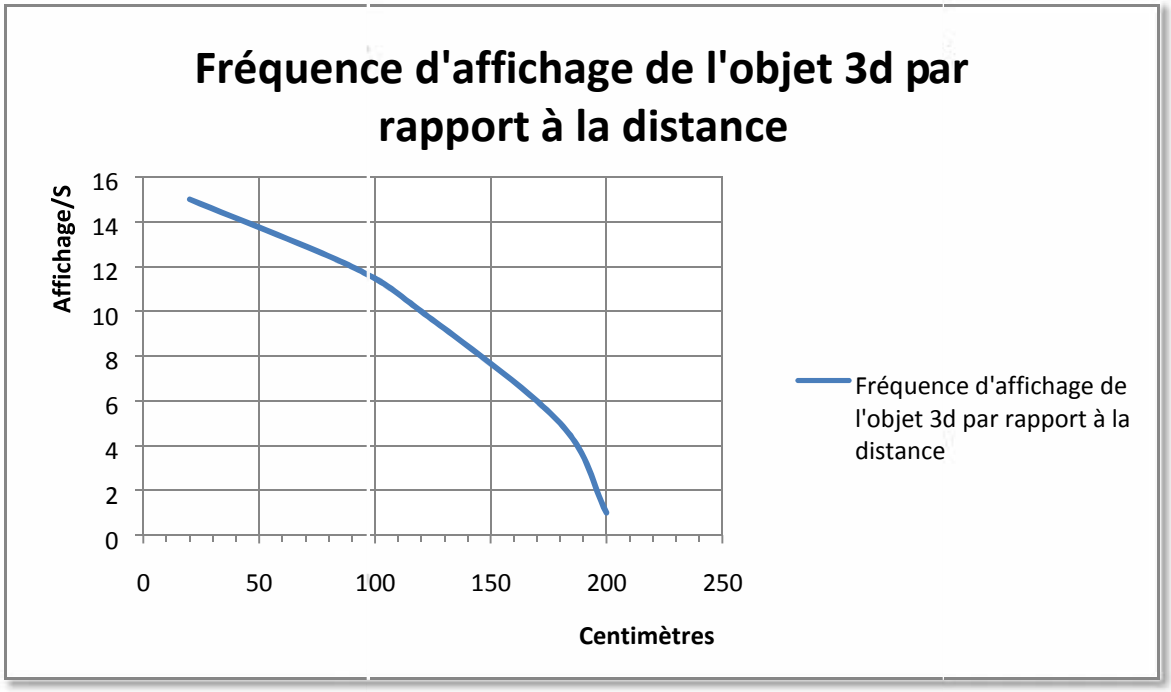


Figure III-53- Fréquence d'affichage de l'objet 3d par rapport à la distance

Nous remarquons depuis le graphe que l'affichage est à son summum lorsque l'objet est à moins de 50 centimètres de la caméra puis s'amenuise de plus en plus qu'on s'éloigne pour enfin disparaitre lorsqu'il est à plus de 2 mètres de la caméra. Il faut souligner que ces résultats sont plutôt satisfaisant, car la lumière n'est pas parfaite, et donc il faut s'attendre à ce que les résultats soient meilleurs en utilisant la lumière du jours.

Nous passons maintenant au graphe représentant la fréquence d'affichage de l'objet 3d par rapport à son orientation, ou l'angle par rapport à la caméra, nous verrons que la fréquence

s'amenuise de plus en plus que le marqueur est parallèle à l'axe de l'image:

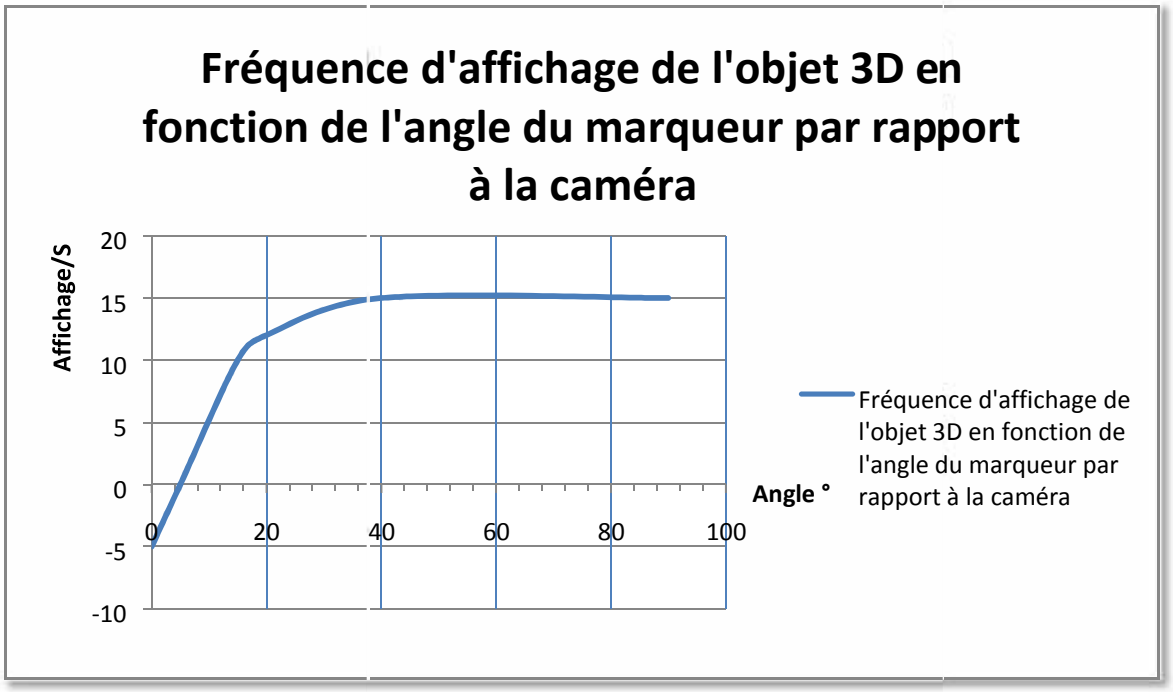


Figure III-54- Fréquence d'affichage de l'objet 3d par rapport à la distance

Dans ce graphe nous voyons que la détection est très stable et à son summum lorsque l'orientation du marqueur est entre 40° et 90°, dans cette plage la détection et le suivi sont très fluides, néanmoins on perd un peu le marqueur dès qu'on rentre dans les angles inférieur à 40° mais la détection reste encore très stable à l'échelle de 12 affichages par seconde, puis après lorsque l'angle est à moins de 20° la détection chute pour presque s'annuler lorsque l'angle est inférieur à 10°. Malgré les résultats de la détection qui sont faibles lorsque le marqueur est presque parallèle à l'axe de la vue, les résultats sont souvent très bon car en général la caméra est surélevée par rapport au marqueur ce qui créé un décalage d'angle suffisant pour que le marqueur soit totalement visible et par conséquent la détection est exécuté d'une manière très correcte.

DISCUSSION

Dans ce chapitre dédié à l'application, nous avons pus construire à partir de zéro une application de réalité augmentée qui utilise les marqueurs sur les quels des objets 3d sont incrustés, nous avons pour cela détecté les contours sur l'image binarisée reçue de la caméra, après cela nous avons restreint la détection sur les contours à 4 coins pour ensuite étaler leurs contenus sur un carré qu'on a décodé. Enfin, nous avons estimé la position du marqueur en utilisant les paramètres intrinsèques de la caméra et les coordonnées des points des marqueurs détectés sur l'image, puis nous avons incrusté des objets 3d sur les marqueurs.

Nous nous sommes également permis de développer certaines fonctionnalités supplémentaires pour notre application tel qu'un module d'importation d'objets 3 D modélisés dans des logiciels tierces dédiés à la création d'images de synthèse. Ainsi que la fonctionnalité multi marqueur qui est très importante dans une application de réalité augmentée, car sans cette fonctionnalité le programme n'aurait pas eu de sens et ne serait pas interactive.

Les résultats obtenus sont très satisfaisants du point de vue performance, l'application est très fluide et tourne à une moyenne de 15 image par seconde, donc le traitement complet d'une seule image prend en moyenne 66 millisecondes, ce qui est très rapide comparé à la lourdeur des calculs effectués sur l'image. Cela a permit au robot de répondre très rapidement et le suivi de la cible s'effectue d'une manière très stable et réactif.

Quand aux résultats du point de vue robustesse, nous avons pu illustrer dans la dernière partie de ce chapitre les nombres d'affichages par seconde en fonction de la distance de l'objet ensuite de son angle de rotation. Par contre nous n'avons pas pu tester d'une manière précise la réponse par rapport à l'éclairage car il nous aurait fallu un studio spécialement dédié aux photos afin de manipuler l'éclairage à notre guise et prendre note des retours. Néanmoins nous pouvons dire que l'application marche dans les conditions d'éclairage normaux, c'est à dire soit éclairage artificiel ou naturel.

CONCLUSION

GÉNÉRALE

Le sujet que nous avons traité dans ce mémoire a pour objectif d'exposer un problème depuis longtemps présent dans le domaine de la vision par ordinateur, à savoir le tracking 3d, qui permet de retrouver et de suivre des objets dans une séquence vidéo ou de reproduire le mouvement de la caméra du monde réel dans une scène virtuelle et d'estimer la position en 3d des objets suivis. Les objets en question peuvent être des marqueurs avec une structure connue ou des objets naturellement présents dans l'environnement réel. Les solutions pour le tracking 3d se trouvent être très nombreuses, particulièrement lorsqu'il s'agit de traitement en temps réel, cela permet, comme dans notre cas, d'asservir d'une manière visuelle un bras robotisé ou un robot plus complexe, cela permet de les rendre autonomes et intelligents, en effet, en analysant l'environnement, ils se développeraient et interagiraient avec les objets qui les entourent d'une manière automatique et évolutive. Ce genre de solutions pourraient aussi servir d'interface intelligente, ou l'on pourrait augmenter la perception du monde réel avec des informations conçues sur ordinateur, tel que la réalité augmentée, application que nous avons développé pour ce thème.

Dans notre étude nous avons pu constater que choisir une approche pour résoudre le problème de tracking reste très lié à l'objectif de l'application pour laquelle est destiné notre travail. Une étude approfondie de beaucoup de méthodes est nécessaire afin de choisir la solution la plus optimale qui soit pour notre application.

Néanmoins, après de nombreuses années de recherche dans ce domaine les systèmes de vision basés sur le tracking restent toujours dépendants des marqueurs placés dans la scène comme les LEDs ou des objets dont le modèle 3D est connu, comme un marqueur. Ces méthodes sont parmi celles qui sont capable, à nos jours, de produire des résultats suffisamment satisfaisants en un temps d'exécution raisonnable. Les autres méthodes sont de plus en plus célèbres de nos jours, comme le projet Natal ou Kinect pour la consol de jeux de Microsoft "Xbox" qui est dotée de lentilles capables de détecter la couleur et la profondeur de l'environnement et des objets, cet appareil est capable de détecter la profondeur ce qui lui permet de déduire la distance à la quelle les objets présents dans la scène sont, ce qui est quasiment impossible sans l'aide des outils mathématique, avec un simple système de caméra embarquée comme le notre. Un autre capteur qui est embarqué dans la Kinect est une caméra infrarouge qui lui permet de voir dans le noir total, chose impossible encore avec une simple caméra.

Les pratiques qui concernent l'utilisation du tracking dans un environnement inconnu, et très riche en informations, sont d'autant plus difficiles car la résolution demande plusieurs

niveaux de traitement et beaucoup d'itérations dans le but d'éliminer un maximum de données aberrantes (c'est pour quoi nous avons limité ce processus à un marqueur), ce qui risque de couter assez cher en ce qui concerne la consommation des ressources du calculateur en place. Toutefois, ce problème est appelé à disparaitre vu l'évolution de la technologie, en effet la puissance que gagnent de plus en plus les ordinateurs et les

appareils mobiles de nos jours, nous permettra de traiter les primitives présentes naturellement dans les images, donc indépendamment des marqueurs artificiels.

Lors de l'élaboration de notre application, nous avons pu constater que le résultat obtenu est très souvent lié à la qualité de la mise en œuvre. Dans notre application, la partie la plus sujette à des variations de résultat est bien la partie de détection du marqueur, car celle ci dépend grandement de l'éclairage et de la résolution et la qualité du capteur, car plus on est loin plus les informations contenues dans le marqueur ne sont pas visibles, c'est pour cela qu'avec une caméra haute résolution la détection serait plus performante, mais risque d'augmenter le temps de calcul car on aurait plus d'informations à traiter dans une seule image. Et donc pour améliorer la fiabilité du système que nous avons développé une meilleur détection est nécessaire.

En général dans la pratique, même les approches les plus robustes souffrent souvent d'échecs, par exemple la perte de la cible nécessite la réinitialisation de la procédure complète, ce qui rend son application moins intéressante du point de vue autonomie. Ce genre de problème est souvent causé par un mouvement brusque ou à la disparition de l'objet cible du champ de vision. Toute fois pour parer à ce problème, il est nécessaire de combiner avec d'autres capteurs, comme un capteur à ultra sons par exemple, qui permet de prédire le mouvement d'un objet, raffiné par une technique de vision similaire à la notre.

À l'état où nous en sommes dans cette application, il nous est possible de créer de multiples applications dérivées du même principe, en guise d'exemple, nous pouvons adapter notre système de détection à d'autres objets cela nous permettrais de nous passer des marqueurs, ou bien, et pourquoi pas dans un projet de doctorat, en utilisant les réseaux de neurones ou le Machine Learning, créer un système qui apprendrait à connaitre les objets et à évaluer leurs taille pour permettre une plus simple estimation et surtout plus précise.

Nous espérons que ce mémoire éveillera une passion et un intérêt scientifique particulier chez les promotions à venir, afin qu'elles puissent persévérer dans ce domaine et améliorer le modeste travail que nous avons entamé, et ce en étudiant et en développant des techniques plus ambitieuses et plus autonomes.

BIBLIOGAPHIE

1. **dobbert, tim.** *matchmoving: the invisible art of camera tracking.* s.l. : sybex, 2005.

2. **Ole.** sb.epfl.ch. [Online] octobre 2014. sb.epfl.ch/files/content/users/123314/files/OLe.pdf.

3. **J.Smith, Warren.** Modern Lens Design : A resource manual. 1992.

4. **thoby, michel.** lens distorsion. *michel.thoby.free.fr.* [Online] août 2009.

http://michel.thoby.free.fr/Fisheye_history_short/International_Standards_about_Distortion.html.

5. **-google.books-, richard taillet.** *optique géométrique.* s.l. : de boeck, 2008.

6. **this, what is.** tracking. *what-is-this.net.* [Online] juillet 2014. http://what-isthis.net/fr/definition/tracking.

7. **Zeeb, Zuber.** *MatchMoving.*

8. **pascaline, parisot.** *Suivi d'objets dans des séquences d'images de scènes déformables : de l'importance des points d'intérêt et du maillage 2D.* 2009.

9. *Object recognition from local scale-invariant features.* **Lowe, David.** août 1999.

10. **Lowe, David.** *Distinctive image features from Scale-Invariant keypoints.* University Of British Columbia : s.n., 2004.

11. **Rosten, Edward.** *High performance rigid body tracking.* University of Cambridge : s.n., 2006.

12. **M. sezgin, B. Sankur.** *Survey over image thresholding techniques and quantitative performance evaluation.* s.l. : Journal of electronic imaging 13, 2004.

13. **Otsu, Nobuyuki.** *A threshold selection method gray-level histograms.* s.l. : IEEE Trans. Sys, Man, Cyber 9 , 1979.

14. **Ping-Sung Liao, tse-Sheng Chen, Pau-Choo Chung.** *A Fast Algorithme for Multilevel Thresolding.* s.l. : J.Inf.Sci.Eng, 2001.

15. **Satoshi, Suzuki.** *Topological Structural Analysis of Digitized Binary Images by Border Following.* Japan : s.n., 1983.

16. **Opencv.** *docs.opencv.org.* [Online] http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html? highlight=pose%20estimation.
17. —. *docs.opencv.org.* [Online] http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.h tml?highlight=solvepnp#cv2.solvePnP.

18. **Daniel lélis baggio, david millan escriva, naureen mahmood, roy shilkrot, shervin emami, khvedchenia levgen, jason saragih.** *Mastering OpenCV with Practical computer vision projects.* s.l. : PACKT publishing, 2012.

19. **sourceforge.** Opencvlibrary. *sourceforge.net.* [Online] 2014. http://opencvlibrary.sourceforge.net/.

20. **TechTerms.** techTerms.com. [Online] 2014. http://www.techterms.com/definition/opengl.

21. **Texas, Instruments.** ULN2003. [Online] www.ti.com/lit/ds/symlink/uln2003a.pdf.

22. **Blum, Jeremy.** Exploring Arduino: tools and techniques for engineering wizardry. 2013.

23. **Nasa.** No More Flying Blind, NASA. *Nasa.gov.* [Online] http://www.nasa.gov/vision/earth/improvingflight/svs_reno.html.

24. **Navy, US.** *F-18 HUD Gun.* s.l. : US Navy, 2004.

25. **force, US Air.** technology conference powers down. *af.mil.* [Online] décember 2007. http://www.af.mil/News/ArticleDisplay/tabid/223/Article/124965/technology-conferencepowers-down.aspx.

26. **Alabaster, Jay.** Personal Technology. *Computerworld.com.* [Online] Juin 28, 2013.

http://www.computerworld.com/article/2498299/personal-technology/pioneer-launchescar-navigation-with-augmented-reality--heads-up-displays.html.

27. **MasterProcreation.** Dispositifs de RA en médecine. *MasterProcreation.free.fr.* [Online] http://masterprocreation.free.fr/XMind/nouvelles_images/tsveta_popova/.

28. **HONG, Eliane.** La réalité augmentée au service des étudiants en médecine. [Online] avril 16, 2014. http://www.atelier.net/trends/articles/realite-augmentee-service-etudiantsmedecine_428788.

TABLE DES ILLUSTRATIONS

FIGURE I-1- LES COMPOSANTES D'UN OBJECTIF. 4

FIGURE I-2- DIFFERENCE ENTRE COURTE ET LONGUE DISTANCE FOCALE....5

FIGURE I-3- DIFFERENCE ENTRE UN ZOOM ET UN DEPLACEMENT EN PROFONDEUR....6

FIGURE I-4- FRUSTUM....7

FIGURE I-5- ANGLE ET CHAMP DE VUE. 7

FIGURE I-6- LES DIFFERENTES POSITIONS DU DIAPHRAGME....8

FIGURE I-7- DISTORSION TANGENTIELLE ET RADIALE. 9

FIGURE I-8- MODELISATION DE LA DISTORSION RADIALE....10

FIGURE I-9- IMAGE DISTORDUE, IMAGE CORRIGEE....11

FIGURE II-1- FLUX DE LA CAMERA REELLE 16

FIGURE II-2- FLUX DE LA CAMERA VIRTUELLE....17

FIGURE II-3- LES DEUX FLUX FUSIONNES....17

FIGURE II-4- DIFFERENTS TYPES DE COINS : (A) JONCTION EN "L", (B) JONCTION EN "V", (C) JONCTION EN "T", (D) JONCTION EN "Y", (E) JONCTION EN "X" ET (F) JONCTION EN "DAMIER"....19

FIGURE II-5- FRISE DES DETECTEURS DE POINTS. 20

FIGURE II-6- PYRAMIDE DE GRADIENTS : 3 OCTAVES DE 5 GRADIENTS. 22

FIGURE II-7- CONSTRUCTION DE LA PYRAMIDE DE DIFFERENCES DE GAUSSIENS (DOG) A PARTIR DE LA PYRAMIDE DE GRADIENTS....23

FIGURE II-8- (A) DETECTION INITIALE, (B) ELIMINATION DES POINTS A FAIBLE CONTRASTE, (C) ELIMINATION DES POINTS SITUES SUR LES ARETES. 24

FIGURE II-9- CONSTRUCTION DE L'HISTOGRAMME DES ORIENTATIONS. 25

FIGURE II-10- CONSTRUCTION D'UN DESCRIPTEUR SIFT....27

FIGURE II-11- 12 POINTS DE DETECTION DANS UNE PARTIE DE L'IMAGE.....32

FIGURE II-12- (A) IMAGE TEST CONTENANT DES POLYGONES AVEC DES LIGNES QUI SE JOIGNENT A DES ANGLES DIFFERENTS. (B) LA PREMIERE IMAGE DE LA SEQUENCE DE SOUS SOL DE L'UNIVERSITE D'OXFORD (512X512). (C) UNE PHOTOGRAPHIE DU KING'S COLLEGE, CAMBRIDGE (768X576).....32

FIGURE II-13- APPLICATION DU DETECTEUR SEGMENT-TEST AVEC R=3 ET N=12. (A) 318 POINTS. (B) 1482 POINT (T=17). (C) 2594 POINTS (T=50). LES POINTS-CLES SONT INDIQUES AVEC DES POINTS ROUGES...33

FIGURE II-14- PROJECTION DU NUAGE DE POINT DANS UNE SCENE 3D.....38

FIGURE III-1- SCHEMA DES CLASSES DU PROGRAMME.....50

FIGURE III-2- SCHEMA DU FONCTIONNEMENT DU PROGRAMME.....51

FIGURE III-3- EXEMPLE DE MARQUEUR 7X7.....53

FIGURE III-4- CAMERA CAPTURANT UN MARQUEUR 54

FIGURE III-5- CONVERSION EN NIVEAU DE GRIS DE L'IMAGE SOURCE 55

FIGURE III-6- BINARISATION DE L'IMAGE EN NIVEAU DE GRIS.....56

FIGURE III-7- AFFICHAGE DES CONTOURS EXTRAITS DE L'IMAGE BINAIRE.....57

FIGURE III-8- SUPPRESSION DE LA PERSPECTIVE 60

FIGURE III-9- BINARISATION DU MARQUEUR.....60

FIGURE III-10- PARTITIONNEMENT DU MARQUEUR 61

FIGURE III-11- 4 REPRESENTATIONS POSSIBLES D'UN SEUL MARQUEUR 62

FIGURE III-12- ROTATION DU MARQUEUR POUR RECONNAISSANCE 64

FIGURE III-13- POSITION DU MARQUEUR AMELIOREE 66

FIGURE III-14- REPRESENTATION DU MARQUEUR EN 3D ET SA PROJECTION SUR LE PLAN DE L'IMAGE.....67

FIGURE III-15- COORDONNEES 3D DU MARQUEUR.....68

FIGURE III-16- DAMIER 9X6 UTILISE POUR LA CALIBRATION 71

FIGURE III-17- OBJET 3D INSERE DANS LA SCENE REELLE (LA CHAISE).....77

FIGURE III-18- PRISE INSTANTANEE DE L'ENVIRONNEMENT DE DEVELOPPEMENT POUR ARDUINO 77

FIGURE III-19- POSITION DU MARQUEUR SUR L'IMAGE 80

FIGURE III-20- INTERFACE DU PROGRAMME 81

FIGURE III-21- PLUSIEURS MARQUEURS SUR LA MEME IMAGE 84

FIGURE III-22- UNE FACE EN 3D.....87

FIGURE III-23- DETECTION DES POINTS CLE D'UN VISAGE ET LEURS COORDONNEES 90

FIGURE III-24- INCRUSTATION D'OBJET 3D SUR LE VISAGE.....91

FIGURE III-25- VTH D'UN AVION DE CHASSE.....92

FIGURE III-26- FIXATION DU VISEUR SUR LE CASQUE.....93

FIGURE III-27- REALITE AUGMENTEE SUR LE PARE BRISE D'UNE VOITURE 93

FIGURE III-28- REALITE AUGEMENTEE DANS LA MEDECINE.....94

FIGURE III-29- REALITE AUGMENTEE DANS L'EDUCATION 95

FIGURE III-30- FREQUENCE D'AFFICHAGE DE L'OBJET 3D PAR RAPPORT A LA DISTANCE 96

FIGURE III-31- FREQUENCE D'AFFICHAGE DE L'OBJET 3D PAR RAPPORT A LA DISTANCE 97