

MÉMOIRE  
PROJET DE FIN D'ÉTUDES

RÉALITÉ AUGMENTÉE SUR TÉLÉPHONE  
PORTABLE, TABLETTE ET LUNETTES

Client : Pascal DESBARATS.

Master 2 Informatique 2016-2017

Cindy BECHER - Alexandre FORTUNA - Raphael JOREL

12 juillet 2018



# Résumé

Ce projet a été réalisé dans le cadre d'un projet de fin d'études en Master 2 informatique spécialité Informatique pour l'Image et le Son. Le but de ce projet a été d'implémenter des briques de base logicielles permettant de détecter des marqueurs de types QR code et des feuilles rectangulaires blanches depuis un flux vidéo issu d'une caméra 2D, afin d'en détecter la position et l'orientation dans l'espace pour afficher par-dessus des informations, comme des modèles 2D ou 3D. Pour prouver la faisabilité et fournir ces briques, le client nous a demandé de développer une application fonctionnant sur téléphones et tablettes Android et d'adapter ce code pour que l'application marche sur un certain type de lunettes de réalité augmentée : Moverio BT-300 d'Epson.

Ce rapport présente le développement et la mise en oeuvre de ces fonctionnalités et expose les limites et les améliorations pouvant être apportées au code ou aux matériels de réalité augmentée actuels.

**Mots clés :** Réalité augmentée, QR Code, détection d'une feuille blanche, lunettes de réalité augmentée.

# Abstract

This project has been realized in the context of an end of studies' project in Master 2 of computer science specialty Computer science for image and sound. The goal of this project has been to implement basic software bricks allowing to detect QR code markers or rectangular white sheets of paper from a 2D camera video flux, to detect their space position and orientation, to display 2D or 3D models above. To prove that it's possible and to do this bricks, our client wanted us to develop an application on Android smartphones and tablets, and to adapt this code for specific augmented reality glasses : the Moverio BT-300 by Epson.

This report presents the development and the implementation of this features, tells the limits and the improvements that can be done in the code or for the current augmented reality material.

**Keywords :** Augmented reality, QR Code, detection of a white sheet of paper, augmented reality glasses.



# Table des matières

<b>1</b>	<b>Domaine / Sujet</b>	<b>1</b>
<b>2</b>	<b>Cas d'utilisation</b>	<b>3</b>
	Modèle 3D sur marqueur . . . . .	3
	Squelette entier sur marqueurs . . . . .	3
	Application feuille blanche . . . . .	3
<b>3</b>	<b>État de l'art / existant</b>	<b>5</b>
3.1	État de l'art . . . . .	5
3.2	Existant . . . . .	5
3.2.1	AR on QR . . . . .	5
3.2.2	Applications scanner . . . . .	5
3.3	Bibliothèques, APIs . . . . .	6
3.3.1	Vuforia . . . . .	6
3.3.2	Wikitude . . . . .	6
3.3.3	Android API . . . . .	6
3.3.4	ARToolKit . . . . .	6
3.3.5	OpenCV . . . . .	6
3.3.6	Zxing . . . . .	7
<b>4</b>	<b>Besoins</b>	<b>9</b>
4.1	Fonctionnels . . . . .	9
4.1.1	Besoins généraux . . . . .	9
4.1.2	Téléphone / tablette Android . . . . .	9
4.1.3	Lunettes EPSON BT300 . . . . .	9
4.1.4	Marqueurs . . . . .	10
	QR Codes . . . . .	10
	Feuille blanche . . . . .	10
4.2	Non-fonctionnels . . . . .	10
4.2.1	Fluidité . . . . .	10
4.2.2	Réutilisabilité, généricité . . . . .	10
4.2.3	Robustesse . . . . .	10
4.2.4	Précision . . . . .	11
<b>5</b>	<b>Architecture</b>	<b>13</b>
5.1	Application QR Codes . . . . .	13
5.1.1	Diagrammes de classes . . . . .	13
5.1.2	Détail de classes . . . . .	14
	MainActivity (Java) . . . . .	15

CameraController (Java) . . . . .	15
QRCodeProcessor (Java / C++) . . . . .	15
RendererWrapper (Java) . . . . .	15
ShaderProgram (C++) . . . . .	15
Mesh (C++) . . . . .	15
5.1.3 Diagramme de séquence . . . . .	16
5.2 Application feuille blanche . . . . .	17
MainActivity . . . . .	17
FindWhitePaper . . . . .	17
Quadrilateral . . . . .	17
5.2.1 Diagramme de séquence . . . . .	18
<b>6 Implémentation</b>	<b>19</b>
6.1 Outils, IDEs . . . . .	19
6.1.1 Android Studio . . . . .	19
6.1.2 Unity3D . . . . .	19
6.2 Travail . . . . .	19
6.2.1 Prototypage avec Unity3D . . . . .	19
6.2.2 Utilisation des marqueurs de type QR Code . . . . .	20
6.2.2.1 Développement sur téléphone / tablette . . . . .	20
Différents supports. . . . .	20
Android / Java. . . . .	20
Android / C++ . . . . .	20
Configuration d'ARToolKit. . . . .	20
Plusieurs niveaux de contrôle. . . . .	21
Détection d'un QR Code. . . . .	21
Passage à OpenCV. . . . .	22
Disposition et orientation. . . . .	22
Recalage. . . . .	22
Lecture du contenu. . . . .	23
Développement du <i>wrapper</i> et implantation sous Android. . . . .	23
Affichage d'un modèle 3D. . . . .	23
6.2.2.2 Développement pour les lunettes Moverio BT-300 d'EPSON	24
6.2.3 Travail avec une feuille blanche . . . . .	24
6.2.3.1 Développement sur téléphone / tablette . . . . .	24
Détection de la feuille blanche. . . . .	24
Détection de contours. . . . .	25
Détection des coins. . . . .	26
Méthode basée sur l'application Scanner. . . . .	26
Orientation et position de la feuille blanche. . . . .	26
6.2.3.2 Développement pour les lunettes Moverio BT-300 d'EPSON	27
Portage du code. . . . .	27
Adaptation aux lunettes. . . . .	28
Différences d'affichage entre les lunettes et les téléphones/tablettes.	28
Recalage de l'image sur la feuille réelle. . . . .	29
<b>7 Tests</b>	<b>31</b>
7.1 QR code . . . . .	31
7.2 Feuille blanche . . . . .	32

<b>8 Bilan / perspectives</b>	<b>37</b>
8.1 Limites des applications . . . . .	37
8.1.1 QR codes . . . . .	37
8.1.2 Feuille blanche . . . . .	37
8.1.2.1 Téléphones et tablettes . . . . .	37
8.1.2.2 Lunettes EPSON Moverio BT-300 . . . . .	37
8.2 La technique . . . . .	39
8.2.1 Le matériel. . . . .	39
8.2.2 Les bibliothèques. . . . .	39
8.3 Remarques . . . . .	39
8.3.1 Note sur les performances de Java et C++ . . . . .	39
8.3.2 Note sur la taille des applications . . . . .	40
8.4 Les améliorations possibles . . . . .	40
8.4.1 Général . . . . .	40
Activités. . . . .	40
8.4.2 QR Codes . . . . .	40
Détection des coins. . . . .	40
<i>Tracking</i> . . . . .	41
Intégration de l'application OpenGL . . . . .	41
Modèles 3D via l'internet. . . . .	41
QR Codes multiples. . . . .	41
8.4.3 Feuille blanche . . . . .	41
Robustesse. . . . .	41
Sélection d'une image ou prise en direct. . . . .	41
Programmer les traitements d'image en C++. . . . .	41
Stéréoscopie. . . . .	42



# Introduction

La réalité augmentée est en expansion depuis quelque années, et beaucoup d'entreprises investissent dans ce domaine. Elle consiste à superposer des informations virtuelles à la réalité, plus précisément à ce que peut voir, entendre, toucher, ou encore sentir l'utilisateur.

Des entreprises telles que Microsoft travaillent sur ce sujet pour proposer des dispositifs permettant donc d'augmenter la quantité d'informations que l'Humain peut percevoir. Dans notre cas nous nous attacherons à ajouter des informations visuelles à la vision de l'utilisateur. Pour cela nous devons utiliser des périphériques ayant les capacités d'analyser l'environnement entourant l'utilisateur. Le plus classiquement, on peut citer les smartphones et tablettes, qui, grâce à leur caméra, permettent une telle analyse. Mais on peut également ajouter les lunettes de réalité augmentée qui s'imposent presque d'elles-mêmes dans ce domaine.

En effet, elles permettent d'afficher des informations directement à la vue de l'utilisateur sans avoir besoin de tenir un écran devant son visage. Ici l'écran est directement intégré aux lunettes, juste devant les yeux de l'utilisateur.

Des communications entre les dispositifs peuvent permettre d'enrichir l'expérience, comme par exemple faire communiquer un casque avec un réfrigérateur connecté. Le casque peut afficher sur la porte du réfrigérateur une liste de courses mise à jour automatiquement par le réfrigérateur.

Également, nos téléviseurs peuvent être remplacés par de simples écrans virtuels visibles avec nos lunettes de réalité augmentée. Ainsi, chaque utilisateur peut regarder le programme qu'il veut, et il n'existe plus de problématique liée à la résolution et à la taille de l'écran, la seule limite étant celle des lunettes.

Les outils pour la réalité augmentée sont pour le moment assez récents, et les applications qui en découlent sont donc encore trop peu nombreuses et assez limitées.

Dans ce rapport, nous présentons une application se contentant d'afficher des objets 2D / 3D sur des marqueurs (type QR code) ou une feuille blanche.



# Chapitre 1

## Domaine / Sujet

Dernièrement, on a vu arriver sur le marché bon nombre d'applications de réalité virtuelle. Mais en ce qui concerne la réalité augmentée, les applications disponibles se limitent pour le moment à de petites démonstrations, sans exploiter toutes les capacités de la réalité augmentée. Ces applications ne constituent pour le moment qu'une vitrine et n'ont pas de réelle utilité dans la vie des utilisateurs.

Mais il ne fait aucun doute que cette technologie saura à l'avenir se montrer des plus utiles pour améliorer le quotidien des utilisateurs, et ce dans plusieurs domaines. Notamment dans l'industrie, dans le cas de prototypage, il serait utile que deux personnes, ou plus, puissent interagir sur le même modèle 3D en même temps sans avoir la contrainte de devoir s'immerger dans un monde virtuel comme c'est le cas avec les casques de réalité virtuelle. Dans la médecine, on pourrait imaginer qu'après un scanner ou une IRM on puisse reproduire un modèle 3D de la partie du corps observée pour la visualiser en direct pendant une opération par exemple ; ce qui pourrait aider les chirurgiens à être plus précis.

Notre client souhaite avoir une application de réalité augmentée sur téléphone et tablette Android, puis sur des lunettes EPSON BT-300, le but étant de visualiser des objets virtuels à travers une réalité augmentée.

Les lunettes EPSON BT-300 utilisent le système d'exploitation Android, d'où l'idée de commencer sur des terminaux accessibles à tous sous Android (smartphones, tablettes) pour développer le cœur de l'application, pour ensuite essayer de porter cette application sur les lunettes.

Notre but est donc de développer une ou plusieurs applications permettant d'afficher des modèles 2D ou 3D sur des marqueurs, qu'ils soient de type QR code ou feuille blanche.

Ce qui intéresse avant tout notre client c'est de trouver l'orientation et la position dans l'espace des marqueurs et des feuilles blanches et de recaler des images ou des objets virtuels par dessus. L'application n'est là que pour montrer que c'est possible.



# Chapitre 2

## Cas d'utilisation

Nous avons pour objectif de faire une ou plusieurs applications permettant de visualiser des objets 2D ou 3D sur des marqueurs de type QR Codes ou sur des feuilles blanches. Ces applications devront permettre quelque cas d'utilisation, énumérés par notre client.

**Modèle 3D sur marqueur.** Un premier cas d'utilisation serait de pouvoir filmer un marqueur avec son téléphone portable Android et observer un objet 3D via l'écran de son téléphone sur ce marqueur. Cette utilisation doit également être possible avec une tablette Android.

Dans un second temps, l'utilisateur pourra porter une paire de lunettes EPSON Moverio BT300 et regarder un marqueur. Il verrait alors apparaître un objet sur ce marqueur.

Dans les deux cas, l'utilisateur pourra déplacer le marqueur et continuer de voir l'objet 3D si le marqueur est toujours dans le champ de vision de la caméra. Lorsqu'il fera pivoter le marqueur, l'objet 3D pivotera de la même façon.

**Squelette entier sur marqueurs.** Notre client devrait pouvoir ainsi avoir des marqueurs sur lui, correspondant par exemple à des os en 3D, et pouvoir voir ses os par le biais de son téléphone ou de ses lunettes de réalité augmentée. L'idéal serait que ces os lui apparaissent à taille réelle, comme s'ils étaient présents dans la réalité. Il pourra donner un marqueur à une autre personne qui percevra l'os depuis son propre point de vue. C'est à dire que si le client et la seconde personne sont face à face et l'os placé entre eux deux, l'un verra une partie de l'os et l'autre verra l'autre partie, que le premier ne peut pas voir.

**Application feuille blanche** Notre client veut pouvoir afficher une image de contour sur une feuille blanche afin de pouvoir dessiner directement sur la feuille à partir du modèle affiché. Cette image devra être chargée depuis la mémoire du téléphone, de la tablette ou des lunettes, ou pourra être prise via l'application elle-même. Les formats de feuilles à privilégier sont les formats A4, A5 et A6. Pour une utilisation optimale, l'image affichée devra toujours apparaître sur la feuille, même lorsque la main de l'utilisateur dessine dessus. Elle devra également suivre les mouvements de la feuille et être parfaitement incorporée dans celle-ci.



# Chapitre 3

## État de l'art / existant

### 3.1 État de l'art

La réalité augmentée permet d'accroître la quantité d'informations qu'un Homme peut capter à partir de ses sens. Le rapport de présentation de la citation [15] expose une application sur Android faisant exactement ce que l'on souhaite pour la partie d'analyse des QR Codes. Cette dernière détecte et lit un QR Code pour ensuite afficher un modèle 3D dessus en lien avec le contenu du QR Code. Malheureusement, le code source de l'application n'est pas disponible et nous ne pouvons donc pas étudier son fonctionnement.

### 3.2 Existant

#### 3.2.1 AR on QR

"*AR on QR*" [3] présente exactement ce que nous voulons faire concernant l'affichage de modèles 3D sur des marqueurs de type QR codes. Seul un rapport décrivant comment marche l'application est disponible, l'application elle-même et le code n'ont pas été trouvés. Cela aurait pu être pratique pour comprendre la logique, mais le rapport aiguille déjà en bonne partie sur une marche à suivre.

#### 3.2.2 Applications scanner

Il existe sur les différentes plate-formes de téléchargement d'applications plusieurs applications permettant de scanner une feuille. Ces applications détectent la feuille et lui appliquent une transformation pour la "redresser" afin de générer une image rectangulaire ne contenant que la feuille. Ce qui peut nous intéresser dans ces applications, c'est la partie concernant la détection de la feuille. De plus, elles calculent une transformation pour "redresser" la feuille, et nous aurons besoin de la transformation inverse puisque, nous le rappelons, notre objectif est d'afficher une image sur la feuille en tenant compte de son orientation, de sa déformation et de sa position.

Cependant, ces applications reposent sur l'analyse d'une seule image et l'algorithme qu'elles utilisent ne peut peut-être pas être utilisé dans une application qui nécessite une réponse rapide.

### 3.3 Bibliothèques, APIs

#### 3.3.1 Vuforia

Vuforia [17] est une plateforme de développement qui permet de réaliser des applications de réalité augmentée. De nombreux tutoriaux sont disponibles sur internet, et permettent "d'attacher" virtuellement un objet 3D à un marqueur. De cette façon, l'application récupère le flux vidéo, détecte le marqueur et affiche l'objet 3D sur ce marqueur. Vuforia met à disposition des outils permettant de travailler sur Android, IOS, Windows Phone ou encore Unity.

Nous n'avons pas utilisé Vuforia pour réaliser nos applications car il ne permet pas de descendre assez bas niveau pour faire tout ce que l'on veut.

#### 3.3.2 Wikitude

Wikitude est une entreprise spécialisée dans la réalité augmentée [20]. Elle propose son propre logiciel permettant de développer des applications qui seront hébergées par leur application Wikitude téléchargeable sur iOS, Android et BlackBerry, ou bien de créer sa propre application indépendante. Wikitude propose également un SDK (*Software Development Kit*) afin d'intégrer leur fonctionnalités dans des projets développés avec d'autres logiciels, par exemple Unity3D.

Nous n'avons pas utilisé Wikitude pour exactement les mêmes raisons qui ont fait que nous n'avons pas utilisé Vuforia.

#### 3.3.3 Android API

L'utilisation de l'API<sup>1</sup> Android [1] est évidemment obligatoire pour développer des applications fonctionnant sur ce système. Cet ensemble de bibliothèques permet d'avoir accès à tous les services que propose Android, notamment l'accès à la caméra (ce qui nous intéressera le plus par la suite) mais également aux différents capteurs, la possibilité d'utiliser les applications de base du système, etc...

#### 3.3.4 ARToolKit

ARToolKit [4] est un SDK<sup>2</sup> permettant de faire du *tracking*<sup>3</sup> de marqueurs. Les marqueurs doivent être pré-déterminés, il n'est pas possible d'associer un type de marqueur et de faire en sorte qu'ARToolKit reconnaisse ce type automatiquement. On peut voir figure 3.1 un exemple de marqueur très classique sous ARToolKit, le marqueur Hiro.

#### 3.3.5 OpenCV

OpenCV [11] est une bibliothèque open source permettant de faire du traitement d'image et de l'apprentissage automatique (*machine learning* en anglais). OpenCV a été créée afin d'accélérer certains traitements. Cette bibliothèque a l'avantage d'être open source, disponible en Java, C, C++, Python, MATLAB et peut être utilisée sous Windows, Linux, Android et Mac OS. Nous pourrons donc utiliser facilement OpenCV dans

---

1. Application Programming Interface

2. Software Development Kit

3. Le tracking permet de suivre un objet dans un flux vidéo



FIGURE 3.1 – Marqueur "Hiro"

notre projet. De plus, c'est une bibliothèque assez répandue, qui dispose d'une grande communauté de programmeurs et d'une bonne documentation.

### 3.3.6 Zxing

Zxing [14] est une bibliothèque permettant de lire le contenu de plusieurs types de codes barres. Les codes barres en 1D, mais également ceux en 2D, comme les QR Codes. Ecrite en Java, cette bibliothèque possède également des *bindings* en d'autres langages, notamment en C++.



# Chapitre 4

## Besoins

### 4.1 Fonctionnels

#### 4.1.1 Besoins généraux

Nous devons récupérer le flux vidéo d'un appareil ou d'une tablette de type Android afin d'y détecter un marqueur binaire de type QR Code. Il faudra ensuite effectuer une action sur ce marqueur. Enfin, nous devrons considérer un marqueur qui ne sera désormais qu'une feuille blanche et réaliser des actions dessus. Nous analyserons donc le flux vidéo afin de détecter les coins de la feuille blanche, voire les bords de cette feuille pour en déduire sa déformation et son orientation.

#### 4.1.2 Téléphone / tablette Android

Dans le cas de la tablette et du téléphone Android, nous devrons, dans un premier temps, récupérer le flux vidéo et afficher une image 2D sur un marqueur binaire. Dans un second temps, nous devrons afficher un modèle 3D sur ce marqueur. Enfin, nous devrons afficher une image 2D sur une feuille blanche qui aura le rôle de marqueur. Nous devrons donc agir sur le flux vidéo précédemment récupéré et réafficher ce flux vidéo avec des informations supplémentaires.

L'image 2D pourra être dans les ressources de l'application dans un premier temps, puis pourra être chargée par l'utilisateur depuis ses fichiers ou prise en direct par l'appareil photo. La déformation et l'orientation du marqueur devront être prises en compte pour afficher l'image 2D ou le modèle 3D, qu'il s'agisse d'un marqueur binaire ou de la feuille blanche.

Il faudra potentiellement prendre en compte le fait que la caméra n'est pas toujours centrée sur l'appareil ce qui pourra induire des transformations particulières à appliquer au modèle.

#### 4.1.3 Lunettes EPSON BT300

Pour ce dispositif, nous n'aurons pas à réafficher l'intégralité du flux vidéo, mais seulement les informations que nous voudrons ajouter au champ de vision de l'utilisateur. Pour les afficher au bon endroit, nous devrons prendre en compte le fait que la caméra n'est pas centrée sur les lunettes et que le flux vidéo qu'on obtient à partir d'elle est décalé par

rapport à la vision de l'utilisateur. De plus, un affichage stéréoscopique sera nécessaire pour que l'utilisateur perçoive l'image en relief. On pourra ajouter une fonctionnalité permettant d'adapter la stéréoscopie à l'écart pupillaire de chacun en jouant sur l'écartement des caméras dans la scène 3D.

#### 4.1.4 Marqueurs

**QR Codes.** Le type de marqueurs que nous utiliserons sera des QR Codes qui sont des marqueurs binaires. On pourra choisir de s'en servir comme marqueur classique ou comme marqueur de contenu avec un lien URL vers une image 2D ou un modèle 3D.

**Feuille blanche.** On utilisera également des feuilles blanches comme marqueur. Pour cela, nous avons potentiellement besoin de connaître la taille de la feuille blanche afin d'en déduire sa position en 3D dans l'espace. On pourra proposer des formats classiques ou l'utilisateur pourra rentrer lui même son propre format de feuille.

L'application scanner dont nous avons parlé plus haut [8] permet de scanner tout type de feuille. Il pourrait être utile de s'inspirer de son code pour laisser l'utilisateur plus libre dans le choix du format de sa feuille.

### 4.2 Non-fonctionnels

#### 4.2.1 Fluidité

L'application devra pouvoir suivre un marqueur et afficher un nombre d'images par seconde fixé par le client à 20 FPS<sup>1</sup> minimum.

**Test :** Compter le nombre d'images par seconde pour estimer le nombre minimal que l'application pourra afficher.

#### 4.2.2 Réutilisabilité, généricité

Nous devrons avoir des briques de base portables sur système Android. Pour cela, il faudra s'assurer que les entrées / sorties seront bien définies et sans effet de bords. Pour assurer la réutilisabilité, une documentation devra être écrite ou le code devra être suffisamment commenté, cela permettra de préciser si des effets de bords ne peuvent être évités. L'architecture devra permettre l'extensibilité des applications.

**Test :** Faire des tests unitaires pour chaque module et s'assurer des résultats. Vérifier que l'architecture permet bien l'extensibilité.

#### 4.2.3 Robustesse

L'application devra avoir une bonne gestion d'erreur et être stable. Les entrées de l'application devront être vérifiées pour ne pas compromettre le fonctionnement de cette dernière.

---

1. *Frames Per Second*

**Test :** Stresser l'application et regarder si elle réagit bien à cette surcharge. Par stresser, nous entendons soumettre le traitement vidéo à plusieurs marqueurs simultanément, afin de voir si l'application gère correctement les marqueurs.

Pour la robustesse aux entrées dans l'application, il faudra tester les cas où les données en entrée ne correspondent pas à ce que l'application attend, pour voir si cette dernière gère correctement ces cas d'erreurs.

#### 4.2.4 Précision

Le placement des modèles devra être visuellement acceptable. Cette notion est assez difficile à définir, car elle dépend de l'appréciation de l'utilisateur. Pour faire simple, il faudra au minimum que l'utilisateur ait la sensation que l'object affiché soit bien sur le marqueur associé.

**Test :** Validation utilisateur, en utilisant un formulaire simple comprenant deux questions :

- Avez-vous l'impression que l'objet est bien sur le marqueur ?
- Si non, à combien estimeriez-vous le décalage entre l'objet et le marqueur ?



# Chapitre 5

## Architecture

### 5.1 Application QR Codes

#### 5.1.1 Diagrammes de classes

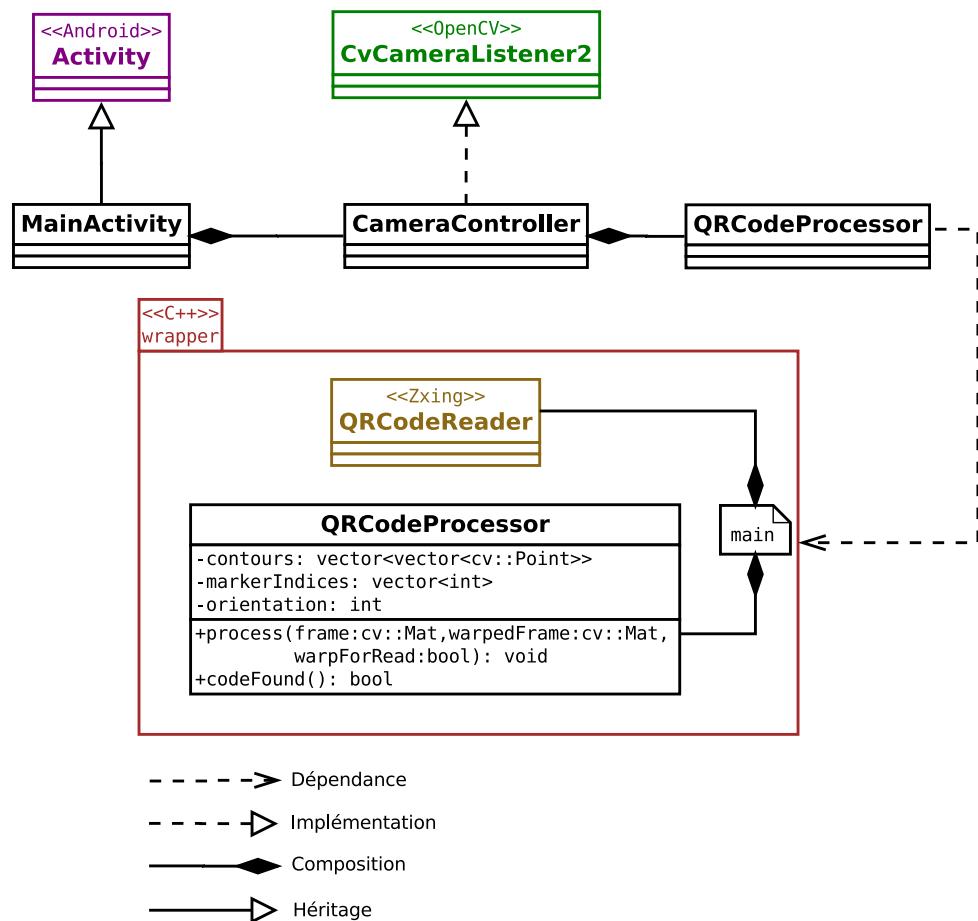


FIGURE 5.1 – Application QR Code

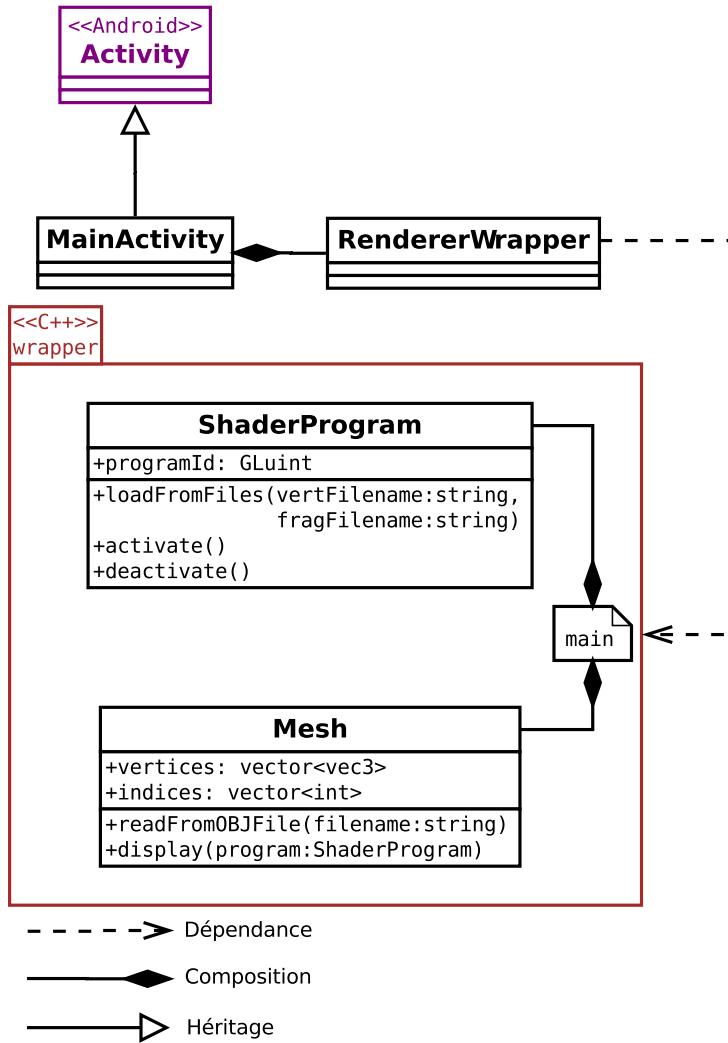


FIGURE 5.2 – Application OpenGL

La présence de deux architectures différentes résulte du fait que le développement complet de l'application sur les QR Codes n'a pas pu être finalisée. De ce fait, deux solutions ont été explorées. La figure 5.1 présente l'architecture d'une application détectant et lisant des QR Codes, et la figure 5.2, quant à elle, montre celle d'une application faisant de l'affichage OpenGL. Ces deux applications ont été développée en partie en C++, pour des raisons de performances, comme nous le verrons plus tard.

Seuls les points jugés importants apparaissent sur ces figures. Ainsi, certains attributs de classe n'apparaissent pas, et certaines fonctions (notamment celles de sous-traitement de la classe `QRCodeProcessor`) ne figurent pas sur le diagramme.

Les liaisons entre Java et C++ sont explicitées par les fichiers `main` qui exposent des services natifs aux classes reliées.

### 5.1.2 Détail de classes

Nous allons maintenant étudier le rôle de chacune des classes. Les classes externes ne sont pas explicitées.

**MainActivity (Java).** Une activité, sous Android, représente plus ou moins une partie de l’application. Une application présente généralement plusieurs activités, chacune ayant des rôles précis. Dans notre cas, une seule activité est présente, car seule la gestion de la caméra et le traitement des images du flux est réalisé. Mais il serait possible de rajouter une activité pour permettre à l’utilisateur de paramétrer sa caméra, la calibrer, etc...

**CameraController (Java).** Cette classe est présente afin de séparer les rôles. Dans les applications d’exemple d’OpenCV, la classe **MainActivity** gère également le traitement de la caméra, mais cela ne respecte pas le *design pattern* MVC<sup>1</sup>. Bien qu’avec une architecture si simple, le respect de ce *design pattern* pourrait être évité, il est important de séparer les rôles quand cela est possible.

**QRCodeProcessor (Java / C++).** Mélange de C++ et de Java, lié par un mécanisme appelé JNI<sup>2</sup>, cette classe assure la détection, le recalage (*warping* selon le vocabulaire d’OpenCV) et la lecture d’un QR Code. L’utilisation de C++ ayant été faite pour des questions de vitesse d’exécution, la classe se sert de fonctions d’OpenCV [11] natives et de Zxing [14] pour la lecture du contenu d’un QR Code. La position 3D du QR Code n’est pas déterminée par cette classe.

**RendererWrapper (Java).** Dans l’application OpenGL, cette classe sert seulement de lien entre Java et C++. Elle donne entre autres un accès aux *assets* de l’application, permettant notamment d’écrire des *shaders* dans des fichiers au lieu de devoir les coder en dur.

**ShaderProgram (C++).** La gestion des *shaders* est un ensemble d’appels à des primitives d’OpenGL, alors cette classe encapsule tout ces traitements pour proposer une interface simplifiée, où seules les références aux fichiers sources des shaders sont nécessaires.

**Mesh (C++).** Représentation d’un maillage 3D à faces triangulaires, cette classe permet de lire un fichier OBJ pour construire le maillage et ensuite afficher ce dernier en utilisant la configuration de *shaders* voulue à partir d’un objet **ShaderProgram**.

- 
1. *Model - View - Controller.*
  2. *Java Native Interface.*

### 5.1.3 Diagramme de séquence

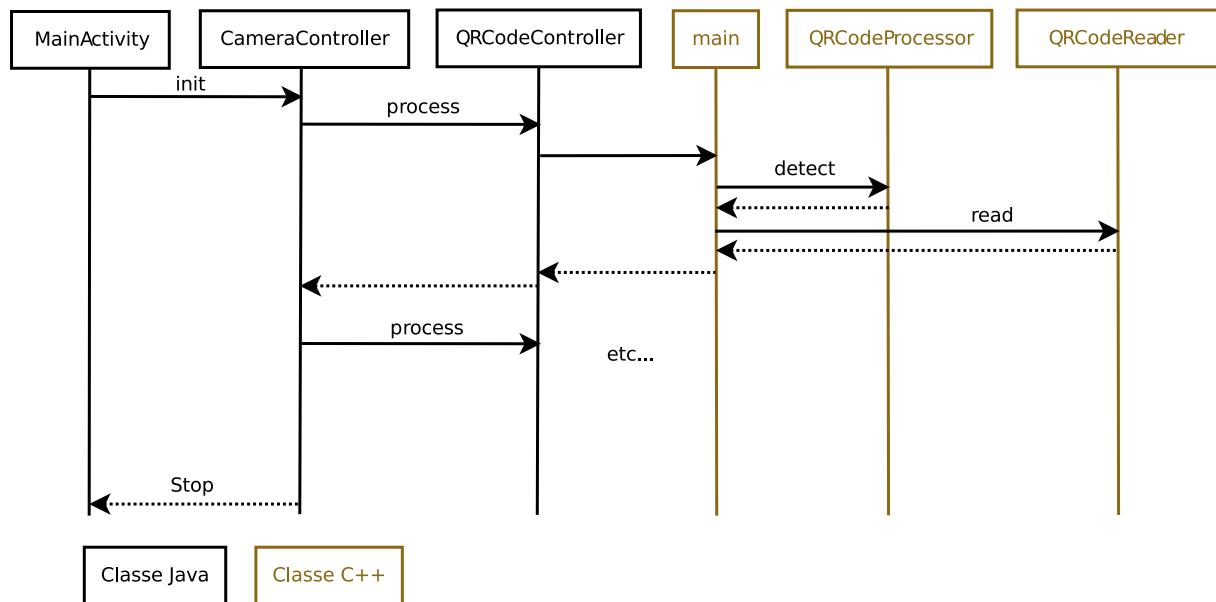


FIGURE 5.3 – Application QR Codes

Seul le diagramme de séquence associé à l’application de détection des QR Codes est présenté ici, car celui que l’on pourrait faire pour l’application OpenGL présente peu d’intérêt au vu de sa simplicité.

L’indication ‘etc...’ explicite le fait que la chaîne principale est faite à l’infini tant que l’application s’exécute. L’action **process** demande au **QRCodeProcessor** de traiter une image. Ensuite cette image est envoyée au *wrapper* (envoi symbolisé avec la flèche entre le classe **QRCodeProcessor** et le fichier **main**), puis le traitement en C++ est fait, d’abord en tentant de détecter un QR Code puis en le lisant s’il a été détecté.

## 5.2 Application feuille blanche

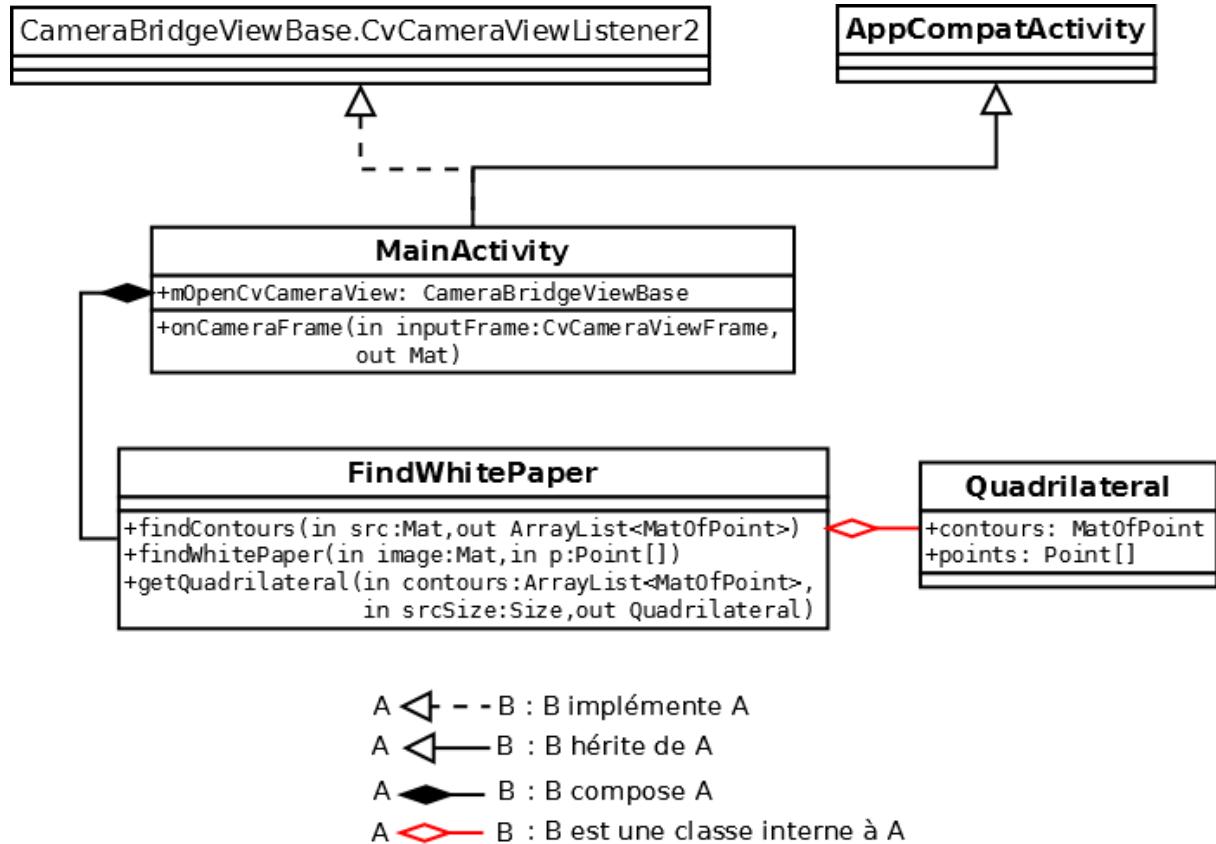


FIGURE 5.4 – Architecture de l’application FeuilleBlanche

Pour cette application, nous avons gardé une architecture très simple par manque de temps et parce que le but n’était pas de fournir une application finie et optimisée. Un schéma de cette architecture est à la figure 5.4. Nous ne respectons pas le *design pattern "Model View Controller"*, mais nous pourrions imaginer le mettre en place de la même façon que dans l’application des QR Codes.

**MainActivity** Ici la classe `MainActivity` s’occupe de gérer la caméra OpenCV en implémentant la classe `CameraBridgeViewBase.CvCameraViewListener2` et la seule activité de notre application. Elle crée également un objet `FindWhitePaper` pour les traitements d’image sur le flux vidéo.

**FindWhitePaper** Cette classe permet de détecter la feuille blanche dans le flux vidéo, ainsi que de recalculer l’image dans ce flux. Dans le cas des lunettes, elle retire également le flux vidéo lorsqu’une feuille blanche est détectée pour n’afficher plus que l’image recalée.

**Quadrilateral** La classe `Quadrilateral` permet simplement de mémoriser un quadrilatère en particulier en gardant l’objet `MatOfPoint` qui correspond à ce quadrilatère que l’on veut trouver ainsi qu’un tableau de `Point` qui sont les quatre coins du quadrilatère.

### 5.2.1 Diagramme de séquence

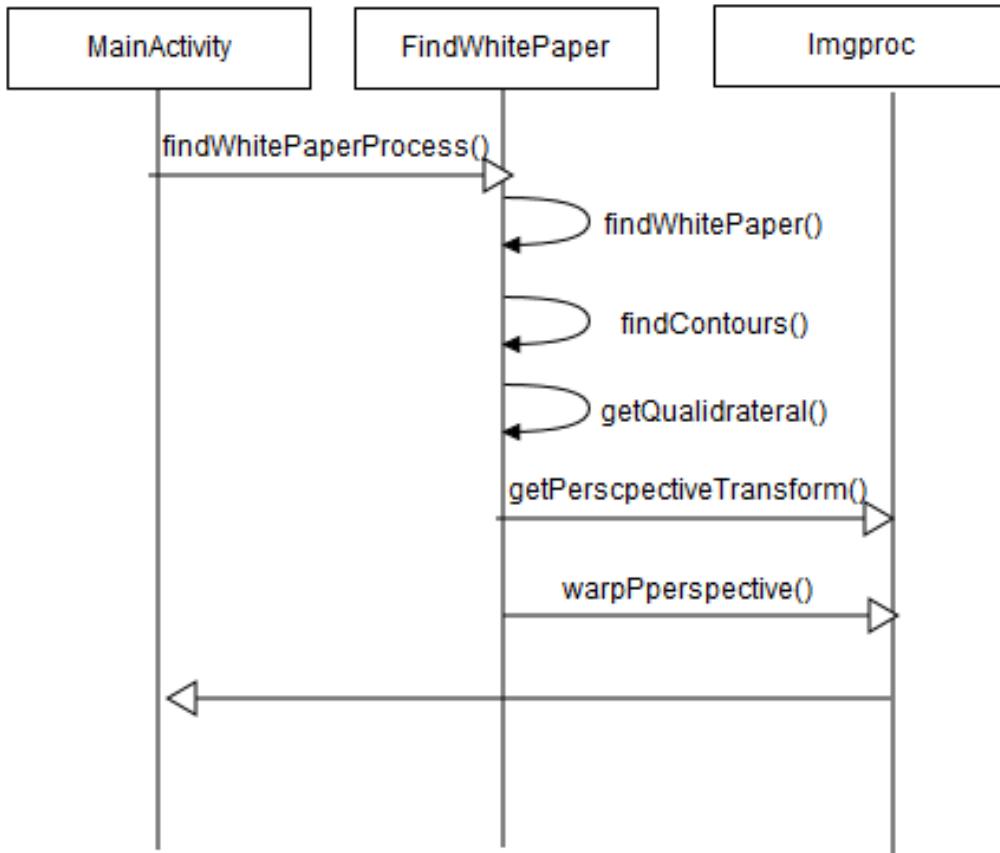


FIGURE 5.5 – Diagramme de séquence pour la détection de la feuille

La figure 5.5 présente un diagramme de séquence concernant la détection de la feuille blanche que ce soit pour l’application Android sous smartphones et tablettes ou pour l’application pour les lunettes Moverio BT-300. Cet enchaînement d’appel de fonctions se fait pour chaque image du flux vidéo.

Tout d’abord, depuis la classe principale de notre application, nous créons un objet de type `FindWhitePaper` dont on appelle la fonction `findWhitePaperProcess(...)` afin d’amorcer la recherche de la feuille blanche. Cette fonction appelle `findWhitePaper(...)` depuis laquelle on va chercher les contours de l’image de la caméra avec `findContours(...)`, une fonction de `FindWhitePaper`, qui, globalement, convertit l’image de la caméra en niveaux de gris et applique un filtre de Canny dessus pour avoir une image de contours. Après cela, elle appelle la fonction `findContours(...)` d’OpenCV qui nous donne accès aux contours. Une fois les contours récupérés, ils sont triés par ordre décroissant de grandeur, puis on appelle `getQuadrilateral(...)` qui permet de tirer un quadrilatère de ces contours, le plus grand possible, et plus particulièrement, les quatre coins de ce quadrilatère. Ces coins nous permettent de calculer la transformation nécessaire à appliquer à l’image qu’on veut afficher sur la feuille blanche, et ce grâce à la fonction `getPerspectiveTransform(...)` du module `Imgproc` d’OpenCV. Il ne nous reste plus qu’à appliquer cette transformation à l’image à afficher grâce à la fonction `warpPerspective(...)` du même module OpenCV.

# Chapitre 6

## Implémentation

### 6.1 Outils, IDEs

#### 6.1.1 Android Studio

Basé sur l'IntelliJ (et non plus sur Eclipse), Android Studio [2] permet de développer des applications Android en mettant à disposition tout un panel d'outils appropriés au développement sur mobile. Il intègre notamment un émulateur de tablettes / smartphones, rendant possible le prototypage sans terminal physique tournant sous Android.

#### 6.1.2 Unity3D

Unity3D [16] est un moteur de jeu multi-plateforme. Il gère le rendu 3D, la physique des objets et offre une interface graphique relativement intuitive pour créer des scènes 3D. Etant très populaire, il existe un nombre important d'extensions que l'on peut intégrer dans des projets, comme par exemple des *bindings* de bibliothèques.

### 6.2 Travail

#### 6.2.1 Prototypage avec Unity3D

Dans un premier temps, nous avons tenté de prototyper une application sur Unity3D avec ARToolKit. Nous avons donc fait une application permettant de voir un modèle 3D de robot lorsqu'un marqueur de type "Hiro" (cf. figure 3.1) était filmé par la webcam de l'ordinateur. Nous avons alors tenté de porter cette application sur un téléphone Android, afin de savoir si nous allions pouvoir travailler facilement avec Unity3D pour réaliser notre projet. Nous avons rencontré de nombreuses difficultés, que ce soit pour prendre en main Unity, ou même pour porter notre application sur Android. De plus, nous avons jugé qu'Unity n'était pas assez souple pour pouvoir modifier le programme original à notre convenance. Nous avons donc décidé de passer sur Android Studio et d'utiliser ARToolKit et OpenCV.

## 6.2.2 Utilisation des marqueurs de type QR Code

### 6.2.2.1 Développement sur téléphone / tablette

**Different supports.** ARToolKit se décline en différents langages et cela sous différentes plateformes. Le *binding* Android est également sous plusieurs déclinaisons. Il est possible de développer tout en Java, mais également de se servir du NDK<sup>1</sup> afin de faire des traitements en C++. Cette possibilité est offerte par Android, et ARToolKit s'en sert pour permettre d'accélérer l'exécution de bouts de code critiques et également pour avoir un contrôle plus bas-niveau.

**Android / Java.** La version Java est l'implémentation la plus haut niveau proposée sur Android. Elle permet de faire rapidement des petites applications pour détecter des marqueurs. Quelques classes permettent aussi de s'éviter le travail fastidieux d'aller récupérer le flux vidéo et ensuite d'afficher des modèles 3D (une classe modélisant un cube est même fournie).

Cependant, cette bibliothèque est bien trop haut niveau pour ce que l'on souhaite faire, le contrôle n'est pas très fin et on est vite limité par les propositions de la bibliothèque. Nous avons donc décidé de passer au développement natif.

**Android / C++.** Il faut savoir que Java propose, en dehors d'Android, un support C++. Cela se nomme la JNI<sup>2</sup>, et il n'est pas aisés de faire communiquer C++ et Java via cette interface. Le principe est de construire des bibliothèques à partir de C++, et de les charger dans les classes Java qui vont se servir de fonctions native.

La possibilité de développer en natif n'avait pas été envisagée au départ du projet, et la prise en main n'a pas été directe. Pour cette raison en partie, certaines tâches qui semblaient faisables au départ dans un temps relativement court, ont pris beaucoup plus de temps que prévu.

**Configuration d'ARToolKit.** La bibliothèque est fournie avec un bon nombre d'exemples, en pur Java et en Java / C++, configurés pour fonctionner directement avec Android Studio. Mais malheureusement, ces exemples ne correspondaient pas aux dernières versions des outils utilisés par l'*IDE*, notamment l'outil de gestion de dépendances Gradle. A cause de cela, les exemples ne fonctionnaient pas directement. Il était possible de corriger cela mais les applications résultantes ne fonctionnaient tout de même pas parce que les bibliothèques faites à partir du code C++ n'étaient pas construites. Cela était sûrement dû à une mauvaise configuration des scripts de compilation, mais ceux-là étant assez complexes, il était difficile de détecter le problème.

À cause de cela, il a fallu apprendre à construire les scripts de compilation pour inclure ARToolKit. Ce travail a été fastidieux, et a énormément retardé les avancées. Les exemples fournis par ARToolKit étaient bien configurés, à l'exception du fichier de configuration de l'application. Un apprentissage en profondeur de l'outil de construction de bibliothèques natives a donc été nécessaire.

---

1. *Native Development Kit*  
2. *Java Native Interface*

**Plusieurs niveaux de contrôle.** ARToolKit propose différents accès aux fonctionnalités sur Android. Il est possible de développer en Java ou en C++ seulement, ou bien de faire un mélange des deux via l'utilisation d'un *wrapper* fourni. L'avantage de l'utilisation de solutions Java, ou Java / C++ est la mise en place relativement aisée des fonctionnalités (accès à la caméra, détection des marqueurs, ...). Mais malheureusement ces deux types de solutions ne sont pas suffisantes pour nos besoins. En effet, pour détecter un QR Code, trois *finder pattern*, visibles figure 6.1, doivent être reconnus et ni la bibliothèque Java ni le *wrapper* C++ permettent de détecter plus d'une occurrence d'un même *pattern*. Pour cela, le niveau le plus bas de contrôle a dû être envisagé, et donc la plupart du code faisant la détection a été écrit en C++.

Les exemples d'ARToolKit ont été d'une grande aide, car après avoir compris pourquoi ils ne fonctionnaient pas et corrigé les problèmes, ils nous ont fournis la logique nécessaire pour détecter les marqueurs. Nous avons donc adapté un exemple pour nos besoins.

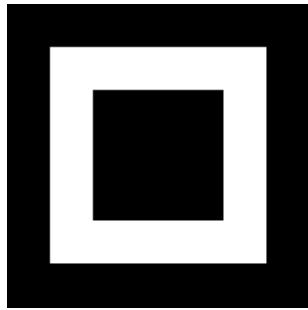


FIGURE 6.1 – *Finder pattern*

**Détection d'un QR Code.** Comme dit précédemment, trois *finder patterns* doivent être détectés pour lire un QR Code. Une fois la limite d'une seule occurrence par marqueur dépassée avec ARToolKit, il a fallu créer un marqueur correspondant à un *finder pattern*. Cela a été assez simple grâce à une application trouvée sur le web [7]. Cette étape n'a donc pas trop posé de problèmes.

Mais les marqueurs classiques d'ARToolKit ont un format particulier, leurs bords prennent une certaine taille sur les marqueurs, et les *finder patterns* ont une largeur de bords différente, comme on peut le voir avec les figures 6.1 et 3.1. Même s'il est possible de modifier la taille des bords par défaut pour qu'ARToolKit considère des bords différents, deux problèmes ont été identifiés, à propos de la création des marqueurs et de la configuration pour que les marqueurs soient reconnus.

**Création des marqueurs.** Comment savoir si les marqueurs étaient correctement créés ? L'application web propose plusieurs formats de marqueurs, de différentes tailles (exprimées en pourcentage), et propose une pré-visualisation avant de créer le marqueur. Même en ayant des paramètres qui donnent des résultats visuellement acceptables, les données renvoyées ne semblent pas fonctionner.

**Configuration de la reconnaissance.** À vrai dire, il est plus probable que la reconnaissance des *finder patterns* ne fonctionne pas à cause de la configuration. Pourtant

en configurant le rapport de la taille des bords en fonction de la taille du marqueur (en format 16x16, 32x32 ou 64x64, les données de marqueurs sont données en nuances de gris par pixel), cela ne fonctionnait pas.

**Passage à OpenCV.** Étrangement, les ressources web concernant l'utilisation d'AR-ToolKit pour reconnaître des QR Codes sont très pauvres. À vrai dire, à part un rapport de projet [3], rien d'autre n'a été trouvé dans ce sens. L'article [15] indique que l'utilisation d'ARToolKit pour ce genre d'application n'est pas pratique, au vu du fait que la bibliothèque utilise ses propres marqueurs.

La page web [12] propose une solution utilisant OpenCV pour détecter et recaler un QR Code face à la caméra, afin de lire son contenu. Le code source étant disponible en ligne, une étude approfondie du code a été faite, afin de recréer le système avec quelques différences de style de programmation et de méthodes pour la détection des coins d'un QR Code.

Pour chaque image traitée, celle-ci est d'abord convertie en niveau de gris avant de se voir appliquer le filtre de *Canny*, pour éviter d'avoir trop de contours détectés par la suite. Une fois les contours détectés, une heuristique est utilisée pour considérer que cinq contours imbriqués correspondent à un *finder pattern*. Cela vient du fait que le filtre de *Canny* fait ressortir cinq (ou six) frontières dans chaque *finder pattern*. Quand trois *finder patterns* sont détectés, un QR Code est considéré présent (figure 6.2).



FIGURE 6.2 – Contours et détection des *patterns* à partir d'une *frame*

**Disposition et orientation.** Un QR Code n'est pas censé se retrouver bien en face de la caméra, il est donc nécessaire de reconnaître les positions relatives des trois *finder patterns* ainsi que l'orientation du QR Code. Pour cela, des calculs de distances entre les *patterns* sont faits, et cela permet notamment de calculer les coins du QR Code.

**Recalage.** En détectant les coins d'un QR Code, il est possible de recaler ce dernier grâce à des fonctions d'OpenCV (figure 6.3). Les contours des *finder patterns* sont approximés, et dès lors que ceux-ci donnent quatre points, des calculs sont faits sur les positions

de ces derniers dans la *frame*, pour pouvoir reconnaître les coins. Dans de rares cas, les coins ne sont pas bien identifiés et le recalage donne lieu à une image illisible pour Zxing.



FIGURE 6.3 – Approximation des bords et recalage

**Lecture du contenu.** Pour lire le contenu du QR Code, nous nous sommes servis de la bibliothèque Zxing [14]. Bien qu’écrite en Java, cette bibliothèque se décline également en C++. Après quelques tests en Java, le développement en C++ a encore été choisi à cause de la contrainte de rapidité. En effet, sans faire de traitements très avancés, une chute importante du *framerate* a été constatée dès lors que Zxing était utilisé. Différentes tailles d’images ont été testées, cela améliorant le *framerate*, mais ce n’était pas assez convaincant. De plus, cela aurait nécessité une communication Java / C++ au niveau des matrices OpenCV, et cela aurait pû être plus compliqué que de tout garder au sein du *wrapper* C++. Le *wrapper* C++ intègre donc également la lecture du QR Code et de ce fait, s’occupe de toute la partie dédiée au traitement des *frames*.

**Développement du *wrapper* et implantation sous Android.** L’avantage d’avoir développé cette partie en C++ est qu’il a été possible de l’écrire sur un ordinateur sans besoin d’un terminal sous Android. La gestion des dépendances étant plus simple sur un ordinateur, la mise en oeuvre technique a été simplifiée (bien que Zxing ait été difficile à intégrer au projet). Il a donc été possible de ne se concentrer que sur l’algorithme, tout en constatant les résultats assez simplement.

De toute évidence, l’implantation sous Android n’allait pas être aussi simple, et cela a été confirmé. L’utilisation d’OpenCV en natif n’a pas été directe, et celle de Zxing encore moins, cette bibliothèque ne disposant pas de fichiers de configuration particuliers pour la lier en natif sous Android. Finalement, un *script* tiré d’un exemple en ligne a été trouvé et adapté pour compiler les fichiers sources de Zxing avec le *wrapper*.

**Affichage d’un modèle 3D.** Cette dernière étape a été commencée dans une application à part, pour apprendre à utiliser OpenGL ES 2.0 sous Android. Pour les mêmes raisons que précédemment, cela a été fait en C++. Ce qui a amené quelques petites complications par rapport à une version Java.

Le travail principal a consisté à écrire des fonctions pour encapsuler les primitives d’OpenGL (création de programmes, compilation des *shaders*, etc...). Du temps a été

consacré également à la gestion des fichiers. Bien qu'il soit possible de donner directement le code source des *shaders*, il est tout de même plus agréable de pouvoir les écrire dans des fichiers. Mais l'accès aux fichiers internes d'une application n'est pas direct en C++, et surtout les primitives pour lire ces fichiers sont assez bas niveaux.

### 6.2.2.2 Développement pour les lunettes Moverio BT-300 d'EPSON

Arrivant au bout de la première moitié du projet sans avoir eu le temps de développer l'intégralité de l'application téléphone, nous nous sommes mis d'accord avec notre client sur le fait que nous ne pourrions pas faire l'application sur les lunettes. Notre client nous a donc suggéré de terminer l'application sur téléphone, et de considérer l'application sur les lunettes comme optionnelle, puisqu'il s'agirait de porter simplement le code. L'application Feuille Blanche a quant à elle été développée sur lunettes, ce qui permet tout de même à notre client d'avoir une première base logicielle sur celles-ci.

### 6.2.3 Travail avec une feuille blanche

Tout d'abord, précisons que pour faire fonctionner l'application sur téléphones et tablettes Android et sur les lunettes EPSON Moverio BT-300, il faudra installer au préalable l'application *OpenCV Manager* [10] disponible sur le Google Play Store. De plus, selon les systèmes, l'application ne demande pas forcément à l'utilisateur si elle peut utiliser la caméra de l'appareil et elle se ferme automatiquement. Pour y remédier il faut autoriser l'application manuellement. Pour cela, il faut aller dans les paramètres du système, dans le gestionnaire d'applications, sélectionner l'application PFEFeuilleBlanche (ou Sample-CameraView), sélectionner "Autorisations" et autoriser l'utilisation de la caméra.

#### 6.2.3.1 Développement sur téléphone / tablette

Le principe du travail avec la feuille blanche est tout d'abord de détecter une feuille blanche, déterminer sa position dans l'espace et son orientation, et afficher dessus une image ayant subi une détection de contours, dans le but de permettre à l'utilisateur de dessiner sur la feuille avec directement un modèle dessus. Pour ce faire, nous avons utilisé la bibliothèque OpenCV qui contient des méthodes que nous pouvons utiliser pour atteindre notre but.

**Détection de la feuille blanche.** La première idée que nous ayons eue a été d'effectuer une binarisation de l'image, puisque notre client nous a précisé que la feuille serait posée sur un fond suffisamment contrasté. Une binarisation permettrait donc d'obtenir une image en noir et blanc (binaire) avec en blanc, seulement la feuille blanche sur laquelle nous allons pouvoir effectuer des opérations (et éventuellement du bruit d'acquisition), et en noir l'arrière-plan et les parties de l'image qui ne nous serviront pas. La binarisation est une opération assez simple et la bibliothèque OpenCV dispose de fonctions optimisées permettant de la faire. Un exemple de binarisation que nous ayons fait se trouve figure 6.4. Pour retirer le bruit, nous avons eu deux idées : tout d'abord un flou Gaussien[18] mais cette fonction d'OpenCV ralenti considérablement les performances. Ensuite, nous avons pensé aux opérateurs morphologiques, en faisant une fermeture (dilatation puis érosion) afin de lisser les contours et de recoller les morceaux de surfaces proches pour fermer les contours disjoints.

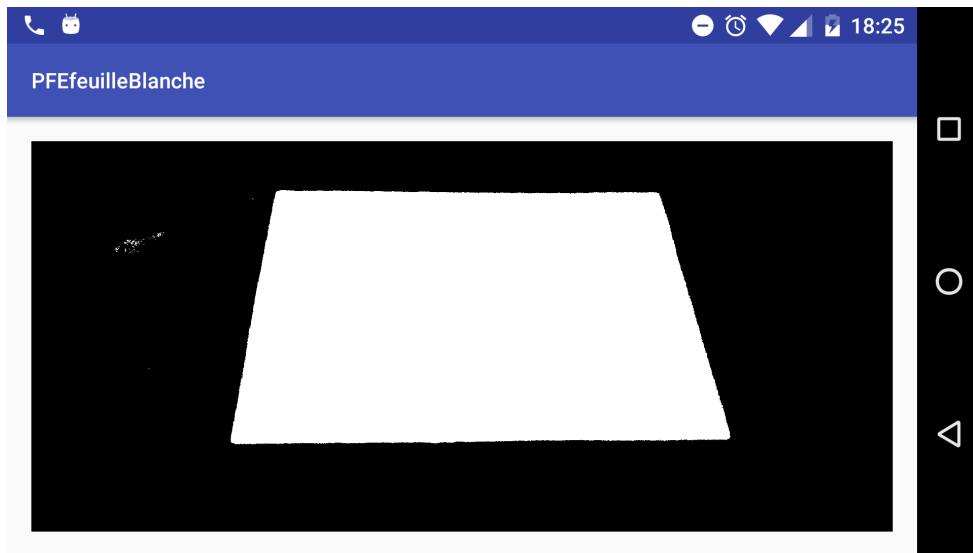


FIGURE 6.4 – *Binarisation du flux vidéo*

**Détection de contours.** On convertit donc notre image en niveaux de gris, puis on la binarise. OpenCV permet également de détecter les contours dans une image grâce à la fonction `findContours(...)`. On demande donc à OpenCV de détecter les contours sur notre image binarisée. On peut observer le résultat figure 6.5. Notre problème avec cette

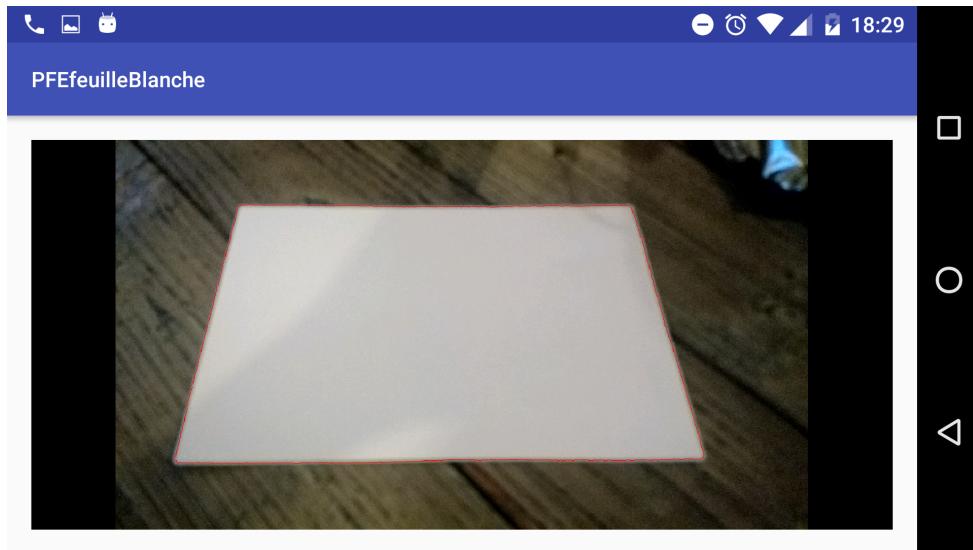


FIGURE 6.5 – *Détection des contours de la feuille*

méthode est qu'elle nous renvoie un nuage de points sur les bords de la feuille et que nous n'avons pas réussi à en tirer des bords droits, malgré l'existence d'une fonction OpenCV (`approxPolyDP(...)`) qui approche des contours par des polygones. Mais les polygones avec lesquelles OpenCV compare les contours sont des polygones avec un nombre quelconque de cotés. Dans notre cas, si un coin de la feuille est occulté ou même simplement corné, nous ne parvenons pas à détecter un quadrilatère mais bien souvent un polygone avec beaucoup plus que quatre côtés. Une solution aurait été de récupérer le polygone

trouvé et de tenter de se rapporter à un quadrilatère. Mais même de cette façon, il aurait été difficile d'être sur que l'on "captait" bien une feuille blanche et non pas une autre zone sur laquelle nous ne voulions pas projeter.

**Détection des coins.** La feuille étant censée être à plat, si on se contente de détecter les coins de la feuille on peut facilement retrouver les bords en reliant les points les uns aux autres. Pour détecter les coins de la feuille nous avons tout d'abord utilisé `cornerHarris(...)` puis `detect(...)` sur un objet de classe `FeatureDetector` d'OpenCV permettant de faire ceci. Cependant, ces fonctions prennent assez de temps et sont plus destinées à des applications qui ne nécessitent pas une réponse très rapide comme on peut en attendre dans notre cas. Il a donc encore fallu trouver une autre méthode.

**Méthode basée sur l'application Scanner.** Nous avons beaucoup bloqué sur la question et multiplié les recherches sur des méthodes nous permettant de faire ce que nous voulions. Finalement, l'idée nous est venue de regarder le code source d'applications Android permettant de scanner des feuilles [8] Ces applications ont un algorithme qui détecte la feuille blanche depuis le flux caméra. De plus, elles appliquent une déformation sur la feuille pour en faire une image rectangulaire. C'est exactement la transformation inverse qui nous intéresse ici : on veut déformer une image rectangulaire pour la faire coïncider avec la déformation de la feuille sur laquelle on veut afficher l'image. Afin de détecter les coins de la feuille, nous nous sommes directement inspirés du code d'une de ces applications. Afin de bien visualiser où sont les coins détectés par le programme, nous avons marqués ces points d'une croix rouge comme on peut le voir sur la figure 6.6. Cette méthode consiste en fait à appeler la fonction `findContours(...)` qui se trouve dans `FindWhitePaper.java`. Cette fonction fait simplement appel à la fonction `findContours(...)` d'OpenCV après avoir fait quelque pré-traitements sur l'image comme un changement de taille, un passage au niveau de gris et un filtre Canny [19].

Ensuite, les contours trouvés sont triés du plus grand au plus petit. On parcours un à un ces contours dans la fonction `getQuadrilateral(...)` et on les approche par un polygone avec la fonction `approxPolyDP(...)` d'OpenCV. Sitôt qu'un contour est considéré comme un quadrilatère on le stocke dans un objet `Quadrilateral` et on arrête la recherche. `approxPolyDP(...)` permet de ne retenir que les points situés à des angles du contours. Ce qui, dans notre cas, nous fait 4 points. Il ne nous reste plus qu'à identifier les points pour savoir lequel est en haut à droite, en haut à gauche, en bas à gauche et en bas à droite. Nous obtenons donc les quatre sommets du quadrilatère trouvé comme on peut le voir figure 6.6.

**Orientation et position de la feuille blanche.** Nous nous sommes rapidement rendus compte qu'il n'était pas forcément nécessaire de savoir l'orientation de la feuille puisqu'il s'agit simplement de déformer l'image à appliquer sur la feuille et de la positionner au bon endroit. OpenCV dispose d'une fonction appelée `getPerspectiveTransform(...)` et `warpPerspective(...)` permettant respectivement de trouver cette déformation puis de remplacer la feuille blanche dans l'image. Il suffit donc d'appeler ces méthodes et de repositionner l'image d'arrivée sur la feuille blanche dans le flux de la caméra. Cependant nous n'avons pas réussi de suite à recaler l'image sur le flux vidéo. Nous obtenions bien une déformation de l'image que nous voulions afficher qui correspondait à la déformation de la feuille blanche mais l'arrière-plan apparaissait noir comme on peut le voir sur la

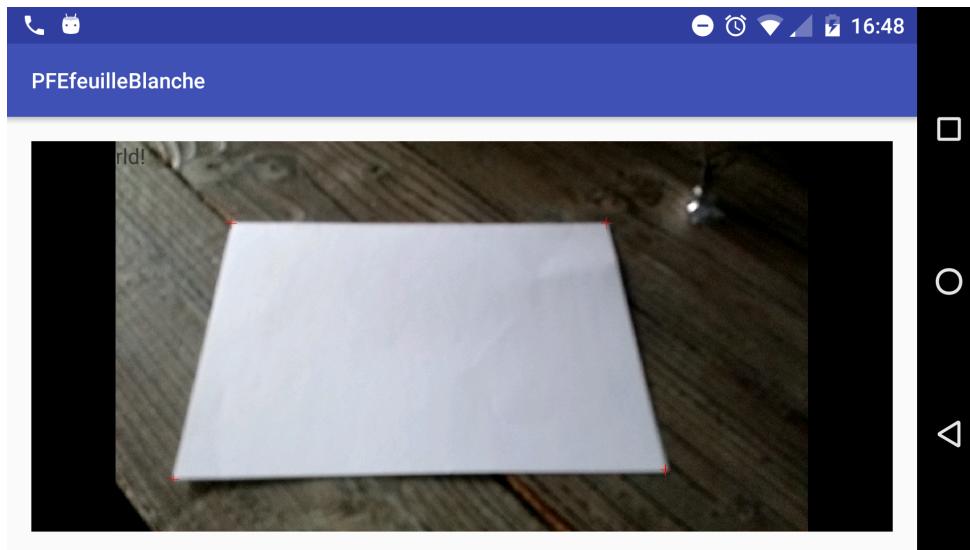


FIGURE 6.6 – Détection des coins

figure 6.7. Notons que sur cette figure l'image qui apparaît n'est pas encore une image de contours. Cette fonctionnalité à été implémentée après cette capture d'écran.

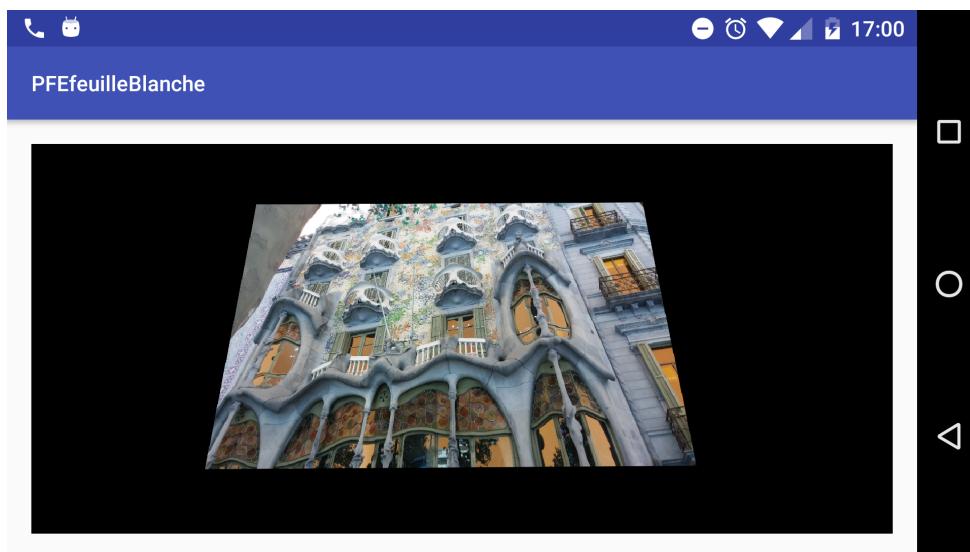


FIGURE 6.7 – Affichage de l'image sur la feuille blanche sans flux vidéo

Nous avions donc du noir à la place du flux vidéo. Nous avons eu l'idée de se servir de cette image comme d'un masque, afin de la recopier dans l'image du flux vidéo mais sans le noir. Après un `copyTo(...)`, on obtient bien le flux vidéo de la caméra avec notre image déformée comme si elle était à la place de la feuille blanche, comme on peut le voir sur la figure 6.8.

#### 6.2.3.2 Développement pour les lunettes Moverio BT-300 d'EPSON

**Portage du code.** Nous avons eu beaucoup de mal à créer une application s'exécutant sur les lunettes, à cause de toutes les dépendances découlant des EPSON Moverio BT 300.



FIGURE 6.8 – *Affichage de l'image sur la feuille blanche avec flux vidéo*

Pour gagner du temps, et puisque notre but était de créer des briques de base logicielles, nous avons fait le choix de reprendre un des exemples qui étaient fournis avec le SDK des lunettes intitulé "Sample Camera View".

Nous avons donc réussi à porter notre code en remplaçant celui qui était dans l'exemple par le nôtre, et en y apportant quelques modifications. Nous disposons donc à ce moment là d'une application détectant les coins de la feuille et remplaçant cette feuille par une image exactement comme le fait l'application pour téléphones et tablettes Android.

**Adaptation aux lunettes.** Les lunettes de réalité augmentée ont pour particularité de superposer des informations virtuelles à la vision de l'utilisateur. L'intérêt d'utiliser notre application pour téléphones et tablettes Android directement sur les lunettes Moverio BT-300 d'EPSON est assez limité. En effet, afficher le flux vidéo de la caméra par dessus la vision de l'utilisateur qui est censé voir sensiblement la même chose n'est pas vraiment utile et peut même être dérangeant pour l'utilisateur.

C'est pourquoi notre objectif était de n'afficher que l'image sur la feuille blanche, sans le flux vidéo. De plus, nous devions faire en sorte que l'affichage de l'image sur les écrans des lunettes se superpose bien à la feuille blanche que l'utilisateur voit de ses propres yeux.

**Differences d'affichage entre les lunettes et les téléphones/tablettes.** Sur les lunettes Moverio BT-300 d'EPSON, tout ce qui apparaît en noir ne sera pas visible par l'utilisateur. Pour éviter d'afficher le flux vidéo dans les lunettes il nous a donc suffi d'afficher l'arrière-plan en noir. Nous pouvons observer le résultat obtenu sur la figure 6.9. Sur cette capture d'écran nous pouvons voir que l'interface a été allégé par rapport à la version téléphones et tablettes. En effet la barre de navigation ainsi que la barre d'outils ont disparu afin que l'utilisateur n'ait pas la sensation d'avoir un affichage conséquent en face de lui. Seule la barre de notification subsiste, ceci pour plusieurs raisons. Cela permet à l'utilisateur de pouvoir consulter ses notifications à tout moment ; cela lui permet également de rester au fait du niveau de la batterie des lunettes ; enfin, cela lui permet

aussi de connaître la largeur de la zone sur laquelle on pourra afficher l'image sur la feuille. Cependant après plusieurs essais de notre part, nous nous sommes rendus compte qu'il était difficile de savoir si la caméra voit la feuille blanche sans voir nous-même le flux vidéo de la caméra. Nous avons donc décidé d'afficher le flux vidéo de la caméra lorsque la caméra ne détecte pas la feuille blanche, et de le supprimer lorsque la feuille est détectée.

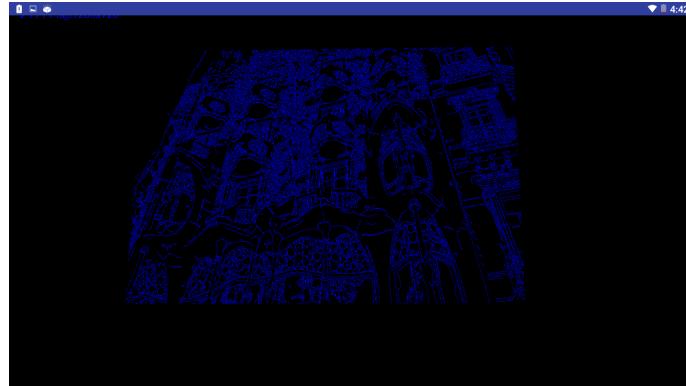


FIGURE 6.9 – Affichage de l'image sur la feuille blanche sur lunettes

**Recalage de l'image sur la feuille réelle.** Notre problème désormais est le décalage entre l'image que l'on affiche et la réalité que l'on perçoit à travers les lunettes ainsi que la mise à l'échelle de l'image. Pour inclure cette fonctionnalité nous devons faire face à plusieurs problèmes. Tout d'abord, un problème dû à la gestion de la caméra par OpenCV, qui nous renvoie un zoom de ce qui est réellement perçu par la caméra. Ceci induit donc une diminution du champ de vision de la caméra qui a pour conséquence de réduire la taille des feuilles blanches utilisées dans un cadre de dessin, et de réduire également la zone de détection de la feuille blanche. De plus les écrans des lunettes ne couvrent pas un très grand angle de vision (environ  $23^\circ$  selon EPSON [5]). Ceci implique que pour pouvoir recaler l'image sur la feuille blanche, il faudra que la perception de la feuille par l'œil de l'utilisateur rentre dans l'écran d'affichage des lunettes. Si ce n'est pas le cas, lors du décalage de l'image, seule une partie de l'image affichée sur la feuille sera visible comme on peut le voir sur la figure 6.10 qui représente simplement un décalage du flux vidéo de la caméra.

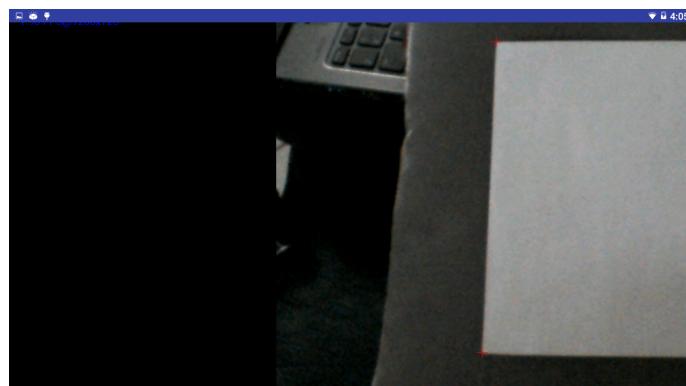


FIGURE 6.10 – Problème lié à l'angle de vision des écrans des Moverio BT-300 d'EPSON



# Chapitre 7

## Tests

Nous souhaitions soumettre un questionnaire simple afin d'avoir une première idée concernant les sensations perçues par l'utilisateur, comme la fluidité, la précision du recalage. A cause des délais très courts, nous n'avons pas eu le temps de faire tester nos applications à d'autres personnes qu'aux membres du groupe. Les tests perceptifs sont donc basés sur le seul jugement de ces membres du groupe et ne sont donc pas fiables malgré l'objectivité des membres sur leur travail. Ils permettent tout de même d'avoir une première idée de la qualité de nos applications.

### 7.1 QR code

Pour cette partie, les tests ont été faits sur une tablette Asus ME173 X (Processeur : MTK MT8125 / 8389 1.5 GHz, RAM : 1024 Mo).

▷ Intitulé du test : Calculer le *framerate*.

Protocole de test :

Utiliser le système natif d'OpenCV sous Android qui détermine le nombre de FPS d'une application.

Résultats :

Sans affichage 3D par-dessus les QR Codes, l'application fonctionne à 15 FPS en moyenne.

Conclusion :

L'utilisation du NDK laissait présager de meilleures performances, mais sachant que la machine de test ne fait que du 20 FPS en moyenne sans traitement, le résultat n'est pas si alarmant.

▷ Intitulé du test : Reconnaissance des QR Codes.

Protocole de test :

Calculer le taux de lecture des QR Codes en fonction du nombre d'images traitées.

Résultats :

Avec une bonne luminosité et sans bouger la tablette, l'application reconnaît et lit environ 10% du temps les QR Codes. Ceci est donc une borne supérieure.

Conclusion :

Ce résultat semble satisfaisant, mais un travail plus approfondi sur l'algorithme de détection pourrait permettre de systématiquement de bien recaler les QR Codes et ainsi éliminer les cas limites où les bords ne sont pas bien identifiés.

- ▷ Intitulé du test : Déterminer la taille minimale des QR Codes que l'on peut traiter.

Protocole de test :

Pour un QR Code donné, et à une distance fixe, réduire la taille du QR Code progressivement jusqu'à qu'il ne soit plus reconnu.

Résultats :

A une distance d'environ 30cm du QR Code, la taille d'un pixel contenu dans le QR Code ne doit pas être inférieure à 1.5mm.

Conclusion :

Avec des images de meilleure qualité, cette limite pourrait être (théoriquement) repoussée.

- ▷ Intitulé du test : Tester la reconnaissance en fonction de la luminosité.

Protocole de test :

Soumettre chacun des algorithmes de détection à des luminosités variées, telle des lumières de différentes couleurs. Baisser la luminosité au fur et à mesure pour déterminer le seuil à ne pas dépasser pour que les algorithmes fonctionnent correctement.

Résultats :

Seules les luminosités importantes permettent d'obtenir des résultats, dès lors qu'il y a de la pénombre, les QR Codes ne sont plus correctement lus.

Conclusion :

Un travail de recherche particulier sur ce point est nécessaire. La gestion de la luminosité semble être un problème récurrent dans ce genre d'applications.

## 7.2 Feuille blanche

- ▷ Intitulé du test : Tester la taille minimale et maximale des feuilles que notre application peut détecter avec les lunettes et le téléphone.

Protocole de test :

Nous testerons la détection de feuille blanche avec une feuille de format A4, A5 et A6, sur un Samsung Galaxy A5 2016.

Résultats :

	A4	A5	A6	A7
Feuille bien détectée	oui	oui	oui	oui

Conclusion : La feuille est bien détectée quel que soit son format. Cependant, nous devons être assez proche de la feuille pour qu'elle soit détectée. Elle doit donc occuper une grande partie du flux vidéo, mais être totalement visible par ce dernier.

Les figures 7.1 7.2 7.3 7.4 montre les résultats de la détection respectivement pour des formats de feuille A4, A5, A6 et A7. Notons qu'à chaque fois la feuille doit occuper assez de place dans le flux de la caméra pour pouvoir être détectée.

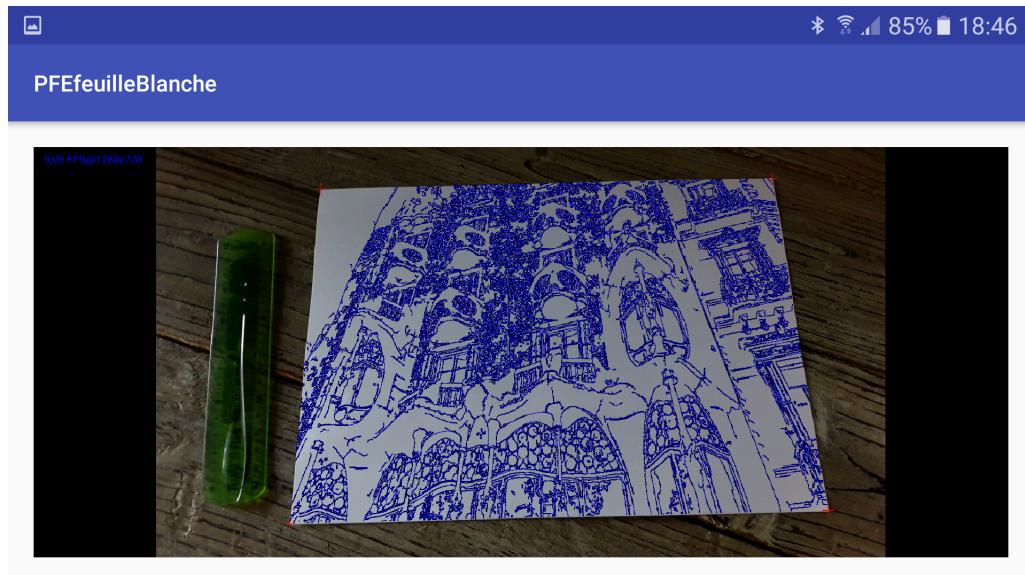


FIGURE 7.1 – Résultats de la détection avec une feuille A4

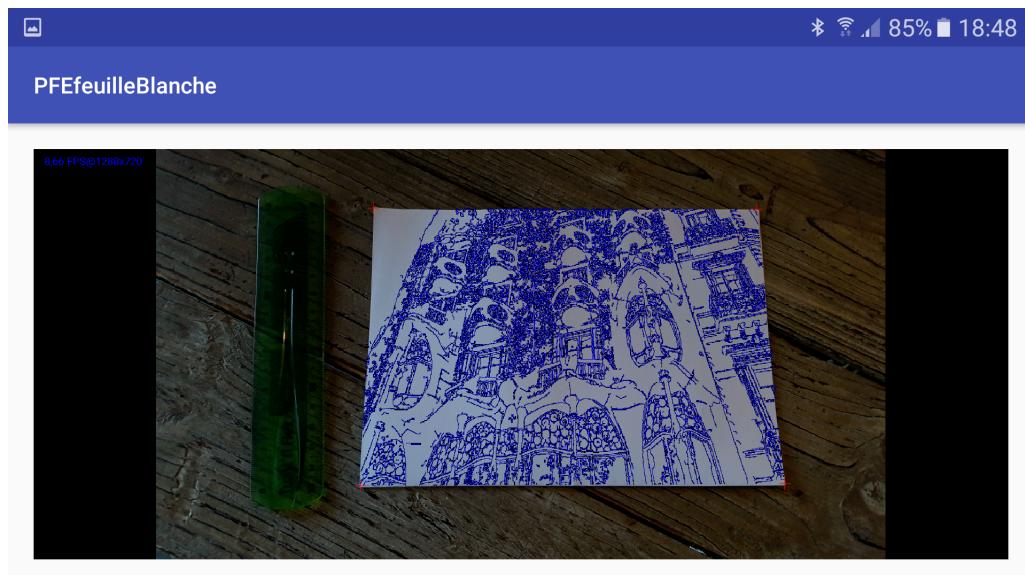


FIGURE 7.2 – Résultats de la détection avec une feuille A5

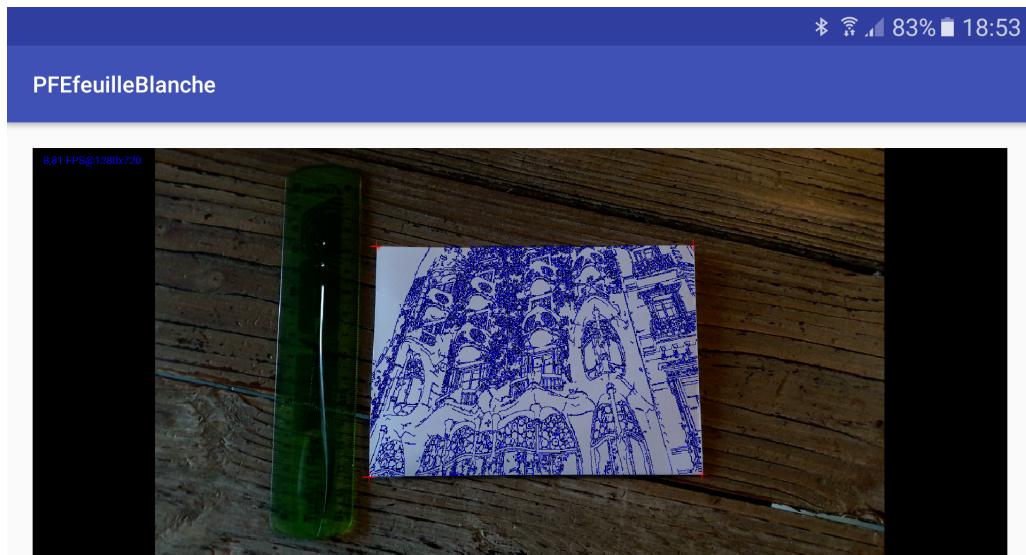


FIGURE 7.3 – Résultats de la détection avec une feuille A6



FIGURE 7.4 – Résultats de la détection avec une feuille A7

- ▷ **Intitulé du test :** Tester la robustesse aux changements d'inclinaison de la feuille avec les lunettes et le téléphone.

#### Protocole de test :

Nous testerons la détection de feuille blanche avec une inclinaison nulle, puis nous tenterons de trouver la limite qui ne nous permettra plus de détecter la feuille blanche.

#### Résultats :

Il nous a été difficile d'établir un protocole de test nous permettant d'avoir des chiffres à l'appui. Il est difficile d'estimer les angles avec lesquels on regarde la feuille. Sur-

tout qu'il faut déterminer trois angles à chaque fois puisque la caméra peut tourner autour de la feuille selon trois dimensions. Néanmoins la figure 7.5 montre un cas de figure extrême. En accentuant encore l'angle de vision de la caméra, la feuille ne sera pas détectée.

Conclusion : Des tests plus précis et plus approfondis seront nécessaires pour pouvoir approuver ou non que l'application est efficace sur ce point. On peut cependant affirmer qu'il vaut mieux être un peu au-dessus de la feuille pour la détecter plutôt que sur les côtés.

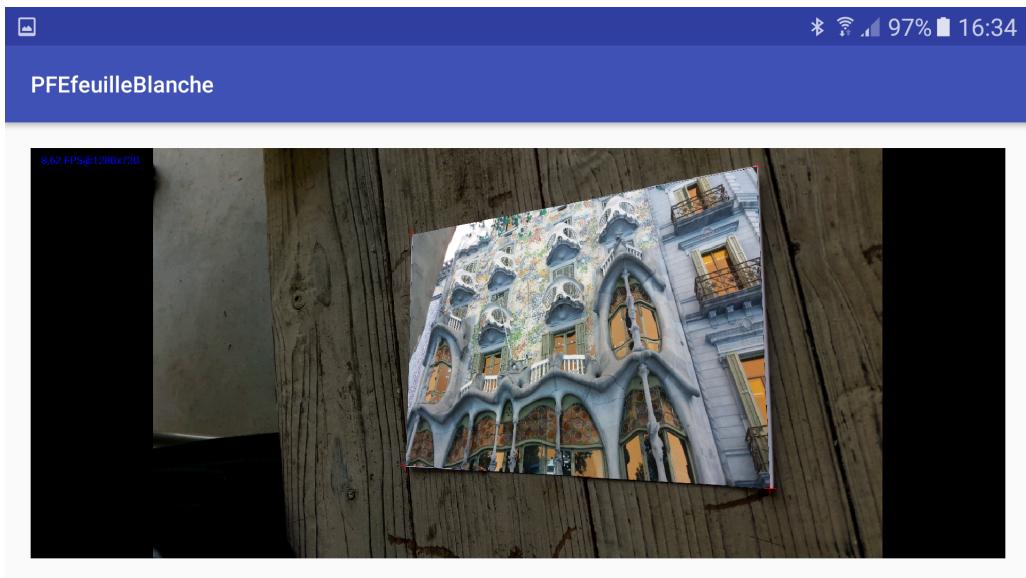


FIGURE 7.5 – Exemple d'un cas limite pour l'angle de vision de la caméra

▷ Intitulé du test : Calculer le *framerate*.

Protocole de test :

Utiliser le système natif d'OpenCV sous Android qui détermine le nombre de FPS d'une application.

Résultats :

	Motorola Moto X 2ème G	Samsung Galaxy A5 2016	Asus ME173X	Epson Moverio BT300
FPS	9	10,5	9	7

Le résultat est calculé lorsque la feuille blanche est détectée. Il est dépendant du matériel, mais en général nous nous retrouvons à 9,5 FPS. De manière générale, la feuille est plus souvent détectée sur le Samsung Galaxy A5 2016 que sur le Motorola Moto X 2ème génération. Si la feuille blanche n'est pas détectée, nous tournons autour de 23 FPS, nous allons même parfois jusqu'à 29 FPS.

Conclusion :

Le besoin non-fonctionnel concernant la rapidité d'exécution n'a pas été tenu. Le

développement avec OpenCV est assez coûteux en temps de calcul, et Java n'est pas réputé pour être un langage rapide. Peut-être qu'en redéveloppant soit-même certaines fonctions d'OpenCV, et en les adaptant à nos besoins très spécifiques, la contrainte pourrait être tenue.

▷ **Intitulé du test : Tester la reconnaissance en fonction de la luminosité.**

Protocole de test :

Soumettre l'application à des luminosités variées, telle des lumières de différentes couleurs. Baisser la luminosité au fur et à mesure pour déterminer le seuil à ne pas dépasser pour que les algorithmes fonctionnent correctement.

Résultats :

De manière générale, l'application est assez sensible aux changements de luminosité. Il faut que la feuille soit bien éclairée, sans ombres trop dures la traversant, et éclairée par une lumière la plus blanche possible.

Conclusion :

Notre application n'est pas très robuste aux changement de luminosité. Nous pourrions imaginer une étude de l'histogramme et un réhaussement de contraste avant de détecter la feuille blanche.

▷ **Intitulé du test : Estimer la qualité du recalage de la feuille blanche sur les lunettes.**

Protocole de test :

Lancer l'application et estimer la qualité du recalage selon différents angles de vue. Se poser deux questions :

- Avons-nous l'impression que l'image est bien sûr la feuille blanche ?
- Si non, à combien estimons-nous le décalage entre l'image et la feuille blanche ?

Résultats : Nous n'avons pas eu le temps de faire tester cette application par des personnes ne faisant pas partie de la réalisation de ce projet. L'avis suivant est le nôtre et ne peut donc pas être considéré comme un résultat fiable. Cependant, nous jugeons bon, voire très bon, le recalage de l'image dans le flux vidéo pour les smartphones et tablettes. Néanmoins, on peut parfois voir très légèrement la feuille blanche autour de l'image intégrée. En ce qui concerne les lunettes de réalité augmentée, le recalage est très limité par le matériel lui-même comme on va le voir dans la partie suivante.

Conclusion : On ne peut pas conclure quant à la qualité de l'intégration de l'image sur la feuille puisque les avis évoqués ci-dessus ne sont que les nôtres et se limitent donc à trois avis. Une chose est sûre cependant : pour les lunettes le résultat n'est pas là et serait difficilement atteignable compte tenu des caractéristiques matérielles des Moverio BT-300 d'Epson.

# Chapitre 8

## Bilan / perspectives

Une grande partie de ce projet a consisté à prendre en main les différentes technologies, puis les différentes bibliothèques. Il a été particulièrement porté sur la technique, et un peu moins sur l'algorithmique. Nous avons été confrontés à plusieurs problèmes, retardant ainsi les avancées attendues.

### 8.1 Limites des applications

#### 8.1.1 QR codes

La contrainte de vitesse a sûrement été le besoin le moins respecté. Cette contrainte a vraiment été une raison importante du développement en C++ de certains traitements. Comme dit juste avant, cela a apporté son lot de complications, et d'heures de configuration. Nous avons pu constater grâce à cela que le développement d'applications en réalité augmentée était assez difficile en pur Java.

#### 8.1.2 Feuille blanche

##### 8.1.2.1 Téléphones et tablettes

Lorsque la main de l'utilisateur passe sur la feuille, la détection de la feuille blanche ne se fait plus. Le même soucis apparaît quand une ombre trop dure vient sur la feuille. Nous ne tenons pas compte du ratio, donc si une feuille carrée est détectée, l'image re-projectée sera déformée. De plus, la projection de l'image ne suit pas forcément les rotations de la feuille blanche. C'est-à-dire que si la feuille est détectée en mode portrait, l'image projetée sera en mode portrait également même si c'est une image en mode paysage comme on peut le voir figure 8.1. Tout comme pour les QR Codes, l'application feuille blanche sur smartphones et tablettes n'atteint pas les 20 FPS.

##### 8.1.2.2 Lunettes EPSON Moverio BT-300

Plus que des problèmes, ce sont des limites de performances que nous rencontrons ici. Et ces limitations techniques nous ont fait nous rendre compte de certains problèmes dans le choix ou l'utilisation des algorithmes de notre application. En effet, le problème du décalage, évoqué section 6.2.3.2 dans le paragraphe sur le recalage de l'image sur la feuille réelle, pourrait être évité si on utilisait une feuille plus petite, ou si l'utilisateur s'éloignait de la feuille ; mais la détection ne fonctionne que lorsque la feuille occupe une

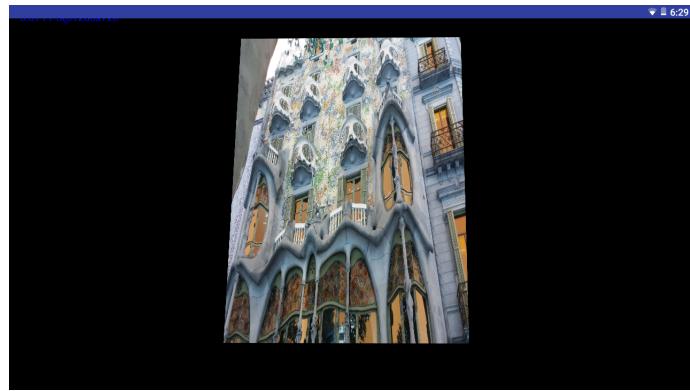


FIGURE 8.1 – *problème d'utilisation de la feuille en mode portrait avec une photo en mode paysage*

partie importante de ce que voit la caméra. Sur la figure 8.2, le cadre violet montre la feuille telle que l'utilisateur la perçoit avec ses yeux ; le cadre bleu, représente la zone sur laquelle les lunettes vont pouvoir afficher des informations supplémentaires ; le cadre gris représente les lunettes elles-mêmes. On voit bien sur le schéma que la feuille réelle ne rentre pas forcément entièrement dans la zone bleue et on ne peut donc pas recaler toute l'image sur la feuille. Précisons que ce schéma n'est pas à l'échelle et qu'en réalité, la zone bleue occupe bien moins le champ de vision de l'utilisateur.

Nous avons également remarqué que l'application était très sensible au type de lumière utilisée et aux ombres. Si la feuille n'est pas totalement visible, elle n'est pas détectée, ce qui empêche toute interaction entre l'utilisateur et la feuille. Nous avons donc cherché d'autres moyens, plus robustes, de détecter la feuille, sans succès. Il se pourrait que les problèmes que nous venons de citer soient dus au zoom que fait OpenCV en gérant la caméra. En effet, ceci engendre une diminution de la résolution de l'image sur laquelle on travaille et peut empêcher la détection de la feuille. Pour le moment, nous n'avons toujours pas trouvé le moyen de retirer ce zoom.

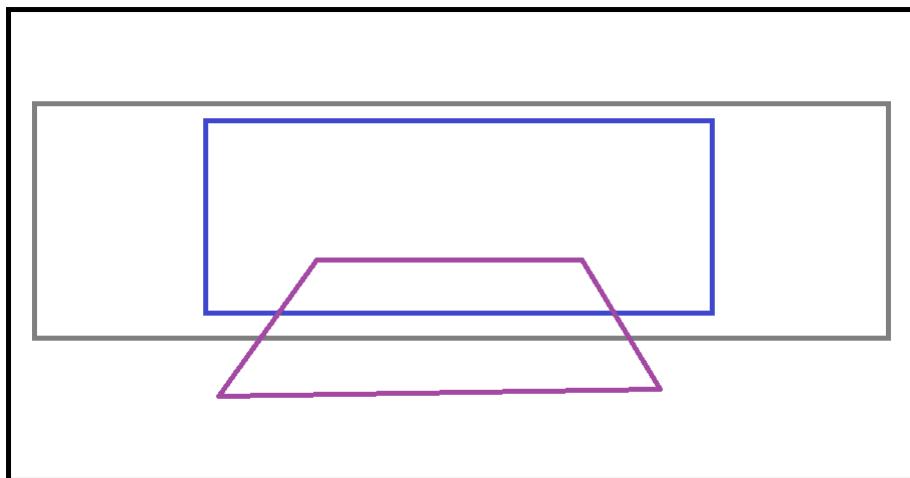


FIGURE 8.2 – *Illustration du problème lié au champ de vision des lunettes*

## 8.2 La technique

### 8.2.1 Le matériel.

L'utilisation d'appareils externes à nos ordinateurs de travail est bien souvent une difficulté importante. En effet, ce sont des technologies qui demandent du temps à apprêhender, à bien comprendre, avant de pouvoir espérer faire quelque chose avec. Dans notre cas, l'utilisation d'Android sur tablettes / téléphones et sur des lunettes a demandé d'entrer dans l'environnement Android, de bien intégrer son fonctionnement, pour ensuite faire nos applications. Les lunettes ayant un fonctionnement différent des tablettes / téléphones, bien que sous Android, ont demandé plus de configuration pour être utilisées.

### 8.2.2 Les bibliothèques.

L'utilisation de bibliothèques externes à Android a rajouté une source de complications supplémentaires. La configuration d'OpenCV sous Android n'a pas été faite sans mal, et celle d'ARToolKit non plus. De plus, à cause de la contrainte de rapidité, OpenCV a été utilisé partiellement en C++ dans l'application QR Code, et l'utilisation de C++ au sein d'une application Android amène de la configuration supplémentaire. Nous n'avons pas pu travailler avec ArToolKit à cause de la limitation de la bibliothèque sous Android. L'utilisation du SDK des lunettes a aussi pris plus de temps que prévu. Puisque le but était d'explorer la programmation sur ces lunettes, nous avons décidé de ne pas passer plus de temps sur la configuration et nous avons repris un exemple fourni avec cette SDK. Nous avons ensuite remplacé le code dans cette application par le notre.

## 8.3 Remarques

### 8.3.1 Note sur les performances de Java et C++

Affirmer que C++ est plus rapide que Java est maintenant une chose qui ne se démontre plus. En effet, tout programmeur sait que la JVM<sup>1</sup> ajoute un surcoût à l'exécution de ses programmes, malgré l'utilisation d'un langage intermédiaire (*bytecode*) qui permet des optimisations. D'ailleurs, la présence de la JNI montre bien que les développeurs du langage Java voulaient permettre d'améliorer les performances des applications codées avec leur langage, en proposant la possibilité de coder certaines parties d'un logiciel en C++.

Pourtant ici, nous sommes dans un contexte différent, car le développement d'applications embarquées (n'oublions pas que le matériel des terminaux Android est quand même généralement moins puissant que celui d'un ordinateur classique) amène son lot de divergences, comme tous les problèmes de configuration dès lors que l'on cherche à utiliser des bibliothèques externes.

Il nous faut considérer ici le coût (plus que probable) de la communication entre Java et C++ au sein de l'application, surtout que les traitements en C++, et donc les appels aux fonctions natives, sont faits à chaque image. Une étude empirique [6] démontre avec quelques exemples simples que les traitements faits en C sont généralement plus rapides

---

1. *Java Virtual Machine*

que ceux faits en Java, mais cette différence tend à disparaître au fur et à mesure des versions d'Android.

Il ne faut pas oublier qu'ici, nous travaillons avec une bibliothèque (OpenCV) qui, en Java, propose des fonctions qui ne sont que des *wrappers* des fonctions natives déjà écrites en C++ (OpenCV est une bibliothèque écrite en C/C++ à la base). S'il y a un coût lors d'une communication entre Java et C++, développer une application OpenCV en Java seulement va engendrer plus de communications entre Java et C++ que si l'on fait toute la partie traitement exclusivement en C++.

### 8.3.2 Note sur la taille des applications

Développer une application en Java avec OpenCV ne donne pas lieu à des applications de tailles importantes ( $\sim$ 5-6 Mo par application). OpenCV fonctionne à la manière d'une bibliothèque classique sur un ordinateur, à savoir qu'il faut télécharger une application, nommée OpenCV Manager [13] et ensuite le *loader* d'OpenCV se charge de chercher la bibliothèque. Cela signifie qu'il faut prendre garde à ne pas utiliser OpenCV avant la fin du chargement (par exemple en créant des matrices trop tôt). Cependant, il est également possible d'inclure toutes les bibliothèques statiques proposées par OpenCV directement dans l'application, mais cela augmente drastiquement la taille de l'application ( $\sim$ 70 Mo par application).

Pour le développement natif, il est nécessaire d'inclure au moins une bibliothèque dynamique et le résultat revient au cas précédent, les applications prennent beaucoup de place. Étonnamment, l'utilisation de Zxing n'engendre pas une augmentation importante de la taille des applications, elles restent autour de 70 Mo. Ce genre de paramètres semble important à prendre en compte tout de même, car la mémoire peut parfois s'avérer être une ressource limitante dès lors que l'on fait du développement embarqué, même s'il est assez commun de nos jours d'étendre la mémoire de nos terminaux par des dispositifs externes (telles les cartes SD).

## 8.4 Les améliorations possibles

### 8.4.1 Général

**Activités.** Les applications ne comportent qu'une seule activité chacune. Cela signifie que seul le traitement de la vidéo est faite. Il serait imaginable de rajouter un menu de paramétrage de la caméra ou une activité se chargeant de calibrer la caméra. Pour pouvoir présenter le produit sous une seule application, il faudrait intégrer les deux applications en une seule en créant une activité intermédiaire pour choisir le type de détection (parce que traquer à la fois les feuilles blanches et les QR Codes pourraient s'avérer très lourd).

### 8.4.2 QR Codes

**Détection des coins.** Bien que ce point ait été amélioré au cours du développement, il reste quelques cas limites où les coins sont mal identifiés. Améliorer ce point demanderait

d'abord une évaluation de la méthode actuelle pour juger de la pertinence de passer du temps à gérer ces quelques cas.

**Tracking.** Lire un QR Code consomme forcément du temps, et en rechercher un nécessite un parcours entiers des *frames*. En exploitant la cohérence d'un flux vidéo, il serait envisageable de *tracker* un QR Code pour améliorer les performances en ne parcourant qu'une sous-partie des *frames*, et également éviter de lire son contenu à chaque fois. Même si le *tracking* n'a pas été implémenté, il est déjà possible de préciser dans le code, si l'on veut lire ou pas le contenu du QR Code.

**Intégration de l'application OpenGL.** L'étape qui suivait la lecture de la détection d'un QR Code était d'afficher un modèle 3D dessus en fonction du contenu. L'application OpenGL qui a été développée a servi essentiellement à voir comment l'on faisait du rendu 3D en natif sous Android. Maintenant il faudrait intégrer cette application à celle détectant les QR Codes et arriver à calculer la position 3D du QR Code pour afficher un modèle 3D sur le QR Code.

**Modèles 3D via l'internet.** Pour afficher un modèle 3D dépendant du contenu du QR Code, il y a deux solutions envisageables : soit l'on met le modèle dans le QR Code, ce qui conduirait sûrement à des QR Codes assez gros, soit l'on indique une URL dans le QR Code et l'on place à cette URL un modèle 3D. Cette dernière voie semble plus intéressante, pour pouvoir garder des QR Codes de tailles raisonnables. Il faudrait alors apprendre à récupérer des ressources web sur Android, et en C++ si possible.

**QR Codes multiples.** Ceci n'a pas du tout été abordé dans le projet, mais une amélioration importante serait de rajouter la détection multiple de QR Codes. Pour cela, il faudrait trouver des propriétés sur les *finder patterns* pour déterminer comment ils forment des QR Codes. Des hypothèses sur les angles peuvent être une piste à explorer. Si le calcul n'est pas trop coûteux, un passage en 3D des positions des *finder patterns* pourrait servir grandement, pour exploiter ces hypothèses avec la troisième dimension (mentionnons par exemple les angles à 45° et 90° entre les *finder patterns*).

#### 8.4.3 Feuille blanche

**Robustesse.** Pour l'instant nous n'avons pas de "textittracking", et ce pourrait être une solution envisageable pour améliorer notre application. En effet, la feuille n'est pas toujours détectée sur toutes les frames, pour diverses raisons. En suivant la feuille au lieu de la détecter à chaque fois, nous n'aurions plus de *clipping*. Il faudrait pour cela suivre les quatre coins de la feuille blanche.

**Sélection d'une image ou prise en direct.** Une autre amélioration serait de pouvoir permettre à l'utilisateur de choisir sa propre image à recaler, ou bien lui permettre d'en prendre une en direct.

**Programmer les traitements d'image en C++.** Nous avons pu voir avec les QR Codes que développer en natif certaines fonctionnalités permettait d'accélérer certains calculs. Une solution pour augmenter le *framerate* serait de développer le traitement d'image fait sur le flux vidéo en natif.

**Stéréoscopie.** Une dernière amélioration serait de profiter des capacités des lunettes au maximum, c'est à dire se servir des deux écrans. Nous pourrions donc effectuer le recalage de la feuille blanche sur chacun des écran séparément, afin d'avoir réellement l'impression que l'image est bien recalée, même au niveau de la profondeur.

# Conclusion

Pour conclure sur le travail fourni, il peut sembler que les objectifs ne sont pas atteints. Mais la plupart des objectifs non atteints sont dus à des limites de puissance des matériels cibles ou à des limites physiques du matériel, notamment pour les lunettes. L'absence d'application QR Code sur les lunettes Moverio BT-300, le faible nombre de FPS de nos applications, nous donne la sensation de ne pas avoir réussi ou travaillé efficacement. Mais il est important de rappeler que pour notre client, l'important n'était pas d'avoir des applications fonctionnelles, mais plutôt d'avoir des briques logicielles de base qu'il pourrait réutiliser, chose que nous avons réussi à faire. De plus, les limites matérielles rencontrées ne sont pas exclues pour notre client. Notamment pour les lunettes, qui permettent certes de réaliser des applications de réalité augmentée mais le fait de ne pas pouvoir adapter l'application feuille blanche sur celles-ci montre que ce ne sont pas des lunettes de réalité mixte. On ne peut pas vraiment ajouter de l'information visuelle et donner l'impression à l'utilisateur que cette information est incrustée dans son environnement. Cela est dû principalement au faible champ de vision des lunettes qui n'est que de 23° alors que la vision humaine a un champ de vision horizontal d'environ 180°. On comprend aisément qu'il est compliqué, surtout avec une simple caméra 2D pour analyser l'environnement, d'intégrer de l'information visuelle autour de l'utilisateur. La réalité mixte n'est donc pas envisageable sur ce type de lunettes mais peut très bien l'être sur d'autres, comme les Hololens de Microsoft [9] par exemple.

# Bibliographie

- [1] ANDROID API, 2017. <https://developer.android.com/index.html>.
- [2] ANDROID STUDIO, 2017. <https://developer.android.com/studio/index.html>.
- [3] AR ON QR, *Augmented reality above qr codes*, 2017. <http://www.cs.technion.ac.il/cggc/Upload/Projects/Aviya%20Levi%20-%20AR%20on%20QR/ARonQR-project%20book.pdf>.
- [4] ARTOOLKIT, 2017. <http://artoolkit.org/>.
- [5] EPSON, *Moverio bt-300 - epson*. <https://www.epson.fr/products/see-through-mobile-viewer/moverio-bt-300>.
- [6] JAVA VS C APP PERFORMANCE, *An empirical comparison between Java and C performance*, 2016. <http://www.androidauthority.com/java-vs-c-app-performance-689081/>.
- [7] MARKER GENERATOR, 2017. <http://flash.tarotaro.org/blog/2009/07/12/mgo2/>.
- [8] MATTKHAW, *Opennotesscanner*, 2017. <https://github.com/ctodobom/OpenNoteScanner/tree/master/app/src/main/java/com/todobom/opennotesscanner>.
- [9] MICROSOFT, *Hololens*. <https://www.microsoft.com/microsoft-hololens/fr-fr>.
- [10] OPENCV, *Opencv manager*. <https://play.google.com/store/apps/details?id=org.opencv.engine&hl=fr>.
- [11] ——, *Open source Computer Vision*, 2017. <http://opencv.org/>.
- [12] OPENCV + QR CODE, *OpenCV : QR Code detection and extraction*, 2017. <http://dsynflo.blogspot.fr/2014/10/opencv-qr-code-detection-and-extraction.html>.
- [13] OPENCV MANAGER, *About*, 2017. <https://play.google.com/store/apps/details?id=org.opencv.engine&hl=en/>.
- [14] QR CODE DECODER, 2017. <https://github.com/zxing/zxing>.
- [15] K. RUAN AND H. JEONG, *An augmented reality system using qr code as marker in android smartphone*, in Engineering and Technology (S-CET), 2012 Spring Congress on, IEEE, 2012.

- [16] UNITY3D, 2017. <https://unity3d.com/>.
- [17] VUFORIA, 2017. <https://www.vuforia.com/>.
- [18] WIKIPÉDIA, *Gaussian Blur*, 2016. <https://en.wikipedia.org/wiki/Gaussian.blur>, consulté le 22/03/2017.
- [19] ——, *Filtre de canny*, 2017. [https://fr.wikipedia.org/wiki/Filtre\\_de\\_Canny](https://fr.wikipedia.org/wiki/Filtre_de_Canny), consulté le 19/03/2017.
- [20] WIKITUDE, 2017. <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>.