ENSEEIHT

Telequid

Projet de fin d'études

Reconnaissance d'images sur smartphones

Vincent Angladon
IMA 2013

Tutrice ENSEEIHT :
Mme Géraldine Morin

Tuteur entreprise :
M. Benjamin Ahsan

Mars - septembre 2013

Table des matières

1 Remerciements

Je voudrais remercier en premier lieu mon encadrant, Benjamin Ahsan qui s'est très impliqué dans la réussite de mon projet et m'a apporté son soutien technique avec l'aide de Julien André, co-encadrant. Cette aide m'a été grandement précieuse pour rentrer dans les projets existants de l'entreprise, ainsi que l'apprentissage et le perfectionnement de mes connaissances en ObjectiveC et GWT entre autre, technologies que j'ignorais avant le début de mon stage.

Je remercie également toute l'équipe LIGUM du laboratoire DIRO de l'université Polytechnique de Montréal et plus particulièrement Pierre Poulin qui m'a accueilli et encadré durant mon séjour au Canada. Je n'oublierai jamais l'accueil très chaleureux que j'ai reçu au Québec.

Je souhaite exprimer ma gratitude à Frédéric Bruel, patron de Telequid qui m'a accordé sa confiance, recruté au sein de la société et a veillé au bon déroulement du projet.

Je remercie Géraldine Morin, ma tutrice technique de l'ENSEEIHT, qui au travers de réunions ponctuelles s'est occupée de veiller au bon déroulement du stage, à l'amélioration de mon rapport et de ma soutenance et m'a permis de rejoindre l'équipe LIGUM avec l'aide de Vincent Charvillat.

2 Introduction

2.1 Présentation de Telequid

Telequid est une société de R&D spécialisée dans les technologies transmedia qui peuvent être utilisées dans des applications de télévision sociale, reconnaissance visuelle et la publicité interactive. L'équipe technique est constituée de deux ingénieurs Enseeihtiens promo 2010 : Benjamin Ahsan et Julien Andre. Frédéric Bruel, fondateur et directeur général de la société est également un ancien diplômé de l'ENSEEIHT.

La société compte parmi ses clients de grands groupes tels que JC Decaux, France Télévision, TF1, et Orange.

Un certain nombre d'applications mobile ont été développées au sein de Telequid :

- iTagCode, un démonstrateur permettant de faire de la reconnaissance d'images, de QR codes et de codes barres. L'application rendue obsolète par uCheck a été retirée de l'App Store d'Apple ;
- uSnap déclinée en versions iOS et Android est une application lancée en partenariat avec la société JC Decaux. Cette application permet de reconnaître un certain nombre d'affiches publicitaires déployées par JC Decaux et d'y associer un contenu multimédia. Usnap repose sur une technologie de reconnaissance d'images robuste développée en interne chez Telequid. L'application peut également reconnaître les QR codes et les codes barres ;
- uCheck disponible sur Google Play et l'App Store d'Apple, est le démonstrateur dernière génération du moteur de reconnaissance d'images de Telequid. À la différence de uSnap, le mode de capture est continu, c'est-à-dire qu'il n'est pas nécessaire d'appuyer sur un bouton pour prendre une photo qui sera analysée. L'application permet également de reconnaître des vidéos ;
- alloTV est un guide de programmes TNT rapide et simple, disponible uniquement sur iPhone. L'application intègre un classement des programmes par acteurs et par popularité, un système de recommandation de programmes, des infos sur les films, les acteurs, un moteur de recherche pour retrouver un acteur ou un programme, des rappels Push, et permet de tweeter sur les émissions en temps réel ;
- iTag est une application second écran pour iPad permettant de consulter un programme télévisé enrichi. L'application reprend les fonctionnalités d'AlloTV et dispose également de fonctionnalités pour partager le programme en cours, recevoir des recommandations de programme TV basées sur ses goûts, de reconnaitre la chaîne en cours ; iTag est en cours d'expérimentation par d'importants opérateurs telecom et groupes media.

Fig. 1 : *Captures d'écrans de uSnap, iTag v1 et alloTV développées chez Telequid.*

La société propose une large palette de services et sdk, dont :

- iRead qui est la technologie de reconnaissance d'images utilisée par uSnap et uCheck ;
- iSyncTV, une technologie de reconnaissance audio permettant de déterminer la chaîne TV regardée ;

- iZapTV permettant de partager les 30 dernières secondes de TV live regardées sur Twitter et Facebook ;
- iSyncTV qui est un service de reconnaissance de publicités télévisées proposant de la publicité interactive.

2.2 Présentation du sujet

2.2.1 Le contexte

La reconnaissance visuelle d'images sur smartphone est un domaine en pleine effervescence. Depuis Google Goggles qui a été la première application grand public populaire à offrir ce service, on trouve aujourd'hui une pléthore d'applications permettant de reconnaître des tableaux dans des musées, des couvertures de livres, des pochettes de CD ou de DVD, des publicités...[1]. De nombreux magasins comme Kiabi, But, IKEA, Leroy Merlin[2] enrichissent déjà leur catalogues avec du contenu destiné aux smartphones accessible par reconnaissance d'images.

2.2.2 Les enjeux

Telequid proposait dans son catalogue actuel un certain nombre de solutions de reconnaissance d'images destinées à des annonceurs souhaitant lier par exemple des affiches publicitaires à un contenu web.

L'annonceur pouvait effectuer les actions suivantes :

- Ajouter et retirer des images à reconnaître regroupées sous forme de campagnes
- Associer et modifier les actions effectuées sur le téléphone après reconnaissance de l'image

L'application de reconnaissance d'images doit donc pouvoir se synchroniser avec les souhaits de l'annonceur.

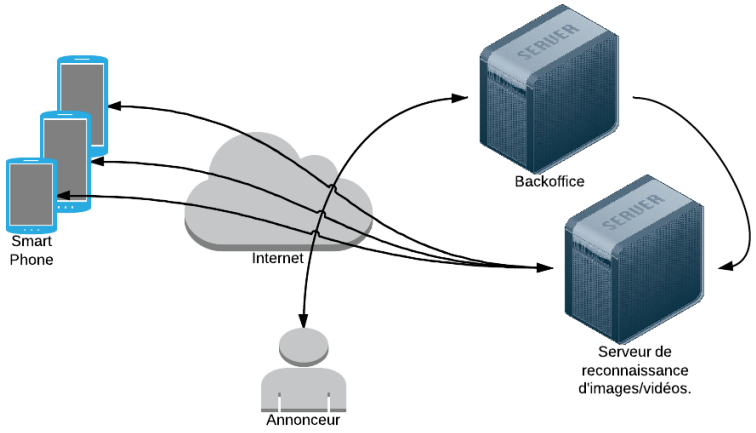

Fig. 2 : *Diagramme des interactions utilisateurs de uCheck : l'annonceur peut créer des campagnes d'annonces depuis le back-office qui notifie le serveur de reconnaissance des nouvelles images à reconnaître ou celles à rajouter. Les téléphones se synchronisent avec ce dernier interlocuteur.*

Afin d'éviter que l'utilisateur du smartphone ait à installer une application par annonceur, certains annonceurs sont regroupés au sein d'une même base de données. Chaque application mobile est donc spécifique à une base de données et peut reconnaître les images de chacun des annonceurs partageant la même base de données.

Ces solutions reposaient toutes sur la même architecture :

[1] https ://www.recognize.im/site/showcaseApps
[2] http ://www.moodstocks.com/gallery/

- une application qui envoie à un serveur les images issues de la caméra ;
- un serveur qui s'occupe du traitement de l'image, et renvoie au téléphone l'image reconnue ainsi qu'une action associée à cette image telle que l'ouverture d'une page web, le lancement d'une vidéo ...

L'objectif de mon stage était de trouver des solutions, permettant de reporter le plus de calculs possibles sur un iPhone. Autrement dit, d'évaluer les solutions permettant de faire de la reconnaissance d'images sur un iPhone, sans communiquer avec un serveur central, à l'exception d'une phase d'initialisation où une synchronisation des images à reconnaître et des actions associées aux reconnaissances sont effectuées. Bien que de très rares concurrents proposaient déjà des solutions de reconnaissance d'images en local, il n'était pas certain que les technologies utilisées sur les serveurs de Telequid pour uCheck puissent être portées sur mobile, faisant de ce sujet un projet de recherche et développement.

L'avantage de reporter les calculs sur le téléphone est de pouvoir effectuer de la reconnaissance d'images sans accès Internet et de réduire les coûts de fonctionnement par la suppression du serveur central. On élimine également la crainte de saturer la charge d'un serveur, dit autrement, les téléphones n'ont plus besoin d'être bridés sur le nombre d'images à traiter par seconde. Nous verrons par la suite comment ces spécifications ont évolué au fur et à mesure du déroulement de mon projet de fin d'études.

2.2.3 La reconnaissance d'images

On distingue quatre problèmes voisins, classés du plus proche au plus éloigné de notre sujet :

la reconnaissance c'est l'identification d'objets prédéfinis au sein d'une image. Un algorithme de reconnaissance fera la différence entre une assiette de cresson et une assiette de mâche, alors qu'un algorithme de catégorisation aura pour finalité de placer ces deux images dans la même classe ;

la recherche d'images par le contenu (content-based image retrieval) qui a pour finalité de retrouver des images dans une base de données en utilisant une autre image comme requête (la réponse est un ensemble d'images) ;

la détection : savoir si un objet appartenant à une catégorie donnée se trouve sur une image et à quel endroit. Pour garantir la vie privée des utilisateurs, Google a mis en place un algorithme de détection des visages et des plaques d'immatriculations sur sa plateforme Google Street View. Il ne s'agit pas de reconnaissance puisque le but n'est pas d'authentifier les différents visages détectés. Les algorithmes de détection sont pour la plupart basés sur du *machine learning*. On citera notamment la méthode de Viola et Jones *(cascade of features)* ;

la catégorisation ou classification qui consiste à attribuer une classe ou une étiquette à une image donnée. Par exemple, elle indiquera si une image donnée est plutôt de type « plage » ou « montagne »... La catégorisation retourne systématiquement un résultat (une classe) pour toute image avec une complexité temporelle souvent inférieure aux algorithmes de détection. Certes, on pourrait placer chaque affiche dans une catégorie distincte, et utiliser un algorithme de catégorisation mais cela ne serait pas judicieux dans notre cas étant donné que le cahier des charges impose un taux d'erreur très faible.

Quelle que soit la méthode, on peut remarquer la présence d'un fossé sémantique, c'est-à-dire que la représentation utilisée par l'ordinateur (descripteurs locaux par exemple) pour la reconnaissance n'a aucun rapport avec la sémantique de l'image, c'est à dire ce que perçoit le cerveau humain. Notamment dans le cadre de notre application, on ne cherche pas à détecter la présence d'un rectangle avant de retrouver la bonne affiche.

Dans notre cadre, il s'agissait uniquement de **reconnaissance**.

2.2.4 Le cahier des charges

L'objectif de notre étude était de concevoir une application capable de reconnaître visuellement des photos pouvant être des affiches publicitaires et de les suivre *(tracking)* ou d'afficher du contenu associé à l'image reconnue. Les contraintes suivantes devaient être respectées :

- compatibilité avec iOS 5.0 ou avec la version 9 du SDK d'Android ;
- un temps de réponse de la reconnaissance d'images inférieur à une seconde sur un iPhone 4 et sur un Nexus 4
- un tracking fluide (10 Hz minimum) ;
- une reconnaissance des images robuste aux changements d'échelle, d'éclairage, ainsi qu'aux occlusions partielles ;
- un taux d'erreur de la reconnaissance minime (moins de 1 pour 3000 tests).

Le nombre d'images maximal à reconnaître n'était pas une contrainte. L'objectif était de pousser les solutions à leurs limites et d'évaluer celles qui permettaient de reconnaître un nombre maximal d'images.

A contrario, le respect des taux d'échec était une contrainte forte qui a déterminé certains de mes choix.

3 Organisation du travail

3.1 Planning

Un premier planning prévisionnel avait été ébauché par mon encadrant, laissant deux semaines au début du stage pour la prise en main de l'environnement Apple, l'apprentissage de l'objectiveC, la conception d'application pour iOS, le développement en GWT et les recherches. Les deux mois suivants devaient être consacrés au développement et à l'évaluation d'algorithmes de calcul de points d'intérêt et de descripteurs optimisés pour des plateformes mobiles. Puis deux mois étaient réservés à la conception d'une structure de donnée permettant de comparer rapidement les descripteurs des images à reconnaître. Les mois restants devaient être consacrés aux autres parties de la pipeline de reconnaissance ainsi qu'à la réalisation éventuelle d'un back-office.

Voici un résumé des différentes étapes effectives du stage (jusqu'à la rédaction du rapport) :

Semaines	Étape / Jalon
1 à 6	Recherches, conception de prototypes pour iOS et Android gérant la reconnaissance et le tracking.
7 à 8	Appui sur divers projets internes.
9 à 10	Optimisation des paramètres de la reconnaissance jouant sur la performance.
11 à 13	Création d'un SDK, design et conception d'un moteur de reconnaissance modulaire. Réflexion sur l'architecture globale du système, début de conception du serveur calculant les points d'intérêt des images à reconnaître et du back-office
14 à 15	Améliorations, modifications et tests de uCheck pour iOS et Android. Fin de conception du back-office.
17 à 21	Recherches et tests sur la réalité augmentée urbaine. Conception d'une version GLSL de l'algorithme de calcul de flot optique avec la méthode de Lucas-Kanade.
23	Corrections de bugs et tentatives d'optimisations du tracking sous iOS
24	Création du SDK du produit fini et de l'application de démonstration.
25	Conception d'un plan de test du SDK, tests, obfuscation du SDK. Analyse de la montée en charge du serveur de reconnaissance d'images uCheck.

3.2 Méthodologie

En réalité, avec l'arrivée d'une version stable d'OpenCV avant le début de mon stage, la partie sur le calcul des points d'intérêt et la structure de recherche des descripteurs fut terminée la seconde semaine. C'est ainsi que la réalisation d'un prototype pour Android, de recherches sur le tracking, ainsi que d'autres tâches se sont greffées au sujet.

La méthodologie de développement retenue était une méthode agile. Tous les jours, un point rapide était effectué avec mon encadrant où on discutait des tâches effectuées, de celles restantes. C'était également l'occasion de parler des difficultés rencontrées et des différentes solutions de contournement envisageables.

3.3 Organisation technique

Afin de garder une trace datée de mes résultats pour la réalisation de mon rapport et avoir un support lors des points quotidiens effectués avec mon encadrant technique, j'ai tenu un journal de bord sous la forme de fichiers textes non formatés où j'inscrivais toutes les tâches effectuées, les nouvelles idées, les difficultés, les résultats, des notes techniques, les publications lues ...

J'ai également utilisé des dépôts Git pour tenir un historique des changements effectués au sein des nombreux projets sur lesquels j'ai travaillé. L'outil Git m'a grandement aidé pour parcourir le passé des projet, notamment pour retracer l'origine de bogues, versionner des changements indépendant via l'utilisation de branches et fusionner les différentes versions du moteur de reconnaissance d'images, utilisé et adapté par l'application iOS, l'application Android et le serveur traitant les images du back-office. L'utilisation de cette hygiène de développement m'a permis d'avancer sans

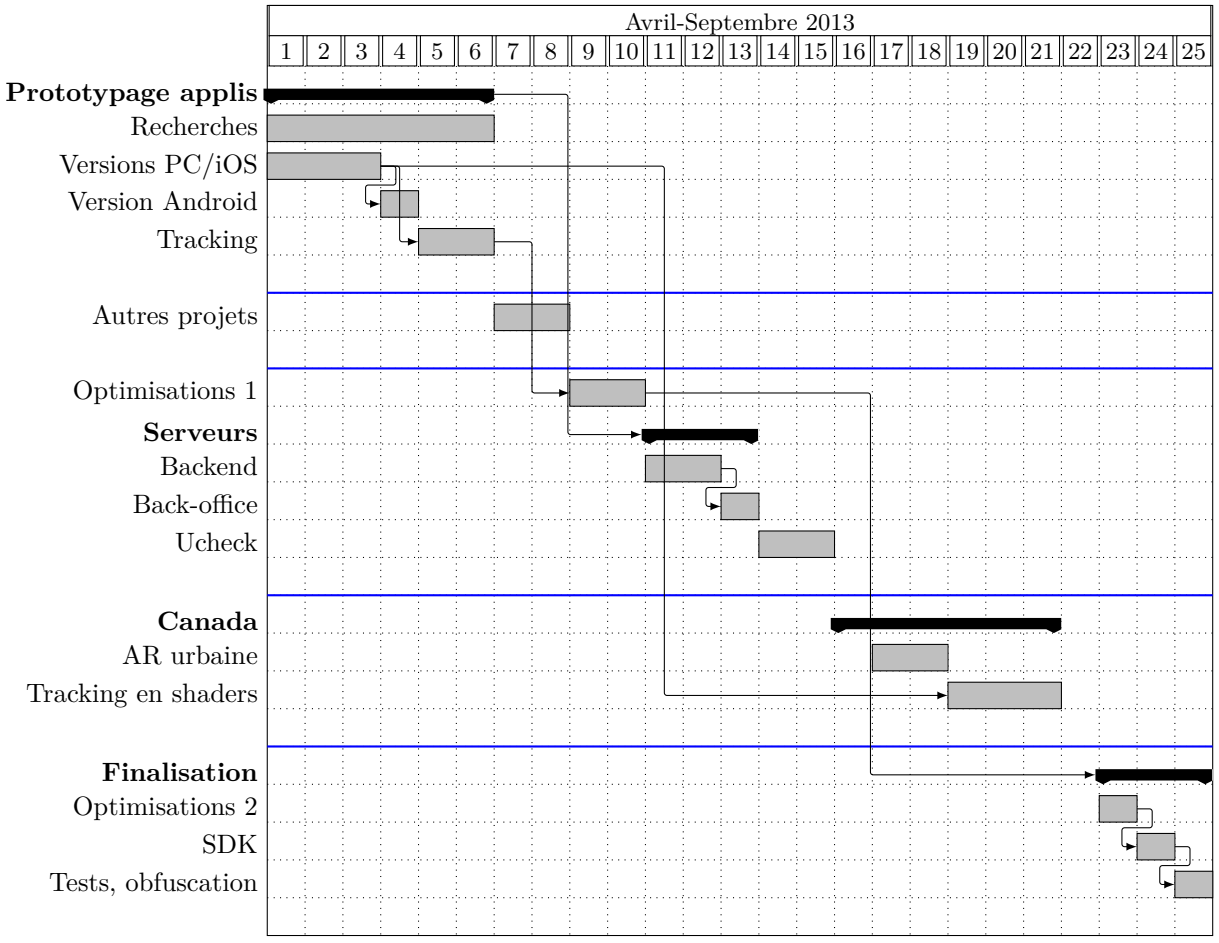

Fig. 3 : *Planning effectif.*

craintes de pertes de changements. Git était plus adapté pour cette tâche que Subversion enseigné à l'ENSEEIHT et utilisé pour versionner l'ensemble des projets de Telequid. En effet, avec svn, les commits sont noyés dans la masse de l'ensemble des projets de l'équipe, et la récupération des changements ainsi que le versionnage passe constamment par le réseau, ce qui est peu pratique.

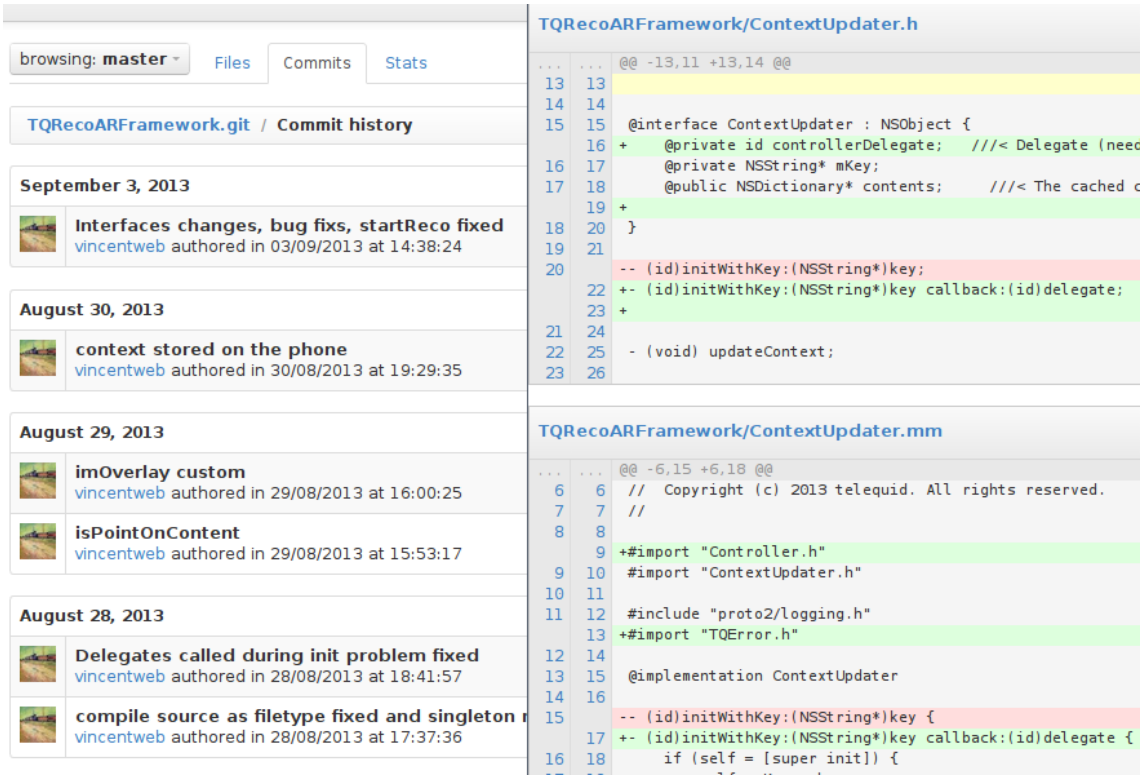

FIG. 4 : *Capture d'écran de GitList, installé sur un serveur distant pendant mon stage pour suivre visuellement les évolutions de mes dépôts et avoir des sauvegardes incrémentales de mes projets.*

4 La reconnaissance d'images sur smartphone

4.1 Les solutions concurrentes

Une des premières missions que j'ai réalisée a été la recherche des solutions concurrentes, la comparaison des services et prix proposés, ainsi que l'analyse de leurs technologies et performances.

Il existe un nombre important de solutions commerciales concurrentes offrant des services de reconnaissance d'images que je résumerai succinctement. La plupart d'entre-elles proposent seulement une architecture de type REST (où la reconnaissance de l'image est déportée sur un serveur). On citera Recognize.im, Pongr, Kooaba, Cortexika et Google Goggles. Enfin, un nombre assez restreint propose un sdk permettant de faire de la reconnaissance d'images directement sur le téléphone. On citera IQEngine, Moodstocks et Smartsy.

Durant cette analyse, j'ai eu l'occasion de mettre en pratique mes connaissances en analyse forensique afin de révéler les bibliothèques utilisées par les solutions concurrentes et leurs choix technologiques. J'ai utilisé divers outils pour extraire les applications des téléphones, convertir et extraire les fichiers de l'application, décompiler, visualiser les symboles des bibliothèques, extraire les chaînes, séparer les architectures cibles … Les résultats de cette analyse ont permis de me conforter dans les choix effectués pour la réalisation du prototype, ainsi que de découvrir de nouvelles bibliothèques.

4.2 Les appareils cibles

L'étude des caractéristiques techniques des appareils cibles a été déterminante dans certains choix de spécification et de conception tels que : les études de faisabilité, les langages de programmation, les frameworks, les options de compilation, …

4.2.1 La famille iOS

La famille iOS était le système d'exploitation prioritaire pour mes tests. D'une part, parce qu'il représente une grand nombre d'utilisateurs répartis sur un faible nombre d'appareils, et d'autre part pour des raisons internes. Le marché est faiblement segmenté, mais l'iPhone5 et les tablettes possèdent des ratio de résolutions d'écran différentes obligeant à développer parfois des interfaces spécifiques à certains modèles.

Les iPods étaient exclus des appareils cibles ainsi que les modèles ne pouvant être mis à jour vers iOS 5, soit les iPhone 1, 3G et 3GS. Ces derniers ont d'ailleurs quasiment disparu et sont complètement dépassés en terme de performances. iOS 5 apportait des fonctionnalités clefs comme un support complet pour l'ARC, le module Core Image, GLKit (surcouche OpenGL ES), … Enfin tous les appareils compatibles iOS5 ont le bon goût d'être compatibles OpenGL ES 2.

L'iphone 4 constitue le modèle le moins puissant de tous les appareils cibles ainsi que de ma plateforme de tests. Il est équipé d'un processeur Cortex-A8 800 MHz, mono-coeur, supportant les instructions NEON, d'une RAM de 512 Mo et d'un GPU PowerVR SGX 535 cadencé à 200 MHz. Il possède également l'ensemble des instruments de mesure nécessaires pour effectuer de la réalité augmentée urbaine sur capteurs, à savoir un gyromètre trois axes, un accéléromètre, un magnétomètre et un GPS. Sa caméra au ratio natif 4 :3 peut prendre des photos en 2592x1936 de résolution maximale. Elle possède un champ de vision horizontal de 61° et 47.5° vertical.

iOS a été conçu de façon similaire à OS X. On retrouve ainsi le même noyau (XNU), la même ABI (Mach-O), le même framework (Cocoa), le même outil de développement (XCode). Le langage de programmation utilisé pour écrire des applications est l'objectiveC, un langage orienté objet, dont la syntaxe particulière rappelle celle de SmallTalk : les appels de méthodes se font par passage de messages. Il est également possible d'utiliser l'ObjectiveC++ qui permet d'intégrer du code C++ dans du code ObjectiveC. Les applications peuvent éventuellement être développées en HTML5 et Javascript, mais le code généré est alors public, et il n'est pas possible d'accéder à toutes les fonctionnalités bas niveau ni de faire du calcul de façon performante. De plus le "look and feel" de l'application sera web et non pas iOS. Néanmoins ce choix est très populaire au sein de la communauté des développeurs d'applications mobiles. Enfin, les équipes de Mono (qui ont

développé une machine virtuelle .NET multi-plateforme ainsi qu'un compilateur C#) ont lancé Xamarin, un produit permettant de développer en C# (un langage de très haut niveau) pour iOS, OS X, Android, et Windows en partageant le même code, sauf pour les vues. Les applications ont le même look and feel que des applications natives et permet d'accéder aux classes Java (pour Android) ainsi qu'aux API natives (telle que UIKit). Cependant, les prix des licences annuelles sont trois plus élevés que pour une licence Apple, et il y a moins de support.

4.2.2 La famille Android

La famille Android est très segmentée en terme de performances, d'architectures, résolutions d'écrans et de versions du système d'exploitation, rendant les essais sur cette plateforme secondaires. Les processeurs des appareils sous Android peuvent être des ARM, des x86 d'INTEL, voire des MIPS. L'appareil utilisé pour la majorité des tests était un Nexus 4, choisi pour sa prise en charge d'Android 4.3 et qui côté performance se place dans la moyenne des téléphone récent sous Android haut de gamme.

En comparaison avec l'iPhone 4, son processeur Qualcomm APQ8064 quadri-coeurs fait de lui une Formule 1.

Un HTC Wildfire sous Android 2.2 a également été utilisé pour vérifier que les modifications que j'avais apporté à l'application uCheck passaient bien sur les anciens modèles ainsi que pour des tests de performance. Son processeur armv6 Qualcomm MSM7225 mono-coeur faisait pâle figure face à celui l'iPhone 4.

Le développement d'applications se fait en Java SE via l'utilisation du SDK Android qui offre approximativement les mêmes possibilité que Cocoa pour iOS. Du code natif peut être écrit en C++ et interfacé au code Java via JNI. Il est également possible d'écrire des applications en HTML5 ou en C#

4.2.3 Conclusion de l'étude des caractéristiques matérielles

L'étude précédente a permis d'établir les conclusions suivantes :

- la compilation vers les architecture cibles armv7 et armv7s sous iOS, armv7, armv6 et x86 pour Android ;
- l'utilisation du C++ pour le développement du moteur de reconnaissance d'images, lié via de l'ObjectiveC/C++ (sous iOS) et du Java-JNI (sous Android) ;
- l'utilisation éventuelle d'OpenGL ES 2 et non pas des versions 1 ou 3 ;
- les optimisations éventuelles via des instructions NEON, l'utilisation d'options de compilation générant des optimisations NEON, la préférence vers des bibliothèques exploitant ces optimisations telles qu'OpenCV ou Eigen ;
- l'emploi du multi-threading et de bibliothèques l'utilisant afin de profiter pleinement des CPU multi-coeur des iPhone4S/5 et des processeurs récents des mobiles sous Android ;
- la faisabilité d'une application de réalité augmentée urbaine.

4.3 Architecture

L'architecture du produit final est découpée en plusieurs parties :

- un back-office[1] permettant aux annonceurs de créer des campagnes, ajouter des images à reconnaître, et des actions associées à la reconnaissance des images.
- un backend traitant les images et vidéos soumises par les annonceurs.
- des applications de reconnaissance d'images tournant sous iOS et Android.

Le stockage des données est effectué par le biais de deux services :

- Une base de données PostgreSQL qui référence les clients, les campagnes, les images à reconnaître et les actions associées à la reconnaissance.

[1] Bien que ce front-end soit destiné au client, nous ne l'avons pas appelé front-office afin de garder les dénominations côté client. En effet, ce front-end va être utilisé par notre client pour déterminer les images reconnues par les applications installées par les clients de notre client.

- Un bucket du service S3 d'AWS (Amazon Web Service) afin de garantir de la haute disponibilité sur les images, les vidéos et les descripteurs des images.

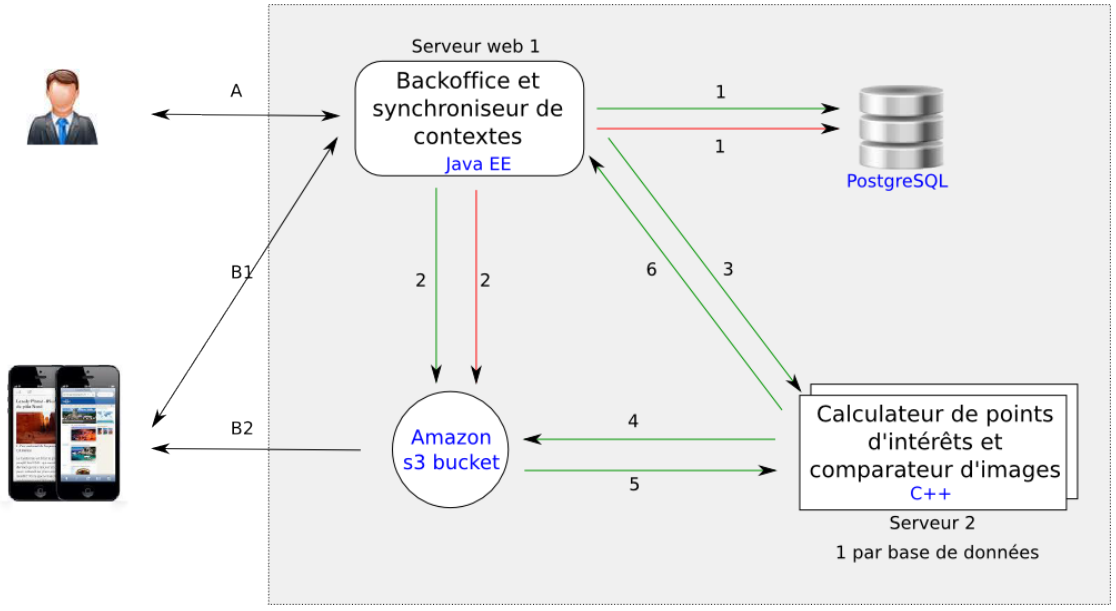

Fig. 5 : *Architecture finale du projet. Les flèches noires représentent les interactions d'un utilisateur avec un composant. Les flèches vertes correspondent aux transactions effectuées lors de l'ajout d'une image à reconnaître, les rouges aux suppressions de ces images. Les numéros présents correspondent à l'ordre des actions*

Cette architecture a été conçue en collaboration avec mon tuteur de stage à la suite de plusieurs itérations. J'ai réalisé l'ensemble des composants de cette architecture, à l'exception du back-office où j'ai repris du code existant d'un autre front-end que j'ai adapté.

Lorsqu'un annonceur connecté via le frontend sur le serveur 1, ajoute une image à reconnaître (flèche A), l'image est uploadée sur le bucket S3 (flèche 2) puis une référence est rajoutée à la base de données (flèche 1) et une notification est envoyée au serveur de traitement des images (serveur 2 via la flèche 3). Ce serveur récupère l'image sur le bucket (flèche 5), calcule ses points d'intérêt et ses descripteurs, et vérifie qu'une image similaire n'est pas déjà présente. Si cette dernière condition est vérifiée, les points d'intérêts sont renvoyés au bucket (flèche 5) et une notification de succès est envoyée au front-end (flèche 6).

Lorsqu'une image est retirée, l'image est déréférencée du serveur 2 (flche 3), de la base de données (flèche 1), et supprimée du bucket s3 (flèche 2).

Lors du lancement de l'application mobile, ce dernier va synchroniser sa liste des descripteurs des images et des actions de reconnaissance avec le back-office.

4.3.1 Le back-office

Le back-office est écrit en GWT et basé sur celui du produit uCheck.

Il comporte :

- Une page d'authentification ;
- Une page de visualisation et de création des campagnes ;
- Une page de visualisation, ajout, modification et suppression des images/vidéos associées à une campagne ainsi que des actions qui sont associées à leur reconnaissance.

L'ensemble des données (à l'exception des images) proviennent d'une base de données PostgreSQL. Le schéma est conçu de telle sorte que plusieurs clients soient rattachés à la même base de

données afin qu'ils puissent utiliser la même application pour reconnaître leurs images. Un client peut créer plusieurs campagnes, lesquelles peuvent contenir plusieurs contenus (images ou vidéos à reconnaître). Chaque contenu peut être associé à plusieurs urls qui sont des objets définissant un lien, un texte et un tag.

La gestion de la base de données est faite à la main sans l'aide de bibliothèque d'ORM. Ce schéma impose l'utilisation de 4 jointures pour la récupération de l'ensemble des images (avec leurs urls associées) provenant d'une base de données donnée.

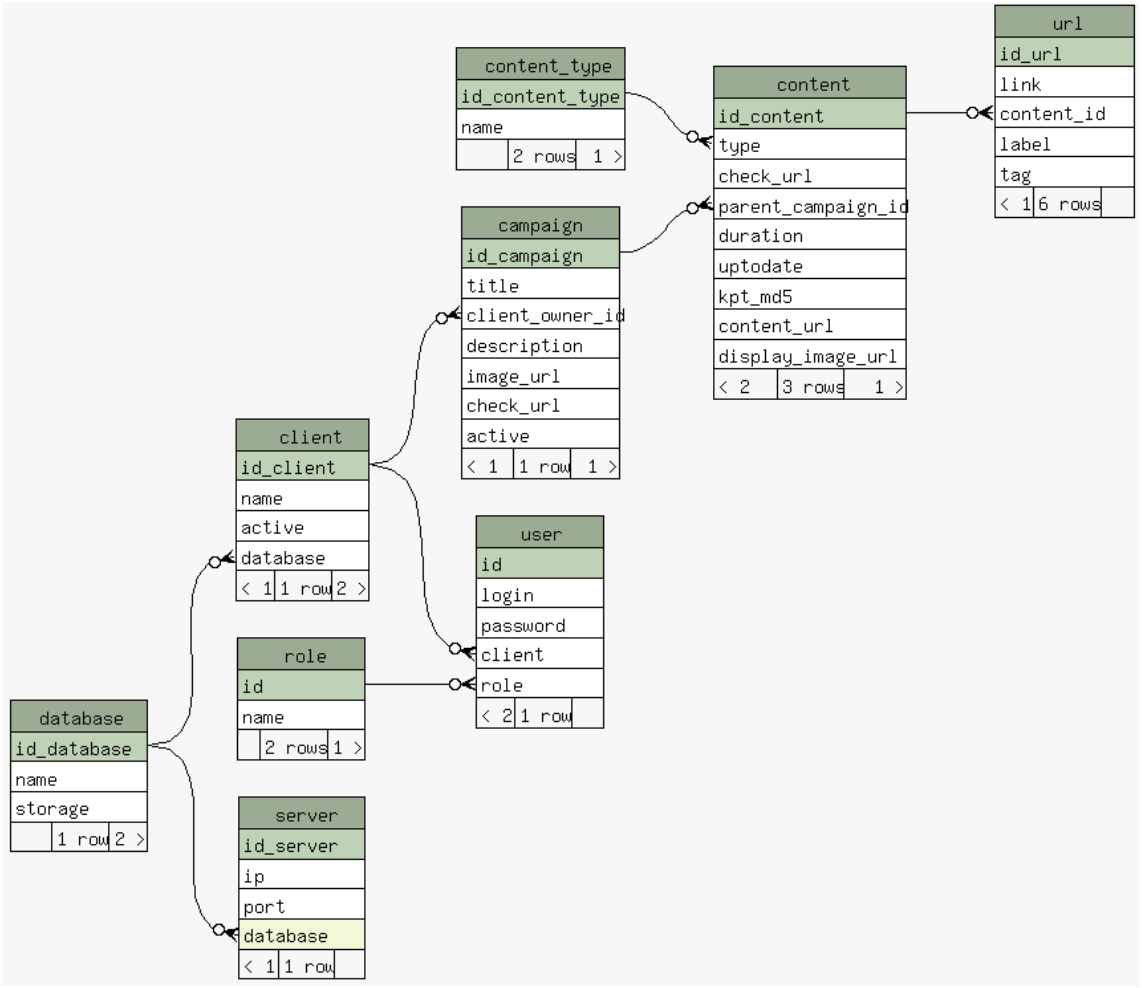

FIG. 6 : *Schéma des relations entre les tables utilisées par le back-office. Les cases en vert clair correspondent aux clefs primaires. Schéma réalisé avec schémaSpy*

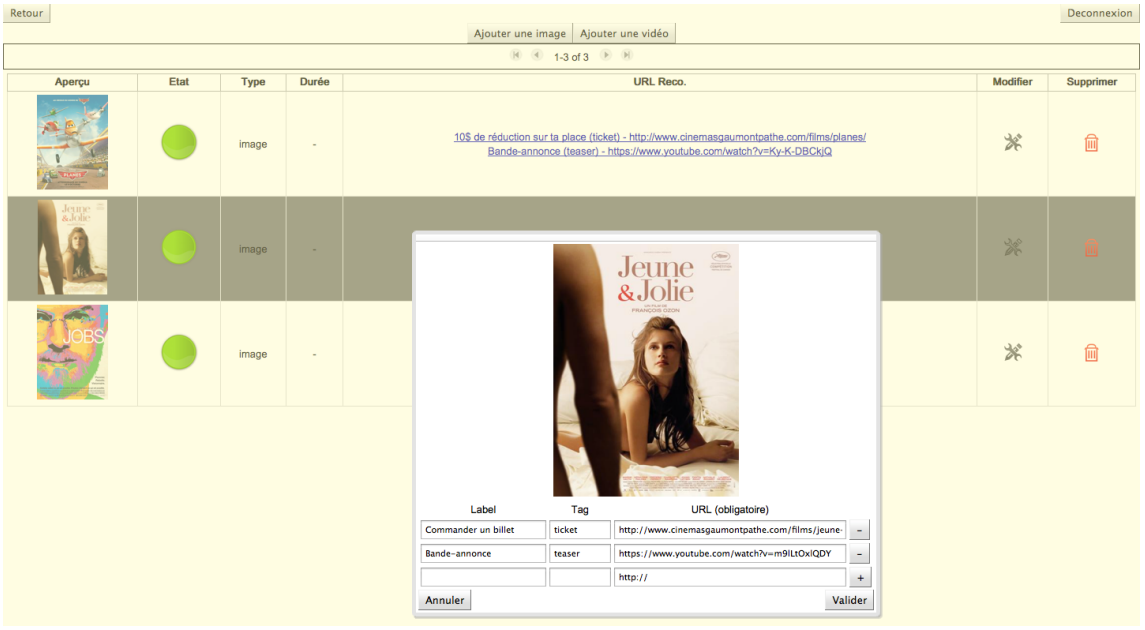

FIG. 7 : *Capture d'écran de la page de modification des images et des actions associées aux reconnaissances de ces images.*

Le back-office met à disposition des mobiles les actions associées à chaque image sous forme d'un fichier JSON. L'ensemble des descripteurs est mis à jour entre le mobile et le serveur via le back-office également. La description de ce mécanisme est décrite dans la suite de ce rapport.

Je me suis également occupé du déploiement de ce service sur un serveur Tomcat7.

4.3.2 Le back-end

Bien que le back-office soit constitué également d'une partie serveur que l'on pourrait appeler backend puisqu'elle gère la logique applicative du front-end, c'est à dire qu'elle va mettre à jour les modèles après avoir fait des vérifications sur les entrées de l'utilisateur et l'avertir en cas d'erreur. Il fallait un autre service, plus apte à faire des calculs intensifs et pouvant être détaché du back-office pour des raisons de cloisonnement applicatif. En effet, lorsqu'une image est uploadée par l'utilisateur, il faut vérifier qu'il n'existe pas déjà une image similaire en base, puis si cette condition est remplie, calculer les points d'intérêt et les descripteurs qui seront envoyés aux téléphones et utilisés pour les comparaisons avec les images suivantes. Nous appellerons "calculateur-comparateur" ce service.

Le moteur de reconnaissance sur les téléphones étant déjà écrit en C++, il fut décidé d'écrire un service en C++ également qui communique avec le back-office via des sockets et des requêtes HTML. Pour simplifier le développement de ce service nous avons utilisé la bibliothèque QT5 afin de gérer la partie socket TCP et des threads de façon plus portable. Le déploiement a été testé sur des serveurs Tomcat7.

Lorsqu'un usager a uploadé un contenu sur le back-office qui l'a envoyé au bucket s3, une trame TCP contenant l'identifiant et des informations relatives à ce contenu est envoyée au calculateur-comparateur. Pour empêcher que deux images identiques envoyées simultanément soient insérées en base, une seule comparaison de l'image est effectuée à la fois. Afin d'éviter d'avoir à maintenir un verrou durant tout le processus de reconnaissance, le calculateur-comparateur n'accepte qu'une seule connexion à la fois (les autres sont mises en attente de par le fonctionnement des sockets). La trame reçue, le contenu à traiter (identifié par la trame) est téléchargé depuis le bucket s3, puis leurs points d'intérêt et descripteurs sont calculés. Cette étape est nécessaire car les téléphones ne comparent pas les images directement mais les descripteurs de ces images (qui ont le bon goût d'être plus légers que les images) comme nous le détaillerons par la suite. Ces derniers sont uploadés

sur le bucket s3 puis comparés aux autres descripteurs. Si la comparaison est négative, on met à jour notre structure de données, puis on notifie le back-office du succès de l'opération en appelant une de ses servlets.

Chaque base de données est associée à un ou plusieurs serveurs calculateur-comparateur. En aucun cas un serveur ne gère plusieurs base de données. D'une part pour des raisons de cloisonnement et d'autre part parce qu'il y a un verrou sur l'étape d'analyse/comparaison de l'image située en aval de la réception de la notification qui empêcherait de traiter simultanément deux notifications correspondant à des images de bases de données d'images différentes.

4.3.3 Les applications mobiles

Les applications mobiles développées pour iOS et Android, sont les parties qui effectuent le plus de calculs, malgré les faible performances de ces plateformes. Comme expliqué précédemment le moteur de reconnaissance est écrit en C++ et est commun aux deux OS retenus, ainsi qu'au calculateur-comparateur. Seule la gestion de la caméra et la synchronisation des descripteurs des images à reconnaître est développée dans le langage natif de la plateforme considérée. Le fonctionnement de la reconnaissance d'images est exposée dans la suite de ce rapport.

Pour la version iOS, j'ai réalisé un framework ainsi qu'une application de démonstration, pour la version Android seulement une application de démonstration.

Les applications synchronisent les descripteurs des images à reconnaître avec le back-office en deux étapes. Dans un premier temps le téléphone envoie une requête POST contenant la liste des identifiants des descripteurs présents en local sur le téléphone. Le back-office calcule l'intersection de cette liste avec la liste courante des descripteurs, et renvoie au format JSON deux listes contenant respectivement les identifiants des descripteurs à télécharger et de ceux à supprimer. Dans un second temps, le téléphone télécharge et supprime les fichiers correspondant et peut enfin générer sa structure de recherche des descripteurs.

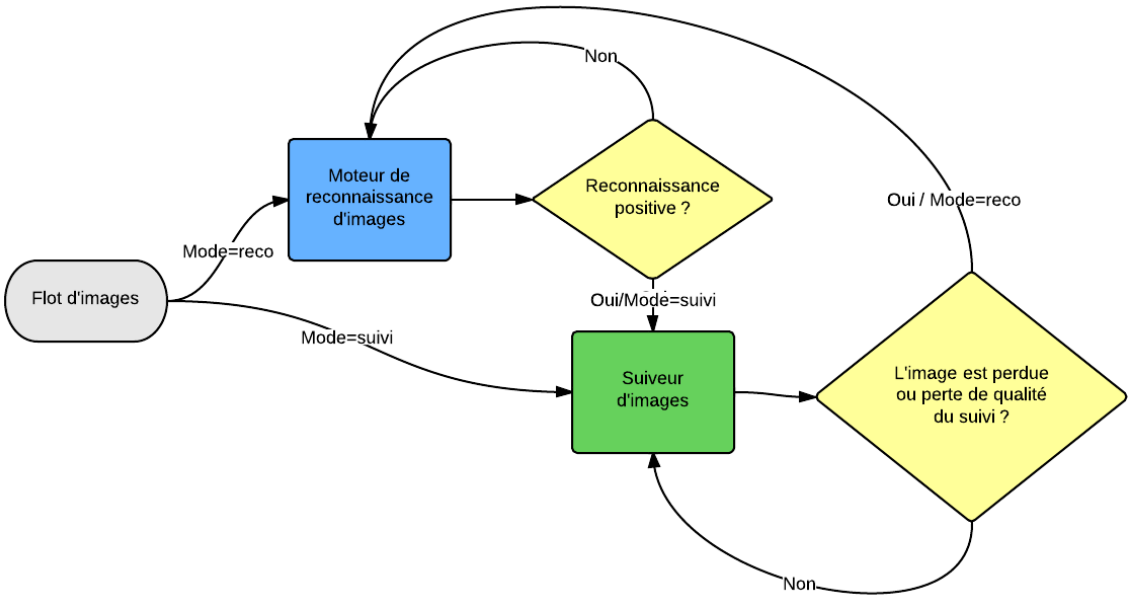

Fig. 8 : *Graphique de flot de contrôle de l'API décrivant les changements de modes entre la reconnaissance d'image et le suivi de l'image reconnue.*

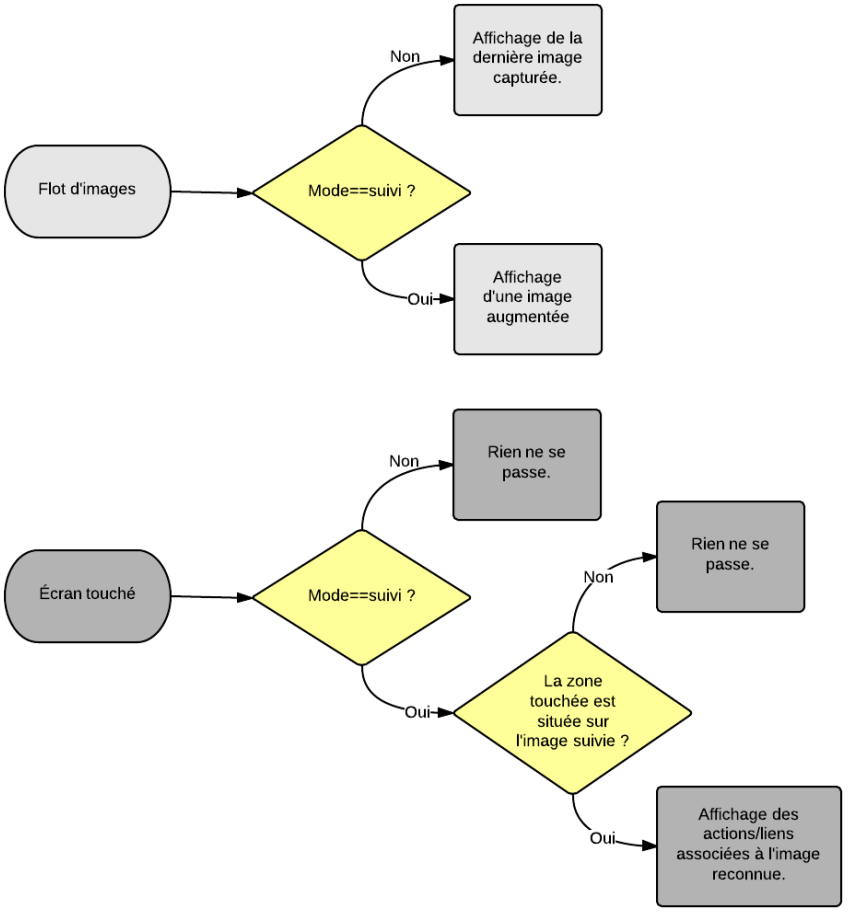

Fig. 9 : *Graphique de flot de contrôle de l'application de démonstration.*

4.4 État de l'art sur la reconnaissance d'images

4.4.1 Historique

4.4.1.1 Différentes approches

David Lowe a été un des précurseurs à proposer une méthode robuste de reconnaissance d'images. Dans sa publication présentant SIFT [Low03], il propose la pipeline suivante :

- calcul des points d'intérêt et de leurs descripteurs via SIFT ;
- appariement des descripteurs ayant la norme L2 la plus proche ; La recherche des descripteurs les plus proches (des images de référence) s'effectue à l'aide d'une variante de l'arbre-kd appelée best-bin-first ;
- calcul de la transformée de Hough pour clustériser les descripteurs ayant le même scaling et orientation (orientation et scaling normalisés avec les valeur du descripteur auquel il est apparié) ;
- estimation de la transformation affine.

Bien que les descripteurs (représentés par un vecteur de 128 flottants) permettent de discriminer de façon efficace les points d'intérêt, un grand nombre de points d'intérêt de l'arrière-plan peuvent être appariés incorrectement et donner lieu à des faux-positifs lors de l'étape de détection. Lowe préconise ainsi de vérifier la cohérence géométrique (spatiale) des appariements en les regroupant en clusters par le biais d'une transformée généralisée de Hough implémentée via une table de hachage. À la sortie de cette clustérisation il obtient des clusters correspondants à des

descripteurs appartenant potentiellement à la même image de référence. Selon David Lowe, appliquer RANSAC ou une méthode des moindres carrés à cette étape pour obtenir directement la transformation (entre l'image de référence, et la photographie contenant l'image détectée) serait inefficace car il reste encore à cette étape plus de 50 % de points aberrants (outliers). Enfin les plus gros clusters sont soumis à une étape de validation au cours de laquelle la transformation affine entre les points appariés est calculée par la méthode des moindres carrés. David Lowe choisit d'estimer une transformée affine car ses descripteurs ne sont invariant qu'à ces transformations et pour la raison qu'il suffit de trois correspondances (en pratique d'avantage pour plus de robustesse) pour les déterminer.

Contrairement à David Lowe, nous n'avons pas opté pour un seuil relatif (garder les appariements dont la distance entre les descripteurs est inférieure à k fois la deuxième meilleure distance, David Lowe prenant $k = 0,8$). En effet, nous avons remarqué que sur certaines images, il pouvait y avoir trois descripteurs qui avaient une distance extrêmement faible avec ceux de l'image de référence, les autres descripteurs correctement appariés ayant une distance faible ou moyenne avec ceux de l'image de référence. Dans cette situation, une valeur de k faible empêchait la sélection de plus de ces trois appariements. A contrario, avec d'autres images, en particulier celles présentant une zone de flou, la deuxième meilleure distance sera très élevée, et le trop grand nombre d'appariements pris en compte fera échouer l'étape de validation géométrique. D'un point de vue théorique, il ne nous paraissait pas non plus judicieux de définir un seuil prenant en compte seulement la valeur minimale de la distribution des distances entre les descripteurs. Nous avons ainsi opté pour un seuil global, qui donnent de meilleurs résultats en pratique et pénalisant volontairement les images floues, qui n'ont pas lieu d'être comparées à des images de références parfaitement nettes.

Une des limitations de la méthode de Lowe est la recherche, pour chaque descripteur de l'image photographiée, du descripteur le plus proche parmi ceux des images de référence. En fonction du nombre d'images de références prises en compte, cette recherche peut être très coûteuse. De plus, lors de l'étape d'appariement des descripteurs, on remarque qu'un nombre non négligeable d'appariements sont faux. Il arrive fréquemment que certains descripteurs d'une image soient proches de descripteurs d'une autre image d'entraînement. L'idée serait de pouvoir déterminer quels descripteurs sont les plus pertinents, et les considérer en priorité.

Gabriella Csurka propose dans son article [CDF*04] de réduire le problème de catégorisation en un problème d'apprentissage supervisé multi-classes où une classe correspond à une catégorie visuelle. Cette méthode intitulée Bag of Keypoints consiste à réduire une image en une liste de régions (features) qui sont caractérisées séparément et évaluées avec un histogramme d'apparence dans les autres images, de la même façon que la pertinence d'un mot est évaluée en fonction de son nombre d'occurrences dans un jeu de documents. Les différentes étapes de l'entraînement de l'algorithme sont les suivantes :

- On calcule sur les images de référence des points d'intérêt et leurs descripteurs associés.
- On quantifie (quantization) les descripteurs obtenus afin de réduire le nombre de mots. Pour cela, on rassemble les descripteurs en clusters : les centres des clusters définissent les mots. En reprenant l'analogie avec les mots, cette étape de clustérisation va permettre de considérer les mots tel que "géométrie", "géométries" et "géométriquement" correspondent à un même mot.
- Chaque image est ainsi représentée par un vecteur dont les composantes correspondent aux fréquences d'apparition de chacun de ces mots. Les composantes des vecteurs sont ensuite ré-ajustées avec des poids qui varient en fonction de la fréquence d'un mot dans l'ensemble de la base. On crée ensuite un annuaire inversé dans lequel chaque mot va pointer vers les images dans lequel il apparaît avec son occurrence.
- Un classificateur de type Bayes ou SVM (Support Vector Machine) va permettre de déterminer quels vocabulaires et paramètres du classificateur donnent les meilleurs résultats de catégorisation.

Pour déterminer la photographie la plus proche d'une image requête, on calcule le vecteur de fréquence des mots de l'image requête que l'on compare avec les vecteurs des images d'entraînement.

Le taux d'erreur minimal obtenu par Gabriella Csurka est de 15%, ce qui peut paraître élevé, mais en réalité, le taux d'erreur sans étape de validation, de la méthode de David Lowe est plus élevé.

Une des limitation de l'approche bag of keypoints est que la clusterisation et l'apprentissage supervisé pour la création du vocabulaire sont coûteux si le nombre d'images est important. De plus il faut recommencer cette étape quand on rajoute une image dans la base de connaissances.

David Nistér and Henrik Stewénius [NS06] proposent une variante intitulée Vocabulary Tree, utilisée pour ranger les mots de façon hiérarchique. Chaque nœud est associé à un cluster de caractéristiques (features) qui est progressivement subdivisé à chaque hauteur, via l'algorithme K-means, en arbre de clusters plus petits.

4.4.1.2 Les solutions sur plateformes mobiles

En 2005, Gerald Fritz et al. [FSP06] est l'un des premiers à proposer une solution de reconnaissance sur mobile. A l'époque, la technologie ne permet pas d'effectuer les traitements en local. Son application externalise ainsi les calculs sur un serveur qui reconnaît par classification de descripteurs i-sift, le bâtiment d'une photographie.

La même année, Jonathon S. Hare et Paul H. Lewis proposaient une application mobile de reconnaissance d'œuvres d'arts [HL05] où la phase de reconnaissance était également effectuée sur un serveur. L'étape de reconnaissance s'appuie sur une approche "bag of keypoints" utilisant les descripteurs de SIFT, à laquelle est ajoutée une étape de validation géométrique consistant à déterminer l'homographie entre les descripteurs appariés entre les images requête et référence.

Un an plus tard, une autre tentative de reconnaissance d'oeuvre d'art est réalisée par Herbert Bay et al. [BFG06] mais cette fois-ci les calculs sont effectués en local sur une tablette équipée d'un processeur Pentium M cadencé à 1.7 GHz. De même que dans la méthode de David Lowe, les descripteurs de SIFT sont appariés en utilisant un seuil relatif, mais au lieu d'utiliser l'algorithme "best bin first" pour comparer les descripteurs, il effectue une recherche linéaire. Herbert Bay et al montre qu'il possible d'atteindre des taux de reconnaissance équivalents en utilisant les descripteurs SURF, avec un gain de rapidité de l'ordre de trois. Sur un corpus de 205 images de dimensions 320x240, il arrive à identifier les œuvres d'art en une minute minimum et avec un taux de succès supérieur à 84.5 %.

Gabriel Takacs et al. [TCG*08] ont travaillé sur l'augmentation des bâtiments d'une ville, où la reconnaissance des photos est effectuée en local sur des téléphones portables de type Nokia N93, N95 et N5700 aux performances nettement inférieures à celle d'un iPhone 4. L'application n'est pas sans rappeler celle de la RATP qui permet d'afficher en réalité augmentée les stations de métros et les lieux d'intérêt. En raison des performances de ces téléphones, la base de données des descripteurs est téléchargée en fonction de la position de l'utilisateur. De plus ils utilisent les données du GPS embarqué pour restreindre la recherche à certaines cellules d'une carte qui appellent loxels, voisines de la position de l'utilisateur. Utilisant en entrée un grand nombre de photos ayant des zones communes, ils sélectionnent les descripteurs qui apparaissent sur plusieurs images, permettant de réduire considérablement la base des descripteurs, calculés à partir d'une implémentation maison de SURF. Ensuite ANN [AMN*98] est employée pour les recherches dans la base des descripteurs. Enfin la validation géométrique est effectuée en utilisant un modèle affine. Gabriel Takacs et al. argumentent ce choix par le fait que l'estimation d'une homographie est plus compliquée à estimer et nécessite d'avantage d'appariements corrects (inliers), alors qu'un modèle affine donnerait des résultats équivalents. Ce choix doit s'expliquer également par le fait qu'en prenant des photos à peu près en face d'un bâtiment, on conserve approximativement le caractère parallèle des lignes des façades faisant face à l'observateur.

Avec des images de taille moyenne 640x480, 250 points d'intérêt par image, 7000 descripteurs dans la base, ils obtiennent une temps de reconnaissance de 2.8s, dont 2.4s pour SURF et une consommation mémoire de 1.8Mo pour la base des descripteurs. L'arbre de recherche des descripteurs est reconstruit à chaque changement de loxel, et nécessite 0.5s. En guise de comparaison, notre prototype actuel sur iPhone 4 effectue une reconnaissance sur des images de tailles 480x640

en 800ms, dont 400ms pour le calcul des points d'intérêt et de leurs descripteurs en utilisant ORB. Avec 256 images de référence comportant 400 points d'intérêt, la construction de notre index prend 1s et nécessite 19Mo, tandis que la recherche dans la base des descripteurs s'effectue en 70ms avec LSH.

4.4.2 Les outils

4.4.2.1 Les points d'intérêt et les descripteurs

Une des étapes de la reconnaissance parmi les plus critiques est le calcul des points d'intérêt et de leurs descripteurs locaux, qui sont des vecteurs représentant les caractéristiques de l'image au voisinage d'un point. D'une part, il s'agit d'une étape très coûteuse en calcul et d'autre part, la qualité des points d'intérêt détermineront la robustesse de la reconnaissance.

Mais tout d'abord, pourquoi pour effectuer de la reconnaissance d'image, s'acharne t-on à utiliser des points d'intérêt couplés à des descripteurs dont la représentation pour un cerveau humain est plutôt abstraite, alors qu'une image peut (comme le fait le cerveau humain) être identifiée par son ensemble de formes et de couleurs qui pourraient être obtenues informatiquement par segmentation ? Tout d'abord, les couleurs sont souvent altérées par l'éclairage (couleur, intensité), des reflets ...Il faut ainsi être très tolérants sur les plages de valeurs acceptables pour distinguer des couleurs d'une palette donnée. Ensuite, les algorithmes de segmentation sont très coûteux en temps de calcul et il n'y a pas d'unicité de la segmentation. C'est très suggestif de dire qu'une segmentation s1 est meilleure qu'une segmentation s2. Selon les paramètres définis, certaines régions se retrouverons fusionner et il sera très délicat de les apparier via des descripteurs de formes aux régions de l'image de référence, d'autant plus ces formes auront subies des transformations homographiques. Tandis que les points d'intérêt et les descripteurs sont robustes à tous ces changements, permettant la reconnaissance. Enfin, une motivation minoritaire consiste à dire que les algorithmes de calcul des points d'intérêt et des descripteurs miment la façon dont la vision humaine fonctionnerait : par la recherches de contours ou de variations de gradients à différentes échelles.

Un bon algorithme de calcul de points d'intérêt/descripteurs doit être répétable, c'est-à-dire que l'on doit retrouver les mêmes points d'intérêt/descripteurs sur des images différentes d'une variation d'éclairage, d'échelle, de rotation ou de changement de perspective. En même temps les descripteurs locaux doivent être assez discriminants afin de permettre un meilleur taux d'appariement. Ce juste milieu est tellement délicat que les efforts pour améliorer l'invariance des descripteurs aboutissent souvent à une perte de distinction des descripteurs.

Malgré son ancienneté, SIFT [Low03] est souvent présenté comme la référence en matière de robustesse, ce qui s'est concrétisé lors de nos tests par les meilleurs taux d'appariements. Toutefois, SIFT souffre de deux gros défauts : sa lenteur (4.8s sur un iPhone 4 pour une image de dimensions 480x356) et la lourdeur de ses descripteurs (128 nombres à virgule flottante).

SURF [BETVG08] nous a également surpris concernant sa robustesse, mais déçu par ses performances : 1.1s pour le calcul des points d'intérêt.

ORB [RRKB11] que nous avons retenu est basé sur une variante de FAST (pour les points d'intérêt) et de BRIEF (pour les descripteurs) rendu invariant aux rotations via la méthode de Rosin. Les descripteurs d'ORB sont des vecteurs de 32 octets. Le choix de ne pas utiliser de nombres à virgule flottant (à contrario de SIFT et SURF) plus coûteux à manipuler, se révèle particulièrement judicieux sur des plateformes mobiles.

Aujourd'hui, des algorithmes de calculs de points d'intérêt et des descripteurs ont été spécialement conçus pour être performants sur des téléphones. On citera notamment Wei-Chao Chen et al. qui propose une optimisation de SURF [CXG*07] CARD [AY11] de Mitsuru Ambai and Yuichi Yoshida basée sur des gradients orientés et l'algorithme de Simon Taylor et Tom Drummond [TD09] basé sur l'algorithme FAST et dont les descripteurs proviennent d'un apprentissage supervisé de fenêtres autour des mêmes points d'intérêt d'images ayant subies des changements de perspective, d'échelle et des flous gaussiens. Malheureusement, le code source de ces algorithmes n'est pas publié et la licence des binaires de CARD interdit toute exploitation commerciale.

4.4.2.2 Comparaison des descripteurs

Une fois les descripteurs de l'image à reconnaître calculée, il faut les comparer à l'ensemble des descripteurs des images de référence. Avec 200 descripteurs par image, il n'est pas raisonnable d'effectuer une recherche linéaire au-delà de 5 images dans la base des descripteurs. Il faut donc une structure de données adaptée, offrant si possible pour les recherches, une complexité temporelle en $o(log(n))$, où n est le nombre de descripteurs dans la base.

S. Arya et al. [AMN*98] explique que si la condition $2^d << n$ (où d est la dimensions des descripteurs) n'est pas vérifiée, alors les structures de données arborescentes classiques sont inefficaces. Lorsque d est supérieur à 2, il faut sacrifier la complexité spatiale pour obtenir une recherche en temps logarithmique. D'où la proposition de chercher un candidat approximatif, c'est-à-dire proche à un epsilon près du plus proche voisin. Ils proposent une structure de données intitulée arbre BBD (balanced box-decomposition) qui permet une recherche linéaire avec une complexité spatiale linéaire et un temps de construction quasi-linéaire.

La même année, Piotr Indyk et Rajeev Motwani proposent LSH (local sensitivity hashing) [IM98]. L'idée de base de LSH est d'utiliser des fonctions de hachage qui auront une forte probabilité de collision pour des vecteurs proches. Pour effectuer une recherche, on calcule les empreintes (hashs) d'un vecteur requête en utilisant ces fonctions de hachage. On sélectionne ainsi des candidats que l'on trie en fonction de leur distance avec le vecteur requête afin de sélectionner les k plus proches voisins parmi ces candidats. Toutefois, dans le cas où le nombre de données est très important, LSH requiert plus d'une centaine de tables de hachage pour assurer que des vecteurs proches auront des empreintes proches.

Qin Lv et al. a proposé depuis une variante de LSH intitulée Muliprobe LSH [LJW*07] permettant de corriger ce défaut. C'est l'implémentation de LSH qui a été retenue dans la bibliothèque OpenCV.

[ML09]

En 2009, Davig G. Lowe et Marius Muja ont étudié les résultats de certaines variantes des kd-tree sur divers jeux de données afin de proposer un algorithme permettant de sélectionner la meilleure variante de kd-tree avec les meilleurs paramètres à partir d'un échantillon du jeu de données à utiliser et des contraintes de l'utilisateur : complexité spatiale, complexité temporelle de création de l'arbre et complexité temporelle des recherches.

D'après les auteurs, il n'y a pas de meilleur algorithme pour un jeu de données donné. Selon la précision des résultats escomptée, selon les propriétés des données (forte corrélations entre dimensions, …), les paramètres de chacun des algorithmes (nombre d'arbres, nombre de fils, ...) ont aussi leur l'importance.

En particulier, si les vecteurs proviennent d'une distribution uniforme, les performances seront comparables à une recherche linéaire.

OpenCV [Bra00] propose une implémentation de FLANN basé sur cet article, qui semble en effet utiliser par défaut les "multiple randomized kd-trees" comme structure de données, car les résultats retournés ne sont pas toujours identiques pour un même vecteur de recherche et une base de recherche donnée.

4.4.2.3 Estimation de l'homographie

Une fois que l'on a déterminé l'image de référence qui intervenait avec le plus grand nombre d'appariements de l'image requête, la détection n'est pas terminée. Afin de réduire le nombre de faux-positifs, on va estimer la position des coins de l'image reconnue et vérifier que ces coins définissent un quadrilatère convexe peu aplati. Pour estimer la position de ces coins, nous avons choisi de calculer l'homographie entre les points d'intérêt appariés des deux images.

Étant souvent sujet à un grand nombre d'appariements incorrects (outliers), nous avons utilisé dans un premier temps RANSAC dont les performances laissaient à désirer (jusqu'à 1.2s pour 200 appariements dans le pire des cas). L'estimation de cette homographie peut être calculée à moindre coût en utilisant un à priori : les distances entre les descripteurs appariés. En effet, les appariements

correspondants aux plus faibles distances ont plus de chances d'être corrects (inliers), d'où l'idée de les considérer en priorité. C'est sur ce constat que se base PROSAC [cit05] qui promet un gain en performance d'ordre 100.

Toutefois, c'est l'algorithme LMEDS que nous avons retenu (qui nécessite plus d'inliers que RANSAC), car pour des performances nettement plus élevées que RANSAC, nous obtenions des résultats comparables. Ceci s'explique par le fait que lors de nos expérimentations, le taux d'appariements erronés était souvent soit faible, soit très élevé. RANSAC aurait sans doute été intéressant si l'on avait eu des cas avec autant d'appariements erronés que corrects.

4.5 L'algorithme de reconnaissance d'images et de tracking

La pipeline retenue de reconnaissance d'images s'inspire de la méthode de David Lowe et comprend les étapes suivantes :

- calcul des points d'intérêt et des descripteurs ;
- recherche des descripteurs les plus proches parmi ceux des images à reconnaître et détermination de l'image potentiellement la plus proche ;
- validation via estimation de l'homographie.

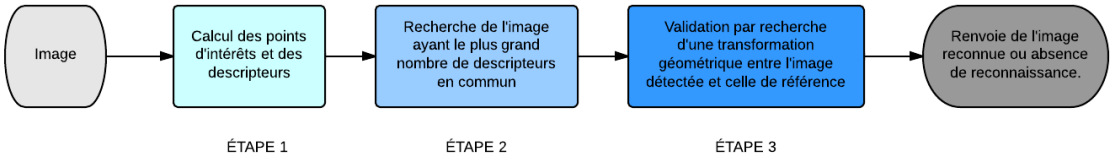

FIG. 10 : *La pipeline de reconnaissance d'images.*

4.5.1 Les points d'intérêt

Le calcul des points d'intérêt et des descripteurs est la première étape de la pipeline. Dans un premier temps j'ai évalué les performances de différents algorithmes de calcul de points d'intérêt et de descripteurs sur des images de taille 480x360 et sélectionné ceux ayant un temps de calcul inférieur à 0.5s sur un iPhone4 pour 800 points d'intérêt. À savoir FAST, BRISK, ORB et STAR pour les détecteurs et BRISK, BRIEF et ORB pour les descripteurs qui sont tous des descripteurs binaires, plus rapides à calculer que leur pendant flottant.

Les candidats retenus ont ensuite été évalués via un test de robustesse plutôt naïf qui consistait à comparer la différence des positions des coins de l'image détectée dont on avait estimé l'homographie après application de l'homographie avec les véritables coins de l'image.

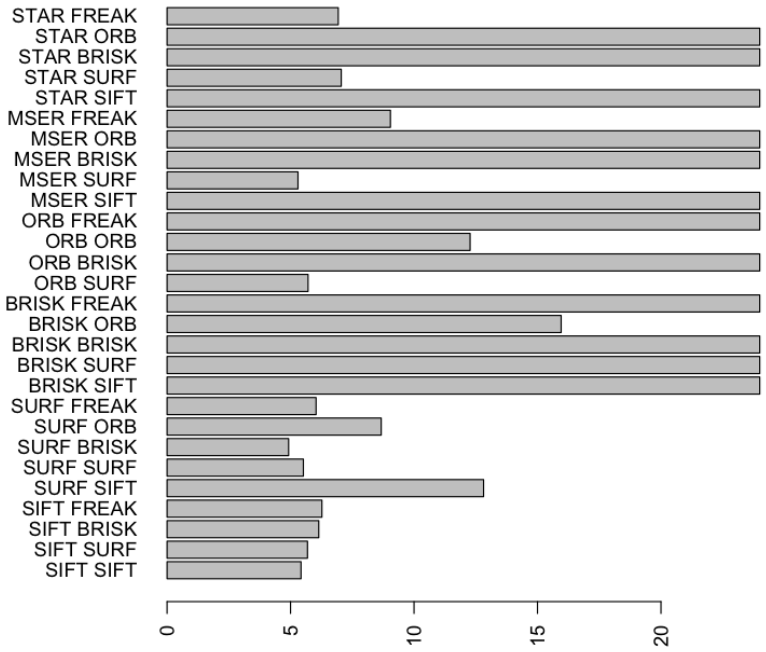

Fig. 11 : *Graphique en bâtons représentant à l'issue du test de robustesse, les moyennes pour chaque image de test des moyennes des sommes des écarts entre les coins estimés et leurs valeurs exactes. Lorsque cette valeur dépassait 24 pour un couple d'images, l'homographie était estimée comme erronée et cette valeur était conservée jusqu'à la fin du test pour éviter d'étirer le graphique généré par le programme. Les tests ont été effectués ici sur tous les couples de détecteur/extracteur, même ceux considérés comme pas assez performants.*

Le jeu d'images de tests était généré en utilisant le logiciel de modélisation 3D Blender, plutôt qu'en appliquant des transformations homographiques à des images issues de matrices d'homographies aléatoires. La raison à cela était que le cahier des charges autorisait le moteur de reconnaissance à échouer si l'angle entre la normale de l'image à reconnaître et l'axe Z du repère caméra était trop important où si l'écart de distance entre l'image et la caméra était également trop grand. Cet angle et cette distance étaient plus facilement paramétrables en passant par des images de synthèse. Un deuxième jeu d'images issues de photos des images à reconnaître prises par l'iPhone fut également utilisé ultérieurement. Le programme d'évaluation naïf écrit en Python recherchait de façon linéaire, parmi tous les descripteurs calculés sur l'image de référence, celui qui était le plus proche (au sens de plusieurs distances dont celle retenue fut la distance de Hamming) du descripteur de l'image à reconnaître, sans effectuer de validation croisée. Une fois l'ensemble des appariements calculés, on gardait seulement ceux dont la distance était inférieure à un certain seuil multiplié par la plus petite distance. Ce seuil était ajusté empiriquement en fonction des descripteurs afin d'éliminer le plus grand nombre d'appariements incorrects (outliers).

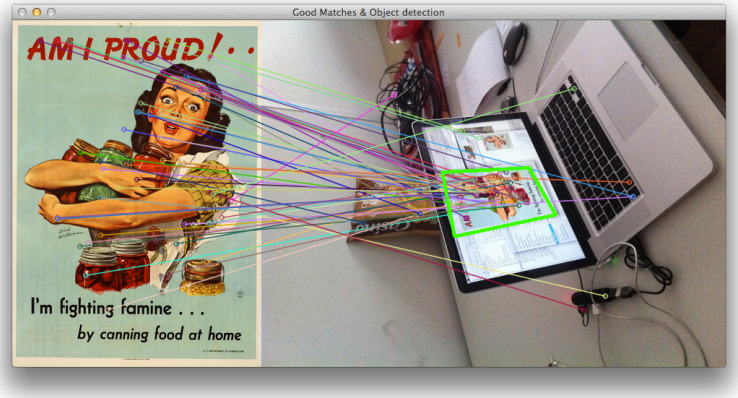

FIG. 12 : *Test de robustesse. À gauche l'image de référence. À droite l'image requête. Les cercles représentent les points d'intérêt, les segments les appariements sélectionnés par le seuil. Le quadrilatère vert correspond à l'image des coins de l'image de référence par l'homographie estimée. On notera que malgré le seuillage des appariements, il y a un grand nombre d'outliers.*

Ce test était naïf dans le sens où il ne mesurait pas vraiment la robustesse intrinsèque des points d'intérêt et des descripteurs mais plutôt leur performance au sein d'un algorithme de reconnaissance prématuré. Pour faire bien, il aurait fallu d'une part mesurer la répétabilité des détecteurs sur des images dont on connaissait l'inclination, la rotation, le changement de luminosité (en ne faisant varier qu'un seul paramètre à la fois) et projeter les points d'intérêt trouvés de l'image de référence sur l'image modifiée (via l'homographie supposée connue) puis évaluer la différence entre les points provenant de cette projection avec ceux de l'image modifiée.

Et d'autre part réaliser un test similaire pour les descripteurs où l'on aurait comparé la distance entre des descripteurs situés sur des points quelconques de l'image de référence avec leur projection situés sur l'image modifiée.

A l'issue de cette phase d'évaluation, c'est le couple de détecteur/extracteur ORB/ORB qui fut retenu. Toutefois conscient de la naïveté de mon évaluation, j'ai conçu mon architecture logicielle de façon à pouvoir facilement changer ultérieurement de détecteur ou d'extracteur.

Plus tard, j'ai effectué des tests afin de déterminer divers paramètres intervenant dans le calcul des points d'intérêt comme

- le nombre de points d'intérêt. Le diminuer permet de diminuer les temps de calcul et les temps de transfert (des descripteurs des images à reconnaître qui sont synchronisés via le réseau), mais engendre des pertes de robustesse. En dessous de 150 points d'intérêt, la robustesse se dégrade rapidement. Au delà de 500 descripteurs les gains en robustesse sont infimes. Un nombre de 200 points d'intérêt offrait un bon compromis ;
- l'utilisation d'un grille afin de forcer une répartition plus homogène des points d'intérêt, qui donnait certes de meilleurs résultats dans certains cas où l'image requête possédait des zones très contrastées localement (les points d'intérêt étaient tous répartis dans cette zone), mais conduisait à une dégradation globale des résultats ;
- le choix entre le calcul des descripteurs sur des images en nuance de gris ou dans un espace de couleurs opposées. Étrangement, les résultats n'étaient pas meilleurs en espace de couleurs opposées et les descripteurs se trouvés alourdis d'un facteur 3 ;
- la dimension des images d'entrainement. Les points d'intérêt et descripteurs de ORB étant moins robustes au scaling que ceux de SURF, la décision fut prise de prendre des images de référence de taille similaire à la taille des images traitées sur le téléphone. Ce choix fut confirmé par les tests. De plus, plus la taille des images de référence ainsi que celles utilisées pour le calcul est grande, meilleure sera la robustesse, au détriment du temps de calcul. Les résolutions 480x360 et 320x288 (qui sont des résolutions natives de l'iPhone 4) furent

retenues.

4.5.2 Comparaison des descripteurs

La comparaison et la recherche des descripteurs constitue la seconde étape de la pipeline. Cette partie était le deuxième nerf de guerre du projet, après les descripteurs. La qualité et la rapidité de la reconnaissance d'image dépend énormément de ses performances. L'objectif de cette phase est de pouvoir déterminer en un temps assez court (<400ms) pour tous les descripteurs de l'image à reconnaître, les descripteurs les plus proches parmi ceux de toutes les images à reconnaître. Soit moins de 2ms pour la recherche d'un descripteur. Une fois les descripteurs appariés, on détermine les images de référence qui interviennent dans le plus grand nombre d'appariements. Ces images candidats sont ensuite évaluées par la phase de validation. Des tests plus poussés ont montré que si l'image candidate correspondant au plus d'appariements n'est pas validée, la suivante (ayant le deuxième plus grand nombre d'appariements) n'est jamais validée.

Une première tentative consista à utiliser l'algorithme FLANN (implémentation d'OpenCV) avec une distance de Hamming, qui donna des résultats très décevants.

Dans un second temps, l'implémentation de l'algorithme LSH multiprobe d'OpenCV fut évalué avec des résultats plus convainquants. Sur un jeu de 80 images requête et 74 images de référence (à reconnaître), seules 5 images n'ont pas été correctement reconnues (l'image candidat était incorrecte).

Un des inconvénients de l'implémentation actuellement est l'obligation de reconstruire l'index de la structure de recherche après l'ajout de nouveaux descripteurs et chaque démarrage de l'application. De plus la sérialisation de cette structure de données n'est pas possible, mais heureusement le temps de construction de l'index ne prend que 7 secondes pour 4900 images.

4.5.3 Validation

La validation est la troisième étape de la pipeline. Le cahier des charges imposait que le taux de faux-positifs soit très faible. L'accent a donc été mis sur la phase de validation afin de filtrer toutes les images mal reconnues par l'étape précédente.

Durant cette phase, les plus proches voisins des descripteurs de l'image reconnue et de l'image à reconnaître sont de nouveau calculés par recherche linéaire et un seuillage des appariements est effectué comme dans l'application de test de robustesse. Certes, cette étape a déjà été réalisée avec LSH précédemment, mais avec un nombre d'appariements inférieur. Les tests ont montré que le sacrifice en temps de calcul (20ms) est négligeable face au gain en nombre d'appariements corrects supplémentaires obtenus. En effet, pour déterminer l'homographie entre l'image requête et l'image de référence, les divers algorithmes d'estimation tels que RANSAC et LMEDS ont besoin d'un ratio de 40% d'inliers. RANSAC et LMEDS donnaient expérimentalement les mêmes résultats, mais les temps de calcul très variables de RANSAC (jusqu'à 120ms en cas d'échec d'estimation) ont donné raison à LMEDS.

Une fois l'homographie déterminée, celle-ci est appliquée aux quatre coins de l'image. Une dernière fonction calcule les angles sur le quadrilatère obtenu afin de vérifier qu'ils correspondent à un quadrilatère peu aplati.

A l'issue de cette étape, le taux de faux positif était inférieur à 1/10000 toutefois, le taux d'images correctement reconnues non validées pouvait dépasser les 5%, dû à des échec d'estimation de l'homographie dans la majorité des cas.

Fig. 13 : *Un cas pathologique où tous les appariements sont corrects, mais l'homographie estimée est trop approximative car ces appariements sont situés sur la même ligne.*

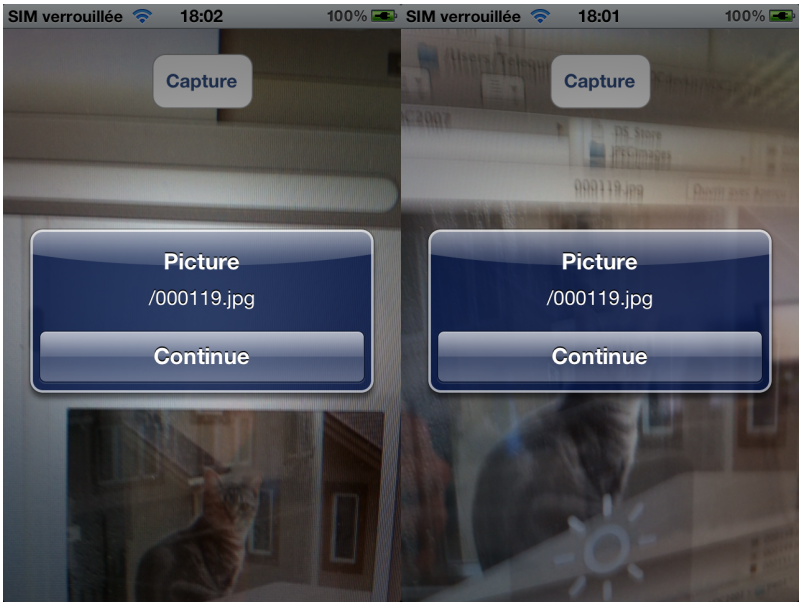

Fig. 14 : *Captures d'écran du prototype pour iPhone du moteur de reconnaissance à l'issue de la 5e semaine montrant la robustesse de la reconnaissance malgré les reflets. L'image du chat est correctement reconnue dans les deux cas.*

4.6 Le tracking

L'objectif du tracking est de pouvoir suivre une image reconnue et d'afficher en sur-impression une autre image.

La pipeline de reconnaissance d'images n'était pas assez rapide pour déterminer en temps réel les coordonnées d'une image reconnue. Pour effectuer du suivi d'image, il existe des méthodes moins coûteuses et plus adaptées comme les algorithmes de calcul de flot optique. Pour simplifier, un flot optique est un champ de vecteur qui représente le déplacement de chaque pixel entre deux images successives. Il est ainsi possible de déterminer rapidement et approximativement la position d'un certain nombre des points d'intérêt de l'image précédente dans l'image courante. L'algorithme de Lucas-Kanade présent dans OpenCV permet d'effectuer cette opération en moins de 100ms pour une image de dimensions 320x288. En déterminant l'homographie entre les deux dernières

images on peut en déduire la transformation à effectuer sur l'image de sur-impression (qui est la multiplication de toutes les homographies précédentes).

Au bout d'un certain nombre d'itérations, la qualité du tracking se dégrade à cause des pertes de précision accumulées par les homographies précédentes et à la diminution du nombre de points d'intérêt qui ont pu être suivis. Le système passe alors en mode de reconnaissance d'image afin de calculer une nouvelle homographie.

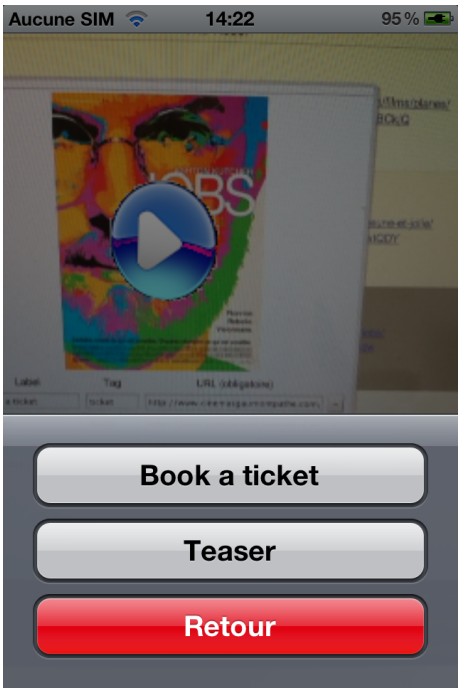

FIG. 15 : *Captures d'écran de l'application de démonstration montrant l'affiche du film Jobs trackée et sur laquelle un bouton "play" est affiché en sur-impression, ainsi que la liste des actions associées à la reconnaissance de l'image qui s'affiche lorsque l'utilisateur touche l'affiche.*

4.7 Les optimisations

Plusieurs optimisations on été envisagées pour accélérer la latence du moteur de reconnaissance et du tracking. Cette partie était une des plus hasardeuse du projet, certaines tentatives n'ont pu porter leur fruits, ainsi peu de temps a été consacré à cette étape en comparaison du temps nécessaire pour tester toutes les voies possibles.

4.7.1 Optimisations CPU

4.7.1.1 La famille x86

Le backend, le serveur de reconnaissance et certains mobiles sous Android tournent sur des Processeurs Intel de la famille x86. Sur ce type de processeur, il est possible d'activer l'utilisation des instructions SSE permettant de bénéficier d'accélérations matérielle sur les calculs faisant intervenir des vecteurs de nombre flottants. Aucun développement spécifique utilisant ces instructions n'a été réalisé, cependant les flags de compilations permettant d'en générer ont été activés.

4.7.1.2 La famille ARM

La famille ARM évolue rapidement à ce jour. On trouve de plus en plus de processeurs multi-coeurs permettant de paralléliser davantage de calculs. Sur les processeurs de la famille ARM on trouve également des instructions SIMD permettant d'effectuer du calcul vectoriel sur flottant sous le nom d'instructions NEON. Ces instructions ont l'inconvénient de ne pas être entièrement compatibles avec le standard IEEE-754, à contrario des VFP. Les VFP permettent également d'effectuer des calculs sur des vecteurs de flottant mais sans aucune parallélisation sur le traitement des éléments des vecteurs, réduisant leur intérêt.

Aucun développement n'a été effectué avec des instructions NEON. Des extraits de code en assembleur NEON ont toutefois été utilisé pour la conversion des images en nuance de gris. Également, quelques fonctions de conversion d'image utilisant le framework Accelerate qui appelle très certainement des fonctions intrinsèques NEON ou du DSP.

Côté OpenCV, peu d'optimisations NEON ont été réalisés. Les développeurs ont préféré favoriser les optimisations réservées aux puces nVidia Tegra3 par le biais d'un sdk conçu par nVidia. On trouve toutefois des essais récents d'optimisation NEON sur la branche de dev d'OpenCV réalisé sur des opérations matricielles et quelques opérations basiques de traitement d'image, mais les gains de performance sur les fonctions optimisées est parfois nul. Ceci est probablement dû au fait que ces instructions utilisent des registres séparés, et que certaines fonctions optimisées doivent effectuer un trop grand nombre de transfert de données entre ces registres.

4.7.2 Optimisations mémoire

La première version du moteur de reconnaissance d'images ne pouvait pas charger une base de plus de 2000 images. Au delà, le processus nécessitait plus des 1Go disponibles et se faisait tuer par le système. Il a fallu adapter certains algorithmes et ajuster leurs paramètres pour aboutir à une application pouvant charger plus de 4000 images sur 512Mo.

Au delà des algorithmes, l'optimisation de la mémoire repose selon les langages sur des règles de programmation strictes.

En Java, on pense être protégé par le ramasse-miette, mais ce dernier ne peut pas faire de miracles, surtout dans le cas de références cycliques. Il convient d'utiliser des références faibles pour aider le ramasse-miette. Pour les parties en C++ il a fallu oublier les mauvaises habitudes du Java et être plus rigoureux. L'utilisation d'inspecteurs de code et de profilers tel que Instruments et Intel VTune Amplifier se sont révélés d'une aide très précieuse.

4.7.3 Les optimisations GPU

4.7.3.1 OpenCL

OpenCL permet de mixer du code GPU et CPU. C'est un projet initié par Apple et repris par le groupe Khronos. Il semblerait avoir été utilisé dans certaines bibliothèques d'Apple telle

que CoreImage, mais les développeurs n'ont pas accès au framework. Il est néanmoins possible d'effectuer du développement OpenCL sur un iPhone jail breaké. [1]

L'utilisation d'OpenCL sur Android est possible sur quelques rares terminaux. Le très faible nombre de puces (dans le domaine du mobile) compatibles avec les spécifications d'OpenCL rend cette technologie peu attrayante.

L'utilisation de la bibliothèque Aparapi (un projet initié par AMD) autoriserait plus de souplesse dans l'utilisation d'OpenCL en convertissant automatiquement le bytecode Java en code OpenCL, si la puce graphique est compatible. Mais mis à part quelques expérimentation il n'existe pas de portage officiel pour Android, sans doute dû à l'arrivée de RenderScript. Cette technologie disponible à partir d'Android 4.2, offre la possibilité de réaliser des calculs intensifs et parallélisables sur les différents coeurs du CPU, le GPU ainsi que les DSP. Toutefois le développeur n'a pas la possibilité de forcer l'utilisation du GPU.

4.7.3.2 CUDA

CUDA est la technologie développée par nVidia permettant également d'exploiter le GPU. Elle est très utilisée sur PC, toutefois aucune des puces actuelles nVidia supportent CUDA. Il faudra attendre 2015 avec l'arrivée des puces Logan et Parker pour pouvoir utiliser cette technologie.

4.7.3.3 OpenGL ES 2

OpenGL ES 2 est une API destinée à l'affichage d'éléments 3D, mais pas au calcul. La seule sortie possible est une image, toutefois celle-ci n'est pas forcément destinée à être affichée sur l'écran. Il est ainsi possible d'exploiter les calculs effectués sur le GPU. Deux parties seulement de la pipeline OpenGL ES 2 sont très personnalisables : la gestion des vertices et des fragments via l'utilisation de shaders qui sont des programmes compilés et exécutés par le GPU. Les vertex shaders sont exécutés parallèlement pour chaque vertex du maillage, tandis que les fragments shaders sont exécutés parallèlement pour chaque fragment (chaque pixel, approximativement).

Dans le cadre du traitement d'image, ce sont les fragments shaders que j'ai principalement utilisé. La comparaison de filtres simples (Sobel, flou Gaussien, convertion de couleurs, ...) avec les performances d'OpenCV était très prometteuse. À titre d'exemple, un filtre de Sobel tournait à 14fps avec OpenGL ES contre 5fps avec OpenCV (sur CPU).

Le tracking étant plutôt lent sur l'iPhone4, j'ai donc décidé d'essayer de ré-écrire une fonction de calcul de flow optique via la méthode de Lucas-Kanade sur GPU, d'autant plus que cet algorithme ce parallélise très bien. Le développement de ces shaders fut lent et très laborieux, en partie à cause de la contrainte de ne pouvoir afficher les valeurs de calculs. J'ai donc réalisé un prototype en Octave imitant le code des shaders afin de vérifier étapes par étapes la justesse des images génères. L'implémentation Octave a elle même été validée par une implémentation indépendante de cet algorithme postée sur MathWorks.

Au bout de 2 semaines de développement, les résultats furent très décevants. Le temps de calcul obtenu sur GPU était de 70ms avec 1 itération et 1 niveau de pyramides gaussiennes, pour 65ms avec OpenCV pour 5 itérations et 3 niveaux de pyramides gaussiennes sur une image de taille 352x288 comprenant une centaine de points d'intérêt (cas d'étude). Ces performances exceptionnelles d'OpenCV sont dues au fait que le flot optique n'est pas calculé sur toute l'image mais seulement sur les pixels possédant des points d'intérêt. En effet, avec une image comportant 2535 points d'intérêt répartis uniformément, le calcul du flot optique prend 425ms avec OpenCV. Il est également fort possible que les faibles performances de mes shaders étaient dues à mon inexpérience. Chaque fabriquant de puces propose des outils de profiling et d'optimisation des shaders que je n'ai pas eu le loisir d'essayer. Des performances plus prometteuses auraient certainement été obtenues en utilisant des outils plus adaptés au développement GLSL.

Afin de terminer et finaliser le framework de reconnaissance dans les temps, cette phase d'expérimentation avec les shaders fut avortée. En outre, il n'est pas envisageable de saturer le GPU

[1] Preuve de concept : `https://github.com/linusyang/opencl-test-ios`

de calcul si un client a besoin d'utiliser OpenGL ES pour de l'affichage 3D.

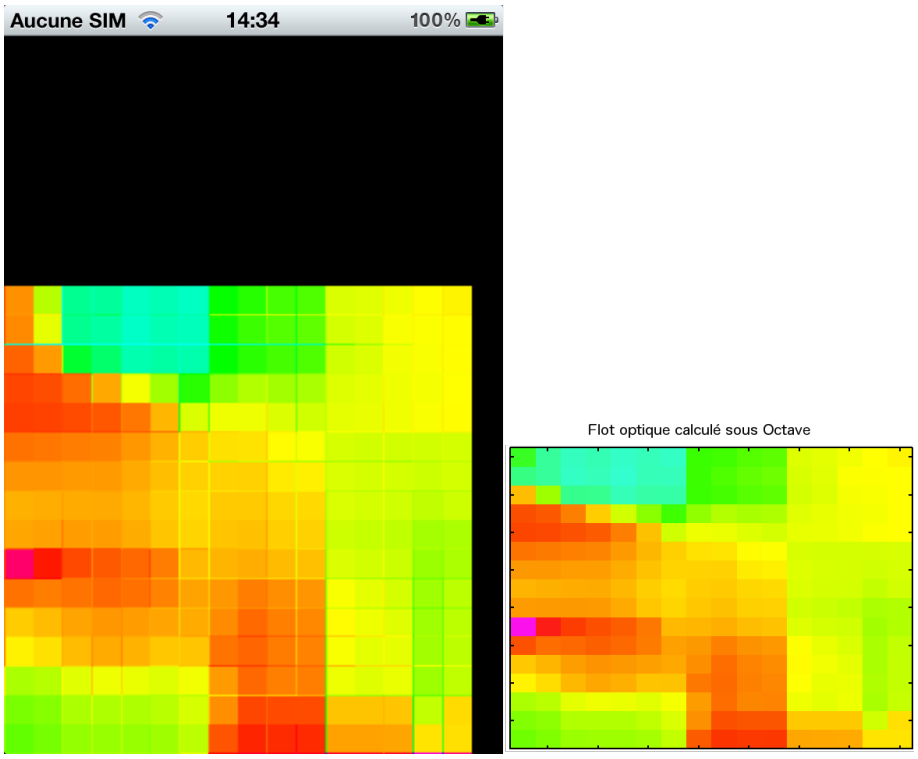

FIG. 16 : *Comparaison entre le flot optique calculé sur une itération par des fragments shaders sur iOS et une version réalisée sous Octave. Les couleurs représentent les angles d'orientation des vecteurs.*

5 Autres travaux

Durant mon stage, j'ai eu l'occasion de travailler ponctuellement sur divers projets décrits dans cette section.

5.1 UCheck

UCheck est la solution de reconnaissance d'images reposant sur le cloud. Bien que déployée depuis un certain temps, la solution avait besoin d'être maintenue. En particulier, des changements importants ont été effectués concernant la communication avec le serveur, les actions de reconnaissance … que j'ai eu la charge de reporter sur l'application Android. Je me suis également occupé de la rédaction de la documentation de l'API, d'un protocole de tests, de la validation de ces tests et du redéploiement de l'application sur Google Play.

J'ai travaillé plus tard sur le backend, et en particulier la réalisation de tests de montée en charge des serveurs de reconnaissance.

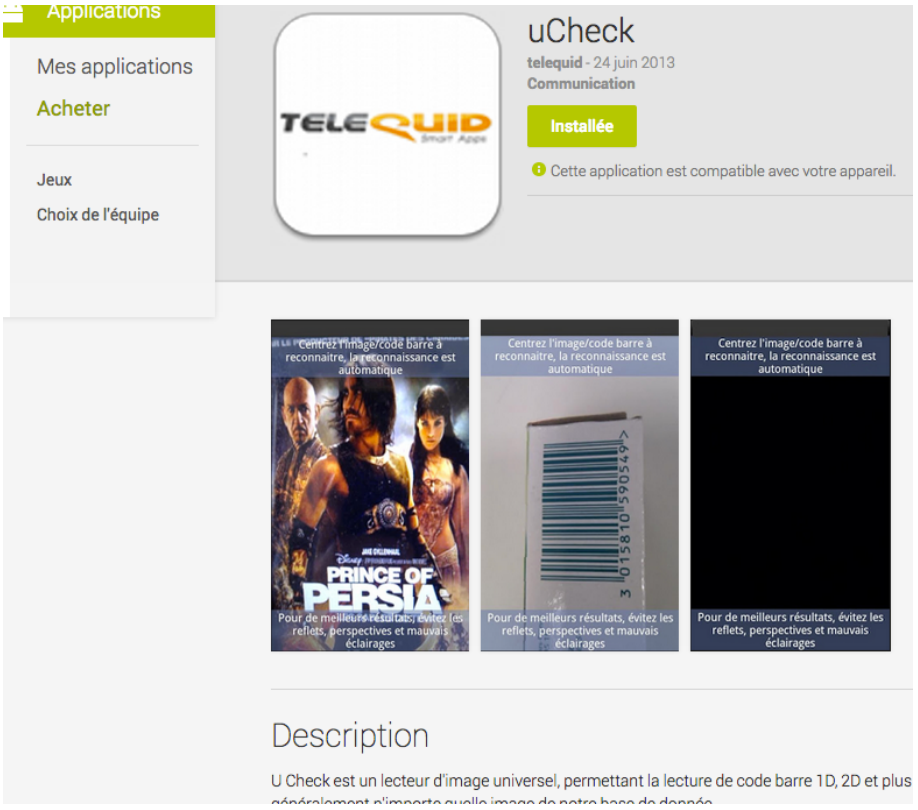

Fig. 17 : *L'application uCheck disponible sur Google Play.*

5.2 Réalité augmentée urbaine

Durant ma visite des laboratoires de recherche de l'Université Polytechnique de Montréal , j'ai fait une parenthèse sur le projet de reconnaissance d'images et de tracking pour aborder un sujet plus ambitieux : la réalité augmentée urbaine. Toutefois, à l'issue d'une phase de recherche et d'expérimentations, il fut décidé d'interrompre le projet car la solution de tracking actuelle n'était pas assez performante sur l'iPhone 4 pour réaliser un prototype satisfaisant, et les autres pistes envisagées était trop ambitieuses pour le temps restant.

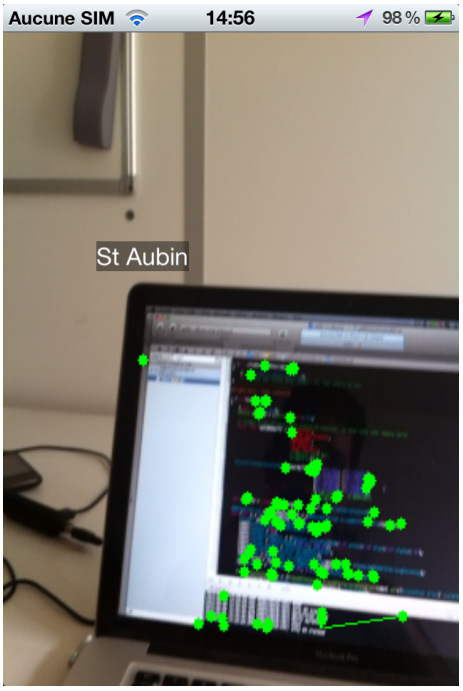

FIG. 18 : *Version préliminaire d'un prototype de réalité augmentée urbaine. Cette application servait à tester la compensation de l'orientation donnée par le magnétomètre avec l'inclinaison de l'appareil, ainsi qu'à afficher des étiquettes (ici "St Aubin") synchronisées avec les différents capteurs, pendant que du suivi de points d'intérêt alimenté à chaque frame par de nouveaux points d'intérêt est effectué.*

6 Conclusion

Durant ce projet de fin d'étude j'ai eu le plaisir et la satisfaction de mener à bien un projet de R&D innovant. Les résultats obtenus sont très convainquants et dépassent les objectifs initiaux concernant le nombre d'images pouvant être reconnues sur le téléphone (jusqu'à 4900 et en moins d'une seconde). Comme dans tout projet, il demeure néanmoins des améliorations concernant les performances du système, en particulier pour le tracking. D'autre problèmes n'ont pas été adressés par choix, comme la gestion du flou, la détection et le tracking simultané de plusieurs images et la recherche partielle visant à être plus robuste dans la discrimination d'images contenant par exemple le même logo.

FIG. 19 : *Impact du flou sur le calcul des appariements et l'estimation de l'homographie. Les points d'intérêt sont dessinés en bleu, les appariements représentés par des lignes, et le quadrilatère vert représente l'image des coins de l'image de référence par l'homographie estimée.*

Avant de commencer mon stage, je n'étais pas familier de l'univers Mac et n'avais jamais réalisé d'applications mobiles. Ces six mois en entreprise m'ont permis d'enrichir considérablement mes compétences techniques via l'apprentissage de nouveaux frameworks et API tel que : Android, Cocoa, GWT, OpenCV, OpenGL ES, QT, … et l'acquisition de nouveaux savoir-faires : obfuscation, reverse-enginering, …Sur le plan scientifique, j'ai pris conscience de l'immensité de l'univers de la reconnaissance d'images ainsi que celui de la réalité augmentée. J'ai découvert l'existence de nombreux algorithmes en traitement d'image, des structures de données complexes, … me permettant de résoudre certaines des problématiques de mon sujet.

Si la réalité augmentée sur mobile en est encore à ses balbutiements, l'avancée constante des technologies permet aujourd'hui de concevoir des algorithmes de vision et de traitement d'images toujours plus complexes et performants et de concevoir un avenir où les téléphones seront capables de détecter et suivre de façon robuste plusieurs objets simultanément, sans monopoliser l'ensemble des ressources disponibles.

L'arrivée dans les années à venir d'OpenVX [2] (le pendant d'OpenGL pour la vision sur calculateur) devrait permettre de contribuer prodigieusement à l'essor des algorithmes de traitement d'images et de jouir d'un plus grand nombre d'accélérations matérielles.

[2] `http://www.khronos.org/openvx`

7 Bibliographie

Références

[AMN*98] ARYA S., MOUNT D. M., NETANYAHU N. S., SILVERMAN R., WU A. Y. : An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM. Vol. 45*, Num. 6 (novembre 1998), 891–923.

[AY11] AMBAI M., YOSHIDA Y. : Card : Compact and real-time descriptors. In *Computer Vision (ICCV), 2011 IEEE International Conference on* (2011), pp. 97–104.

[BETVG08] BAY H., ESS A., TUYTELAARS T., VAN GOOL L. : Speeded-up robust features (surf). *Comput. Vis. Image Underst.. Vol. 110*, Num. 3 (juin 2008), 346–359.

[BFG06] BAY H., FASEL B., GOOL L. V. : Gool. interactive museum guide : Fast and robust recognition of museum objects. In *In Proc. Int. Workshop on Mobile Vision* (2006).

[Bra00] BRADSKI G. :. *Dr. Dobb's Journal of Software Tools* (2000).

[CDF*04] CSURKA G., DANCE C. R., FAN L., WILLAMOWSKI J., BRAY C. : Visual categorization with bags of keypoints. In *In Workshop on Statistical Learning in Computer Vision, ECCV* (2004), pp. 1–22.

[cit05] :. *Matching with PROSAC - progressive sample consensus* (2005), vol. 1.

[CXG*07] CHEN W.-C., XIONG Y., GAO J., GELFAND N., GRZESZCZUK R. : Efficient extraction of robust image features on mobile devices. In *Proceedings of the 2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality* (Washington, DC, USA, 2007), ISMAR '07, IEEE Computer Society, pp. 1–2.

[FSP06] FRITZ G., SEIFERT C., PALETTA L. : A mobile vision system for urban detection with informative local descriptors. In *Proceedings of the Fourth IEEE International Conference on Computer Vision Systems* (Washington, DC, USA, 2006), ICVS '06, IEEE Computer Society, pp. 30–.

[HL05] HARE J. S., LEWIS P. H. : Content-based image retrieval using a mobile device as a novel interface. 64–75.

[IM98] INDYK P., MOTWANI R. : Approximate nearest neighbors : towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (New York, NY, USA, 1998), STOC '98, ACM, pp. 604–613.

[LJW*07] LV Q., JOSEPHSON W., WANG Z., CHARIKAR M., LI K. : Multi-probe lsh : efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB '07, VLDB Endowment, pp. 950–961.

[Low03] LOWE D. G. : Distinctive image features from scale-invariant keypoints, 2003.

[ML09] MUJA M., LOWE D. G. : Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications* (2009), pp. 331–340.

[NS06] NISTER D., STEWENIUS H. : Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2* (Washington, DC, USA, 2006), CVPR '06, IEEE Computer Society, pp. 2161–2168.

[RRKB11] RUBLEE E., RABAUD V., KONOLIGE K., BRADSKI G. R. : Orb : An efficient alternative to sift or surf. In *ICCV* (2011), Metaxas D. N., Quan L., Sanfeliu A., Gool L. J. V., (Eds.), IEEE, pp. 2564–2571.

[TCG*08] Takacs G., Chandrasekhar V., Gelfand N., Xiong Y., Chen W.-C., Bismpigiannis T., Grzeszczuk R., Pulli K., Girod B. : Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *Proceedings of the 1st ACM international conference on Multimedia information retrieval* (New York, NY, USA, 2008), MIR '08, ACM, pp. 427–434.

[TD09] Taylor S., Drummond T. : Multiple target localisation at over 100 fps. In *British Machine Vision Conference* (September 2009).

8 Glossaire

8.1 Scientifique

Terme	Définition
Détecteur	Anglicisme qui désigne un algorithme de calcul de points d'intérêts
Extracteur	Anglicisme qui désigne un algorithme de calcul de descripteurs
FLANN	(Fast Library for Approximate Nearest Neighbour) Bibliothèque de recherche rapide de voisins approximatifs dans des espaces de grande dimension. En fonction du jeu de données, la bibliothèque sélectionne un algorithme ainsi que des paramètres. Les structures de données utilisées peuvent être en autre des arbres-kd aléatoires ou des arbres k-means hiérarchiques.
Flot optique	champ de vecteur de mouvement apparent entre deux images consécutives.
Homographie	Transformation à 8 degrés de liberté pouvant d'écrire entre autre toutes les projections d'une caméra du modèle sténopé. Les applications homographiques ne conservent que les droites.
LMEDS	(Least MEDian of Sqares) algorithme d'estimation de paramètres basée sur la minimisation d'un problème non linéaire.
LSH	(Local Sensitivity Hash) algorithme de recherche de vecteurs voisins basé sur des tables de hashage construites de façon à ce que leur probabilité de collision soit élevée lorsque les vecteurs sont proches.
ORB	(Oriented FAST and Rotated Brief) détecteur et extracteur basés sur FAST et BRIEF
Point d'intérêt	Point d'une image représentant souvent un coin qui peut être déterminé de façon robuste, de façon indépendante (jusqu'à une certaine limite) des variations de changement d'échelle, d'illumination, de rotation et d'angles de vue.
RANSAC	(RANdom SAmple Consensus) Algorithme itératif d'estimation de paramètres dans un problème sur-contraint contenant des données aberrantes.
Transformation affine	Transformation qui conserve les droites et les rapports de distances.

8.2 Technique

Terme	Définition
ABI	(Application binary interface) spécifie le format des fichiers binaire résultatnt de la compilation (exemple les fichiers .o). Ce format est spécifique au système d'exploitation, et à l'architecture du processeur.
APK	(Application PacKage) est le format de fichier des applications sous Android.
ARC	(Automatic Reference Counting) est un système de gestion automatique de la mémoire. A l'instar de Java où un ramasse miette tourne en tache de fond dans la JVM pendant l'exécution du code, l'ARC est statique. C'est à dire que c'est à l'étape de la compilation que le code source est analysé et que des instructions supplémentaires pour libérer la mémoire ou retenir des références sont ajoutées.
ARM	(Advanced Risc Machine) désigne une famille de processeurs à instructions réduite et basse consommation. Les processeurs ARM font généralement partie intégrante de SoC.
AWS	(Amazon Web Services) plateforme de cloud computing d'Amazon.
Back-office	Application d'arrière-boutique, offrant une interface utilisateur réservé à des administrateurs qui permet de modifier du contenu qui sera affiché par un autre application qui peut être un service web, ou une application mobile. Un back-office est réservé à des utilisateurs intermédiaires.
Backend	Ensemble des logiciels qui fournissent les données au backend. Souvent des serveurs effectuant des calculs et en interaction avec des bases de données.
Cocoa	Framework OS X et iOS offrant un vaste panels de bibliothèques permettant d'effectuer de la capture vidéo, de la communication réseau, du traitement d'image sommaire, du parsing, des interfaces graphiques, mais surtout l'ensemble des types objets de base et arborescents.

DSP	(Digital Signal Processor) unité de traitement du signal numérique. L'utilisation des instructions du DSP permet par exemple de calculer matériellement des transformées de Fourrier.
EABI	(Embedded Application Binary Interface) ABI sur plateforme embarquées.
FPU	(Floating Point Unit) unité de calcul flottant simple et double précision.
Front-office	Application destinée aux utilisateurs finaux.
Frontend	Interface entre l'utilisateur est le backend.
GLSL	Langage de programmation dérivé du C permettant d'écrire des shaders pour OpenGL.
GWT	(Google Web Toolkit) framework Java permettant de générer des interfaces utilisateurs en HTML/AJAX/Javascript.
ISP	(Image Signal Processor) puce de traitement d'images, présente souvent sur le SoC du processeur. L'ISP est utilisée notamment pour l'auto-focus, la balance des blancs, l'exposition auto, la compression JPEG …
MIPS	(Microprocessor without Interlocked Pipeline Stages) architecture de processeurs RISC utilisés principalement pour les systèmes embarqués.
NEON	unité de calcul de type SIMD permettant l'accélération de calculs vectoriels. C'est le pendant des instructions SSE sur x86.
OpenCV	(Open Source Computer Vision Library) bibliothèque C++ (avec des bindings Java, Python, Matlab, ...) de vision assistée par ordinateur et de traitement d'image incluant des fonctions d'apprentissage automatique et deiverses structures de données. C'est un projet initié par Intel, actuellement open source. La plupart des fonctions possèdes divers optimisations CPU voire GPU.
ORM	(Object Relational Mapping) fournit une traduction des entrées d'une table d'une base de données sous forme d'objets et du schéma en classe.
REST/RESTful	(Representational state transfer) C'est une architecture logicielle pour des systèmes distribués de type client-serveur sans état avec diverses propriétés. Un exemple typique sont certaines API de Google où on peut envoyer une requête HTTP à un serveur et on reçoit la réponse sous forme d'un fichier XML ou JSON.
RISC	(Reduced Instruction Set Computing) famille de processeurs comportant un jeu d'instructions très réduit mais optimisé. C'est la famille la plus courante en système embarqué.
Shader	Programme compilé et exécuté sur le GPU permettant de modifier finement le comportement d'une pipeline de rendu 3D telle que OpenGL ou DirectX.
SoC	(System on Chip) circuit intégré regroupant divers composants tels qu'un processur, un GPU, de la RAM, un ISP, un co-processeur, , un DSP, …de même que les micro-controlleurs.
SIMD	
VCS	(Version Control System) système de gestion de révisions permettant à une équipe de développeurs de gérer les modifications (fusion, historiques, versions, …) apportées à un projet. Parmi les logiciels de VCS les plus connus on citera SVN et Git.
VFPU	(Vector Floating Point Unit) unité de calculs flottant vectoriel. Les calculs sont toutefois effectués de façon séquentielle sur les éléments des vecteurs, rendant les VFPU obsolètes depuis l'arrivée des instructions NEON.