École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
5e année
2015 - 2016

Projet ASR

Traitement d'images en temps réel

Encadrants
Mathieu Delalandre
mathieu.delalandre@univ-tours.fr

Université François-Rabelais, Tours

Étudiants
Richard Barruet
richard.barruet@etu.univ-tours.fr
Julien Amouriq
julien.amouriq@etu.univ-tours.fr

DI5 2015 - 2016

Version du 11 janvier 2016

Table des matières

Table des figures

Introduction

Nous travaillons dans le cadre d'un projet de lecteur d'étiquettes de bouteille de vin. Depuis l'API 21 (correspondant à Android 5), google a sorti une nouvelle API de développement pour la caméra. L'API *camera* est donc remplacée par l'API *camera2*.

Dans le cadre le la lecture d'étiquettes de bouteilles de vin, il faut optimiser l'accès à la caméra pour garantir un haut taux de rafraîchissement. Dans cette optique nous allons étudier les moyens d'optimisation offerts par la nouvelle API de Google.

Nous verrons dans un premier temps la mise en place du développement , puis L'API Camera 2 et enfin les résultats d'expérimentation.

Mise en place du développement

1.1 Matériel requis

Pour tester notre application utilisant le service Camera2, nous avons besoin d'un téléphone équipé du système Android en version 5.0 (Lollipop). La version 5.0 fournit l' API 21 dans laquelle est apparue pour la première fois le package android.hardware.camera2 .

Le samsung Galaxy S4 qui nous a été prêté était livré à l'origine avec Android 4.2.2. Il a donc fallut le mettre à jour.

1.2 outils de développement

Pour réaliser notre projet, nous avons mis utilisé plusieurs outils intégrés à Android Studio, l'environnement de développement intégré pour Android. En particulier nous avons utilisé le logcat.

Lorsqu'une application est exécuté sur Android Studio, le logcat permet de visualiser les résultats de l'éxécution, même si celle-ci est effectué sur un téléphone par cable USB. C'est très pratique pour récupérer facilement les résultats d'expérimentations : dans le code source, pour écrire dans le logcat, il faut utiliser l'instruction Log.d(mot_clef,contenu_log). Le mot_clef permettra de trier le logcat, et le contenu est le contenu à afficher dans le logcat.

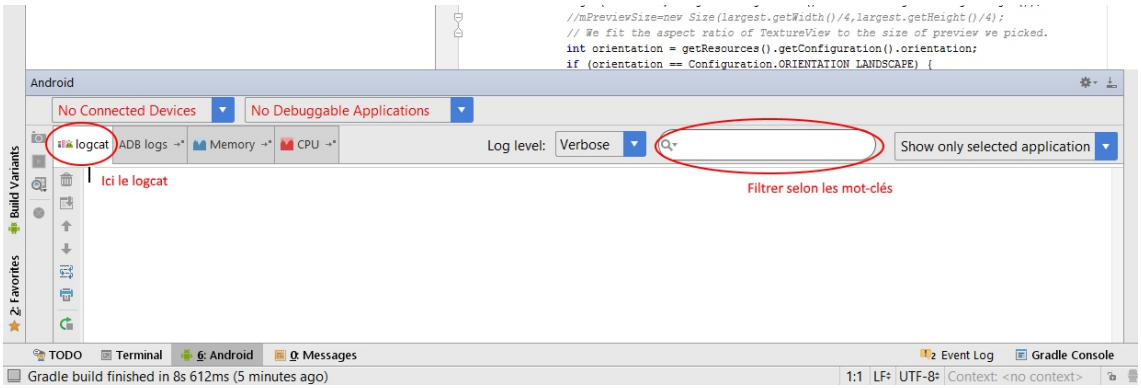

FIGURE 1.1 – Anatomie du LogCat (Android Studio)

Il est possible de filtrer le contenu du LogCat selon des mot-clefs. *En effet lors de l'écriture d'une chaîne de caractères dans le LogCat, le programmeur doit y associer un mot clé (cf plus haut). Cela permet de simplifier la lecture des logs car par défaut tous les logs sont mélangés dans la même fenêtre.

L'API Camera2

L'API Camera2 a été introduite par Google fin 2014 avec Android 5.0 (Lollipop). Elle a pour objet de fournir des classes d'accès standardisées aux appareils photos des téléphones portables, indépendamment des caractéristiques matériel du téléphone.

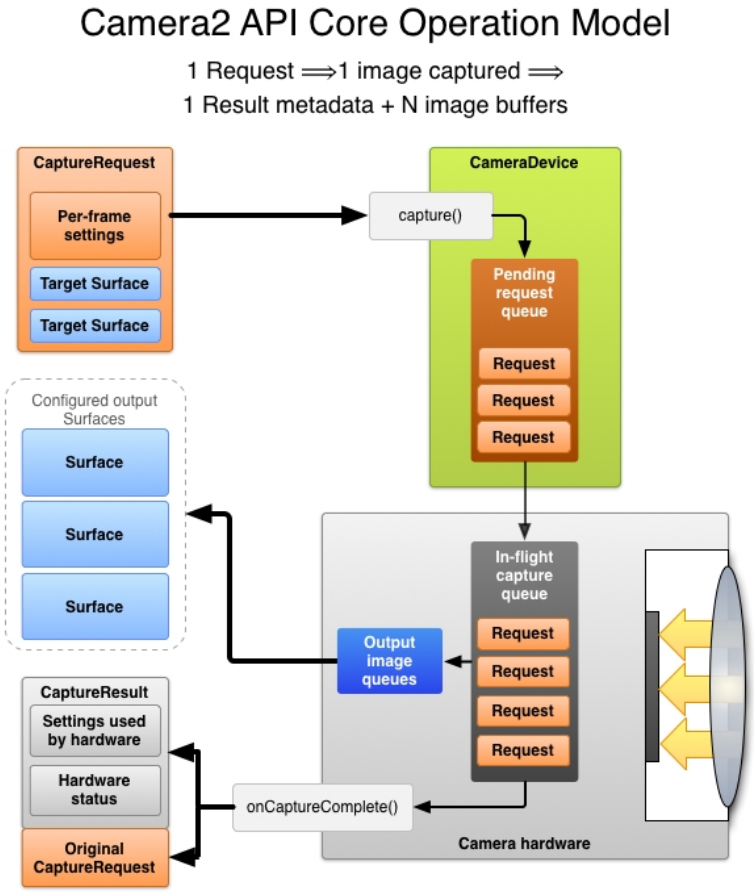

FIGURE 2.1 – Procédé de capture d'image avec l'API Camera2 [1]

L'API permet de définir des objets *CameraDevice* désignant les différentes caméras disponibles sur l'appareil. Chaque *CameraDevice* peut alors recevoir (ou créer) des *CaptureSession* contenant des requêtes *CaptureRequest*, correspondant aux "commandes" faites à la caméra correspondante. Une file d'attente est créée dans laquelle sont envoyées les requêtes. Ces *CaptureRequest* sont équipés de *flags* permettant de configurer la plupart des paramètres de la caméra :

— L'activation automatique (ou pas) de composants (flash, focus...)
— Les pré-traitements intégrés (Nuances de gris, binairisation, application de filtres...)
— ...

Ces paramétrages sont limités par les possibilités de la caméra ciblée. Demander une option indisponible ne provoquera généralement pas de comportement d'erreur, mais provoquera un comportement par défaut

(l'absence de l'option demandée)

Chaque session peut créer des requêtes "répétées", qui s'exécutent indéfiniment (utilisé pour capturer des vidéos). Il faudra aussi préciser une ou plusieurs cibles pour la capture. Ces cibles peuvent être :

— Une surface (*SurfaceView* ou *SurtaceTexture*) pour l'affichage direct dans l'application
— Un objet *ImageReader* intermédiaire, permettant l'accès aux données capturé, utilisé pour leur sauvegarde.

Lors de la capture, des *CaptureResult* sont générés, contenant les informations relatives à la capture et a son paramétrage (défini dans les *CaptureRequest*).

L'image elle est envoyée directement vers les cible désignées. Dans le cas d'une *Surface*, l'image est directement affichée. Dans le cas d'un *ImageReader*, un buffer contient les données de l'image peut être accédé, laissant aux développeur la liberté d'exporter les données.

Résultats d'expérimentations

La littérature sur l'API camera2 n'étant pas encore très fournie (et pour cause, c'est une technologie très récente !), et la documentation officielle n'étant pas toujours très didactique, nous avons avons mené une batterie de tests pour mieux comprendre le fonctionnement de l'API. Il s'agit donc d'un travail de rétro-ingénierie.

3.1 Prise en main

Nous avons pris le parti de reprendre et modifier un exemple de code standard livré avec Android Studio ("File>Import Sample..."). Il s'agit de l'exemple Camera2Basic

Nous avons commencé par étudier les différents mode de capture de la caméra : nous avons modifié la résolution et le type d'image (couleur ou nuances de gris) temps moyen nécessaire à la transmission de 30 images par la caméra. Cette valeur évolue selon le paramétrage de la caméra. La méthode de calcul est la suivante : On enregistre le temps système, puis on capture 30 images et on enregistre à nouveau le temps système. On peut en déduire le nombre d'images par secondes à en utilisant la fonction $f(x) = \frac{1}{\frac{x}{30}} = \frac{30}{x}$.

	image en nuances de gris	image en couleurs	image en couleurs (négatif)
basse résolution	2,078 *(14,4 ips)*	2,071 *(14,5 ips)*	2,137 *(14 ips)*
haute résolution (best available =...x...)	1,285775 *(23,3 ips)*	1,40825 *(21,3 ips)*	2,15827 *(13,9 ips)*

TABLE 3.1 – Récapitulatif du temps pour avoir 30 images en mode interruption (en secondes)

Dans le tableau 3.1 ci-contre, on peut voir les résultats pour 200 mesures sur chaque cas : les meilleures performances sont obtenues en moyenne avec une image en nuance de gris haute résolution. Les mesures ont également servi à tracer des courbes de répartition (figures 3.1, 3.2,3.3,3.4).

Sur la figure 3.1, d'abord, on voit que le temps pour capturer trente images HD en couleurs varie entre 1 et $2,5$, avec l'essentiel des mesures sous les 1,8s (c'est-à- dire au moins 17 ips et 30 ips au maximum). La figure 3.2 permet de comparer les différentes configurations pour une image HD : en rouge l'image négative, très resserrée autours de 2,05ms ; en bleu l'image en couleur qui offre de meilleures performances maximales mais des effectifs répartis entre 1,05 et 2 ; et en vert l'image en nuances de gris davantage resserrée autours de 1,15. L'image en nuance de gris haute résolution est le meilleur choix en mode haute définition. Le fait que les captures en nuance de gris et couleurs donnent des résultats similaires indique qu'il y a une gestion matérielle du flux, tandis que le flux négatif lui n'est pas supporté directement par la caméra et induit des calculs supplémentaires. Cela expliquant la grande différence de performance entre ce mode et les deux autres.

Comparons maintenant ces résultats avec d'autres résultats obtenus en calculant le nombre d'images par secondes à chaque image et non toutes les 30 images (figure 3.5). Pour convertir les valeurs d'images par secondes en temps pour afficher 30 images, nous avons appliqué la formule $f(x) = \frac{30}{x}$. Force est de constater que les deux courbes ne se ressemblent pas puisque l'essentiel de la courbe 3.1 se situe entre 1,05 et 1,55 alors que l'essentiel de la courbe 3.5 se situe lui entre 0,6 et 1.

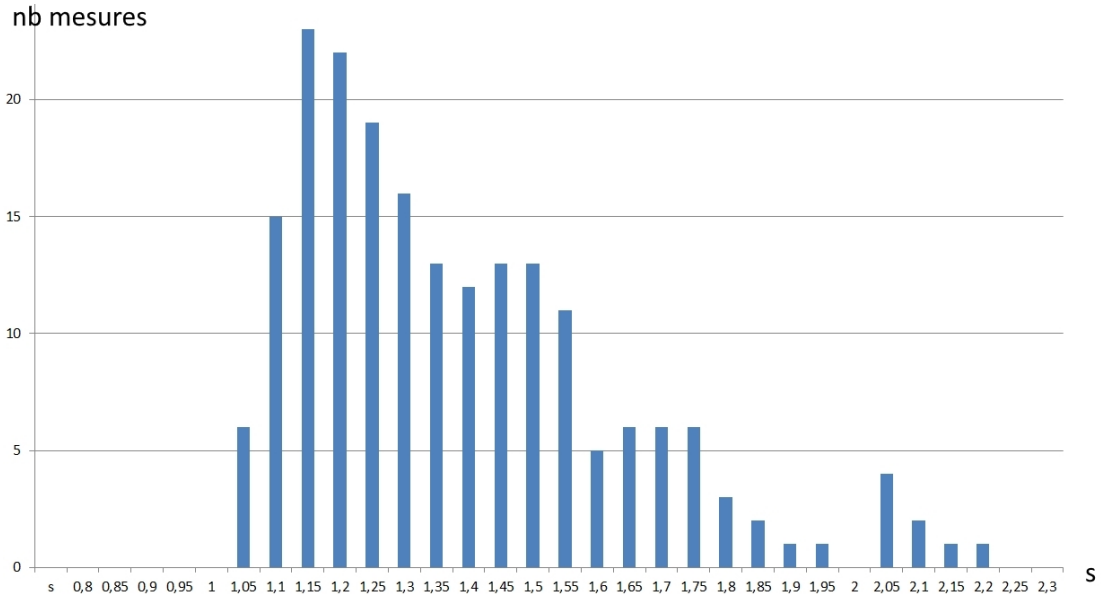

FIGURE 3.1 – Nombre de mesures en fonction du temps de capture (hd couleur)

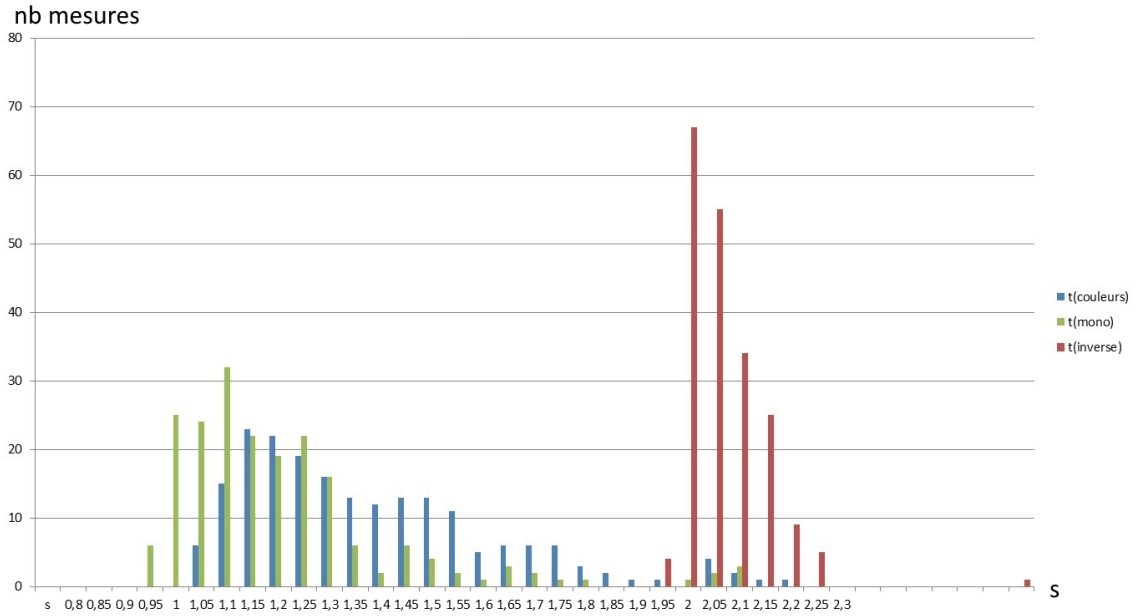

FIGURE 3.2 – Nombre de mesures en fonction du temps de capture , en hd, et selon différents critères

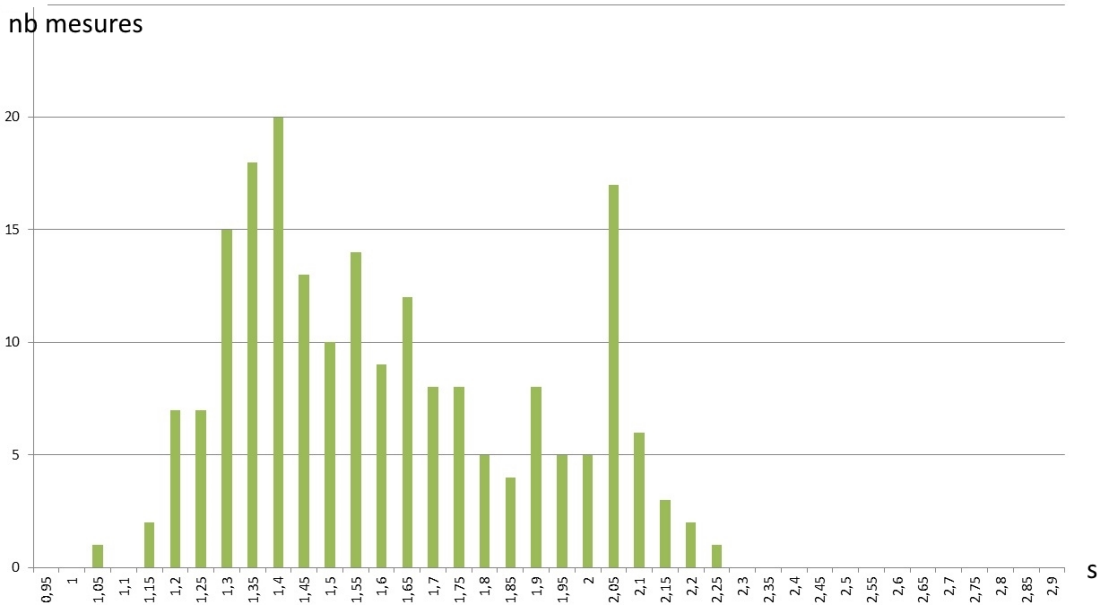

FIGURE 3.3 – Nombre de mesures en fonction du temps de capture, en basse résolution (640x480)

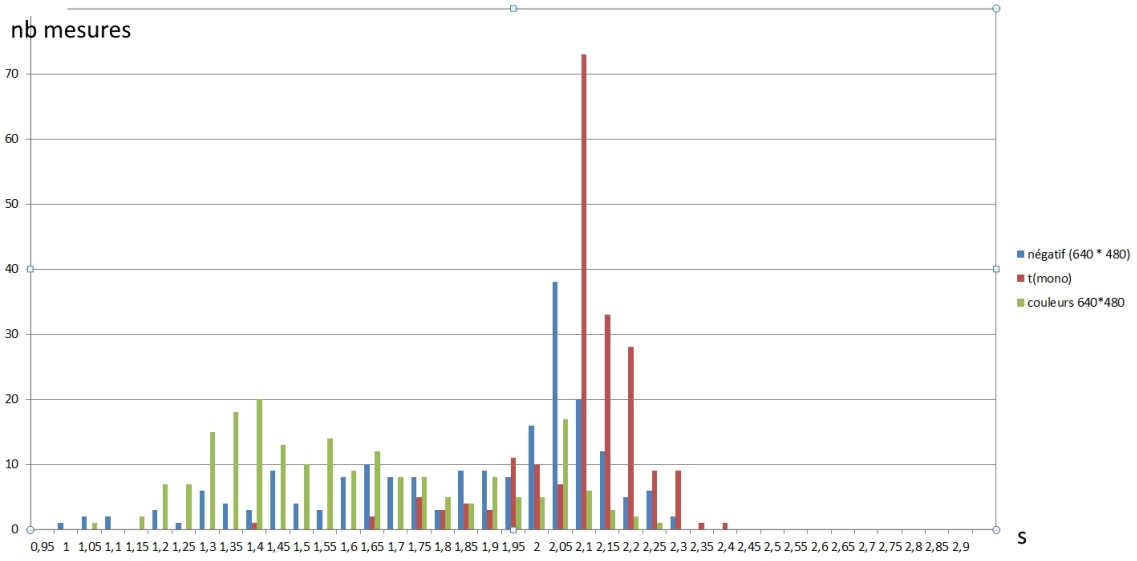

FIGURE 3.4 – Nombre de mesures en fonction du temps de capture,en basse résolution sous différent critères

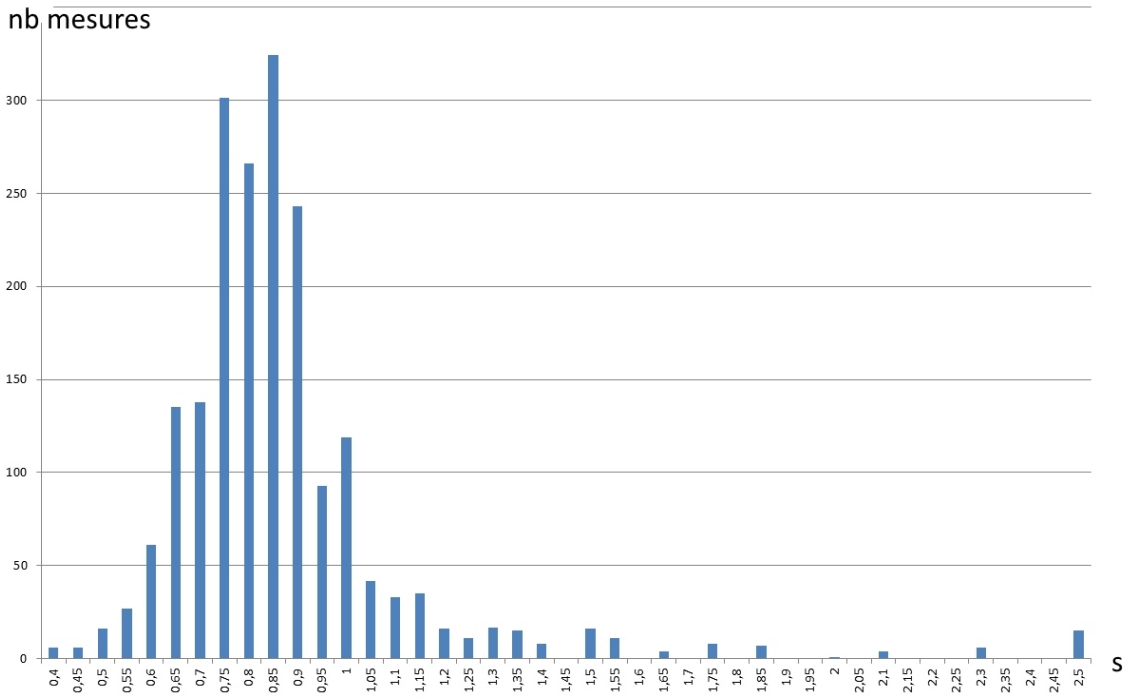

FIGURE 3.5 – Nombre de mesures en fonction du temps de capture de 30 images pour une vidéo en HD couleur (mesuré image par image)

3.2 Ajustement du protocole de mesure

Avant de continuer plus en avant dans la recherche, nous devons comprendre d'où vient un tel écart alors qu'en principe la courbe 3.1 devrait présenter de meilleurs performances que la courbe 3.5. En fait on y fait trente fois moins de mesures, cela peut expliquer une imprécision de mesure . Voici les hypothèses prises en compte :

— la transmission de chaque valeur sur le logcat par usb aussitôt la mesure faite consomme des ressources et perturbe la mesure, il faut donc enregistrer les valeurs en mémoire et ne les rapatrier qu'à la fin du test. Nous avons décidé d'enregistrer les valeurs dans un fichier texte pour se passer complètement du logcat qui perturbe le fonctionnement de l'application.

— Il serait intéressant d'utiliser les deux méthodes de calcul pendant une même prise d'image pour éviter les variations issues de la camera entre deux captures.

Nous avons mis en place ces deux points et avons procédé au nettoyage de la mémoire avant toute mesure(à l'aide d'un outil de "boost"). Il apparaît dans les mesures que si l'on groupe les valeurs prises image par image par paquets de trente et qu'on les compare avec les mesures faites toutes les trente images, on obtient les mêmes mesures (cf table 3.2).

	image par image	par paquets de 30 images
temps moyen mesuré (s)	1,8713 *(16,03 ips)*	1,8710 *(16,03 ips)*

TABLE 3.2 – Comparaison des deux mode de prise de mesure

Désormais pour faire nos mesures nous ne transférerons plus nos résultats directement sur le logcat et nous effectuerons un nettoyage de la mémoire avant de lancer les séries de mesures. Enfin nous prendrons 6000 mesures au lieu de 200.

3.3 Suite des expérimentations

Nous avons créé une nouvelle application de test qui mesure en même temps les temps entre chaque image et les temps entre 30 images pour nous assurer que le mode de comptage des images n'influe pas sur les performances Les figures 3.6 et 3.7 présentent les résultat d'une *même* capture vidéo. La seule différence réside dans le mode de calcul puisque sur la figure 3.6 on considère chaque image individuellement alors que sur la figure 3.7 on considère des paquets de 30 images. Le regroupement en paquets de 30 images doit théoriquement diminuer l'erreur de mesure. En effet si à chaque mesure on fait une erreur, attendre de capturer plusieurs images divisera l'erreur du temps d'affichage d'une image par le nombre d'images mesurées (dans notre cas par 30).

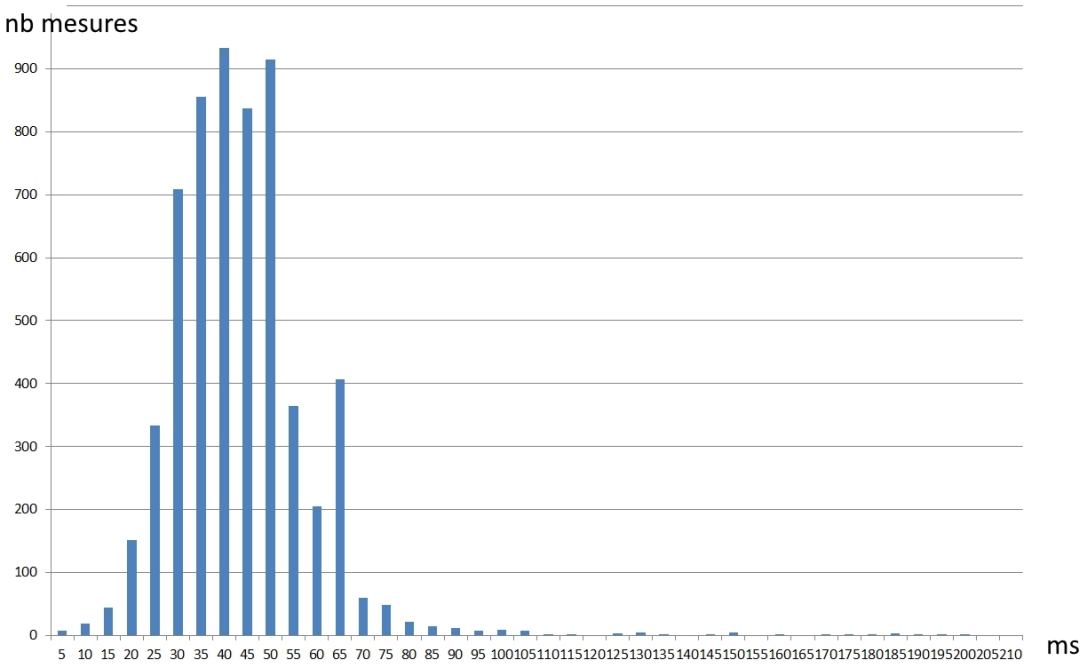

FIGURE 3.6 – Nombre de mesures en fonction du temps de capture pour une vidéo en HD couleur (mesuré image par image)

Que remarque-t-on en étudiant les graphiques ? Le graphique sur la figure 3.6 semble plus précis que celui sur la figure 3.7. Et c'est normal car la figure 3.7 contient 30 fois moins d'échantillons. En mesurant plus de valeurs on devrait affiner cette courbe. Nous avons essayé de prendre davantage de valeurs cependant nous n'avons pas réussi à prendre trente fois plus d'images : le temps de capture est trop grand .

3.4 DMA

Forts de toutes ces expériences, on souhaite vérifier que le téléphone prend les photos en passant par un accès DMA ou assimilé. On se propose maintenant de charger au maximum la mémoire du téléphone avec des applications et de prendre des mesures : si les performances n'évoluent pas ou peu on pourra en déduire que le téléphone prend les photos en passant par un accès DMA ou assimilé.

DMA est l'acronyme anglais d'accès direct à la mémoire . Il s'agit d'une fonctionnalité permettant au matériel d'accéder dans certaines conditions à la mémoire vive sans passer par le processeur[3] .Comme cela est schématisé sur la figure 3.8, la DMA permet de décharger le processeur des opérations d'entrée-sortie qu'il devrait normalement gérer (figure 3.8,"PIO").

Avec la DMA, le processeur initialise le transfert puis est libre d'effectuer d'autres tâches : le transfert est délégué au contrôleur DMA et continue sans l'intervention du processeur. A la fin du transfert, le

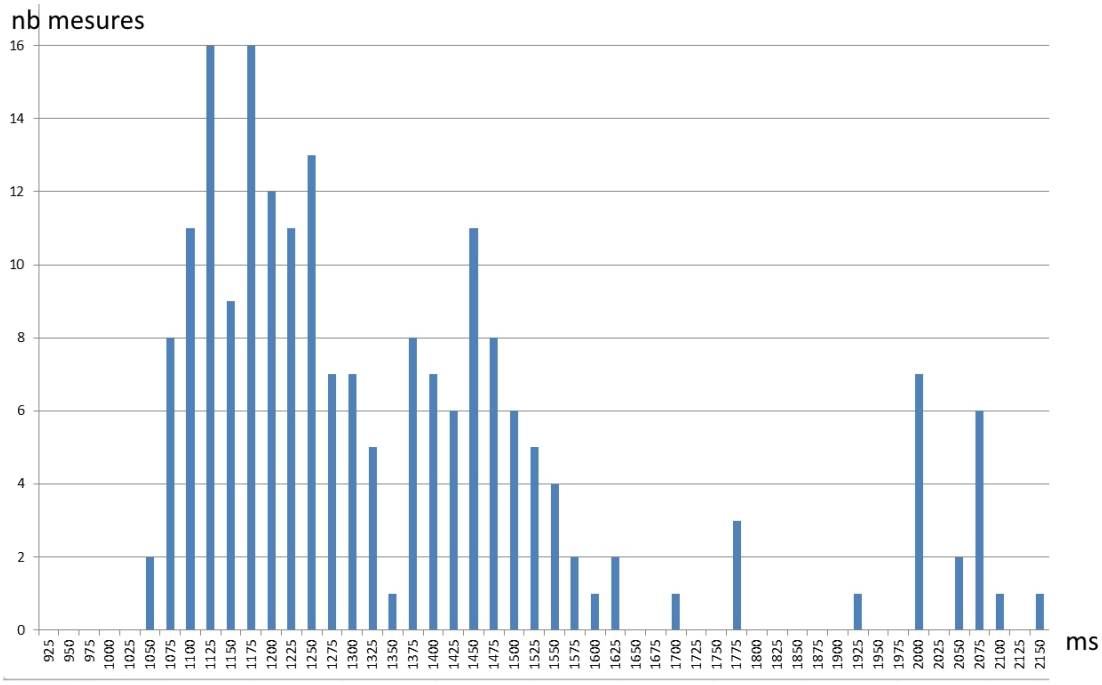

FIGURE 3.7 – Nombre de mesures en fonction du temps de capture pour une vidéo en HD couleur (mesuré par lots de 30 images)

processeur reçoit une interruption de la part du contrôleur DMA. Dans le cas des téléphones Android, cela permet d'optimiser les captures vidéos : même si le processeur est très chargé, cela a très peu d'influence sur le transfert de l'image de la caméra à la mémoire par DMA. La capture de flux vidéo en accès DMA serait une garantie de fluidité dans le cadre de la lecture d'étiquettes de bouteilles de vin : le traitement du flux vidéo aurait d'avantage de ressources processeur.

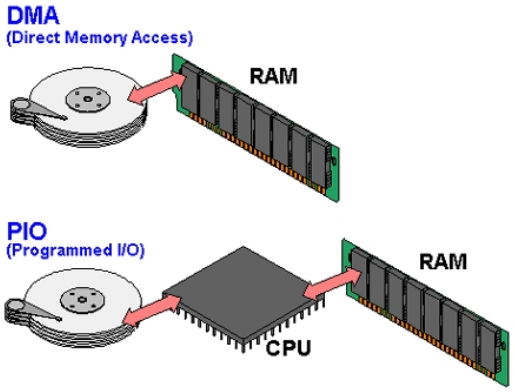

FIGURE 3.8 – Comparaison entre transfert avec et sans DMA [2]

Sur la figure 3.9, on peut voir que la répartition des temps de mesures entre deux images est centrée en 65. Le calcul de la moyenne donne 65,9 ms entre chaque capture en moyenne ; alors que sur la figure 3.6 la même courbe , toujours sur 6000 mesures est centrée en 45. Il y a donc un décalage d'une vingtaine de millisecondes lorsque la mémoire du téléphone est chargée. Ce n'est pas un écart très important,et la répartition des mesures reste très propre, *on peut donc en conclure qu'il y a bien un mécanisme de DMA encapsulé dans l'API.*

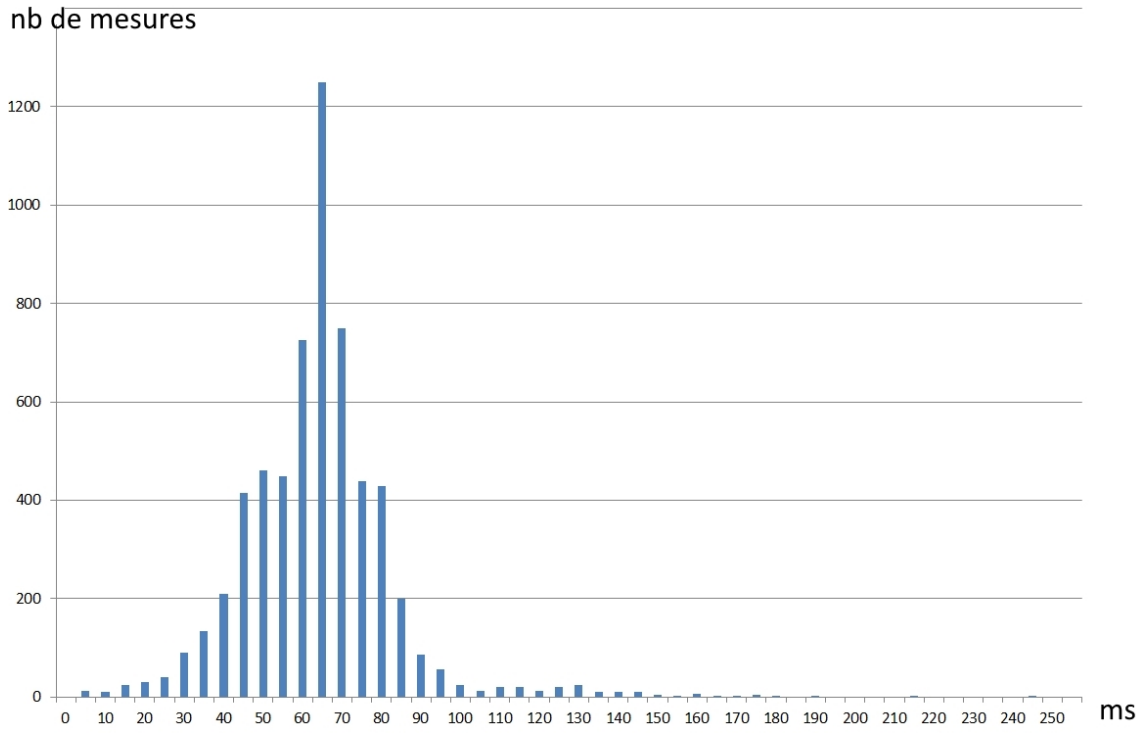

FIGURE 3.9 – Nombre de captures en fonction du nombre de ms entre deux images avec un réglage HD sur un téléphone volontairement surchargé

Préparations pour le traitement d'image

4.1 Accès au flux de données

Comme expliqué dans le chapitre 2, le résultat d'une capture peut être dirigé directement vers une *Surface*, pour l'affichage en temps réel du flux de la caméra. Cependant cette méthode ne permet aucun accès aux données. Pour accéder aux données il faut indiquer comme cible de la requête un *ImageReader*

```
captureBuilder.addTarget(mImageReader.getSurface());
```

Pour récupérer le contenu d'une capture, on récupère l'événement correspondant à la disponibilité d'une image dans le *ImageReader*

```
private final ImageReader.OnImageAvailableListener mOnImageAvailableListener
                = new ImageReader.OnImageAvailableListener() {

        @Override
        public void onImageAvailable(ImageReader reader) {
                //Interaction avec ImageReader
        }
};
```

L'objet *ImageReader* contient une *Image*, laquelle contient un *ByteBuffer*. Pour assurer l'affichage en temps réel, il faut renvoyer ces données vers la surface d'affichage, après avoir effectué les traitements souhaités.

Conclusion

En conclusion, on a étudié le fonctionnement de la camera sous Android, avec utilisation de l'API 21. Cette API encapsule un accès à la caméra en mode DMA, ce qui autorise le traitement en temps réel du flux vidéo. Une légère baisse du taux d'images par secondes est observé lorsque la mémoire du téléphone est très chargée, il faudra donc gérer la mémoire avec parcimonie pour rester sous la barre des un trentièmes de seconde entre chaque image et ainsi offrir une bonne fluidité à l'utilisateur. Tous les modes de captures ne sont pas supportés matériellement : si les captures couleurs et nuance de gris le sont, la capture en négatif ne l'est pas et induit des calculs supplémentaires.

Notons enfin que google a sorti fin 2015 l'API 23 sur la toute dernière version d'Android 6, disponible sur certains appareils. Cette version de l'API propose de nouveaux types de requête : dont les requêtes dites "reprocessable" qui prévoient d'insérer directement un traitement d'image dans le processus de traitement du flux. Cette API pourrait permettre une encore meilleure exploitation du matériel.

Annexes

A.1 Environnement de traitement d'image

Le code Java n'est pas particulièrement optimisé, en particulier pour les calculs intenses et/ou parallélisés. Heureusement, l'API Android dispose de plusieurs solutions pour accélérer de tels calculs et les rendre utilisable sur appareil mobile.

A.1.1 NDK et Renderscript

NDK est l'outil de développement de code natif permettant d'intégrer du code *C* dans un projet Java pour Android. Ce code, dit *natif*, est plus optimisé que le code java classique et peut se montrer plus adapté quand un programme effectue des taches lourdes qui provoqueraient un ralentissement qui pourrais se montrer critique pour une application visant à fonctionner en temps réel.

RenderScipt est un outil similaire permettant de créer des *script*s aussi codés en *C*, optimisés pour les calculs intensifs et le traitement d'images. RenderScript possède toutefois quelques différences clés avec NDK :

- La transmission des données entrantes et sortantes est gérée par des *Allocations* destinées à subir le traitement du script.
- Utilisé pour les calculs intensifs et le traitement d'images, RenderScript est présenté comme plus optimisé que NDK.
- Les scripts ne subirons qu'une compilation unique, qui sera valide et adaptée pour tous les appareils supportant RenderScript.

RenderScript à l'inconvénient d'augmenter la complexité du code, et se montre moins polyvalent que NDK. Mais il reste tout indiqué pour l'utilisation qui nous intéresse.

A.2 RenderScript - Installation et fonctionnement

La documentation de l'API [1] Avant de pouvoir utiliser RenderScript, il convient de configurer les projet pour déclarer son utilisation et la version de RenderScript utilisée. Cela se fais en indiquant dans le fichier *build.gradle* :

```
renderscriptTargetApi 21
```

Dans le code Java, les fonctionnalités RenderScript requière le package associé :

```
import android.renderscript.*;
```

Les scripts de traitements sont réalisés en C dans un fichier *<script>.rs* (Exemple A.2).

Un script dois contenir au moins une fonction *kernel* correspondant au code que l'on souhaite exécuter depuis le code Java. Cette fonction accepte au minimum un paramètre *entrée* et produisant une sortie, et d'autres paramètres optionnels.

Pour utiliser les scripts crées, il faut respecter quelques initialisations :

Le contexte *RenderScript* (normalement unique) dois être initialisé pour pouvoir contrôler le reste de nos objets liés.

1. RenderScript - http ://developer.android.com/guide/topics/renderscript/compute.html

```
#pragma version(1)
#pragma rs java_package_name(com.example.android.camera2basic)
#pragma rs_fp_relaxed

uchar4 __attribute__((kernel)) invert(uchar4 in, uint32_t x, uint32_t y) {
  uchar4 out = in;
  out.r = 255 - in.r;
  out.g = 255 - in.g;
  out.b = 255 - in.b;
  return out;
}
```

FIGURE A.1 – Exemple de script - Inversion des couleurs

Au moins un objet *Allocation* entrée/sortie dois être crée. Ces objets servent à transmettre les données a traiter entre les objet Java Sources et Cibles et les Scripts. Une *Allocation* est généralement créer à partit d'un objet *Bitmap*, mais il est possible de la créer depuis un autre type d'objet.

Les allocation sorties ont aussi l'avantage de pouvoir être envoyées directement à une *Surface* d'affichage. Les Allocations entrées devrons être peuplés avec les données à traiter via leur fonctions *copy*.

Il faut ensuite initialiser les objets *ScriptC* à partir du contexte, correspondant au script crées. Pour notre exemple *nega.rs*, il faut initialiser un script de la façon suivante :

```
ScriptC_nega mScript = new ScriptC_nega(mRS);
```

Il existe aussi un éventail de *ScriptIntrinsic* déjà implémentés, offrant quelques traitements aux développeurs, comme des fonctions de conversion de l'image YUV vers RGB.

Il reste aussi possible d'indiquer les paramètres supplémentaires via les fonctions associées. Pour indiquer le paramètre entier x présent dans notre exemple, il fait utiliser la fonction :

```
set_x(int);
```

Le kernel pourra ensuite être lancé. Chaque exécution est sérialisée, et conservera l'ordre de lancement. Lors de l'exécution, la totalité des *Allocation*s entrée est traitée, et le résultat est transmis vers l'*Allocation* sortie. Il restera donc à extraire les données de ces Allocations vers nos objets Java, puis de détruire le contexte RenderScript lorsqu'il ne sera plus nécessaire.

Problèmes à l'implémentation Malgré la présence d'une documentation officielle présentant les fonctionnalités, il existe peu d'informations concernant leur mise en place, ou l'implémentation de RenderScript et Camera2 simultanément. Nous n'avons donc pas réussi à produire une application mettant en place ces 2 composant simultanément (notamment la transmission vers les Allocation, et leur exportation)

A.3 Autres environnements

NDK et *RenderScript* ne sont pas les seuls environnements permettant du traitement intensif. On peut par exemple mentionner la possibilité d'utiliser un contexte *OpenGL* pour effectuer les opérations graphiques, ou plus simplement d'utiliser du code Java classique si le traitement est suffisamment léger.

Bibliographie

[1] www.frandroid.com. http://www.frandroid.com/android/developpement/230402_comment-lapi-camera2-google-permettra-enfin-faire-belles-photos-android, juillet 2014.

[2] Dma, mai 2015. http://encyclopedia2.thefreedictionary.com/dma.

[3] Accès direct à la mémoire, mai 2015. https://fr.wikipedia.org/wiki/Acc%C3%A8s_direct_%C3%A0_la_m%C3%A9moire.

Traitement d'images en temps réel

Département Informatique
5e année
2015 - 2016

Projet ASR

Résumé : Ce rapport présente l'utilisation de l'API camera 2 pour exploiter le flux d'images de la caméra sous Android. Plusieurs essais ont été effectués pour déterminer la configuration permettant une utilisation optimale du matériel.Au fil des expérimentations on distingue le fonctionnement interne de l'API et les moyens d'optimisation offerts par celle-ci.

Mots clefs : android,camera2,DMA

Abstract: This report present the uses of the Camera 2 API to exploit the image flow from a camera in an Android device. Severals experiments were made to determine an optimal materiel configuration. Over the experiments, the API's inner workings and the optimisation possibilities are distinguished.

Keywords: android,camera2,DMA

Encadrants
Mathieu Delalandre
mathieu.delalandre@univ-tours.fr

Université François-Rabelais, Tours

Étudiants
Richard Barruet
richard.barruet@etu.univ-tours.fr
Julien Amouriq
julien.amouriq@etu.univ-tours.fr

DI5 2015 - 2016