25 mars 2011

MCSB0

Projet de fin d'Etude *INF592*

Vidéo surveillance, Streaming vidéo et contrôle de caméra via Android

- MEMOIRE -

Jerome NAHELOU - Quentin NEBOUT - Romain SOLVE - Fabien QUINTARD

Enseignant encadrant : Yérom-David BROMBERG

Table des matières

Chapitre 1

Etude De l'existant

1.1 Le systéme d'exploitation Android

La premiére version d'Android (1.0) est parue en septembre 2008, développé par Google et l'open Handset alliance. Les différentes versions mettent à jour l'OS, permettant ainsi de corriger les bugs et d'ajouter des fonctionnalités. Voici les derniéres versions disponibles :

- 1.6 (Donut) : Version basée sur le noyaux linux 2.6.29, sortie en septembre 2009.
  - Interface native pour l'appareil photo, la camera et la galerie
  - Permet la selection de plusieurs photos pour la suppression
  - Mise à jour de la detection vocale
  - Mise à jour permettant la recherche dans les favoris, l'historique, les contacts et Internet
  - Mise à jour de la technologie CDMA/EVDO, 802.1x, VPNs
  - Support résolution WVGA
- 2.0/2.1 (eclair) : Version basée sur le noyaux linux 2.6.29, sortie en décembre 2009 /janvier 2010.
  - Réorganisation UI
  - Supporte plus de résolution d'écran
  - Supporte HTML 5
  - Meilleur contraste pour les fonds d'écran
  - Amélioration vitesse Hardware
  - Classe MotionEvent améliorer pour supporter les multiples "touch event"
  - Zoom digital
  - Bluetouth 2.1
  - Microsoft Exchange Server
- 2.2 (froyo) : Version basée sur le noyaux linux 2.6.32, sortie en mai 2010.
  - Optimisations générales de la vitesse, de la mémoire et des performances d'Android OS.
  - Intégration du moteur JavaScript V8 de Chrome dans le navigateur
  - Hotspot Wi-fi et USB Thetering
  - Support des mots de passe numériques et alphanumériques
  - Support de l'upload de fichiers dans le navigateur
  - Support de l'installation d'applications sur la mémoire extensible
  - Support des écrans é haute densité de pixels (320 dpi)
  - Support d'Adobe Flash 10.1
- 2.3 (Gingerbread) : Version basée sur le noyaux linux 2.6.35, sortie en décembre 2010.
  - Support des grands écrans à résolutions extra-larges (WXGA et plus)
  - Support de la VoIP et SIP
  - Support des formats vidéo WebM/VP8, et l'encodage audio AAC
  - Nouveaux effets audio tels que la réverbération, l'égalisation, la virtualisation du casque audio et accentuation des graves
  - Support du NFC
  - Amélioration de la fonction copier/coller et sélection du texte
  - Refonte du clavier virtuel (multi-touch) et de l'autocomplétion

– Garbage collector pour de meilleures performances
– Support de nouveaux capteurs (comme le gyroscope et le baromètre)
– Ajout d'un gestionnaire de téléchargement
– Amélioration de la gestion de l'alimentation et du contrôle des applications
– Passage au système de fichiers ext4

1.2 Les caméras

Il existe de nombreuses caméras ip, ainsi que plusieurs constructeurs, qui proposent plus ou moins de services. Voici plusieurs caméras ptz ainsi que les fonctionnalités proposées.

1.2.1 Caméras 'Axis'

Q1755 50 Hz

La caméra AXIS Q1755 50hz est une caméra recommendé pour surveiller des zones qui demandent une finesse de capture élevé. Cette caméra prend en charge aussi bien les flux H.264 que Motion JPEG à la fréquence d'image maximale (50hz).

- Qualité HDTV (1080i en 16/9)
- Zoom et mise au point automatique
- Stockage local (lecteur de carte SDHC intégré)
- Alimentation par Ethernet
- Fonctions vidéo intelligentes
- Zoom optique 10x, son zoom numérique 12x
- Fonctions jour & nuit

215 PTZ-E

La caméra AXIS 215 PTZ-E est une caméra jour/nuit, résistant aux intempéries, donc recommandé pour surveiller en extérieur.

- Caméra à haute résolution 704x480 pixels (NTSC), 704x576 (PAL)
- Fonction PTZ (Pan Tilt Zoom) : fonction de balayage horizontal, vertical et de zoom instantané sans perte de résolution.
- Flux MPEG-4 et Motion JPEG simultanés
- Sécurité réseau assurée par protection de mot de passe multiniveau, filtrage d'adresses IP, cryptage HTTPS et authentification IEEE 802.1X
- Conformité à la norme de Qualité de service (QoS), pour être certain de disposer de la bande passante nécessaire à la transmission des flux vidéo et des commandes de contrôle sur le réseau
- Prise en charge du protocole Internet version 6 (IPv6), en plus de la version 4 (IPv4)
- Interface de programmation d'applications (API) performante, permettant l'intégration logicielle et comprenant l'API AXIS VAPIX et l'AXIS Media Control SDK. Mémoire flash permettant de charger des applications intégrées.
- Vue Panoramique 360 ° avec retournement automatique

214 PTZ

La caméra AXIS 214 PTZ est une caméra à hautes performances destinée à la surveillance professionnelle.

- Caméra à haute résolution 720x576 pixels (PAL), 720x480 (NTSC)
- Détection de mouvement vidéo intégrée et gestion avancée des événements
- Plusieurs niveaux d'accès utilisateur avec protection par mot de passe, filtrage d'adresses IP, cryptage HTTPS et authentification IEEE 802.1X
- Compatible avec le concept de Quality of Service (QoS) ce qui permet de réserver la capacité réseau allouer à la vidéo et de classer les opérations de surveillance essentielles par ordre de priorité sur un réseau QoS
- Compatible avec les protocoles Internet IPv4 et IPv6, ce dernier permettant de bénéficier d'une quantité plus importante d'adresses IP disponibles

1.2.2 Caméras 'D-link'

DCS-7510

La caméra D-Link DCS-7510 est une caméra d'extérieur à vision diurne et nocturne équipée de LEDs infrarouges, idéale pour les surveillances en continu.

- Filtre anti-infrarouges amovible intégré, pour un rendu des couleurs amélioré le jour et de bons résultats en conditions d'éclairage faible
- LED infrarouges pour détecter les mouvements jusqu'à 50 mètres dans l'obscurité totale
- Objectif auto iris pour une qualité d'image optimale
- Boîtier étanche (certification IP66 de protection contre la pluie et la poussière) avec pare-soleil amovible
- Zoom numérique 16x
- Fonction vidéo mobile 3G, pour diffuser des flux vidéo en direct sur un téléphone mobile ou un PDA 3G
- Flux MJPEG et MPEG-4 simultanés
- Entrée/sortie numérique pour la connexion aux capteurs et alarmes
- Prise en charge d'un système audio bidirectionnel (nécessite un haut-parleur et un microphone externes)
- Port RS-485
- Enregistrement direct sur un boîtier NAS ou un enregistreur vidéo réseau

DCS-5635

La caméra D-Link DCS-5635 est une caméra à hautes performances destinée à la surveillance professionnelle. Pour optimiser la bande passante et améliorer la qualité d'image, la webcam DCS-5635 permet une compression vidéo en temps réel aux formats MJPEG, MPEG-4 et H.264.

- Zoom optique 10x + zoom numérique, pour mieux filmer les détails
- Fonctions d'inclinaison et de rotation, pour une parfaite flexibilité
- Facile à installer et à utiliser, la webcam DCS-5635 peut être montée sur une surface plane ou via une fixation murale
- Port Secure Digital
- Compatible liaison sans fil Wi-Fi b/g/n et support de la surveillance mobile 3GPP
- Caméra à haute résolution 720x576 pixels (PAL), 720x480 (NTSC)

1.2.3 Caméras 'Cisco'

VC 240

Cette caméra à été spécialement créée pour les petites entreprises, qui offre une bonne qualité d'image, résiste à l'eau et la poussière, donc permet une installation interne ou externe.

- accès à des vidéos et du son en direct de votre entreprise où que vous soyez, de jour comme de nuit, à l'aide d'un ordinateur ou d'un téléphone portable connecté à Internet
- intégration d'alarmes, de capteurs de portes, de détecteurs de mouvements et d'autres systèmes de gestion dans votre solution vidéo
- réception d'alertes automatiques (notamment des enregistrements vidéo ou des images fixes) en cas de détection de mouvements dans les locaux
- montage de la caméra en intérieur/extérieur, y compris dans des environnements difficiles, au plafond ou sur les murs
- recherche rapide dans les archives vidéo, configuration et contrôle de votre caméra avec le logiciel de vidéosurveillance inclus.
- résolution maximale 640 x 480 (VGA)

VC 220

La caméra Cisco VC220 est une caméra de surveillance d'intérieur, qui s'adresse aux petites entreprises.

- accès à des vidéos et du son en direct de votre entreprise où que vous soyez, de jour comme de nuit, à l'aide d'un ordinateur ou d'un téléphone portable connecté à Internet
- intégration d'alarmes, de capteurs de portes, de détecteurs de mouvement et d'autres systèmes de gestion dans votre solution vidéo
- réception d'alertes automatiques (notamment des enregistrements vidéo ou des images fixes) en cas de détection de mouvement dans vos locaux

- recherche rapide dans les archives vidéo, configuration et contrôle de votre caméra avec le logiciel de vidéosurveillance inclus
- montage de la caméra au plafond ou sur un mur.
- résolution maximale 640 x 480 (VGA)

Il existe evidemment beaucoup d'autres caméras et constructeurs, mais ceux-ci proposent moins de services que celles décrites ci-dessus.

1.3 Les protocoles de transfert

1.3.1 HTTP

Les requétes HTTP permettent d'intéragir de multiples façon avec la caméra. Celles-ci permettent d'utiliser le mécanisme PTZ de la caméra (Pan Tilt Zoom), de faire une capture d'écran, d'activer la détection de mouvements, etc. En effet, la méthode GET transmet les données via l'url, ce qui permet d'utiliser toutes les fonctionnalités de la caméra. Exemple d'url :

```
http://<nom-serveur>/axis-cgi/jpg/image.cgi?resolution=320*240&camera=1
```

Permet de récuperer un capture au format jpg, de résolution 320*240 pixels.

```
http://<nom-serveur>/axis-cgi/ptz/ptzupdate.cgi?pan=15&tilt=25
```

Fait bouger la caméra de 15 unités vers la droite et de 25 vers le haut.

Si la requette necessite de recevoir des données, celle-ci seront contenue dans la reponse. Par exemple, dans le cas où nous envoyons la requete permettant de récupéré une capture (decrite ci-dessus). L'image sera contenu dans la reponse sous la forme :

```
HTTP/1.0 200 OK\r\n
Content-Type: image/jpeg\r\n
Content-Length: 15656\r\n
\r\n
<JPEG image data>\r\n
```

1.3.2 RTSP

Real Time Streaming Protocol (protocole de streaming en temps-réel) est un protocole de communication destiné aux systémes de streaming. Il permet de contréler un serveur de média à distance, offrant des fonctionnalités typiques d'un lecteur vidéo telles que lecture et pause. Exemple de requéte :

```
PLAY rtsp://myserver/axis-media/media.amp?videocodec=h264&resolution=640
        x480
RTSP/1.0 CSeq: 4
User-Agent: Axis AMC
Session: 12345678
Authorization: Basic cm9vdDpwYXNz
```

Cette requéte permet de mettre en marche la lecture de la caméra, ou de reprendre la lecture en cas de mise en pause.

1.4 Les formats videos

1.4.1 MPEG-4

Définition

Le MPEG4 est une norme de codage vidéo. Celui-ci permet de gérer toutes les nouvelles applications multimédias comme le téléchargement et le streaming sur Internet, le multimédia sur téléphone mobile, etc.

Avantages

- L'avantage principal du MPEG-4 est qu'il permet de s'adapter à beaucoup de supports, tels ceux cités ci-dessus. - Permet également de transmettre le son avec la video.

Inconvénients

- Utilisation impossible avec HTTP

1.4.2 M-JPEG ou Motion JPEG

Définition

le M-JPEG est un codec vidéo qui compresse les images une à une en JPEG.

Avantages

- Permet une utilisation avec HTTP - Compression des images plus rapide que le MPEG-4

Inconvénients

- Non transmission du son

1.5 Les formats audio

1.5.1 G.711

Définition

Le G.711 est une norme de compression audio. * Échantillonnage : 8000 Hz * Bande passante sur le réseau : 64 ou 56 kbit/s * Type de codage : MIC (Modulation d'impulsion codée)

1.5.2 G.726

Définition

Le G.726 est une norme de compression audio. * Bande passante sur le réseau : 16, 24, 32 ou 40 kbit/s * Type de codage : Modulation par impulsions et codage différentiel adaptatif (MICDA)

1.5.3 AAC

Définition

Le AAC (Advance Audio Coding), est un algorithme de compression audio, qui a pour but de réduite la qualité, pour offrir un meilleur débit binaire.

Afin d'avoir une idée de ce qui existait déjà pour la surveillance à distance via la plateforme Android (de préférence pour caméra de type Axis), nous avons réalisé un listing des applications et fonctionnalités proposées en distingant les applications payantes des applications gratuites. Certains concepteurs proposent une version restreinte gratuite et une version payante beaucoup plus sophistiquée. Nous avons remarqué que la plupart avait implémenté les fonctionnalités basiques (comme contrôle de caméra et snapshot) mais très peu proposent la détection de mouvement.

1.5.4 Applications Gratuites

VHS Viewer for Axis

- Snapshot
- Enregistrement vidéo
- Allume la lampe de la camera si disponible

Tiny Cam Monitor

- Visualisation de 4 caméras simultanément
- Partage des caméras via mail
- Import/Export des paramètres
- Snapshot sur carte SD
- Zoom digital
- Existe en version payante

DS Cam

- Stream live
- Snapshot
- Enregistrement vidéo
- Pas de contrôle de caméra

XNET Mobile Viewer

- 16 cameras simultanement
- Multiple camera view
- Snapshot
- Rotation d'image

1.5.5 Applications Payantes

Viewer for Axis Cam

- Rotation "swipe"
- Snapshot
- Sauvegarde sécurisée des login/pass

Ip Cam Viewer

- Widget
- +350 modèles compatibles

Tiny Cam Monitor Pro

Implémentation des fonctions ci-dessous sur la version lite :

- Affichage de 16 caméras simultanément
- Gestion des layouts (déplacement des SurfaceView tactile)
- Recherche de caméras sur un réseau lan
- SSL support
- Détection de mouvement

uCamPro

- Optical Zoom
- Snapshot
- Affichage portrait/paysage

MEyePro

- 4 vues simultanées
- Direct Stream

Mobile Cam Viewer

- 2 caméras
-

Chapitre 2

Analyse des besoins

Notre application a pour but principal d'afficher le flux d'une caméra distante (de marque Axis) et de pouvoir la contrôler directement à partir du périphérique basé sur Android. Il a fallu d'abord définir les besoins essentiels à la réalisation d'une telle application aux niveaux technique et fonctionnel, puis ensuite s'attacher à rendre cette application performante, fiable et ergonomique.

2.1 Besoins fonctionnels

Les besoins suivants sont nécessaires au fonctionnement de l'application, cependant certains sont plus importants que d'autres et donc prioritaires pour arriver à la finalité du projet.

- **Flux vidéo** : l'application doit retransmettre le flux vidéo et audio de la caméra d'une manière fluide et sans interruption.
- **Multivue** : elle doit proposer le choix d'afficher une seule vue en plein écran ou plusieurs vues simultanément sur le même écran.
- **Liste de caméras** : il doit pouvoir gérer une liste de caméras avec différentes actions d'ajout, de modification et de suppression d'entités.
- **Import/Export** : la liste de caméras doit pouvoir être importée ou exportée pour afin de la rendre portable.
- **Contrôle PTZ tactile** : l'utilisateur doit pouvoir contrôler les propriétés Pan/Tilt/Zoom de la caméra tactilement d'une manière fluide.
- **Fonctions avancées** : les fonctionnalités spécifiques à la caméra doivent pouvoir être utilisées (comme l'autofocus ou l'auto-iris par exemple).
- **Capture** : une capture d'écran doit pouvoir être réalisée (avec choix de la résolution) et enregistrée sur le périphérique.
- **Détection de mouvements** : la détection de mouvements doit pouvoir être activée/désactivée et gardée en tâche de fond avec les différentes fenêtres de détection choisies
- **Réglage des propriétés** : l'utilisateur doit pouvoir régler les propriétés générales de l'application (comme le temps de réponse, le taux de rafraîchissement ou encore la sensibilité) et les propriétés spécifiques à la détection de mouvements
- **Notifications** : un système de notifications doit prévenir l'utilisateur du succès de la capture d'écran et de la présence d'un mouvement détecté
- **QrCode** : une caméra doit pouvoir être ajoutée à partir de la lecture d'un QrCode.
- **Réseau** : l'application doit s'adapter au réseau disponible Wifi/3G par le choix de la résolution de l'image reçue.

2.2 Besoins non fonctionnels

L'application ne serait pas vraiment performante si certains aspects non liés au fonctionnement n'étaient pas pris en compte. Nous avons donc tenu compte des besoins suivants pour développer un produit final Android intéressant.

- **Performances** : l'application doit avoir de bonnes performances au niveau transmission et une vitesse de rafraichissement satisfaisante pour permettre la surveillance directe.
- **Sûreté d'exécution** : l'exécution doit être sûre en cas de faible connectivité et ne pas terminer brusquement.
- **Réactivité de contrôle** : la vitesse de réponse doit être la plus faible possible pour proposer un contrôle tactile de la caméra réactif.
- **Fiabilité de sauvegarde** : les données sauvegardées par l'application doivent être intactes et retrouvées sans problèmes.
- **Ergonomie** : l'application doit offrir une bonne ergonomie, une interaction avec l'utilisateur intuitive et être facile d'utilisation pour tous publics.
- **Réactivité de détection** : le temps entre le mouvement détecté et la notification d'alerte résultante doit être le plus court possible.
- **Compatibilité** : l'application doit être compatible avec les différents modèles de caméras Axis et s'adapter aux fonctionnalités variables.

L'ensemble de ces besoins ont été implémenté, à l'eception de la retransmission du son. En effet les codec utilisé par notre caméra (G.711 ,G.726, AAC) ne sont pas implémentés dans les versions d'android que nous disposons. Seul le codec AAC fait son apparition à partir de la version 3.0 d'android.

Chapitre 3

Implémentation

3.1 Projet Android

Un projet Android est découpé selon une arborescence précise. Lors de la création d'un projet, sous Eclipse, il faut choisir le nom du projet. Ce projet appartient à un package qui se présente sous la forme suivante : "extension_domaine"."nom_de_domaine".android."nom_du_projet". Il faut ensuite choisir le nom de l'Activity qui sera lancée au démarrage. Pour finir, il faut choisir le nom de l'application qui sera affiché dans le menu principal. Aprés la création du projet, nous obtenons une architecture prédéfinie, qui contient plusieurs dossiers et fichiers :

- Un dossier *src* qui devra contenir l'ensemble des fichiers sources d'un même package (plusieurs packages peuvent être créés dans le dossier *src*)
- Un fichier *R.java*, généré par l'ADT, qui contient l'ensemble des références aux ressources du projet
- Un dossier *assets*, qui contient l'ensemble des données qui seront chargées sur le mobile (ou tablette) lors de la compilation
- Un dossier *res*, qui contient l'ensemble des ressources relatives aux projet. Ce dossier contient plusieurs sous-dossiers :
  - Le dossier *drawable* contient l'ensemble des images utilisées par l'application
  - Le dossier *layout* contient les fichiers *.xml* qui décrivent les interface de l'application
  - Le dossier *values* contient des fichiers qui décrivent les constantes utilisées par l'application
  - Le dossier *menu* contient les differents menus utilisés
  - Le dossier *xml* contient le ficher de mise en forme du *SharedPreferences* utilisé
  - Les dossiers *values-[CodeLangue]* contiennent les traductions des chaînes utilisées dans le projet.
- Pour conclure, nous trouvons un fichier *AndroidManifest.xml* qui définit le comportement de l'application. Nous y trouvons par exemple le nom, l'icône par défaut, le thème, la version minimale, les activités, les services de l'application, etc...

3.2 L'objet Caméra

3.2.1 Description

L'objet Caméra est implémenté dans la classe *Camera.java*.
Celle-ci regroupe les informations suivantes :

- un descriptif de la caméra. Exemple : "Caméra du Jardin"
- le protocole de communication à utiliser (HTTP ou RTSP).
- l'URL d'accès. Exemple : "http ://192.168.1.20"
- le port (par défaut le port 80 est utilisé).
- le nom d'utilisateur (login).
- le mot de passe.
- le canal (nécessaire lors de l'utilisation de plusieurs caméras pour une même adresse).
- un identifiant unique appelé *uniqueID* défini par la position de la caméra dans la liste sur la page d'accueil.

– le groupe assigné lors de la mise en route de la détection de mouvements (compris entre 0 et 9) appelé *groupeID*.

Afin de pouvoir communiquer une caméra entre plusieurs *activités (activity)* nous avons choisi de rendre cette classe *serializable* pour ne pas devoir passer une à une chacune des caractéristiques.
Cette classe contient également une primitive capable de générer un entier unique pour le couple *uniqueID* et *groupID*, afin de pouvoir lancer plusieurs fenêtres de détection de mouvements pour une même caméra (*getMotionDetectionID*).

3.2.2 Construction de l'objet

La création d'une caméra se fait via l'interface graphique définie par le fichier *add_cam.xml*.

Nous y retrouvons des champs de texte éditables pour chacune des caractéristiques de la caméra ainsi qu'un bouton permettant l'ajout de caméra via *QRCode*.
Un **QRCode** est un code-barres à deux dimensions. Selon *Wikipedia*[1] un QRCode peut contenir plus de 4296 caractères alphanumériques contre 10 à 13 pour un code-barre classique.
L'exemple suivant illustre l'équivalence *caméra-représentation xml-QRCode*

```
<?xml version='1.0' encoding='UTF-8' standalone='yes'?>
    <camList>
        <camera>
            <id>Jardin</id>
            <adresse>http://192.168.1.20:80</adresse>
            <channel>1</channel>
        </camera>
    </camList>
```

Le lecteur de codes QR est implémenté par la bibliothèque *zxing*[2]. Lors d'un clic sur ce bouton, nous appelons l'activité *SCAN* de la bibliothèque zxing avec comme argument *QR_CODE_MODE*. Si la bibliothèque n'est pas disponible sur l'appareil, nous faisons appel à *l'Android market* afin de télecharger

1. http ://fr.wikipedia.org/wiki/Code_QR
2. http ://code.google.com/p/zxing/

l'application *zxing* (qui implémente l'ensemble des fonctions proposées par la bibliothèque).

```
public void onClick(View v) {
    try {
        Intent intent = new Intent(
        "com.google.zxing.client.android.SCAN");
        intent.putExtra("SCAN_MODE", "QR_CODE_MODE");
        startActivityForResult(intent, 0);
    }
    catch (ActivityNotFoundException e) {
        String marketSearch = "market://details?id=com.google.zxing.client.
                android";
        Intent updateIntent = new Intent(Intent.ACTION_VIEW, Uri.parse(
                marketSearch));
        startActivity(updateIntent);
    }
}
```

Listing 3.1 – Lancement de l'activité zxing ou de l'Android market.

Il existe plusieurs générateurs de codes QR disponibles sur internet dont celui implémenté par la bibliothèque *zxing*[3].

Pour ajouter la nouvelle caméra, il ne reste à l'utilisateur qu'à cliquer sur le bouton *Ajouter* pour revenir sur l'écran d'accueil et constater l'ajout de la caméra. Cependant il reste possible de revenir à l'écran d'accueil sans sauvegarder les changements en cliquant sur *Fermer*.

3.2.3 Affichage des caméras

Une fois la phase de création réussie, nous avons implémenté une nouvelle activité permettant d'afficher l'ensemble des caméras ajoutées par l'utilisateur. Cette activité s'appelle *Home* et est contenue dans la classe *Home.java*.

Android définit un cycle de vie pour chaque activité selon le diagramme décrit par la *figure 3.2*.

Ce cycle de vie nous permet d'initialiser, d'interrompre, ou de détruire les différentes vues et fonctionnalités de notre activité. Dans le cas de l'activité *Home* (qui est l'activité d'accueil de notre application), ce cycle se traduit par :

- **onCreate** :
  1. Allocation les ressources à l'aide du constructeur de la classe parente (*super.onCreate(savedInstanceState)* ;)
  2. Récupération des préférences de l'utilisateur via un *SharedPreferences* décrit ultérieurement.
  3. Démarrage du service de détection de mouvements.
  4. Affichage d'un pop-up d'aide et astuces (si l'utilisateur n'a pas choisi de le désactiver).
  5. Lecture de la liste des caméras déjà enregistrées si disponible.
  6. Mise à jour et implémentation des *listeners* de la liste affichant les caméras.

- **onActivityResult** :
  Lorsqu'une activité démarre une autre activité via les *startActivity(intent)* ou *startActivityForResult(intent, 1)* celle-ci passe en arrière-plan pour exécuter l'activité décrite par l'*intent*. Durant cet état elle peut être soit "tuée" par le gestionnaire de tâches, soit en attente d'un résultat. C'est pourquoi l'état *onResume* de la *figure 3.1* englobe également l'état *onActivityResult*.
  Pour faire appel à l'activité permettant d'ajouter une caméra, l'utilisateur doit utiliser le *Menu* (défini dans la classe *Home.java*), puis cliquer sur "Ajouter une caméra". Cette action lancera l'activité *addCam* et attendra le résultat (la nouvelle caméra).

3. http ://zxing.appspot.com/generator/

Quand l'utilisateur aura cliqué sur le bouton "Ajouter" de l'activité *addCam*, celle-ci se terminera en ayant défini comme résultat le code *OK*, et comme valeur *extra*, associée au tag défini par la variable*camTag*, la caméra précédemment *sérialisée*.
Dans la fonction *onActivityResult*, il ne reste plus qu'à désérialiser la caméra, à l'ajouter dans la liste de caméras déjà ajoutées (appelé *camList*), et pour finir, à mettre à jour l'affichage.

– **onDestroy** :
Afin de ne pas devoir entrer les caméras à chaque démarrage de l'application, nous avons choisi de sérialiser puis de sauvegarder la liste des caméras dans un fichier pour les ajouter automatiquement à chaque démarrage. Cette sauvegarde s'effectue juste avant de libérer les ressources en surchargeant la fonction *onDestroy*.

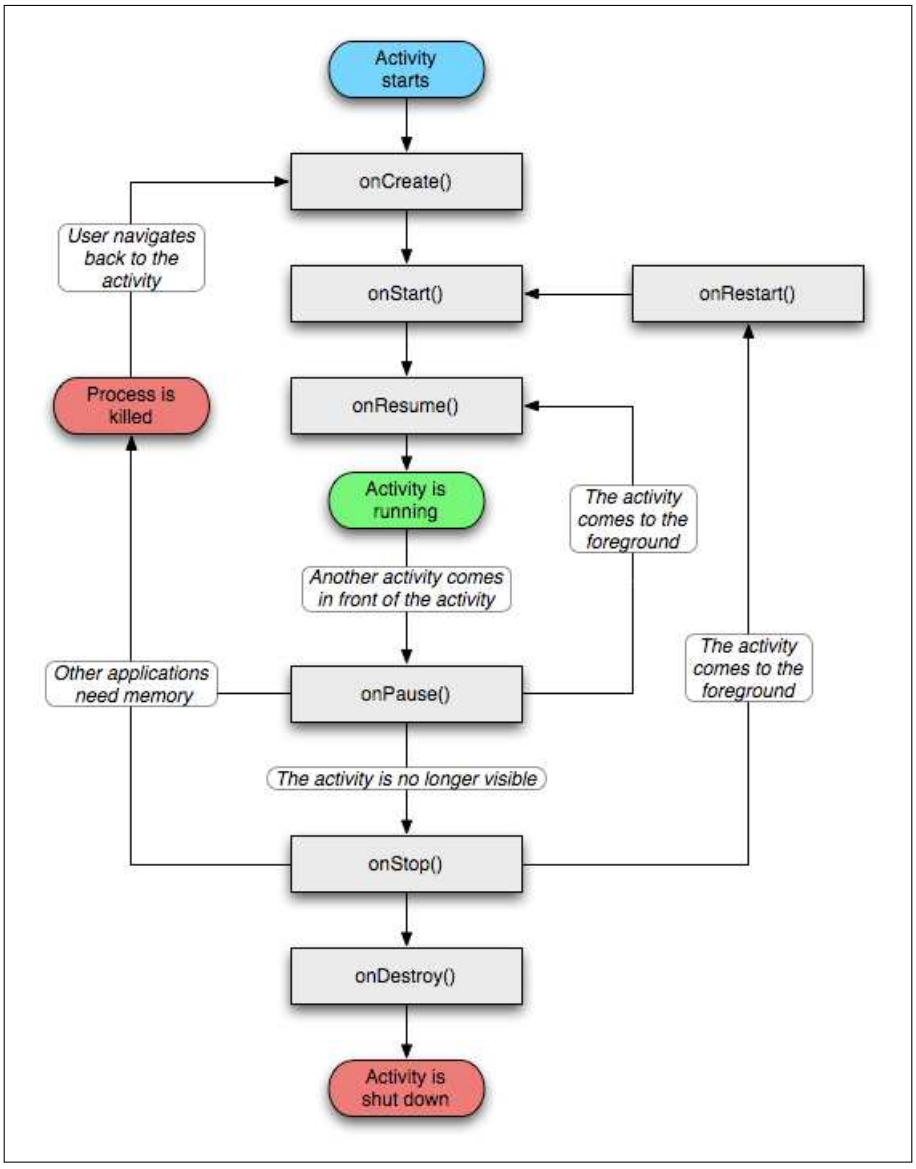

FIGURE 3.1 – Android Activity Life Cycle [4]

4. http ://developer.android.com/reference/android/app/Activity.html

L'affichage de l'activité *Home* est décrit ci-dessous, nous pouvons y retrouver une liste personnalisée contenant :

- L'identifiant unique de la caméra,
- Suivi de son descriptif,
- Puis en indication son URL.

Nous pouvons également voir en bas de l'image le menu qui apparaît lors de l'appui sur la touche menu du téléphone.

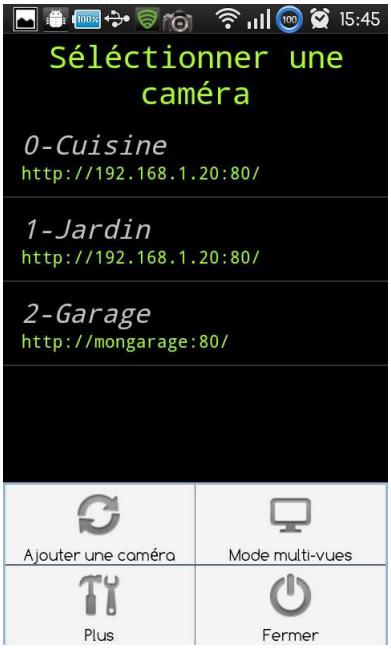

FIGURE 3.2 – Home SnapShot

3.2.4 Modification et Suppression de l'objet

Lors d'un appui long sur un élément de la liste (*setOnItemLongClickListener*), une alerte apparaît pour demander à l'utilisateur s'il souhaite modifier ou supprimer l'élément (la caméra).

- S'il choisit de supprimer l'élément, une seconde alerte apparaît pour confirmer ou annuler son choix.
- S'il choisit de modifier la caméra, l'application lance une nouvelle activité adaptée à la modification de la caméra. Il peut alors appliquer ou annuler ses changements sur le même principe que l'ajout d'une caméra.

```
public boolean onItemLongClick(AdapterView<?> arg0,
View arg1, final int position, long arg3) {
    AlertDialog alert;
    AlertDialog.Builder builder = new AlertDialog.Builder(activity);
    builder.setMessage(getString(R.string.messageChoose))
    .setCancelable(false)
    .setPositiveButton(getString(R.string.boutonModifier),
    new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int id) {
            Intent intent = new Intent(activity.getApplicationContext(),
                EditCam.class);
            Bundle objetbunble = new Bundle();
            objetbunble.putSerializable(getString(R.string.camTag),camList.
                get(position));
            intent.putExtra(getString(R.string.camPosition), position);
            intent.putExtras(objetbunble);
            dialog.cancel();
            startActivityForResult(intent,EDIT_CODE);
        }
    }
    ).setNegativeButton(getString(R.string.boutonSupprimer),
    new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int id) {
            removeCam(position);
            dialog.cancel();
        }
    }
    );
    alert = builder.create();
    alert.show();
    return true;
}
```

Listing 3.2 – Gestion d'un appui long sur un élément de la liste.

3.3 Communication avec la caméra

3.3.1 Envoi de requêtes HTTP

Afin de communiquer avec la caméra, Axis met à disposition des développeurs une API nommée *VAPIX*[5]. Cette interface est basée sur le protocole HTTP, permettant un accès aux fonctionnalités de la caméra par de simples URLs. Par exemple, pour lister les paramètres réseau de la caméra nous utilisons l'URL suivante :

```
http://myserver/axis-cgi/admin/param.cgi?action=list&group=Network
```

En cas de réussite de la requête, nous recevons la réponse suivante :

```
HTTP/1.0 200 OK\r\n
Content-Type: text/plain\n
\n
root.Network.IPAddress=<adresse ip>\n
```

5. http ://www.axis.com/techsup/cam_servers/dev/cam_http_api_index.php

```
root.Network.SubnetMask=<masque reseau>\n
```

Certaines requêtes comme le contrôle PTZ de la caméra ne provoquent pas de réponse (code HTTP 200) mais indiquent simplement que la requête a bien été reçue (code HTTP 204), cependant nous ne sommes pas informés de la réalisation de l'action demandée. Nous avons implémenté le mécanisme de requêtes *HTTP* vers la caméra à l'aide de l'objet *HttpURLConnection* dans la méthode *sendCommand* de la classe *CameraControl*. La commande passée en paramètre est concaténée à l'adresse IP de la caméra pour construire l'URL visée. Nous initialisons alors une connexion HTTP vers l'URL avant d'ajouter un *timeout* au bout duquel la requête est considérée comme un échec, cette valeur étant paramétrable par l'utilisateur dans les préférences de l'application. Le paquet est également forgé avec le champ *Authorization* qui englobe les identifiants encodés en *base64* pour l'authentification :

```
con.setRequestProperty("Authorization",
base64Encoder.userNamePasswordBase64(cam.login, cam.pass));
```

Nous avons pour cela réutilisé la classe existante *base64Encoder*[6] et ajouté une méthode qui renvoie les identifiants en *base64* sous la forme *login :password*. Par conséquent, l'authentification est réalisée à chaque requête, même si toutes les actions ne nécessitent pas de droits spécifiques. La méthode retourne enfin l'objet *HttpURLConnection* qui pourra être utilisé pour analyser la réponse et récupérer le contenu avec *getInputStream* dans le cas d'une capture par exemple.

3.3.2 Chargement de la configuration

Avant de récupérer le flux vidéo distant, nous utilisons la méthode *loadConfig* dans le constructeur de la classe *CameraControl* afin de récupérer et établir la liste des fonctionnalités supportées par la caméra. Cette méthode permet aussi de récupérer certains paramètres comme les résolutions et formats d'image disponibles. Le principe est d'envoyer deux requêtes à la caméra puis de parser le contenu des réponse pour remplir les deux tableaux d'entiers suivants :

- *functionProperties* qui conserve les différentes propriétés des fonctions *PAN*, *TILT*, *ZOOM*, *FOCUS* et *IRIS*.
- *currentConfig* qui conserve l'état actuel des fonctionnalités représenté par les constantes *NOT_SUPPORTED*, *DISABLED* ou *ENABLED*.

La première requête est envoyée à l'adresse suivante :

```
http://myserver/axis-cgi/com/ptz.cgi?info=1
```

Le résultat est un long message texte avec pour chaque ligne une chaîne au format *variable=valeur*. Il suffit alors de parser ligne par ligne en isolant le mot *variable* et en le comparant avec différentes occurences. Par exemple, le code suivant récupère les capacités de la caméra pour la fonctionnalité "pan" :

```
if (property.contains("pan")) {
    if (property.contentEquals("pan"))
    functionProperties[PAN] += ABSOLUTE;
    else if (property.contentEquals("rpan"))
    functionProperties[PAN] += RELATIVE;
    else if (property .contentEquals("continuouspantiltmove")) {
        functionProperties[PAN] += CONTINUOUS;
        functionProperties[TILT] += CONTINUOUS;
    }
}
```

Les constantes identifiant les différentes capacités possibles sont : *ABSOLUTE*, *RELATIVE*, *DIGITAL*, *AUTO*, *CONTINUOUS*. A la fin du parsage de la première requête, *currentConfig* est mis à jour pour les fonctionnalités concernées par *functionProperties* avec la boucle suivante :

```
for (int i = 0; i < NB_BASIC_FUNC; i++)
if (functionProperties[i] > 0)
```

6. http ://www.rgagnon.com/javadetails/java-0084.html

```
currentConfig[i] = ENABLED;
```

La deuxième requête est quant à elle envoyée à l'adresse suivante :

```
http://myserver/axis-cgi/admin/param.cgi?action=list&group=Properties.
    Motion.Motion,Properties.Audio.Audio,Properties.Image
```

Le parsage de la réponse permet de mettre à jour la configuration relative à la détection de mouvements, à l'audio et à l'image (résolutions, rotations, formats vidéo).

Une fois le chargement de la configuration effectué, il sera possible de savoir si telle fonctionnalité est supportée et/ou activée par l'appel aux méthodes respectives *isSupported* et *isEnabled*. De même, *enableFunction* et *disableFunction* permettront de faire varier l'état d'une fonction (*ENABLED*/*DISABLED*). Cependant nous n'avons pas utilisé ces deux dernières méthodes car il n'est pas possible de savoir dans quel état actuel se trouve la caméra pour une certaine fonctionnalité. En effet, plusieurs personnes pouvant agir tour à tour sur les paramètres de la caméra.

3.3.3 Liste des commandes utilisées

Le tableau suivant présente les différentes commandes distantes que nous avons utilisées dans notre application Android, associant l'adresse pointée (basée toujours sur l'adresse IP de la caméra) à l'action effectuée par la caméra.

Partie variable URL	Action
axis-cgi/com/ptz.cgi ?info=1	Retourne la liste globale des capacités fonctionnelles de la caméra
axis-cgi/admin/param.cgi ? action=list&group=Properties...	Retourne l'état actuel des paramètres spécifiés (Motion, Audio, Image, ...)
axis-cgi/mjpg/video.cgi ?resolution=<resolution>	Retourne le flux MJPEG de résolution <resolution>
axis-cgi/jpg/image.cgi ?resolution=<resolution>	Retourne une capture d'écran de résolution <resolution>
axis-cgi/operator/param.cgi ? action=add&group=Motion&template=motion	Ajoute une fenêtre de détection de mouvements
axis-cgi/operator/param.cgi ? action=remove&group=Motion.<groupeID>	Supprime une fenêtre de détection de mouvements existante d'identifiant <groupeID>
axis-cgi/motion/motiondata.cgi ? group=<groupeID>	Retourne à intervalles réguliers les niveaux de détection de la fenêtre d'identifiant <groupeID>
axis-cgi/operator/param.cgi ? action=update&Motion.M<groupeID>.<param>=<valeur>	Met à jour les paramètres de la fenêtre de détection d'identifiant <groupeID>
axis-cgi/operator/param.cgi ? action=update&Motion.M<groupeID>.<param>=<valeur>	Met à jour les paramètres de la fenêtre de détection d'identifiant <groupeID>
axis-cgi/com/ptz.cgi ?<param>=<valeur> – <param>parmi rpan, rtilt, rzoom, riris, rfocus, rbrightness – <valeur>une valeur numérique	Effectue l'action <param>correspondante (pan, tilt, zoom, iris, focus) en appliquant <valeur>de manière relative (calculé à partir de la position actuelle)
axis-cgi/com/ptz.cgi ?<param>=<valeur> – <param>une valeur parmi autofocus, autoiris, ircutfilter, backlight – <valeur>une valeur parmi on, off, auto	Active/désactive ou met en mode automatique la fonctionnalité <param>

Table 3.1 – Liste des commandes HTTP

3.4 Gestion du flux vidéo

3.4.1 Vue Simple

Cette partie consiste a décrire l'implémentation de la retransmission de la vidéo d'une seule caméra sur le téléphone. Cette implémentation se trouve dans la classe *Video.java* qui définit l'activité *Video*.

Cycle de vie

Précédemment, nous avons décrit le cycle de vie de l'activité *Home*. Ici nous avons également ajouté des fonctionnalitées aux différents états afin de ne pas consommer de ressources inutilement. En effet, lorsque l'activité n'est plus au premier plan, il est inutile de continuer à retransmettre le flux vidéo.
C'est également le cas pour l'utilisation du verrou nous permettant d'empêcher le téléphone de se mettre en veille lors de la retransmission de la vidéo. Le code suivant illustre l'arrêt de la vidéo et la libération du verrou (wl) lorsque l'activité n'est pas active.
Il existe plusieurs types de verrous. Pour les utiliser, il faut récupérer une instance du gestionnaire d'énergie *PowerManager* à l'aide de la fonction :

```
PowerManager pm = Context.getSystemService(Context.POWER_SERVICE).
```

Puis il faut demander explicitement la création d'un nouveau verrou (*PowerManager.WakeLock*) à l'aide de la fonction :

```
pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "Tag");
```

Ces différents verrous sont définis par le premier argument. Il existe quatre types de verrous dont les caractéristiques sont définies ci-dessous :

Flag Value	CPU	Screen	Keyboard
PARTIAL_WAKE_LOCK	On*	Off	Off
SCREEN_DIM_WAKE_LOCK	On	Dim	Off
SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off
FULL_WAKE_LOCK	On	Bright	Bright

WakeLock flags table[7]

Nous avons choisi d'utiliser un verrou *SCREEN_DIM_WAKE_LOCK* afin de garder l'écran simplement allumé avec une luminosité au minimum et ne pas consommer excessivement la batterie du téléphone.

7. http ://developer.android.com/reference/android/os/PowerManager.html

```
/**
* Called when Activity start
*/
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.video);
    setRequestedOrientation(0);

    PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE)
       ;
    wl = pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My Tags");
    (...)
}
/**
* Resume video and acquire wakelock when activity resume.
*/
public void onResume() {
    super.onResume();
    wl.acquire();
    if (pause) {
        mv.resumePlayback();
        pause = false;
    }
}
/**
* Stop video and release wakelock when activity sleep
*/
public void onPause() {
    pause = true;
    wl.release();
    if (mv != null)
    mv.stopPlayback();
    super.onPause();
}

/**
* Stop Video before destroy
*/
public void onDestroy() {
    if (mv != null)
    mv.stopPlayback();
    super.onDestroy();
}
```

Listing 3.3 – Video life-cycle

ConnectivityManager

L'accès à internet pour un téléphone disposant d'un système d'exploitation comme Android se fait par des couches physiques de nature différente. Le débit est donc fortement dépendant du support utilisé, c'est pourquoi nous faisons appel à la classe *ConnectivityManager*. Celle-ci permet de récupérer la nature du support et ansi de définir la résolution de la vidéo à demander auprès de la caméra. Nous avons principalement distingué deux types de réseaux : *WiFi* et les autres réseaux mobiles (*3G*, *3G+*, *EDGE*, ...). Nous avons alors défini la correspondance suivante qui nous permet d'obtenir un taux de rafraîchissement correct en *WiFi* (moyenne de 25 FPS pour une couverture totale), la résolution minimale

étant utilisée pour les autres réseaux :

Réseau	Résolution
WiFi	320x240
Autres	168x120

Pour récupérer le type de réseau, nous faisons une nouvelle fois appel à la fonction :

```
ConnectivityManager mConnectivity = (ConnectivityManager) getSystemService(
    Context.CONNECTIVITY_SERVICE);
```

Puis en récupérant une instance du réseau actif :

```
NetworkInfo info = mConnectivity.getActiveNetworkInfo();
if (info != null && info.isConnected()) {
    int netType = info.getType();
    if (netType == ConnectivityManager.TYPE_WIFI) {
        /*... Code pour un support de type WiFi ...*/
    }
    else {
        /*... Code pour les autres supports ...*/
    }
```

Format Vidéo

Android est capable d'encoder et de décoder un grand nombre de format audio et vidéo (cf Android Supported Media Formats [8]) à travers deux protocoles (*HTTP ou RTSP*).
La caméra mise a notre disposition propose uniquement deux formats vidéo : *MJPEG* via le protocole *HTTP* et *MPEG4* via le protocole *RTSP*.
Cependant malgré que le système d'exploitation est capable d'utiliser le protocole *RTSP*, celui-ci ne nous permet pas de modifier les requêtes afin d'y ajouter l'authentification nécessaire pour la lecture de la video. Celle-ci est donc accessible uniquement si la caméra autorise les utilisateurs anonyme à la visionner.
Nous sommes donc limités au protocole *HTTP* dans lequel nous avons expliqué précédemment la manière dont nous nous authentifions.

Le format *MJPEG* transporté via le protocole *HTTP* n'étant pas géré par le *mediaPlayer* [9], nous avons dû trouver un autre moyen d'afficher la video.

MjpegView

Grâce à la documentation de la caméra [10] on peut constater que la vidéo est en réalité composée de la succession d'une multitude d'images. Pour afficher la vidéo il faut donc simplement faire défiler chacune des images récupérées dans la réponse de la requête permettant d'obtenir le flux de vidéo *MJPEG*.
Nous avons trouvé sur internet une composant graphique *OpenSource* appelé *MjpegView* [11] réalisant exactement ce traitement. Nous avons tout de même dû effectuer quelques modifications pour l'adapter à notre utilisation (comme l'ajout d'une fonction *resumePlayback* pour redémarrer la lecture de la vidéo par exemple).
Comme chacun des composants graphique, il peut être ajouter dans une activité ou défini directement dans une vue au format *xml*.

8. http ://developer.android.com/guide/appendix/media-formats.html
9. http ://developer.android.com/reference/android/media/MediaPlayer.html
10. http ://www.axis.com/techsup/cam_servers/dev/cam_http_api_2.php
11. http ://www.anddev.org/multimedia-problems-f28/mjpeg-on-android-anyone-t1871-30.html

```
<de.mjpegsample.MjpegView.MjpegView
    android:id="@+id/surfaceView1"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" />
```

Listing 3.4 – video.xml

```
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    MjpegView mv = new MjpegView(this);
    setContentView(mv);
    mv.setSource(URL);
    mv.setDisplayMode(MjpegView.SIZE_BEST_FIT);
    mv.showFps(true);
    (...)
}
```

Listing 3.5 – mjpegViewer.java

Interface de contrôle de la caméra

Nous avons choisi de mettre à disposition les commandes permettant le contrôle de la caméra dans cette activité. Ainsi l'utilisateur peut observer instantanément les modifications.

Le *LayoutInflater* est une service du système d'exploitation qui permet de charger une vue a partir d'un fichier *.xml*. Nous allons utiliser ce service pour ajouter, ou supprimer un ensemble de composant graphique à la vue retransmettant la vidéo.

```
RelativeLayout screen = (RelativeLayout) findViewById(R.id.RelativeLayout01
    );
LayoutInflater inflater = (LayoutInflater) getSystemService(Context.
    LAYOUT_INFLATER_SERVICE);
/* Add view from mds_video.xml */
inflater.inflate(R.layout.mds_video, screen, true);
/* Remove all component contained in the mainLayout */
screen.removeView(findViewById(R.id.mainLayout));
```

Listing 3.6 – LayoutInflater utilisation

Voici les différentes vue produites :

FIGURE 3.3 – Minimal view for control video.

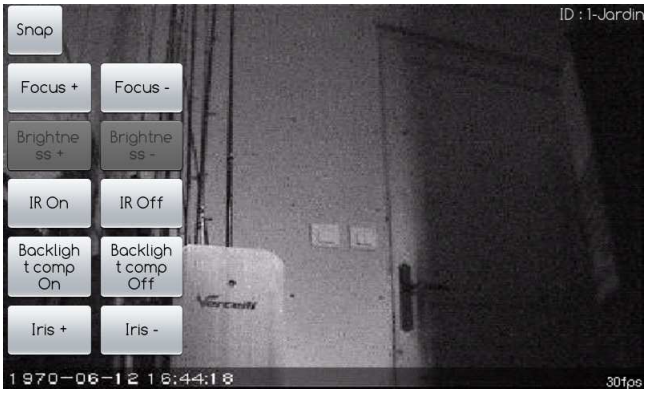

FIGURE 3.4 – Adding components contained in the file *adv_video.xml*.

FIGURE 3.5 – Adding components contained in the file *mds_video.xml*

Toutes ces vues peuvent être ajouter ou supprimer en utilisant le menu personnalisé dans l'activité *Video*.

3.4.2 Multi-Vue

Nous avons souhaité implémenter la possibilité de visionner plusieurs caméras simultanément sur une même vue. De nouvelles contraintes sont donc apparuent :

- Comment télécharger et afficher plusieurs vidéos *mjpeg* provenant de différentes sources ?
- Pouvoir choisir le nombre de caméras à afficher.
- Comment choisir les caméras ?
- Comment contrôler les caméras ?

Playerthread video

Le thread lancé au démarrage d'une activité est appelé *UI Thread*[12] ou *User Interface thread*. Le rôle de se thread est d'exécuter le cycle de vie de l'activité. C'est également lui qui gère les interactions utilisateurs et l'affichage. Seul l'UI thread peut modifier l'affichage ou capturer une interaction.

Pour afficher simultanément plusieurs vidéos, nous devons donc utiliser plusieurs threads afin d'obtenir un affichage fluide des vidéos.
Chaque thread aura pour tache de télécharger l'image, puis de signaler à l'UI qu'une nouvelle image est disponible afin d'actualiser l'affichage.

12. http ://davy-leggieri.developpez.com/tutoriels/android/ui-thread/

Il existe plusieurs mécanisme pour effectuer cette notification.

– La première méthode est d'enfiler un objet de type Runnable, dans la file d'exécution de l'UI à l'aide de la fonction :

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        /* Performed by the UI thread */
        imageView[index].invalide();
    }
}
);
```

Listing 3.7 – Example of use runOnUiThread

– Une autre méthode consiste à notifier l'UI à l'aide de *Message* grâce à la mise en place d'un *handler* qui exécute une action pour chaque message reçu.

```
/* Performed by the UI thread */
public static Handler myViewUpdateHandler = new Handler() {
    public void handleMessage(Message msg) {
        if (msg.what == GUIUPDATEIDENTIFIER) {
            /* Uptdate View message */
            imageView[index].invalide();
        }
        if (msg.what == URLERRORIDENTIFIER) {
            /* Other message type */
        }
        super.handleMessage(msg);
    }
}
;
```

Listing 3.8 – Example of message handler

Le diagramme suivant décrit l'exécution de l'activité *MultiVideo*. A noter la présence d'un délai entre chaque téléchargement d'une nouvelle image. Ce délai nous permet de ralentir le nombre d'image par seconde dans le but de réduire considérablement la consommation de la bande passante. L'utilisateur à la possibilité de régler lui même ce délai dans le menu des paramètres.

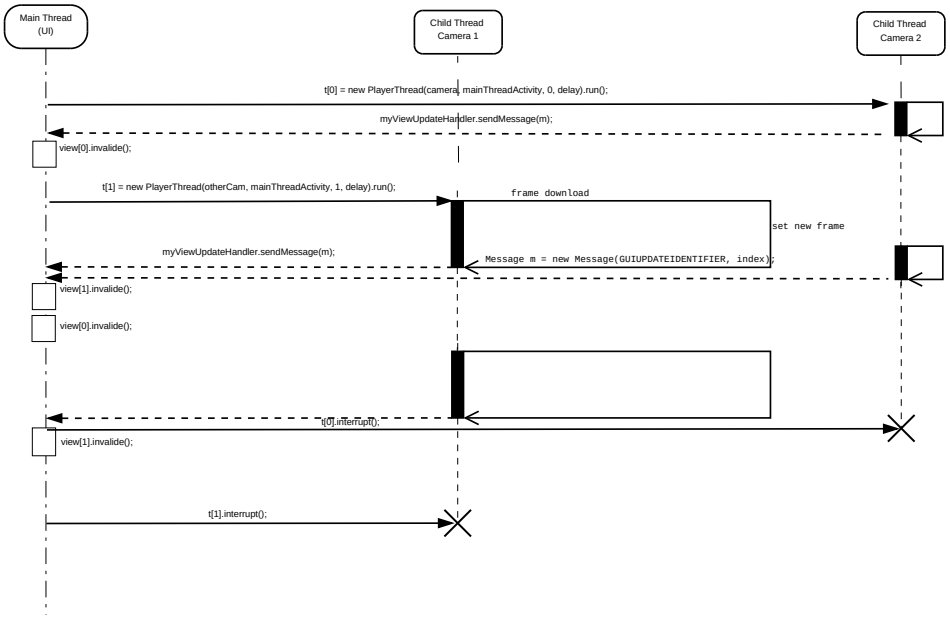

FIGURE 3.6 – Multi-View sequence diagram

Layout personnalisés

Maintenant que vous sommes capable d'afficher plusieurs vidéos simultanément, nous nous sommes intéressés à la disposition de celles-ci sur l'écran. Pour sa faire nous offrons dans un premier temps à l'utilisateur la possibilité de choisir le nombre de caméra à afficher (entre 2 et 6).
Ceci à l'aide d'un alert dialog qui affichera sous forme d'une liste, le nombre de caméra à afficher. Puis lors de la capture d'un clic sur l'un des éléments de la liste, nous lancerons l'activité *MultiVideo* en y ajoutant un paramètre supplémentaire appelé *nbViewTag* qui défini le nombre de vue ainsi récupéré.

```
AlertDialog.Builder builder = new AlertDialog.Builder(activity);
builder.setTitle(getString(R.string.cameraAlertTitle));
builder.setSingleChoiceItems(nb_view, -1, new DialogInterface.
    OnClickListener(){
    public void onClick(DialogInterface dialog, int item) {
        dialog.dismiss();
        Intent intent = new Intent(activity, MultiVideo.class);
        Bundle objetbunble = new Bundle();
        objetbunble.putSerializable(getString(R.string.camListTag), camList)
    ;
        intent.putExtras(objetbunble);
        intent.putExtra(getString(R.string.nbViewTag),(item + 2));
        startActivity(intent);
    }
}
);
AlertDialog alert = builder.create();
alert.show();
```

Listing 3.9 – Multi-Video launcher

Enfin dès le lancement de l'activité *MultiVideo*, il ne nous reste plus qu'à récupérer les paramètres pour définir le layout à utiliser.

Nous allons maintenant commenter un problème technique rencontrer lors de la mise en place des layouts. Nous avons expliqué précédemment la manière dont nous récupérions le nombre de caméra à

afficher. Afin d'afficher les images récupérées par les différents threads, nous devons dans un premier temps récépérer chacune des adresses des différentes *ImageView* défini dans les cinq layout.
Pour se faire, toutes les *ImageView* auront un identifiant défini par leurs positions sur l'écran :

- image0 pour le 1er emplacement
- image1 pour le second
- image5 pour le 6eme emplacement (le plus grand disponible)

En nommant toutes les *ImageView* des différents layout avec un identifiant équivalent, nous pouvons déduire l'ensemble des adresses en récupérant celle de l'image0. En effet lors de la compilation, le compilateur génère un fichier appelé *R.java*. Dans ce fichier on trouve l'ensemble des adresses pour chaque classe, layout, et objet de l'application trié par ordre alphabétique. Cette dernière propriété nous garantie que l'adresse de l'image1 vaut l'adresse de l'image0 plus un, etc .... Nous sommes donc en mesure de récupérer chacune des *ImageView* afin de les mettre a jours, lors de la réception de nouvelle images mais aussi de définir les *listener* pour chaque images permettant de capturer les interactions.

```
/*
* get R.id.image0 address and inc it to find R.id.image1,
* R.id.image2, ... , R.id.image.n
*/
int dep = R.id.image0;
for (int i = 0; i < nbView; i++) {
    img[i] = (ImageView) findViewById(dep + i);
    /* Set Image and Listener for each view */
    camView[i] = null;
    img[i].setImageResource(R.drawable.cadre);
    img[i].setOnClickListener(new myOnClickListener(i));
    img[i].setOnLongClickListener(new myOnLongClickListener(i));
}
```

Listing 3.10 – ImageView address resolver

FIGURE 3.7 – Multi-View 2 camera

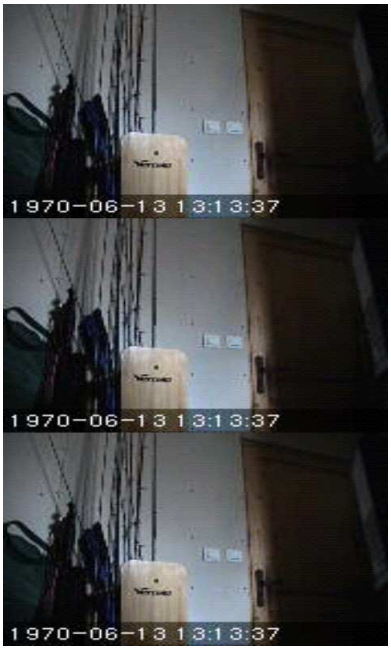

FIGURE 3.8 – Multi-View sequence 3 camera

FIGURE 3.9 – Multi-View 5 camera

Utilisation

L'utilisateur a également la possibilité de choisir la caméra à afficher sur chacune des *ImageView*.

- Le premier clique sur l'une d'elle affiche une liste permettant de choisir la caméra à afficher.
- Un nouveau clique stop la vidéo.
- Un long clique lance l'activité *Video* pour permettre à l'utilisateur de contrôler la caméra. Une fois les réglages effectuer, il peut revenir sur la vue précèdente en appuyant sur le bouton retour de son téléphone.

Le diagramme d'etat suivant illustre les differents changements d'etat des videos lorsque l'utilisateur interagi avec l'application. On y retrouve principalement l'arret de toutes les vidéos lors du passage en simple vue, ainsi que le redemarrage des vidéos lors du retour de celle-ci.

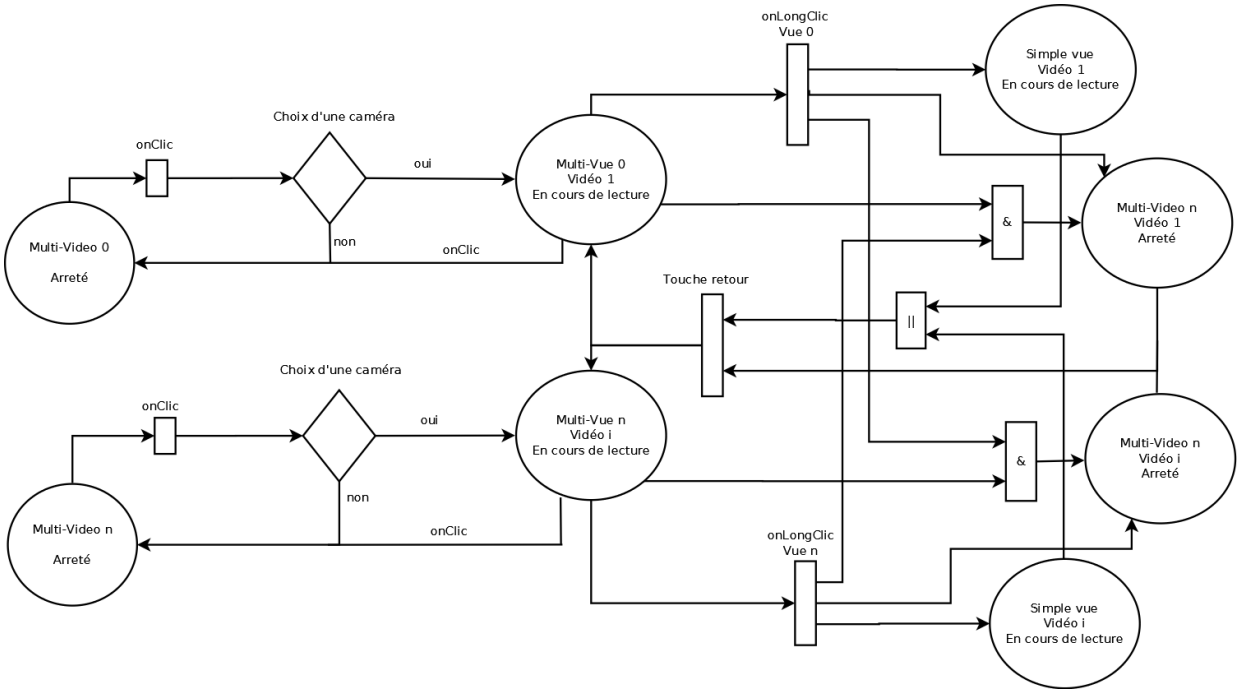

FIGURE 3.10 – Diagramme d'etat pour une 2-vue.

3.5 Contrôle de la caméra

3.5.1 CameraControl

La communication pour le contrôle du Pan/Tilt/Zoom et des fonctionnalités spécifiques (snapshot, contrôles avancés) de la caméra s'effectuent à travers la classe *CameraControl*. Les fonctions *changeValFunc* et *switchAutoFunc* permettent d'envoyer les requêtes HTTP à la caméra pour changer la valeur des différents paramètres de celle-ci : - *PAN*, *TILT*, *FOCUS*, *IRIS*, *BRIGHTNESS* identifient des paramètres prenant une valeur flottante - *AUTOFOCUS*, *AUTOIRIS*, *AUTO_IR*, *BACKLIGHT* identifient des paramètres dont la valeur est comprise dans *on*, *off*, *auto* Une réponse HTTP de code *HttpURLConnection.HTTP_NO_CONTENT* (code 204) indique la réussite de la requête, cependant nous ne savons pas si l'action a bien été effectuée.

La fonction *takeSnapshot* réalise la requête de capture d'écran avec comme résolution la valeur passée en paramètre. Elle retourne les données renvoyées par la caméra sous forme d'un objet *Bitmap*.

Cette classe s'occupe également des activation/désactivation de la détection :

- *addMotionD* ajoute une fenêtre de détection.
- *removeMotionD* supprime une fenêtre de détection existante.
- *updateMotionDParam* met à jour certains paramètres de la fenêtre de détection comme les coordonnées ou la sensibilité de détection.

Lors du chargement du menu et du layout des contrôles avancés de la caméra, nous vérifions si les fonctionnalités correspondantes sont supportées et/ou activées à l'aide des fonctions *isSupported* et *isEnabled*. Un appui sur les boutons du menu (celui-ci apparaissant lors de l'appui sur la touche "Menu") déclenchera une alerte texte si la fonctionnalité n'est pas disponible, alors que pour le layout des contrôles avancés nous définissons la disponibilité de la fonctionnalité directement par l'état du bouton (activé/désactivé).

3.5.2 TouchListener

Le contrôle du PTZ tactile a été implémenté à l'aide de la classe *TouchListener*. Nous avons défini deux gestes possibles avec un ou plusieurs pointeurs (doigt, stylet, ...) pour notre application :

- un déplacement avec un seul pointeur pour faire bouger la caméra (Pan/Tilt)

– un écartement/rapprochement de deux pointeurs pour zoomer/dézoomer

Cette classe implémente l'interface *OnTouchListener* fournissant une fonction *onTouch* dans laquelle nous gérons chaque mouvement du ou des pointeurs à partir d'un objet *MotionEvent*. Un geste sous Android est défini par une suite de mouvements de type *MotionEvent*. Il existe une dizaine de types d'actions possibles pour *MotionEvent*, cependant nous utilisons seulement les 4 suivants pour notre application :

– *ACTION_DOWN* : le premier pointeur a été ajouté (le geste est commencé) et l'événement contient les coordonnées du pointeur
– *ACTION_POINTER_DOWN* : un nouveau pointeur a été ajouté
– *ACTION_POINTER_UP* : un pointeur a été enlevé
– *ACTION_UP* : le dernier pointeur a été enlevé (le geste est terminé) et l'événement contient les coordonnées du dernier pointeur

A partir de ces types d'actions, nous pouvons définir le déplacement de la caméra par la suite *ACTION_DOWN - ACTION_UP* et la gestion du zoom par la suite :
ACTION_DOWN - ACTION_POINTER_DOWN - ACTION_DOWN_UP - ACTION_UP (nous considérons cependant que le zoom n'est réalisé qu'à partir de l'événement *ACTION_POINTER_DOWN*).

Le champ *mode* contient l'état actuel du geste, soit une des constantes *NONE*, *DRAG* ou *ZOOM*. Nous récupérons également les dimensions de la vue par *getWidth* et *getHeight* pour la mise à l'échelle des distances de déplacements. Pour représenter les coordonnées des pointeurs, nous avons utilisé le type *PointF*.

Nous avons implémenté dans cette classe des méthodes de calcul géométrique entre deux objets *PointF* :

– *calculateMoveX* pour la différence en abscisse.
– *calculateMoveY* pour la différence en ordonnée.
– *calculateDistance* pour la distance géométrique.

Nous avons également développé des méthodes pour mettre à l'échelle le déplacement des pointeurs sur l'écran par rapport au déplacement réel réalisé par la caméra :

– *scaleMoveX* pour le déplacement en abscisse, en utilisant la largeur de l'écran *getHeight*.
– *scaleMoveY* pour l'échelle en ordonnée, en utilisant la hauteur de l'écran *getWidth*.
– *scaleZoom* pour le zoom/dézoom, en utilisant *zoomStep*.

Afin de rendre moins "agressif" le déplacement sur le téléphone, nous avons mis en place un paramètre de sensibilité, celui-ci étant modifiable dans les préférences. Le thread est également endormi quelques millisecondes afin d'éviter la surcharge d'événements à traiter. Le tableau ci-dessous montre le traitement réalisé par la méthode *onTouch* dont le code est présenté après.

Type ACTION	Geste du déplacement	Geste du zoom
DOWN	Mise en mémoire des coordonnées du point de départ *mode = DRAG*	
POINTER_DOWN		Calcul et mise en mémoire de la distance de départ *mode = ZOOM*
POINTER_UP		Calcul et mise en mémoire de la distance d'arrivée
UP	Calcul des déplacements horizontaux et verticaux Mise à l'échelle des déplacements *mode = NONE*	Calcul du ratio entre les deux distances Mise à l'échelle du ratio (avec le champ *zoomStep*) *mode = NONE*

TABLE 3.2 – Algorithme de traitement onTouch

```
public boolean onTouch(View v, MotionEvent event) {
    width = v.getWidth();
```

```
    height = v.getHeight();
    switch (event.getAction() & MotionEvent.ACTION_MASK) {
        case MotionEvent.ACTION_DOWN:
        current.set(event.getX(), event.getY());
        mode = DRAG;
        break;

        case MotionEvent.ACTION_POINTER_DOWN:
        currentDist = calculateDistance(
        new PointF(event.getX(0), event.getY(0)),
        new PointF(event.getX(1), event.getY(1)));
        mode = ZOOM;
        break;

        case MotionEvent.ACTION_POINTER_UP:
        startDist = currentDist;
        currentDist = calculateDistance(
        new PointF(event.getX(0), event.getY(0)),
        new PointF(event.getX(1), event.getY(1)));
        break;

        case MotionEvent.ACTION_UP:
        if (mode == DRAG) {
            PointF start = new PointF(current.x, current.y);
            current.set(event.getX(), event.getY());
            float moveX = scaleMoveX(calculateMoveX(start, current));
            float moveY = scaleMoveY(calculateMoveY(start, current));
            camC.changeValFunc(CameraControl.PAN, -1* moveX / sens, -1 *
         moveY
            / sens);
        }
        else if (mode == ZOOM) {
            Log.i(TAG, "startDist=" + startDist);
            Log.i(TAG, "currentDist=" + currentDist);
            if (Math.abs(startDist - currentDist) > 10) {
                float ratio = (currentDist / startDist > 1) ? currentDist
                / startDist : -1 * (startDist / currentDist);
                camC.changeValFunc(CameraControl.ZOOM, scaleZoom(ratio), 0);

            }
        }
        mode = NONE;
        try {
            /* Bloc UI thread to not spam request */
            Thread.sleep(200);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        break;
    }
```

3.6 Fonctionnalités avancées et réglages

Les caméras *Axis* offrent un grand nombre de services pour lesquels nous avons implémenter la possibilité de les utiliser directement avec le téléphone.

3.6.1 Snapshot

Parmi ces services, la camera propose à l'utilisateur la possibilité de prendre une photo instantanée. Pour ce faire il nous suffit d'envoyer une requête *HTTP* à l'adresse suivante :

```
http://<servername>/axis-cgi/jpg/image.cgi
```

Dans cette requête, nous avons la possibilité d'ajouter de nombreux paramètres comme la résolution, le taux de compression, l'affichage de la date, l'heure, etc. Toutes ces options sont décrites dans la documentation de la caméra [13]. L'image est contenu dans la réponse :

```
HTTP/1.0 200 OK\r\n
Content-Type: image/jpeg\r\n
Content-Length: <image size>\r\n
\r\n
<JPEG image data>\r\n
```

Il ne nous reste qu'a convertir le flux en image à l'aide de la fonction :

```
Bitmap bmp = BitmapFactory.decodeStream(stream);
```

puis l'enregistrer dans un fichier :

```
bmp.compress(Bitmap.CompressFormat.JPEG, 80, fichier);
```

Afin de prévenir l'utilisateur de la réception de la capture, nous avons implémenté une fonction permettant d'ajouter une notification personnalisée dans la barre de notification du téléphone.
En plus de notifier l'utilisateur, nous lui offrons la possibilité de visionner directement la photo (via le visionneur d'images par defaut) en cliquant sur la notification.

```
public static void statusBarNotificationImage(Activity activity, Bitmap bmp
    , String text, String path, int id, String tag) {
    NotificationManager notificationManager;
    notificationManager = (NotificationManager) activity.getSystemService(
      Context.NOTIFICATION_SERVICE);
    Notification notification = new Notification(R.drawable.camera, "Camera-
      Axis", System.currentTimeMillis());
    notification.contentView = new RemoteViews(activity.getPackageName(), R.
      layout.notification);
    /* Notification action (Open gallery) */
    Intent intentNotification = new Intent();
    intentNotification.setAction(android.content.Intent.ACTION_VIEW);
    intentNotification.setDataAndType(Uri.fromFile(new File(path)), "image/
      png");
    PendingIntent pendingIntent = PendingIntent.getActivity(activity.
      getApplicationContext(), 0, intentNotification, 0);
    notification.defaults |= Notification.DEFAULT_VIBRATE;
    notification.contentIntent = pendingIntent;
    notification.contentView.setImageViewBitmap(R.id.Nimage, bmp);
    notification.contentView.setTextViewText(R.id.Ntext, text);
    notificationManager.notify(tag, id, notification);
}
```

13. http ://www.axis.com/techsup/cam_servers/dev/cam_http_api_2.php#api_blocks_image_video_jpeg_snapshot

Listing 3.11 – ImageView address resolver

Ainsi nous obtenons la notification suivante :

FIGURE 3.11 – Notification SnapShot

3.6.2 Autres réglages

Afin d'améliorer la visibilité de la caméra, celle ci propose un grand nombre de fonction comme le réglage de la luminosité, de la mise au point (focus), l'ouverture ou la fermeture de l'iris, et beaucoup d'autres fonctions également disponible sur la documentation [14] de la caméra.

Nous avons choisis d'implémenter certaines d'entre elles dont nous fournissons l'accès via l'interface de contrôle avancé disponible dans le menu de l'activité *Video*.

FIGURE 3.12 – Advance Controls

- Focus + / Focus - : Augmenter ou diminuer la mise au point.
- Brightness + / Brightness - : Augmenter ou diminuer la luminosité
- Iris + / Iris - : Augmenter ou diminuer l'exposition.

14. http ://www.axis.com/techsup/cam_servers/dev/cam_http_api_2.php#api_blocks_ptz_control

– IR On / IR Off : Activer ou desactiver le filtre anti-IR
– Backlight comp On / Backlight comp Off : Activer ou desactiver la correction pour la surexposition à la lumière.

Mais nous avons également offert la possibilité à l'utilisateur d'utiliser les réglages automatique de la caméra toujours à partir du menu de l'activité *Video*.

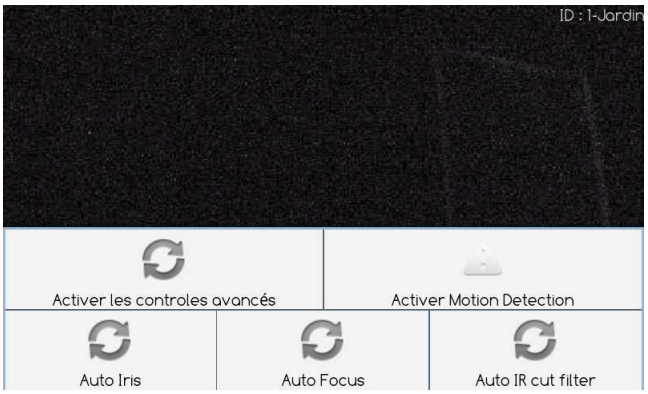

FIGURE 3.13 – Advance Controls

3.7 Détection de mouvements

3.7.1 Présentation

La détection de mouvement est une mécanisme utilisé pour détecter les changements de position, et les mouvements d'objets ou d'individus dans une zone couverte par un dispositif de video-surveillance. Selon une traduction de la définition de wikipedia [15] :

> " Ceci peut être réalisé soit par des dispositifs mécaniques qui interagissent physiquement avec la zone ou par des dispositifs électroniques qui permettent de quantifier et mesurer les changements dans l'environnement donné. Le mouvement peut être détecté par : son (capteurs acoustiques), l'opacité (capteurs optiques et infrarouges et des processeurs d'images vidéo), le géomagnétisme (capteurs magnétiques, magnétomètres), la réflexion de l'énergie transmise (radar laser infrarouges, capteurs ultrasoniques, capteurs et radars micro-ondes), induction électromagnétique (détecteurs à boucle inductive), et les vibrations (triboélectrique, sismiques, et des capteurs d'inertie). "

La caméra mise à notre disposition contient un service capable de détecter les mouvements couvert par l'objectif ou limité à une zone particulière.
Nous avons souhaiter offrir à l'utilisateur la possibilité de détecter les mouvements et d'en être averti en temps réel sur son téléphone. C'est pourquoi nous avons implémenté un service à l'écoute de chacune des caméras afin de détecter les mouvements sans que l'application soit active.

3.7.2 Utilisation de service Android

Sous Android, les services sont des composants qui permettent l'exécution de fonctionnalités en arrière plan pendant une durée indéterminée même lorsque les activités de l'application ne sont pas lancée. Pour créer un service il faut dans un premier temps le déclarer dans le fichier *AndroidManifest.xml* :

```
<application android:icon="@drawable/camera"
    android:label="@string/app_name">
    <activity <!-- activities list --> />
        <service android:name="MotionDetectionService" />
```

15. http ://en.wikipedia.org/wiki/Motion_detection

```
    </application>
```

Puis en implémentant le service dans le fichier *MotionDetectionService.java*. Tout comme les activités, les services ont un cycle de vie :

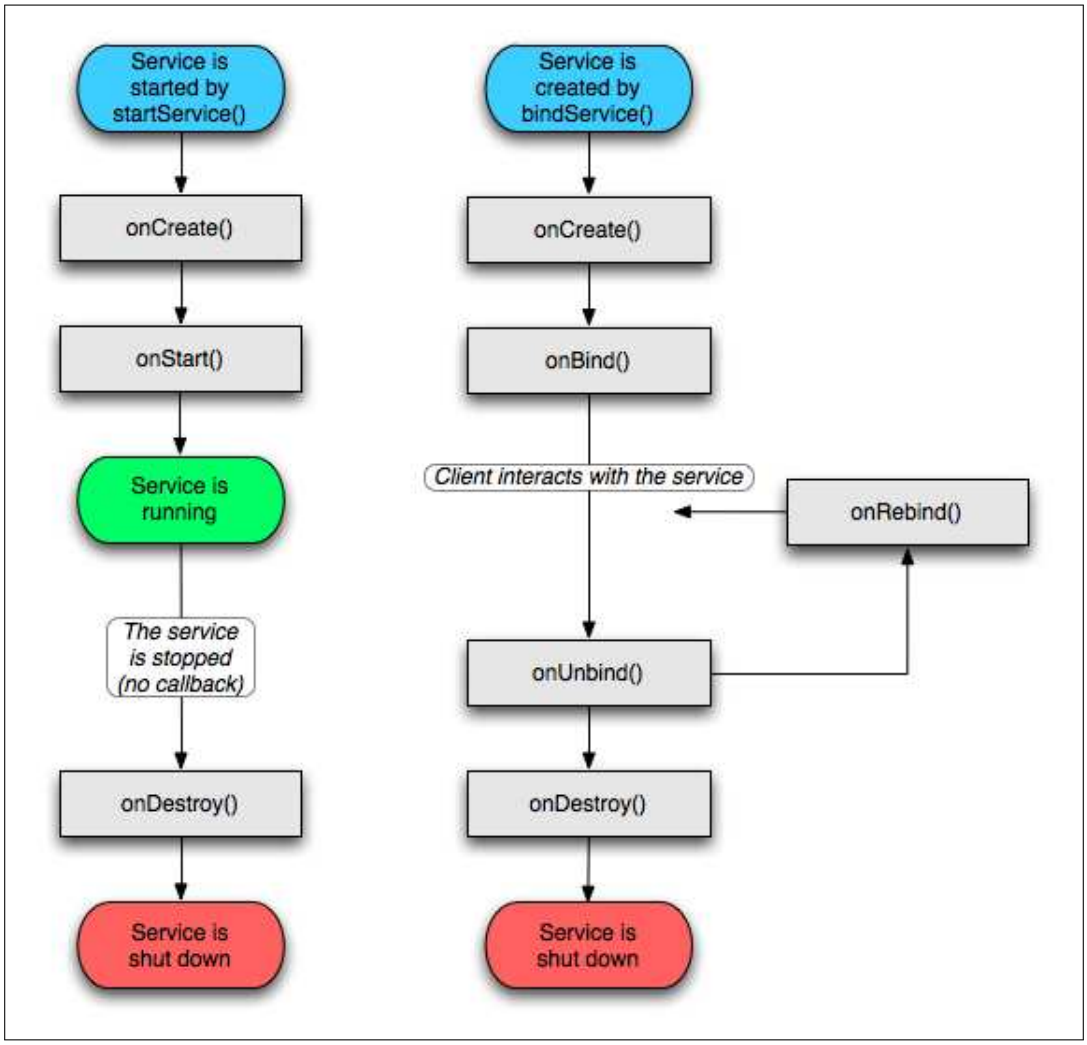

FIGURE 3.14 – Service Life Cycle [16]

Ce schéma illustre la possibilité de contacter un service de deux manières différentes. La première (cycle de gauche) se limite à envoyer des instructions au service pour y déclencher un traitement défini dans la fonction :

```
@Override
public int onStartCommand(Intent intent, int flags, int startId){
    /* work to do */
}
```

La seconde (cycle de droite) permet à une activité de se lier à un service afin de pouvoir utiliser toutes les ses méthodes public.
N'ayant pas besoin de communiquer autrement avec les services que pour leurs demander d'effectuer un traitement, nous avons choisi d'implémenter le 1er cycle.

16. http ://www.linuxtopia.org/online_books/android/devguide/guide/topics/fundamentals.html

3.7.3 Activation du service

Les caméras Axis peuvent gérer jusqu'à dix fenêtres de detection de mouvement par caméra simultanément. Pour ajouter une fenêtre (avec les valeurs par défaut) il suffit d'envoyer une requête *HTTP GET* a l'adresse :

```
http://myserver/axis-cgi/operator/param.cgi?action=add&group=Motion&
    template=motion
```

Pour ajouter une fenêtre défini par sa position sur l'écran, il faut spécifier explicitement la position de la fenêtre :

```
http://myserver/axis-cgi/operator/param.cgi?action=add&group=Motion&
    template=motion
&Motion.M.Top=500
&Motion.M.Bottom=7000
&Motion.M.Left=5000
&Motion.M.Right=8500
```

Chacune de ces requêtes nous fournira le numéro du groupe (de 0 a 9) associer a la requête, nous permettant plus tard de recuperer le niveau de la detection. La réponse est de la forme :

```
HTTP/1.0 200 OK\r\n
Content-Type: text/plain\r\n
\r\n
M<group number> OK\r\n
```

Nous avons de ce fait differencié les détections de mouvement a l'aide d'un entier unique defini tel que :

```
public int getMotionDetectionID(int startID) {
    return groupeID + (uniqueID * 10) + startID;
}
```

Chaque caméra dispose d'un numéro unique (sa position dans la liste). Si elles sont toutes capable de créer dix fenetres de detection, en multipliant leur numéro par dix et en additionnant le numero de groupe on peut garantir l'unicité de l'identifiant. Nous ajoutons également un entier *startID* afin de ne pas commencer à compter à partir de zero mais à partir de la valeur contenu dans *startID*.

Pour sélectionner la fenêtre à détecter, nous avons implémenté un nouveau composant graphique appelé *drawRectOnTouchView* dont la fonction est de dessiner un rectangle et de récupérer la position de la fenêtre à partir d'un clique de l'utilisateur.

drawRectOnTouchView

Comme pour le lecteur *mjpeg*, l'implémentation d'un nouveau composant graphique se fait en étendant la classe *View*. Dans une premier temps il faut définir les constructeurs (Création de la surface sur laquelle dessiner, définition des couleurs, ...). Puis il est nécessaire de surcharger la méthode :

```
@Override
protected void onDraw(Canvas canvas)
```

afin de dessiner le composant. Dans notre utilisation, nous souhaitons dessiner dynamiquement un rectangle en fonction des interactions fournis par l'utilisateur. Android nous propose de surcharger une fonction qui est appeler lors de chaque interactions avec le composant appelé :

```
@Override
public boolean onTouchEvent(MotionEvent event)
```

Grâce a cette fonction, nous allons dessiner un rectangle qui suivra le doigt de l'utilisateur jusqu'à ce qu'il relâche la pression sur l'écran. Précédemment nous avons présenté les actions définies par les

MotionEvent. Nous allons à nouveau utiliser ces interactions pour définir le comportement de notre composants :

```
@Override
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
        start.set(x, y);
        isDraw = false;
        break;
        case MotionEvent.ACTION_MOVE:
        end.set(x, y);
        isDraw = false;
        invalidate();
        break;
        case MotionEvent.ACTION_UP:
        end.set(x, y);
        isDraw = true;
        invalidate();
        break;
    }
    return true;
}

@Override
protected void onDraw(Canvas canvas) {
    canvas.drawColor(Color.TRANSPARENT);
    canvas.drawLine(start.x, end.y, end.x, end.y, mPaint);
    canvas.drawLine(end.x, start.y, end.x, end.y, mPaint);
    canvas.drawLine(start.x, start.y, start.x, end.y, mPaint);
    canvas.drawLine(start.x, start.y, end.x, start.y, mPaint);
}
```

Listing 3.12 – Draw Rectangle Component

Lors de l'appuie sur l'ecran, nous sauvegardons les coordonnées du point, puis dès le déplacement du doigt de l'utilisateur, on commence à tracer le rectangle. Enfin lorsque l'utilisateur relève son doigt, on sauvegarde la position du point, puis on signal que le dessin est dessiné à l'aide du flag *isDraw*. La fonction *onDraw* nous permet de dessiner les arêtes du rectangle à partir des coordonnées sauvegardées.

Enfin en affectant la couleur "transparente" au fond du composant, on peut le superposer au dessus d'une vue *mjpeg* pour dessiner un rectangle sur notre video.

Une fois la détection activée auprès de la caméra, nous pouvons démarrer notre service en lui indiquant le groupe fournit par la caméra.

3.7.4 Détection et Signalisation

Dans cette partie nous allons étudier le comportement du service. La detection de mouvement auprès des caméra *Axis* se fait au travers de la requête GET à l'adresse :

```
http://myserver/axis-cgi/motion/motiondata.cgi?group=<group number>
```

Dans le résultat de cette requête, nous nous intéressons à la valeur du niveau de detection. Si celui-ci dépasse le seuil défini dans les paramètres utilisateur, le service déclenche une notification sous forme de vibration du téléphone ainsi que la prise d'une photo instantannée. La photo sera alors sauvegardé dans le dossier *"/sdcard/com.myapps.camera/"* sous la forme : "MD-time-¡heure en ns¿.jpeg".

Afin de ne pas “spammer” l’utilisateur de notification, nous avons décider d’établir un délai minimum entre deux notifications réglable dans les paramètres.

```
HTTP/1.0 200 OK\r\n
Content-Type: multipart/x-mixed-replace; boundary=axismdb\r\n\r\n
--axismdb\r\n
Content-Type:text/plain\r\n\r\n
group=0; level=28; threshold=45; \r\n
group=1; level=43; threshold=25; \r\n
--axismdb\r\n
Content-Type:text/plain\r\n\r\n
group=0; level=54; threshold=45; \r\n
group=1; level=38; threshold=25; \r\n
--axismdb\r\n
Content-Type:text/plain\r\n\r\n
group=0; level=49; threshold=45; \r\n
group=1; level=19; threshold=25; \r\n
--axismdb\r\n
.
.
.
```

5.4.5 Get the Motion Detection level[17].

3.7.5 Implémentation et lancement de la tache

Le traitement à effectuer pour chacune des détections lancées sera de “parser” le contenu de la réponse citée ci-dessus, puis avertir l’utilisateur en cas de mouvement. Grâce a la fonction *onStartCommand* défini ci-dessus nous somme en mesure de lancer des threads dont le rôle sera justement d’effectuer ce traitement (un thread par caméra).
Lorsqu’un mouvement sera détecté par l’un des threads, nous utiliserons le même principe de handler que le *PlayerThread* pour prévenir le service activant ainsi une notification.

17. http ://www.axis.com/techsup/cam_servers/dev/cam_http_api_2.php#api_blocks_motion_get_md_level

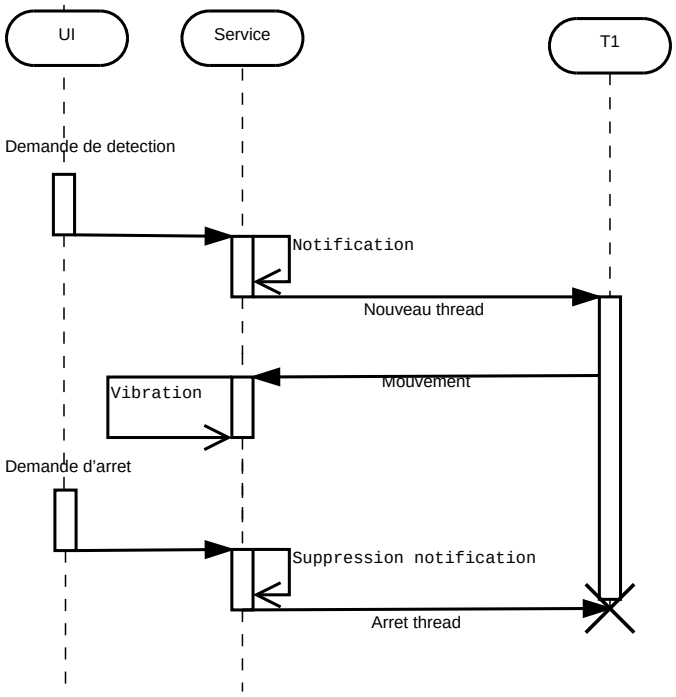

FIGURE 3.15 – Service Sequence Diagramm

De plus chaque demande de détection de mouvement auprès d'une caméra créera une notification "en cours" dans la barre de notification du téléphone. Ainsi l'utilisateur pourra accéder facilement et rapidement à la caméra dont il souhaite surveiller les mouvements.

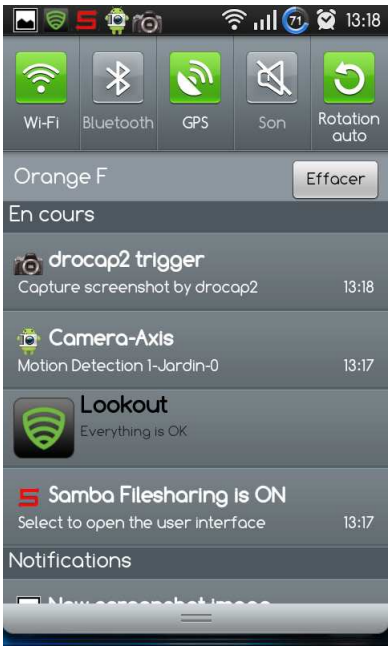

FIGURE 3.16 – Running Notification

3.8 Gestion des préférences

3.8.1 Import / Export

Cette partie s'intéresse à la sauvegarde de la liste des caméras. En effet si l'utilisateur décide de desinstaller l'application, ou si il change de téléphone, nous avons souhaité qu'il ai la possibilité de partager la liste de ses caméras. Pour ce faire nous avons implémenté deux fonctions pour importer et exporter cette liste.
La facilitée de modification, et la portabilité du langage *xml* nous a conduit à formaliser les éléments de cette liste sous forme de différentes balises imbriquées :

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
    <camList>
        <camera>
            <id>home</id>
            <adresse>http://192.168.1.20/</adresse>
            <channel>1</channel>
        </camera>
        <camera>
            <id>home ext</id>
            <adresse>http://82.---.---.---/</adresse>
            <channel>1</channel>
        </camera>
    </camList>
```

Pour des raisons de sécurités évidente, nous n'avons pas souhaité sauvegarder les noms d'utilisateurs et mots de passe, c'est pourquoi lorsque l'utilisateur importe ses caméras, il devra tout de même les modifier une à une afin de remplir les champs en question.

L'écriture et la lecture de se fichier se fait au travers de deux fonctions :

```
public static void xmlWrite(ArrayList<Camera> camList, String url)
public static ArrayList<Camera> xmlRead(String url)
```

La première crée et rempli le fichier en utilisant la classe *XmlSerializer* à partir des caméras fourni en paramètres. La seconde lit et alloue une liste de caméras grâce à la classe *DocumentBuilderFactory* qui nous permet de "parser" le fichier afin de récupérer le contenu de chacune des balises à l'aide de la fonction :

```
doc.getElementsByTagName("camera");
```

Nous avons choisis d'exporter les caméras dans une zone mémoire accessible aux utilisateurs ne disposant pas de téléphone "rooté". C'est pourquoi le dossier de destination par défaut est "/sdcard/-com.myapps.camera/". Mais lors de l'exportation l'utilisateur peut tout de même modifier ce chemin en le modifiant manuelement lors de l'apparition de l'*AlertDialog*.

FIGURE 3.17 – Export Dialog

3.8.2 Shared Preferences

Tout au long de se projet nous avons implémenté un grand nombre de fonction disposant de paramètres défini par l'utilisateur. La classe *SharedPreferences* permet de sauvegarder des couples "donnée/clée" afin de faire perdurer ces données entre les sessions.
Il est nécessaire d'établir plusieurs étapes pour utiliser cette classe :

1. Implémentation de la vue pour accéder à nos paramètres :

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/
    android"
    xmlns:seekbarattr="http://schemas.android.com/apk/res/com.myapps">

    <PreferenceCategory android:title="Delai">
        <EditTextPreference
            android:key="limitFPS"
            android:title="Delai FPS"
            android:numeric="decimal"
            android:summary="Une description" />
        <EditTextPreference android:key="@string/NotifTO"
            android:numeric="decimal"
            android:title="Delai de Notification Motion Detection"
            android:summary="Une description" />
    </PreferenceCategory>
    <PreferenceCategory android:title="Autre">
        <CheckBoxPreference
            android:key="@string/isWelcome"
            android:title="Desactiver l'astuce"
            android:summary="Une description" />
    </PreferenceCategory>
</PreferenceScreen>
```

2. Il faut ensuite créer une classe pour utiliser la vue précedement défini et initialiser les différents composants :

```
public class MesPreferences extends PreferenceActivity{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.preferences);
        preferences = PreferenceManager.getDefaultSharedPreferences(this
      );

        /* Show current delay */
        preferenceManagerUtils.setEditTextPreferenceValueFromPreference(
      this,
        preferences, getString(R.string.TimeOut),
        getString(R.string.defaultTimeOut));
        /*  .... */
    }
}
```

3. Il ne reste plus qu'à utiliser le *SharedPreferences* en récupérant une instance de celui-ci dans l'activité principale :

```
preferences = PreferenceManager.getDefaultSharedPreferences(this);
/* Print tricky */
if (preferences.getBoolean(getString(R.string.isWelcome), true) ==
    false) {
    Dialog_welcome myDialog = new Dialog_welcome(this,R.style.
    theme_dialog);
    myDialog.show();
}
```

Nous sommes donc en mesure de régler un nombre illimité de paramètres comme :
- Le Delai d'attente entre chaque téléchargement d'image pour le *PlayerThread*.
- Le temps de non réponse maximum pour un requête *HTTP*.
- L'interval entre deux notification lors de la detection de mouvement.
- L'activation ou non du pop-up d'astuces au démarrage.
- La sensibilité du PTZ réglable a l'aide d'une "seekBar" disponible en Open Source a l'adresse suivant :
  http ://www.codeproject.com/KB/android/seekbar_preference.aspx
- Le seuil de détection de mouvement.

3.9 Multilangues, Interface et Partage

Multilangues

Android offre la possibilité à une application d'adapter son langage en fonction de la langue du téléphone. Nos chaînes de caractères étant déjà regroupées dans le fichier *res/values/strings.xml*, nous avons décider de traduire notre application en quatre langues : Allemand, Anglais, Espagnol, Français. Pour ce faire il suffit de créer 4 fichiers :
- *res/values-de/strings.xml*
- *res/values-en/strings.xml*
- *res/values-es/strings.xml*
- *res/values-fr/strings.xml*

contenant respectivement la traduction des chaînes de caractères en fonction des langues citées ci-dessus.

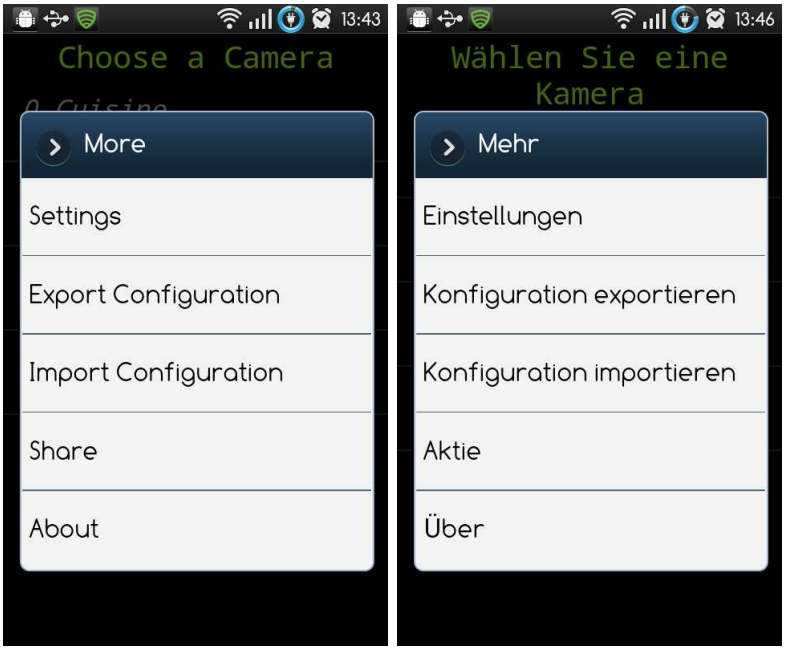

FIGURE 3.18 – English - Deutsch Translation

Interface

Nous avons défini les couleurs, la police, et la taille des différents textes à l'aide de feuille de styles (équivalent *css* pour Android) dans le fichier : *res/values/attrs*. L'idée est de surcharger le style par défaut de chacun des composants graphique afin d'y apporter quelques modifications (comme la couleur par exemple) créant ainsi notre propre thème indépendant des thèmes implémentés par les constructeurs (*HTC Sens*, *Samsung TouchWiz*, *Motorola Motoblur*, ...).

```
<style name="CustomTheme" parent="android:Theme.NoTitleBar">
    <item name="android:colorBackground">
        @color/black
    </item>
    <item name="android:buttonStyle">
        @style/StandardButton
    </item>
    <item name="android:textViewStyle">
        @style/StandardTextView
    </item>
    <item name="android:editTextStyle">
        @style/StandardEditText
    </item>
    <item name="android:listViewStyle">
        @style/StandardListView
    </item>
</style>

<style name="StandardTextView" parent="@android:style/Widget.TextView">
    <item name="android:textColorHint">
        @color/greenyellow
    </item>
    <item name="android:textColor">
        @color/silver
    </item>
```

```
    <item name="android:typeface">
        monospace
    </item>
</style>
```

Partage

Enfin pour finir sur le chapitre concernant l'implémentation, nous avons fourni à l'utilisateur une fonction permettant le partage de l'application (respectivement d'une caméra) via un message (resp. un fichier join) pouvant être diffusé par de nombreuses applications (mail, sms, "facebook", ...). Pour ce faire il faut dans un premier temps, définir l'action de l'intent (dans notre cas nous souhaitons partager des données, nous utilisons donc le flag *ACTION_SEND*). Puis il faut spécifier le type des données à partager (ici il s'agit simplement d'une chaîne de caractères). Enfin nous pouvons démarrer l'activité choisi lors de l'affichage de l'*AlertDialog*.

```
final Intent messIntent = new Intent(Intent.ACTION_SEND);
messIntent.setType("text/plain");
messIntent.putExtra(Intent.EXTRA_TEXT, getString(R.string.messageShare));
startActivity(Intent.createChooser(messIntent, getString(R.string.
        shareTitle)));
```

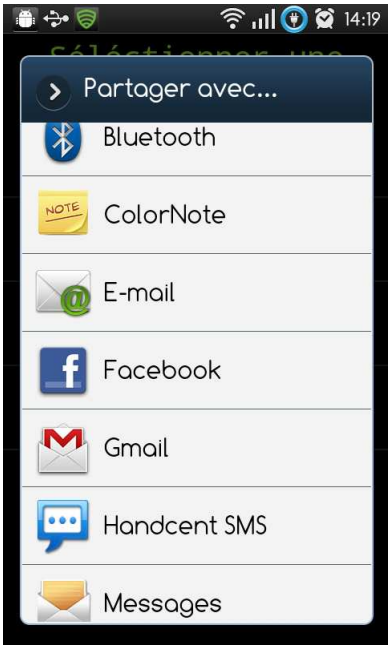

FIGURE 3.19 – Share with...

Chapitre 4

Tests

4.1 Android JUnit Test

Android propose un framework permettant d'effectuer nos propre tests unitaire. Pour ce faire nous avons créé un nouveau projet de type *Android Test Project*. Toute classe de test de ce projet devra heriter de la classe *TestCase* (ou d'une des ses sous-classes) pour permettre l'initialisation du téléphone. Puis nous avons surchargé la methode *setUp* afin de definir l'etat du téléphone à l'instant où le test est lancé.

```
@Override
protected void setUp() throws Exception {
    super.setUp();
    mActivity = this.getActivity();
    src = new Camera("test", "root", "root", "192.168.1.20", 80, "http", 1);
}
```

Une fois ces étapes réalisées, nous pouvons implémenter l'ensemble de nos tests unitaire dans des fonctions distinctes. Nous avons choisis de tester septs fonction de notre projet :

1. Le premier test a pour but de tester l'ajout de plusieurs caméras, puis dans le cas où cette étape reussie, d'exporter les caméras, de vider la liste et de les réimporter afin d'observer d'éventuel dysfonctionnements.

```
public void testimport() {
    /* Ajout 3 cameras */
    camList = new ArrayList<Camera>();
    int position = camList.size();
    src.setUniqueID(position);
    camList.add(position, src);
    assertEquals(camList.size(), 1);

    position = camList.size();
    src.setUniqueID(position);
    camList.add(position, src);
    assertEquals(camList.size(), 2);

    position = camList.size();
    src.setUniqueID(position);
    camList.add(position, src);
    assertEquals(camList.size(), 3);

    /* Export */
    boolean res = xmlIO.xmlWrite(camList, exportPath, exportName);
    assertEquals(res, true);
```

```
    /* Purge la liste */
    camList.clear();
    assertEquals(camList.size(), 0);

    /* Import */
    camList = xmlIO.xmlRead(exportPath + exportName);
    assertEquals(camList.size(), 3);
}
```

Listing 4.1 – Test 1 : Import/Export

2. Le second test porte sur l'ajout d'une notification dans la barre de notification, puis de sa suppression. Ce test require la presence de l'utilisateur pour observer les resultats, en effet, durant le test precedent la presence d'*assert* nous garanti le resultat. Ici c'est l'observation de l'apparition d'une notification qui nous permet de constater la reussite du test.

```
public void testNotification() {
    /* ajout notif */
    int index = 5;
    Resources res = mActivity.getResources();
    Drawable drawable = res
    .getDrawable(com.myapps.R.drawable.hello_android);
    Bitmap bmp = ((BitmapDrawable) drawable).getBitmap();
    notificationLauncher.statusBarNotificationImage(mActivity, bmp, "
      test",
    "blabla", index, "tag");
    NotificationManager notificationManager;
    notificationManager = (NotificationManager) mActivity
    .getSystemService(Context.NOTIFICATION_SERVICE);
}
```

Listing 4.2 – Test 2 : Notification

3. Le troisieme test ressemble au second à l'exception que celui-ci test les notification dite *En Cours*.

```
public void testRunningNotification() {
    int index = 10;
    Intent notificationIntent = new Intent(
    mActivity.getApplicationContext(), Video.class);

    Bundle objetbunble = new Bundle();
    objetbunble.putSerializable(
    mActivity.getString(com.myapps.R.string.camTag), src);
    notificationIntent.putExtras(objetbunble);
    notificationIntent.setAction(Intent.ACTION_MAIN);
    /* Ajout de data inutile pour l'application mais necessaire pour
      que la fonction filterEquals() retourne faut pour deux intents
      avec des extras differents */
    notificationIntent.setDataAndType(Uri.parse("value"), "Camera");
    PendingIntent contentIntent = PendingIntent.getActivity(
    mActivity.getApplicationContext(), 0, notificationIntent, 0);
    notificationLauncher
    .statusBarNotificationRunning(mActivity.getApplication(),
    contentIntent, index, "ca tourne !");

    notificationLauncher.removeStatusBarNotificationRunning(
    mActivity.getApplication(), index);
```

```
}
```

Listing 4.3 – Test 3 : Notification En-Cours

Nous allons maintenant passer aux fonctions de contrôle ainsi qu'aux services fourni par la caméra :

5. Nous allons commencer par un simple test nous permettant de verifier l'etablissement d'une connection lors de l'appel à la fonction *sendCommand*. L'assert nous permet de garantir l'etablissement de la connection quelque soit le resultat de la requette.

```
public void testSendCommand() {
    CameraControl c = new CameraControl(src, mActivity);
    HttpURLConnection res = null;
    try {
        res = c.sendCommand("axis-cgi/com/ptz.cgi?info=1&camera=1");

    }
    catch (IOException e) {
        e.printStackTrace();
    }
    assertNotNull(res);
}
```

Listing 4.4 – Test 4 : Connection HTTP

6. La garantie du test précédent nous permet d'effectuer des tests de plus en plus precis. Ce 5eme test a pour but de tester la prise de photo instantannée ainsi que la sauvegarde de celle-ci sur la carte mémoire. Afin de constater que la caméra nous a bien envoyer la bonne photo, l'utilisateur peut la visionner dans le dossier *"/sdcard/com.myapps.camera/"*.

```
public void testSnap() {
    CameraControl camC = new CameraControl(src, mActivity);
    String[] resolutions = camC.getResolutions();
    String fileNameURL = "/sdcard/com.myapps.camera/";
    String fileName = "Snap-" + System.currentTimeMillis() + ".jpeg";
    Bitmap bmp;
    try {
        bmp = camC.takeSnapshot(resolutions[0]);
        assertNotNull(bmp);
        snapShotManager.saveSnap(bmp, fileNameURL, fileName);
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Listing 4.5 – Test 5 : SnapShot et sauvegarde

7. La détection de mouvement fait necessite l'interaction entre une activité et un service lors de son lancement, c'est pourquoi nous avons testé séparement l'ajout et la suppression d'une fenetre de la detection en elle meme comme le montre les deux tests suivant. Le premier de ces deux test verifie que la caméra retourne bien le numéro de groupe attribué à la fenetre. Si celui-ci est correct, l'assert ne se declanche pas, on peut alors liberer la fenetre de la caméra.

```
public void testAddRemoveMotionDetection() {
    CameraControl c = new CameraControl(src, mActivity);
    assertNotNull(c);
    int groupe = -1;
    try {
```

```
        groupe = c.addMotionD();
        assertNotSame(groupe, -1);

        groupe = c.removeMotionD();
        assertEquals(groupe, -1);
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    catch (CouldNotCreateGroupException e) {
        e.printStackTrace();
    }
}
```

Listing 4.6 – Test 6 : Ajout et suppression d'une fenetre

8. Ce dernier test est probablement le plus important de cette partie. Si il termine, il permet de constater le déclanchement de la notification lors de la capture d'un mouvement (snapshot plus vibration du téléphone). L'idée est de mettre en route la detection de mouvement sur la caméra *src* (fenetre par defaut). Une fois celle-ci lancée, nous faisons boucler le programme tant que la caméra ne detecte pas de mouvements. Si un mouvement est détecté, le processus sort de la boucle, puis stoppe la détection nous garantissant la présence d'un mouvement.

```
public void testMD() {
    try {
        CameraControl c = new CameraControl(src, mActivity);
        int group = c.addMotionD();
        c.cam.setGroup(group);

        Intent intent = new Intent(mActivity, MotionDetectionService.
            class);
        Bundle objetbunble = new Bundle();
        objetbunble.putSerializable(mActivity.getString(R.string.camTag)
            ,
        c.cam);
        intent.putExtras(objetbunble);
        int lim = 20;
        long delay = 1000;
        intent.putExtra("limit", lim);
        intent.putExtra("delay", delay);
        mActivity.startService(intent);
        while(MotionDetectionService.detected == false){
            Thread.sleep(1000);
            /* passer devant la cam pour continuer */
        }

        int indice;
        if ((indice = MotionDetectionService.isAlreadyRunning(c.cam)) !=
            -1) {
            MotionDetectionService.stopRunningDetection(c.cam,
            mActivity.getApplication(), indice);

            c.removeMotionD();
        }

    }
    catch (IOException e) {
```

```
            e.printStackTrace();
        }
        catch (CouldNotCreateGroupException e) {
            e.printStackTrace();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
```

Listing 4.7 – Test 7 :Détection des mouvements

4.2 Analyse des connections via WireShark

Grace au périphérique virtuel fournit dans le sdk d'Android, nous sommes en mesure d'analyser le trafic entre la caméra et le périphérique (téléphone ou tablette). Pour ce faire nous avons utilisé le logiciel *Wireshark*[1].
Grace au trafic enregistré par Wireshark nous avons pu constater que :

- L'ensemble des connections ouverte sont correctement fermées grace l'utilisation de la fonction *disconnect()* definie par la classe *HttpURLConnection*. L'exemple ci-dessus represente le demarrage d'une simple vue pour la caméra d'adresse *192.168.1.20*. Pour effectuer ce test nous avons ralenti l'execution de l'application afin d'éviter l'entrelassement des paquets. On retrouve les deux connections permettant la récupération des informations relative à la caméra (connections 51576 et 51577), ainsi que la demande du flux vidéo (connection 51578). La présence du second paquet fin ainsi que son acquitement n'apparaissent pas sur la capture dut à la latence engendré par la fermeture et la liberation des sockets/ressources.

Source	Destination	Protocol	Info
192.168.1.13	192.168.1.20	TCP	51576 > http [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
192.168.1.20	192.168.1.13	TCP	http > 51576 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1
192.168.1.13	192.168.1.20	TCP	51576 > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
192.168.1.13	192.168.1.20	HTTP	GET /axis-cgi/com/ptz.cgi?info=1&camera=1 HTTP/1.1
192.168.1.20	192.168.1.13	TCP	http > 51576 [ACK] Seq=1 Ack=201 Win=6432 Len=0
192.168.1.20	192.168.1.13	HTTP	HTTP/1.0 200 OK (text/plain)
192.168.1.20	192.168.1.13	TCP	http > 51576 [FIN, ACK] Seq=1086 Ack=201 Win=6432 Len=0
192.168.1.13	192.168.1.20	TCP	51576 > http [ACK] Seq=201 Ack=1087 Win=63155 Len=0
192.168.1.13	192.168.1.20	TCP	51577 > http [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
192.168.1.20	192.168.1.13	TCP	http > 51577 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1
192.168.1.13	192.168.1.20	TCP	51577 > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
192.168.1.13	192.168.1.20	HTTP	GET /axis-cgi/admin/param.cgi?action=list&group=Properties.Motion.Motion,
192.168.1.20	192.168.1.13	TCP	http > 51577 [ACK] Seq=1 Ack=272 Win=6432 Len=0
192.168.1.20	192.168.1.13	TCP	[TCP segment of a reassembled PDU]
192.168.1.20	192.168.1.13	HTTP	HTTP/1.1 200 OK (text/plain)
192.168.1.13	192.168.1.20	TCP	51577 > http [ACK] Seq=272 Ack=329 Win=63913 Len=0
192.168.1.13	192.168.1.20	TCP	51577 > http [FIN, ACK] Seq=272 Ack=329 Win=63913 Len=0
192.168.1.13	192.168.1.20	TCP	51578 > http [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
192.168.1.20	192.168.1.13	TCP	http > 51577 [ACK] Seq=329 Ack=273 Win=6432 Len=0
192.168.1.20	192.168.1.13	TCP	http > 51578 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460 SACK_PERM=1
192.168.1.13	192.168.1.20	TCP	51578 > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
192.168.1.13	192.168.1.20	HTTP	GET /axis-cgi/mjpg/video.cgi?resolution=160x120 HTTP/1.1
192.168.1.20	192.168.1.13	TCP	http > 51578 [ACK] Seq=1 Ack=207 Win=6432 Len=0
192.168.1.20	192.168.1.13	TCP	[TCP segment of a reassembled PDU]
192.168.1.20	192.168.1.13	TCP	[TCP segment of a reassembled PDU]

FIGURE 4.1 – Wireshark capture

- Notre PlayerThread reduit considerablement la consomation de la bande passante au detriment des fps. Exemple : la comparaison de deux capture sur un même interval de temps nous a permi d'effectuer une comparaison du nombre de paquet echangé ainsi que leurs taille.
- En supplément du test numéro 7, nous observons que le service mi en place pour la détéction de mouvement continue à echanger des informations avec la caméra malgres la fermeture de l'application.

1. http ://www.wireshark.org/

4.3 Manipulation avec la caméra

La derniere phase de test consiste à verifier la totalité des services fourni par l'application en observant les effets sur la caméra.
Nous avons grace à cela verifier que :

- La totalité des controles (PTZ) implémentés correspondent aux mouvements de la caméra.
- Les fonctions de controle avancé sur la vidéo s'activent correctement.
- La détéction de mouvements se declanche correctement (en adéquation avec les test precedents)
- La vidéo observée sur le téléphone correspond à la zone pointée par la caméra (en fonction de la vue et des parametres utilisées ainsi que de la couverture réseaux).

Chapitre 5

Optimisations / Extensions futures

Malgrès que le projet semble fonctionnel, il reste tout de même de nombreuses améliorations et optimisations à apporter.

Enregistrement vidéo et retransmission du signal audio

Actuelement notre projet ne permet pas l'enregistrement de vidéo dut principalement à l'utilisation du format *MJPEG*. En effet, le seul moyen pour nous d'enregistrer la vidéo serai la sauvegarde successive des images recu. Un solution plus performente serait de surcharger un partie ou l'ensemble des classes permettant l'utilisation du protocole *rtsp* afin d'y integrer un mécanisme d'authentification pour permettre l'utilisation de la vidéo *mpeg4*. Ainsi nous pourrions remplacer le player *mjpegView* par le media player proposé par android, afin de bénéficier de ses nombreux avantages (lecture, pause, enregistrement grace au *mediaRecorder*, retransmission du son grace au format *mpeg4* ...).

Édition des fenêtres de détection de mouvement

Nous offrons a l'utilisateur la possibilité de détécter les mouvement sur une fenetre au prealablement dessinée sur la vidéo. Cependant actuelement, le seul moyen d'editer cette fenetre est d'arreter la détéction puis de relancer celle-ci avec la nouvelle fenetre. En effet, meme si l'utilisateur modifie la fenetre durant la detection, nous n'avons pas eu le temps d'implémenter la mise à jour de sa position aupres de la caméra.
Pour ce faire, l'idée serait d'implementer une extension de notre classe *drawRectOnTouchView* plus adapté à notre utilisation. Par exemple, la surcharge de la fonction *onTouchEvent()* nous permetterait d'ajouter une mise a jour de la caméra (passé en parametre lors de sa construction) comme le montre l'esquisse de code suivante :

```
public class drawRectOnTouchViewForCamera extends drawRectOnTouchView {
    private CameraControl cam;
    private boolean isDetectionRunning;
    public drawRectOnTouchView(Context context, AttributeSet attrs, camera c
    ,
    boolean etat) {
        super(context,attrs);
        this.cam= new CameraControl(context,c);
        this.isDetectionRunning = stat;
    }


    public void updateWindow(drawRectOnTouchView d){
        int absoluteTop = (int) (getStart().y * 10000 / d.getBottom());
        int absoluteBottom = (int) (getEnd().y * 10000 / d.getBottom());
        int absoluteRight = (int) (getEnd().x * 10000 / d.getRight());
        int absoluteLeft = (int) (getStart().x * 10000 / d.getRight());
```

```
        camC.updateMotionDParam("Top", ""
        + absoluteTop);
        camC.updateMotionDParam("Bottom", ""
        + absoluteBottom);
        camC.updateMotionDParam("Right", ""
        + absoluteRight);
        camC.updateMotionDParam("Left", ""
        + absoluteLeft);
    }


    @Override
    public boolean onTouchEvent(MotionEvent event) {
        /* Recuperer et trace le rectangle comme le defini la classe
    drawRectOnTouchView
        actuelement utilisee */
        super(event);
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
            break;
            case MotionEvent.ACTION_MOVE:
            break;
            case MotionEvent.ACTION_UP:
            if(isDetectionRunning){
                updateWindow(this);
            }
            break;
        }
        return true;
    }
}
```

Ainsi grace à la classe *drawRectOnTouchViewForCamera*, à chaque fois que l'utilisateur mettre a jours la fenetre, celle-ci sera directement (à plus ou moins la latence réseau) mise a jour sur la caméra.

Ajout d'évènements suite à la détection (mail / snapshot)

Lorsque qu'un mouvement est détecté par la caméra, notre notification consiste a prendre un cliché instantanné, puis d'emettre une vibration de l'appareil. Il sera pertinant d'ajouter à l'utilisateur la possibilité de regler le type de notification. Ceci pourrait etre effectuer simplement par l'activation ou la desactivation de ces traitements dans le *SharedPreference* deja implémenté.
Nous pourrions utiliser la meme methode que celle utilisé pour l'activation ou la desactivation de la fenetre d'astuce. Il suffirait ensuite, dans le handler du service de detection de mouvement, de verifier l'etat (activer ou desactiver) des actions implémentées.

Compte utilisateur

La Caméra propose de nombreuses fonctions permettant de l'administrer a distance. Par exemple l'envoie d'une requete *GET* à l'adresse

```
http://myserver/axis-cgi/admin/pwdgrp.cgi?
action=add&user=joe&pwd=foo&grp=axuser&sgrp=axadmin:axoper:axview&comment=
    Joe
```

ajoute l'utilisateur joe avec le mot de pass foo. (Exemple tiré de la documentation officiel).
Parmis les principales fonctions d'adninistration disponible, on retrouve :

- Ajout-Suppression d'utilisateurs/groupes

- Reset factory
- Backup-Restore
- Redemarrage du serveur
- ...

On peut conclure qu'il existe encore un grand nombre de service fourni par la caméra à implémenter, même si la pluspart d'entre eux ne necessite aucunes modification majeur.