A. Lejeune, S. Piérard, M. Van Droogenbroeck et J. Verly. **Utilisation de la Kinect**. *Linux Magazine France*, N°151, pages 16-29, Juillet-Août, 2012.

Utilisation de la Kinect

Antoine Lejeune
Sébastien Piérard
Marc Van Droogenbroeck
Jacques Verly

Juillet 2012

Résumé

Fin 2010, Microsoft lançait la Kinect pour Xbox 360, la première caméra 3D destinée au grand public. Une semaine plus tard sortait la première librairie permettant d'utiliser l'appareil sur un ordinateur personnel. Depuis lors, des centaines d'applications ont vu le jour utilisant l'information de profondeur capturée par la Kinect pour analyser le mouvement humain ou guider des robots. Dans cette article, nous allons voir comment développer une application utilisant la Kinect sous GNU/Linux.

Table des matières

1 Contexte

1.1 Images et caméras de profondeur

Une caméra RGB, comme n'importe quelle webcam, fournit une image représentant la lumière réfléchie par les éléments dans la scène. Le principe physique de capture de la Kinect est différent. La Kinect est une caméra vidéo 3D, c'est-à-dire qu'elle fournit des images de la profondeur de la scène, à savoir la distance entre la caméra et les objets présents dans la scène. Une image de profondeur se présente comme une image monochromatique (c'est-à-dire à niveaux de gris) qui, pour un ensemble d'éléments d'une image (appelés *pixels*) arrangés dans un tableau à deux dimensions, associe une valeur représentative de la distance physique entre le point de la scène et la caméra.

Il existe un grand nombre de techniques pour capturer la profondeur d'une scène. Certaines techniques se basent sur des capteurs spécifiques permettant de réaliser une mesure physique de la profondeur. D'autres techniques tentent de calculer la profondeur de manière indirecte en se basant sur des images couleur, par exemple en examinant les ombres ou le mouvement dans une scène. Quand deux caméras sont utilisées, on parle alors de vision stéréoscopique. La popularité de la vision stéréoscopique s'explique par l'analogie avec le système visuel humain et la disponibilité de caméras couleur à bas prix.

Cependant, les systèmes de vision stéréoscopiques ne sont capables que de calculer la profondeur de la scène pour un ensemble restreint de pixels correspondant à des zones de la scène ayant une forte structure locale, reconnaissable facilement d'une image à l'autre. Par exemple, il est impossible d'obtenir par stéréoscopie une information fiable sur la profondeur d'un objet de couleur unie, par exemple une feuille blanche qui occuperait le champ entier de la caméra.

Des caméras 3D permettent de mieux capturer la profondeur d'une scène. Ces caméras 3D, dites actives, possèdent leur propre source de lumière pour calculer la profondeur. Par exemple, les caméras dites à temps de vol envoient une lumière visible ou infrarouge et mesurent le temps entre l'émission d'une onde et la réception du signal réfléchi par la scène, à la manière d'un écho. Comme la vitesse de la lumière est une constante, la distance s'obtient en multipliant le temps mesuré, préalablement divisé par deux (pour avoir un trajet simple), par la vitesse de la lumière en espace libre, qui vaut 3×10^8 mètres par seconde. Le mode de fonctionnement de la Kinect est encore différent.

1.2 Principes de fonctionnement de la Kinect

La Kinect est une caméra 3D active de type stéréoscopique[2]. Comme le montre la figure 1, elle comprend deux parties :

- une source de lumière infrarouge « structurée » ;
- une caméra infrarouge.

La méthode se base sur les mêmes principes géométriques que ceux utilisés pour la stéréoscopie. En stéréoscopie, on exploite le fait que plus un objet central est proche des deux caméras, plus l'objet est décalé vers la droite dans l'image de gauche et vers la gauche dans l'image de droite. De même, pour un objet distant, le décalage est plus faible. Cependant, grâce à sa propre source lumineuse, la Kinect lève les indéterminations inhérentes à la stéréoscopie dans les zones dépourvues de texture. En effet, la lumière émise par la Kinect permet de donner une texture facilement identifiable à toute la scène filmée. La figure 2 montre un exemple de la mire, d'allure aléatoire, projetée en infrarouge (non visible) par la Kinect.

Comme le montre la figure 1, la Kinect dispose également d'une caméra couleur, en plus du dispositif stéréoscopique de vision 3D. Pour chaque pixel de l'image, elle fournit en fin de compte 4 informations : les 3 composantes de couleur et la profondeur (sauf aux bords de l'image). D'une point de vue technique, la Kinect produit des images ayant une résolution de 640×480 pixels

FIGURE 1 – Face avant de la Kinect : caméra émettant une mire de points dans l'infrarouge, caméra infrarouge, caméra couleur et réseau de microphones

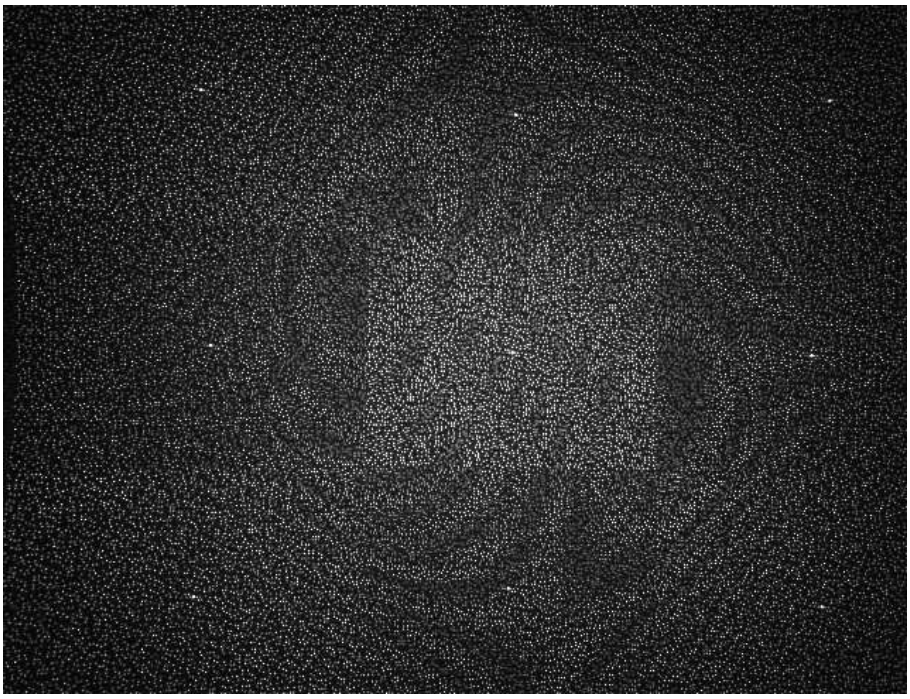


FIGURE 2 – Mire émise par la Kinect telle que perçue par la caméra infrarouge.

à une cadence de 30 images par seconde. L'angle d'ouverture de la caméra 3D de la Kinect est d'environ 58°[3] et est plus petit que celui de la caméra couleur de la Kinect. Ainsi, la zone de la scène vue par la caméra couleur englobe toujours celle vue par la caméra de profondeur.

2 Outils logiciel pour utiliser la Kinect

La Kinect est munie d'un port USB classique qui permet de récupérer les images en couleur et les images de profondeur par logiciel. Le thème principal de la suite de cet article est la récupération des données par logiciel. Mis à part la librairie développée par Microsoft, toutes les librairies mentionnées dans cette section sont multi-plateforme et fonctionnent sous GNU/Linux, Mac OS X et Microsoft Windows.

2.1 Outils spécifiques à la Kinect

Dès la sortie de la Kinect, Adafruit Industries, entreprise de matériel électronique open source, a sponsorisé le développement d'un driver open source en offrant une récompense de 1000$ à la première personne qui proposerait un driver. À peine une semaine plus tard, Hector Martin a publié le premier driver open source pour la Kinect sous le nom de libfreenect, empochant au passage la récompense. La communauté OpenKinect[7] s'est alors créée autour du driver et elle continue depuis lors son développement. Si le but premier d'un driver est la possibilité de récupérer les données de la Kinect (l'image en couleur et l'image de profondeur), certaines librairies offrent davantage.

Fin 2010, PrimeSense, l'entreprise ayant réalisé le capteur de profondeur utilisé par la Kinect, a publié OpenNI[5] qui est un framework open source permettant de développer des applications utilisant des interactions naturelles (voix, mouvements du corps, etc.). Ce framework est accompagné d'un driver open source pour la Kinect et d'un middleware propriétaire appelé NITE qui s'occupe à proprement parler du traitement des images de profondeur. Le middleware NITE permet de segmenter les différentes personnes en face de la caméra, d'obtenir leur pose ou encore de reconnaître certains gestes. Le framework OpenNI est conçu pour être indépendant des capteurs utilisés et des algorithmes de traitement. En pratique cependant, seules les caméras de profondeur basées sur la technologie de PrimeSense sont supportées et il n'existe toujours que le middleware propriétaire NITE pour utiliser les fonctions avancées de traitement.

En mars 2012, trois caméras de profondeur basées sur la même technologie que la Kinect sont disponibles sur le marché :

- la Kinect pour Xbox, première caméra du genre,
- la caméra ASUS XtionPRO, et
- la Kinect for Windows, une version adaptée de la Kinect pour Xbox, permettant notamment d'obtenir la profondeur de scènes « proches ».

Plus récemment, Microsoft a réalisé sa propre librairie, propriétaire, pour utiliser la Kinect sous Windows. Cette librairie offre des fonctions de traitement supérieures à ses concurrentes : un meilleur algorithme de reconnaissance de pose (celui utilisé pour la Xbox) et un algorithme de localisation des sources sonores grâce au réseau de microphones de la Kinect. Notons que Microsoft a publié certaines de ses techniques développées pour la Kinect, notamment l'algorithme de reconnaissance de suivi de la pose[8].

On peut également citer le programme Kinect RGBDemo[1], fourni sous licence LGPL, qui permet de visualiser les images de la Kinect ainsi que de leur appliquer certains traitements (reconstruction de l'environnement, seuillage, ...). Il s'agit d'un ensemble de programmes fort complet.

2.2 Outils génériques

La caméra Kinect offre une voie nouvelle pour l'interprétation d'images. Fort heureusement, de nombreuses méthodes de traitement d'images existent déjà et peuvent s'appliquer aux images

de profondeur. La librairie OpenCV[4] fait figure de référence dans le domaine du traitement des images. Fournie sous licence BSD, elle propose des fonctions pour manipuler les images, les filtrer, analyser des vidéos (suivi, extraction de l'avant-plan, etc) ou détecter et reconnaître des objets.

La librairie Point Cloud Library (pcl)[6], projet sœur d'OpenCV, offre quant à elle des fonctions de traitement pour des données 3D. Ces fonctions s'appliquent sur une représentation des images de profondeur sous la forme de nuage de points 3D (il n'est alors plus question d'une organisation spatiale des points en un tableau). Les traitements possibles avec la librairie pcl sont par exemple la segmentation, l'extraction de points d'intérêt, le rendu de surface, etc. Les méthodes disponibles dans cette librairie ne sont par contre pas toujours exécutées en temps réel, ce qui rend la réalisation d'applications interactives plus compliquée. Cependant, de plus en plus de fonctions sont reprogrammées pour GPU afin d'améliorer les vitesses de traitement. Une implémentation de la technique Kinect Fusion[9], permettant la construction en temps réel d'un modèle 3D d'une scène en fusionnant des images de profondeur capturées de différents points de vue, a récemment fait son apparition dans la librairie pcl.

3 À vous de jouer : préparation de l'environnement pour la récupération et l'interprétation des données de la Kinect

Dans la suite de cette article, nous allons montrer comment réaliser une application interactive en C utilisant la Kinect avec les librairies OpenNI pour l'acquisition et OpenCV pour l'affichage des images.

3.1 OpenNI : présentation et installation

Avant d'expliquer comment effectuer l'installation, parlons de cette librairie. Distribué sous LGPL, le framework OpenNI est une couche abstraite capable de s'interfacer avec différents types de matériel et fournissant des fonctions pour le développement d'applications utilisant des interactions naturelles. Pour un fonctionnement correct, il faut installer des modules implémentant certaines couches ou fonctionnalités du framework. La figure 3 schématise l'architecture du framework.

En principe, OpenNI supporte quatre types de matériel :
- des capteurs 3D tels que la Kinect ;
- des caméras RGB comme celle intégrée dans la Kinect ;
- des caméras infrarouges ;
- des appareils audio (microphone ou réseau de microphones).

Par exemple, le module SensorKinect permet d'utiliser la Kinect avec OpenNI.

Les composants logiciels « middleware » implémentent des fonctions de traitements des données acquises grâce aux différents capteurs. OpenNI propose des fonctions correspondant à quatre types de traitements :
- l'analyse du corps : récupération de la pose du corps sous forme d'une information sur la position et l'orientation des différentes articulations du corps humain ;
- l'analyse du pointage avec la main : permet de suivre la position d'une main pointant vers la caméra ;
- la détection de certains gestes particuliers comme, par exemple, deux mains qui applaudissent ;
- l'analyse de la scène : extraction de l'avant-plan, segmentation du sol, etc.

Il n'existe pour le moment que l'implémentation fournie par le middleware propriétaire NITE.

3.1.1 Organisation de l'API

Avant de réaliser une première application avec OpenNI, examinons l'API d'OpenNI qui est organisée autour de deux concepts essentiels.

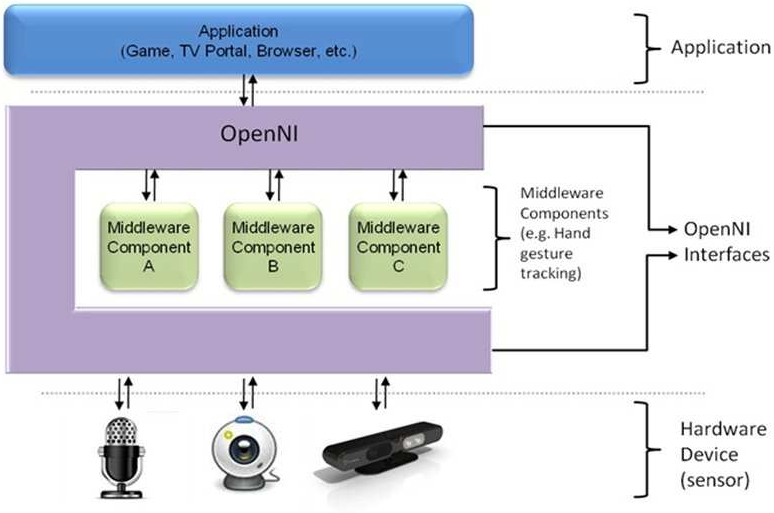


FIGURE 3 – Fonctionnement en couche de OpenNI. Image issue du guide de l'utilisateur d'OpenNI.

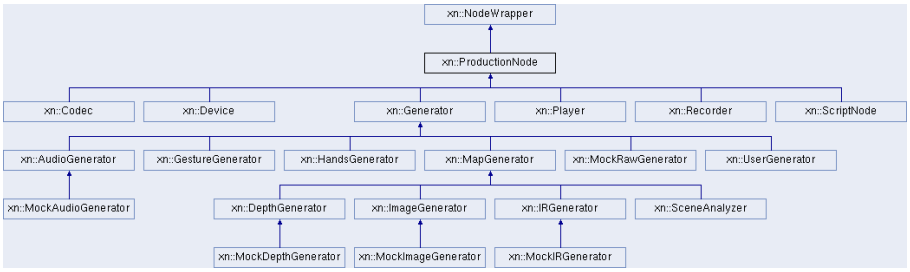


FIGURE 4 – Schéma d'héritage de `xn::ProductionNode` généré par Doxygen.

Le premier concept est celui de « contexte OpenNI », matérialisé par la structure de donnée `XnContext`. C'est le point d'entrée dans le framework. La première étape de toute application utilisant OpenNI consiste à initialiser un contexte. Ce dernier détermine les différents types de données produites, quand de nouvelles données sont accessibles, etc.

Le deuxième concept est relatif à la production de données. Plus précisément, toutes les données produites par OpenNI, que ce soient les données acquises par un capteur ou les données calculées par les composants logiciel, sont accessibles par le biais d'un nœud de production (« Production Node »). Comme on peut le voir dans la figure 4, chaque type de donnée produite est associé à une classe correspondante qui hérite de la classe `xn::ProductionNode`. Dans l'API C, toutes ces classes sont regroupées au sein d'un même type de donnée abstrait `XnNodeHandle`.

La création d'un nœud de production se fait toujours en passant par le contexte OpenNI. Ce dernier va s'occuper de lancer la génération des données, de faire interagir les différents nœuds de production, d'attendre que toutes les données générées soient actualisées, etc. Nous verrons le détail des fonctions à utiliser plus loin.

3.1.2 Installation de OpenNI

On en connaît assez pour l'instant sur OpenNI. Le temps est venu d'installer la librairie.

OpenNI n'est pas encore packagé par la majorité des grandes distributions GNU/Linux. Il est donc nécessaire de l'installer « manuellement ». Pour se faire, deux choix s'offrent à nous : une compilation « maison » ou récupérer une version pré-compilée. Par simplicité, nous optons dans cet article pour la deuxième solution. Pour une bonne compatibilité avec le module d'interface avec la Kinect, il faut choisir la version *unstable* du framework.

Pour récupérer une version pré-compilée d'OpenNI, il faut se rendre sur le site http ://www.openni.org/Downloads/ et sélectionner, par exemple pour un Linux x64, « **OpenNI Binaries > Unstable > OpenNI Unstable Build for Ubuntu 12.04 x64 (64-bit) v1.5.4.0** ». Le fichier téléchargé sera alors « **openni-bin-dev-linux-x64-v1.5.4.0.tar.bz2** ».

L'installation est très simple :

```
$ tar jxf openni-bin-dev-linux-x64-v1.5.4.0.tar.bz2
$ cd OpenNI-Bin-Dev-Linux-x64-v1.5.4.0/
$ sudo ./install.sh
```

Ces instructions procèdent à l'installation des différentes librairies partagées dans **/usr/lib**, des fichiers d'include dans **/usr/include/ni** et de deux programmes :

- **niReg** : ce logiciel permet d'enregistrer localement, auprès d'OpenNI, les différents modules installés (drivers et composants logiciel)
- **niLicense** : ce logiciel gère les éventuelles clés de licence pour les modules complémentaires.

Un ensemble d'exemples est également fourni dans le dossier Samples de l'archive téléchargée.

3.1.3 Installation du module SensorKinect

Le site officiel d'OpenNI propose également un module matériel nommé « PrimeSensor ». Cependant, ce dernier ne fonctionne pas avec la caméra de Microsoft. Il faut installer le module SensorKinect, qui lui est capable d'acquérir des images de profondeur, couleur et infrarouge avec la Kinect.

Le code source est disponible sur github à cette adresse : https ://github.com/avin2/SensorKinect. Une version pré-compilée est également disponible dans le répertoire Bin de ce dépôt. Par exemple, pour un Linux 64 bits, il faut télécharger le fichier https ://github.com/avin2/SensorKinect/blob/unstable/Bin/Sensor

L'installation se fait de la même façon que pour OpenNI :

```
$ tar jxf SensorKinect093-Bin-Linux64-v5.1.2.1.tar.bz2
$ cd Sensor-Bin-Linux-x86-v5.1.2.1/
$ sudo ./install.sh
```

Le script d'installation s'occupe lui-même de s'enregistrer auprès d'OpenNI.

Pour tester si tout fonctionne correctement, il suffit de tester les exemples fournis avec OpenNI. Attention tout de même, les dernières versions du noyau Linux sont fournies avec un driver pour la caméra couleur de la Kinect qui est incompatible avec le module SensorKinect. Il faut soit décharger le module **gspca_kinect** :

```
$ lsmod | grep gspca_kinect
gspca_kinect            12792  0
gspca_main              27610  1 gspca_kinect
$ sudo rmmod gspca_kinect
```

soit le « blacklister » avant d'utiliser la Kinect avec OpenNI :

```
$ sudo su
# echo "blacklist␣gspca_kinect"
    >> /etc/modprobe.d/blacklist.conf
```

Ainsi, le module concurrent pour la Kinect ne sera plus chargé au prochain démarrage.

3.1.4 Installation du middleware NITE

NITE étant un module propriétaire, il n'existe qu'une version binaire disponible sur le site d'OpenNI. Sur le site http ://www.openni.org/Downloads/OpenNIModules.aspx, il faut sélectionner, par exemple pour un Linux x64, **« OpenNI Compliant Middleware Binaries > Unstable > PrimeSense NITE Unstable Build for Ubuntu 12.04 x64 (64-bit) v1.5.2.21** ». Le fichier téléchargé sera alors « **nite-bin-linux-x64-v1.5.2.21.tar.bz2** ». Pour l'installer, il faut faire :

```
$ tar jxf nite-bin-linux-x64-v1.5.2.21.tar.bz2
$ cd NITE-Bin-Dev-Linux-x86-v1.5.2.21
$ sudo ./install.sh
```

Il y a également un script « **uninstall.sh** » pour désinstaller le middleware. Le dossier « Documentation » contient un document décrivant les différentes caractéristiques des algorithmes implémentés (cadre de fonctionnement, limitations, ...).

3.2 OpenCV

OpenCV est habituellement packagé dans les principales distributions. Sous Ubuntu 12.04, on peut faire :

```
$ sudo apt-get install libcv2.3 libcv-dev
   libhighgui2.3 libhighgui-dev
```

4 Créer une application avec OpenNI

Nous allons maintenant créer un programme de visualisation avec les librairies OpenNI et OpenCV donnant toutes les clés qui permettent de développer une application interactive complète. Au cours de la réalisation de cette application, nous allons voir comment :

- récupérer les images de profondeur et en couleur de la Kinect ;
- faire correspondre les pixels de l'image de profondeur et ceux de l'image en couleur ;
- transformer les coordonnées projectives (colonne et rangée du pixel avec sa profondeur) en coordonnées cartésiennes exprimées en millimètre et dans un repère centré sur la Kinect ;
- réaliser la détection de personne devant la caméra ;
- réaliser une détection de pose ;
- obtenir la position de différentes articulations (squelette) de la personne devant la caméra ;
- afficher le squelette et les différentes images capturées avec OpenCV.

OpenCV et OpenNI proposent tous les deux une interface C et C++. Dans cet article, nous travaillerons en C.

4.1 Initialiser le contexte OpenNI

L'extrait de code C suivant montre comment on initialise et ferme un contexte OpenNI.

```
#include <XnOpenNI.h>
#include <stdlib.h>
#define NI_CHECK_ERROR(status, message) \
 if(status!= XN_STATUS_OK) { \
  xnPrintError(status, message); \
  exit(EXIT_FAILURE); \
 }

int main(int argc, char **argv)
{
 XnStatus status;
 XnContext *context;
 status = xnInit(&context);
 NI_CHECK_ERROR(status,"Initialisation d'OpenNI");
 [...]
 xnContextRelease(context);
 return 0;
}
```

Les fonctions dont l'exécution peut produire une erreur retourne un code d'erreur du type **XnStatus**. Plusieurs fonctions permettent de convertir ce statut en un message d'erreur cohérent. On a donc défini une macro qui permet de gérer les erreurs de OpenNI de manière concise : dès qu'une erreur se produit, un message d'erreur est affiché dans la console et le programme se termine.

Les fonctions les plus simples pour initialiser et fermer un contexte sont **xnInit** et **xnContextRelease** respectivement.

4.2 Créer des nœuds de production pour les caméras de profondeur et couleur

Pour créer un nœud de production, il suffit d'appeler la fonction créant le type adéquat de nœud de production. Elles ont toutes une même nomenclature :

```
XnStatus xnCreateXXX(XnContext *, XnNodeHandle *,
    XnNodeQuery *, XnEnumerationErrors *);
```

Le premier argument des constructeurs est toujours le contexte OpenNI. Le deuxième argument est un pointeur vers une variable de type **XnNodeHandle** qui représente un nœud de production. Il ne faut pas allouer soi-même la mémoire pour ce type de donnée. Les deux derniers arguments sont optionnels. L'avant-dernier permet de sélectionner un nœud particulier quand il existe plusieurs composants logiciel implémentant ce nœud de production. Le dernier argument permet de récupérer la liste de toutes les erreurs survenues au cours de l'appel du constructeur.

Nous allons donc créer deux nœuds de production pour notre application ; un pour les images de profondeur et un pour les images en couleur :

```
XnNodeHandle depthGenerator, imageGenerator;
status = xnCreateDepthGenerator(context, &depthGenerator, NULL, NULL);
NI_CHECK_ERROR(status, "Création du nœud de production des images de profondeur");
status = xnCreateImageGenerator(context, &imageGenerator, NULL, NULL);
NI_CHECK_ERROR(status, "Création du nœud de production des images en couleur");
```

4.3 Recalage des images produites

Un pixel de l'image de profondeur ne correspond pas exactement à un pixel de l'image couleur. En effet, les deux capteurs sont situés à des positions physiques différentes sur la Kinect. OpenNI fournit un mécanisme permettant de recaler des images de différents générateurs. Dans les termes d'OpenNI, cela revient à changer le point de vue d'un générateur d'images pour le faire correspondre au point de vue d'un autre générateur. Cela nécessite que le driver implémentant ce générateur en soit capable. OpenNI fournit un mécanisme pour vérifier si le driver fournit cette fonctionnalité.

```
if(!xnIsCapabilitySupported(depthGenerator, XN_CAPABILITY_ALTERNATIVE_VIEW_POINT))
{
 fprintf(stderr, "Impossible de changer le point de vue des images de profondeur\n");
 exit (EXIT_FAILURE);
}
else
{
 xnSetViewPoint(depthGenerator, imageGenerator);
}
```

C'est donc très simple de spécifier le nouveau point de vue à utiliser. Il suffit de passer en argument les deux générateurs d'images concernés par le changement de point de vue. On peut noter qu'avec la Kinect, il vaut mieux changer le point de vue des images de profondeur vers le point de vue de la caméra couleur. En effet, l'angle de vue de la caméra couleur est supérieur à celui de la caméra de profondeur et la partie de la scène capturée en profondeur est toujours incluse dans celle capturée en couleur. La figure 5 montre l'effet du recalage sur une image capturée avec la Kinect.

4.4 Boucle principale du programme

Sauf surprise —tout dépend de savoir si vous êtes en train de faire une démonstration...—, le programme est fonctionnel. Celui-ci doit donc constamment récupérer les différentes images et les afficher. On doit donc réaliser une boucle infinie dans laquelle on va implémenter ces opérations. Si d'autres traitements sont nécessaires, c'est généralement dans cette boucle qu'il faut les réaliser.

Dans le cas où le traitement à réaliser se fait plus rapidement que l'acquisition d'une nouvelle image, il faudra mettre le programme en attente. OpenNI fournit des fonctions qui permettent de réaliser cela automatiquement.

(a) (b) (c)

FIGURE 5 – (a) Image couleur, Images de profondeur (b) non-recalée et (c) recalée.

En premier lieu, il faut demander à OpenNI de démarrer les générateurs précédemment créés :

```
xnStartGeneratingAll(context);
```

Ensuite, on peut lancer la boucle et récupérer les différentes images.

```
int die = 0;
while(!die)
{
 status = xnWaitAndUpdateAll(context);
 if(status!=XN_STATUS_OK)
 {
  die = 1;
  continue;
 }
 [...]
}
```

Nous utilisons une variable booléenne **die** pour, le cas échéant, sortir de la boucle en cas d'erreur. La fonction **xnWaitAndUpdateAll** permet d'attendre que tous les générateurs aient acquis ou produit de nouvelles données. Elle permet aussi d'avoir une synchronisation entre les images de profondeur et couleur par exemple.

4.5 Récupération des données et interfaçage avec OpenCV

OpenNI fournit plusieurs fonctions pour accéder aux données : une version simple retournant un pointeur vers les données de l'image (**xnGetRGB24ImageMap**, **xnGetGrayscale8ImageMap**, **xnGetDepthMap**) ou une version plus évoluée retournant des meta-données sur les images (taille de l'image, profondeur maximale, ...). Nous allons utiliser la première version en dehors de la boucle principale pour obtenir les informations nécessaires à la création des structures OpenCV qui vont contenir les images.

```
XnImageMetaData *imageMetaData = xnAllocateImageMetaData();
XnDepthMetaData *depthMetaData = xnAllocateDepthMetaData();
xnGetDepthMetaData(depthGenerator, depthMetaData);
xnGetImageMetaData(imageGenerator, imageMetaData);
IplImage *depthMap = cvCreateImage(cvSize (depthMetaData->pMap->FullRes.X,
                                           depthMetaData->pMap->FullRes.Y),
                                   IPL_DEPTH_16U, 1));
IplImage *imageMap = cvCreateImage (cvSize (imageMetaData->pMap->FullRes.X,
                                            imageMetaData->pMap->FullRes.Y),
                                   IPL_DEPTH_8U, 3));
uint16_t maxDepth = depthMetaData->nZRes;
xnFreeDepthMetaData(depthMetaData);
xnFreeImageMetaData (imageMetaData);
[...]
while(!die)
{
 [...]
 memcpy(depthMap->imageData, xnGetDepthMap(depthGenerator), depthMap->imageSize);
 memcpy(imageMap->imageData, xnGetRGB24ImageMap(imageGenerator), imageMap->imageSize);
 [...]
}
cvReleaseImage(&depthMap);
cvReleaseImage(&imageMap);
```

Le code des lignes 1 à 4 alloue la mémoire nécessaire pour contenir les méta-données avec les fonction `xnAllocateXXXMetaData` et on récupère les méta-données relatives aux générateurs. Les structures `XnImageMetaData` et `XnDepthMetaData` sont semblables ; elles contiennent toutes les deux :

- un pointeur `pMap` vers une structure `XnMapMetaData` qui contient des informations sur la taille, le format ou la fréquence d'acquisition des images, et
- un pointeur `pData` vers les données de l'image.

Pour les images de profondeur, il y a en plus un champs `nZRes` contenant la profondeur maximale de la caméra. Cette donnée est utile, notamment pour afficher les cartes de profondeur comme une image en niveau de gris et donc ramener l'ensemble des valeurs de profondeur entre 0 et 255. Aux lignes 12 et 13, on libère la mémoire allouée pour les méta-données avec les fonctions `xnFreeXXXMetaData`.

De la ligne 5 à 10, on déclare des images OpenCV du type `IplImage`. On utilise la fonction `cvCreateImage` qui crée et alloue la mémoire d'une telle image. En fin de programme, on libère la mémoire en utilisant la fonction `cvReleaseImage` (lignes 22 et 23). La fonction `cvCreateImage` prend, en argument, la taille de l'image, le nombre de bits par pixel et le nombre de canaux. Les données de profondeur de la Kinect sont encodées sur 2 octets par pixel (`IPL_DEPTH_16U`) et représentent sa profondeur en millimètre. Pour l'image couleur, on va récupérer une version encodée en RVB et donc nécessitant trois canaux de 8 bits par pixel (`IPL_DEPTH_8U`).

Dans la boucle, aux lignes 18 et 19, on copie les données fournies par OpenNI dans les champs `imageData` des images `IplImage`. Le champs `imageSize`, qui contient la taille de l'image en octet, sert à spécifier la longueur de la zone mémoire à copier.

La Kinect n'est pas toujours capable de calculer la profondeur sur toute la scène. En effet, il faut que la mire émise par la Kinect soit correctement réfléchie par la scène. Or certains revêtements ou les vitres ne réfléchissent pas correctement la mire. De plus, de par l'écartement entre le projecteur et la caméra infrarouge, des objets en avant de la scène peuvent faire « ombre » à des surfaces en retrait et la mire n'est donc pas visible par la caméra infrarouge dans ces zones d'ombre. Dans l'image de profondeur récupérée de la Kinect, les pixels pour lesquels la caméra n'a pas été en mesure de calculer la profondeur sont mis à 0. Il faut également noter que la Kinect n'est pas capable de mesurer des profondeurs inférieures à quelques dizaines de centimètre à cause d'un phénomène de saturation (trop de lumière réfléchie).

4.6 Traitement des images et changement du système de coordonnées

Généralement, après avoir réaliser l'acquisition des images, on veut réaliser un traitement dessus pour résoudre un problème. Dans notre cas, nous allons simplement calculer la distance moyenne entre la Kinect et chacun des pixels de l'image (en ignorant les pixels pour lesquels nous n'avons pas de mesure de profondeur).

Le calcul de la distance moyenne est réalisé en deux étapes. En premier lieu, les coordonnées « projectives » (colonne, rangée, profondeur) de chaque pixel sont converties en coordonnées cartésiennes. Le système cartésien a son origine au niveau de la Kinect avec l'axe X pointant vers sa droite, l'axe Y vers le haut et l'axe Z vers l'avant (voir figure 6). OpenNI réalise cette conversion en allant récupérer les paramètres de calibration enregistrés en usine dans la Kinect. Tout est alors prêt pour calculer la distance entre chaque pixel et l'origine du système cartésien dans le but d'obtenir la moyenne.

```
1  float avgDist = 0;
2  int validPixels = 0;
3  XnPoint3D *projectiveCoordinates = malloc (sizeof (XnPoint3D) * depthMap->width * depthMap->height);
4  XnPoint3D *realCoordinates = malloc (sizeof (XnPoint3D) * depthMap->width * depthMap->height);
5  for (int row = 0; row < depthMap->height; row++)
6  {
7   for (int col = 0; col < depthMap->width; col++)
8   {
9    uint16_t depth = CV_IMAGE_ELEM (depthMap, uint16_t, row, col);
10   if (depth == 0)
11    continue;
12   XnPoint3D *p = &projectiveCoordinates[validPixels++];
13   p->X = row; p->Y = col; p->Z = depth;
```

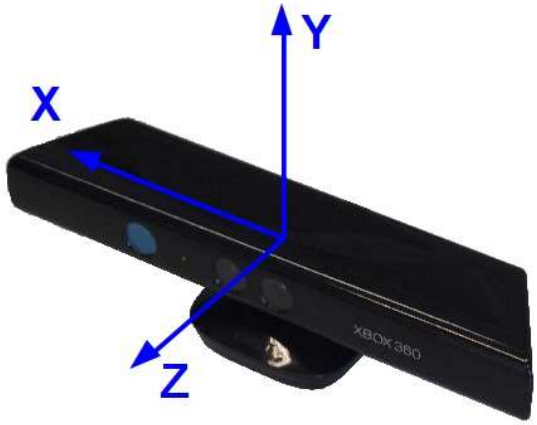


FIGURE 6 – Système de coordonnées « réelles » utilisés par OpenNI.

```
14  }
15 }
16 xnConvertProjectiveToRealWorld (depthGenerator, validPixels, projectiveCoordinates, realCoordinates);
17 for (int i = 0; i < validPixels; i++)
18 {
19  XnPoint3D p = realCoordinates[i];
20  avgDist += sqrt (p.X * p.X + p.Y * p.Y + p.Z * p.Z);
21 }
22 avgDist /= validPixels;
```

Concrètement, on commence par déclarer les variables **avgDist** et **validPixels** qui vont contenir respectivement la distance moyenne recherchée et le nombre de pixels valides, c'est-à-dire ayant une profondeur supérieure à 0. Ensuite, deux tableaux de **XnPoint3D** sont alloués pour contenir tous les pixels de l'image ; même si certains pixels non valides seront écartés, la taille de **projectiveCoordinates** et de **realCoordinates** est choisie pour éviter tout débordement de la mémoire. De la ligne 5 à 15, on parcourt un à un chaque pixel de l'image et s'il est valide, il est inséré dans le tableau **projectiveCoordinates** et le compteur de pixels valides est incrémenté (lignes 12 et 13). La macro **CV_IMAGE_ELEM** (ligne 9) permet d'accéder directement à un pixel d'une image.

À la ligne 16, a fonction **xnConvertProjectiveToRealWorld** convertit les coordonnées de tous les pixels valides. On récupère les nouvelles coordonnées dans le tableau **realCoordinates**. La fonction **xnConvertProjectiveToRealWorld** permet de réaliser la conversion inverse si cela est nécessaire (c'est utile parfois pour faire de la réalité augmentée). Remarquons que la conversion des coordonnées de tous les pixels est réalisée en un seul appel de fonction. En effet, cette conversion pixel par pixel ralentirait considérablement le programme.

Après la conversion des pixels en coordonnées cartésiennes, le programme calcule la distance euclidienne de chaque pixel à l'origine (la Kinect elle-même) et accumule les distances dans la variable **avgDist** ; enfin, il calcule la moyenne de cette distance (ligne 17 à 22).

4.7 Détection et suivi de personnes

L'interface d'OpenNI qui permet de détecter les personnes situées devant une caméra s'appelle un générateur d'utilisateur (*User Generator*). De base, ce générateur est capable de détecter les différents utilisateurs situés devant la caméra et de fournir une image où chaque pixel est annoté avec l'identifiant de l'utilisateur dont il fait partie. OpenNI propose plusieurs extensions au générateur d'utilisateur pour interpréter les mouvements humains, dont notamment :

- l'extension « squelette » qui permet d'obtenir la position de différentes articulations du corps humain ;
- l'extension de détection de pose qui permet de détecter quand un utilisateur se place selon une pose précise (par exemple, les bras en l'air).

La seule implémentation disponible du générateur d'utilisateur est celle développée par la société PrimeSense : le middleware propriétaire NITE.

Les données générées étant dépendantes du contenu de la scène, OpenNI fournit un mécanisme de fonction de rappel, appelée aussi fonction de callback, pour signaler au programme différents événements comme l'apparition d'un nouvel utilisateur. Créons maintenant notre générateur d'utilisateur. Avant la boucle principale :

```
[...]
status = XnNodeHandle userGenerator = xnCreateUserGenerator (context, &userGenerator, NULL, NULL);
NI_CHECK_ERROR (status, "Erreur de création du générateur d'utilisateur");
[...]
```

OpenNI offre la possibilité d'avoir une fonction de rappel pour la détection d'un nouvel utilisateur et une autre pour la perte d'un utilisateur. Le prototype de ces fonctions de rappel est :

```
void (*userCallback) (XnNodeHandle userGenerator, XnUserID user, void *cookie);
```

Dans notre boucle principale, on peut obtenir l'ensemble des utilisateurs détectés par OpenNI :

```
while (!die)
{
 [...]
 uint16_t userCount = xnGetNumberOfUsers (userGenerator);
 XnUserID users[userCount];
 xnGetUsers (userGenerator, users, &userCount);
 [...]
}
```

La fonction **xnGetUsers** prend les arguments suivants : le générateur d'utilisateur, un tableau de **XnUserID** préalablement alloué et un pointeur vers la taille du tableau alloué. À la sortie de la fonction, le pointeur est mis à jour pour contenir le nombre d'utilisateur détecté. L'utilisation de **xnGetNumberOfUsers** est donc redondante mais permet de s'assurer d'avoir un tableau assez grand pour contenir tous les utilisateurs détectés.

4.7.1 Détection de pose

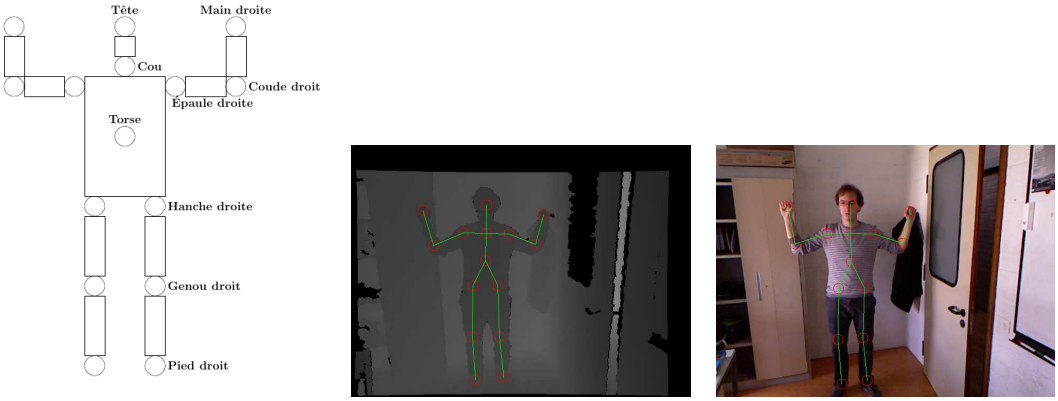
La détection de pose est une extension du générateur d'utilisateur. Pour vérifier si elle est disponible, on peut faire :

```
if(!xnIsCapabilitySupported (userGenerator, XN_CAPABILITY_POSE_DETECTION))
{
 return -1;
}
```

La liste des poses détectables peut être obtenue avec la fonction **xnGetAllAvailablePoses**. Les poses sont représentées par une chaîne de caractères. Par exemple, la figure 7a montre la pose « Psi » reconnaissable par le middleware NITE. Pour détecter une pose, il faut premièrement demander à OpenNI de tenter de détecter cette pose :

```
xnStartPoseDetection(userGenerator, "Psi", user);
```

Il faut donc spécifier la pose recherchée et l'identifiant de l'utilisateur pour lequel on veut détecter une pose. La détection étant dépendante du contenu de la scène, une fonction de rappel doit être utilisée pour signaler quand la pose est détectée.



(a) (b) (c)

FIGURE 7 – (a) Schéma des différentes articulations supportées par NITE (issu de la documentation de NITE) (b) et (c) Articulations détectées par NITE, affichées respectivement sur la carte de profondeur et sur l'image en couleurs.

```
1 void poseDetectedCallback(XnNodeHandle userGenerator, const XnChar *poseName, XnUserID user, void *
      cookie);
2 [...]
3 XnCallbackHandle poseCallbackHandle;
4 status = xnregisterToPoseDetected (userGenerator, &poseDetectedCallback, cookie, &poseCallbackHandle)
      ;
5 [...]
```

La variable **poseCallbackHandle** permet de désenregistrer la fonction de rappel si nécessaire. La variable **cookie** est un pointeur vers des données qu'on veut éventuellement récupérer depuis la fonction de rappel.

Il ne reste plus qu'à implémenter la fonction de rappel pour qu'elle effectue une tâche voulue quand une pose est détectée. Nous la définirons bientôt. D'abord, regardons comment fonctionne la capacité « squelette ».

4.7.2 La capacité « squelette »

La capacité « squelette » permet de récupérer la positions de différentes articulations du corps humain, à savoir :

- la **tête** (XN_SKEL_HEAD) ;
- le **cou** (XN_SKEL_NECK) ;
- le **torse** (XN_SKEL_TORSO) ;
- les **épaules** (XN_SKEL_RIGHT_SHOULDER, XN_SKEL_LEFT_SHOULDER) ;
- les **coudes** (XN_SKEL_RIGHT_ELBOW, XN_SKEL_LEFT_ELBOW) ;
- les **mains** (XN_SKEL_RIGHT_HAND, XN_SKEL_LEFT_HAND) ;
- les **hanches** (XN_SKEL_RIGHT_HIP, XN_SKEL_LEFT_HIP) ;
- les **genoux** (XN_SKEL_RIGHT_KNEE, XN_SKEL_LEFT_KNEE) ;
- les **pieds** (XN_SKEL_RIGHT_FOOT, XN_SKEL_LEFT_FOOT) ;
- les clavicules (XN_SKEL_RIGHT_COLLAR, XN_SKEL_LEFT_COLLAR) ;
- les poignets (XN_SKEL_RIGHT_WRIST, XN_SKEL_LEFT_WRIST) ;
- la taille (XN_SKEL_WAIST) ;
- les chevilles (XN_SKEL_RIGHT_ANKLE, XN_SKEL_LEFT_ANKLE) ; et
- le bout des doigts (XN_SKEL_RIGHT_FINGERTIP, XN_SKEL_LEFT_FINGERTIP).

Le middleware NITE ne fournit l'information de position que pour les articulations mentionnées en gras ci-dessus (voir figure 7a). On peut également spécifier un sous-ensemble d'articulations à suivre , qu'il s'agisse de toutes les articulations (XN_SKEL_PROFILE_ALL), uniquement des membres supérieurs (XN_SKEL_PROFILE_UPPER), uniquement des membres inférieurs

(XN_SKEL_PROFILE_LOWER) ou uniquement de la tête et des mains (XN_SKEL_PROFILE_HEAD_HANDS). Dans notre cas, nous voulons suivre toutes les articulations possibles :

```
1 xnSetSkeletonProfile (userGenerator, XN_SKEL_PROFILE_ALL);
```

Avant d'obtenir ces informations, il faut que la librairie se calibre pour l'utilisateur. La calibration peut être automatique (comme depuis la version 1.5 de NITE) ou dépendante d'une pose particulière (précédemment, NITE demandait la détection de la pose « Psi » pour calibrer le squelette).

On implémente alors la fonction de rappel de détection d'un nouvel utilisateur pour démarrer la calibration si elle est automatique (ligne 10 à 13 du code repris ci-après), ou lancer la détection de pose si elle est nécessaire (ligne 4 à 9) :

```
1  void newUserCallback (XnNodeHandle userGenerator, XnUserID user, void *cookie)
2  {
3   printf ("Nouvel␣utilisateur␣avec␣l'identifiiant␣%d\n", user);
4   if (xnNeedPoseForSkeletonCalibration (userGenerator))
5   {
6    char calibrationPose[20];
7    xnGetSkeletonCalibrationPose (userGenerator, calibrationPose);
8    xnStartPoseDetection (userGenerator, calibrationPose, user);
9   }
10  else
11  {
12   xnRequestSkeletonCalibration (userGenerator, user, true);
13  }
14 }
```

La fonction **xnNeedPoseForSkeletonCalibration** est une fonction booléenne indiquant le besoin d'une détection de pose pour la calibration. La fonction **xnGetSkeletonCalibrationPose** permet de récupérer le nom de la pose à détecter pour la calibration et **xnStartPoseDetection** demande au générateur d'utilisateur de lancer la détection. Implémentons maintenant la fonction de rappel pour une pose détectée :

```
1 void poseDetectedCallback (XnNodeHandle userGenerator, const XnChar *poseName, XnUserID user, void *
      cookie)
2 {
3  xnRequestSkeletonCalibration (userGenerator, user, true);
4 }
```

La fonction **xnRequestSkeletonCalibration** permet donc de calibrer le squelette. Cette calibration n'est pas immédiate et peut ne pas fonctionner correctement. Une fonction de rappel est encore utilisée pour signaler la fin d'une calibration, réussie ou non.

```
1  void calibrationCompleteCB(XnNodeHandle userGenerator, XnUserID user,
2                             XnCalibrationStatus status, void *cookie)
3  {
4   if(status == XN_CALIBRATION_STATUS_OK)
5   {
6    xnStartSkeletonTracking(userGenerator, user);
7   }
8   else if(xnNeedPoseForSkeletonCalibration(userGenerator))
9   {
10   char calibrationPose[20];
11   xnGetSkeletonCalibrationPose(userGenerator, calibrationPose);
12   xnStartPoseDetection(userGenerator, pose, user);
13  }
14  else
15  {
16   xnRequestSkeletonCalibration(userGenerator, user, true);
17  }
18 }
```

Cette fonction vérifie si la calibration s'est terminée correctement. Dans ce cas, le suivi du squelette démarre avec la fonction **xnStartSkeletonTracking** (ligne 4 à 7). Sinon, on tente une nouvelle calibration (ligne 8 à 17). La fonction de rappel doit être donnée à OpenNI avec la fonction **xnRegisterToCalibrationComplete**.

Tout est maintenant en ordre pour récupérer la position des différentes articulations.

4.7.3 Récupérer la position des articulations et dessin du squelette

Dans la boucle principale, on va vérifier pour chaque utilisateur si son squelette est actuellement suivi. Dans ce cas, on dessine les différentes articulations sur l'image couleur de la Kinect :

```
1  for(int i = 0; i < userCount; i++)
2  {
3   if(!xnIsSkeletonTracking(userGenerator, users[i])
4    continue;
5   drawJoint(depthGenerator, userGenerator, imageMap, users[i], XN_SKEL_HEAD);
6   [...]
7   drawJoint(depthGenerator, userGenerator, imageMap, users[i], XN_SKEL_RIGHT_FOOT);
8  }
```

Regardons maintenant la fonction **drawJoint** :

```
1  void drawJoint(XnNodeHandle depthGenerator, XnNodeHandle userGenerator,
2                 IplImage *img, XnUserID user, XnSkeletonJoint joint)
3  {
4   if(!xnIsSkeletonTracking(userGenerator, user))
5    return;
6   XnSkeletonJointPosition jointPosition;
7   xnGetSkeletonJointPosition(userGenerator, user, joint, &jointPosition);
8   if(jointPosition.fConfidence == 0.0)
9   {
10   return;
11  }
12  // Convert the real world coordinate of the joint to projective
13  XnPoint3D projectivePosition;
14  xnConvertRealWorldToProjective(depthGenerator, 1, &projectivePosition,
15    &jointPosition.position);
16  CvPoint p = cvPoint(projectivePosition.X, projectivePosition.Y);
17  cvCircle(img, p, 10, cvScalar (255, 0, 0, 0), 0, 8, 0);
18 }
```

La fonction **xnGetSkeletonJointPosition** permet de récupérer la position d'une articulation sous la forme d'une structure **XnSkeletonJointPosition** (lignes 6 et 7). Cette structure contient deux champs :

- le champs **position** de type **XnPoint3D** qui donne les coordonnées cartésiennes de l'articulation et
- le nombre flottant **fConfidence** indiquant le degré de confiance que l'algorithme a dans la position qu'il a calculée.

Pour les articulations non-suivies par NITE, ce degré de confiance vaudra toujours 0. C'est pourquoi dans ce cas, nous quittons tout de suite la fonction (ligne 8 à 11). Ensuite, pour dessiner l'articulation, il faut réaliser un changement de système de coordonnées pour récupérer la position du pixel correspondant à l'articulation (ligne 12 à 15). Nous utilisons enfin la fonction **cvCircle** d'OpenCV pour dessiner un cercle de rayon de 10 pixels et de couleur rouge (lignes 16 et 17). Le résultat est montré à la figure 7c.

Ouf, il est maintenant temps d'enfin afficher les images.

4.8 Afficher les images

OpenCV fournit des fonctions pour afficher des images dans une fenêtre. Cependant, ces fonctions ne sont pas compatibles avec le format utilisé par les images de profondeur (ce serait trop simple...). Ces fonctions d'affichage ne sont capables d'afficher que des images encodées en RVB (1 octet par couleur et par pixel) ou encodée en niveau de gris (1 octet par pixel). Pour afficher une carte de profondeur en niveau de gris, il faut donc normaliser les données entre 0 et 255. À la valeur 0, on assigne la profondeur minimale, soit 0 et à la valeur 255, on assigne la profondeur maximale de la Kinect, précédemment récupérée à partir des méta-données des images de profondeur.

On définit une fonction **displayRangeImage** prenant comme argument le nom de la fenêtre d'affichage, un pointeur vers l'image et la valeur de profondeur maximale.

```
1  void displayRangeImage(const char *name, const IplImage *image, const uint16_t maxDepth)
2  {
3   IplImage *normalized = cvCreateImage(cvGetSize(image), IPL_DEPTH_8U, 1);
4   for(int row = 0; row < image->height; row++)
5   {
6    for(int col = 0; col < image->width; col++)
```

```
7    {
8     CV_IMAGE_ELEM(normalized, uint8_t, row, col) = (255 * CV_IMAGE_ELEM(image, uint16_t, row, col)) /
          maxDepth;
9    }
10  }
11  cvShowImage(name, normalized);
12  cvReleaseImage(&normalized);
13 }
```

Dans cette fonction, on commence par créer une image **normalized** ayant les mêmes dimensions que l'image de profondeur **image** mais qui ne possède qu'un seul canal de 8 bits par pixel (ligne 3). On parcourt alors tous les pixels de l'image de profondeur et pour chaque pixel, on normalise la valeur (ligne 4 à 10).

Après cette boucle, à la ligne 11, on appelle la fonction **cvShowImage** d'OpenCV pour afficher l'image nouvellement créée. Cette fonction la copie pour ses propres besoins et donc, en dernier lieu, à la ligne 12, on libère la mémoire allouée pour l'image **normalized**.

Avec OpenCV, il n'y a pas besoin de créer une fenêtre avant d'afficher le contenu d'une image. La fonction **cvShowImage** se charge de créer la fenêtre si aucune fenêtre du même nom n'existe précédemment.

OpenCV affiche les images couleurs en supposant un modèle de couleur BVR (Bleu Vert Rouge) au lieu de l'habituel RVB. On définit ainsi une fonction **displayRGBImage** qui va réaliser la conversion du modèle de couleur et afficher l'image. On utilise la fonction OpenCV **cvCvtColor** qui permet de réaliser cette conversion.

```
1 void displayRGBImage(const char *name, const IplImage *image)
2 {
3  IplImage *imageBGR = cvCreateImage(cvGetSize(image), IPL_DEPTH_8U, 3);
4  cvCvtColor(image, imageBGR, CV_RGB2BGR);
5  cvShowImage(name, imageBGR);
6  cvReleaseImage(&imageBGR);
7 }
```

Revenons dans la boucle principale du programme :

```
1 while (!die)
2 {
3  [...]
4  displayRangeImage("Image␣de␣profondeur", depthMap);
5  displayRGBImage("Image␣couleur", imageMap);
6  char c = cvWaitKey (5);
7  if (c == 'q')
8   die = 1;
9 }
```

À la fin de cette boucle, on affiche nos deux images et on fait ensuite appel à la fonction OpenCV **cvWaitKey** qui prend en argument le nombre de millisecondes à attendre qu'une touche du clavier soit pressée. Une valeur de 0 indique une attente infinie.

La fonction **cvWaitKey** est particulièrement importante : elle sert à récupérer les événements du clavier mais aussi à exécuter les tâches d'affichage en attente. Cela veut dire que si **cvWaitKey** n'est pas appelée après un appel à **cvShowImage**, l'image ne sera jamais rafraîchie.

4.9 Compilation

Maintenant que le programme est fini, on l'enregistre dans le fichier **main.c**. Le programme peut maintenant être compilé et exécuté grâce aux commandes suivantes :

```
1 $ gcc main.c -std=gnu99 `pkg-config --cflags --libs opencv` -I/usr/include/ni -lOpenNI -o
      main
2 $ ./main
```

Nous avons ainsi fini notre premier programme utilisant la Kinect. Cela a permis d'apprendre le fonctionnement de base des librairies OpenNI et OpenCV, la manipulation d'images de profondeur et en couleur et comment suivre la pose d'un être humain, image par image.

5 Conclusion

Avec le lancement de la Kinect, Microsoft a frappé fort en proposant un condensé de technologie pour un prix particulièrement abordable. Le potentiel de la Kinect ne s'arrête pas à certains jeux et de nombreux projets utilisent déjà la Kinect. Cet article détaille comment récupérer les informations de la Kinect et développer une application en langage C utilisant le framework OpenNI. Ce n'est certes pas simple, mais le jeu en vaut la chandelle. En moins d'une heure, vous voilà prêt à développer vos propres applications. Place à votre imagination !

Le code source complet du programme est disponible à l'adresse
http ://www2.ulg.ac.be/telecom/kinect-linuxmag.tar.bz2.

Références

[1] N. Burrus. Rgbdemo. http://labs.manctl.com/rgbdemo/.

[2] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli. Depth mapping using projected patterns, 2010. US Patent Application 20100118123.

[3] K. Konolige and P. Mihelich. Technical description of kinect calibration. http://www.ros.org/wiki/kinect_calibration/technical.

[4] OpenCV. Opencv. http://opencv.willowgarage.com/wiki/.

[5] OpenNI. Home. http://openni.org/.

[6] PCL. Point cloud library. http://pointclouds.org/.

[7] OpenKinect project. Main page. http://openkinect.org/wiki/Main_Page.

[8] Microsoft Research. Contributions to kinect for xbox 360. http://research.microsoft.com/en-us/about/feature/contributionstokinectforxbox360.aspx.

[9] Microsoft Research. Kinectfusion project page. http://research.microsoft.com/en-us/projects/surfacerecon/.