Université du Québec
École de technologie supérieure
Département de génie logiciel et des T.I.

Sujets émergeant en technologie de l'information

Rapport de Laboratoire

Numéro du laboratoire	1
Nom du laboratoire	Capture, traitement et affichage d'images 3D
Étudiant(s)	Simon-Olivier Harel Patrick Lavallée
Code(s) permanent(s)	HARS10068806 LAVP12048408
Numéro d'équipe	
Cours	GTI780
Session	E2017
Groupe	1
Chargé(e) de laboratoire	Louiza Oudni
Date	11-06-2017

Table des matières

Liste des figures

Introduction

Ce laboratoire a pour objectif de familiariser l'étudiant à la composition des images stéréoscopiques et des écrans Dimenco, au travers des fonctionnalités de la Kinect V2 de Microsoft. Le nouveau modèle de la Kinect utilise le système *Time of flight* pour calculer l'information de profondeur lui permettant d'obtenir une précision de profondeur plus élevée.

Une application est développée pour capter l'information provenant des senseurs de la Kinect afin de reconstruire les images pouvant être affichées sur l'écran Dimenco. Les senseurs de couleurs et de profondeurs contiennent cette information. De plus, une image d'en-tête, contenant l'information relative au format d'image supporté par l'écran, doit être insérée dans le coin supérieur gauche de l'image résultante. Et qui plus est, le contenu doit être plein écran et sans bordure. L'alimentation de l'application se fera en *live feed.*

Le présent document présentera d'abord le flot du traitement d'une l'image source jusqu'à sa destination au travers un diagramme *Flowchart*. Ensuite, pour chacune des sections de l'énoncé, une description des détails et justifications des traitements sera fournie. De surcroît, il contient une discussion sur les difficultés rencontrées lors de l'implémentation ainsi que des résultats obtenus.

Flowchart de l’application

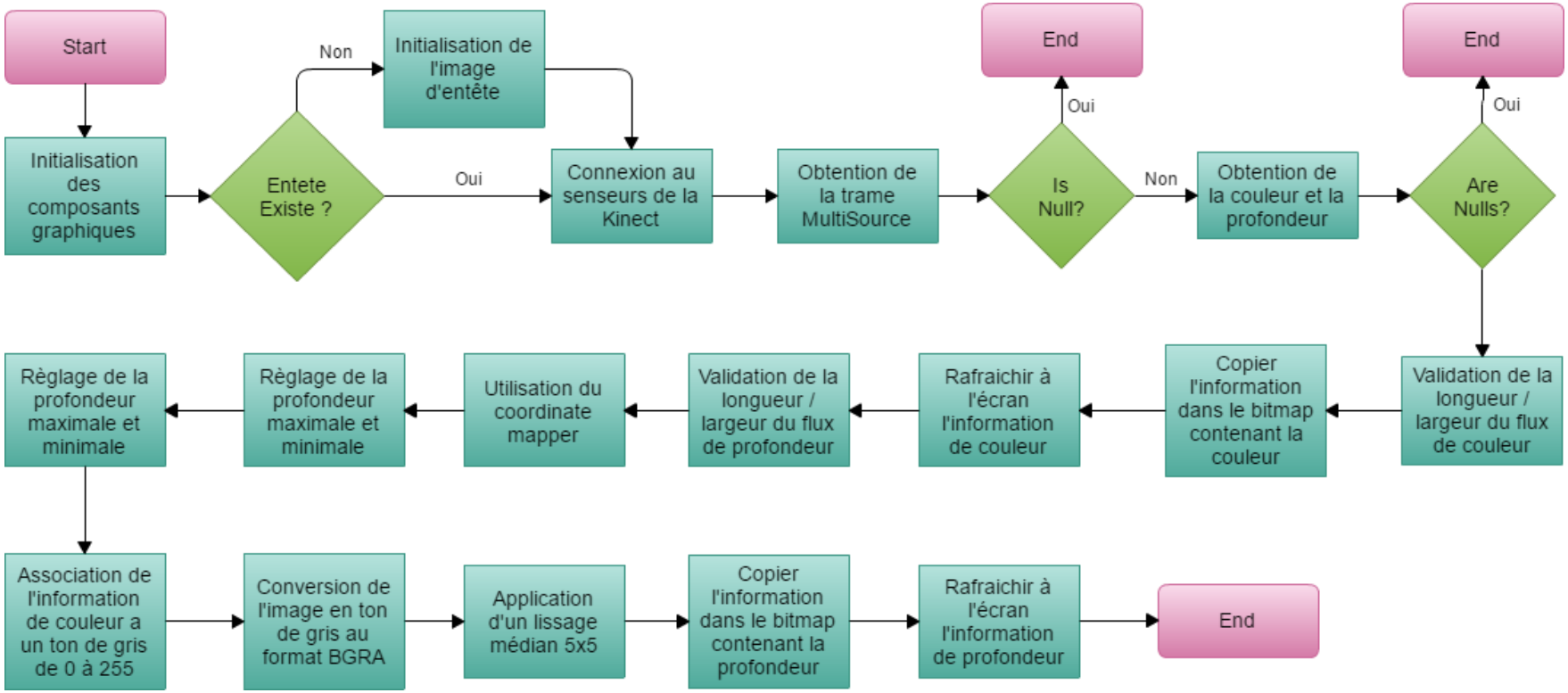

Figure 1 - Aperçu du fonctionnement de l'application

Section I – Connexion aux senseurs de couleur et de profondeur

La solution de base offrait simplement l'information du senseur de couleur, ce même flux était affiché deux fois de part et d'autre de l'écran. Toutefois, l'application de base contenait l'architecture nécessaire pour facilement s'y retrouver. Avec la contrainte de ne pas utiliser la classe *DepthFrameSource*, la classe *MultiSourceFrameReader* a été utilisée. Elle permet d'obtenir l'information nécessaire au travers une seule classe. La variable membre *frameReader* de la classe *MainWindow* a été modifiée en conséquence.

La nouvelle classe n'enregistre pas le même événement qui réagit à l'acquisition du signal. Ci-bas les modifications apportées à l'enregistrement de l'évènement ainsi que la signature de la méthode.

```
// wire handler for frame arrival
this.frameReader.MultiSourceFrameArrived += this.Reader_MultiSourceFrameArrived;
```

Figure 2 - Enregistrement de l'évènement de capture d'image

```
/// <summary>
/// This event will be called whenever the Multi source Frame Reader receives a new frame
/// </summary>
/// <param name="sender">sender</param>
/// <param name="frameArrivedEvent">args of the event</param>
1 reference | Patrick Lavallée, 11 days ago | 1 author, 2 changes
void Reader_MultiSourceFrameArrived(object sender, MultiSourceFrameArrivedEventArgs frameArrivedEvent)
{
```

Figure 3 - Nouvelle signature de méthode

Finalement, lorsqu'une image est reçue par l'application, la source est décortiquée en information de couleur et de profondeur. Une validation s'assure que toute l'information est disponible afin que l'application puisse effectuer son traitement.

```
// Acquire multisource frame containing the color and depth information of the kinect stream
MultiSourceFrame multiSourceFrame = frameArrivedEvent.FrameReference.AcquireFrame();
if (multiSourceFrame == null)
{
    return;
}

// Sanity check on frames instantiation. Can't have one failing.
// Abort if either of the frames are null
ColorFrame colorFrame = multiSourceFrame.ColorFrameReference.AcquireFrame();
DepthFrame depthFrame = multiSourceFrame.DepthFrameReference.AcquireFrame();
if(colorFrame == null || depthFrame == null)
{
    return;
}
```

Figure 4 - Acquisition de la couleur et de la profondeur

Section II - Acquisition et affichage de la couleur

Pour faire suite à la section 1 où il a été démontré comment acquérir un cadre de couleur, cette section focus sur l'affichage de la trame obtenue.

Premièrement, il faut obtenir la description de la trame à l'aide de la classe *FrameDescription*. Elle sera utilisée pour s'assurer que la grosseur de la trame obtenue de la Kinect concorde avec le bitmap utilisé pour afficher la couleur à l'écran.

Le bitmap est d'abord verrouillé aux changements, permettant de travailler librement avec l'objet. Ensuite, les vérifications de grosseur de trame vs grosseur de bitmap sont effectuées. Finalement, l'information de l'objet *ColorFrame* est copiée dans le bitmap d'affichage et l'instance du bitmap est déverrouillée.

```
using (KinectBuffer colorBuffer = colorFrame.LockRawImageBuffer())
{
    // Lock the colorBitmap while we write in it.
    this.colorBitmap.Lock();
    isColorBitmapLocked = true;

    // Check for correct size
    if (colorDescription.Width == this.colorBitmap.Width && colorDescription.Height == this.colorBitmap.Height)
    {
        //write the new color frame data to the display bitmap
        colorFrame.CopyConvertedFrameDataToIntPtr(
            this.colorBitmap.BackBuffer,
            (uint)(colorDescription.Width * colorDescription.Height * BYTESPERPIXELS),
            ColorImageFormat.Bgra);

        // Mark the entire buffer as dirty to refresh the display
        this.colorBitmap.AddDirtyRect(new Int32Rect(0, 0, colorDescription.Width, colorDescription.Height));
    }

    // Unlock the colorBitmap
    this.colorBitmap.Unlock();
    isColorBitmapLocked = false;
}
```

Figure 5 - Affichage de la couleur

Section III – Traitement de la profondeur

Le traitement de l'image de profondeur demande quelques étapes afin d'ordonnancer celle-ci à l'image couleur. La première étape consiste à générer une trame de profondeur alignée à celle de couleur par l'utilisation de la méthode **CoordinateMapper** fournie par l'objet KinectSensor. L'image suivante illustre son utilisation.

```
//------------------------------------------------------------
// Effectuer la correspondance espace Profondeur---Couleur
//------------------------------------------------------------

FrameDescription colorFrameDescription = this.kinectSensor.ColorFrameSource.FrameDescription;
this.kinectSensor.CoordinateMapper.MapColorFrameToDepthSpaceUsingIntPtr(depthBuffer.UnderlyingBuffer,
                depthBuffer.Size, new DepthSpacePoint[colorFrameDescription.Width * colorFrameDescription.Height]);
```

Figure 6: Correspondance couleur profondeur

Afin de spécifier le sens de la correspondance, l'utilisation de la méthode *MapColorFrameToDepthSpaceUsingInPtr* a été sélectionnée. Il aurait été possible d'utiliser d'autres méthodes de correspondance, mais celle-ci a été sélectionnée par le fait que l'image de couleur détient 4 fois plus de pixels que celle de profondeur. Par souci de précision de la correspondance, celle-ci se fera dans le sens de l'image ayant le plus d'informations vers celle ayant le moins d'infirmation.

Après avoir effectué la correspondance, un traitement sur le format de la valeur de la profondeur a été réalisé. Le code suivant illustre un choix important pour le traitement de la profondeur, celui de l'utilisation de la méthode *DeptMaxReliableDistance* (maxDepth) qui offre une distance maximale de 4 500 mm qui viendra influencer notre formule de passage entre la profondeur en millimètres acquise par la Kinect et celle en niveau de gris compatible avec l'écran. Afin d'appliquer cette formule qui sera d'écrite sous peu, la valeur de profondeur (Depth) devra comprise être 0 et 4500 mm afin d'y appliquer la correspondance sinon elle sera affecté de 0.

La formule est la suivante :

$$= \text{Depth/MapDetphToByte}$$

$$= \text{Depth/ } (-1*(\text{ maxDepth / 256 }))$$

$$= -1* (\text{Depth / maxDepth }) * 256$$

$$= -1* (\text{Depth / 4500 }) * 256 ;$$

Où : Depth est compris entre 0 et 4500 mm et maxDepth vaut 4500 mm.

L'image suivante illustre l'application de cette formule dans le code. Il est remarquable que cette formule doit être appliquée à tous les pixels constituant l'image de profondeur.

```
private unsafe void ProcessDepthFrameData(IntPtr depthFrameData, uint depthFrameDataSize, ushort minDepth, ushort maxDepth)
{
    // depth frame data is a 16 bit value
    ushort* frameData = (ushort*)depthFrameData;

    int MapDepthToByte = -1 * (maxDepth / 256);

    // convert depth to a visual representation
    for (int i = 0; i < (int)(depthFrameDataSize / this.depthFrameDescription.BytesPerPixel); ++i)
    {
        // Get the depth for this pixel
        ushort depth = frameData[i];

        // To convert to a byte, we're mapping the depth value to the byte range.
        // Values outside the reliable depth range are mapped to 0 (black).
        this.depthPixels[i] = (byte)(depth >= minDepth && depth <= maxDepth ? (depth / MapDepthToByte) : 0);
    }

    //  Utiliser la ligne ci-dessous pour l'image de profondeur
    Image<Gray, byte> depthImageGray = new Image<Gray, byte>(RAWDEPTHWIDTH, RAWDEPTHHEIGHT);

    //-----------------------------------------------------------
    // Traiter l'image de profondeur
    //-----------------------------------------------------------
    depthImageGray.Bytes = depthPixels;

   // Une fois traitée convertir l'image en Bgra
   depthImageBgra = depthImageGray.Convert<Bgra, byte>();

    //Application d'un filtre
   depthImageBgra = depthImageBgra.SmoothMedian(3);

}
```

Figure 7 : Application de la formule de correspondance profondeur en valeur de gris.

Section IV – Correction du contenu de l'image d'en-tête

Après discussion, le calcul de l'image d'en-tête peut s'effectuer à l'aide de C#. Le laboratoire n'utilise donc pas de programme MatLab. L'application génère elle-même son fichier d'en-tête à chaque exécution seulement si nécessaire. La conception permet de pouvoir générer le type d'en-tête souhaité, mais ne supporte pour l'instant que le format stéréoscopique (Color + 2D Depth).

L'information relative à l'image de l'en-tête est modélisée par la classe *AbstractHeader*. Un patron *Manufacture* est utilisé pour l'instanciation de l'objet d'en-tête. Le contenu du Message H ainsi que le type d'en-tête est fourni à la construction de l'objet. Le message est ensuite automatiquement transformé en une représentation binaire. La transformation du message se fait via la méthode d'extension *HexToBinaryBytes()* de la classe *String* permettant une syntaxe simple à l'utilisation.

```
/// <summary>
/// Instantiate a new Header object
/// </summary>
/// <param name="headerType">The type of the header</param>
/// <param name="H">The header's message to be converted to bitmap</param>
1 reference | Patrick Lavallée, 9 days ago | 1 author, 1 change
public AbstractHeader(HeaderType headerType, string H)
{
    this.HeaderType = headerType;
    this.HeaderMessage = H;

    this._binaryMessage = this.HeaderMessage.HexToBinaryBytes();
}
```

Figure 8- Convertion du message d'en-tête à l'instanciation

Pour les besoins du laboratoire, seulement la classe enfant *StereoscopicHeader* est instantiable. La construction de l'information de l'image se fait au travers la méthode *BuildImageBuffer().* C'est cette méthode qui contiendra la logique de transformation en sous-pixels bleus.

```
protected override byte[] BuildImageBuffer()
{
    var imageBuffer = new byte[BUFFERSIZE];

    // Foreach byte
    for (int byteIndex = 0; byteIndex < this._binaryMessage.Count(); byteIndex++)
    {
        // Foreach bit - Starting with the lightest bit at last index
        var currentByte = this._binaryMessage.ElementAt(byteIndex);
        for (int bitIndex = currentByte.Length - 1; bitIndex >= 0; bitIndex--)
        {
            // Determine which index needs to be written. We start from the lightest bit
            // up to the heaviest one. The first bit to be fed to the H function is the last
            // one but needs to be treated as Index == 0.
            var convertedBitIndex = Math.Abs(bitIndex - (currentByte.Length - 1));
            var bufferIndex = H(byteIndex, convertedBitIndex);

            imageBuffer[bufferIndex] = currentByte[bitIndex].ToMinMaxByteValue();
        }
    }

    return imageBuffer;
}
```

Figure 9 - Construction de l'image d'en-tête

Finalement, une fois l'information de l'image calculée, le fichier peut être créé sur le disque.

```
var imageBuffer = this.BuildImageBuffer();

this.HeaderImage = new Bitmap(
    imageBuffer.Length, // columns
    1,                  // rows
    imageBuffer.Length, // stride
    PixelFormat.Format8bppIndexed,
    Marshal.UnsafeAddrOfPinnedArrayElement(imageBuffer, 0));

this.HeaderImage.Save(Path.Combine(filePath, "EnteteModifiee.bmp"));
```

Figure 10 - Création de l'image d'en-tête sur le disque

L'image d'en-tête est générée à chaque exécution à partir de la méthode InitializeHeader() de la classe *MainWindow.*

```
private void InitializeHeader()
{
    var applicationPath = System.IO.Path.GetDirectoryName(
        System.IO.Path.GetDirectoryName(
            Directory.GetCurrentDirectory()));

    var header = HeaderFactory.Create(HeaderType.Stereoscopic);

    header.EnsureBitmap(applicationPath);

    this.HeaderImage.Source = header.HeaderImage.ToImageSource();
}
```

Figure 11 - Initialisation de l'image d'en-tête

Section V - Améliorer la profondeur obtenue

L'amélioration de la qualité de l'image de profondeur peut être représentée par deux étapes distinctes. Soit une première étape servant à traiter l'image par l'application d'un filtre afin de diminuer le bruit et ensuite d'appliquer une méthode pour redessiner les pixels ayant été discrétisés par l'application du filtre.

Concernant l'application d'un filtre, la librairie emgu cv propose différents filtres tels que le SmoothBlur, SmoothBillateral, SmoothGaussian et le SmoothMedian. Chacun de ces filtres à ses particularités sur le temps de traitement de l'image et sur la qualité du résultat obtenu. Pour ce laboratoire, le filtre appliqué est le SmoothMedian(3). Par souci de rapidité du traitement, le paramètre affecté à cette méthode a été choisi de manière empirique, soit une valeur de 3. Cette valeur signifie que pour un pixel de l'image de profondeur un calcul de la moyenne des valeurs de profondeur sur une fenêtre de 3x3 pixels est calculé et réaffecté au pixel pour lequel les calculs a été effectué.

Concernant l'application d'une méthode pour redessiner les pixels, cette librairie propose une méthode telle que inpaint(). Cette méthode sert à récupérer l'intensité des pixels afin d'améliorer le contour des objets de l'image. Dans cette première étape de ce laboratoire, la détection de contour n'a pas été appliquée. Par contre, l'utilisation de cette méthode demande en paramètre un masque afin d'appliquer les informations de ce masque sur l'image. Suite à quelques recherches pour obtenir une piste de solution, une série d'étapes est proposée dans la section discussion.

Discussion

Architecture logicielle vs Performance

Pour améliorer les performances de l'application l'équipe a décidé de sacrifier une architecture logicielle extensible et de traiter directement dans la méthode main l'information envoyée de la Kinect. La classe *AbstractSourceProcessor* permet de généraliser le concept de source multiple d'information en offrant une méthode *Process* générique. L'instanciation d'un processeur de source était abstraite derrière un patron Fabrique exposant une classe statique experte de la création. Néanmoins, cette approche générait beaucoup d'appels de fonction subséquents ralentissant le processus de traitement séquentiel de l'image. Les classes sont tout de mêmes dans le projet à titre d'artéfacts.

Par ailleurs, la parallélisation du traitement améliorerait grandement la performance de l'application en distribuant la charge sur des *Thread* différents.

Tests unitaires

Une suite de test unitaire est fournie avec la solution. Cette suite cible le processus de création de l'image d'en-tête et compte 12 tests et couvre :

- Traduction du message hexadécimal en binaire.
- Conversion d'un bit à une valeur de 0 ou 255
- Écriture de l'image sur le disque

Amélioration de la profondeur obtenue

Amélioration de la qualité de l'image :

Tel que discuté précédemment dans la section 5, voici une proposition de démarche [1] qui pourrait être explorée afin d'améliorer la qualité du rendu final de l'image 3D :

1. Découper l'image originale à l'aide du filtre gaussien
2. Soustrayez l'image floue de l'original (le résultat est appelé masque) pour éliminer l'arrière-plan et obtenir les régions des bords
3. ajouter une partie pondérée du masque à l'image d'origine en multipliant le Masque (les bords uniquement) par K pour améliorer les régions des bords

Suite à ces étapes, il sera primordial de revenir sur l'étape de l'application d'un filtre ainsi que de valider l'utilisation du filtre SmoothMedian(3).

[1] https://dsp.stackexchange.com/questions/15176/sharpening-an-image-using-emgu

Conclusion

Le laboratoire visait à familiariser les étudiants aux concepts de l'imagerie stéréoscopique. Pour y parvenir, la Kinect V2 de Microsoft était à l'étude. Une trame de couleur et de profondeur a été capturée. Une correspondance entre les pixels de la trame de couleur et ceux de la trame de profondeur a permis d'obtenir des images de même proportion.

L'architecture entourant l'image d'en-tête permet de pouvoir étendre le modèle à d'autre type de téléviseurs.

Il s'agit d'un laboratoire très formateur permettant de comprendre les outils qui se font présentement sur le marché et les méthodes étudier afin d'obtenir un affichage 3D toujours plus riche et complet.