**Captura de imagen básica con Kinect**

El sensor **Kinect** de Microsoft es un dispositivo muy potente que proporciona servicios de captura de imagen, medición de distancias y reconocimiento de posturas corporales y de expresiones faciales, algo que lo hace apropiado para una infinidad de aplicaciones. En este artículo introductorio mostraré como usarlo para capturar diferentes tipos de imagen.

Aunque **Kinect** nació como un sensor para la consola de juegos **Xbox**, existe una versión con adaptador para Windows, por lo que se pueden desarrollar todo tipo de aplicaciones basadas en este sensor en esta plataforma. Para ello, además de disponer del sensor y del adaptador, debes descargarte el SDK de **Kinect** para Windows. Existen varias versiones. La que yo utilizaré aquí es la última en este momento, la [Kinect SDK version 2.0](https://www.microsoft.com/en-us/download/details.aspx?id=44561).

Los ejemplos de código que mostraré utilizan el lenguaje **C#**, aunque también puedes usar si lo prefieres **C++**. Existen tutoriales bastante completos, por ejemplo este es un [tutorial para aplicaciones de la tienda de Windows](http://kinect.github.io/tutorial/). En el SDK de **Kinect**, los ejemplos son aplicaciones **WPF**. En mi caso, para variar, los ejemplos serán aplicaciones Windows Forms. En este enlace puedes [descargar el código fuente de la solución KinectImage](http://software-tecnico-libre.es/Descargas/csharp-dwnld/kinectimage), escrita en csharp con **Visual Studio** **2015**.

Al crear un proyecto .NET que use **Kinect**, debes incluir siempre la referencia al ensamblado **Microsoft.Kinect**, que encontrarás en la pestaña de extensiones, también deberás incluir una instrucción **using** para el espacio de nombres **Microsoft.Kinect**.

La clase que representa al sensor es **KinectSensor**, esta es la clase con la que debe comenzar todo el proceso. Primero, consigue una instancia a ella:

private KinectSensor \_kSensor = null;  
…  
\_kSensor = KinectSensor.GetDefault();

Es posible conectar varios sensores **Kinect** a un mismo equipo. De esta manera solo conseguirás el sensor por defecto, pero de momento no nos vamos a complicar la vida. A partir de esta instancia, podemos suscribirnos al evento **IsAvailableChanged** para que el sensor nos notifique cuando está disponible o deja de estarlo:

\_kSensor.IsAvailableChanged +=   
 new EventHandler<IsAvailableChangedEventArgs>(Sensor\_IsAvailableChanged);

El siguiente paso es poner en marcha el sensor:

\_kSensor.Open();

El sensor nos puede proporcionar lecturas (frames) de varios tipos: **Color**, para imágenes en color, **Infrared**, para imágenes obtenidas con un sensor infrarrojo, que nos permiten ver en la oscuridad, **Depth**, que nos proporciona información sobre la distancia a la que están los objetos, y algunas más de las que hablaré en otros artículos. Para leer cada uno de estos tipos de imagen debemos obtener un objeto reader específico, o bien podemos obtener un objeto multireader que nos permite obtener imágenes sincronizadas de diferentes tipos, este último es el enfoque de mi aplicación ejemplo:

private MultiSourceFrameReader \_fReader = null;  
…  
\_fReader = \_kSensor.OpenMultiSourceFrameReader(  
 FrameSourceTypes.Color |   
 FrameSourceTypes.Depth |   
 FrameSourceTypes.Infrared);

En la llamada indicamos los tipos de imágenes que deseamos recibir. Ahora debemos decidir la forma de obtenerlas. Podemos optar por solicitar una imagen cuando lo creamos conveniente o, la forma más común, suscribirnos al evento **MultiSourceFrameArrived** para ser notificados cada vez que haya datos disponibles.

\_fReader.MultiSourceFrameArrived +=   
 new EventHandler<MultiSourceFrameArrivedEventArgs>(Image\_FrameArrived);

En el controlador del evento extraeremos los diferentes tipos de imagen. Hay que tener en cuenta que no siempre vendrán imágenes de todos los tipos, por lo que hay que controlar los valores nulos:

private void Image\_FrameArrived(object sender,  
 MultiSourceFrameArrivedEventArgs e)  
{  
 if (!\_bProcessing)  
 {  
 try  
 {  
 \_bProcessing = true;  
 MultiSourceFrame frame = e.FrameReference.AcquireFrame();  
 if (frame == null)  
 {  
 return;  
 }  
 using (ColorFrame cframe =   
 frame.ColorFrameReference.AcquireFrame())  
 {  
 DrawColorImage(cframe);  
 }  
 using (InfraredFrame iframe =   
 frame.InfraredFrameReference.AcquireFrame())  
 {  
 DrawInfraredImage(iframe);  
 }  
 using (DepthFrame dframe =   
 frame.DepthFrameReference.AcquireFrame())  
 {  
 DrawDepthImage(dframe);  
 }  
 }  
 catch  
 {  
 }  
 finally  
 {  
 Refresh();  
 Application.DoEvents();  
 \_bProcessing = false;  
 }  
 }  
}

La razón por la que utilizo la variable booleana **\_bProcessing** es para evitar que se acumulen las llamadas y dé la sensación de que la aplicación se queda colgada. No volvemos a procesar una nueva frame hasta que no hayamos mostrado la anterior.

La imagen más sencilla de mostrar es la imagen en color. Simplemente hay que copiar los bytes de la imagen en un **Bitmap** y mostrarlo en pantalla:

private void DrawColorImage(ColorFrame cframe)  
{  
 if (cframe != null)  
 {  
 Bitmap cbmp = null;  
 BitmapData bd = null;  
 try  
 {  
 pbColor.Width = cframe.FrameDescription.Width == 1920 ? 853 : 640;  
 cbmp = new Bitmap(cframe.FrameDescription.Width,   
 cframe.FrameDescription.Height);  
 bd = cbmp.LockBits(  
 new Rectangle(0, 0, cframe.FrameDescription.Width,   
 cframe.FrameDescription.Height), ImageLockMode.ReadWrite,   
 PixelFormat.Format32bppArgb);  
 byte[] image = new byte[4 \* cframe.FrameDescription.Width \*  
 cframe.FrameDescription.Height];  
 if (cframe.RawColorImageFormat == ColorImageFormat.Bgra)  
 {  
 cframe.CopyRawFrameDataToArray(image);  
 }  
 else  
 {  
 cframe.CopyConvertedFrameDataToArray(image,  
 ColorImageFormat.Bgra);  
 }  
 Marshal.Copy(image, 0, bd.Scan0, image.Length);  
 cbmp.UnlockBits(bd);  
 bd = null;  
 pbColor.Image = cbmp;  
 }  
 catch  
 {  
 }  
 finally  
 {  
 if ((cbmp != null) && (bd != null))  
 {  
 cbmp.UnlockBits(bd);  
 }  
 }  
 }  
}

El formato apropiado para las imágenes es **ColorImageFormat.Bgra**. Si ya viene en ese formato, podemos hacer una copia más rápida sin convertir los datos con **CopyRawFrameDataToArray**. No intentéis usar la versión de la función que copia los datos directamente al **IntPtr** del miembro **Scan0** del **BitampData** porque se producirá una excepción (habría que activar el código no seguro, lo cual es innecesario).

La imagen infrarroja nos proporciona para cada píxel un valor que puede estar entre 0 y 65535, y habrá que convertirlo a un color, en este caso en un tono de gris, píxel a píxel. Yo utilizo la primera captura para encontrar un valor mínimo y máximo de píxel para hacer la conversión lo más clara posible:

private void DrawInfraredImage(InfraredFrame iframe)  
{  
 if (iframe != null)  
 {  
 Bitmap ibmp = null;  
 BitmapData bd = null;  
 try  
 {  
 ushort[] idata = new ushort[iframe.FrameDescription.Width \*  
 iframe.FrameDescription.Height];  
 iframe.CopyFrameDataToArray(idata);  
 if (\_maxIR < 0)  
 {  
 for (int ix = 0; ix < idata.Length; ix++)  
 {  
 \_maxIR = Math.Max(\_maxIR, idata[ix]);  
 \_minIR = Math.Min(\_minIR, idata[ix]);  
 }  
 \_scIR = \_maxIR - \_minIR;  
 }  
 else  
 {  
 ibmp = new Bitmap(iframe.FrameDescription.Width,  
 iframe.FrameDescription.Height);  
 bd = ibmp.LockBits(new Rectangle(0, 0,  
 iframe.FrameDescription.Width,  
 iframe.FrameDescription.Height),  
 ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);  
 int[] ipixels = new int[iframe.FrameDescription.Width \*  
 iframe.FrameDescription.Height];  
 for (int ix = 0; ix < idata.Length; ix++)  
 {  
 int pxint = (255 \* (idata[ix] - \_minIR)) / \_scIR;  
 ipixels[ix] = Color.FromArgb(pxint, pxint, pxint).ToArgb();  
 }  
 Marshal.Copy(ipixels, 0, bd.Scan0, ipixels.Length);  
 ibmp.UnlockBits(bd);  
 bd = null;  
 pbInfrared.Image = ibmp;  
 }  
 }  
 catch  
 {  
 }  
 finally  
 {  
 if ((ibmp != null) && (bd != null))  
 {  
 ibmp.UnlockBits(bd);  
 }  
 }  
 }  
}

En cuanto a la imagen basada en distancias, nos proporciona un valor para cada píxel que representa la distancia al sensor en milímetros. Podemos convertir estas distancias en tonos de gris y construir un **Bitmap** con el resultado, también procesando píxel a píxel. En este caso, la menor y mayor distancia nos la proporcionan las propiedades **DepthMinReliableDistance** y **DepthMaxReliableDistance** del objeto **DepthFrame**.

private void DrawDepthImage(DepthFrame dframe)  
{  
 if (dframe != null)  
 {  
 Bitmap dbmp = null;  
 BitmapData bd = null;  
 try  
 {  
 int[] bdepth = new int[dframe.FrameDescription.Width \*  
 dframe.FrameDescription.Height];  
 ushort[] bw = new ushort[dframe.FrameDescription.Width \*  
 dframe.FrameDescription.Height];  
 dframe.CopyFrameDataToArray(bw);  
 ushort dmax = dframe.DepthMaxReliableDistance;  
 ushort dmin = dframe.DepthMinReliableDistance;  
 int rec = dmax – dmin;  
 for (int ix = 0; ix < bw.Length; ix++)  
 {  
 int dpixel = (bw[ix] >= dmin && bw[ix] <= dmax) ?  
 (256 \* (bw[ix] – dmin)) / rec : 0;  
 bdepth[ix] = Color.FromArgb(dpixel, dpixel, dpixel).ToArgb();  
 }  
 dbmp = new Bitmap(dframe.FrameDescription.Width,  
 dframe.FrameDescription.Height);  
 bd = dbmp.LockBits(new Rectangle(0, 0,  
 dframe.FrameDescription.Width, dframe.FrameDescription.Height),  
 ImageLockMode.ReadWrite, PixelFormat.Format32bppArgb);  
 Marshal.Copy(bdepth, 0, bd.Scan0, bdepth.Length);  
 dbmp.UnlockBits(bd);  
 bd = null;  
 pbDepth.Image = dbmp;  
 }  
 catch  
 {  
 }  
 finally  
 {  
 if ((dbmp != null) && (bd != null))  
 {  
 dbmp.UnlockBits(bd);  
 }  
 }  
 }  
}

Y eso es todo lo necesario para construir una aplicación simple basada en **Kinect**, en el ejemplo que os he puesto podéis ver las tres imágenes simultáneamente, y el rendimiento no está nada mal, sobre todo usando un puerto **USB 3.0**, que es lo recomendado. En próximos artículos trataré de algo mucho más interesante que podemos conseguir con este sensor, el reconocimiento de gestos, posturas y movimientos.