Inicialización, creación de nodos y lectura de datos.

```
// Utilizaremos nRetVal para detectar si se han producido errores al
// ejecutar una determinada instrucción. Para ello, la inicializamos
// al estado de "no_errores".
XnStatus nRetVal = XN_STATUS_OK;

xn::Context context;

// Declaramos la variable "context" y llamamos al método Init() para
// inicializar el contexto
nRetVal = context.Init();

// Declaramos y creamos un nodo de producción (en este caso, un nodo
// generador de imagen de profundidad).
xn::DepthGenerator depth;
nRetVal = depth.Create(context);

// Y le indicamos que comience a generar datos.
nRetVal = context.StartGeneratingAll();


// Este bucle while se encargaría de leer la información generada por
// el nodo y procesarla de algún modo.
bool bShouldRun = true;
while (bShouldRun)
{
    // Leemos los datos con un WaitOneUpdateAll, es decir, esperamos a
    // que el nodo especificado ("depth") tenga información nueva
    // disponible, y actualizamos todos los nodos asociados a context.
    nRetVal = context.WaitOneUpdateAll(depth);
    if (nRetVal != XN_STATUS_OK)
    {
        // Error en la lectura.
        printf("Failed          updating          data:          %s\n",
xnGetStatusString(nRetVal));
        continue;
    }

    // Con este método, obtenemos un mapa de profundidad
    const XnDepthPixel* pDepthMap = depth.GetDepthMap();

   // A continuación vendría el código que utilizaría el mapa de
   // profundidad con una finalidad determinada (por ejemplo,
   // almacenar los datos en una imagen OpenCV y mostrarla por
   // pantalla).
   // El bucle terminaría poniendo la variable bShouldRun a false.
}

// Cerramos la aplicación, devolviendo los recursos utilizados.
context.Shutdown();
```

Cadenas de producción

```
// Este trozo de código consulta las cadenas de producción disponibles
// para unas determinadas especificaciones

// Primero creamos un objeto query, para indicar dichas
// especificaciones
xn::Query query;

// Especificamos el fabricante cuyas implementaciones queremos buscar
nRetVal = query.SetVendor("MyVendor");

// Y las capacidades que exigimos a nuestra cadena de producción (en
// este caso, queremos disponer de la capacidad Skeleton)
query.AddSupportedCapability(XN_CAPABILITY_SKELETON);

// Dadas las especificaciones, procedemos a listar las cadenas de
// producción que las satisfacen
xn::NodeInfoList possibleChains;
nRetVal = context.EnumerateProductionTrees(XN_NODE_TYPE_USER, &query,
possibleChains, NULL);

// Si la variable nRetVal no ha generado errores, eso significa que al
// menos hay una cadena de producción disponible que satisface
// nuestros requisitos. Nos quedamos, por ejemplo, con la primera
// cadena encontrada.
xn::NodeInfo selected = *possibleChains.Begin();

// Una vez tenemos la cadena de producción identificada la
creamos/inicializamos.
nRetVal = context.CreateProductionTree(selected);

// Y seleccionamos el nodo generador de usuario (el último en esta
// cadena) para trabajar con los datos que nos ofrezca. Como hemos
// especificado como condición que esté presente la capacidad
// Skeleton, tenemos garantizado que el nodo generador de usuario
// podrá generar datos de usuario y esbozar el esqueleto del usuario
// conforme a dichos datos
xn::UserGenerator userGen;
nRetVal = selected.GetInstance(userGen);

// A continuación, se utilizaría dicha información para los propósitos
de nuestra aplicación.
```

Gestión de mapas/imágenes.

```
// De forma similar a lo visto anteriormente, inicializamos el
// Contexto y un nodo generador de mapas (de profundidad en este
// caso).
XnStatus nRetVal = XN_STATUS_OK;

Context context;
nRetVal = context.Init();

DepthGenerator depth;
nRetVal = depth.Create(context);

// Lo configuramos con resolución VGA y tasa de frames de 30 FPS. Para
// ello, usamos una variable XnMapOutputMode que especifica los datos
// de configuración.
XnMapOutputMode mapMode;
mapMode.nXRes = XN_VGA_X_RES;
mapMode.nYRes = XN_VGA_Y_RES;
mapMode.nFPS = 30;
nRetVal = depth.SetMapOutputMode(mapMode);

// Indicamos al nodo que comience a generar datos
nRetVal = context.StartGeneratingAll();

// Identificamos el índice al píxel central de la imagen VGA. Dado que
// la resolución de la imagen es (XN_VGA_X_RES x XN_VGA_Y_RES), al
// disponer la imagen en forma de array lineal, el punto central será:
XnUInt32 nMiddleIndex =
      XN_VGA_X_RES * XN_VGA_Y_RES/2 + XN_VGA_X_RES/2;

// Pasamos al bucle principal de la aplicación
bShouldRun = true;
while (bShouldRun)
{
      // Actualiza datos en el nodo
      nRetVal = context.WaitOneUpdateAll(depth);

      // Y los extrae en forma de imagen
      const XnDepthPixel* pDepthMap = depth.GetDepthMap();

      // A continuación escribimos por pantalla la distancia
      // correspondiente al píxel central (creando un sistema muy
      // básico para detectar proximidad, por ejemplo, con un robot)
      printf("Middle pixel is %u millimeters away\n",
            pDepthMap[nMiddleIndex]);
}

context.Shutdown();
```

Acceso a los datos de las coordenadas de la mano (Hand Point)

```
// Definimos el gesto que vamos a utilizar para identificar la mano.
// En este caso, "Click" es un gesto en el que "empujamos" con nuestra
// mano en dirección al sensor y la retraemos seguidamente (parecido a
// tocar un botón imaginario que estuviera en el aire)
#define GESTURE_TO_USE "Click"

// Nuestras variables globales ahora son un generador de gestos y un
// generador de coordandas de la mano.
xn::GestureGenerator g_GestureGenerator;
xn::HandsGenerator g_HandsGenerator;

// Esta función callback se activará cuando se haya reconocido un
// gesto.
void XN_CALLBACK_TYPE
Gesture_Recognized(xn::GestureGenerator& generator,
                   const XnChar* strGesture,
                   const XnPoint3D* pIDPosition,
                   const XnPoint3D* pEndPosition, void* pCookie)
{
        // Una vez hemos reconocido un gesto, dejamos de buscarlo con
        // el generador de gestos, y comenzamos a hacer un tracking de
        // la mano en torno a la posición final del gesto detectado.
        printf("Gesture recognized: %s\n", strGesture);
        g_GestureGenerator.RemoveGesture(strGesture);
        g_HandsGenerator.StartTracking(*pEndPosition);
}

// Esta función callback indica qué hacer mientras se está detectando
// un gesto (en principio, no queremos que se haga nada en esta
// aplicación)
void XN_CALLBACK_TYPE
Gesture_Process(xn::GestureGenerator& generator,
                const XnChar* strGesture,
                const XnPoint3D* pPosition,
                XnFloat fProgress,
                void* pCookie)
{}

// Este callback se ejecutará una vez hemos reconocido una mano del
// usuario.
void XN_CALLBACK_TYPE
Hand_Create(xn::HandsGenerator& generator,
            XnUserID nId, const XnPoint3D* pPosition,
            XnFloat fTime, void* pCookie)
{
  printf("New Hand: %d @ (%f,%f,%f)\n", nId,
         pPosition->X, pPosition->Y, pPosition->Z);
}

// Esta función callback se ejecutará cuando se detecte que la mano ha
// cambiado su posición (en nuestro caso, la dejamos vacía)
void XN_CALLBACK_TYPE
Hand_Update(xn::HandsGenerator& generator,
            XnUserID nId, const XnPoint3D* pPosition,
            XnFloat fTime, void* pCookie)
{
}

// Por último, esta función callback indica cuando la mano se ha
```

```
// perdido (ha dejado de hacerse un seguimiento de la misma). En este
// caso, hay que reactivar nuevamente la detección del gesto en el
// generador de gestos, para que pueda volver a detectarse la posición
// de la mano perdida.
void XN_CALLBACK_TYPE
Hand_Destroy(xn::HandsGenerator& generator,
             XnUserID nId, XnFloat fTime,
             void* pCookie)
{
  printf("Lost Hand: %d\n", nId);
  g_GestureGenerator.AddGesture(GESTURE_TO_USE, NULL);
}



void main()
{

  XnStatus nRetVal = XN_STATUS_OK;

  Context context;
  nRetVal = context.Init();

  // Inicializamos los generadores globales
  nRetVal = g_GestureGenerator.Create(context);
  nRetVal = g_HandsGenerator.Create(context);

  // Y registramos las callbacks definidas anteriormente.
  XnCallbackHandle h1, h2;
  g_GestureGenerator.RegisterGestureCallbacks(Gesture_Recognized,
                                              Gesture_Process,
                                              NULL, h1);
  g_HandsGenerator.RegisterHandCallbacks(Hand_Create, Hand_Update,
                                         Hand_Destroy, NULL, h2);

  // Indicamos al contexto que ponga en funcionamiento la generación
  // de datos para todos los nodos
  nRetVal = context.StartGeneratingAll();

  // Y especificamos el gesto que queremos detectar.
  nRetVal = g_GestureGenerator.AddGesture(GESTURE_TO_USE);

  while (TRUE)
  {
    // A partir de aquí, sólo queda actualizar los nodos de producción
    // y los callbacks definidos anteriormente se encargan de mostrar
    // por consola los datos de la posición de la mano.
    nRetVal = context.WaitAndUpdateAll();

  }

  context.Shutdown();
}
```

Acceso a los datos del generador de audio

```
// Inicializamos el contexto y el generador de audio que vamos a
utilizar.
Context context;
nRetVal = context.Init();

AudioGenerator audio;
nRetVal = audio.Create(context);

// Especificamos las características del formato en el que se van a
// producir los datos de audio. Concretamente, están configurados en
// calidad CD (frecuencia de muestreo 44100 muestras/s, estéreo, y 16
// bits por muestra).
XnWaveOutputMode waveMode;
waveMode.nSampleRate = 44100;
waveMode.nChannels = 2;
waveMode.nBitsPerSample = 16;
nRetVal = audio.SetWaveOutputMode(waveMode);

while (true)
{
      // Actualizamos los datos (en este caso, actualizamos cuando hay
      // nuevos datos disponibles en el generador de audio).
      nRetVal = context.WaitOneUpdateAll(audio);


      // Tomamos los datos almacenados en el buffer, y también leemos
      // el tamaño de dichos datos en una variable auxiliar
      const XnUChar* pAudioBuf = audio.GetAudioBuffer();
      XnUInt32 nBufSize = audio.GetDataSize();

      // Aquí se introduciría el código que analizaría estos datos
      // (por ejemplo, podría reproducirlos).
}

// Clean up
context.Shutdown();
```