UNIVERSIDAD POLITÉCNICA DE VALENCIA

INGENIERÍA SUPERIOR DE INFORMÁTICA

Curso académico 2011-2012

Proyecto Fin de Carrera

LIBRERÍAS WRAPPER PARA EL ACCESO A CÁMARAS KINECT Y HERRAMIENTAS PARA EL CALIBRADO DE LAS CÁMARAS RGB Y PROFUNDIDAD

Director: Alberto José Pérez Jiménez

Codirector: Sergio Sáez Barona

Autor: Fernando Palero Molina

2 de Abril del 2012

Agradecimientos

Quiero dar las gracias a Alberto Perez y Sergio Saez por su confianza, sus conocimientos facilitados y su apoyo y paciencia que han sido clave para llevar a cabo este proyecto.

Quiere agradecer también a mi novia por su paciencia y apoyo a lo largo del desarrollo de este proyecto y como no, dar las gracias a mis padres y mi hermana que tanto tiempo han esperado este momento apoyándome con todo lo que han podido a lo largo de la carrera.

Finalmente quiero dar las gracias a todos los amigos y compañeros. Me gustaría hacer una lista con todos los nombres, pero me temo que seria larga y llena de ausencias. Gracias a todos por el apoyo y los buenos ratos que hemos pasado juntos.

ÍNDICE DE CONTENIDO

Introducción

Objetivos y motivación del proyecto.

Tras la aparición de la cámara Kinect la cual permite a los usuarios controlar e interactuar con la consola sin necesidad de tener contacto físico con un controlador de videojuegos tradicional, mediante una interfaz natural de usuario que reconoce gestos, comandos de voz y objetos e imágenes. El dispositivo tiene como objetivo primordial aumentar el uso de la Xbox 360, más allá de la base de jugadores que posee en la actualidad.

Kinect utiliza técnicas de reconocimiento de voz y reconocimiento facial para la identificación automática de los usuarios. La aplicación puede utilizar la funcionalidad de seguimiento Kinect y el sensor de giro motorizado para ajustar la cámara para que el usuario se mantenga en el marco, incluso cuando se mueve.

En noviembre de 2010 se desarrolló un controlador para GNU/Linux que permite el uso de la cámara RGB y las funciones de profundidad.

Los objetivos de este proyecto son realizar un estudio de las diferentes librerías para GNU/LINUX que permiten el uso del dispositivo, el encapsulado de las librerías para simplificar su uso y para trabajar con el formato netpbm.

Utilizando las librerías encapsulada realizaremos una aplicación de calibrado de las cámaras de RGB e infrarrojos mediante un algoritmo de etiquetado semiautomático de las marcas de los patrones de calibración y un algoritmo de calibración.

Estudio sobre las diferentes librerías que existen para la cámara kinect

Las librerías que vamos a estudiar son:

OpenNI (Open Natural Interacción)

Define APIs para escribir aplicaciones utilizando interacción natural. Las APIs de OpenNI están compuestas por un conjunto de interfaces para escribir aplicaciones de interacción natural. El principal objetivo de OpenNI es crear una API estándar que sea capaz de comunicarse con:

- Los sensores de visión y audio ,Dispositivos que ven y oyen las figuras y su alrededor.

- Middleware para la percepción de la visión y el sonido. Los componentes de software que analizan el audio y la información visual que almacenan los datos de la escena y luego los interpretan.

Freenect

Proveé una interfaz asíncrona donde creamos callbacks, las cuales son llamadas por los sensores cuando la información esta disponible. También ofrece la posibilidad de tener una interfaz síncrona donde tenemos una función que pida una imagen de profundidad o de video y devuelva la información.

Esta librería se ha implementado mediante el uso de hilos para manejar los callbacks y un buffer para proveer a la interfaz del cliente. También garantiza la obtención de la información de los buffers ordenadas según el evento más reciente.

Encapsulado de las librerías para simplificar su uso y para trabajar con el formato OpenCV

Encapsulado de librerías

El encapsulamiento de librerías consiste en una capa de código la cual traduce la librería existente, en esta caso la librería de la cámara kinect, en una interfaz compatible, más comprensible o más fácil de usar, esto es útil para:

- Refinar interfaces complicadas o pobres.
- Permitir que el código trabaje junto a otro código que en principio no es compatible, per ejemplo: formatos de datos incompatibles.
- Permitir lenguajes cruzados y/o la interoperabilidad en tiempo de ejecución.

OpenCV

OpenCV es una librería de procesamiento de imagen la cual contiene un **toolkit** gráfico y funciones de reconocimiento. Estas funciones nos permiten interactuar con el sistema operativo, los archivos del sistema y el hardware, todas estas funcionalidades se agrupan en la librería **HighGUI. HighGUI** nos permite abrir ventanas, mostrar imágenes, leer y escribir archivos gráficos y manejar los eventos de teclado y ratón.

Aplicación de calibrado

La aplicación de calibrado se divide en 2 partes:

Etiquetado semiautomático de las marcas de patrones de calibrado.

Se va ha utilizar un algoritmo de etiquetado, el cual etiqueta aquellos píxels activos próximos como pertenecientes al mismo objeto. La entrada al algoritmo es la imagen binarizada, en este caso el patrón de calibrado, que contiene los pixeles activos correspondientes de la imagen, en donde cada etiqueta identifica al objeto al cual dicho pixel pertenece.

El algoritmo se basa en la unión de conjuntos para realizar el etiquetado, es decir, junta objetos etiquetados por separado cuando estos están unidos por al menos un pixel. Una ventaja de este algoritmo es su rápidez y complejidad lineal, pero la desventaja obvia es que objetos que sabemos que son distintos, al estar juntos los toma como uno solo.

Una vez se aplica el algoritmo de etiquetado, puede ocurrir que de los objetos encontrados no pertenecen al patrón de calibrado, la selección los puntos que pertenecen al patrón se realiza de forma manual, trazando un polígono regular sobre la zona donde esta el patrón de calibrado para obtener los objetos deseados.

Cuando se tengan los objetos deseados se procederá a ordenarlos para su uso posterior en el calibrado. El etiquetado se realizara con las imágenes obtenidas con la cámara RGB y con las obtenidas con la cámara de infrarrojos.

Para realizar el algoritmo se va ha utilizar el lenguaje de programación C.

Algoritmo de calibrado.

La calibración de las cámaras consiste en la estimación de los parámetros intrínsecos de las mismas los cuales modelan la geometría interna de la cámara y las características ópticas del sensor. Los parámetros extrínsecos miden la posición y la orientación de la cámara respecto al sistema de coordenadas establecido para la escena. La calibración da la relación del sistema de coordenadas entre las coordenadas de la cámara RGB y las coordenadas de la cámara de infrarrojos.

Para calibrar las cámaras utilizamos una plantilla 2D y obtenemos imágenes de las diferentes cámaras a las cuales se les aplica el algoritmo de etiquetado semiautomático para obtener diversos puntos de la imagen y establecer una relación entre las coordenadas de los puntos entre ambas cámaras.

Estudio de las diferentes librerías para la cámara Kinect

Resumen

En este apartado vamos ha hablar sobre las diferentes librerías que se han propuesto para el proyecto, cuál se ha elegido y cámo instalar la librería.

Librerías

Freenect

Desarrolla un driver para la cámara kinect, coordinando a un gran grupo de personas que trabajan con este nuevo periférico. El driver se llama Freenect, y hasta el momento es capaz de capturar la imagen de la cámara con una resolución VGA (640×480), así como los datos de profundidad, y controlar el pequeño motor en su base. Sin embargo, con kinectComp (interfaz diseñada para la cámara kinect) sólo podemos acceder a los datos de profundidad e imagen.

Una versión experimental de Freenect consigue capturar el audio de sus micrófonos y cambiar el estado del LED frontal. Y trabajan en un sistema llamado Fakenect con el cual puedas grabar una sesión con una kinect, para trabajar posteriormente con esa sesión sin necesidad de tenerla conectada al PC.

OpenNI

¿Qué es OpenNI?

OpenNI (Interacción Natural Abierta) es multi-idioma, multi-plataforma, define un marco para las API donde poder escribir aplicaciones que utilizan Interacción Natural. El API de OpenNI se compone de un conjunto de interfaces para escribir aplicaciones de Interacción Natural. El objetivo principal de OpenNI es formar una API estándar que permite la comunicación con ambos:

- Sensores de visión y audio (los dispositivos que "ven" y "escuchan" las figuras y sus alrededores.)

- Middleware(los componentes de software que analizan los datos de audio y vídeo grabados desde el escenario y lo interpretan) para la percepción de la visión y del audio . Por ejemplo, un software que recibe información visual, como una imagen, devuelve la ubicación de la palma de una mano dentro de la imagen.

OpenNI proporciona un conjunto de APIs para implementar el uso de los sensores, y un conjunto de APIs para el uso de componentes middleware. Con el objetivo de romper las dependencias entre los sensores y el middleware, la API de OpenNI permite a las aplicaciones ser escritas y portables sin ningún esfuerzo adicional para operar con los diferentes módulos middleware. Las APIs de OpenNI también permiten a los

desarrolladores de módulos middleware escribir algoritmos para el tratamiento de datos en bruto, sin tener en cuenta que sensor ha producido los datos y ofrece a los fabricantes de sensores la capacidad de construir sensores compatibles con la librería OpenNI.

El estandar del API de OpenNI permite a los desarrolladores de aplicaciones de interacción natural realizar el seguimiento de escenas 3D mediante la utilización de tipos de datos que son calculados desde la entrada de los sensores. Las aplicaciones pueden ser escritas independientemente de los proveedores de sensores o middleware.

Abstracción de capas

- La capa superior: Representa el software que implementa aplicaciones de interacción natural en la parte superior de OpenNI.

- La capa intermedia: Representa OpenNI, proviene de una interfaz de comunicación que interactua con los sensores y los componentes middleware, que analizan la información de los sensores.

- La capa inferior: Muestra los dispositivos hardware que se encargan de la captura de la imagen y del audio de la escena.

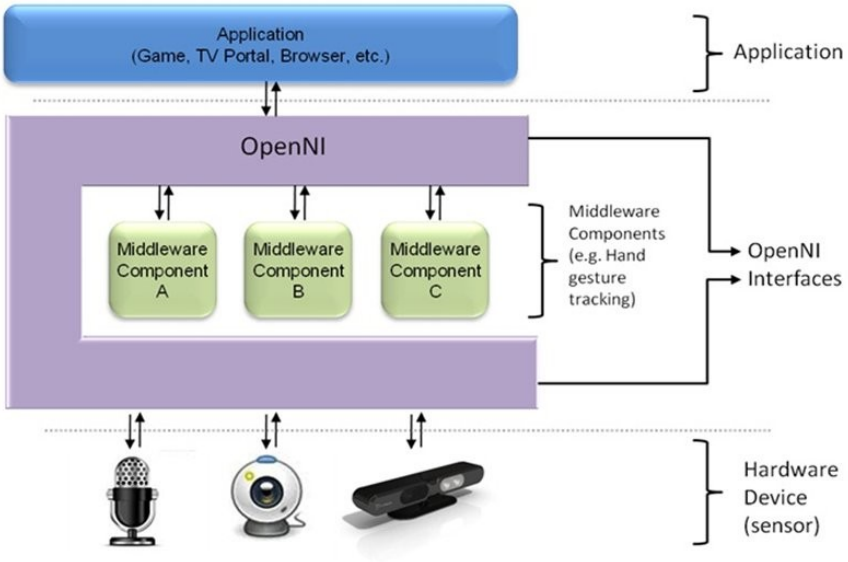


Figura 0

Modulos

El marco de trabajo de OpenNI es una capa abstracta que provee de una interfaz para los dispositivos físicos y componentes middleware. El API permite registrar múltiples componentes en el marco de trabajo. Estos compnenetes son denominados módulos y son usados para producir y procesar los datos de los sensores. Seleccionar los componentes de hardware del dispositivo o middleware requeridos es fácil y flexible.

Los módulos soportados son:

Sensores:

- Sensor 3D
- Camara RGB
- Camara IR
- Dispositivo de audio, un micrófono o una matriz de micrófonos.

Middleware:

- Analizador del cuerpo: es un componente software que procesa los datos obtenidos del sensor y genera información relacionada con el cuerpo.
- Analizador de manos: componente software que procesa los datos obtenidos por los sensores y genera un conjunto de puntos que determinan la localización de las manos.
- **Detección de gestos:** componente software que procesa los datos obtenidos por los sensores y identifica los gestos.
- **Analizador de escena:** componente software que analiza la escena para:
    1. Separa las objetos que pertenecen al primer plano de la escena(las figuras) del fondo.
    2. Las coordenadas del plano del suelo.
    3. La identificación individual de las figuras de la escena.

Comparación de librerías

Para usar una cámara kinect con Freenect, es necesario comunicarse con un componente llamado kinectComp, este componente dispone de una interfaz desde la cual podemos conseguir los datos de imagen y profundidad para poder trabajar con ellos en nuestro componente.

En realidad kinectComp se encarga de controlar la kinect, haciendo de intermediario entre el driver (controlador) de la kinect y los demás componentes.

El driver se está desarrollando desde OpenKinect. El driver se llama Freenect, es capaz de capturar la imagen de la cámara con una resolución VGA (640×480), así como los datos de profundidad, y controlar el pequeño motor en su base. Aunque, con el componente kinectComp sólo podemos acceder a los datos de profundidad e imagen.

Por otra parte OpenNI permite comunicarse con los sensores de audio, video y sensor de profundidad de Kinect, también proporciona una API que sirve de puente entre el hardware del equipo, NITE Middleware y las aplicaciones e interfaces del S.O. La idea es

facilitar el desarrollo de aplicaciones que funcionen con interacción natural, como gestos y movimientos corporales.

Actualmente OpenNI permite la captura de movimiento en tiempo real, el reconocimiento de gestos con las manos, el uso de comandos de voz y utiliza un analizador de escena que detecta y distingue las figuras en primer plano del fondo, Por estas razones vamos a utilizar OpenNI.

Instalación de OpenNI

Paquetes

Los paquetes que se necesitan para poder utilizar las cámaras de infrarrojos, RGB y profundidad son:

OpenNI, es un Framework de código abierto llamado , que permite manejar en el ordenador y en múltiples plataformas el periférico de Microsoft. Se descarga con el programa git. El enlace para descargar es:

Enlace: https://github.com/OpenNI/OpenNI

Pasos para la instalación

Primero debemos instalar el driver de la cámara, el paquete OpenNI, para ello entramos dentro de:

(1) OpenNI/Plantform/Linux-x86/Build

Dentro de la carpeta Build ejecutamos los siguientes comandos:

(2) make && sudo make install

Finalmente instalamos los controladores de lo sensores de la cámara kinect, para ello nos situamos en la carpeta:

(3) Sensor/Plantform/Linux-x86/Build

Nuevamente dentro de la carpeta Build ejecutamos los siguientes comandos:

(4) make && sudo make install

Problemas en la instalación

Si el PC no puede compilar con el flag mss3, debemos modificar el fichero Makefile para indicar que compile con el flag mss2.

Encapsulado de la librería para simplificar su uso

Resumen

El encapsulado de una librería es una colección de subrutinas o clases usadas para el desarrollo de software, ofrecen una capa de código a partir de una interfaz existente que es más sencilla de usar.

En nuestro caso vamos a encapsular la librería OpenNI y la llamaremos librería easykinect que tendrá las funcionalidades de activar y desactivar la cámara kinect y obtener las imágenes de profundidad, infrarrojos y RGB. Para ello vamos a utilizar el lenguaje de programación “C” y el formato OpenCV.

Funciones librería OpenNI

Variables

XnContext: Esta variable es la más importante ya que indica el estado actual del dispositivo o contexto en la aplicación que estamos usando. El contexto debe ser inicializado antes de usar las diferentes funciones de la librería OpenNi. Una vez inicializado se pueden utilizar todas las funciones sin problemas. Una vez finalizada la aplicación debemos liberar la memoria utilizada para el contexto.

XnNodeHandle: Esta variable se utiliza para crear nodos de producción o generadores. Los generadores de encargan de obtener la información de la cámara como por ejemplo las imágenes de profundidad, de infrarrojos, etc.

Los nodos generadores no empiezan inmediatamente a generar los datos, para ello la aplicación debe indicarle al nodo cuando lo debe hacer.

XnStatus: La variable *XnSatatus* almacena el valor devuelto de una función, devolviendo 0 si a ocurrido un error y otro valor en cuando esta todo correcto.

XnMapOutputMode: Con esta variable le indicamos al nodo generador como queremos obtener la información. En el caso de las cámara le podemos indicar el tamaño de la imagen 640x480 y el número de frames por segundo por ejemplo 30.

XnRGB24Pixel: Se almacena en la variable los datos que devuelve en nodo generador de RGB.

XnDepthPixel: Se almacena en la variable los datos que devuelve en nodo generador de profundidad.

XnIRPixel: Se almacena en la variable los datos que devuelve en nodo generador de IR.

Funciones

XnInit

Se encarga de inicializar el contexto para poder utilizar los diferentes generadores más adelante.

```
XnStatus xnInit(XnContext **ppContext)
```

ppContext [salida] Devuelve la posición del puntero que apunta al contexto.

XnStartGeneratingAll

Al llamar esta función indicamos a todos los nodos generadores creados previamente que empiecen a generar datos sino están haciéndolo ya.

```
XnStatus xnStartGeneratingAll(XnContext *pContext);
```

pContext [entrada] Le pasamos el contexto previamente creado.

XnStopGeneratingAll

Esta función se encarga de hacer que los generadores dejen de suministrar información.

```
XnStatus xnStopGeneratingAll(XnContext *pContext);
```

pContext [entrada] Le pasamos el contexto previamente creado.

XnGetStatusString

Nos devuelve una cadena que nos indica el error producido.

```
char * xnGetStatusString(XnStatus Status);
```

Status [entrada] Le pasamos el valor de estado que ha devuelto una función.

XnShutdown

Libera la memoria que se crear al inicializar el contexto y a partir de este momento las llamadas a las funciones de la librería OpenNI ya no pueden ser utilizadas, ya que esto produciría un error.

xnShutdown(XnContext *context);

pContext [entrada] Le pasamos el contexto previamente creado.

XnCreateImageGenerator

Crea un nodo generador que se utilizará para obtener las imágenes de la cámara RGB.

```
XnStatus xnCreateImageGenerator(XnContext *pContext,
                                XnNodeHandle *phImageGenerator,
                                XnNodeQuery *pQuery,
                                XnEnumerationErrors *pErrors)
```

pContext [entrada] Se le pasa el contexto en el cual queremos crear el generador de imagen.

phImageGenerator [salida] El manejador del generador de imagen.

pQuery [entrada] Es opcional. Se puede usar para indicar el tipo de generador de imagen que se quiere crear. Si no se especifica, la función crea el generador de imagen que este disponible.

pErrors [salida] Es opcional. Devuelve una lista de errores.

XnSetMapOutputMode

Esta función a partir de la variable XnMapOutputMode ,la cual se debe inicializar previamente con los parámetros deseados, le indica al nodo generador como debe ser la información de salida que se devuelve.

```
XnStatus xnSetMapOutputMode(XnNodeHandle NodeHandle,
                            XnMapOutputMode * MapOutputMode);
```

NodeHandle [entrada] El manejador que queremos modificar.

MapOutputMode [entrada] El tipo de salida que se ha elegido.

XnWaitOneUpdateAll

Actualiza todos los nodos generadores que contiene el contexto, mientras espera por un nodo especifico para recibir información.

```
XnStatus xnWaitOneUpdateAll(XnContext * pContext,
                            XnNodeHandle hNode)
```

pContext [entrada] Le pasamos el contexto previamente creado.

Hnode [entrada] Le indicamos el nodo al que queremos que espere.

XnGetRGB24ImageMap

Obtiene el mapa actual de la imagen de RGB de 24 bits. El mapa es actualizado después de la llamada a **XnWaitOneUpdateAll.** Se asume que el nodo esta en el formato RGB de 24 bits.

XnRGB24Pixel * xnGetRGB24ImageMap(XnNOdeHandle hInstance)

hInstance [entrada] Le pasamos el manejador que hemos creado para la imagen.

XnCreateDepthGenerator

Crea un nodo generador que se utilizará para obtener las imágenes de profundidad la cámara.

XnStatus xnCreateDepthGenerator(XnContext *pContext,
XnNodeHandle *phImageGenerator,
XnNodeQuery *pQuery,
XnEnumerationErrors *pErrors)

pContext [entrada] Se le pasa el contexto en el cual queremos crear el generador de imagen.

phImageGenerator [salida] El manejador del generador de imagen.

pQuery [entrada] Es opcional. Se puede usar para indicar el tipo de generador de imagen que se quiere crear. Si no se especifica, la función crea el generador de imagen que este disponible.

pErrors [salida] Es opcional. Devuelve una lista de errores.

XnGetDepthMap

Obtiene el mapa actual de la imagen de profundidad de 12 bits. El mapa es actualizado después de la llamada a **XnWaitOneUpdateAll.**

XnDepthPixel * xnGetDepthMap(XnNOdeHandle hInstance)

hInstance [entrada] Le pasamos el manejador que hemos creado para la imagen de profundidad.

XnCreateIRGenerator

Crea un nodo generador que se utilizará para obtener las imágenes de infrarrojos de la cámara.

```
XnStatus xnCreateIRGenerator(XnContext *pContext,
                             XnNodeHandle *phImageGenerator,
                             XnNodeQuery *pQuery,
                             XnEnumerationErrors *pErrors)
```

pContext [entrada] Se le pasa el contexto en el cual queremos crear el generador de imagen.

phImageGenerator [salida] El manejador del generador de imagen.

pQuery [entrada] Es opcional. Se puede usar para indicar el tipo de generador de imagen que se quiere crear. Si no se especifica, la función crea el generador de imagen que este disponible.

pErrors [salida] Es opcional. Devuelve una lista de errores.

XnGetIRMap

Obtiene el mapa actual de la imagen de infrarrojos de 12 bits. El mapa es actualizado después de la llamada a **XnWaitOneUpdateAll.**

```
XnDepthPixel * xnGetIRMap(XnNOdeHandle hInstance)
```

hInstance [entrada] Le pasamos el manejador que hemos creado para la imagen de infrarrojos.

EasyKinect : encapsulado de las funciones OpenNI.

Variables

kinect_t: es una estructura compuesta por:

- ***XnContext *pContext:*** en el apartado anterior se habló de esta variable.
- ***XnNodeHandle Node_Image:*** Utilizamos Nodo_Imagen para obtener las imagenes RGB y de infrarrojos, ya que es el mismo nodo el que las genera.
- ***XnNodeHandle Node_Deep:*** Nodo_Deep se utiliza para obtener las imágenes de profundidad.

Funciones EasyKinect

Publicas

knt_create

Esta función crea la estructura kinect, de esta forma encapsulamos el contenido de la estructura en la librería easykinect haciendo que su uso sea transparente además de solucionar el problema de compatibilidad entre los lenguajes C y C++.

El problema reside en que la librería OpenNI esta desarrollada en C++ y para compilar cualquier programa que utilice las funciones OpenNI debemos utilizar un compilador de C++ y la librería easykinect se quiere utilizar para desarrollar programas en C. Para ello como se ha comentado anteriormente en la librería easykinect encapsulamos la librería OpenNI y debemos declarar en la cabecera donde están las funciones como `extern "C"`.

```
struct  kinect * knt_create(){
  kinect_t *k;
  k = (kinect_t *)malloc(sizeof(kinect_t));
  if(k == NULL)
  {
    perror("No se puede crear la estructura kinect_t.\n");
    exit(-1);
  }
  k->pContext = NULL;
  return k;
} // End knt_create
```

knt_destroy

Se utiliza para liberar la estructura kinect.

```
void knt_destroy( kinect * k)
{
  free(k);
}// End knt_destroy
```

knt_Init

Se utiliza para inicializar los parámetros de la cámara kinect y así poder utilizar la funciones definidas para la obtención de la imagen.

Donde tipo puede ser:

- IMAGE para obtener la imagen RGB.

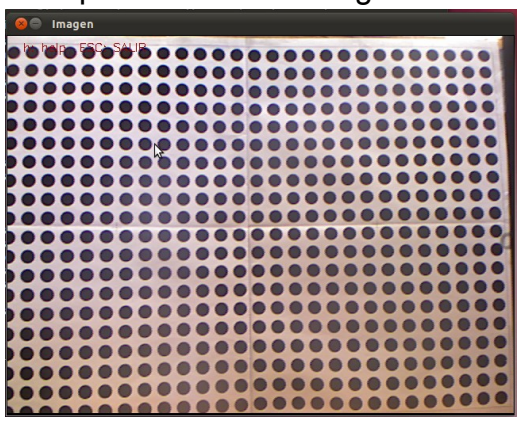


Figura 1

- IR para obtener la imagen de infrarrojos.

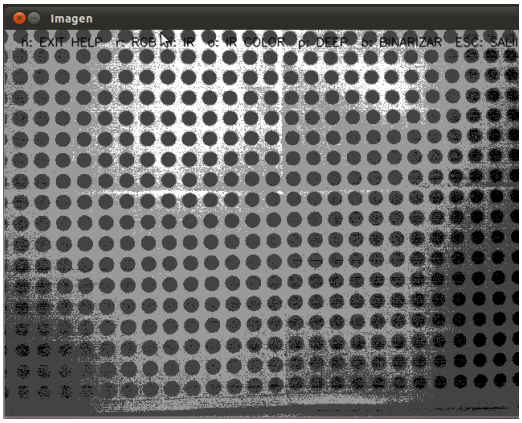


Figura 2

- IR_COLOR para obtener la imagen de infrarrojos en pseudocolor.

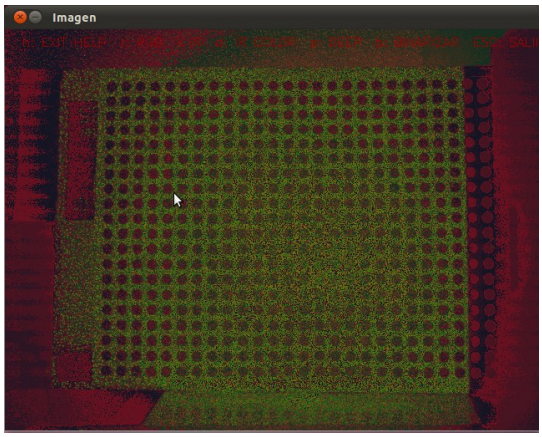


Figura 3

```
void knt_Init(
                kinect *k, // Puntero a la estructura de control de la cámara
                int tipo // Indica el modo de visualización de la cámara
                )
{
 //Declaración de variables
    XnStatus nRetVal;
// Inicializamos la variable nRetVal a verdadero,
// más tarde comprobaremos si el valor ha cambiado, lo cual indica
// que sucedió algún error durante la ejecución de la función knt_Init
    nRetVal = XN_STATUS_OK;
// Con la función xnInit inicializamos el driver OpenNI
    nRetVal = xnInit(&(k->pContext));
// Las funciones knt_create_node_deep y knt_create_node_image
// crean los nodos que se encargan de obtener los datos de las
// diferentes cámaras que luego se transforman en imágenes.
// Aquí creamos un nodo de profundidad y dependiendo del
// valor de 'tipo' creamos un nodo RGB o de infrarrojos.
    k->Node_Deep = knt_create_node_deep( k->pContext);
    k->Node_Image = knt_create_node_image( k->pContext, tipo);
// La función xnStartGeneratingAll pone en funcionamiento los nodos
// creados para empezar a obtener las imágenes de la cámara
    nRetVal = xnStartGeneratingAll(k->pContext);

// Comprobamos que la ejecución haya tenido éxito.
```

```
        if(nRetVal != XN_STATUS_OK)
        {
          printf("Error ejecutando nodo: %s\n",xnGetStatusString(nRetVal));
        }
    }// End knt_Init
```

knt_Stop

Se utiliza para cerrar todos los nodos activos y liberar la memoria utilizada durante la aplicación

```
void knt_Stop(
                kinect *k // Puntero a la estructura de control de la cámara
                )
{
     // Declaración de variables
        XnStatus nRetVal;
   // Inicializamos la variable nRetVal a verdadero,
   // más tarde comprobaremos si el valor ha cambiado, lo cual indica
   // que sucedió algún error durante la ejecución de la función knt_Stop
        nRetVal = XN_STATUS_OK;
    // La función xnStopGeneratingAll hace que los nodos creados para
    // obtener las imágenes paren de generar.
        nRetVal = xnStopGeneratingAll(k->pContext);

    // Comprobamos que la ejecución haya tenido éxito.
        if(nRetVal != XN_STATUS_OK)
        {
         printf("Error parando nodo de IR:
               %s\n",xnGetStatusString(nRetVal));
        }

    // Apagamos el driver OpenNI
         xnShutdown(k->pContext);
    }// End knt_Stop
```

knt_change_view

Se utiliza para ir alternando las diferentes opciones de visualización de la cámara:

Donde tipo puede ser:

- IMAGE para obtener la imagen RGB.
- IR para obtener la imagen de infrarrojos.
- R_COLOR para obtener la imagen de infrarrojos en pseudocolor.

```
void knt_change_view(
                kinect *k, // Puntero a la estructura de control de la cámara
                int tipo // Indica el modo de visualización de la cámara
                        )
{
// Comprobamos si el modo de imagen al que queremos cambiar no
// esta activo. Si no esta activo, cambiamos el modo de imagen
// especificado.
    if((tipo_nodo == IR || tipo_nodo == IR_COLOR) && (tipo == IR || tipo
        == IR_COLOR))
    {
        // Mantenemos el modo de visualización
        tipo_nodo = tipo;
    }else{
        // Cambiamos el tipo de visualización por el modo indicado  en
        // la variable 'tipo'. El valor de la variable 'tipo' se asigna a la
        // variable global 'tipo_nodo', que se encarga de conservar
        // modo de visualización.
        tipo_nodo = tipo;
        knt_Stop(k);
        knt_Init(k, tipo_nodo);
    }
}// End knt_change_view
```

Hay que tener en que no se puede activar a la vez la cámara de infrarrojos (IR) y la RGB debido a que los dos dispositivos utilizan el mismo bus de datos para enviar la información.

knt_get_image

Se utiliza para obtener las imágenes de infrarrojos y de RGB. Dependiendo del valor que se le pase a la función ***knt_change_view()*** se obtendrá un tipo de imagen.

```
IplImage* knt_get_image(
                kinect *k // Puntero a la estructura de control de la cámara
                                )
{
// Dependiendo del valor de la variable global 'tipo_nodo' la función
// devuelve una imagen RGB, de infrarrojos o de infrarrojos pseudocolor
  switch(tipo_nodo)
  {
    case IMAGE:
      return knt_get_image_RGB(k);
      break;
    case IR:
      return knt_get_image_IR(k);
      break;
    case IR_COLOR:
      return knt_get_image_IR_color(k);
      break;
    default:
      printf("Opción incorrecta");
      return NULL;
      break;
  }
}// End knt_get_image
```

knt_get_image_deep

Con esta función obtenemos una imagen de profundidad en escala de grises de 8 bits.

```
IplImage* knt_get_image_deep(
                kinect *k // Puntero a la estructura de control de la cámara
                                )
```

```
{
// Declaramos las variables.
    XnStatus nRetVal;
    IplImage* imgk;
    XnDepthPixel* pDepthMap;
    int i = 0, j = 0;

// Inicializamos la variable nRetVal a verdadero,
// más tarde comprobaremos si el valor ha cambiado, lo cual indica
// que sucedió algún error durante la ejecución de la función
// knt_get_image_deep
    nRetVal = XN_STATUS_OK;
// Creamos una imagen de tamaño de pixel 8 bits y un canal, para
//  almacenar la imagen de la cámara de profundidad
   imgk = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 1);
// Con la función xnWaitOneUpdate, esperamos hasta recibir los datos
// del nodo de profundidad
   nRetVal = xnWaitOneUpdateAll(k->pContext, k->Node_Deep);
// Comprobamos que no haya habido ningún error en la obtención de
// datos
   if(nRetVal != XN_STATUS_OK)
   {
      printf("Error actualizando información de profundidad: %s\n",
            xnGetStatusString(nRetVal));
   }
// La función xnGetDepthMap, nos devuelve el mapa de profundidad
// que obtenemos del nodo de profundidad. La información que se
// obtiene es lista de punteros, donde cada nodo de la lista representa
// el valor de un píxel en formato 12 bits.
    pDepthMap = xnGetDepthMap(k->Node_Deep);
// Pasamos la información que obtenemos del nodo de profundidad a
// una imagen en escala de grises en formato OpenCV.
// Para ello recorremos la lista de píxeles y aplicamos un
// desplazamiento a derecha de 4 bits, por un lado para que coincida
// con el tamaño de píxel de OpenCV y por otro eliminamos los 4 bits
// más bajos ya que es donde más ruido se genera.
```

```
    for( i = 0; i < 480; i++)
      for( j = 0; j < 640; j++)
      {
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j] =
           *pDepthMap>>4;
            ++pDepthMap;
      }
     return imgk;
}// End knt_get_image_deep
```

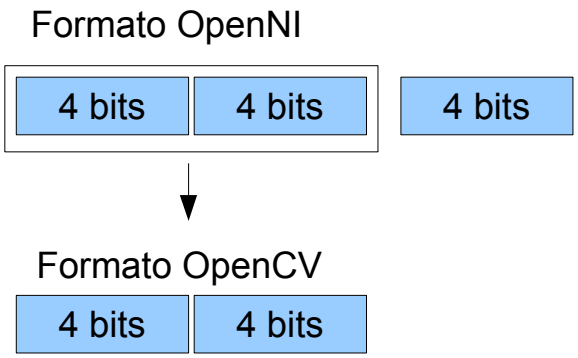


Figura 4

knt_get_image_deep_color

Con esta función obtenemos una imagen de profundidad en pseudo de 24 bits.

```
IplImage* knt_get_image_deep_color(
                kinect *k // Puntero a la estructura de control de la cámara
                )
{
 // Declaración de variables
   XnStatus nRetVal;
   IplImage* imgk;
   XnDepthPixel* pDepthMapColor;
   int i = 0, j = 0;
```

```
// Inicializamos la variable nRetVal a verdadero,
// más tarde comprobaremos si el valor ha cambiado, lo cual indica
// que sucedió algún error durante la ejecución de la función
// knt_get_image_deep_color
    nRetVal = XN_STATUS_OK;
// Creamos una imagen de tamaño de pixel de 24 bits y 3 canales, para
// almacenar la imagen de profundidad de la cámara
  imgk = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 3);

// Con la función xnWaitOneUpdate, esperamos hasta recibir los datos
// del nodo de profundidad
    nRetVal = xnWaitOneUpdateAll(k->pContext, k->Node_Deep);
// Comprobamos que no haya habido ningún error en la obtención de
// datos
    if(nRetVal != XN_STATUS_OK)
    {
      printf("Error actualizando información: %s\n",
            xnGetStatusString(nRetVal));
    }

// La función xnGetDepthMap, nos devuelve el mapa de profundidad
// que obtenemos del nodo de profundidad. La información que se
// obtiene es lista de punteros, donde cada nodo de la lista representa
// el valor de un píxel en formato 12 bits.
     pDepthMapColor = xnGetDepthMap(k->Node_Deep);
// Pasamos la información que obtenemos del nodo de profundidad a
// formato OpenCV, para una imagen RGB.
// Para ello recorremos la lista de píxeles y aplicamos
// un desplazamiento a derecha de 4 bits, así eliminamos los 4 bits
// más bajos ya que es donde más ruido se genera. Ahora con los
// 8 bits que nos quedan los distribuimos en los 3 canales de color.
// Los distribuimos de la siguiente forma:
// 2 bits que se desplazan a la parte alta del color azul, canal 0.
// 3 bits que se desplazan a la parte alta del color verde, canal 1.
// 3 bits que se desplazan a la parte alta del color rojo, canal 2.
// De esta forma obtenemos una imagen en pseudocolor.
```

```
        for( i = 0; i < 480; i++)
          for( j = 0; j < 640; j++)
          {
                *pDepthMapColor=(*pDepthMapColor>>4);
                ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
               >nChannels+0] = (((*pDepthMapColor>>6)&0x02)<<6)|0x2D;
                ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
               >nChannels+1] = (((*pDepthMapColor>>3)&0x03)<<5)|0x1D;
                ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
               >nChannels+2] = (((*pDepthMapColor)&0x03)<<5)|0x1D;
                ++pDepthMapColor;
          }
        return imgk;
    }// End knt_get_image_deep_color
```

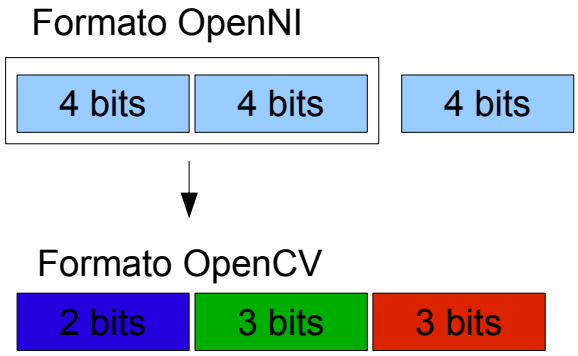


Figura 5

Privadas

knt_create_node_image_RGB

Se utiliza para crear el nodo generador de imagen RGB.

```
    XnNodeHandle knt_create_node_image_RGB(
            XnContext* pContext // Puntero a la variable de contexto del
            )                          // driver OpenNI

    {
    // Declaración de variables
       XnStatus nRetVal;
```

```
    XnNodeHandle NodoImg;
    XnMapOutputMode outputMode;
  // Inicializamos la variable nRetVal a verdadero,
  // más tarde comprobaremos si el valor ha cambiado, lo cual indica
  // que sucedió algún error durante la ejecución de la función knt_Stop
    nRetVal = XN_STATUS_OK;
  // La función xnCreateImageGenerator se encarga de crear el nodo
  // generador de imagen RGB, el cual se encarga de esperar y obtener
  // los datos de la cámara RGB
    nRetVal = xnCreateImageGenerator(pContext, &NodoImg, NULL,
          NULL);
  // Comprobamos si se ha creado el nodo correctamente
    if(nRetVal != XN_STATUS_OK)
    {
      printf("Error creando nodo imagen: %s\n",
          xnGetStatusString(nRetVal));
    }
  // Configuramos el nodo generador indicándole que tamaño de
  // imagen y los frames por segundos tiene la cámara RGB
    outputMode.nXRes = 640;
    outputMode.nYRes = 480;
    outputMode.nFPS = 30;
    nRetVal = xnSetMapOutputMode(NodoImg, &outputMode);
  // Comprobamos si se ha inicializado correctamente
  // los valores de la cámara RGB
   if(nRetVal != XN_STATUS_OK)
   {
     printf("Error configurando nodo imagen: %s\n",
          xnGetStatusString(nRetVal));
   }
    return NodoImg;
  }// End knt_create_node_image
```

knt_create_node_deep

Se utiliza para crear el nodo generador de profundidad.

```
XnNodeHandle knt_create_node_deep(
        XnContext* pContext // Puntero a la variable de contexto del
        )                   // driver OpenNI
{
// Declaración de variables
  XnStatus nRetVal;
   XnNodeHandle NodoImg;
   XnMapOutputMode outputMode;
// Inicializamos la variable nRetVal a verdadero,
// más tarde comprobaremos si el valor ha cambiado, lo cual indica
// que sucedió algún error durante la ejecución de la función
// knt_create_node_deep
   nRetVal = XN_STATUS_OK;
// La función xnCreateDepthGenerator se encarga de crear el nodo
// generador el cual se encarga de esperar y obtener
// los datos de la cámara de profundidad.
   nRetVal = xnCreateDepthGenerator(pContext, &NodoImg, NULL,
        NULL);
// Comprobamos si se ha creado el nodo correctamente
   if(nRetVal != XN_STATUS_OK)
   {
     printf("Error creando nodo imagen de profundidad: %s\n",
        xnGetStatusString(nRetVal));
   }
// Configuramos el nodo generador indicándole que tamaño de
// imagen y los frames por segundos tiene la cámara de profundidad.
   outputMode.nXRes = 640;
   outputMode.nYRes = 480;
   outputMode.nFPS = 30;
   nRetVal = xnSetMapOutputMode(NodoImg, &outputMode);
// Comprobamos si se ha inicializado correctamente
// los valores de la cámara de profundidad
  if(nRetVal != XN_STATUS_OK)
  {
     printf("Error configurando nodo imagen de profundidad: %s\n",
        xnGetStatusString(nRetVal));
```

```
    }
    return NodoImg;
}// End knt_create_node_deep
```

knt_create_node_IR

Se utiliza para crear el nodo generador de imagen de infrarrojos.

```
XnNodeHandle knt_ create_node_IR(
        XnContext* pContext// Puntero a la variable de contexto del
        )                          // driver OpenNI
{
// Declaración de variables
  XnStatus nRetVal;
   XnNodeHandle NodoImg;
   XnMapOutputMode outputMode;
 // Inicializamos la variable nRetVal a verdadero,
 // más tarde comprobaremos si el valor ha cambiado, lo cual indica
 // que sucedió algún error durante la ejecución de la función knt_create_node_IR
  nRetVal = XN_STATUS_OK;
// La función xnCreateIRGenerator se encarga de crear el nodo
// generador el cual se encarga de esperar y obtener
// los datos de la cámara de infrarrojos.
   nRetVal = xnCreateIRGenerator(pContext, &NodoImg, NULL, NULL);
// Comprobamos si se ha creado el nodo correctamente
   if(nRetVal != XN_STATUS_OK)
   {
     printf("Error creando nodo imagen de IR: %s\n",
        xnGetStatusString(nRetVal));
   }
// Configuramos el nodo generador indicándole que tamaño de
// imagen y los frames por segundos tiene la cámara de infrarrojos.
   outputMode.nXRes = 640;
   outputMode.nYRes = 480;
   outputMode.nFPS = 30;
   nRetVal = xnSetMapOutputMode(NodoImg, &outputMode);
  // Comprobamos si se ha inicializado correctamente
```

```
  // los valores de la cámara de profundidad
if(nRetVal != XN_STATUS_OK)
{
     printf("Error configurando nodo imagen de IR: %s\n",
          xnGetStatusString(nRetVal));
   }

   return NodoImg;
}// End knt_create_node_IR
```

knt_create_node_image

Esta función se encarga de crear el tipo de nodo IR o RGB.
Dependiendo de la variable tipo:

- IMAGE crearemos un nodo RGB.
- IR o IR_COLOR creamos un nodo de infrarrojos.

```
XnNodeHandle knt_get_node_image(
          XnContext* pContext, // Puntero a la variable de contexto del
                               // driver OpenNI
          int tipo // Indica el modo de visualización de la cámara
          )
{
// Dependiendo de valor de la variable 'tipo' se creara un tipo de nodo.
  switch(tipo)
  {
    case IMAGE:
// Almacenamos en la variable global 'tipo_nodo' el tipo de nodo que
// vamos a crear
     tipo_nodo=IMAGE;
// Creamos el nodo generador de imagen y devolvemos el manejador
// del nodo
     return knt_create_node_image_RGB( pContext);
     break;
   case IR:
// Almacenamos en la variable global 'tipo_nodo' el tipo de nodo que
// vamos a crear
```

```
        tipo_nodo=IR;
    // Creamos el nodo generador de imagen y devolvemos el manejador
    // del nodo
        return knt_create_node_IR(pContext);
        break;
      case IR_COLOR:
    // Almacenamos en la variable global 'tipo_nodo' el tipo de nodo que
    // vamos a crear
        tipo_nodo=IR_COLOR;
    // Creamos el nodo generador de imagen y devolvemos el manejador
    // del nodo
        return knt_create_node_IR(pContext);
        break;
      default:
        printf("Opción incorrecta");
        return NULL;
        break;
     }
    }// End knt_get_node_image
```

knt_get_image_RGB

Con esta función obtenemos una RGB de 24 bits.

```
    IplImage* knt_get_image_RGB(kinect_t *k)
    {
     // Declaración de variables
        XnStatus nRetVal;
        IplImage* imgk;
        XnRGB24Pixel* pImghMap;
        int i = 0, j = 0;
      // Inicializamos la variable nRetVal a verdadero,
      // más tarde comprobaremos si el valor ha cambiado, lo cual indica
      // que sucedió algún error durante la ejecución de la función knt_get_image_RGB
        nRetVal = XN_STATUS_OK;
    // Creamos una imagen de profundidad de 8 bits y 3 canales (BGR),
    // para almacenar la imagen RGB e la cámara.
        imgk = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 3);
```

```
// Con la función xnWaitOneUpdate, esperamos hasta recibir los datos
// del nodo RGB
    nRetVal = xnWaitOneUpdateAll(k->pContext, k->Node_Image);
// Comprobamos que no haya habido ningún error en la obtención de
// datos
   if(nRetVal != XN_STATUS_OK)
   {
      printf("Error actualizando información: %s\n",
            xnGetStatusString(nRetVal));
    }
// La función xnGetRGB24ImageMap, nos devuelve el mapa de
// color que obtenemos del nodo RGB.
// La información que se obtiene es una lista de punteros, donde cada
// nodo de la lista representa el valor de un píxel en formato 24 bits.
// Para acceder de forma más cómoda a los valores RGB del píxel, se
// pueden acceder a ellos con los punteros nBlue, nGreen, nRed.
    pImghMap = xnGetRGB24ImageMap(k->Node_Image);
// Pasamos la información que obtenemos del nodo RGB a
// formato OpenCV, para una imagen de color RGB.
// Para ello recorremos la lista de píxeles, y asignamos a cada canal el
// color correspondiente. Como en OpenCV los colores se representan
// en formato BGR en vez de RGB, hacemos la siguiente asignación:
// Canal 0 = nBlue.
// Canal 1 = nGreen.
// Canal 2 = nRed.
    for( i = 0; i < 480; i++)
      for( j = 0; j < 640; j++)
      {
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk
           ->nChannels+0] = pImghMap->nBlue;
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk
           ->nChannels+1] = pImghMap->nGreen;
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk
           ->nChannels+2] = pImghMap->nRed;
            ++pImghMap;
      }

    return imgk;
```

```
}// End knt_get_image_RGB
```

knt_get_image_IR

Con esta función obtenemos una imagen de infrarrojos en escala de grises de 8 bits.

```
IplImage* knt_get_image_IR(
        kinect_t *k // Puntero a la estructura de control de la cámara
        )
{
//Declaración de variables
  XnStatus nRetVal;
  int i = 0, j = 0;
  XnIRPixel* pIRMap;
  IplImage* imgk;
 // Inicializamos la variable nRetVal a verdadero,
 // más tarde comprobaremos si el valor ha cambiado, lo cual indica
 // que sucedió algún error durante la ejecución de la función knt_get_image_IR
   nRetVal = XN_STATUS_OK;
 // Creamos una imagen de tamaño de pixel 8 bits y un canal, para
 //  almacenar la imagen de infrarrojos que de la cámara
   imgk = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 1);
// Con la función xnWaitOneUpdate, esperamos hasta recibir los datos
// del nodo de infrarrojos.
    nRetVal = xnWaitOneUpdateAll(k->pContext, k->Node_Image);
  // Comprobamos que no haya habido ningún error en la obtención de
  // datos
   if(nRetVal != XN_STATUS_OK)
   {
     printf("Error actualizando información de IR: %s\n",
          xnGetStatusString(nRetVal));
   }
// La función xnGetIRMap, nos devuelve el mapa de infrarrojos
// que obtenemos del nodo de infrarrojos. La información que se
// obtiene es una lista de punteros, donde cada nodo de la lista
// representa el valor de un píxel en formato 12 bits.
   pIRMap = xnGetIRMap( k->Node_Image);
// Pasamos la información que obtenemos del nodo de infrarrojos a
// una imagen en escala de grises en formato OpenCV.
```

```
// Para ello recorremos la lista de píxels y aplicamos
// un desplazamiento a derecha de 4 bits, por un lado para que coincida
// con el tamaño de píxel de OpenCV y por otro eliminamos los 4 bits
// más bajos ya que es donde más ruido se genera.
    for( i = 0; i < 480; i++)
     for( j = 0; j < 640; j++)
     {
          ((uchar *)(imgk->imageData + i*imgk->widthStep))[j] =
         (*pIRMap)>>4;
          ++pIRMap;
     }
    cvNormalize(imgk, imgk, 255, 0, CV_MINMAX);

    return imgk;
} // End knt_get_image_IR
```

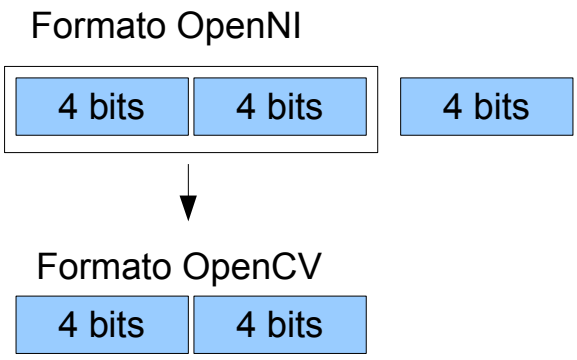


Figura 6

knt_get_image_IR_color

Con esta función obtenemos una imagen de infrarrojos en pseudocolor de 24 bits.

```
IplImage* knt_get_image_IR_color(
          kinect_t *k // Puntero a la estructura de control de la cámara
          )
{
// Declaración de vriables
    XnStatus nRetVal;
    IplImage* imgk;
    XnIRPixel* pIRMapColor;
   int i = 0, j = 0;
```

```
// Inicializamos la variable nRetVal a verdadero,
// más tarde comprobaremos si el valor ha cambiado, lo cual indica
// que sucedió algún error durante la ejecución de la función
// knt_get_image_IR_color
    nRetVal = XN_STATUS_OK;
// Creamos una imagen de tamaño de pixel 24 bits y 3 canales, para
// almacenar la imagen de infrarrojos de la cámara.
    imgk = cvCreateImage(cvSize(640, 480), IPL_DEPTH_8U, 3);

// Con la función xnWaitOneUpdate, esperamos hasta recibir los datos
// del nodo de infrarrojos
    nRetVal = xnWaitOneUpdateAll(k->pContext, k->Node_Image);
// Comprobamos que no haya habido ningún error en la obtención de
// datos
   if(nRetVal != XN_STATUS_OK)
   {
      printf("Error actualizando información: %s\n",
            xnGetStatusString(nRetVal));
    }
// La función xnGetIRMap, nos devuelve el mapa de infrarrojos
// que obtenemos del nodo de infrarrojos. La información que se
// obtiene una lista de punteros, donde cada nodo de la lista representa
// el valor de un píxel en formato 12 bits.
     pIRMapColor = xnGetIRMap(k->Node_Image);
// Pasamos la información que obtenemos del nodo de infrarrojos a
// una imagen RGB en formato OpenCV.
// Para ello recorremos la lista de píxeles y aplicamos
// un desplazamiento a derecha de 4 bits, así eliminamos los 4 bits
// más bajos ya que es donde más ruido se genera. Ahora con los
// 8 bits que nos quedan los distribuimos en los 3 canales de color.
// Los distribuimos de la siguiente forma:
// 2 bits que se desplazan a la parte alta del color azul, canal 0.
// 3 bits que se desplazan a la parte alta del color verde, canal 1.
// 3 bits que se desplazan a la parte alta del color rojo, canal 2.
// De esta forma obtenemos una imagen en pseudocolor.
    for( i = 0; i < 480; i++)
      for( j = 0; j < 640; j++)
```

```
        {
            *pIRMapColor=(*pIRMapColor>>4);
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
           >nChannels+0] = (((*pIRMapColor>>6)&0x02)<<6)|0x2D;
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
           >nChannels+1] = (((*pIRMapColor>>3)&0x03)<<5)|0x1D;
            ((uchar *)(imgk->imageData + i*imgk->widthStep))[j*imgk-
           >nChannels+2] = (((*pIRMapColor)&0x03)<<5)|0x1D;
            ++pIRMapColor;
        }

    return imgk;
}// End knt_get_image_IR_color
```

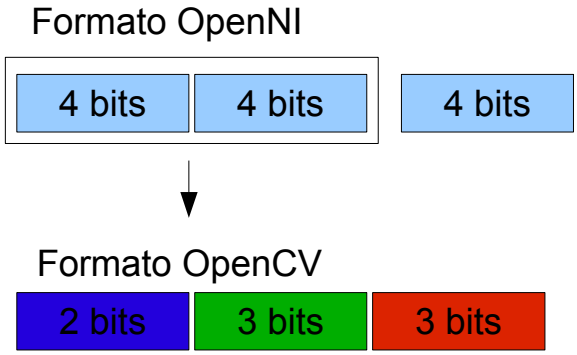


Figura 7

Ejemplo de uso de la librería easykinect.h:

```
int main( int   argc, char *argv[] )
{
  struct kinect  *k;
  IplImage       *img, *deep;
  int            key, end;

// La función knt_Create, se encarga de inicializar la estructura kinect
  k = knt_Create();

// La función knt_Init, se encarga de encender el driver OpenNI y crear
// el nodo de profundidad y en esta caso el nodo RGB
  knt_Init(k, IMAGE);
```

```
  end = FALSE;
  do {
// La función knt_get_image, se encarga de obtener la imagen RGB en
// formato OpenCV
    img = knt_get_image(k);


// La función knt_get_image_deep, se encarga de obtener la imagen de
// profundidad en formato OpenCV
    deep = knt_get_image_deep(k);
// Mostramos las imágenes obtenidas por la cámara
    cvShowImage("Imagen", img);
    cvShowImage("Deep", deep);
    key = cvWaitKey(5) & 0xFF;
    switch (key) {
    case 'q':
// Salimos del programa
      end = TRUE;
      break;
    case 'i':
// Cambiamos el modo de capturar la imagen, ahora pasamos a
// obtener la imagen de infrarrojos
      knt_change_view(k, IR);
      break;
    case 'o':
// Cambiamos el modo de capturar la imagen, ahora pasamos a
// obtener la imagen de infrarrojos en pseudocolor
      knt_change_view(k, IR_COLOR);
      break;
    case 'r':
// Cambiamos el modo de capturar la imagen, ahora pasamos a
// obtener la imagen d RGB
       knt_change_view(k, IMAGE);
      break;
    } // end switch
// Liberamos la memoria de las imágenes
    cvReleaseImage(&img);
    cvReleaseImage(&deep);
  } while (!end);
```

```
// La función knt_Stop se encarga de detener todos los nodos
// generadores creados y de apagar el driver OpenNI
knt_Stop(k);
// La función knt_Destroy se encarga de liberar la memoria reservada
// para la estructura kinect
knt_Destroy(k);

  return 0;

} // end main()
```

Algoritmo de etiquetado semiautomático de las marcas de los patrones de calibración

Resumen

En este capítulo se presentan los conceptos necesarios para el procesamiento digital de imágenes, y una breve descripción de los métodos utilizados para realizar la segmentación del target de calibración.

Un sistema de procesamiento de imágenes puede ser utilizado para desarrollar diferentes tipos de tareas entre las que se encuentran Inspección de procesos, Visión Artificial, Control de Calidad, entre otras. El presente trabajo utiliza las técnicas de procesamiento de imágenes para desarrollar un programa que permita el calibrado de la cámara kinect a partir de imágenes adquiridas de la cámara RGB y de infrarrojo del dispositivo.

En el procesamiento de imágenes, la segmentación es una etapa determinante en el procesamiento de imágenes. El objetivo de esta etapa es aislar los objetos de interés, para luego realizar el análisis de sus características. Existen una gran cantidad de algoritmos para realizar la segmentación de objetos en una imagen, pero su utilización depende de la aplicación específica. Como el objetivo en este caso es encontrar los centros de los círculos del target se va ha utilizar el algoritmo de etiquetado ya que los objetos no se solapan entre si y sus fronteras son fáciles de detectar.

Segmentación y etiquetado

Resumen

Para el segmentado y etiquetado vamos a utilizar el algoritmo de detección de contornos de OpenCV (***cvFindContours()***), para ello previamente vamos ha explicar exactamente que es un contorno. Un contorno es una lista de puntos que representan, por un lado o por otro, una curva en una imagen. Esta representación puede ser diferente dependiendo de las circunstancias del momento. Hay muchas formas de representar una curva. En ***OpenCV*** que es la librería que se va a utilizar, representa los contornos como secuencias en las cuales cada entrada guarda la información acerca de la localización del siguiente punto en la curva. Las secuencias se representa con la estructura ***CvSeq.***

La función ***cvFindContours()*** obtiene los contornos a partir de imágenes binarias. Las cuales se pueden obtener utilizando la algoritmos de umbralizado, donde los bordes son implícitamente las fronteras entre las regiones, en nuestro caso sería un cambio de color de 0 a 255. Una vez obtenemos los contornos los seleccionaremos utilizando las funciones ***cvPointPolygonTest()***, la cual se encarga de comprobar si un punto es dentro de un polígono y la función **cvMatchShapes()**, la cual comprueba que las figuras seleccionadas concuerden con un patrón previamente calculado.

Umbralizado adaptativo

Como el umbralizado global, el umbralizado adaptativo es usado para separar los objetos que pertenecen al fondo de los que no pertenecen al fondo basándose en las diferentes intensidades de los píxeles en cada región. El umbralizado global usa un umbral fijo para todos los píxeles de la imagen y por lo tanto funciona correctamente solo si el histograma de la imagen de entrada contiene picos nítidamente separados que corresponden con los objetos deseados y el fondo. Sin embargo esto no ocurre con todas las imágenes. El umbralizado adaptativo, selecciona un umbral individual para cada píxel basándose en el rango de valores de intensidad de la vecindad del píxel. Esto permite obtener un buen umbralizado para una imagen que su histograma de intensidad no contiene picos distintivos.

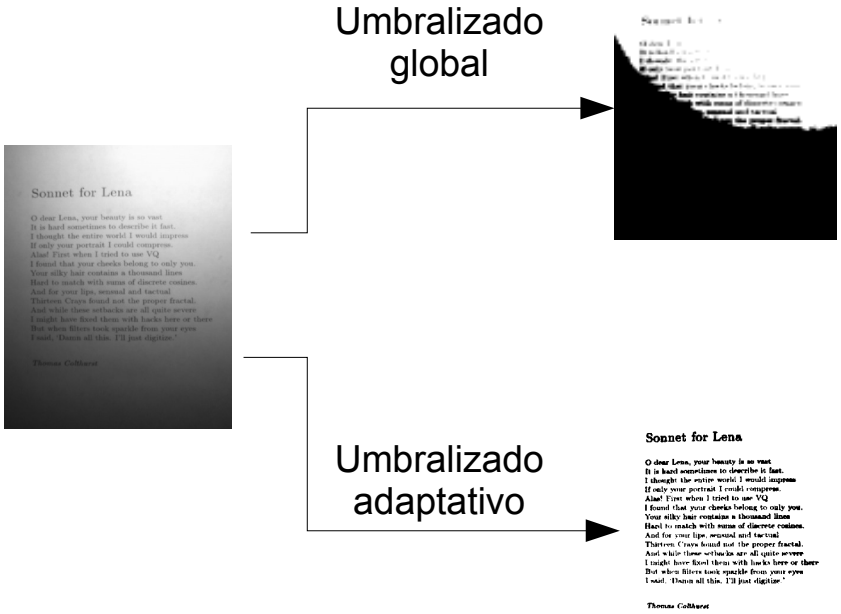


Figura 8: Ejemplo umbralido adaptativo

El umbralizado es usado para segmentar la imagen poniendo todos los píxeles cuyo valor de intensidad este por debajo del umbral se le asigna un valor de fondo por ejemplo 0 o 255 si estamos en escala de grises y los píxeles que son igual o superior al umbral se le asigna un valor que indica que el objeto no forma parte del fondo por ejemplo 1 o 0 si estamos en una escala de grises.

A diferencia de la umbralización convencional dodne el valor del umbral se aplica para todos los píxeles por igual, el umbral adaptativo cambia el umbral de forma dinámico dependiendo las condiciones de luz en la imagen, este cambio se produce como resultado de una fuerte iluminación o sombra en la imagen.

El algoritmo de umbralización adaptativa toma como valor de entrada una imagen en escala de grises y, en implementaciones simples, la salida es una imagen binarizada la cual representa la segmentación. Para cada pixel en la imagen, su umbral es calculado. Si el valor del píxel está por debajo del umbral se establece como valor de fondo, de lo contrario toma el valor de primer plano.

Existen dos métodos para encontrar el umbral:

1. El método de Chow y Kanek
2. El método de umbralización local.

Ambos métodos se basan en la idea de que las regiones pequeñas son más propensas a tener una iluminación uniforme, siendo por tanto más adecuadas para calcular el umbral. Chow y Kaneko divide la imagen en una matriz de sub-imágenes superpuestas y busca el umbral óptimo para cada sub-imagen investigando su histograma. El umbral para cada píxel individual se calcula interpolando los resultados de las sub-imágenes. El inconveniente de este método es que su coste computacional es elevado y no es apropiado para aplicaciones de tiempo real.

Un método alternativo para umbralizar es el umbral local, el cual examina estadísticamente los valores de intensidad de los vecinos de cada píxel. El valor que se obtiene de la estadística depende en gran parte de la imagen de entrada. Se utilizan funciones simples y rápidas tales como la media de la distribución de la intensidad local,

$$T = \text{media} \qquad (1)$$

la mediana,

$$T = \text{mediana} \qquad (2)$$

o la media de el valor mínimo y máximo.

$$T = \frac{max + min}{2} \qquad (3)$$

El tamaño de la vecindad debe ser los bastante grande para abarcar píxeles de fondo y píxeles de primer plano, sino obtendremos un umbralizado pobre. Por otra parte, elegir regiones con un tamaña demasiado grande perderemos la propiedad que tienen las regiones pequeñas de tener una iluminación uniforme. Este enfoque tiene un coste computacional mas bajo que el anterior y produce buenos resultados en determinadas aplicaciones.

Contornos

Como se ha comentado anteriormente, la función ***cvFindCountours()*** es la que se encarga de obtener los contornos a partir de una imagen binarizada, que se ha binarizado utilizando un algoritmo de umbralizado adaptativo ,y a partir de la imagen binarizada se obtienen las fronteras de los objetos que se guardan en la estructura **CvSeq**.

Antes de empezar a explicar como funciona la rutina, es interesante saber que estructura de representación se utiliza, en este caso se utiliza un *árbol de contornos* (los métodos de representación de contornos vienen de Suzuki **[Suzuki85]**), que es importante entender como utilizarlo.

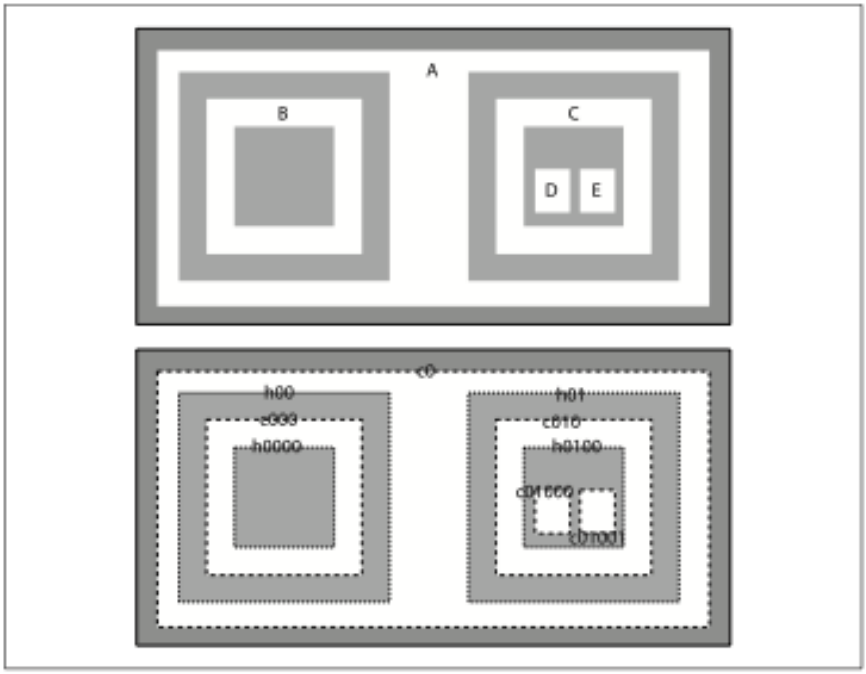


Figura 9: Imagen de test pasada a cvFindContours(); los contornos encontrados son de los dos tipos, contornos exteriores (linea discontinua) o agujeros (linea punteada)

Si observamos la figura 2, la cual representa la funcionalidad de ***cvFindCountours().*** La parte superior de la figura muestra un test de la imagen que contiene un número de regiones en blanco (etiqueta A hasta E) dentro de regiones negras. La parte inferior de la imagen representa la misma imagen pero los contorno han sido obtenidos por ***cvFindContours().*** Estos contornos son enumerados con c_X o h_X, donde "c" son los contornos, "h" representan agujeros y X es un número. Algunos de estos contornos son lineas discontinuas; eso representa las fronteras exteriores de las regiones blancas. OpenCV y ***cvFindContours()*** distingue entre las fronteras exteriores y las interiores que están representadas con lineas punteadas.

El concepto de contención aquí es importante para muchas aplicaciones. Por esta razón, en OpenCV se puede indicar como se construirá el *árbol de contornos* (Los árboles de contornos aparecen por primera vez en Reeb **[Reeb46]** y fue más tarde desarrollado por **[Bajaj97], [Kreveld97]**) con los contornos encontrados, lo cual codificará la relación de contención en la estructura. En el ejemplo de la figura un posible árbol de contorno sería, el contorno c0 el nodo raíz, con los agujeros h00 y h01 como hijos. Estos a su vez tendrían como hijos los contornos que directamente ellos contienen y así sucesivamente.

FindContours

Ahora vamos a explicar la función ***cvFindContours()*** que parámetros necesita y como interpretar los resultados que nos devuelve.

```
Int cvFindContours(
        IplImage*                   img,
        CvMemStorage*               storage,
        CvSeq**                     firtsContour,
        int                 headerSize    = sizeof(CvContour),
        CvContourRetrievalMode      mode          = CV_RETR_LIST,
        CvChainApproxMethod         method        =
        CV_CHAIN_APPROX_SIMPLE
);
```

El primer argumento es la imagen de entrada, debe ser de 8 bit y un solo canal, es decir, una imagen de escala de grises de 8 bits que será interpretada como binaria (por ejemplo, todos los píxeles diferentes de 0 son 1 y el resto 0). Cuando ejecutamos la función, ***cvFindContours()***, usaremos la imagen de entrada como espacio de uso temporal para realizar los cálculos, así que si necesitamos la imagen para un uso posterior se debe realizar una copia y luego pasarla a la función. El siguiente argumento, ***storage***, indica el lugar de la memoria donde se pueden almacenar los contornos. Esta área de almacenamiento debe ser asignada con ***cvCreateMemStorage().*** El siguiente argumento es ***firstContour***, el cual es un puntero a ***CvSeq*.*** La función ***cvFindContours()*** se encarga de asignar el puntero, solo se le debe pasar un puntero a puntero y en ***firstContour*** estará el puntero que apunta a la cabeza del árbol de contorno construido. El valor que devuelve ***cvFindContours()*** será el número total de contornos encontrados.

```
CvSeq* firstContour = NULL;
cvFindContours( ..., &firstContour, ... );
```

HeaderSize simplemente le aporta a ***cvFindContours()*** un poco más de información de los objetos que serán asignados; que se indica como ***sizeof(CvContour)***. Finalmente, tenemos los parámetros ***mode*** y ***method,*** los cuales indican que se va a computar exactamente y como se va a calcular.

La variable ***mode*** (modo) se puede inicializar con cualquiera de las cuatros opciones disponibles en OpenCV: ***CV_RETR_EXTERNAL, CV_RETR_LIST, CV_RETR_CCOMP, o CV_RETR_TREE.*** El valor de ***mode*** indica exactamente que contornos queremos encontrar y como queremos que se muestren. En particular, la manera en la cual las variables del nodo del árbol (***h_prev, h_next, v_prev, v_next***) van a ser “enganchados” se determina mediante el valor de ***mode.*** En la ***firugra 3,*** podemos ver la representación de las diversas tipologías según el valor que tenga ***mode.*** En todos los casos, las estructuras se pueden entender como niveles los cuales relacionados por los enlaces horizontales (***h_next y h_prev***) y estos niveles están separados de uno a otro por

enlaces verticales (***v_next y v_prev***).

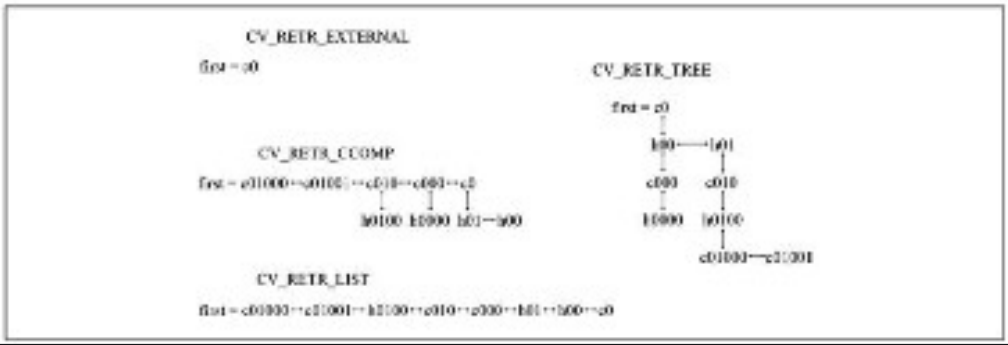


Figura 10: representa las diferentes formas de “enganchar” los nodos del árbol Usando cvFindContours().

Opciones ***mode:***

CV_RETR_EXTERNAL

Recupera solos los extremos exteriores. En la figura 2, en la figura solo hay un contorno exterior, como se puede apreciar en la figura 3 que muestra que los puntos de contorno seleccionados son la secuencia más externa y no tiene más conexiones.

CV_RETR_LIST

Recupera todos los contornos y los pone en una lista. La figura 3 muestra la lista resultante de el test de la imagen de la figura 2. En este caso, se han encontrado 8 contornos y todos ellos están conectados de uno a otro por ***h_prev*** y ***h_next*** (***v_prev*** y ***v_next*** no se usan aquí).

CV_RETR_CCOMP

Recupera todos los contornos y los organiza en 2 niveles jerárquicos, donde el nivel superior son las fronteras externas de los componente y el segundo nivel son las fronteras de los agujeros. Si observamos la figura 3, podemos ver que hay 5 fronteras exteriores, de las cuales tres contienen agujeros. Los agujeros están conectados con sus correspondientes fronteras exteriores por ***v_next*** y ***v_prev***. Las frontera más exterior c0 contiene 2 agujeros. Como ***v_next*** pude contener solo un valor, el solo puede tener un hijo. Para solucionar esto, todos los agujeros dentro de c0 están conectados uno a otro por los punteros ***h_prev*** y ***h_next***.

CV_RETR_TREE

Recupera todos los contornos y reconstruye la jerarquía entera de los contornos anidados. En nuestro ejemplo (Figura 2 y 3), el nodo raíz es el más exterior que en este caso es el contorno c0. Debajo de c0 esta el agujero h00, el cual esta conectado a otro agujero h01 en el mismo nivel. Cada uno de estos agujeros a su vez tienen hijos (los contornos c000 y c010, respectivamente), los cuales están conectados a sus padres por el link vertical. Esto continua hacia abajo hasta los contornos más interiores en la imagen, los cuales son los nodos hoja en el árbol.

Los siguiente cinco valores pertenecen a ***method*** (como los contornos son calculados).

CV_CHAIN_CODE
La contornos de salida están representados mediante códigos de cadenas de Freeman

CV_CHAIN_APPROX_NONE
Convierte todos los puntos de código de cadenas en puntos (X, Y).

CV_CHAIN_APPROX_SIMPLE
Comprime los segmentos horizontales, verticales y diagonales, dejando solo los puntos finales.

CV_CHAIN_APROX_TC89 o CV_CHAIN_APPROX_TC89_KCOS
Aplica uno de los métodos del algoritmo de aproximación de cadenas Teh-Chin.

CV__LINK_RUNS
Este algoritmo es totalmente diferente de los anteriores, donde los enlaces horizontales son segmentos de unos; el único modo de recuperación permitido para este método es **CV_RETR_LIST**.

Para el programa que estamos realizando las opciones que vamos ha utilizar son ***CV_RETR_LIST*** para obtener una lista de contornos, ya que en nuestra aplicación no buscamos diferenciar si es hueco o no, solo queremos que detecte todos los contornos. Lo que tenemos de tener en cuenta es que el primer nodo de la lista lo tenemos que desechar, ya que es el que corresponde al contorno del polígono de selección. El polígono de selección lo creamos con el ratón pinchando sobre el dibujo para rodear la zona de interés, donde se encuentran los objetos del target que queremos utilizar para el calibrado. En cuanto a como queremos que se represente la información utilizaremos el flag ***CV_CHAIN_APPROX_NONE***, ya que nos interesa obtener los puntos del contorno para obtener el centroide, que se obtendrá de la siguiente forma.

```
for (c = ui->contor.first_contour; c != NULL; c = c->h_next) {
  //Inicializamos el valor del centroide
  c->centroide = cvPoint(0, 0);
  // Centroide es la variable de la estructura donde se almacenara el centro
            del contorno

      for (i = 0; i < c->total ; i++) {

   p = CV_GET_SEQ_ELEM( CvPoint, c, i );
  //CV_GET_SEQ_ELEM es una macro que se utiliza para obtener los
      puntos          almacenados en la estructura CvSeq contenida
      en CvContour
```

```
            c->centroide.x += p->x;
                c->centroide.y += p->y;

                }/* End for */
                //Calculamos el punto medio
                c->centroide.x /= c->total;
                c->centroide.y /= c->total;

    }/* End for */
```

Selección semi-automática

Una vez ya se ha realizado la binarización y segmentación de la imagen, seleccionamos los objetos que nos interesan, como el objetivo es seleccionar los elementos del target que hemos preparado para calibrar la cámara, trazamos un rectángulo alrededor de los objetos del patrón de calibrado para obtener los centroides de los objetos, que en este caso son círculos.

Una vez tenemos todos los círculos los ordenamos de izquierda a derecha y de arriba abajo, la ordenación la realizamos con la función ***qsort()*** que utiliza la ordenación QuickSort. La ordenación de los elementos del target es necesario porque debemos indicar al programa de calibrado cual es la correspondencia de los puntos de la imagen respecto los puntos del mundo, para ello relacionamos el primer elemento de la lista ordenada con el punto en el mundo real correspondiente y así sucesivamente para todos los puntos de cada imagen obtenida.

Para realizar la selección semi-automática hemos utilizado las siguientes funciones ***cvPointPolygonTest(), cvMatchShape() y cvDrawContours()*** que a continuación vamos a explicar su uso.

```
double cvPointPolygonTest(
            const CvArr* contour,
            CvPoint2D32f pt,
            int     measure_dist
);
```

PointPolygonTest

La función ***cvPointPolygonTest()*** forma parte del conjunto de herramientas de OpenCV, esta función permite comprobar si un punto esta dentro de un polígono. En particular, se el argumente ***measure_dist*** es diferente de 0 la función devuelve la distancia más cercana a la frontera del polígono; esta distancia sera 0 si el punto de está dentro del contorno y positiva si el punto esta fuera. Si ***measure_dist*** es 0 entonces los valores de retorno son 1, -1 y 0 dependiendo si el punto esta dentro, fuera, o en borde del

polígono, respectivamente. El contorno puede ser una secuencia o una matriz de puntos de dos dimensiones (x, y) de tamaño *N.*

Comparación de contornos

Una de las tareas más comunes asociadas a los contornos es la comparación entre ellos. Nosotros tenemos dos contornos calculados los cuales queremos comparar o comparar un contorno calculado con algún tipo abstracto de plantilla de contornos. OpenCV nos ofreces funciones para ambas posibilidades, aunque nosotros nos vamos solo a centrar en la primera. Comparar dos contornos calculados, lo que vamos ha hacer es pasarle al principio del programa el patrón que queremos utilizar como plantilla y después cuando obtenemos los contornos con la selección semi-automática compararlos con la plantilla calculada al principio del programa. Para ello vamos a utilizar la función ***cvMatchShape().***

MatchShape

La función ***cvMatchShape()*** utiliza los momentos Hu ***[Hu62]*** para comparar patrones. Un momento es un forma muy simple de comparar dos contornos. Los momentos son una importante característica de comparación de contornos que utiliza todos los píxeles del contorno para el calculo. En general, nosotros definimos el (p, q) momento de un contorno como:

$$m_{(p,q)} = \sum_{(i=0)}^{n} \left(I(x,y) x^p y^q \right) \qquad (4)$$

Donde '*p'* es el orden de '*x'* y '*q'* es el orden de *'y'*, en donde *orden* significa la potencia que se aplica a la componente correspondiente en la suma mostrada. El sumatorio se realiza sobre todos los píxeles frontera del contorno (denotados *n* en la ecuación). Aplicando la formula, si *p* y *q* ambos son 0, entonces el momento m_{00} es justamente el tamaño en pixeles del contorno.

El momento que hemos descrito anteriormente describe una característica rudimentaria del contorno que puede ser usada para compara dos contornos. Sin embargo, los momentos resultantes que son calculados no son los mejores parámetros para comparar en algunos casos particulares. Por lo caul, una forma frecuente de uso es la normalización de los momentos, de esta manera objetos con la misma forma pero con tamaños diferentes tienen valores similares. Otro inconveniente de los momentos descritos en el párrafo anterior es que son dependiente del espacio de coordenadas elegido, esto significa que si el objeto es rotado no se realizara una buena comparación.

Como ya hemos comentado anteriormente OpenCV provee de rutinas para el calculo de momentos normalizados, los *momentos invariantes Hu.* Tales como:

*Momento central normalizado*s básicamente el mismo que los momentos descritos anteriormente pero con la excepción que los valores de *x* e *y* usados en la formula son desplazados utilizando el valor de la media.

$$\mu_{(p,q)} = \sum_{(i=0)}^{n} \left(I(x,y)(x - x_{(med)})^{p} (y - y_{(med)})^{q} \right) \qquad (6)$$

donde $x_{med} = m_{10}/m_{00}$ y $y_{med} = m_{01}/m_{00}$.

Los *momentos normalizados* son lo mismo que los momentos centrales a excepción que se dividen por el memento m_{00} elevándolo a una cantidad apropiada (con apropiado se entiende que realizamos una normalización donde el momento sea similar para cualquier tamaño que tenga el objeto):

$$\eta_{(p,q)} = (\mu_{(p,q)}) / (m_{(00)}^{((p+q)/2)+1}) \qquad (7)$$

Finalmente, los *momentos invariantes Hu* son combinaciones lineales de los momentos centrales. La idea aquí esta en que, combinando diferentes momentos centrales normalizados, es posible crear funciones invariantes que representen diferentes aspectos de la imagen que sean invariantes a la escala, la rotación y reflexión.

$$h_1 = \eta_{20} + \eta_{02} \qquad (8)$$

$$h_2 = (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \qquad (9)$$

$$h_3 = (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta 03)^2 \qquad (10)$$

$$h_4 = (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta 03)^2 \qquad (11)$$

$$h_5 = (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})((\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2) + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) \qquad (12)$$

$$h_6 = (\eta_{20} - \eta_{02})((\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \qquad (13)$$

$$h_7 = (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) - (\eta_{30} - 3\eta_{12})(\eta_{21} + \eta_{03})(3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2) \qquad (14)$$

Si observamos la figura 4, podemos hacernos una idea de como funcionan los momentos Hu. Observamos que los momentos tienden a ser pequeños cuando nosotros nos movemos a valores grandes. Esto no es raro ya que, por definición los momentos Hu de orden elevado según la formula de normalización, la potencia que se usa para normalizar también es más grande. Como cada uno de estos factores es menor que 1, el producto de cantidades elevadas tiende a ser pequeño.

Figura 11: Imágenes de cinco caracteres

	H1	H2	H3	H4	H5	H6	H7
A	2,84E-001	1,96E-003	1,48E-002	2,27E-004	-4,15E-007	1,00E-005	-7,94E-009
I	4,58E-001	1,82E-001	0,00E+000	0,00E+000	0,00E+000	0,00E+000	0,00E+000
O	3,79E-001	2,62E-004	4,50E-007	5,86E-007	1,53E-013	7,78E-009	-2,59E-013
M	3,47E-001	4,78E-004	7,26E-005	2,62E-006	-3,61E-011	-5,72E-008	-7,22E-024
F	3,19E-001	2,91E-002	9,40E-003	8,22E-004	3,87E-008	2,02E-005	2,29E-006

Tabla 1: valores de los momentos Hu para cinco simples caracteres de la figura 11

Como podemos observar en la tabla la letra “I”, la cual es simétrica 180º tanto rotándola como en reflexión, tiene un valor exacto a 0 desde H3 hasta H7; y la “O”, la cual tiene una simetría similar, tiene todos los valores de todos los momentos diferentes de 0. Por lo que se puede entender que los valores de H3 hasta H7 siete representan momentos donde la figura es rotada o se le aplica reflexión y H1, H2 representan valores donde la figura se desplaza en cualquier dirección.

```
double cvMatchShapes(
   const void* object1,
   const void* object2,
   int     method,
   double       parameter = 0
);
```

OpenCV utiliza la función ***cvMatchShapes()*** la cual hace más fácil la comparación de momentos Hu. Para utilizar esta función se les deben suministrar dos objetos en escala de colores de grises o dos contornos, nosotros en nuestro programa pasaremos dos contornos a la rutina. La variable ***method*** se utiliza para indicar una de las 3 posibilidades de comparación que ofrece ***cvMatchShapes()***.

Tabla 2: Los diferentes métodos usados por cvMatchShapes()

Valores de method	CvMatchShapes() valores devueltos
CV_CONTOURS_MATCH_I1	$I_1(A,B)=\sum_{i=1}^{7}\left\|1/m_i^A-1/m_i^B\right\|$
CV_CONTOURS_MATCH_I2	$I_2(A,B)=\sum_{i=1}^{7}\left\|m_i^A-m_i^B\right\|$
CV_CONTOURS_MATCH_I3	$I_3(A,B)=\sum_{i=1}^{7}\left\|(m_i^A-m_i^B)/m_i^A\right\|$

En la tabla vemos m_i^A y m_i^B son definidas como:

$$m_i^A=sign(h_i^A)*\log|h_i^A|$$
$$m_i^A=sign(h_i^B)*\log|h_i^B|$$

donde h_i^A y h_i^B son los momentos Hu de A y B, respectivamente.

Cada una de las tres constantes definas en la tabla 2 tienen diferentes significados según lo que se quiera comparar. El método elegido determina el valor devuelto de ***cvMatchShapes().***

En el caso del programa que se ha realizado, se han probado los diferentes métodos y el que ofrece mejor resultado es ***CV_CONTOURS_MATCH_I2.***

Al programa de calibrado se le pasa el patrón que queremos utilizar para calibrar, en este caso usamos un círculo. Una vez cargada la imagen del patrón, obtenemos su contorno y se le pasa a la función ***cvMatchShapes()***, la cual realiza la comparación.

Calibrado de la cámara

Resumen

Al observar la imagen resultante por la pantalla se aprecia claramente que la imagen de la cámara RGB y la información de profundidad que nos da el sistema infrarrojo no coinciden. Se puede apreciar claramente que el error se incrementaba con la distancia.

Para arreglar el problema nos damos cuenta que existe una traslación entre la imagen de la cámara RGB y de la imagen captada por la cámara IR debido a la separación que hay entre ambas. Por otro lado hay que tener en cuenta que el emisor de infrarrojos está desplazado hacia la derecha respecto a la cámara IR, lo que generará una “sombra de infrarrojos” a la izquierda de cualquier objeto.

La traslación entre los ejes focales de las dos cámaras (IR y RGB), se puede calcular a groso modo midiendo la distancia entre las dos cámaras, que nos la podría dar el fabricante. Sin embargo, hay que tener en cuenta que la precisión en la fase de construcción de la Kinect no es perfecta, y por este motivo cada Kinect tiene una traslación específica.

Por este motivo existe la necesidad de calibrar cada kinect con la que queramos trabajar.

Para el calibrado de cada cámara se han estudiado dos programas de calibrado el programa de calibrado diseñado por el departamento del DISCA que utiliza el algoritmo de calibrado DLT y las funciones de calibrado de la librería OpenCV.

El modelo Pinhole de cámara

Ahora vamos a presentar el modelo matemático del calibrado. Veremos como se combina junto la técnica DLT para representar y resolver el problema de calibrado.

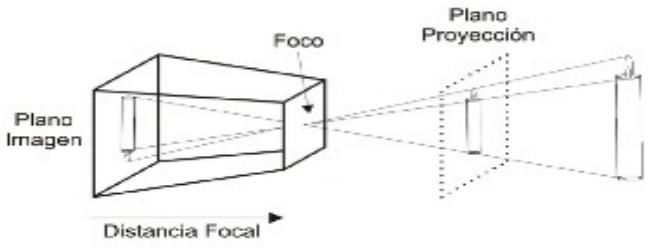
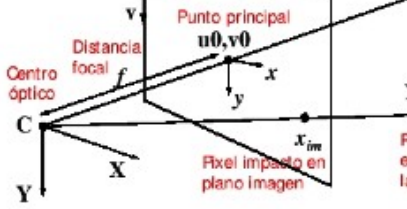


Figura 12: Modelo cámara oscura

El modelo de cámara en el cual se basa DLT es ***Pinhole***. Asume la intuición de que todos los rayos atraviesan una caja por un agujero (foco de la cámara) para impactar en el otro lado de la caja (plano imagen) (ver figura 5). El comportamiento de las lentes según este modelo es lineal. Sin embargo las lentes reales tienen distorsiones radiales que proviene de la fabricación y que hacen que el comportamiento de dicha lente no sea



lineal. De ahí la necesidad de añadir cierta correcciones a este modelo para acercarlo lo más próximo al comportamiento real de una cámara.

En este modelo, el sistema de referencia de la cámara se sitúa en el centro de la proyección, haciendo que el eje Z sea el eje óptico de la cámara de tal manera que el plano de la imagen se sitúa perpendicular al eje óptico a una distancia igual a la distancia focal de la cámara. La intersección del eje principal con el plano de la imagen se denomina punto principal (Ver figura 6).

Normalmente, el plano de la imagen se sitúa delante del punto de proyección C que se supone constante. De esta manera obtenemos una imagen sin inversión. El modelo ***Pinhole*** sirve para modelar lentes delgadas ya que estas no desvían mucho el rayo que las atraviesa. Sin embargo en el caso de las lentes gruesas este modelo no resulta adecuado.

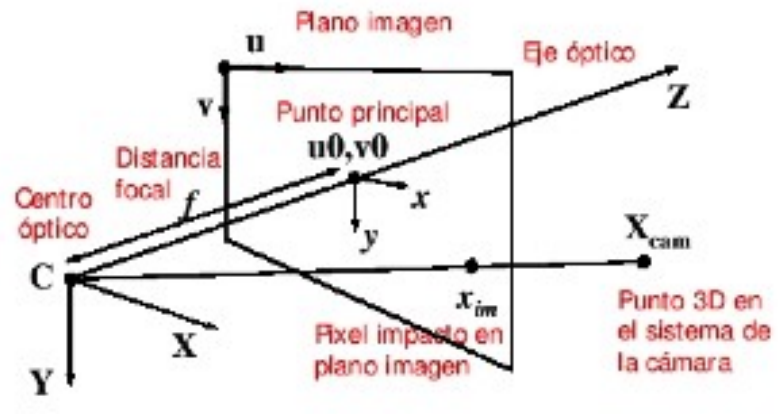


Figura 13: Esquema del modelo Pinhole

Dada una cámara cualquiera, ésta se caracteriza con dos tipos de parámetros:

- ***Parámetros intrínsecos:*** Dependen del modelo utilizado para representar la cámara. En el modelo *Pinhole* son:
  - f_x**:** distancia focal multiplicada por el factor de tamaño de píxeles en el eje X, S_x.
  - f_y**:** distancia focal multiplicado por el factor de tamaño en píxeles en el eje Y, S_y.
  - (U_0, V_0)**:** punto principal.
- ***Parámetros extrínsecos:*** Estos representan la posición y orientación de la cámara en el mundo. En general se representan con dos matrices genéricas, una de rotación y otra de translación (RT). Pero en el caso de rotación hay varias maneras de representarla (cuatriones, ángulos de Euler, foco de atenuación + rol, etc).

El problema de calibrado consiste en hallar los parámetros intrínsecos y extrínsecos de la cámara.

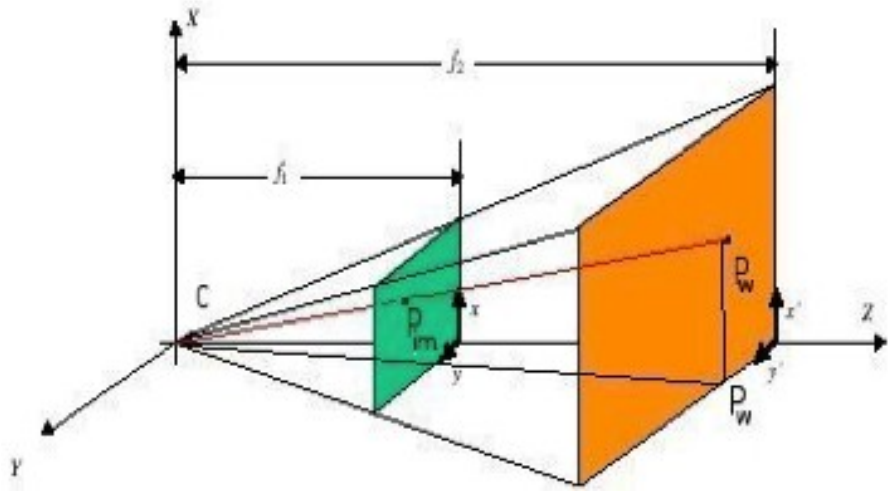


Figura 14: Representación de la cámara

Como podemos observar en la figura anterior el triángulo (C, P_w, P_w), el punto P_w^{cam} expresado en el sistema de coordenadas de la cámara con coordenadas *[X,Y,Z]* se proyecta en un punto del plano de la imagen P_{im} de coordenadas (x,y). Aplicando el teorema de *Tales* en este triángulo obtenemos:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z} * \begin{bmatrix} X \\ Y \end{bmatrix} \qquad (13)$$

El siguiente paso es convertir el punto (x,y) en (u,v) que vienen a ser los píxeles correspondientes en el sensor de la imagen. Para ello, tenemos que saber el tamaño de los píxeles en horizontal y vertical. La conversión se hace utilizando las siguientes ecuaciones:

$$u = S_x * x + U_0 \qquad (14)$$

$$v = S_y * y + V_0 \qquad (15)$$

Donde (U_0, V_0) son las coordenadas del punto principal en píxeles.

La correspondencia de un punto 3D P_w^{cam} a otro 3D P_{im} no es única. Dado un punto P_{im} (u, v) todos los puntos que pertenecen a la recta que une el centro de proyección C con el P_w^{cam} y P_{im} impactan en el mismo punto P_{im}.

Utilizando coordenadas homogéneas, la ecuación 13 se puede expresar:

$$\lambda\begin{bmatrix}u\\v\\1\end{bmatrix}=k*\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}_{w}^{cam} \qquad (15)$$

$$\lambda\begin{bmatrix}u\\v\\1\end{bmatrix}=\begin{bmatrix}f_x & 0 & U_0 & 0\\0 & f_y & V_0 & 0\\0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}_{w}^{cam} \qquad (16)$$

El punto P_w^{cam} está expresado en el sistema de coordenadas de la cámara. En la vida real las coordenadas vienen expresadas respecto a otro sistema de referencia absoluto que no tiene porqué ser el de la cámara. Dado un punto P_w^{absl} expresado en un sistema de referencia en el universo, para hallar el punto en la imagen P_{im} correspondiente a este punto lo primero que debemos hacer es expresarlo en el mismo sistema de referencia de la cámara. Sólo entonces podemos aplicar la ecuación 16 para hallar el punto P_{im}. De modo genérico, para pasar del sistema de coordenadas absoluto al sistema de referencia de la cámara, tenemos que aplicar una rotación y una traslación (alguna de ellas podría ser nula). Las matrices correspondientes a este cambio de base se denominan matriz de rotación y traslación intrínseca, Rt_{ext}.

Este cambio de coordenadas se puede expresar de forma matricial en coordenadas homogéneas con la siguiente ecuación:

$$P_w^{cam}=R*T*P_w^{absl} \qquad (17)$$

$$\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}_{cam}=\begin{bmatrix}r_{11} & r_{12} & r_{13} & t_x\\r_{21} & r_{22} & r_{23} & t_y\\r_{31} & r_{32} & r_{33} & t_z\\0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}_{w} \qquad (18)$$

Recapitulando lo anterior, para calcular el punto P_{im} correspondiente a un punto P_w^{absl} cualquiera tenemos que hacer los siguientes pasos:

- Trasladar P_w^{absl} al punto P_w^{cam} expresado en el sistema de referencia de la cámara utilizando la ecuación 18
- Proyectar el punto P_w^{cam} sobre la imagen utilizando la ecuación 16

Combinando estas dos ecuaciones obtenemos la ecuación general para proyectar cualquier punto 3D del universo sobre el plano de la imagen: la ecuación 20. En la vida real esto no siempre es posible, pues el sensor de la cámara es de tamaño limitado y habrá ciertos puntos que salen del campo de visión.

$$P_{\Im} = K * R * T * P_w^{absl} \qquad (19)$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_{cam} = \begin{bmatrix} f_x & 0 & U_0 & 0 \\ 0 & f_y & V_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w \qquad (20)$$

Calibrado DLT

Después de haber visto los elementos que compone el problema de la calibración desde el punto de vista matemático, llega el momento de estudiar la solución basada en DLT.

Matriz genérica de proyección

La idea detrás de esta técnica es estudiar el paso de 3D a 2D de una cámara de tal manera que dado un patrón de calibrado del cual se conoce con antelación la posición de ciertos puntos que pertenecen al mismo, estudiar la correspondencia entre estos puntos 3D y los correspondiente en 2D una vez capturada una imagen del patrón con la cámara.

Dicho esto, el diagrama de entrada y salida de DLT sería:

Figura 15: entrada y salida del calibrador

Si reescribimos la ecuación 20 para dejar sólo una matriz como incógnita: La *matriz genérica de proyección*, ésta tendría el siguiente aspecto:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ h_{31} & h_{32} & h_{33} & h_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w \qquad (21)$$

Cálculo de la matriz genérica de proyección

El primer paso hacia la solución es calcular las once incógnitas de la matriz M ya que h_{34} se puede fijar a un valor constante. Siguiendo un razonamiento similar al del rectificador harían falta como mínimo seis emparejamientos para resolver todos los elementos de la matriz ya que cada punto proporciona dos ecuaciones de la siguiente

manera:

$$u=(h_{11}*x+h_{12}*y+h_{13}*z+h_{14})/(h_{31}*x+h_{32}*y+h_{33}*z+1) \qquad (22)$$

$$u=(h_{21}*x+h_{22}*y+h_{23}*z+h_{24})/(h_{31}*x+h_{32}*y+h_{33}*z+1) \qquad (23)$$

Dicho estos, nuestro patrón de calibración debería contener al menos seis puntos para hallar la matriz de calibración. Las posiciones 3D de estos puntos constituyen el conocimiento a priori del calibrador. Se le deben introducir las posiciones 2D de tal manera que cada punto 3D tiene un punto 2D como pareja que viene a ser la proyección del mismo sobre la imagen.

Con seis puntos podremos formular doce ecuaciones para resolver once incógnitas. Se trata de un sistema de ecuaciones lineales *sobredimensionado* ya que disponemos de un número mayor de ecuaciones que de incógnitas por resolver. La solución en este caso no sería exacta ya que cada subconjunto de once ecuaciones posibles daría una solución distinta pero parecida a la que daría otro subconjunto. Nuestro objetivo es hallar la solución que menos error comete a la hora de pasar 3D a 2D. Esta tarea se conoce como optimización de un sistema sobredimensionado.

El problema equivalente trabajando con una sola dimensión, es buscar una línea que pase por un conjunto de puntos que no están alineados. Evidentemente encontrar la línea exacta que pasa por todos los puntos es imposible. Lo que se puede hacer es hallar la línea que menor distancia tiene respecto todos ellos. En este caso en vez de una línea tenemos que buscar una matriz, y en vez de puntos tenemos varios conjuntos de puntos 2D proyectados. Se trata de buscar la matriz que cometa el mínimo error a la hora de proyectar los puntos 3D a sus correspondientes en 2D respecto a sus posiciones originales que se saben a priori.

Una vez construido el sistema de ecuaciones *sibredomensionado* lo resolvemos. El resultado son once incógnitas que forman la matriz M (ver ecuación 21). Esta manera de solucionar el sistema de ecuaciones nos abre el abanico para utilizar más puntos en el patrón con lo que la solución obtenida podrá ser más precisa.

Descomposición RQ

El siguiente paso es descomponer la matriz genérica de proyección, a partir de ahora M, en *KR [I | - C]* donde K es la matriz de intrínsecos, R es la matriz de rotación y T la de traslación.

Para ellos nos apoyamos en la descomposición RQ. Una variante de QR. La matriz K es una matriz de 3x3 triangular superior, R es de 3x3 una matriz de rotación ortogonal y T una matriz de traslación de 3x1. El algoritmo de descomposición consiste en los siguiente pasos.

Dada la matriz M_{3x4} se puede ver como *M = [N | p4]* si aplicamos la QR a la inversa de N el resultado es:

$N^{-1} = QS$
$(N^{-1})^{-1} = (QS)^{-1}$
$N = S^{-1}Q^{-1}$
$N = KR$

Donde $K = S^{-1}$ y $R = Q^{-1}$ (La inversa de una matriz ortogonal es otra matriz ortogonal y la inversa de una matriz superior triangular es otra de las mismas características)

La descomposición QR no es única. Para forzar la unicidad ponemos restricción de signo positivo sobre la distancia focal de tal manera que si f_x es negativa entonces invertimos el signo de la primera columna de K y la fila correspondiente de R. Lo mismo para f_y en caso de que sea negativa.

El siguiente paso, siempre dentro de la descomposición de la matriz M, es obtener la posición 3D del foco de la cámara. Con el último resultados tenemos.

$$M = KRT = KR [I | - C] \quad (24)$$

Sabemos que M = [N | p4] de ahí:

$$M = N [I | N^{-1}p4] \quad (25)$$

Comparando las dos últimas ecuaciones:

$$C = N^{-1} * p4 \quad (26)$$

Con esto quedan determinados los parámetros de la cámara:

$$M = KRT \quad (27)$$

Las matrices R y T representan la rotación y la posición de la cámara respectivamente respecto al marco de referencia asociado al patrón de calibrado. R es una matriz de rotación genérica y T representa la posición 3D de la cámara.

Calibrado OpenCV

Ahora vamos a explicar el segundo método estudiado para el calibrado de cámara que también se basa en la cámara Pinhole.

Proyección geométrica básica

La relación entre los puntos Qi (Puntos del mundo real) en el mundo físico con coordenadas (X_i, Y_i, Z_i) a puntos proyectados en la pantalla con coordenadas (x_i, y_i) se llama transformación de proyección. Cuando trabajamos con dicha proyección, es

conveniente usar coordenadas homogéneas. Las coordenadas homogéneas asociadas a un punto en el espacio de proyección de dimensión *n* son normalmente expresadas como un vector de (n+1) dimensiones (ejemplo, x, y ,z se transforman en x, y, z ,w), con la restricción añadida que dos puntos cualquiera con valores proporcionales son equivalentes. En nuestro caso, el plano de la imagen es el espacio de proyección y tiene dos dimensiones, así nosotros podremos representar puntos en este plano como vectores de tres dimensiones q = (q_1, q_2, q_3). Recordando que todos los puntos que tienen valores proporcionales en el espacio de proyección son equivalentes, nosotros podemos obtener las coordenadas del píxel actual dividiendo por q_3. Estos nos permite disponer de los parámetros que definen nuestra cámara (por ejemplo, f_x, f_y, c_x y c_y) en una simple matriz de 3x3, la cual llamaremos *matriz intrínseca de la cámara* (las aproximaciones que utiliza OpenCV para obtener los parámetros intrínsecos de la cámara derivan de Heikkila y Silven **[Heikkila97]**). La proyección de los puntos en el mundo físico en el de la cámara se resume por la siguiente forma:

$$q = MQ, \qquad donde \qquad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad (28)$$

El resultado de esta multiplicación nos indica que w = Z y como *q* esta en coordenadas homogéneas, nosotros podemos dividir por w o Z pata recuperar la definición anterior.

Pinhole es un modelo ideal, el cual es muy útil para la geometría tridimensional de la visión. Recordando, sin embargo, que la cámara Pinhole recibe poca luz; así que en la práctica la generación de la imagen es muy lenta ya que debemos esperar a tener suficiente luz acumulada para obtenerla. Para conseguir obtener las imágenes más rápido, se puede acumular más luz sobre una área grande y curvada en la cual converge la luz en el punto de proyección. Para conseguirlo se una lente. Una lente puede focalizar una gran cantidad de luz para conseguir la imagen más rápidamente, pero en consecuencia se introduce una distorsión que hay que eliminar. Esta distorsión la eliminan las funciones de calibrado de OpenCV.

Calibración

Ahora que hemos hablado de como se obtienen de forma matemática los parámetros intrínsecos de la cámara, lo siguiente que vamos a explicar es como calibrar con OpenCV para calcular la matriz intrínseca y el vector de distorsión.

La función que hemos utilizada es ***cvCalibrateCamera2()***. Esta rutina, el método de calibración consiste en captar los puntos de una estructura conocida la cual tiene muchos puntos fáciles de identificar individualmente. Los puntos deben ser capturados desde diferentes distancias o ángulos, en nuestro caso desde diferentes distancias, así es posible calcula la posición y orientación de la cámara en cada imagen obtenida y también los parámetros intrínsecos de la cámara.

¿ Cuántos puntos e imágenes son necesarios ?

En OpenCV hay cuatro *parámetros intrínsecos* (f_x, f_y, c_x, c_y) y cinco de *distorsión*: tres radiales (k_1, k_2, k_3) y dos tangenciales (p_1, p_2). Los parámetros intrínsecos son directamente relacionados a la geometría 3D y los parámetros de distorsión que se relacionan con la geometría 2D, nos dan las restricciones del problema de calibración. Con tres puntos de un patrón conocido conseguimos seis datos, en principio esta información es suficiente para resolver los cinco parámetros de distorsión, por supuesto cuanta más información más robusto será el resultado. De esta forma con solo una imagen del patrón es suficiente para obtener los parámetros de distorsión. Por otro lado, para los parámetros extrínsecos, son necesarios tres parámetros de rotación y tres de traslación, es decir, un total de seis parámetros. Junto con los cuatros intrínsecos tenemos un total de diez parámetros a resolver en cada vista.

Entonces si tenemos N puntos y K imágenes en diferentes posiciones. ¿Cuantas vista y puntos serán necesarios para resolver todos los parámetros?

- Con K imágenes del patrón tendríamos 2NK variables, ya que cada punto en la imagen consta de una 'x' y una 'y'.
- Ignorando los parámetros de distorsión por el momento, tenemos 4 parámetros intrínsecos 6K parámetros extrínsecos, ya que necesitamos 6 parámetros de localización en el patrón por imagen.
- Finalmente tenemos $2\mathrm{NK} \geq 6\mathrm{K}+4$ o $(N-3)K \geq 2$.

Esto significa que si N = 5 entonces nosotros necesitaríamos solo K= 1 imagen, pero K debe ser mayor de 1. La razón por la cual K debe ser mayor a 1 es debido a que el algoritmo de calibrado calcula matriz homográfica y para ello necesita más de una imagen. Con esta nueva restricción, con K imágenes con cuatros puntos por imagen sería suficiente (4-3)K > 1, estos nos da que K > 1. En la práctica para obtener un buen resultado con diez imágenes y ochenta puntos por imagen es suficiente.

¿Cómo se calcula el calibrado?

Hay muchas formas de obtener los parámetros de la cámara, en OpenCV se utiliza un método que funciona muy bien en objetos planares. El algoritmo de OpenCV utiliza para calcular lo distancia focal y el centro óptico basado en el método de Zhang **[Zhang00]** y para calcular los parámetros de distorsión utiliza el método basado en Brown **[Brown71]**

Para empezar la explicación vamos a suponer que la cámara no tiene distorsión mientras resolvemos los otros parámetros de calibración. Por cada imagen del patrón, obtenemos una matriz de homografía, la cual se representa mediante vectores columnas, $H = [\, h_1\ h_2\ h_3\,]$, donde cada $h_{1,2,3}$ es un vector de 3x1. Entonces, la matriz H es igual a la matriz intrínseca M de la cámara mediante la combinación de dos matrices columnas de rotación, r_1 y r_2, y el vector de traslación t; después de incluir el facto de escala s, nos queda:

$$H = \left[h_1\, h_2\, h_3 \right] = sM \left[r_1\, r_2\, t \right] \qquad (29)$$

Resolviendo las ecuaciones, obtenemos:

$$h_1 = sMr_1 \quad o \quad r_1 = \lambda M^{-1} h_1 \qquad (30)$$
$$h_2 = sMr_2 \quad o \quad r_2 = \lambda M^{-1} h_2 \qquad (31)$$
$$h_3 = sMr_3 \quad o \quad r_3 = \lambda M^{-1} h_3 \qquad (32)$$

Donde, $\lambda = \frac{1}{s}$.

Los vectores de rotación son ortogonales, además r_1 y r_2 son ortonormales. Ser vector ortonormal implica dos cosas:

- El producto escalar del vector de rotación es 0.
- Las magnitudes de los vectores son iguales.

Empezando por el producto escalar, tenemos que:

$$r_1^T \cdot r_2 = 0 \qquad (33)$$

Para cualquier vector 'a' y 'b' tenemos tenemos $(a\,b)^T = a^T b^T$, así podemos sustituir por r_1 y r_2 y obtenemos la primera restricción:

$$h_1^T M^{-T} M^{-1} h_2 = 0 \qquad (34)$$

Donde A^{-T} es lo mismo que $(A^{-1})^T$. También sabemos que las magnitudes de los vectores de rotación son iguales:

$$\|r_1\| = \|r_2\| \quad o \quad r_1^T r_1 = r_2^T r_2 \qquad (35)$$

Substituyendo por r_1 y r_2 obtenemos la segunda restricción:

$$h_1^T M^{-T} M^{-1} h_1 = h_2^T M^{-T} M^{-1} h_2 \qquad (36)$$

Con estas dos restricciones obtenemos:

$$B = M^{-T} M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix} \qquad (37)$$

Sustituyendo se obtiene:

$$B=\begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & (\frac{c_x^2}{f_x^2})+(\frac{c_y^2}{f_y^2})+1 \end{bmatrix} \qquad (38)$$

Usando la matriz B, ambas restricciones tienen la forma general de $h_i^T B h_j$ cada una. El resultado de multiplicar las restricciones es la matriz B. Como B es una matriz simétrica, puede ser escrita como un vector escalar de seis componente. Organizando los elementos necesarios de B en el nuevo vector *b*, el resultado es:

$$h_i^T B h_j = v_t^{ij} b = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2}+h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1}+h_{i1}h_{j3} \\ h_{i3}h_{j2}+h_{i2}h_{j3} \\ h_{i3}h_{j3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix} \qquad (39)$$

Usando la definición de v_{ij}^T , nuestras dos restricciones se pueden escribir como:

$$\begin{bmatrix} v_{12}^T \\ (v_{11}-v_{22})^T \end{bmatrix} b = 0 \qquad (40)$$

Si nosotros obtenemos K imágenes del patrón, entonces podemos apilar K de estas ecuaciones juntas:

$$Vb = 0 \qquad (41)$$

Donde *V* es una matriz de 2Kx6. Como antes, si $K \geq 2$ entonces la ecuación puede resolverse con b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T. Los parámetros intrínsecos son:

$$f_x = \sqrt{\frac{\lambda}{B_{11}}} \qquad (42)$$

$$f_y = \sqrt{\frac{(\lambda B_{11})}{(B_{11}B_{22}-B_{12}^2)}} \qquad (43)$$

$$c_x = \frac{(-B_{13}f_x^2)}{\lambda} \qquad (44)$$

$$c_y = \frac{(B_{12}B_{13}-B_{11}B_{23})}{(B_{11}B_{22}-B_{12}^2)} \qquad (45)$$

Donde:

$$\lambda = \frac{(B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23})))}{B_{11}} \qquad (46)$$

Por otro lado los parámetros extrínsecos (rotación y traslación) se calculan a partir de las ecuaciones que derivan de las condiciones de homografia:

$$r_1 = \lambda M^{-1} h_1 \qquad (47)$$
$$r_2 = \lambda M^{-1} h_2 \qquad (48)$$
$$r_3 = r_1 x r_2 \qquad (49)$$
$$r_3 = \lambda M^{-1} h_3 \qquad (50)$$

Aquí el parámetro de escalar esta definido utilizando la condición de ortonormalidad $\lambda = \frac{1}{\left\| M^{-1} h_1 \right\|}$.

Debemos ir con cuidado cuando usamos datos reales y ponemos los vectores de rotación juntos (R = [r_1 r_2 r_3]), ya que no obtendremos una matriz de rotación exacta para la que $R^T R = R R^T = I$.

Para solucionar este problema, un truco que se usa normalmente es tomar la descomposición de valor singular (SVD) de R. SVD es un método de factorización de una matriz a dos matrices ortonormales, U y V y una matriz intermedia D con un valor de escala en la diagonal. Esto nos permite transformar R en $R = UDV^T$. Como R por si misma es ortonormal, la matriz D debe ser la matriz identidad I así que $R = UIV^T$. Por lo tanto se puede realizar el calculo para que R se transforme en una matriz de rotación R a partir de su valor singular de descomposición, ajustando la matriz D para que sea la identidad y multiplicando SVD con los nuevos valores, conformando la matriz de rotación R'.

A pesar de todo este trabajo, aun nos falta tratar el tema de la distorsión de las lentes. Para ello usamos los valores intrínsecos de la cámara encontrados previamente, también debemos poner los parámetros de distorsión a 0 para empezar a resolver el sistema de ecuaciones.

Los puntos que percibimos de la imagen esta realmente en lugar equivocado debido a la distorsión. Siendo (x_p, y_p) los puntos de la cámara pinhole perfecta y (x_d, y_d) los puntos distorsionados.

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} (f_x X^W)/(Z^W + c_x) \\ (f_y X^W)/(Z^W + c_y) \end{bmatrix} \qquad (51)$$

Una vez obtenidos los parámetros de calibración sin distorsión aplicamos la siguiente

substitución:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2\mathrm{p}_1 x_d y_d + p_2 (r^2 + 2 x_d^2) \\ p_1 (r^2 + 2\mathrm{y}_d^2) + 2\mathrm{p}\mathrm{y}_d^2 x_d y_d \end{bmatrix} \qquad (52)$$

Aplicando la formula anterior obtenemos los parámetros de distorsión.

CalibrateCamera2

Finalmente vamos ha hablar de la función que nos ofrece OpenCV para el calibrado de la cámara. La rutina nos devuelve la *matriz intrínseca de la cámara, los coeficientes de distorsión, los vectores de rotación y los vectores de traslación (uno por imagen).* Los dos primero constituyen los parámetros intrínsecos de la cámara y los dos últimos son los extrínsecos que nos indica donde están los objetos y su orientación. Los coeficientes de distorsión (K_1, k_2, p_1, p_2 y k_3) son los coeficientes de las ecuaciones de distorsión radial y tangencial, los cuales son útiles cuando queremos corregir la distorsión. La matriz intrínseca es tal vez el resultado más interesante ya que nos permite realizar la transformación del coordenadas 3D a 2D y al revés.

Vamos a examinar los parámetros de la función de calibrado.

```
Void cvCalibrateCamera2(
                CvMat* object_points,
                CvMat* imagen_points,
                int* point_counts,
                CvSize image_size,
                CvMat* intrinsic_matrix,
                CvMat* distorsion_coeffs,
                CvMat* rotation_vectors      = NULL,
                CvMat* translation_vectors   = NULL,
                int flags                    = 0
);
```

El primer argumento es *object_points*, una matriz de Nx3 que contiene las coordenadas físicas de cada una de los K puntos en cada una de las M imágenes de patrón de calibrado. Este argumento, *object_points,* también es el encargado definir las unidades físicas y la estructura del sistema de coordenadas que se va a utilizar. En nuestro caso el sistema de coordenadas definido es el siguiente: las coordenadas 'x', 'y' y 'z' se miden en milímetros. La distancia entre los puntos definen la unidad, es decir, si continuamos utilizando como ejemplo nuestro programa, cada punto tiene de separación 20 mm en cada lado, el mundo de la cámara, entonces la unidad en las coordenadas del objeto y la cámara se expresara como mm/90.

El segundo argumento es *image_points*, el cual es una matriz con dimensiones Nx2 que contiene las coordenadas en píxles de todos los puntos pasados en.

El argumentos *point_counts* indica el número de puntos en cada imagen; se le pasara como argumento una matriz de dimensiones Mx1, donde M es el número de imágenes. La *image_size* seutiliza para indicar el tamaño en píxeles de la imagen (por ejemplo cvSize(640, 480)).

Los dos siguientes argumentos, *intrinsic_matrix y distorsion_coeffs*, son los parámetros intrínsecos de la cámara. Estos parámetros pueden ser tanto de entrada como de salida. Cuando se usan como entrada, los valores de las matrices afectan al resultado de salida. Cual de las matrices es usada como entrada depende del parámetro *flags*. La matriz intrínseca deberá ser una matriz de dimensiones 3x3 y la matriz de coeficientes de distorsión deberá ser de dimensiones 5x1 (k_1, k_2, p_1, p_2, k_3).

Los parámetros *rotation_vectors y translation_vectors* son los parámetros extrínsecos de la cámara, nos devuelven una matriz de Mx3 (donde M es el número de imágenes), es decir, nos devuelven un vector de rotación y traslación por imagen. Como en nuestro caso queremos que nos devuelva los parámetros extrínsecos globales, no uno por imagen, estos parámetros los ponemos a NULL y luego con la función cvFindExtrinsicCameraParams2() le pasamos los parámetros intrínsecos ya calculados y obtenemos los parámetros extrínsecos globales de la cámara.

El argumento *flags* nos permite tener un poco de control sobre como queremos que se realiza el calibrado de la cámara. Esto se puede hacer utilizando los siguientes valores que se pueden combinar con la operación OR booleana.

CV_CALIB_USE_INTRINSIC_GUESS
Normalmente la matriz intrínseca es calculada por cvCalibrateCamera2() sin información adicional. En particular, los valores iniciales de os parámetros de c_x y c_y son tomados directamente del argumento *image_size.* Si ponemos este flag, los valores que contengan la matriz *intrinsic_matrix* se suponen como válidos y se utilizaran para optimizar le resultado.

CV_CALIB_FIX_PRINCIPAL_POINT
Este flag se puede usar con CV_CALIB_USE_INTRINSIC_GUESS. Si se usa solo CV_CALIB_FIX_PRINCIPAL_POINT, entonces el punto principal es el centro de la imagen. Si usamos el flag con CV_CALIB_FIX_PRINCIPAL_POINT, entonces el punto principal que se utiliza es el suministrado en el argumento *intrinsic_matrix.*

CV_CALIB_FIX_ASPECT_RATIO
Si se activa este flag junto con CV_CALIB_USE_INTRINSIC_GUESS, entonces la optimización producirá una variación de f_x y f_y según el ratio que se le indique en el argumente *intrinsic_matrix*. Si solo se activa el flag CV_CALIB_FIX_ASPECT_RATIO, entonces los valores de f_x y f_y del parámetro *intrinsic_matrix* pueden ser arbitrario y solo se considerará su ratio.

CV_CALIB_FIX_FOCAL_LENGTH

Este flag utiliza una rutina de optimización de los parámetros f_x y f_y pasados en *intrisic_params*.

CV_CALIB_FIX_K1, CV_CALIB_FIX_K2 y CV_CALIB_FIX_K3

Fija los parámetros de distorsión radial k_1, k_2 y k_3. Los flags de distorsión radial pueden aplicarse utilizando cualquier combinación. En general, el último parámetro se fija a 0 a no se que utilicemos una lente ojo de pez.

CV_CALIB_ZERO_TANGENT_DIST

Este flag es importante para calibrar cámaras 'high-end' las cuales tienen muy poca distorsión tangencial. Poner los parámetros con valores próximos a 0 puede producir ruido y problemas de estabilidad numérica. Utilizando este flag los parámetros de distorsión tangencial p_1 y p_2, son inicializados a 0.

En nuestro programa hemos utilizado los flags de calibrado CV_CALIB_FIX_ASPECT_RATIO y CV_CALIB_USE_INTRINSIC_GUESS, utilizando estos dos flags le indicamos al algoritmo de calibrado que la coordenada Z variará, de esta forma la coordenada Z del objeto que utilizamos variará en cada imagen que tomemos para el calibrado. Por otra parte le indicamos a la rutina de calibrado que el ratio es 1:1, quedando de esta forma la matriz intrínseca que le pasamos a cvCalibrateCamera2().

$$M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1000 & 0 & 240 \\ 0 & 1000 & 320 \\ 0 & 0 & 1 \end{bmatrix}$$

Como podemos ver en la matriz f_x y f_y son 1000 porque estamos calibrado en milímetros y además queremos que el ratio sea 1:1 y le indicamos el centro de la imagen que en esta caso es x = 240 e y = 320. Esta matriz utilizará la función de calibrado para optimizar el resultado.

Programa de Calibrado

El programa de calibrado consiste en obtener las imágenes de la cámara kinect, utilizando las funciones previamente implementadas y explicadas en secciones anteriores, aplicamos el algoritmo de segmentación para obtener los centros de los círculos que componen el target y una vez obtenemos todos los puntos deseados se le pasan a la función de calibrado. Para comprobar que la calibración funciona perfectamente en el programa podemos hacer un test de calibrado.

Parámetros de entrada

El programa de calibrado se invoca desde consola, los parámetros de entrada del programa son:

uso: kinect-calib-cams-internal [-p cadena] [-l iteraciones] [-c columnas] [-r filas] [--IR|--RGB]

p : le pasamos el fichero del patrón que queremos que encuentre en el target

c : indicamos el número de columnas del target

l : número de imágenes que se quieren captar

r : indicamos el número de filas del target

IR : le indicamos al programa que activa la cámara de infrarrojos de la cámara kinect.

RGB : le indicamos al programa que activa la cámara RGB de la cámara kinect.

En principio el patrón que utilizamos en el programa es el circulo por el que esta compuesto el target. La opción -r y -c nos indican en número de filas y columnas respectivamente, por defecto tienen los valores r = 20 y c = 28. La opción -l indica cuantas imágenes queremos capturar con la cámara kinect, por defecto l = 3. Finalmente las opciones IR e RGB, indican qué cámara de las que contiene la kinect queremos calibrar, la opción IR indica la cámara de infrarrojos y la opción RGB activa la cámara RGB.

Figura 16: patrón de selección

Calibrado

Ejecutamos el programa con las siguientes opciones, -p punto.jpg –RGB, no modificamos el número de filas y columnas ya que por defecto están las dimensiones del

target.

Cuando se inicia el programa aparece la ventana de información explicando las opciones del programa.

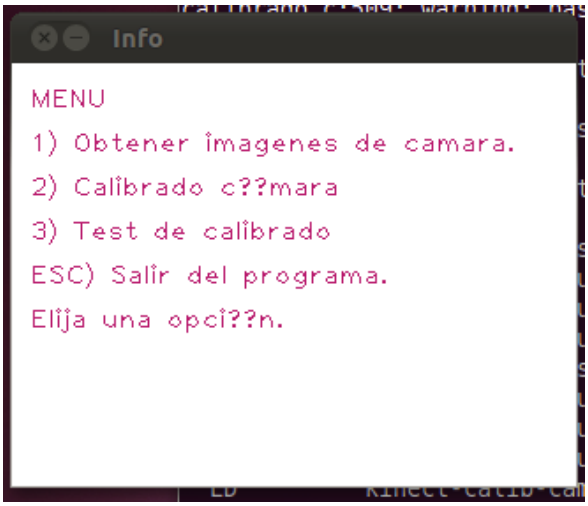


Figura 17: Ventana de información

Como vemos nos ofrece tres opciones, la primera obtener las imagen de la cámara, la segunda opción es la de calibrar cámara y la tercera sirva para realizar el test de calibrado.

Pulsamos la tecla '**1**' como indica el programa y aparece una ventana en modo video para ajustar el target para que se visualiza correctamente.

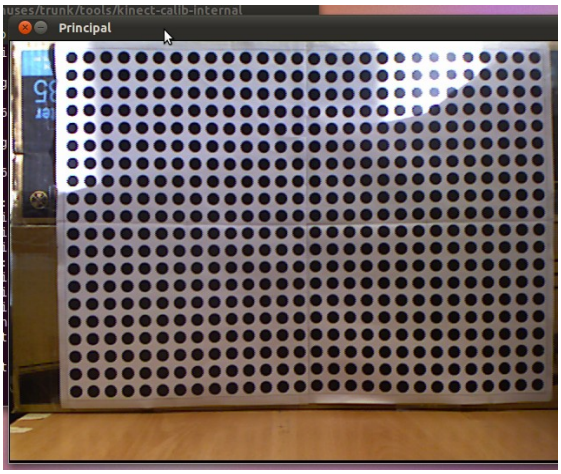


Figura 18: Modo video

Una vez ajustamos el target, pulsamos la tecla '**2**' para obtener los puntos para el calibrado. Aparecen dos ventanas nuevas la de binarización, en esta ventana aparece la

imagen en blanco y negro y debemos modificar los valores del umbral y región para conseguir el resultado deseado en el algoritmo de umbralizado. En nuestro caso, buscamos que se visualicen correctamente los puntos del target.

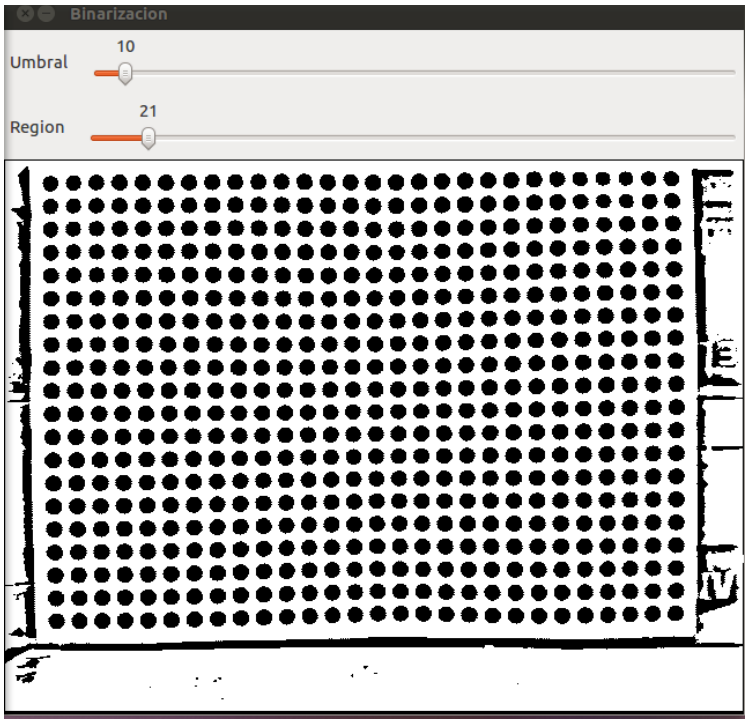


Figura 19: Ventana de binarización

Una vez obtenemos el umbralizado deseado, en la segunda ventana, la de Seleccionar, seleccionamos el área del target con el ratón pintando un rectángulo alrededor del área y modificamos los parámetros para conseguir encontrar los puntos del patrón de calibrado. Los parámetros son:

- Max. y Min. Per., los cuales indican el perímetros mínimo que debe tener el circulo del target para ser identificado como parte del target y el perímetro máximo, es decir, qué tamaño como máximo debe tener el perímetro del círculo para considerarlo parte del patrón de calibrado.
- Error, este parámetro indica cual es el error máximo que aceptamos para identificar los círculos del target.
- Parámetro Dist. Patrón, indica la distancia del target a la cámara en milímetros.

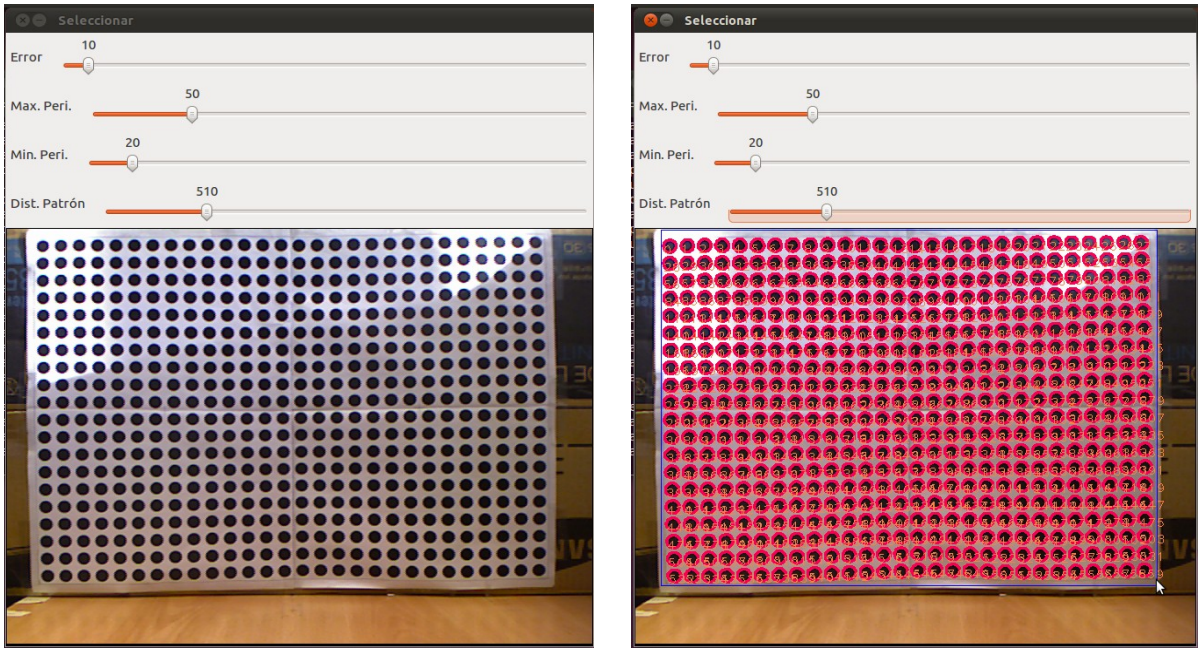


Figura 20: Ventana de selección

Una vez encontrados todos los puntos y ordenados, si estamos satisfechos del resultado pulsamos la tecla '**g**' para guardar los puntos para la función de calibración. Una vez guardados pulsamos otra vez la tecla '**1**' volvemos a la visualizar en modo video, posicionamos el patrón de calibrado en el sitio correspondiente y volvemos a aplicar los pasos anteriores. Estos pasos los realizaremos hasta que hayamos obtenido todas las imágenes, que por defecto son 3. Podemos saber cuantas imágenes faltan por calibrar en la ventana de información. En la zona de información nos indica las iteraciones que quedan por hacer.

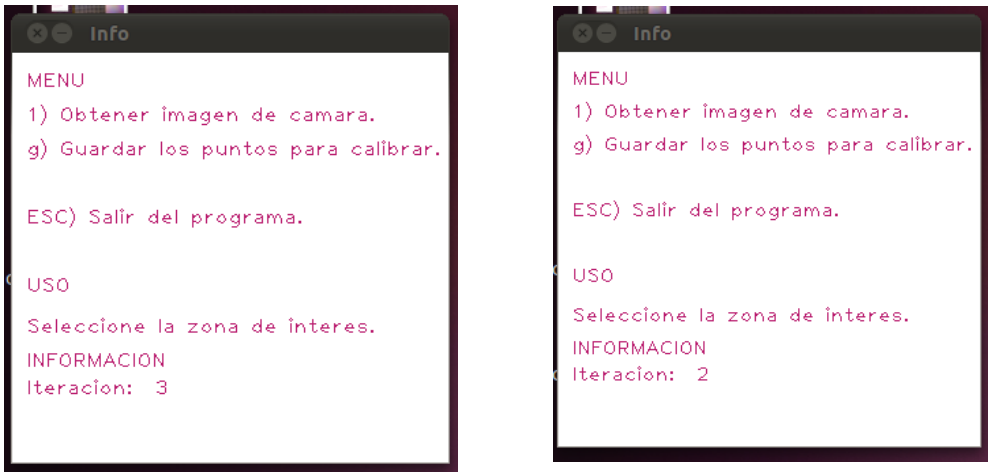


Figura 21: Ventana de información

Una vez obtenemos todos los puntos de todas la imágenes capturadas se calculara la matriz intrínseca, la matriz de rotación y la de traslación.

Matriz intrinseca:
533.003 0.000 312.005
0.000 533.003 234.906
0.000 0.000 1.000
Matriz rotación:
3.125 -0.016 0.014
Matriz Translacion
0.137 -35.982 7.929

Test

Una vez tenemos los parámetros intrínsecos y extrínsecos comprobamos que la calibración haya tenido éxito. Para ello se han implementado dos métodos de testeo. El primero consiste en introducir un punto 3D por la entrada estándar, el terminal, y pinta sobre la imagen una 'x' en la posición correspondiente. El segundo método consiste en desplazar la barra que hay sobre la imagen que indica la distancia a la que esta la cámara del target y dibujara una 'x' en cada esquina del patrón decalibrado.

Para empezar debemos pulsar la tecla '**3**', nos aparecerá una ventana en la que podemos ver las imágenes en modo video y la ventana de información indicándonos que opciones de testeo tenemos.

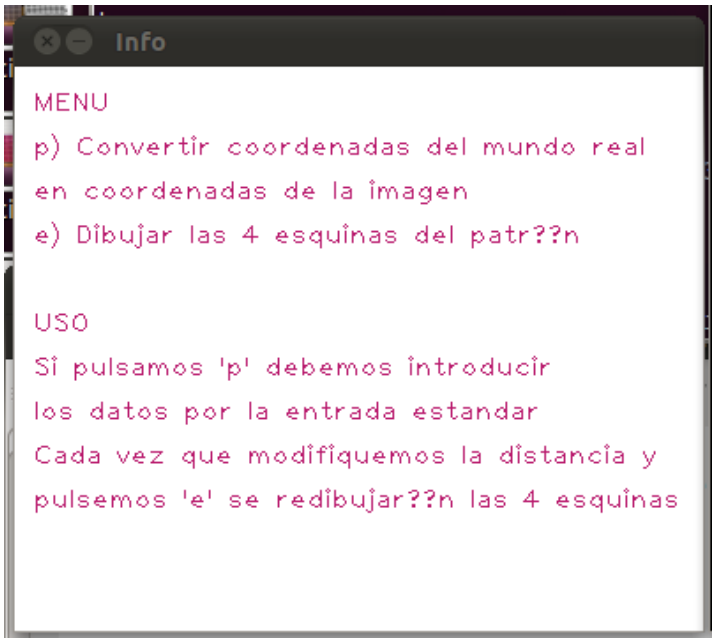
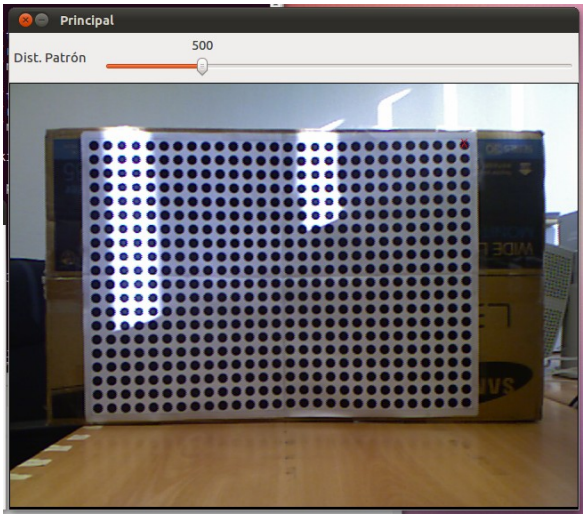


Figura 22: Ventana de información de test

Como vemos en la ventana de información, si pulsamos la tecla '**p**' nos aparecerá un mensaje en la consola donde debemos introducir las coordenadas del mundo, la coordenada Z debe ser negativa. Una vez introducida la coordenada aparecerá dibujada con una '**x**' en la ventana principal.




```
Introduce coordenadas mundo real (x y z): 270 190 -500

Imagen: 511.210, 68.550
```

Figura 23: Ejemplo de calibrado 1

El segundo método de test consiste en modificar el parámetro Dist. Patrón para indicarle a qué distancia está el patrón de calibrado de la cámara en milímetros, una vez hecho ésto pulsamos la tecla '**e**' para que nos dibuje una '**x**' en cada esquina del target. Cada vez que modificamos la distancia debemos presionar la tecla '**e**' para que vuelva a realizar los cálculos.

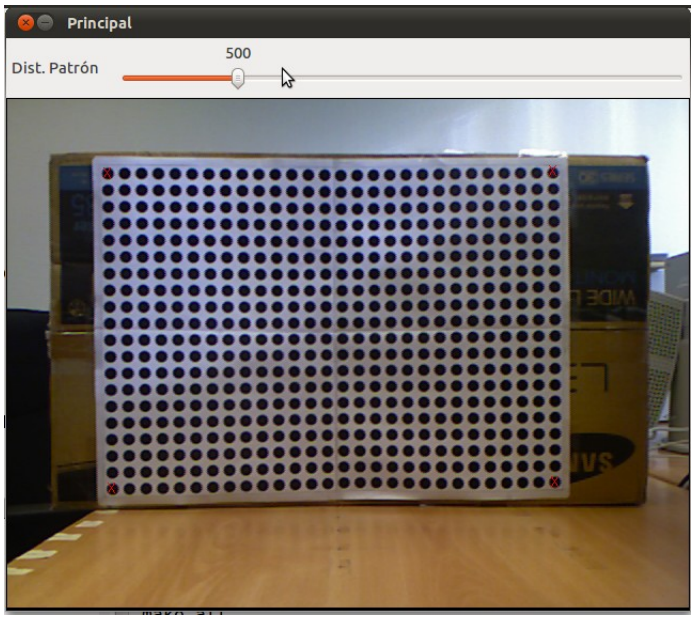


Figura 24: Método de test 2

Conclusiones

Durante la memoria se han abordados diferentes problemas, primero el encapsulado de las funciones OpenNI para poder trabajar con la cámara Kinect de manera más sencilla y segundo la calibración de sus dos cámaras, así como los pasos y técnicas que hemos seguido para resolverlo. Aquí se hablará de las conclusiones que se han sacado de realizar el proyecto.

El objetivo de programar una librería de encapsulamiento para funciones de OpenNI, para hacer más cómodo e intuitivo el uso de las diferentes cámaras de la Kinect se ha conseguido. Como se puede ver en apartados anteriores donde se explicaron las funciones de la librería easykinect, de las diversas funciones que han que utilizar en OpenNI, con easykinect solo es necesario una variable, la estructura Kinect, y ocho funciones. Con esto conseguimos que el usuario que quiera usar dicha cámara no necesita saber nada más como usar easykinect. Aunque uno de los inconvenientes de encapsular las funciones de la librería OpenNI ha sido, que la librería para obtener la información de la cámara Kinect esta hecha sobre C++, como consecuencia se ha debido de modificar la librería de forma que el usuario final que vaya a utilizar la librería no se percate de dicho problema.

Podemos decir que una librería encapsulada es muy útil para hacer que usuarios menos expertos pueden utilizar rutinas más complejas sin tener que saber como funcionan o librerías con una interficie muy compleja sea más intuitiva. Por otra parte, también es interesante el encapsulado ya que puedes hacer que funciones de un lenguaje se puedan utilizar en otro, como en nuestro caso de C++ a C.

Una vez ya tenemos el encapsulamiento podemos observar que la cámara de infrarrojos y la cámara RGB, no se pueden activar a la vez, debido a que ambas utilizan el mismo bus para transmitir los datos. Si se pueden activar junto la cámara de profundidad.

En segundo lugar, el objetivo de calibrar la cámara se ha cumplido, en un principio se quería realizar la calibración utilizando el algoritmo DLT del departamento del DISCA, pero debido a la falta de información y de tiempo no se consiguió un calibrado óptimo así que se decidió cambiar de algoritmo y utilizar el algoritmo de la librería OpenCV, que tiene funciones para el calibrado y contiene herramientas útiles para el procesamiento de imágenes, con la cual si obtuvimos el calibrado deseado.

Una vez obtenido el calibrado se observa que los parámetros intrínsecos varían de una ejecución a otro. Esto se debe a las distintas fuentes de error durante el proceso de calibración, empezando por el conjunto de puntos introducido por el usuario. Por otra parte cuantos más puntos insertemos más preciso será el resultado pero siempre queda un error residual imposible de evitar. Todos estos factores contribuyen a que los datos obtenidos de una ejecución a otra cambien ligeramente.

El sistema semiautomático de detección de los objetos del patrón funciona correctamente. Funciona bastante bien aunque la iluminación no sea bastante buena cuando calibramos la cámara RGB, la cámara de infrarrojos es bastante sensible a la luz. Además la fuente emisora de infrarrojos emite ruido que aparece en la imagen, para evitar esto debemos taparla para eliminar dicho ruido y así se obtener una mejor detección.

La intervención del usuario en el proceso iterativo de calibración produce una fuente de errores grande, lo cual produce inestabilidad en los resultados. Por esta razón nuestro programa está diseñado para que la intervención del usuario sea mínima, de esa manera quitamos parte del error y aumentamos la estabilidad de los resultados obtenidos.

Trabajos futuros

En este proyecto se ha conseguido calibrar la cámara Kinect y hacer un pequeño test donde pasándole puntos del mundo real (3D) los transforma en puntos de la imagen (2D). El siguiente paso podría ser la operación inversa convertir puntos 2D a 3D. Una vez conseguida la operación inversa se podría utilizar como sistema de visión a un robot.

Como se ha descrito anteriormente, con las cámara calibradas y calculando la correspondencia de un punto de la imagen de RGB a un punto de la imagen de profundidad se podría realizar un programa que calcule la distancia de los objetos a la cámara. Un uso de esta aplicación podría ser para que el robot detectara la proximidad de obstáculos o hacer un filtro en la imagen y obtener objetos que este a una cierta distancia.

Estudiar más funcionalidades de la cámara Kinect, tales como el reconocimiento de voz y reconocimiento facial para la identificación automática de los usuarios, la aplicación seguimiento Kinect y el sensor de giro motorizado para ajustar la cámara para que el objeto seleccionado se mantenga en el marco, incluso cuando se mueve.

También queda pendiente hacer que el programa de calibrado sea compatible con el algoritmo de calibración DLT.

Bibliografía

Libros

[Suzuki85] S. Suzuki and K. Abe, “Topological structural analysis of digital binary images by border following,” Computer Vision, Graphics and Image Processing 30 (1985): 32–46.

[Reeb46] G. Reeb, “Sur les points singuliers d’une forme de Pfa completement integrable ou d’une fonction numerique,” Comptes Rendus de l’Academie des Sciences de Paris 222 (1946): 847–849.

[Bajaj97] C. L. Bajaj, V. Pascucci, and D. R. Schikore, “The contour spectrum,” Proceed- ings of IEEE Visualization 1997 (pp. 167–173), 1997.

[Kreveld97] M. van Kreveld, R. van Oostrum, C. L. Bajaj, V. Pascucci, and D. R. Schikore, “Contour trees and small seed sets for isosurface traversal,” Proceedings of the 13th ACM Symposium on Computational Geometry (pp. 212–220), 1997.

[Hu62] M. Hu, “Visual pattern recognition by moment invariants,” IRE Transactions on Information Theory 8 (1962): 179–187.

[Heikkila97] J. Heikkila and O. Silven, “A four-step camera calibration procedure with implicit image correction,” Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (p. 1106), 1997.

[Zhang00] Z. Zhang, “A flexible new technique for camera calibration,” IEEE Transactions on Pattern Analysis and Machine Intelligence 22 (2000): 1330–1334.

[Brown71] D. C. Brown, “Close-range camera calibration,” Photogrammetric Engineering 37 (1971): 855–866.

enlaces:

Umbralizado adaptativo
http://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm

Documentación OpenNI
http://openni.org/Documentation/

Kinect, freenect, OpenKinect, y OpenNI
http://landerpfc.wordpress.com/2011/02/18/kinect-freenect-openkinect-openn/

OpenNI, el driver Open Source oficial de Kinect
http://www.muylinux.com/2010/12/17/openni-el-driver-open-source-oficial-de-kinect/