Apéndice A

La cámara Microsoft Kinect

A.1. Especificaciones técnicas

La cámara Kinect es un periférico de entrada desarrollado por Microsoft para jugar en la videoconsola Xbox 360 que salió a la venta en Noviembre de 2010. Se trata de un controlador que tiene el propósito de ser capaz de permitir jugar a los videojuegos sin necesidad de ningún mando, percibiendo y reconociendo los cuerpos de los jugadores y los movimientos que realizan, así con reconocimiento de voz. Para ello hace uso de dos cámaras frontales, una convencional de RGB y un sensor de distancia, y de una serie de micrófonos.

El sistema de percepción de profundidad consta de tres partes básicas: el proyector láser de infrarrojos, el sensor CMOS y el microchip que procesa la información. Fue creado y desarrollado por PrimeSense, una compañía Israelí experta en innovación. Contrariamente a lo que podríamos suponer en un principio, no se trata de un sensor basado en tiempo de vuelo, sino que su funcionamiento se basa en la proyección de un patrón de puntos pseudo-aleatorio y su lectura y triangulación mediante el sensor CMOS. En la sección A.2 se entra más en detalle en cuanto a funcionamiento del sistema.

En la Figura A.1 se puede ver con claridad las distintas partes que componen la cámara de forma numerada:

1. Conjunto de micrófonos. Está compuesto por cuatro micrófonos, que permiten la comunicación con el dispositivo y darle órdenes al mismo. Su configuración hace que sea capaz de reconocer la localización de donde proviene el audio dentro de la habitación, y están programados para captar y erradicar el posible ruido de ambiente.

2. Proyector láser. Posee las siguientes propiedades [6]:

   - Se trata de un láser no modulado, no produce pulsos en su salida sino que se mantiene a nivel constante.
   - La longitud de onda es de 830nm.
   - Consta de un sistema de difracción que subdivide el rayo en múltiples instancias, proyectando un patrón de puntos pseudo-aleatorio.

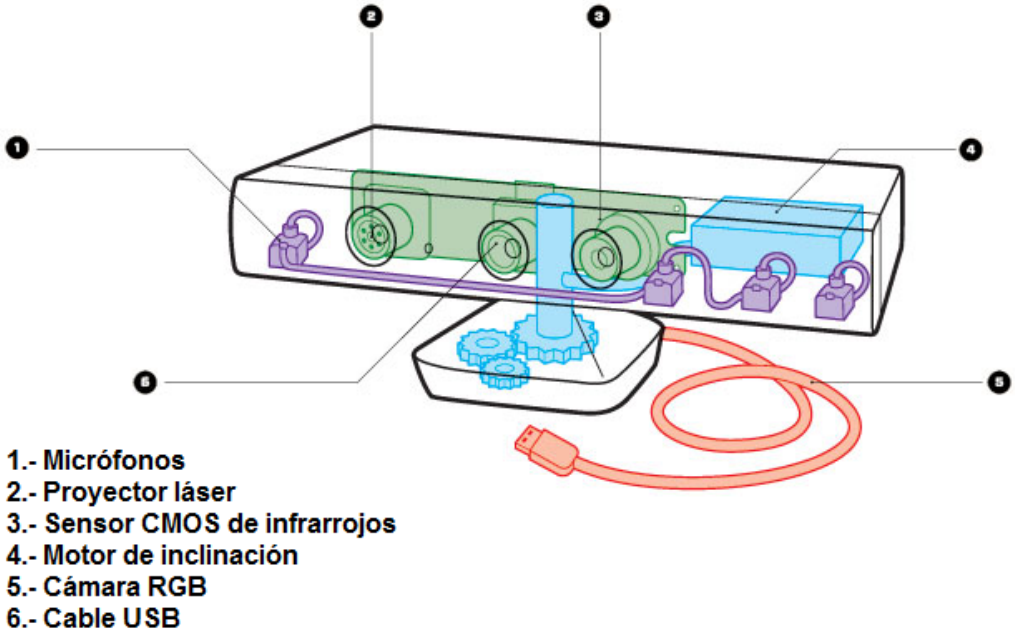
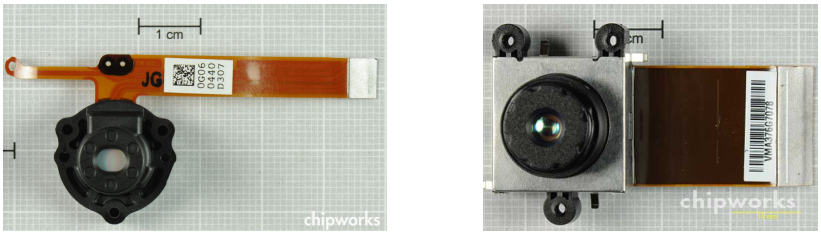


Figura A.1: Esquema de los elementos de la Kinect



(a) **(b)**

Figura A.2: (a) Emisor láser que proyecta el patrón de puntos. (b) Sensor CMOS que capta el patrón emitido para calcular la profundidad

- La potencia medida a la salida de la Kinect es de 60mW. No es dañina para los ojos, si bien es probable que la difracción inducida haya introducido pérdidas.
- Posee un estabilizador de temperatura que se encarga de mantener el láser a temperatura constante para no alterar la longitud de onda de salida.

3. Sensor de infrarrojos. Microsoft no ha desvelado de qué componente específico se trata. Expertos en ingeniería inversa han determinado que se trata del sensor CMOS monocromo de 1/2"MT9M001C12STM, de la marca Micron (Figura A.2b). Posee las siguientes propiedades:

   - Tamaño de píxel: $5,2\mu m$

- Formato de vídeo 5:4 con resolución SXGA 1280 × 1024 (1,3 Megapíxeles).
- 30 fps programables.
- Rango de temperaturas de trabajo: 0ºC a 70ºC
- Posee un filtro de paso de infrarrojos a la misma frecuencia del láser. Experimentos realizados con otras fuentes de luz (visible, 950nm) producen una influencia en el sensor mínima.
- Aunque se trata de un sensor de infrarrojos, se sitúa en el rango de captación del espectro electromagnético del infrarrojo cercano (entre 780 nm y 2.500 nm, ya que mide 830 nm). Esto hace que sea totalmente imposible medir temperaturas, al ser el rango óptimo utilizado por cámaras termográficas el del infrarrojo medio (en concreto, entre 3.000 nm y 14.000 nm).
- El campo de visión en horizontal de la cámara es de 57º (Figura A.3) y el rango máximo de distancia desde la cámara para la captación de profundidad es de 0,8 metros a 4 metros, considerándose un rango óptimo de 1,2 metros a 3,5 metros. En Febrero de 2012 salió al mercado una nueva Kinect diseñada para usar en Windows, que amplía el rango de cercanía a 0,4 metros al poseer un «modo cercano».

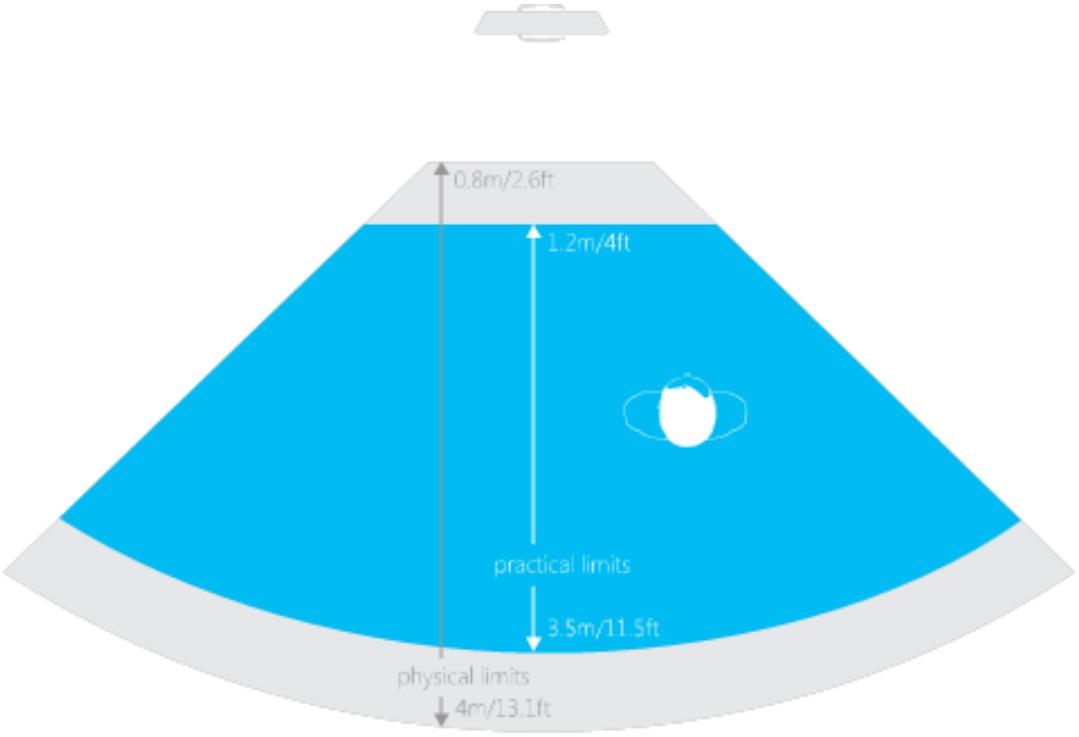


Figura A.3: Campo de visión horizontal y rango de proximidad de la Kinect

- El campo de visión vertical de la cámara estática es de 43,5º (Figura A.4).

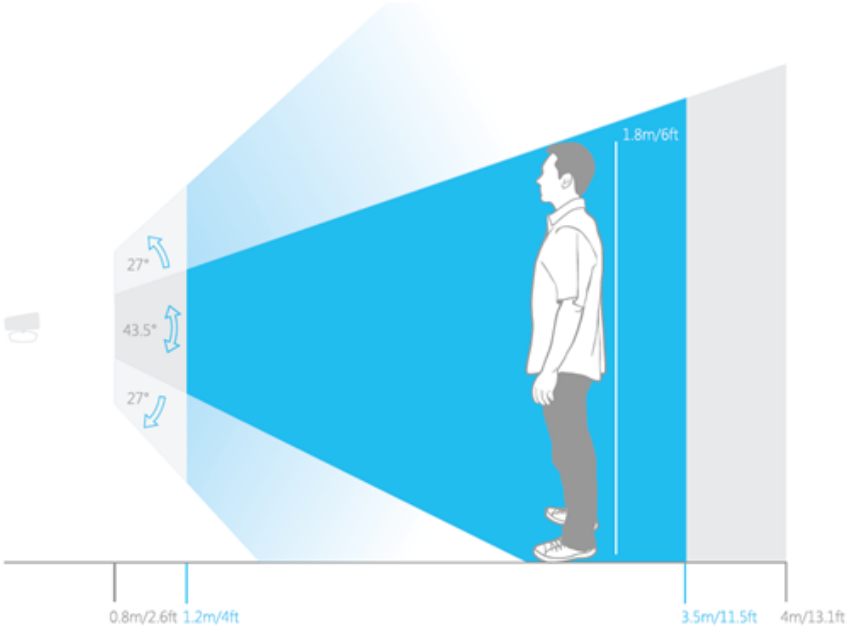


Figura A.4: Campo de visión vertical y variación del mismo provocada por la inclinación del motor

4. Motor de inclinación de la Kinect. La cámara está equipada con un motor que inclina arriba o abajo la cámara $\pm 27^{o}$ para poder abarcar cómodamente el cuerpo de los usuarios sin importar la altura a la que está situada la cámara (arriba o abajo de la televisión) (Figura A.4). Posee un acelerómetro que indica la inclinación que posee en cada instante.

5. Cámara RGB. Al igual que ocurría con el sensor de profundidad, se ha conocido la identidad del componente a través de la ingeniería inversa. Es de la misma familia, un Micron MT9M112 (Figura A.5), un sensor de imagen CMOS System-on-Chip. Posee las siguientes características técnicas:

   - Tamaño de píxel: $2,8\mu m \times 2,8\mu m$
   - 1,3 Megapíxeles
   - Resolución: 1.280H $\times$ 1.024V
   - Filtro de color de Bayer.
   - 15 fps a máxima resolución, 30 fps a 640×512
   - Rango de temperaturas de trabajo: -30°C a +70°C
   - Soporta VGA, QVGA, CIF y QCIF

6. Cable conector USB. La conexión tanto a la consola como al ordenador se hace mediante USB 2.0, lo cual limita en parte la cantidad de información que se puede procesar por segundo, que teóricamente podría ascender a 60MB/s aunque suele acercarse más a los 35MB/s. Esto hace que no se

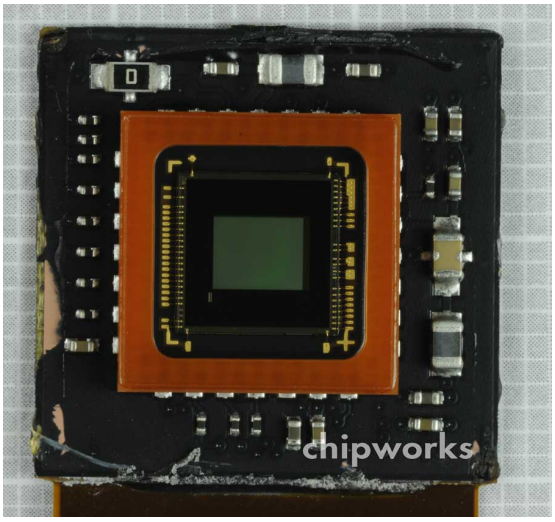


Figura A.5: Foto de la cámara extraída de la Kinect

pueda transmitir toda la información de imágenes a plena resolución y con una tasa de fps de 30, por lo que constituye uno de los motivos por los que la resolución de las cámaras no es la que finalmente llega a procesarse en el ordenador.

Con el USB se pueden alimentar las cámaras, si bien es cierto que el poseer motor de inclinación hace que la potencia aportada no sea suficiente y tenga que contarse a su vez con una conexión a la red eléctrica.

Estas partes descritas se encuentran montadas dentro de una caja de plástico negra y alargada, donde las cámaras y el láser permanecen unidos a una placa metálica que los mantiene situados a la distancia oportuna (Figura A.6a). Si se doblase esa placa se alteraría la imagen de profundidad obtenida. La caja va unida a la base mediante una barra móvil que es accionada por el motor de inclinación.

Paralelamente a la placa metálica se sitúan dos placas con microchips que se encargan de controlar el sistema y procesar la información recogida por los sensores (Figura A.6b). Entre ellos destaca el PS1080 SoC, el auténtico «cerebro» de la Kinect, que se encarga de procesar la información de profundidad, ejecutando los algoritmos necesarios dentro del propio chip, así como la de imagen RGB y microfonía (Figura A.7). Cuenta con una interfaz USB 2.0 que es la que transmite la información procesada. El procesamiento de la profundidad se hace pues, dentro de la Kinect, y no depende de a qué esté conectada. La información no se procesa en el ordenador, éste sólo la lee.

Las especificaciones técnicas del microchip se muestran en la tabla A.1. Ahí se ponen en evidencia algunas restricciones que se han visto en cuanto a los componentes, como el campo de visión, la resolución de las imágenes de profundidad y el rango de operación. Los *fps* no se ven limitados por el chip, sino por los sensores en sí y la conexión USB en este caso, igual que la resolución de la imagen de color.

(a) **(b)**

Figura A.6: a) Sensores y láser empotrados en una placa metálica. b) Placas paralelas a la metálica con la electrónica y microchips que controlan el dispositivo.

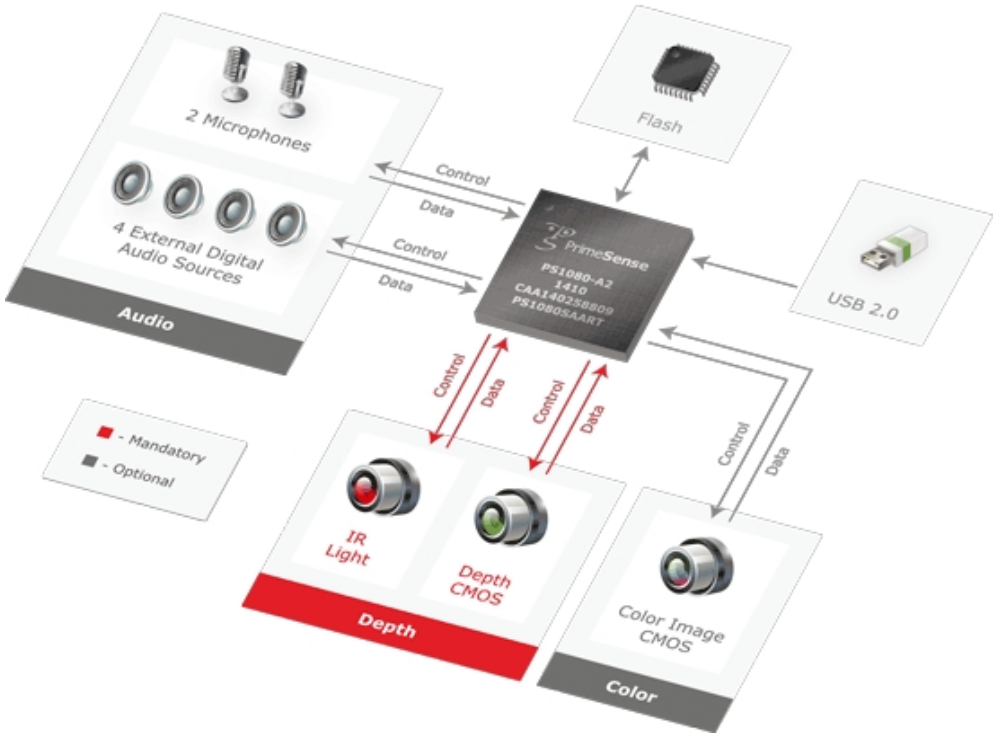


Figura A.7: Esquema de funcionamiento del PS1080

Especificación técnica	**PS1080**
Campo de visión	58º Horizontal, 45º Vertical, 70º Diagonal
Tamaño de la imagen de profundidad	VGA (640 × 480)
Resolución espacial en x/y a 2m de distancia	3mm
Resolución en Z (profundidad) a 2m de distancia	1cm
Frames por segundo máximos	60 fps
Rango de operación	0.8m - 3,5m
Tamaño de la imagen de color	UXGA (1600 × 1200)
Audio: Micrófonos integrados	Dos micrófonos
Audio: Entradas digitales	Cuatro entradas
Interfaz de datos	USB 2.0
Fuente de energia	USB 2.0
Consumo	2.25W
Dimensiones (Ancho x alto x espesor)	$14cm \times 3,5cm \times 5cm$
Ambiente de uso	Cerrado
Temperatura de operación	0ºC - 40ºC.

Tabla A.1: Especificaciones del PS1080

Además, la Kinect está equipada con un ventilador que se activa cuando la temperatura alcanzada es mayor de 70ºC (recordar que el valor máximo del rango de operación de los sensores ronda esa temperatura) y que se alimenta por la toma USB.

A.2. Funcionamiento del sensor de profundidad

De todas las características y prestaciones que tiene la Microsoft Kinect descritas en la sección A.1, una de las más interesantes para el proyecto es la percepción de profundidad. Por ello, esta sección va a comentar su funcionamiento básico y la base matemática que hay detrás de su óptica.

Las claves del funcionamiento son parcialmente desconocidas al no haberse revelado explícitamente, pero análisis realizados por expertos [6, 10, 9] junto con la observación de las patentes que PrimeSense han ido publicando en los últimos años dejan entrever algunos de sus secretos.

Observando las imágenes captadas por la Kinect, a priori no puede asegurarse qué método emplea. Se sabe que dispone de un emisor láser y de un receptor que capta las señales reflejadas del láser. Podría tratarse de un sensor de tiempo de vuelo (*Time-Of-Flight*), aquel que la distancia de la escena se determina cronometrando el tiempo de viaje de ida y vuelta de un pulso de luz. Dado que

la luz del láser se difracta en numerosos puntos, de este modo podría obtenerse la información de la escena completa. Pero el láser no emite pulsos, no está modulado.

La Kinect utiliza un procedimiento de luz estructurada para extraer la profundidad. El emisor emite un patrón de puntos, y la observación por medio del sensor de las variaciones en el patrón indican la forma y situación de los objetos mediante un proceso de triangulación. Para realizar la triangulación ha de haber un previo patrón de referencia.

El láser emite un rayo único que se divide formando el patrón, que es constante y se proyecta en la escena. Una captura realizada por una cámara capaz de captar señales infrarrojas nos muestra su aspecto en la Figura A.8

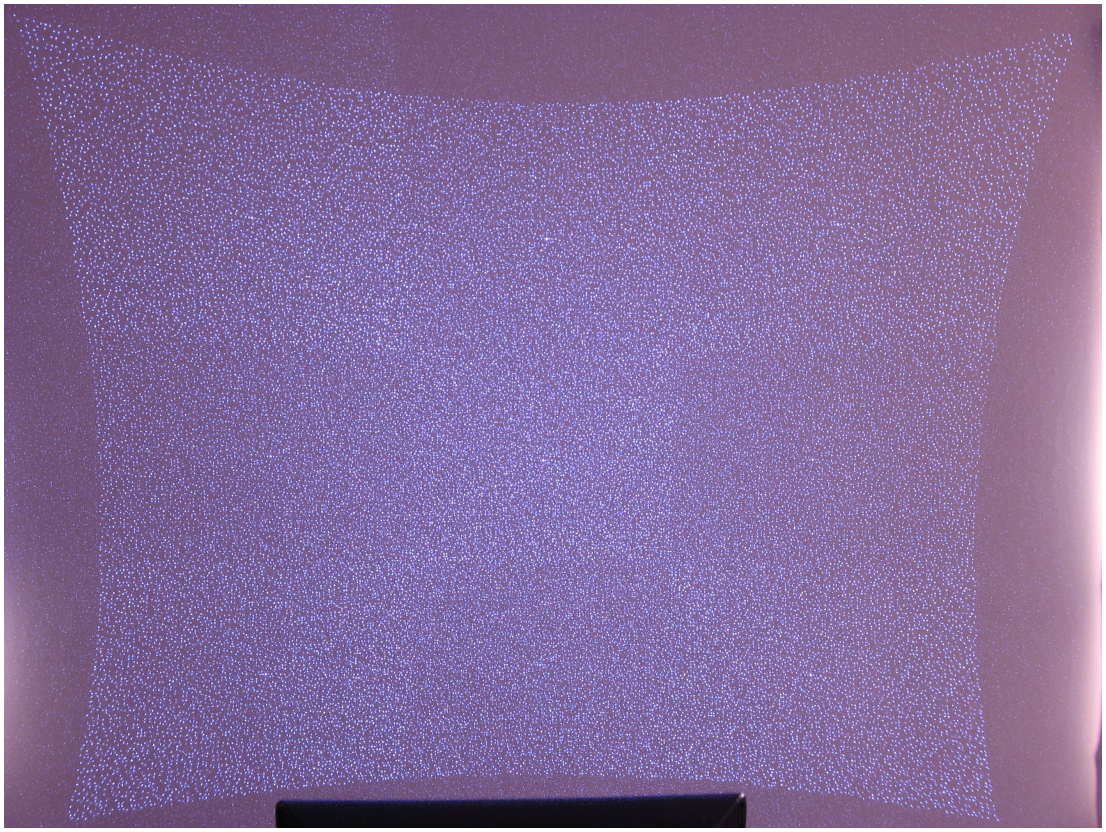
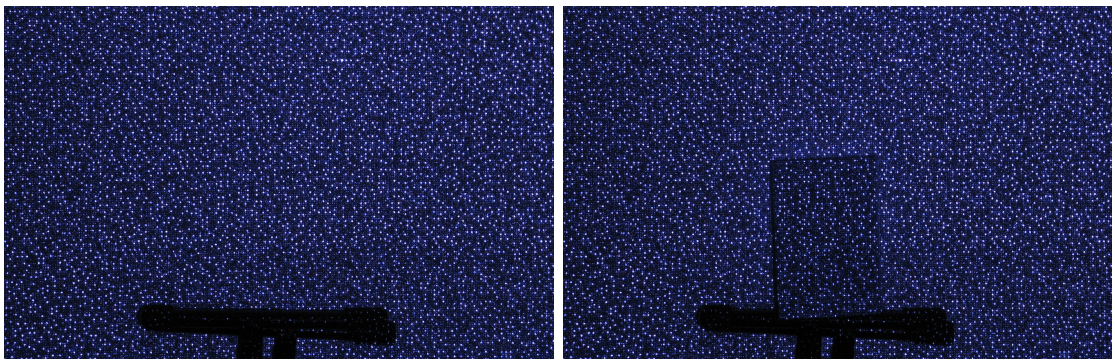


Figura A.8: Captura del patrón de puntos sobre una pared

En ella se ve cómo el número de puntos es muy elevado, y su posición no sigue una ordenación aparente, sino que parece aleatorio, pese a tratarse siempre del mismo y ser constante. Sin embargo también se puede observar que parece que se divide en nueve subsecciones formando una matriz 3×3 donde el centro de cada una aparece ligeramente más iluminado. Una observación más exhaustiva permite además ver que hay algunos puntos que están menos iluminados con mucha menos intensidad que otros. Se desconoce la razón por la que el patrón tiene estas características, pueden tratarse de consecuencias derivadas del difractor empleado o formar parte de una serie de «pistas» que ayudan al sistema a establecer las

correlaciones para la triangulación.



(a) **(b)**

Figura A.9: Distorsión del patrón al introducir un objeto en escena. a) Pared sin nada. b) Se ha situado un libro delante de la pared

El patrón de referencia se obtiene capturando un plano a una distancia conocida del sensor, y permanece almacenado en la memoria desde el proceso de fabricación de la cámara. Cuando un patrón de puntos se proyecta hacia un objeto cuya distancia al sensor es mayor o menor que la del plano de referencia, el patrón se distorsiona desde el punto de vista del sensor, ocasionándose desplazamientos de los puntos en la dirección de la línea que une el proyector láser con la cámara infrarroja. Esto se ilustra en la Figura A.9, donde se ve cómo se distorsiona el patrón proyectado sobre la pared sin nada delante al introducir un libro en la escena. Si pensamos en la pared como el plano de referencia, se ve cómo el patrón, proyectado por el láser desde la derecha, se desplaza en la zona del libro hacia el lado derecho produciéndose una especie de sombra a la izquierda del libro, puesto que el sensor capta la imagen desde la izquierda y los rayos se ven interceptados y no llegan a la zona oscura. El principio de funcionamiento general de la Kinect es similar, solo que en vez de la pared es un plano precargado en la memoria a una distancia determinada.

Para contemplar y medir los desplazamientos en los puntos que permiten ejecutar la triangulación ha de establecerse una correlación entre los puntos de la imagen captada y el patrón de referencia. Se desconocen los detalles exactos de funcionamiento de la misma, pero en esencia parece basarse en una búsqueda por secciones de la imagen y una comparación posterior hasta encontrar puntos que sean coincidentes en ambos patrones. Una vez encontrada alguna coincidencia se procede a ir agrandando la región mirando los píxeles colindantes y presuponiendo que las diferencias en cada superficie no son muy grandes. Se sigue aumentando la región hasta el momento en que ocurren grandes variaciones en todas las direcciones que hagan pensar que se trata de una región distinta. Se deja de mirar en esa región y se toma otro punto coincidente. Pese a la complejidad del proceso la Kinect es capaz de computarlo de forma rápida.

Para describir el proceso matemático de obtención de distancias nos apoyamos en la Figura A.10. En ella se ve la relación triangular entre el punto del objeto a

tratar k visto desde el sensor relativo al plano de referencia y la disparidad medida d. Se considera el origen de las coordenadas tridimensionales situado en la cámara infrarroja, con el eje Z ortogonal al plano de la imagen y dirigido al objeto y el eje X en la línea base b que une la cámara con el proyector perpendicular al anterior.

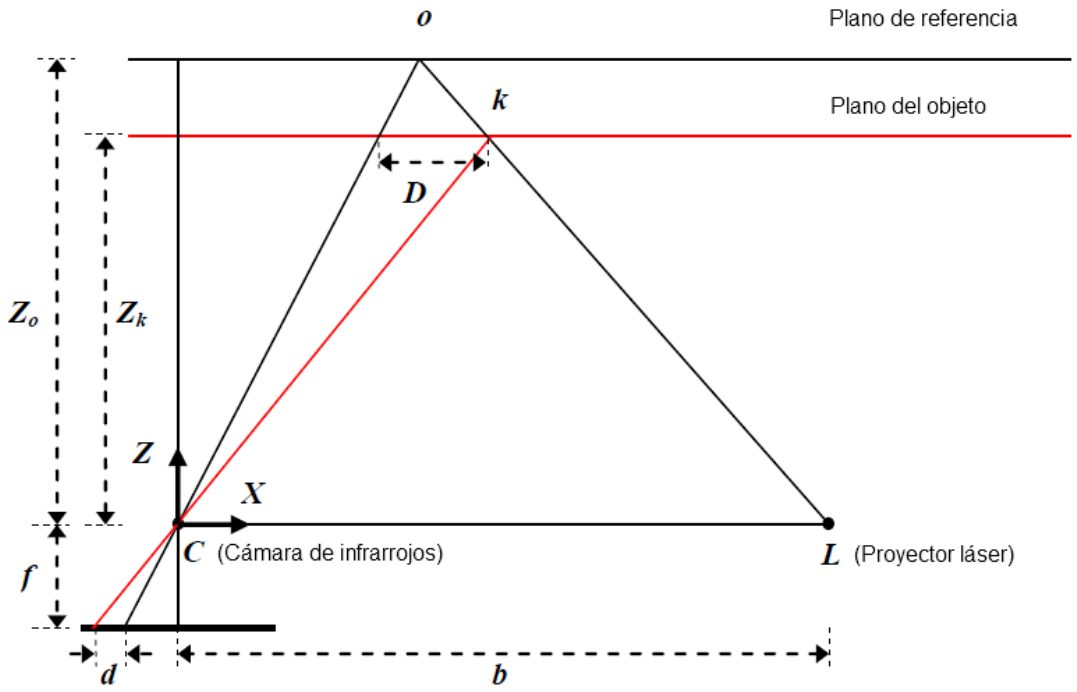


Figura A.10: Esquema de la triangulación empleada para obtener el valor de profundidad desde la disparidad captada por el sensor

En el esquema se ha situado el plano del objeto a una distancia Z_k menor que la distancia del plano de referencia Z_o. Por tanto, al proyectarse el patrón desde L, el punto proyectado que en el plano de referencia visto por C se situaba en o, ahora se situará en k, al haber ahora un objeto delante. Desde C se percibe como un desplazamiento en el eje X a la derecha de magnitud D. Si se hubiese considerado el objeto más lejano al plano de referencia, el desplazamiento hubiese sido hacia la izquierda.

Lo que mide y registra el sensor no es directamente la distancia D del espacio del objeto, sino la disparidad d. Para cada punto k se mide dicho parámetro y se obtiene un mapa de disparidad. Observando los triángulos que forman la imagen obtenemos:

$$\frac{D}{b} = \frac{Z_o - Z_k}{Z_o} \tag{A.1}$$

y:

$$\frac{d}{f} = \frac{D}{Z_k} \tag{A.2}$$

Donde Z_k, la distancia (profundidad) del punto k del objeto, es la variable que se quiere obtener; b es la longitud de la base entendida como la distancia entre la cámara y el proyector; f es la distancia focal del sensor infrarrojo; D es la distancia real de desplazamiento del punto k en el espacio del objeto y d es la disparidad observada en el espacio de la imagen. Sustituyendo D de ecuación A.2 en ecuación A.1 y despejando se obtiene:

$$Z_k = \frac{Z_o}{1 + \frac{Z_o}{fb} d} \tag{A.3}$$

Los parámetros Z_o, f y b se pueden determinar del proceso de calibración. La coordenada Z de cada punto junto con f definen la escala de la imagen para ese punto. Las coordenadas planimétricas del objeto en cada punto se pueden calcular a partir de las coordenadas de la imagen y la escala:

$$\begin{aligned} X_k &= -\frac{Z_k}{f}(x_k - x_o + \delta x) \\ Y_k &= -\frac{Z_k}{f}(y_k - y_o + \delta y) \end{aligned} \tag{A.4}$$

Donde x_k e y_k son las coordenadas en la imagen del punto, x_o e y_o son las coordenadas del punto principal, es decir, del *offset* de la imagen y δx y δy son las correcciones de la distorsión de la lente. Tanto los valores de *offset* como los de las correcciones también se pueden obtener del proceso de calibración.

Con los parámetros de la calibración conocidos, se puede completar la relación entre los puntos medidos de la imagen (x, y, d) y las coordenadas del objeto (X, Y, Z) de cada punto. Así se puede generar una nube de puntos de cada imagen de disparidad.

La disparidad así descrita es una medida de la «profundidad inversa», valores más grandes de la misma significan distancias más cortas. Pero el valor de disparidad que retorna la Kinect, d' está normalizado con un *offset* según la relación:

$$d' = d_{offset} - 8d \tag{A.5}$$

De modo que los valores que se obtienen mediante el software sí que crecen con la distancia. Por esto, a efectos prácticos, se han utilizado los valores de disparidad normalizada d' como si de la profundidad se tratase puesto que se comportan de igual manera y evita la adición de cálculos extra. A través de d' es posible obtener mapas de disparidad donde cada punto tiene una representación (x, y, d'). Dichos mapas son los que se denominan habitualmente «imágenes de profundidad» (Figura A.11).

Las imágenes de profundidad tal y como llegan al sistema después de procesarse dentro de la cámara son imágenes de 11-bits de 640×480. En el entorno de trabajo aparecen como matrices cuyos píxeles tienen un valor numérico de 0 a $2^{11} = 2047$. El valor de cada elemento se corresponde con la disparidad normalizada obtenida en cada punto. Los píxeles donde no haya información de profundidad los consideraremos de valor cero, aunque en ROS lo recibamos como

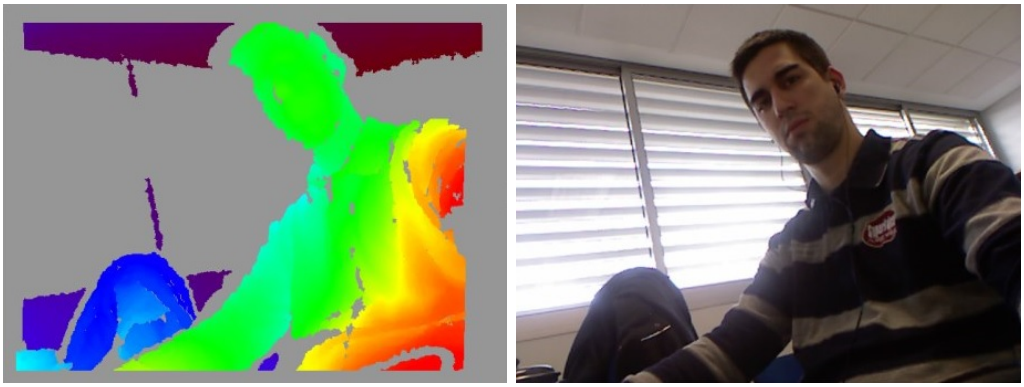
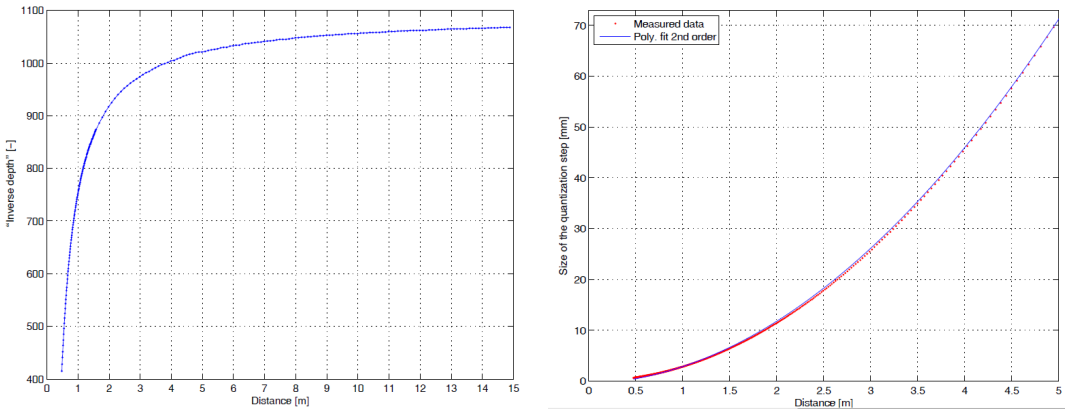


Figura A.11: Mapa de disparidad (izquierda) calculado por la Kinect e imagen RGB convencional

Not a Number. Para interpretar correctamente los valores de disparidad, estudios detallados del proceso de calibración de la Kinect ([10]) proporcionan los gráficos de la Figura A.12.



(a) **(b)**

Figura A.12: Correspondencia entre disparidad d y distancia real en metros (a). Tamaño de paso en milímetros entre dos valores consecutivos de disparidad en función de la distancia en metros (b). Gráficas obtenidas de [10]

En la Figura A.12a se representa la relación de los valores con las distancias reales en metros. En el entorno de trabajo se emplean valores de disparidad entre 600 y 700, que se sitúan por debajo del metro (la distancia entre la cámara y los distintos puntos de la mesa están en ese intervalo). Por otra parte, en la Figura A.12b se tiene idea de la precisión que se obtiene con este sensor a diferentes distancias. La curva obtenida ha sido sometida a un análisis de regresión que cumple la función.

$$q(z) = 2{,}73z^2 + 0{,}74z - 0{,}58[mm] \tag{A.6}$$

Así, en los intervalos en los que se trabaja en este proyecto $(0{,}6 - 0{,}9m)$ obtenemos una precisión de entre 0,8468 y $2{,}2973mm$, según a la distancia a la

que se encuentren los objetos. Dicha precisión disminuye con la distancia, por lo que se demuestra que la Kinect mide mejor en distancias cortas siempre y cuando no sea inferior al límite de correcta medición (aproximadamente $0{,}5m$).

A.3. Conectividad de la cámara con el ordenador

El cable que sale de la Kinect tiene una terminación especial para conectarla a la Xbox. No obstante se dispone de un adaptador con una terminación de puerto USB que utilizaremos para conectarla al ordenador. La energía que proporciona el USB no es suficiente para alimentar a los sensores y el motor de inclinación de la cámara, por lo que también se dispone de una conexión adicional a la corriente eléctrica en forma de adaptador de corriente (Figura A.13).

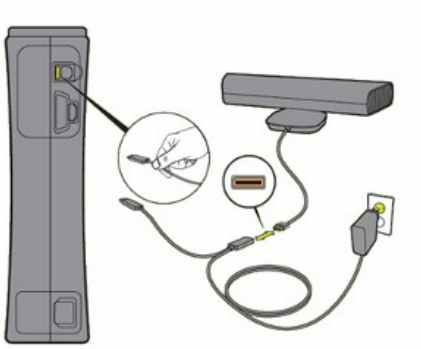


Figura A.13: Adaptador de USB y corriente para la Kinect

La mayor parte del desarrollo llevado a cabo internacionalmente con la Kinect se ha hecho empleando *drivers* y librerías para programar en C++ o C# ([6], [3]). Por ello, en este proyecto utilizamos ROS que tiene un compilador para C++ y para Python. Concretamente, se programa en C++, con el que se realiza los algoritmos, además de realizar el tratamiento de imágenes con la librería OpenCV, la cual nos ofrece muchas posibilidades.

A.4. Imágenes RGB y de profundidad

El uso de las imágenes RGB y de profundidad constituye uno de los pilares de este proyecto, por lo que es conveniente dejar claro en qué consisten y cómo se manejan, así como ayudar a entender la visualización de las mismas.

La obtención de la imagen RGB se obtiene de la misma manera que con una cámara convencional. RGB es un modelo que permite representar cualquier color mediante la mezcla de tres colores primarios. En este caso, y como indican las siglas, *rojo,verde y azul (Red,Green,Blue)*.

La obtención de la imágenes por parte de la cámara de infrarrojos se basa en un sistema de luz estructurada, donde un patrón de puntos láser proyectado por la propia cámara (Figura A.14) es leído por el sensor infrarrojo y procesado por

el propio *chip* interno de la Kinect. En él se establece una comparación con un patrón a una distancia conocida y mediante triangulación se obtienen los valores de disparidad d en cada punto. Los valores de disparidad no se corresponden con valores directos de distancia, el cálculo de la misma requiere el conocimiento de ciertos parámetros de calibración. No obstante, los valores de disparidad que llegan a la Kinect están normalizados y se comportan de igual manera, siendo crecientes con la distancia.

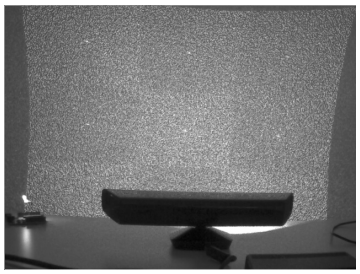


Figura A.14: Patrón de puntos láser proyectado.

(a) **(b)**

Figura A.15: Imágenes obtenida: (a) Imagen RGB (b) Imagen profundidad

Las imágenes RGB (Figura A.15a) que se obtienen son matrices 640x480 de tres canales, donde cada canal es un color (rojo,verde o azul). Cada pixel varía entre un valor de 0 y 255. Las imágenes de profundidad (Figura A.15b) que se obtienen son matrices 640x480 de un solo canal. Como se observa en la figura correspondiente, hay más o menos intensidad de gris, dependiendo de la distancia a la que se encuentren los objetos, pero hay zonas negras donde la cámara no ha podido obtener datos. Esos puntos son devueltos como *NaN (Not a Number)*, pero tratados posteriormente para convertirlos a un valor 0.

A.4.1. Obtención de las imágenes RGB y de profundidad

El primer objetivo es conseguir las imágenes de profundidad y RGB para, a partir de ellas, empezar con la detección de los recipientes. En un primer momento, se pensó en utilizar el software Matlab, ya que es un entorno de simulación más extendido, permitiendo la posible integración en otros sistemas. Sin embargo,

hay una gran desventaja al utilizar este software. Tras una búsqueda exhausta de información, se llegó a un punto: la imposibilidad de poder obtener los dos tipos de imágenes (RGB y profundidad) de manera simultánea. En una primera opción se intentó capturar los dos tipos en Matlab por medio de archivos *mex-files* (Matlab Executable). Un *mex-file* proporciona una interfaz entre Matlab y subrutinas escritas en C++ que, al ser compilado, permite al código que no pertenece a Matlab ser involucrado dentro de los programas y funciones de este sistema. De esta forma, habría que tomar primero una imagen (RGB) y posteriormente otra (profundidad), lo que aumentaría considerablemente los tiempos al obtener dos imágenes consecutivas del mismo tipo (estando entre 1.2 - 1.3 segundos), perdiendo uno de los objetivos que queremos conseguir, que es el obtener las imágenes a tiempo real.

Para conseguir nuestro objetivo, ROS nos proporciona las herramientas necesarias, permitiendo la publicación o subscripción a flujos de datos. Gracias a los drivers de OpenNI, podemos controlar la cámara Microsoft Kinect. Al conectarnos con ella, ROS nos ofrece una cantidad de nodos a los cuales nos podemos subscribir para poder recibir información de ellos, mostrándonos las opciones por su consola al hacer la petición de la lista de los nodos disponibles.

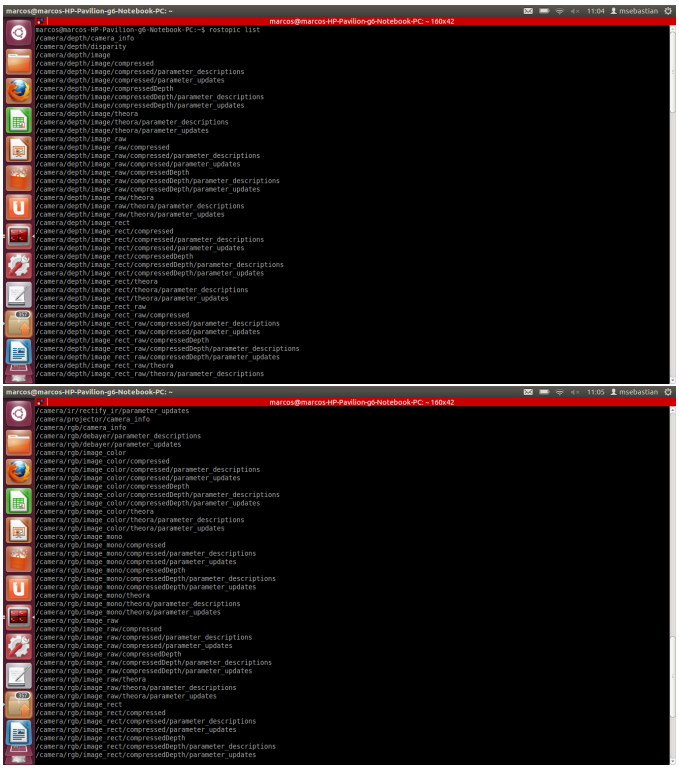


Figura A.16: Consola para el manejo de ROS con los nodos que ofrece la cámara para su posible subscripción

Como vemos, hay muchas posibilidades debido a los diversos nodos que se crea y que ROS nos ofrece: imagen de disparidad, comprimida, de color, en blanco y negro, etc...En nuestro programa creamos dos subscriptores: uno para la imagen

RGB y otro para la imagen de profundidad. De esta forma, podemos obtener el flujo de imágenes de una manera simultánea, cumpliendo uno de los objetivos del proyecto. Concretamente, nos subscribimos al nodo */camera/rgb/imagecolor* para conseguir la imagen de la cámara convencional, y nos subscribimos al nodo */camera/depth/imagerect* para conseguir la imagen de profundidad. ROS también nos permite crear distintos apartados para realizar el tratamiento de imágenes de forma simultánea. Así, en nuestro programa, se crean dos funciones distintas a raíz del programa principal: uno donde tratamos la imagen RGB, y otro donde manejamos la imagen de profundidad. Todo ello dentro del mismo proceso de toma de imagen. Así, cada vez que llegua uno de estos frames nuevos, son tratados para conseguir los objetivos que estamos buscando. De esta manera, los datos se pueden tratar a la vez y conseguir que todo sea a tiempo real.

Aunque teóricamente la cámara Microsoft Kinect trabaja a 30 frames por segundo, su conectividad por cable USB hace que esta cifra baje hasta prácticamente la mitad. En cada frame, el programa modifica la imagen y la muestra en una ventana de ROS, donde visualizamos el resultado. Las ventanas son dos, una para cada tipo de imagen y el tratamiento que le vamos dando. El tamaño de las imágenes recibidas es de 640x480. Éstas son convertidas a una matriz del entorno para poder recoger los datos y manejarlas con las herramientas tanto de ROS como de OpenCV y C++.

Apéndice B

Cálculo analítico del plano de referencia

Para calcular el plano en tres dimensiones se necesitan las coordenadas y distancia de al menos tres puntos pertenecientes a él. Dado que para la homografía hay que coger cuatro puntos que no incluyan tres colineales, se utilizan esos mismos puntos.

Tras la detección automática de las esquinas, conocemos los puntos P_1, P_2, P_3 y P_4, de los cuales se pueden obtener, por ejemplo, los vectores $\overrightarrow{u}$ y $\overrightarrow{v}$ como:

$$\begin{aligned} \overrightarrow{u} &= \overrightarrow{P_1P_2} = (u_1, u_2, u_3) = (x_2 - x_1, y_2 - y_1, z_2 - z_1) \\ \overrightarrow{v} &= \overrightarrow{P_1P_3} = (v_1, v_2, v_3) = (x_3 - x_1, y_3 - y_1, z_3 - z_1) \end{aligned} \tag{B.1}$$

Éstos forman parte del plano que vamos a calcular, al igual que P_1. Se supone un punto $X = (x, y, z)$ que pertenece al plano, por lo que sabremos que el vector $\overrightarrow{P_1X}$ es coplanario con $\overrightarrow{u}$ y $\overrightarrow{v}$:

$$\overrightarrow{P_1X} = \lambda\overrightarrow{u} + \mu\overrightarrow{v} \tag{B.2}$$

$$(x - x_1, y - y_1, z - z_1) = \lambda(u_1, u_2, u_3) + \mu(v_1, v_2, v_3) \tag{B.3}$$

$$(x, y, z) - (x_1, y_1, z_1) = \lambda(u_1, u_2, u_3) + \mu(v_1, v_2, v_3) \tag{B.4}$$

De forma que un punto X está en el plano si tiene solución el sistema:

$$\begin{cases} x - x_1 = u_1\lambda + v_1\mu \\ y - y_1 = u_2\lambda + v_2\mu \\ z - z_1 = u_3\lambda + v_3\mu \end{cases} \tag{B.5}$$

Para ello el sistema tiene que ser compatible determinado en las incógnitas λ y μ. Por tanto, el determinante de la matriz ampliada del sistema con la columna de los términos independientes tiene que ser igual a cero:

$$\begin{vmatrix} x - x_1 & u_1 & v_1 \\ y - y_1 & u_2 & v_2 \\ z - z_1 & u_3 & v_3 \end{vmatrix} = 0 \tag{B.6}$$

Desarrollando el determinante:

$$\begin{vmatrix} u_2 & v_2 \\ u_3 & v_3 \end{vmatrix} (x - x_1) - \begin{vmatrix} u_1 & v_1 \\ u_3 & v_3 \end{vmatrix} (y - y_1) + \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} (z - z_1) = 0 \tag{B.7}$$

Se obtiene:

$$A = \begin{vmatrix} u_2 & v_2 \\ u_3 & v_3 \end{vmatrix} \quad B = - \begin{vmatrix} u_1 & v_1 \\ u_3 & v_3 \end{vmatrix} \quad C = \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} \tag{B.8}$$

Sustituyendo:

$$A(x - x_1) + B(y - y_1) + C(z - z_1) = 0 \tag{B.9}$$

Operando y dando a D el valor

$$D = -Ax_1 - By_1 - Cz_1 \tag{B.10}$$

se obtiene la ecuación general del plano

$$Ax + By + Cz + D = 0 \tag{B.11}$$

con lo cual, dando valores de 1 a 640 en x y de 1 a 480 en y se obtiene una imagen 640×480 que es el plano calculado con esos cuatro puntos.

El proceso se repite tres veces más rotando los puntos de las esquinas empelados, de modo que en vez de coger P_1, P_2 y P_3 se prueba con P_2, P_3 y P_4; P_3, P_4 y P_1 y P_4, P_1 y P_2. Al final resulta el plano que menos error tiene de todas las combinaciones.

Apéndice C

Cálculo de homografías

C.1. Concepto de homografía

La necesidad de realizar una homografía viene dada principalmente por encontrar la correspondencia entre la imagen de profundidad y la imagen RGB en la calibración de la cámara. Una homografía es una transformación proyectiva que permite transformar la imagen tomada a un nuevo plano donde aparece corregida la distorsión ocasionada por la perspectiva y muestra la imagen en la situación ideal.

Una homografía es una aplicación invertible, que se denomina h, que transforma puntos situados en un plano a otro de modo que la colinealidad se mantiene. Es decir, si tres puntos de un plano están contenidos en una línea, lo siguen estando en el plano transformado. Más precisamente:

Una homografía (o proyectividad) es una aplicación invertible h desde P^2 a sí misma tal que tres puntos x_1, x_2 y x_3 coinciden en la misma línea sí y sólo sí $h(x_1)$, $h(x_2)$ y $h(x_3)$ también lo hacen.

Una definición equivalente en forma algebraica es la siguiente:

> *una aplicación* $h: \quad P^2 \to P^2$ *es una homografía sí y sólo sí existe una matriz* H *no singular 3x3 que para cada punto de* P^2 *representado como vector de coordenadas homogéneas* x *se cumple que* $h(x) = Hx$.

Que un punto del espacio bidimensional esté descrito en coordenadas homogéneas significa que está definido por tres coordenadas. De tal modo un punto de dimensiones (x, y) se representa por $(\frac{x}{w}, \frac{y}{w}, w)$ donde por simplicidad se considera en una primera instancia que $w = 1$. Dado que estamos trabajando con coordenadas homogéneas, la matriz H también lo será. Al igual que ocurre con la representación homogénea de un punto, sólo el ratio entre los elementos de la matriz es significante, luego por haber 8 ratios independientes entre los 9 elementos de H, hay 8 grados de libertad a la hora de calcular la matriz.

Sabiendo eso, se puede realizar la transformación proyectiva de las coordenadas de un punto $p(x, y)$ a un punto $p'(x', y')$. La solución no homogénea es:

$$\begin{aligned} x' = \frac{x'_1}{x'_3} &= \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + h_{33}} \\ y' = \frac{x'_2}{x'_3} &= \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + h_{33}} \end{aligned} \tag{C.1}$$

De modo que se generan dos ecuaciones por cada punto $p(x, y)$. Si se conoce el punto $p'(x', y')$ únicamente se tienen como incógnitas del sistema a los elementos de la matriz H. Siendo que dicha matriz tiene únicamente ocho grados de libertad, teniendo cuatro puntos conocidos en ambas proyecciones se pueden obtener las ocho ecuaciones necesarias para resolver y calcular H. La única restricción a tener en cuenta es que esos cuatro puntos han de estar en una «posición general», es decir, que entre ellos no haya tres alineados en ningún caso.

Una vez obtenida H se puede invertir para pasar indistintamente de un plano a otro.

C.2. Procedimiento

Como se ha visto en el subapartado anterior, se necesita calcular la matriz H. Hay muchos algoritmos para ello, el que se va a comentar es el de transformación linear directa (*Direct Linear Transformation algorithm*, o DLT) [27].

Ello requiere partir de un conjunto de cuatro puntos que se conoce que se corresponden en ambos planos proyectivos, $\mathbf{x}_i \leftrightarrow \mathbf{x}'_i$. La transformación la da la ecuación $\mathbf{x}'_i = H\mathbf{x}_i$, como ya se ha comentado. Esta ecuación se puede expresar como el producto vectorial tal como $\mathbf{x}'_i \times H\mathbf{x}_i = 0$.

La j-ésima fila de la matriz H se llamará $\mathbf{h}^{jT}$, tal que:

$$H\mathbf{x}_i = \begin{pmatrix} \mathbf{h}^{1T}\mathbf{x}_i \\ \mathbf{h}^{2T}\mathbf{x}_i \\ \mathbf{h}^{3T}\mathbf{x}_i \end{pmatrix} \tag{C.2}$$

Escribiendo $\mathbf{x}'_i = (x'_i, y'_i, w'_i)^T$, el producto vectorial se puede dar como:

$$\mathbf{x}'_i \times H\mathbf{x}'_i = \begin{pmatrix} y'_i\mathbf{h}^{3T}\mathbf{x}_i - w'_i\mathbf{h}^{2T}\mathbf{x}_i \\ y'_i\mathbf{h}^{1T}\mathbf{x}_i - w'_i\mathbf{h}^{3T}\mathbf{x}_i \\ y'_i\mathbf{h}^{2T}\mathbf{x}_i - w'_i\mathbf{h}^{1T}\mathbf{x}_i \end{pmatrix} \tag{C.3}$$

Y dado que $\mathbf{h}^{jT}\mathbf{x}_i = \mathbf{x}\mathbf{h}^j$ para $j = 1, ..., 3$, da un conjunto de tres ecuaciones de los términos de H:

$$\begin{bmatrix} \mathbf{0}^T & -w'_i\mathbf{x}_i^T & y'_i\mathbf{x}_i^T \\ -w'_i\mathbf{x}_i^T & \mathbf{0}^T & -x'_i\mathbf{x}_i^T \\ -y'_i\mathbf{x}_i^T & x'_i\mathbf{x}_i^T & \mathbf{0}^T \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix} = 0. \tag{C.4}$$

Que tiene la forma $A_i\mathbf{h} = \mathbf{0}$, donde A_i es una matriz 3x9 y $\mathbf{h}$ es un vector de 9 componentes que forman la matriz H.

$$\mathbf{h} = \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix}, \quad H = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \tag{C.5}$$

Donde h_i es el i-ésimo elemento de $\mathbf{h}$.

De las tres ecuaciones obtenidas, solo dos de ellas son linealmente independientes, por lo que, como ya vimos, es necesario usar cuatro puntos para obtener 8 ecuaciones que nos permitan calcular la H completa. Se puede eliminar la tercera fila de la ecuación (C.4) y obtener:

$$\begin{bmatrix} \mathbf{0}^T & -w'_i\mathbf{x}_i^T & y'_i\mathbf{x}_i^T \\ -w'_i\mathbf{x}_i^T & \mathbf{0}^T & -x'_i\mathbf{x}_i^T \end{bmatrix} \begin{pmatrix} \mathbf{h}^1 \\ \mathbf{h}^2 \\ \mathbf{h}^3 \end{pmatrix} = 0. \tag{C.6}$$

que origina una matriz A_i de dimensiones 2x9.

Apéndice D

Android

D.1. ¿Qué es Android?

Android es un sistema operativo pensado inicialmente para móviles. Sin embargo, lo que le hace diferente es que está basado en Linux, un núcleo de sistema operativo libre, gratuito y multiplataforma.Este sistema permite programar aplicaciones en una variación de Java, proporcionando las interfaces necesarias para desarrollar aplicaciones que accedan a las distintas funciones del teléfono (GPS, llamadas, agenda, etc...) de una forma muy sencilla mediante este lenguaje orientado a objetos. Esta sencillez, junto con las herramientas de programación gratuitas, hace que haya una gran cantidad de aplicaciones disponibles, ya que cualquier usuario puede actuar como desarrollador de aplicaciones para luego funcionar sobre este sistema operativo.

Una de las mejores características es que es completamente libre. No hay que pagar nada por programar en este sistema, ni para incluir en ningún dispositivo que lo tenga (móviles, tablets, ordenadores). Por ello, es muy popular entre fabricantes y desarroladores, ya que los costes para lanzar una aplicación son muy bajos. Cualquiera puede bajarse el código fuente, modificarlo, inspeccionarlo o compilarlo. Esto proporciona seguridad a los usuarios, ya que algo que es abierto permite detectar fallos más rápidamente. Y también a los fabricantes, ya que pueden adaptar mejor el sistema operativo a los terminales.

La estructura del sistema operativo Android se compone de aplicaciones que se ejecutan en un framework Java de aplicaciones orientadas a objetos sobre el núcleo de las bibliotecas de Java en una máquina virtual Dalvik con compilación en tiempo de ejecución. Las bibliotecas escritas en lenguaje C incluyen un administrador de interfaz gráfica (surface manager), un framework OpenCore, una base de datos relacional SQLite, una Interfaz de programación de API gráfica OpenGL ES 2.0 3D, un motor de renderizado WebKit, un motor gráfico SGL, SSL y una biblioteca estándar de C Bionic. El sistema operativo está compuesto por 12 millones de líneas de código, incluyendo 3 millones de líneas de XML, 2,8 millones de líneas de lenguaje C, 2,1 millones de líneas de Java y 1,75 millones

de líneas de C++.

D.1.1. Historia

Inicialmente, fue desarrollado por Android Inc., y cuando aún era desconocido, fue comprado por Google en 2005. Hasta noviembre de 2007 sólo hubo rumores, pero en esa fecha se lanzó la Open Handset Alliance, que agrupaba a muchos fabricantes de teléfonos móviles, chipsets y Google y se proporcionó la primera versión de Android, junto con el SDK para que los programadores empezaran a crear sus aplicaciones para este sistema. Google liberó la mayoría del código de Android bajo la licencia Apache, una licencia libre y de código abierto.

Aunque los inicios fueran un poco lentos, debido a que se lanzó antes el sistema operativo que el primer móvil, se ha colocado rápidamente como el sistema operativo para móviles más vendido del mundo, alcanzando su auge en 2010. Las unidades vendidas de teléfonos inteligentes con Android se ubican en el primer puesto en los Estados Unidos, en el segundo y tercer trimestres de 2010, con una cuota de mercado de 43,6 % en el tercer trimestre. A nivel mundial alcanzó una cuota de mercado del 50,9 % durante el cuarto trimestre de 2011, más del doble que el segundo sistema operativo (iOS de Apple, Inc.) con más cuota. En la actualidad, Android ha transcendido los teléfonos móviles para trascender a dispositivos más grandes, como las tablets, ordenadores, e incluso consolas de juego.

Figura D.1: Diferentes dispositivos donde android ya está disponible

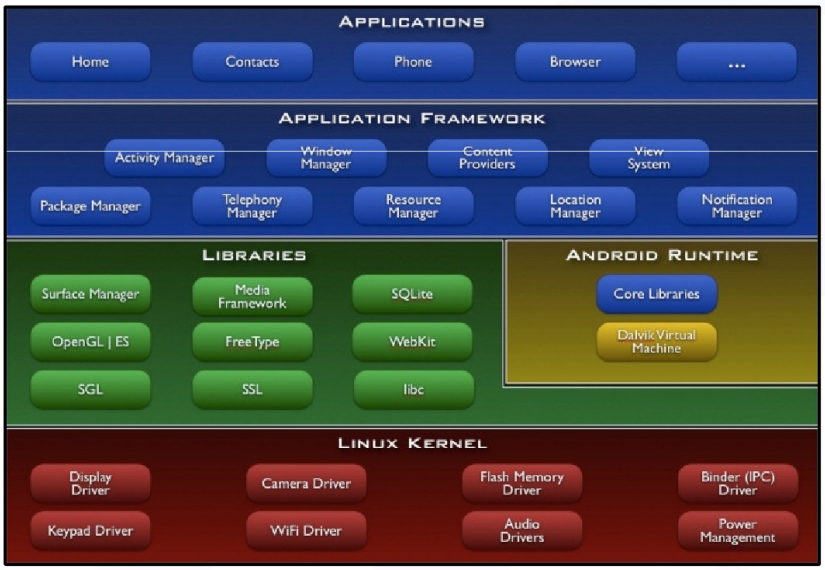


Figura D.2: Esquema de la arquitectura Android

D.1.2. Arquitectura Android

- **Aplicaciones:** Este nivel contiene, tanto las incluidas por defecto de Android como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las API y librerías de los niveles anteriores.

- **Framework de aplicaciones:** Representa fundamentalmente el conjunto de herramientas de desarrollo de cualquier aplicación. Toda aplicación que se desarrolle para Android, ya sean las propias del dispositivo, las desarrolladas por Google o terceras compañías, o incluso las que el propio usuario cree, utilizan el mismo conjunto de API y el mismo framework, representado por este nivel.

- **Librerías:** La siguiente capa se corresponde con las librerías utilizadas por Android. Éstas han sido escritas utilizando C/C++ y proporcionan a Android la mayor parte de sus capacidades más características. Junto al núcleo basado en Linux, estas librerías constituyen el corazón de Android.

- **Tiempo de ejecución de android:** Al mismo nivel que las librerias de Android se sitúa el entorno de ejecución. Éste lo constituyen las Core Libraries, que son librerias con mulititud de clases Java y la máquina vistual Dalvik.

- **Núcleo Linux:** Android utiliza el núcleo de Linux 2.6 como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los drivers necesarios para que cualquier componente hardware

pueda ser utilizado mediante las llamadas correspondientes. Siempre que un fabricante incluye un nuevo elemento de hardware, lo primero que se debe realizar para que pueda ser utilizado desde Android es crear las librerias de control o drivers necesarios dentro de este kernel de Linux embebido en el propio Android.

D.1.3. Características

- **Diseño de dispositivo:** La plataforma es adaptable a pantallas de mayor resolución, VGA, biblioteca de gráficos 2D, biblioteca de gráficos 3D basada en las especificaciones de la OpenGL ES 2.0 y diseño de teléfonos tradicionales.
- **Almacenamiento:** SQLite, una base de datos liviana, que es usada para propósitos de almacenamiento de datos.
- **Conectividad:** Android soporta las siguientes tecnologías de conectividad: GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, HSDPA, HSPA+, NFC y WiMAX.
- **Mensajería:** SMS y MMS son formas de mensajería.
- **Navegador web:** El navegador web incluido en Android está basado en el motor de renderizado de código abierto WebKit, emparejado con el motor JavaScript V8 de Google Chrome.
- **Soporte multimedia:** Android soporta los siguientes formatos multimedia: WebM, H.263, H.264 (en 3GP o MP4), MPEG-4 SP, AMR, AMR-WB (en un contenedor 3GP), AAC, HE-AAC (en contenedores MP4 o 3GP), MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF y BMP.
- **Soporte para hardware adicional:** Android soporta cámaras de fotos, de vídeo, pantallas táctiles, GPS, acelerómetros, giroscopios, magnetómetros, sensores de proximidad y de presión, sensores de luz, gamepad, termómetro, aceleración por GPU 2D y 3D.
- **Entorno de desarrollo:** Incluye un emulador de dispositivos, herramientas para depuración de memoria y análisis del rendimiento del software.
- **Multi-táctil:** Android tiene soporte nativo para pantallas capacitivas con soporte multi-táctil que inicialmente hicieron su aparición en dispositivos como el HTC Hero. La funcionalidad fue originalmente desactivada a nivel de kernel (posiblemente para evitar infringir patentes de otras compañías). Más tarde, Google publicó una actualización para el Nexus One y el Motorola Droid que activa el soporte multi-táctil de forma nativa.
- **Bluetooth**

- **Videollamada**
- **Multitarea:** Multitarea real de aplicaciones está disponible, es decir, las aplicaciones que no estén ejecutándose en primer plano reciben ciclos de reloj.
- **Características basadas en voz:** La búsqueda en Google a través de voz está disponible como .Entrada de Búsqueda"desde la versión inicial del sistema.
- **Tethering:** Android soporta tethering, que permite al teléfono ser usado como un punto de acceso alámbrico o inalámbrico (todos los teléfonos desde la versión 2.2, no oficial en teléfonos con versión 1.6 o inferiores mediante aplicaciones disponibles en Google Play.

Apéndice E

ROS

ROS (Robot Operating System, o Sistema Operativo Robótico en español) es un framework para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo. Provee librerías y herramientas para ayudar a los desarrolladores de software a crear aplicaciones para robots. También provee abstracción de hardware, controladores de dispositivos, librerías, herramientas de visualización, comunicación por mensajes, administración de paquetes y más. ROS está bajo la licencia open source, BSD.ROS se desarrolló originalmente en 2007 bajo el nombre de switchyard por elLaboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford . Desde 2008, el desarrollo continúa primordialmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado. ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. La librería está orientada para un sistema UNIX (Ubuntu (Linux) es el sistema soportado aunque también se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows considerados como experimentales). Se puede programar en C++ o Python, a través de una serie de módulos.

ROS tiene dos partes básicas: la parte del sistema operativo, ros, como se ha descrito anteriormente y ros-pkg, una suite de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o en inglés stacks)

que implementan la funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc. ROS es software libre bajo términos de licencia BSD. Esta licencia permite libertad para uso comercial e investigador. Las contribuciones de los paquetes en ros-pkg están bajo una gran variedad de licencias diferentes.

En este proyecto, se va a utilizar la parte de visión robótica, que es la que nos va a proporcionar los nodos para poder conseguir las imágenes de profundidad y de RGB.

Las áreas que incluye ROS son:

- Un nodo principal de coordinación.
- Publicación o subscripción de flujos de datos: imágenes, estéreo, láser, control, actuador, contacto, etc.
- Multiplexación de la información.
- Creación y destrucción de nodos.
- Los nodos están perfectamente distribuidos, permitiendo procesamiento distribuido en múltiples núcleos, multiprocesamiento, GPUs y clústeres.
- Login.
- Parámetros de servidor.
- Testeo de sistemas.

Las áreas que incluirán las aplicaciones de los paquetes de ROS son:

- Percepción
- Identificación de Objetos
- Segmentación y reconocimiento
- Reconocimiento facial
- Reconocimiento de gestos
- Seguimiento de objetos
- Egomoción
- Comprensión de movimiento
- Estructura de movimientos (SFM)

- Visión estéreo: percepción de profundidad mediante el uso de dos cámaras
- Movimientos
- Robots móviles
- Control
- Planificación
- Agarre de objetos

Apéndice F

Transmisión de datos y protocolos de comunicación

F.1. Método de transmisión de datos: Sockets

Socket, como su propio nombre indica (enchufe) es un método de comunicación entre un servidor y un cliente. De esta forma, dos programas pueden intercambiarse un flujo de datos de una forma fiable y ordenada. Gracias a ellos, se forma una especie de canal de conexión, que constituye el mecanismo para la entrega de paquetes de datos. Aunque existen varios tipos, nos centramos en los sockets de TCP/IP, que ofrece los protocolos de TCP y UDP.

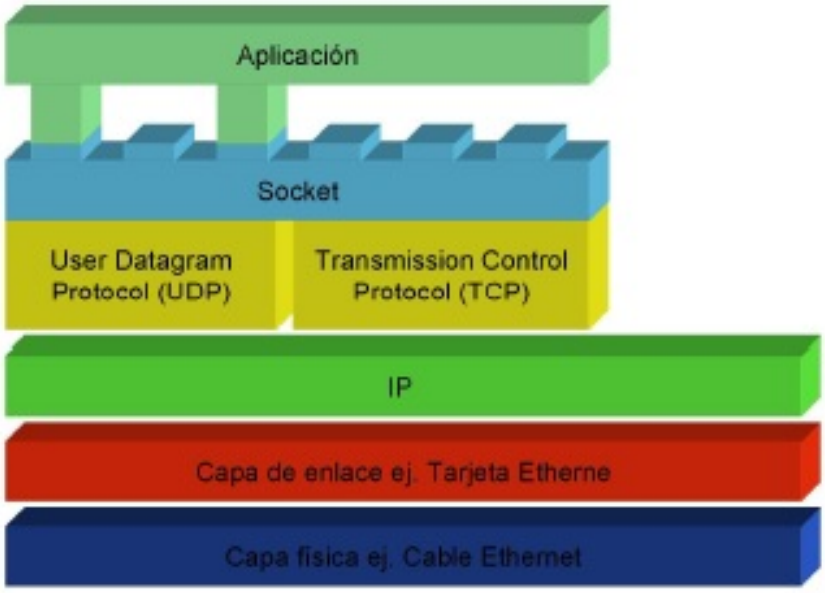


Figura F.1: Situación de los sockets en capas

La forma en la que intercambian información es muy similar al proceso de lectura y escritura de un fichero. Así, lo que queramos mandar es lanzado por el socket de salida, mientras que la información es recibida por el socket destino.

Un socket se identifica siempre por la dirección IP y por el número de puerto, que debemos especificar al principio del programa. De esta forma, quedan identificados el socket origen y el socket destino. La particularidad que tienen frente a otros mecanismos de comunicación entre procesos (IPC - Inter-Process Communication) es que posibilitan la comunicación aun cuando ambos procesos estén corriendo en distintos sistemas unidos mediante una red. De hecho, el API de sockets es la base de cualquier aplicación que funcione en red puesto que ofrece una librería de funciones básicas que el programador puede usar para desarrollar aplicaciones en red. Por esta razón, los sockets ofrecen la mejor solución para realizar la comunicación entre dos lenguajes de programación distintos como son C++ y Android.

Los sockets para TCP/IP permiten la comunicación de dos procesos que estén conectados a través de una red TCP/IP. En una red de este tipo, cada máquina está identificada por medio de su dirección IP que debe ser única. Sin embargo, en cada máquina pueden estar ejecutándose múltiples procesos simultáneamente. Cada uno de estos procesos se asocia con un número de puerto, para poder así diferenciar los distintos paquetes que reciba la máquina.

Es importante mencionar que a la aplicación creada en Android para el dispositivo, hay que darle unos permisos a la hora de poder comunicarse, para poder acceder al Wi-fi y a internet. Estos permisos son los siguientes:

- android.permission.ACCESS_WIFI_STATE: Permite a las aplicaciones acceder a la información referente a las redes Wi-Fi.
- android.permission.INTERNET: Permite a las aplicaciones abrir redes de sockets para la comunicación.
- android.permission.ACCESS_NETWORK_STATE: Permite a las aplicaciones acceder a la información sobre las redes.
- android.permission.CHANGE_WIFI_MULTICAST_STATE: Permite a las aplicaciones entrar en el modo Wi-Fi multidifusión.

F.2. Protocolos de comunicación

Como hemos mencionado anteriormente, los dos protocolos más usados son TCP y UDP, siendo los dos pertenecientes a la capa de transporte. En el caso de los sockets, estos son nombrados Socket Stream si se trata del protocolo TCP, o Socket Datagram si se trata del protocolo UDP.

A grandes rasgos, las características del protocolo TCP son las siguientes:

- Protocolo orientado a conexión
- Secuenciado

- Sin duplicación de paquetes
- Libre de errores

De esta forma, proporciona un transporte fiable de información entre aplicaciones, realizando retransmisiones ante la pérdida de paquetes y asegurando que los paquetes llegan de forma ordenada, ya que, como su propio nombre indica (Transmission Control Protocol) el propio protocolo realiza las tareas de control. Sin embargo, esta fiabilidad repercute en la eficiencia del sistema, disminuyéndola, ya que para poder gestionar todas las tareas anteriores se tiene que añadir mucha información a los paquetes que se quieren enviar. Como estos paquetes que se quieren enviar tienen un tamaño máximo, cuanta más información sobre el protocolo tengamos que añadir, menor espacio para la información útil podremos disponer. De hecho, el segmento TCP tiene una sobrecarga de 20 bytes en cada segmento, mientras que UDP sólo añade 8 bytes.

Por otro lado, las características del protocolo UDP son las siguientes:

- No orientado a conexión
- Flujo de datos bidireccional
- Los paquetes no tienen por qué llegar ordenados
- Puede haber pérdidas de paquetes o llegar con errores

La transmisión que proporciona UDP no es del todo fiable, ya que apenas añade la información necesaria al paquete para la comunicación extremo a extremo. Lo utilizan aplicaciones como NFS o para copiar ficheros entre ordenadores remotos, pero sobre todo se emplea en la transmisión de audio y video, o para tareas de control a través de una red. Es el mejor protocolo para realizar programas que requieran transmitir a tiempo real, ya que no produce retardos para establecer la conexión, no necesita mantenerse en estado de conexión y por lo tanto no realiza un seguimiento de estos datos. De esta manera, una aplicación con protocolo UDP es capaz de soportar más clientes activos que en una aplicación con TCP. Se podría decir que sirve como "multiplexor/demultiplexor"para enviar y recibir datagramas, usando puertos para dirigir los datagramas como se muestra en la Figura F.2.

Por todas las características que se han explicado, el protocolo que mejor se adapta a lo que queremos hacer es el protocolo UDP, ya que en nuestro proyecto va a ser necesario transportar imágenes y transmitir flujos de datos a tiempo real. El uso de UDP es utilizado siempre cuando resulta más importante transmitir con velocidad que garantizar el hecho de que lleguen absolutamente todos los bytes.

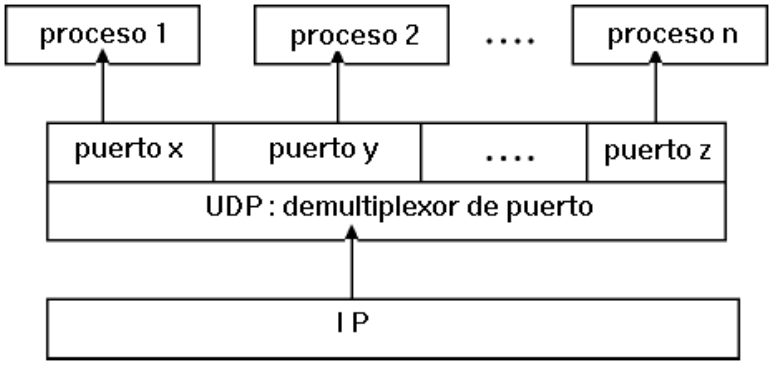


Figura F.2: Esquema del socket actuando como multiplexor/demultiplexor

F.3. Comunicación: Servidor

De una forma resumida y concisa, el servidor es el programa que permanece pasivo a la espera de que alguien solicite conexión con él, normalmente, para pedirle algún dato. Cliente es el programa que solicita la conexión para, normalmente, pedir datos al servidor. El servidor será programado en el PC en lenguaje C++, mientras que el cliente se va a programar en Android. En nuestro caso, al tratarse de una comunicación UDP, no se realiza una conexión permanente, sino que el servidor esperará un paquete del cliente, recogiendo la dirección IP de la que proviene, para poder responder a ese terminal.

Los pasos que debe seguir un programa servidor son los siguientes:

- Abrir un socket con la función *socket()*.
- Asociar el socket a un puerto con *bind()*.
- Leer mensaje con *recvfrom()*.
- Responder mensaje con *sendto()*.

F.3.1. Abrir socket

Se abre el socket de la forma habitual, con la función *socket()*. Esto simplemente nos devuelve un descriptor de socket, que todavía no funciona ni es útil. Este descriptor es usado para poder asociar después a la comunicación.

F.3.2. Asociar el socket con un puerto

Con el sistema operativo Ubuntu, se disponen de 65536 puertos para hacer conexiones con sockets, numerados de 0 a 65535. Sin embargo, del 0 al 1023 están reservados al sistema. Por ello, tenemos que escoger un puerto que vaya del 1024

al 65536. En nuestro caso, el puerto elegido es el 7000, aunque le podríamos haber asignado cualquier otro.

Una vez hemos abierto un socket, se utiliza la función *bind()*, que sirve para comunicar al sistema operativo que puerto es el que queremos utilizar para la transmisión. Para decir que puerto queremos coger, tenemos dos opciones. Una de ellas es indicar directamente cuál es el puerto que queremos utilizar. La otra es indicar con permisos de root en el fichero /etc/services el puerto que queremos atender, y darle un nombre a este puerto. De esa forma, el sistema operativo sabría que ese puerto está reservado. Sin embargo, por simplicidad utilizaremos la primera opción, indicando directamente el puerto por el que queremos realizar la comunicación, que deberá ser el mismo que indicaremos en la aplicación Android.

De esta forma tenemos los parámetros definidos para comenzar la comunicación. Primero, debemos de recibir la petición del cliente que se quiere conectar al servidor. Posteriormente, una vez que sepamos con quién nos vamos a comunicar, podemos realizar la comunicación bidireccional.

F.3.3. Recibir mensaje en el servidor

Para recibir mensajes del servidor, la función para los sockets UDP es *recvfrom()*. Esta función es "bloqueante", por lo que el programa se queda esperando a recibir una petición y no continua. Esto supone un problema, ya que la toma de imágenes debe de ser continua, y no puede haber nada en el programa que bloquee la llegada de frames. Por ello, la solución pasa por realizar dos procesos: un proceso que toma las imágenes tanto de profundidad como RGB, mientras que el otro proceso es el que haría de servidor. De esta forma, aunque el proceso del servidor se quede bloqueado esperando la llegada de un socket, el proceso de toma de imágenes es continuo y nunca deja de tomar los frames necesarios para realizar la detección y seguimiento de los recipientes. Aunque debemos tener en cuenta que hay que tener datos en común entre estos dos procesos, como las imágenes y la potencia que le pasemos cada vez que variemos el fuego de uno de los recipientes.

La imagen RGB que nosotros construimos para pasar al dispositivo móvil se trata en el mismo proceso de toma de imágenes, ya que a cada imagen que se toma de la cámara, ésta es tratada y mostrada por una ventana. Así que debe haber una comunicación entre los dos procesos, de tal forma que cuando el proceso del servidor necesite la imagen, el proceso de toma de imágenes se la entregue para su posterior envío.

Para crear otro proceso en C++ hay dos opciones: o crear un proceso hijo (*fork*) o crear un hilo (*thread*). En un primer momento se probó a crear un proceso hijo mediante la función *fork*, sin embargo, esta opción fue desechada ya que, aunque el hijo hereda las mismas variables que el padre, cada uno las modifica de forma individual. Por ejemplo, si una variable es cambiada de valor por el hijo, en éste proceso tendrá un nuevo valor, sin embargo en el proceso padre

no habrá variado.

Por ello, la solución por la que se opta es realizar un hilo, a través de la función *thread*. De esta forma, podemos compartir variables y matrices, permitiendo una comunicación entre el proceso de toma de imágenes y el servidor, de tal forma que cualquiera de los dos pueda estar al tanto de las potencias y las imágenes.

A la función *recvfrom()*, le pasamos una estructura vacía, pero que es de vital importancia: *struct sockaddr*. Esta estructura recoge los datos del cliente que nos ha enviado el mensaje. De esta forma, podemos identificar al otro programa y responderle, produciendo así la comunicación bidireccional.

Si queremos identificar errores, esta función nos devuelve -1 en caso de que se haya producido un error. Sin embargo, si funciona correctamente, nos devuelve el número de bytes leídos.

F.3.4. Respuesta al cliente

A la hora de responder al cliente, la función utilizada es *sendto()*. Esta función admite los mismos parámetros que la función *recvfrom()*, aunque como es obvio, la estructura sockaddr no está vacía, sino que contiene los datos que ha recibido del cliente.

Al igual que *recvfrom()* devuelve un -1 en caso de error, devuelve el número de bytes escritos en el caso de que la comunicación haya sido correcta.

F.4. Comunicación: Cliente

El cliente se va a programar en Android. Esta aplicación necesita conocer desde un primer momento la dirección del servidor y el puerto de comunicación, para así poder enviarle un paquete de tal forma que éste conozca de donde proviene el paquete y enviarle una respuesta.

Los pasos a seguir son los siguientes:

- Creación del socket
- Creación del paquete de datagrama, ya sea para recibir o para enviar.
- Enviar paquete
- Recibir paquete

F.4.1. Creación socket

En un primer lugar, se utiliza la función *Datagram socket* para crear un socket que permita la comunicación, tanto para enviar como para recibir. En este caso,

no hay que pasarle ningún parámetro, sólo abrirlo, ya que la dirección del servidor y el puerto de comunicación se indican cuando se crea el paquete que vamos a enviar.

F.4.2. Creación paquete

Aunque el concepto es el mismo tanto para la creación de un paquete para enviar datos como para recibirlos, hay diferencias entre uno y otro. Para la creación de los paquetes UDP, se utiliza la función *Datagram Packet*, tanto para el envío como para recibir.

En el caso del paquete a enviar hay que introducirle varios parámetros. En primer lugar, se introduce el array de bytes que se quiere enviar y su longitud. Y posteriormente, se le indica la dirección del servidor y el puerto por donde se va a producir la comunicación. De esta manera, el paquete se envía al servidor, que escucha por el puerto establecido en común, y que recoge la dirección del cliente, para posteriormente enviarle información a esa misma dirección.

En el caso del paquete recibido, simplemente se le indica el buffer donde almacenar la llegada de datos y la longitud del mismo.

F.4.3. Enviar paquete

A la hora de enviar un paquete de datagrama, utilizamos la función *socket.send()*, siendo *socket* el nombre con el que hayamos llamado en un principio a nuestro socket, e introduciéndole el nombre del paquete que hemos creado para el envío.

F.4.4. Recibir paquete

En el caso de la recepción, utilizamos *socket.receive()*, introduciéndole el paquete que hemos creado para recibir la información. Aunque hemos utilizado un paquete para la recepción, hay que tener en cuenta que los datos se encuentran almacenados en el buffer que hemos creado.

Sin embargo, en este caso nos surge un problema. Al igual que ocurre con el servidor programado en C++, la función *receive* es bloqueante, por lo que el programa no avanza si se queda bloqueada. En Android, si hay una función bloqueante en la parte principal de una actividad, el programa deja de responder. Por lo tanto, se crea un *thread* que separe el proceso cliente de la actividad principal, para que así pueda operar de una forma secundaria manteniendo la parte principal del programa sin ninguna función que la pueda bloquear. Para hacer cambios en la interfaz de esa actividad, utilizaremos un *handler*, función que permite comunicar este proceso creado para el cliente con la parte principal

de la actividad. Esto sirve de mucha utilidad a la hora de poder mostrar la imagen de la placa en el dispositivo.

Apéndice G

Métodos de detección

Para la detección de contornos existen una variedad de métodos. Los más utilizados son el método de Sobel[19], el de Prewitt[20], el de Roberts[21] o el de Canny[22]. Los tres primeros se basan en los distintos niveles de grises que hay en una imagen de blanco y negro. Aplicando máscaras de convolución a la imagen, se resaltan los cambios bruscos en la intensidad, lo que se corresponderá con los contornos que realmente estamos buscando.Cada uno tiene sus ventajas y desventajas.

G.1. Método de Roberts

El método de Roberts utiliza una máscara más sencilla. Presenta como gran desventaja que considera muy pocos píxeles de entrada para hacer la aproximación, lo que provoca que sea muy sensible al ruido y nos permite solamente marcar los puntos de borde, es decir, su localización, pero no la orientación de los mismos. No obstante, esta misma desventaja lo convierte en un operador muy simple que trabaja muy bien con imágenes binarias y con una gran velocidad de cómputo.

Ventajas:

- Buena respuesta en bordes horizontales y verticales.
- Buena localización.
- Simpleza y rapidez de cálculo.

Desventajas:

- Mala respuesta en bordes diagonales.
- Sensible al ruido.
- Empleo de máscaras pequeñas.

- No da información acerca de la orientación del borde.
- Anchura del borde de varios píxeles.

G.2. Método de Sobel

El método de Sobel utiliza la misma máscara que Prewitt, sin embargo, el ruido es menor ya que posee un suavizado que reduce el ruido, permitiendo mejores medidas. Sobel es el operador usado más comúnmente y en la práctica proporciona una buena detección de bordes diagonales.

Ventajas:

- Buena respuesta en bordes horizontales y verticales.
- Diversidad de tamaños en las máscaras.
- Proporcionan un suavizado además del efecto de derivación

Desventajas:

- Mala respuesta en bordes diagonales.
- Lentitud de cálculo.
- No da información acerca de la orientación del borde.
- Anchura del borde de varios píxeles.

G.3. Método de Prewitt

El método de Prewitt utiliza una máscara más compleja que el de Roberts, por lo que aunque no sea tan rápido computacionalmente, permite unos mejores resultados en la detección de los bordes. A diferencia del operador de Sobel, el operador de Prewitt proporciona una mejor detección de los bordes verticales y horizontales en comparación con los bordes diagonales. No obstante, en la práctica no se aprecia una gran diferencia entre ambos.

Ventajas:

- Buena respuesta en bordes horizontales y verticales.
- Poco sensible al ruido.
- Proporciona la magnitud y dirección del borde.

Desventajas:

- Mala respuesta en bordes diagonales.
- Lentitud de cálculo.
- Anchura del borde de varios píxeles.

G.4. Método de Canny

En 1986, Canny propuso un método para la detección de bordes que ofrecía mejores resultados que los métodos vistos anteriormente aunque presentaba una mayor complejidad computacional.

El método de Canny se basa en tres criterios principales:

- El criterio de detección, que expresa el hecho de evitar la eliminación de bordes importantes así como no suministrar falsos bordes.
- El criterio de localización, que establece que la distancia entre la posición real y la posición localizada para el borde debe ser minimizada.
- El criterio de respuesta única, que establece la necesidad de que el detector retorne un único punto por cada punto de borde verdadero. Esto implica que el detector no debe encontrar múltiples píxeles de borde donde solo existe uno.

Uno de los métodos relacionados con la detección de bordes es el uso de la primera derivada, que utiliza el valor cero en todas las regiones donde no varía la intensidad y tiene un valor constante en toda la transición de intensidad. Por lo tanto, un cambio de intensidad se manifiesta como un cambio brusco en la primera derivada, característica que es utilizada para detectar un borde, y en la que se basa el algoritmo de Canny.

El algoritmo de Canny consta de tres grandes pasos:

- **Obtención del gradiente:** Para la obtención del gradiente, lo primero que se realiza es la aplicación de un filtro gaussiano a la imagen original con el objetivo de suavizar dicha imagen y conseguir la eliminación del ruido que pueda existir. Sin embargo, hay que tener cuidado de no aplicar un suavizado excesivo, puesto que se podrían perder ciertos detalles de la imagen y provocar que el resultado final no fuese el esperado. Una vez que se suaviza la imagen, para cada píxel se obtiene la magnitud y módulo(orientación) del gradiente, obteniendo así dos imágenes.

- **Supresión no máxima al resultado del gradiente:** Las dos imágenes generadas en el paso anterior sirven de entrada para generar una imagen con los bordes adelgazados. El procedimiento es el siguiente: se consideran cuatro direcciones identificadas por las orientaciones de 0º, 45º, 90º y 135º con respecto al eje horizontal. Para cada píxel se encuentra la dirección que mejor se aproxime a la dirección del ángulo de gradiente. Posteriormente se observa si el valor de la magnitud de gradiente es más pequeño que al menos uno de sus dos vecinos en la dirección del ángulo obtenida en el paso anterior. De ser así, se asigna el valor 0 a dicho píxel, en caso contrario se asigna el valor que tenga la magnitud del gradiente. La salida de este segundo paso es una imagen con los bordes adelgazados después de realizarse la supresión no máxima de puntos de borde.

- **Histéresis de umbral a la supresión no máxima:** La imagen que ha sido obtenida en el paso anterior suele contener máximos locales creados por el ruido. Una solución para eliminar dicho ruido es la denominada histéresis del umbral. El proceso de histéresis consiste en tomar la imagen obtenida en el paso anterior, obtener la orientación de los puntos de borde de la imagen y tomar dos umbrales de forma que el primero sea más pequeño que el segundo. Para cada punto de la imagen se debe localizar el siguiente punto de borde no explorado que sea mayor que el segundo valor de umbral. A partir de dicho punto, se siguen las cadenas de máximos locales conectados en ambas direcciones perpendiculares a la normal del borde siempre que sean mayores que el primer valor de umbral. De esta forma, se marcan todos los puntos explorados y se almacena la lista de todos los puntos en el contorno conectado. Es de este modo como se logra eliminar, con este paso, las uniones en forma de Y de los segmentos que confluyen en un determinado punto.

Como conclusión, se puede indicar que el algoritmo de Canny tiene como principal ventaja su gran adaptabilidad para poder ser aplicado a diversos tipos de imágenes, además de no disminuir su rendimiento ante la presencia de ruido en la imagen original. Sin embargo, tiene como desventaja el hecho de que al realizar el suavizado de la imagen se pueden difuminar ciertos bordes aunque con eso se consiga reducir el ruido.

Este algoritmo es uno de los mejores métodos para la detección de bordes, el cual aplica métodos de diferencias finitas basado en la primera derivada y cuya popularidad se debe, además de a sus buenos resultados, a su sencillez, la cual que permite una gran velocidad de procesamiento al ser implementado.