UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

REPRESENTACIÓN DE NUBES DE PUNTOS EN AMBIENTES VIRTUALES

TESIS
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
HUGO ENRIQUE ESTRADA LEÓN

Director de Tesis:
DR. JESÚS SAVAGE CARMONA
Facultad de Ingeniería, UNAM

CIUDAD UNIVERSITARIA, CD. MX. ENERO, 2017

Agradecimientos

La presente tesis marca la culminación de la etapa de mis estudios de maestría, en la cual, participaron varias personas, en los momentos difíciles y en los momentos de felicidad.

Son muchas las personas a las que me gustaría agradecerles. Algunas están aquí conmigo y otras en mis recuerdos. Sin importar en donde estén o si alguna vez llegan a leer estas dedicatorias quiero darles las gracias por formar parte de mí, por todo lo que me han brindado y por hacer más amena esta etapa de mi vida.

Dedicada con un profundo agradecimiento y admiración a mis padres Catalina y Johnny por todo el apoyo, sobre todo por la confianza y libertad que me dieron para decidir mi camino.

A mi hermano Hector, que a su muy particular manera me brindó su apoyo en los momentos que lo necesite.

A Denisse por todo el amor y los momentos felices que hemos vivido.

A toda mi familia por el apoyo incondicional que nunca me han dejado de brindar.

De manera especial expreso mi más sincero agradecimiento a mi director de tesis el Dr. Jesús Savage, por darme la oportunidad de pertenecer a su grupo de trabajo, por la confianza y por el tiempo invertido en la revisión de este trabajo; a la maestra Norma Elva Chávez por toda la ayuda y consejos brindados durante los últimos años, toda mi gratitud y respeto para ustedes.

A mis amigos, sin excluir a ninguno, pero en especial a Julio, Reynaldo, Arely, Erika, David, Beto, Charly, Angel, José, Mario, Vero y Stephanie. Muchas gracias por todos los momentos que hemos pasado juntos y porque han estado conmigo siempre, haciendo de esta etapa de mi vida una verdadera aventura.

A mis sinodales Mat. Ana Luisa Solis, M.I Abel Pacheco, Dr. Pablo Pérez, Dr. Miguel Padilla por su tiempo dedicado a revisar este trabajo, así como sus siempre oportunas observaciones que han contribuido de forma significativa a mejorar esta tesis.

A todo el equipo de Bio-Robótica. Ha sido un placer trabajar con ustedes este par de años. Gracias por su apoyo y su amistad.

Al IIMAS y a la Universidad Nacional Autónoma de México por ser mi alma mater. Mi deseo es siempre poner su nombre en alto.

Agradezco también a CONACYT por la beca otorgada durante la realización de mi maestría.

Por ultimo agradezco a la DGAPA-UNAM por el apoyo proporcionado para la realización de esta tesis a través del proyecto PAPIIT IG100915 "Desarrollo de técnicas de la robótica aplicadas a las artes escénicas y visuales".

A todos y cada uno de ellos les dedico cada una de estas páginas de mi tesis.

Hugo Enrique Estrada León

Índice

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

En la actualidad el uso de simuladores no es una novedad, por lo que son cada vez más las disciplinas que adoptan está técnica como un complemento para el desarrollo de sus actividades, ya que el uso de la simulación permite probar algoritmos antes de llevarlos a un nivel físico. Esto puede ayudar a resolver la incertidumbre de si dichos algoritmos tienen un desempeño adecuado y posteriormente en la toma de decisión de implementar dicho algoritmo.

La idea de tener robots que sean usados como ayudantes en la industria o incluso en ambientes domésticos se ha convertido en un objetivo de alta prioridad en investigaciones actuales. Por lo que se ha generado una vertiente en donde la meta es crear sistemas que sean cada vez más independientes y que realicen tareas cada vez más complejas, hasta llegar a tener sistemas autónomos que ayuden a llevar a cabo actividades cotidianas; tal es el caso de los robots de servicio.

La reconstrucción de superficies tiene como principal propósito realizar una representación tridimensional a partir de datos obtenidos principalmente por algún tipo de sensor, los cuales no siempre obtienen medidas sin ruido.

A partir del año 2010, los sensores RGB-D (Microsoft Kinect, por ejemplo) se convirtieron en un dispositivo de costo accesible para obtener datos tridimensionales del ambiente. Estos sensores capturan, en cada pixel, la información de color, así como también la distancia de la cámara al área representada por el pixel (profundidad). Además, en el caso particular de un robot de servicio, este tipo de dispositivos, pueden servir para obtener datos de entrada para desempeñar funciones importantes, como pueden ser la localización y mapeo simultáneos o el reconocimiento de personas y acciones, etc.

Una nube de puntos es un conjunto de vértices en un sistema de coordenadas tridimensional. Estos vértices se identifican habitualmente como coordenadas X, Y, y Z y son representaciones de la superficie externa de un objeto. Típicamente, hablamos de nubes de puntos en espacios 2D/3D cuando nos referimos a datos como pueden ser: profundidad, color, textura, forma, etc., que obtenemos mediante sensores como: cámaras, láseres, sónares u otros dispositivos. Los puntos que representan un espacio 3D (y para los cuales no existe

información del 100% de los puntos) se le llama nube de puntos, puntos aleatorios que representan un espacio 3D.

En el presente trabajo, nos enfocaremos al campo de la robótica y en específico a la simulación del ambiente de un robot de servicio. La simulación a realizar será mediante la reconstrucción de superficies a partir de información adquirida por una cámara RGB-D (Kinect), la cual forma parte del sistema de sensores del robot Justina, un robot móvil de servicio desarrollado en la Facultad de Ingeniería de la UNAM.

1.1. Objetivo

El objetivo general del presente trabajo es:

Desarrollar un sistema que permita construir una representación gráfica del mundo en el que se desenvuelve un robot de servicio; específicamente el robot desarrollado en el laboratorio de Bio-Robótica de la Facultad de Ingeniería, Justina. Los datos con los que se realizará la construcción del ambiente virtual son nubes de puntos. La captura de los datos se hará mediante un sensor Kinect con el que cuenta el robot de servicio Justina.

Para cumplir con este objetivo, se plantearon las siguientes actividades:

- Adquirir imágenes de profundidad (nubes de puntos) provistos por el sensor Kinect montado sobre el robot Justina.
- Implementar un algoritmo que permita hacer el registro de varias nubes de puntos para generar una sola y así obtener una toma general del espacio en donde se encuentra el robot.
- Desarrollar un método o métodos para realizar un muestreo de las nubes de puntos, con la finalidad de optimizar recursos de memoria.
- Implementar un algoritmo para obtener el cierre cóncavo de las nubes de puntos para generar una representación geométrica del mundo del robot.
- Visualizar el ambiente virtual dentro del simulador utilizado por los desarrolladores del software del robot Justina.
- Generar una metodología para el uso del sistema desarrollado y probarlo experimentalmente.

1.2. Hipótesis

La hipótesis que se generó al iniciar el desarrollo de este trabajo y en la cual se basa el objetivo del mismo es:

Para obtener una representación geométrica de un "mundo" no es indispensable tener toda la información, basta con tener un muestreo representativo de este para conseguirlo. Para ello se implementarán dos métodos de muestreo de datos: voxelización y cuantización vectorial.

Para concluir este trabajo y validar la hipótesis mencionada anteriormente, se desarrollaron una serie de pruebas experimentales.

1.3. Metas

Obtener conocimiento del estado del arte actual que existe sobre la construcción de ambientes virtuales utilizando sensores de profundidad y de la simulación de navegación dentro de los ambientes construidos. Lograr que los módulos desarrollados sean funcionales y queden tan optimizados como se pueda en el periodo de trabajo.

1.4. Relevancia y contribución del trabajo

La interfaz a desarrollar permitirá la construcción de ambientes virtuales mediante nubes de puntos. Esta interfaz será de utilidad para el grupo del Laboratorio de Bio-Robótica de la Facultad de Ingeniería, que está a cargo del Dr. Jesús Savage, como medio para probar, modificar y validar algoritmos de planeación de rutas utilizando técnicas de Inteligencia Artificial dentro de un ambiente virtual generado a través de nubes de puntos.

1.5. Organización de la tesis

El presente trabajo es resultado de la investigación realizada en el Laboratorio de Bio-Robótica en la Facultad de Ingeniería de la UNAM. Se encuentra estructurado de forma secuencial, siguiendo los pasos que se fueron desarrollando durante la elaboración del mismo. Al final, se encuentra un apartado de apéndices, donde se presentan partes del código que integran el sistema desarrollado.

En este primer capítulo se ha presentado la introducción del problema que pretende solucionar este trabajo, los objetivos y metas planteadas, así como la relevancia y contribución de esta tesis.

A continuación, en el capítulo 2, se presentarán algunos antecedentes en los que se basa este trabajo, así como otros trabajos relacionados y el estado del arte.

El capítulo 3 presenta una serie de técnicas que se utilizan en el procesamiento de nubes de puntos.

El capítulo 4 presenta el desarrollo del sistema para realizar la construcción del ambiente virtual. El capítulo describe la implementación realizada para cada uno de los módulos que conforman a dicho sistema.

Posteriormente, el capítulo 5 presenta algunos resultados en pruebas que se hicieron con el sistema diseñado, se analizan las ventajas y desventajas de este sistema.

Finalmente, en el Capítulo 6 se exponen las conclusiones obtenidas a lo largo del proceso de investigación surgido a partir de este trabajo. Además, se plantean ideas sobre el trabajo a seguir para continuar mejorando el sistema desarrollado.

Capítulo 2

Antecedentes

En este capítulo se presentan los conceptos y temas en los cuales está basada esta tesis. En primer lugar se da una breve explicación de en qué consiste la reconstrucción de modelos a partir de imágenes RGB-D; después se hace una explicación más detallada del sensor Kinect, que es la cámara RGB-D utilizada en los trabajos que se mencionarán en el estado del arte y el sensor a utilizar en esta tesis; se hace un análisis a sus componentes y se dan algunas especificaciones técnicas. De igual forma se dará una explicación de un modelo de cámara y los fundamentos teóricos para transformar un mapa de profundidad a puntos en un sistema coordenado de referencia.

Se presenta el estado del arte, mencionando algunos trabajos relacionados en la reconstrucción de ambientes 3D utilizando sensores RGB-D. El primero es sobre el trabajo llamado “KinectFusion” [1] y [2], el cual es pionero en el ámbito de la reconstrucción de modelos en 3D, se da una breve explicación de cómo es el funcionamiento de este sistema; además, se describe el pipeline utilizado para obtener el modelo 3D. El segundo trabajo es titulado: “RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments” [3], para este trabajo se describirá como es que resolvieron cada uno de los 3 componentes principales de su sistema.

2. 1. Modelado de entornos interiores con mapeo RGB-D

Los modelos de entornos en 3D se usan intensamente en numerosas actividades reales, como por ejemplo la navegación. Crear modelos tridimensionales en tiempo real facilita la construcción de autómatas que necesitan tener un conocimiento del entorno cambiante en el que tienen que desarrollar su función.

El problema es que el hardware necesario para crear modelos 3D es caro y no está al alcance del público en general. Pero hay una excepción a esta afirmación: las cámaras RGB-D. Este tipo de cámaras son capaces de capturar imágenes en RGB y asociar, a cada uno de los puntos tomados, información de profundidad.

Con esta información se pueden crear modelos tridimensionales que pueden ser muy exactos dependiendo de que tan exacta sea la información tomada por la cámara y de que tan bueno sea el proceso utilizado para crear el modelo.

La mayoría de los sistemas de creación de modelos mediante imágenes tienen 3 componentes principales: primero, el que se encarga de alinear los cuadros consecutivos; segundo, el que se encarga de detectar ciclos; y tercero, el que consigue un alineamiento global de toda la secuencia de datos.

La reconstrucción de superficies tiene como principal propósito realizar una representación tridimensional a partir de datos obtenidos principalmente por algún tipo de sensor, los cuales no siempre obtienen medidas sin ruido. Para lograr la reconstrucción de superficies se han utilizado diferentes enfoques. Existen algoritmos [4], [5] que trabajan sobre nubes de puntos no organizadas, que pueden ser utilizados en grandes conjuntos de datos y que no toman en cuenta la incertidumbre ni la temporalidad del proceso de adquisición. Algunos otros, conocidos como métodos paramétricos, realizan un empalme de las muestras consecutivas (adquiridas por un sensor) y unen los puntos para ajustar localmente polígonos [6].

2.2. Características y especificaciones del Kinect

El sensor Kinect fue desarrollado por la empresa Microsoft. Fue puesto a disposición del público en noviembre del 2010, como un dispositivo enfocado principalmente a la interacción natural en videojuegos, mediante poses y movimientos del cuerpo de los usuarios.

Sin embargo, al ser este un sensor de muy bajo costo (comparándolo con otros sensores que proveen información similar, como pueden ser los escáneres láser), rápidamente atrajo la atención de muchos investigadores en el área de robótica. En la actualidad, es común ver robots que utilizan este tipo de sensor para desempeñar diversas tareas, como puede ser la de localización y mapeo simultáneo (SLAM) [7].

Muchos de los detalles del funcionamiento de este sensor no se conocen con exactitud, dado que se consideran un secreto comercial. Sin embargo, se sabe que para obtener el mapa de profundidad utiliza una técnica similar a la de proyección de luz estructurada. Esta técnica consiste básicamente en proyectar un patrón conocido de puntos de luz infrarroja para posteriormente capturar el cambio que se genera en dicho patrón debido a las superficies de los objetos en los que incide.

Analizando este cambio, se puede obtener una imagen de la profundidad de los objetos al plano de la cámara.

Una vez obtenida la información de profundidad, ésta se relaciona con la información obtenida por lo cámara RGB para obtener imágenes de 4 canales. Estos son, tres con la información de color (RGB) y uno con información de profundidad (D). Con la información de profundidad y los parámetros de la cámara, se puede calcular la posición tridimensional de cada pixel de la imagen con el origen referenciado al centro de la cámara del sensor.

Además de los componentes que permiten construir las imágenes RGB-D, los cuales son una cámara de color, un emisor infrarrojo y un sensor infrarrojo, el sensor Kinect cuenta con un arreglo de 4 micrófonos y un motor que le permite girar con un ángulo de "tilt" en un rango de 54 grados. En la figura 2-1 se puede ver el diseño de la arquitectura del sensor.

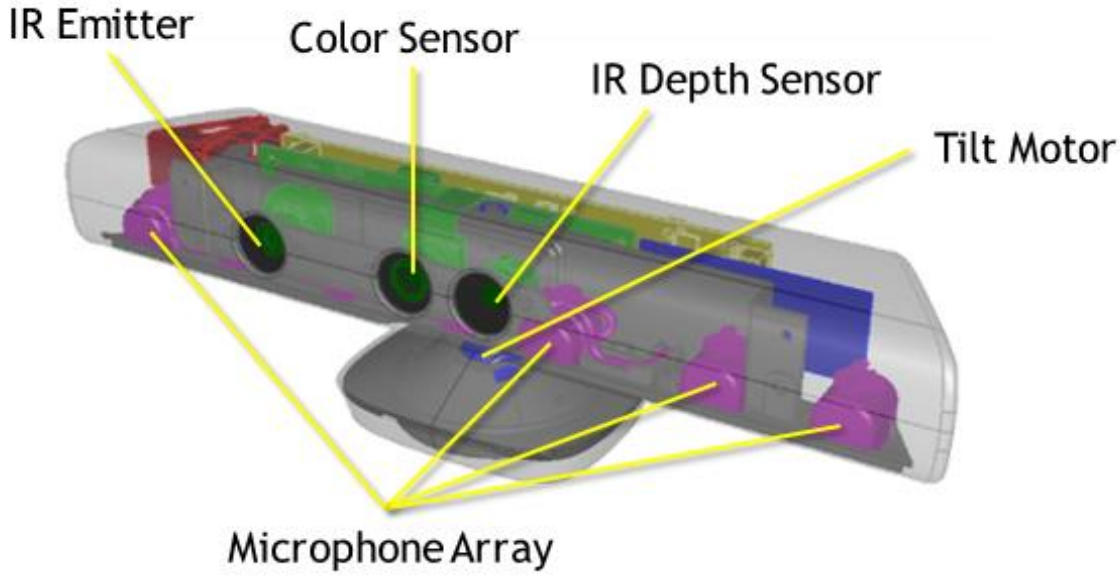


Figura 2-1. Estructura interna del sensor Kinect [8].

En la tabla 2-1 se enumeran las características de las imágenes de color y profundidad [8]. Además, se especifica la distancia máxima y mínima para poder obtener información de profundidad de un objeto.

Característica	Valor
Ángulos de visión	43º en vertical y 57º en horizontal
Resolución máx. de color	1280 x 960
Resolución máx. de profundidad	640 x 480
Rango de profundidad	0.8 - 4.0 metros
Frecuencia máx.	30 cuadros por segundo

Tabla 2-1. Características del Kinect.

La imagen de profundidad que genera el sensor Kinect comúnmente presenta datos sin información, es decir, pixeles en donde el valor de profundidad es 0, los cuales representan error en la lectura. Existen diversas fuentes para este tipo de errores. A continuación se describen algunas de las más importantes según [9]:

Condiciones de luz: en escenas con luz muy intensa, se presentan errores en la detección del patrón de puntos proyectado, lo que da origen a regiones sin información o errores en la medición.

Distancia a los objetos: si los objetos se encuentran fuera del rango de medición del sensor, no se obtendrá el valor de profundidad de estos. Igualmente, si la distancia está fuera del rango, pero se tiene lectura, esta será muy ruidosa.

Distancia entre el sensor IR y el emisor IR: dado que el sensor IR y el emisor IR se encuentran físicamente separados por una distancia considerable en el sensor, partes de la escena pueden aparecer ocluidas o con sombras. Esto se debe a que, dependiendo de la escena, un objeto puede estar siendo proyectado por el emisor IR, pero no detectado por el sensor IR, lo que se traduce en regiones sin información.

Propiedades y orientación de las superficies de los objetos: las propiedades reflectantes de las superficies de los objetos y su ángulo respecto al plano de la imagen pueden generar errores grandes en la medición e inclusive generar regiones sin información. Esto pasa comúnmente con materiales como metales, cerámicas o plásticos transparentes.

2.3. Modelo de cámara

El modelo de cámara (pinhole en inglés) nos permite establecer la relación que existe entre los pixeles de una imagen y los puntos de una escena. Particularmente, entre un pixel de un mapa de profundidad y el punto en el espacio que representa.

Pensemos en un sistema de referencia tridimensional, que llamaremos sistema de coordenadas de la cámara, con ejes ***x, y*** y ***z***, que siguen la convención de la mano derecha, con origen en un punto ***O,*** llamado centro de proyección; y en un plano, que llamaremos plano de la imagen o sensor, paralelo al plano ***xy***, que intersecta al eje ***z*** a una distancia ***f***. En este plano tenemos un sistema de referencia bidimensional, ***x'***, ***y'***, llamado sistema coordenado de la imagen, con origen en el punto en que el plano intersecta el eje ***z,*** la descripción de este sistema se puede observar en la figura 2-2.

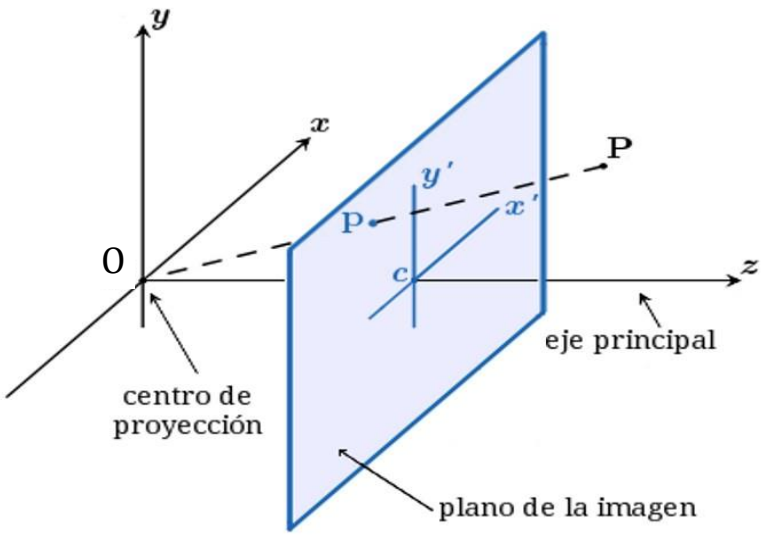


Figura 2-2. Modelo de cámara.

En este modelo, los puntos $\boldsymbol{P = (x, y, z)^T}$ de una imagen tridimensional se proyectan hacia el centro de proyección ***O***. Un pixel $\boldsymbol{p = (x', y')^T}$ se obtiene de la intersección de la línea que pasa por ***P*** y ***O*** con el plano de la imagen.

La relación entre $\boldsymbol{P}$ y $\boldsymbol{p}$ puede determinarse fácilmente mediante triángulos semejantes, esta relación se ve expresada en la ecuación 2.1, y puede verse en la imagen 2-3.

$$\begin{cases} x' = f\dfrac{x}{z} \\ y' = f\dfrac{y}{z} \end{cases} \qquad (2.1)$$

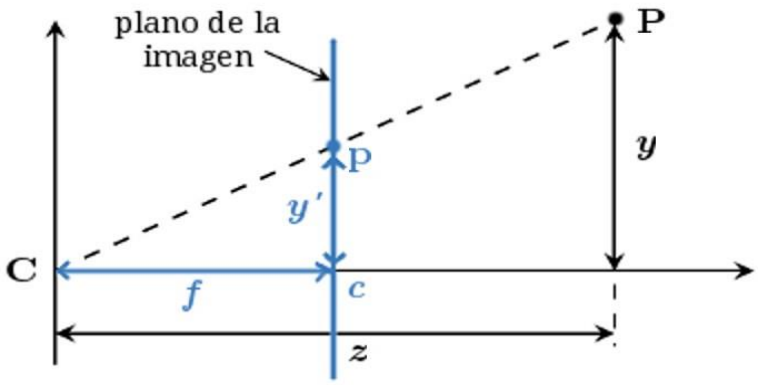


Figura 2-3. Relación entre y y *y'.*

Las ecuaciones 2.1 pueden escribirse en forma matricial, y considerando que el centro de la imagen puede estar en cualquier punto ($\boldsymbol{c_x}$,$\boldsymbol{c_y}$) del plano de la imagen, como sigue:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} f\dfrac{x}{z} + c_x \\ f\dfrac{y}{z} + c_y \end{pmatrix} \qquad (2.2)$$

Estas ecuaciones son útiles cuando tanto la longitud focal, la posición de los puntos de intersección y el centro de la imagen en unidades de longitud (milímetros, por ejemplo) son conocidas. Pero usualmente trabajamos con pixeles como unidad, por lo que resulta conveniente tener una formulación en estas unidades. En las siguientes dos subsecciones de este capítulo damos un vistazo rápido a las coordenadas homogéneas y a los parámetros intrínsecos y extrínsecos de una cámara, que en conjunto permiten obtener la formulación deseada.

2.3.1 Coordenadas homogéneas

En geometría proyectiva es muy común asociarle a un punto $\boldsymbol{p = (x', y')^T}$ del plano cartesiano una representación en coordenadas homogéneas tridimensionales $\tilde{p} = (\lambda x', \lambda y', \lambda)^T$ con $\boldsymbol{\lambda} \in \mathbb{R}$. Dada una representación en coordenadas homogéneas $\tilde{p}$, el punto p puede obtenerse dividiendo las dos primeras coordenadas de $\tilde{p}$ entre la tercera coordenada (siempre que no sea cero). Geométricamente, podemos entender el conjunto de coordenadas homogéneas de un punto bidimensional como la recta que pasa por el origen de un sistema tridimensional, que tiene por ecuación $L(\lambda) = \lambda(x', y', 1)$, y que intersecta al plano $z = 1$ en $(x', y', 1)$.

Un punto $\boldsymbol{P} = (x, y, z)^T$ en tres dimensiones cartesianas también puede ser representado en coordenadas homogéneas como $\widetilde{\boldsymbol{P}} = (\lambda x, \lambda y, \lambda z, \lambda)^T$.

Esta representación permite, entre otras cosas, expresar relaciones no lineales, como la ecuación 2.2, como el producto de matrices:

$$z\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad (2.3)$$

2.3.2. Parámetros intrínsecos y extrínsecos de una cámara

Las imágenes se capturan utilizando un arreglo de dispositivos (con tecnología CCD o CMOS) que convierten la luz a electrones. Estos sensores producen la información de elementos discretos llamados pixeles. Hasta ahora hemos considerando las cantidades f, c_x, c_y en unidades de longitud, pero resulta más conveniente trabajar con pixeles, por lo que introducimos las constantes k_x y k_y como los pixeles por unidad de longitud (pixeles/unidad de longitud) en los ejes x y y.

Con estas constantes podemos obtener la ecuación 2.7 a partir de las siguientes:

$$z\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \boldsymbol{K}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} f_x & 0 & c'_x \\ 0 & f_y & c'_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad (\boldsymbol{2.4})$$

Con $\boldsymbol{f_x = k_x f, f_y = k_y f, c'_x = k_x c_x}$ y $\boldsymbol{c'_y = k_y c_y}$. Estos parámetros se llaman intrínsecos, por lo que la matriz $\boldsymbol{K}$ suele llamarse matriz de parámetros intrínsecos, la cual se obtiene mediante algún método de calibración; que permiten obtener dichos parámetros del sensor de profundidad y de la cámara RGB de un Kinect.

En muchas ocasiones, los puntos estarán referidos en un sistema de coordenadas diferente al de la cámara, que es el que hemos contemplado hasta ahora. Este sistema suele llamarse sistema de coordenadas globales. La relación entre el sistema de la cámara y el de coordenadas globales se puede representar con una matriz de rotación $\boldsymbol{R}$ y un vector de traslación $\boldsymbol{t}$, llamados parámetros extrínsecos. $\boldsymbol{R}$ y $\boldsymbol{t}$ permiten relacionar los dos sistemas, de manera que el equivalente de un punto $P_m = (x_m, y_m, z_m)^T$, en coordenadas globales, es un punto $\boldsymbol{P}$ en coordenadas de cámara dado por:

$$P = RP_m + t \qquad (2.5)$$

Al expresar el punto P_m en coordenadas homogéneas como $\tilde{P}_m = (x_m, y_m, z_m, 1)^T$ y al combinar las ecuaciones 2.4 y 2.5, podemos obtener una expresión que asocia los puntos en un sistema de coordenadas globales con los pixeles de una proyección:

$$z\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = K(Rt)\tilde{P}_m = \begin{pmatrix} f_y & 0 & c'_x \\ 0 & f_y & c'_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} r_{11}\, r_{12}\, r_{13}\, t_1 \\ r_{21}\, r_{22}\, r_{23}\, t_2 \\ r_{31}\, r_{32}\, r_{33}\, t_3 \end{pmatrix}\begin{pmatrix} x_m \\ y_m \\ z_m \\ 1 \end{pmatrix} \qquad (2.6)$$

2.3.3. Conversión de un mapa de profundidad a coordenadas globales

Bajo el modelo de la cámara de pinhole y tomando la ecuación 2.4 podemos, fácilmente, determinar las coordenadas globales de un punto $P_m = (x_m, y_m, z_m)^T$ en una escena a partir de su posición (x', y') en el mapa de profundidad.

Las ecuaciones 2.7, obtenidas a partir de 2.4, nos dan las coordenadas del punto en coordenadas de la cámara, por lo que solamente hace falta multiplicar por una matriz T_m que represente la transformación del sistema de la cámara al sistema de coordenadas globales (Ecuación 2.8).

$$\begin{cases} x = \dfrac{(x' - c'_x)z}{f_x} \\ y = \dfrac{(y' - c'_y)z}{f_y} \\ z = z \end{cases} \qquad (2.7)$$

$$\begin{pmatrix} x_m \\ y_m \\ z_m \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13}\, t_1 \\ r_{21} & r_{22} & r_{23}\, t_2 \\ r_{31} & r_{32} & r_{33}\, t_3 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \qquad (2.8)$$

2.4. Metodologías para la construcción de modelos 3D

En los anteriores subtemas de este capítulo, se describieron conceptos y se sentaron bases teóricas y matemáticas para el desarrollo de esta tesis, sin embargo no se ha descrito ningún tipo de metodología para la construcción de modelos 3D, por lo que en los siguientes subtemas se presentará el estado del arte en este ámbito, teniendo como base las metodologías utilizadas en dos trabajos: el primero titulado "KinectFusion" [1] y [2] y el segundo que lleva por nombre "RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments" [3]. Todo esto para tener un panorama general de cómo se puede atacar el problema de la construcción de modelos 3D y así posteriormente poder establecer nuestra propia metodología.

2.5. KinectFusion

KinectFusion [1] y [2], presentado en el año 2011, proporciona un escaneo de objetos y rápida creación de modelos 3D altamente detallados utilizando un sensor (Kinect). Modelos 3D de objetos grandes o ambientes pueden ser generados moviendo el sensor Kinect alrededor de dicho objeto o ambiente, mientras que para objetos pequeños pueden ser escaneados moviendo el objeto enfrente del sensor.

2.5.1. Características principales de KinectFusion

KinectFusion [1] y [2] crea rápidamente modelos 3D altamente detallados, tomando datos de profundidad capturados por el sensor Kinect y realiza un promedio de las lecturas de las secuencias de los cuadros. Esto permite que el sistema pueda crear un modelo más detallado de lo que sería posible con una sola lectura. El sistema de KinectFusion [1] y [2] reconstruye un modelo con superficies alisadas, mediante la integración de datos de profundidad obtenidos con el Kinect, a lo largo de un periodo de tiempo, desde múltiples puntos de vista.

La posición de la cámara (localización y orientación) es rastreada cuando se mueve el sensor, como la posición de cada uno de los cuadros es conocida y como se tiene el conocimiento de cómo están relacionados entre ellos, estos múltiples puntos de vista de los objetos o del ambiente pueden fusionarse (promediarse) en un solo volumen en una reconstrucción de voxeles.

2.5.2. Pipeline del procesamiento de KinectFusion

El Pipeline del procesamiento de KinectFusion [1] y [2] involucra varias etapas, que van desde los datos en crudo de las imágenes de profundidad hasta la reconstrucción 3D. En la figura 2-4 se observa un esquema del pipeline de KinectFusion [1] y [2], que será explicado brevemente.

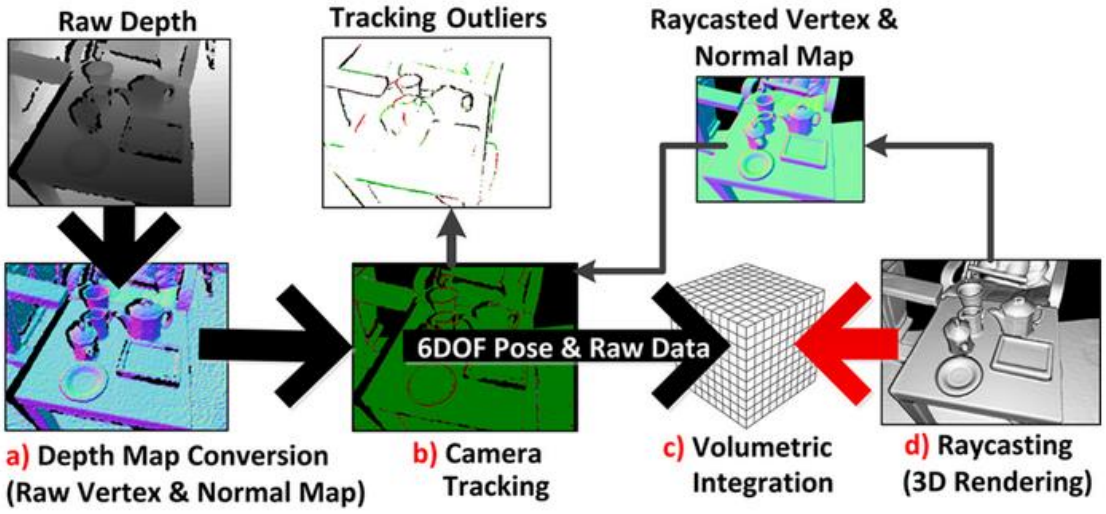


Figura 2-4. Pipeline de KinectFusion [1] y [2].

- La primera etapa es la conversión del mapa de profundidad. Se toma la información de profundidad directamente adquirida por el sensor Kinect y se convierte en profundidad dada en metros, en formato de punto flotante, seguido de una conversión opcional a una nube de puntos orientada, la cual consiste en puntos o vértices 3D en el sistema de coordenadas de la cámara, y de las normales de la superficie (orientación de la superficie).

- La segunda etapa calcula la posición global o de mundo de la cámara (su localización y orientación) y calcula esta posición cuando el sensor se mueve en cada cuadro, usando un algoritmo iterativo de alineamiento; de esta forma el sistema siempre conoce la posición actual del sensor con respecto del cuadro inicial. Existen dos algoritmos en KinectFusion [1] y [2]. El primero, llamado **NuiFusionAlignPointClouds**, puede ser usado tanto para alinear nubes de puntos calculados a partir de la

reconstrucción con nuevas nubes de puntos entrantes de la cámara de profundidad del Kinect, o de forma individual (por ejemplo, para alinear dos cámaras separadas viendo a la misma escena). El segundo algoritmo, **AlignDepthToReconstruction**, provee un resultado más preciso en cuanto al seguimiento de la cámara cuando trabaja con una reconstrucción de volumen, sin embargo, puede ser menos robusto con objetos que se mueven en la escena. Si el seguimiento se ve interrumpido en este escenario, la cámara se vuelve a alinear con la última posición rastreada y el seguimiento debería poder continuar.

- La tercera etapa es la fusión (o integración) de los datos de profundidad de las posiciones del sensor conocidas en una representación volumétrica del espacio alrededor de la cámara. Esta integración de los datos de profundidad se realiza por cada cuadro, de manera continua, con un promedio de ejecución para reducir el ruido, con ello pueden ser manejados algunos cambios dinámicos en la escena (tales como pequeños objetos siendo removidos o añadidos).

- La última etapa consiste en la reconstrucción del volumen por medio de un rendering, utilizando la técnica de raycasting. Este renderizado de la imagen se hace típicamente para la última posición de la cámara, pero no está limitada a ella.

El tamaño de los volúmenes escaneados puede ser de hasta 8 m^3. Las resoluciones de cada voxel pueden llegar a ser de hasta 1-2 mm, pero no es posible tener ambas de manera simultánea.

2.5.3. Tracking de KinectFusion

El tracking que realiza KinectFusion [1] y [2] únicamente utiliza el flujo de los datos de profundidad provistos por el sensor Kinect. El tracking se basa en que exista suficiente variación de profundidad en cada cuadro, de modo que pueda emparejar lo que ve entre los cuadros y calcular la diferencia en la posición. Si se apunta el Kinect a una pared plana o a una escena que en su mayoría sea plana, no existirá suficiente variación de profundidad para que el tracking tenga éxito. Escenas desordenadas tienen un mejor resultado.

2.6. Mapeo RGB-D: Usando Cámaras de Profundidad para Modelar Ambientes Interiores

Este artículo [3] trata de un algoritmo para la creación de modelos 3D a partir de datos obtenidos de una cámara RGB-D. Se describe cómo fue resuelto cada uno de los tres problemas a los que se enfrenta un sistema de este tipo. Se detalla el algoritmo con el cual se realiza el modelado de ambientes, se comentan los resultados obtenidos al aplicar el algoritmo en pruebas reales, se mencionan las conclusiones obtenidas por los autores tras haber analizado las pruebas y también, se menciona una posibilidad para trabajos a futuro.

2.6.1. Análisis del funcionamiento del sistema

En la figura 2-5 se muestra un diagrama del proceso RGB-D mapping propuesto en este artículo. Se parte de la información RGB y de profundidad que proporciona la cámara RGB-D.

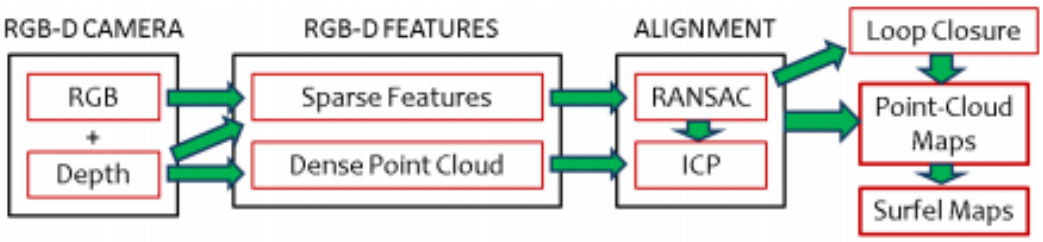


Figura 2-5. Diagrama de flujo del RGB-D mapping [3].

La información de profundidad proporciona una densa nube de puntos (Dense Point Cloud) que representa la profundidad que hay en cada punto. Con la información de profundidad y con las imágenes RGB se crean unos conjuntos de características dispersas (Sparse Features) que no es más que un conjunto de puntos que tienen asociados la información de color y de profundidad.

Estos dos modelos de datos (Sparse Features y Dense Point Cloud) son los datos de entrada del componente RGBD-ICP que es el encargado de alinear los distintos cuadros y de detectar los posibles ciclos que puedan haber. Recordemos que el alineamiento de cuadros es uno de los tres problemas que hay que resolver para tener un modelo 3D a partir de una cámara RGB-D.

2.6.2. Alineamiento de cuadros

Para realizar el alineamiento de los cuadros, lo primero que se implementa es el algoritmo RANSAC RGB-D ICP. RANSAC es el acrónimo de *Random Sample Consensus* [10] que es un método iterativo que trata de estimar los parámetros de un modelo matemático a partir de un conjunto de datos de entrada.

Por otra parte ICP *(Iterative Closest Point)* [11] es un método para la alineación de modelos tridimensionales basado únicamente en la geometría, y en ocasiones el color. El algoritmo es usado para el registro de nubes de puntos cuando se tiene una estimación inicial de la posición relativa de una de las nubes de puntos.

En el caso particular de este artículo para el uso de este algoritmo, los parámetros de entrada son: un cuadro con información de RGB-D (P_s), otro cuadro a comparar (P_t) y una transformación (T_p) que inicialmente tiene el valor de la transformación identidad. El algoritmo utilizado se puede observar en el algoritmo 2-1.

Algoritmo 2-1: Algoritmo RGB-D ICP para el emparejamiento de dos cuadros RGB-D

Entrada: cuadro RGB-D fuente $\boldsymbol{P_s}$, cuadro RGB-D objetivo $\boldsymbol{P_t}$, transformación anterior $\boldsymbol{T_p}$
Salida: Transformación relativa optimizada **T***

1. $\boldsymbol{F_s}$ ← Extract_RGB_Point_Features ($\boldsymbol{P_s}$);
2. $\boldsymbol{F_t}$ ← Extract_RGB_Point_Features ($\boldsymbol{P_t}$);
3. (**T***, $\boldsymbol{A_f}$) ← Perform_RANSAC_Aligment ($\boldsymbol{F_s}$, $\boldsymbol{F_t}$);
4. **If** $|A_f| <$ **γ then**
5. **T*=** $\boldsymbol{T_p}$;
6. $\boldsymbol{A_f}$**= 0;**
7. **end**
8. **repeat**
9. $\boldsymbol{A_d}$ ← Compute_Closest_Points (**T***, $\boldsymbol{P_s}$, $\boldsymbol{P_t}$);
10. **T*** ← Optimize_Alignment (**T***, $\boldsymbol{A_f}$, $\boldsymbol{A_d}$);
11. **until** (Change (**T***) ≤ θ) **or** (Iterations > MaxIterations);
12. **return T***;

Algoritmo 2-1. Algoritmo RGB-D ICP.

Lo primero que se hace en este algoritmo es, para cada uno de los dos cuadros, extraer las características visuales de ambos, en conjunto con la información de profundidad asociada a cada cuadro con el fin de obtener puntos característicos en 3D (líneas 1 y 2 del algoritmo). En este artículo se utilizan las características obtenidas del algoritmo de SIFT *(Scale-Invariant Feature Transform)* [12], el cual es un algoritmo de visión por computadora que se utiliza para extraer características relevantes de las imágenes, estas características posteriormente se pueden utilizar en reconocimiento de objetos, detección de movimiento, registro de imágenes, entre otras aplicaciones. La figura 2-6 muestra un ejemplo de dichas características.

Una vez que se obtienen los dos conjuntos de características, se aplica RANSAC [10] para hacer el alineamiento y para obtener la mejor transformación T* entre estos dos conjuntos y el conjunto de pares de puntos A_f, que han generado la mejor transformación (línea 3 del algoritmo).

Figura 2-6. Características de SIFT.

Si el conjuntos de pares de puntos (A_f) que se han encontrado es menor que un cierto umbral (línea 4 del algoritmo), no se puede afirmar que la transformación T* sea la adecuada, por lo tanto, se inicializa la transformación con el valor que tenía en la iteración anterior (línea 5 del algoritmo) y se vacía el conjunto de pares de puntos A_f (línea 6 del algoritmo).

En la línea 8 del algoritmo empieza el proceso principal del algoritmo. La función Compute_Closest_Point aplica a cada uno de los puntos del cuadro de entrada P_s la transformación T* y para el punto obtenido trata de encontrar en el cuadro de salida P_t el punto más cercano. Para ello se pueden utilizar combinaciones de distancia euclidiana, diferencia de color y diferencia en la forma.

El resultado de esta operación es el conjunto de asociaciones de pares de puntos Ad, que indica para cada punto del cuadro de entrada el punto del cuadro de salida que tiene más posibilidades de ser el mismo punto.

La función Optimize_Alignment de la línea 10 del algoritmo simplemente minimiza el error de alineamiento entre los puntos, utilizando para ello la transformación T* y los dos conjuntos de pares de puntos A_f y A_d. El objetivo es conseguir una nueva transformación T* que minimice los errores.

El proceso se repetirá hasta que la transformación no se modifique en una iteración o hasta que el número de iteraciones llegue a un umbral en cuyo caso se devolverá la última transformación calculada (líneas 11 y 12 del algoritmo). La matriz obtenida es la que se puede aplicar para alinear correctamente dos cuadros consecutivos de la cámara. por lo que el primer problema de los tres que se tenían que resolver ya está resuelto. Recordemos que los otros dos son la detección de ciclos y la construcción global del modelo.

2.6.3. Detección de ciclos

El alineamiento de cuadros (frames) no es perfecto y va introduciendo errores a medida que se avanza en la construcción del modelo global. Estos errores se van acumulando llegando incluso a representar la misma región del mundo real en diferentes partes del modelo. Esto es lo que se conoce como un ciclo y es el segundo problema que hay que resolver.

La primera aproximación que se podría realizar para resolver este problema es ejecutar el algoritmo de RANSAC [10] entre el cuadro actual y todos los cuadros anteriores que haya capturado la cámara. Aunque esto resolvería el problema, es computacionalmente prohibitivo, por lo que hay que utilizar otra aproximación.

La solución a este problema ha sido definir los cuadros clave (keyframes) que no son más que unos cuadros especiales que se obtienen a medida que se van alineando dichos cuadros. Cada vez que se procesa un cuadro, se intenta asociar también con el último cuadro clave de forma que si se consigue alinear, el cuadro procesado pasará a ser un cuadro clave. La idea es tener un conjunto de cuadros especiales que no se puedan alinear entre sí.

Cada vez que se crea un nuevo cuadro clave, se intenta alinear con los anteriores cuadros clave. No es necesario aplicar el costosísimo RANSAC [10] en todos, ya que se puede aplicar un proceso previo que puede descartar dos cuadros rápidamente.

Si dos cuadros clave son candidatos, se aplicará el RANSAC [10] entre ellos con el fin de averiguar si se trata de dos cuadros que puedan ser alineados. En caso afirmativo, se detectará un ciclo y se asociarán los dos cuadros al mismo espacio del modelo. Ahora ya sólo queda resolver el tercer problema para crear el modelo completo que es la construcción global del modelo.

2.6.4. Construcción global del modelo

Existen 3 componentes de los que no hemos hablado todavía y que tienen que ver con la construcción global del modelo. El primero de ellos, conocido como Optimización Global (Global Optimization), consiste en representar los cuadros y las relaciones entre ellos en forma de un grafo que permite detectar posibles ciclos. Para ello se usa el TORO [13] y [14], un sistema para minimizar de manera eficiente el error en aquellos grafos donde los vértices están parametrizados por componentes de traslación y rotación y las aristas representan restricciones entre los parámetros y matrices de covarianza asociadas.

La segunda estrategia de optimización es utilizar el método de *Sparse Bundle Adjustment (SBA)* [15], [16] y [17], que sirve para minimizar el error en la re proyección de puntos entre los cuadros. La idea principal de este método es tomar en cuenta las características visuales semejantes que aparezcan en varios cuadros, así como también el ajuste de la estimación de las ubicaciones en 3D, junto con las posiciones de la cámara, haciendo la optimización robusta a la estimación de profundidades con cierto grado de incertidumbre.

Cada cuadro de una cámara RGB-D tiene 640 x 480 = 307200 puntos. Si a esto le sumamos que al ir alineando cuadros se van añadiendo puntos, es fácil observar que el modelo resultante tendrá un número de puntos muy elevado. Si queremos representar el modelo habrá que convertir este conjunto de puntos en una estructura de datos más apropiada para su visualización.

En este ejemplo en particular utilizaron el método de los surfels [18]. Un surfel consiste en una localización, una orientación de la superficie, un tamaño y un color. Las nubes de puntos se pueden añadir, borrar y modificar de esta forma de representar los modelos, que es mucho más eficiente y está mucho más próxima a una representación para su visualización.

Capítulo 3

Procesamiento de nubes de puntos

Como se ha mencionado en los anteriores capítulos, los datos de entrada con los que se estará trabajando en este trabajo son las nubes de puntos. De igual forma, ya se ha mencionado que éstas nubes de puntos poseen información no solo de posición en los ejes X, Y y Z, sino que también en ocasiones almacenan información de color.

Esto provoca que se tenga una excesiva cantidad de información para procesar. En una captura de una nube de puntos de tamaño estándar (640X480) tenemos 307200 puntos por procesar, lo cual no es tarea fácil.

Por lo anterior, en la hipótesis mencionada al inicio de este trabajo, se mencionó que para obtener una buena representación geométrica del mundo no es necesaria toda ésta información por lo que a continuación se abordaran dos métodos para hacer el muestreo de las nubes de puntos: voxelización y cuantización vectorial. El muestreó por voxelización se lleva acabo modificando el tamaño de los voxeles para así obtener una menor cantidad de puntos y por otro lado, en la cuantización vectorial se varía el número de regiones en las que se divide la nube de puntos, logrando el mismo propósito que la voxelización; es decir, reducir el tamaño de la nube de puntos.

De igual forma se mencionará en que consiste el registro de nubes de puntos para obtener una captura total del mundo a través de capturas parciales de este.

3.1 Voxelización

El enfoque más simple para realizar la voxelización de un sólido consiste en comprobar la inclusión del centro de cada voxel en el sólido. Para el cálculo de la inclusión suele utilizarse algún algoritmo basado en el Teorema de la Curva de Jordan [19]. El principal problema de este método es que puede ser calificado de fuerza bruta, ya que tiene un bajo rendimiento debido a que para cada punto a ser

probado hay que ejecutar el algoritmo de intersección de rayo-solido, que implica a su vez la utilización del algoritmo rayo-triangulo.

3.1.1 Algoritmo basado en Scanline

Una forma muy sencilla de voxelizar un sólido consiste en utilizar una variante del algoritmo scanline utilizado en 2D. Con este procedimiento se lanzan rayos siguiendo una dirección alineada con alguno de los ejes coordenados. Cada rayo intersecta al sólido en varios puntos contenidos en una fila del espacio de voxeles resultante. Ordenando las intersecciones se localizan los intervalos de voxeles de la fila que quedan fuera y los que quedan dentro del sólido.

Aunque este método es más eficiente que el de la fuerza bruta, sigue presentando los mismos problemas de precisión y los casos especiales derivados del uso de los algoritmos de intersección rayo-triangulo. También presenta problemas de aliasing, que pueden atenuarse mediante técnicas de filtrado [20] o de distancia de campos [21]. En cualquier caso, se puede considerar este método como una técnica básica de referencia. En la figura 3-1 se muestran ejemplos de voxelización utilizando este método, variando la resolución de los voxeles.

Figura 3-1. Voxelizaciones realizadas con scanline a distintas resoluciones.

3.1.2. Algoritmo de Jones

El algoritmo de Jones [21] ofrece un método que voxeliza un modelo utilizando una función de distancia punto a triangulo. Con este enfoque, cada voxel del espacio resultante es tratado como un punto (su centro) y se calcula su distancia a cada triangulo del sólido. Hay varias optimizaciones que aumentan el rendimiento, pero en general es un método muy lento. Como pasa con la mayoría de los métodos, no puede voxelizarse el interior del sólido.

3.1.3 Otros Algoritmos

Existen otro tipo de metodologías como la que presenta Haumont [22], la cual convierte escenas poligonales completas en una representación voxelizada. Otros más como el algoritmo de Fang [23] y el algoritmo de Karabassi [24] que se enfocan en las coordenadas de profundidad de las imágenes (Z-buffer).

3.2 Cuantización Vectorial

La cuantización vectorial es un tipo de cuantización que trabaja en bloques de datos, en lugar de datos aislados, y dichos bloques pueden ser representados por vectores, de ahí el nombre. La ventaja de trabajar sobre bloques de datos radica en que es más fácil lograr una buena compresión sin tener tantas pérdidas y, por lo tanto, se puede lograr que la longitud de palabra sea menor para una distorsión dada, o una menor distorsión para una longitud de palabra dada. A continuación se presenta la definición matemática de la cuantización vectorial:

Un cuantizador vectorial de dimensión **K** y tamaño **N** es un mapeo del espacio R^K, $\tilde{x} = [x_1 \dots x_K]$ a un conjunto con **N** elementos $C = \{\bar{y}_1, \dots, \bar{y}_N\}$ donde y_i pertenece al conjunto de los números reales. Asociado a cada y_i se tienen regiones R_i tales que:

$$R_i = \{x; Q(x) = y_i\}$$

Las cuales cumplen con las siguientes condiciones:

1. $R_i \cap R_j = \emptyset; \quad i \neq j; \quad i, j = 1, \dots \dots, N$
2. $\bigcup_{i=1}^{N} R_i = R^K$

3.2.1. Construcción de un cuantizador vectorial

La creación de un cuantizador vectorial cuenta de tres pasos específicos, los cuales son: crear un codebook (o libro de palabras o diccionario), un codificador y finalmente el decodificador.

1. **Diccionario:** Se genera con la finalidad de poseer todos los vectores que van a representar la información transmitida y este se genera únicamente al inicio del diseño, por lo que en general, no se modifica su contenido, pero eso depende de la aplicación. Para su construcción existen diversos algoritmos, siendo el algoritmo LBG [25] de los más usados.

   Algunos otros métodos para crear el diccionario inicial son los siguientes:

   - **Agrupamiento:** Este proceso se inicia con todos los vectores de la secuencia de entrenamiento como vectores de código, se agrupan por parejas y son sustituidos por su centroide. Aquí, los vectores del diccionario no son generalmente elementos de la secuencia de entrenamiento, pero son muy pesados computacionalmente.

   - **Separación:** Aquí se comienza con el centroide de la secuencia de entrenamiento como único integrante del diccionario. Luego se divide en dos vectores de código, perturbándolo; y se aplica la iteración de Lloyd. Se repite este procedimiento hasta llegar al tamaño del diccionario buscado. Este procedimiento es muy utilizado en el algoritmo LBG [25].

2. **Codificación:** Este es el proceso que carga con la mayor dificultad, ya que se encarga de realizar una búsqueda para poder asociar un vector del diccionario creado con anterioridad, el cual debe ser lo más cercano posible y que sea capaz de representar al vector de entrada. Es decir, el proceso consiste en tomar un vector **X** que corresponda a la entrada y compararlo con el contenido del diccionario, y de esta forma saber con cuál de ellos se tiene la menor distorsión posible (es decir, la menor distancia). Una vez localizado, se marca su posición mediante los índices **i** y manda esta información al decodificador. A continuación se presenta el funcionamiento general de la cuantización vectorial: El primer paso consiste en agrupar los datos que provienen de la fuente en bloques, o vectores. El codificador y el decodificador contienen un conjunto de vectores **L**-dimensionales llamado codebook (libro de códigos, o simplemente, diccionario). Los vectores de este 'diccionario' se llaman 'vectores de código' y se seleccionan de tal manera que sean una representación de las muestras emitidas.

El diccionario asigna índices a cada vector código, y el codificador busca el vector código más cercano al vector de entrada. Después se transmite el índice de este vector código y el decodificador genera el mismo vector código, produciendo como salida la secuencia de componentes del vector código.

3. **Decodificación:** Este proceso se encarga de identificar la posición del vector en el diccionario, entregando los datos que posee dicho vector, con lo que finalmente se recupera la información.

3.2.2. EL ALGORITMO LLOYD GENERALIZADO (LBG)

El algoritmo de Lloyd generalizado (LBG) [25], se puede resumir en los siguientes pasos:

a) Dado un alfabeto $Y_m = \{y_i\}$, encuentre la partición óptima de las celdas de cuantización, usando la condición de vecino más cercano.

$$R = \{x: d(x, y_i) \leq d(x, y_j)\} \; \forall \; j \neq i$$

b) Usando la condición de centroide, encuentre el Y_{m+1}, el alfabeto de reproducción óptimo para el conjunto de agrupamientos encontrado.

c) Calcular la distorsión promedio para Y_{m+1}, y si ha cambiado por una cantidad muy pequeña desde la última iteración, detenerse. En caso contrario poner $m + 1 \rightarrow m$ y regresar al paso a).

3.3. Registro

El problema de alinear un conjunto de capturas distintas de nubes de puntos en un único y completo modelo es conocido como registro. El registro consiste en encontrar tanto la orientación y la posición relativa de las capturas adquiridas de manera separada en un solo sistema de coordenadas globales, tales que las áreas de intersección entre ellas se traslapen perfectamente.

El registro de un par de nubes de puntos es conocido como registro de parejas, y su salida es usualmente una matriz (4x4) de transformación rígida que representa la rotación y la traslación que debería haber sido aplicada a una de las nubes (fuente) de tal forma de que quede perfectamente alineada con la otra nube (objetivo o modelo).

Los pasos a seguir en el registro de parejas se muestran en la figura 3-2, en la cual se representa una solo iteración del algoritmo.

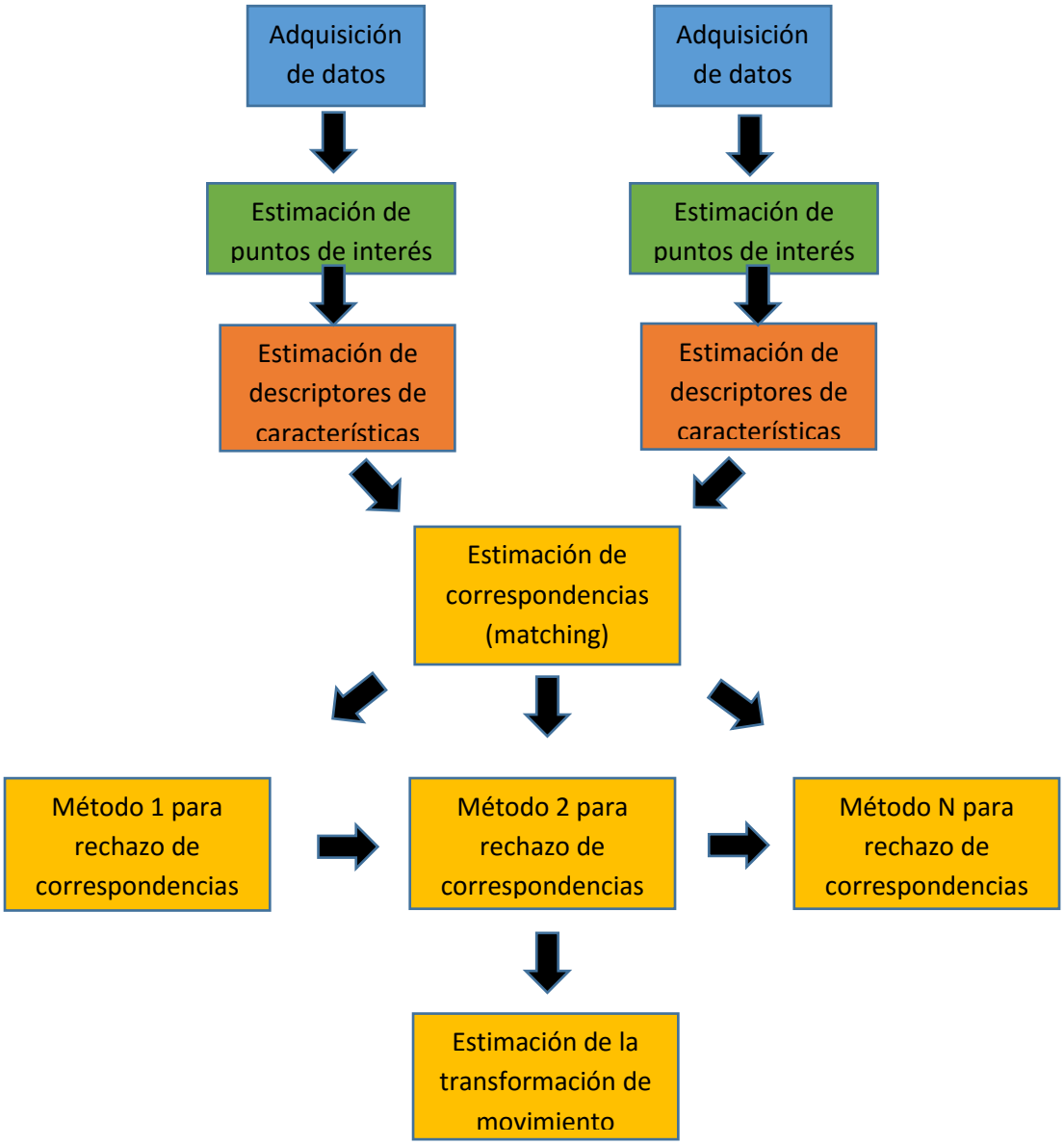


Figura 3-2. Diagrama que ejemplifica el proceso para el registro de nubes de puntos.

La serie de pasos para realizar el registro de parejas es el siguiente:

- De un conjunto de puntos, identificar los puntos de interés (por ejemplo, keypoints) que mejor representen a la escena en ambos conjuntos;

- Para cada punto de interés, ejecutar un descriptor de características;

- A partir del conjunto de descriptores de características, junto con las posiciones XYZ en ambos conjuntos de datos, estimar un conjunto de correspondencias, basado en las similitudes entre las características y posiciones;

- Dado que asumimos que los datos son ruidosos, no todas las correspondencias son válidas, así que se rechazan aquellas malas correspondencias que contribuyen negativamente al proceso de registro;

- Del conjunto restante de buenas correspondencias, estimar la transformación de movimiento.

3.3.1. Módulos del Registro

- Daremos una pequeña explicación de en qué consisten los pasos del pipeline, los cuales se describen brevemente a continuación:

3.3.1.1. Puntos de interés

Un *keypoint* es un punto de interés que posee una "propiedad especial" dentro de la escena, tal como la esquina de un libro o alguna letra distintiva escrita sobre él. Existe un número de diferentes puntos de interés disponibles en PCL *(Point Cloud Library)* [28] tales como NARF [27], SIFT [12] y FAST [26]. Alternativamente, se podría tomar cada punto o cada subconjunto de puntos característicos. El problema de alimentar una estimación de correspondencias con dos capturas del Kinect directamente es que se tienen 300K puntos en cada cuadro, así que se tendrían 300k^2 correspondencias.

3.3.1.2. Descriptores de Características

Con el conjunto de características que encontramos debemos de extraer aquellas, donde uniremos la información y generaremos vectores para compararlos entre ellos.

3.3.1.3. Estimación de correspondencias

Dados dos conjuntos de vectores de características de dos capturas adquiridas con el Kinect, debemos encontrar la correspondencia de las características para encontrar las partes que se traslapan en los datos. Dependiendo del tipo de características que se usen se pueden utilizar diferentes métodos para encontrar correspondencias.

Para emparejamiento de puntos (usando los puntos, coordenadas XYZ como características) existen diferentes métodos para datos organizados y desorganizados:

- Emparejamiento por fuerza bruta,
- Arboles kd, búsqueda del vecino más cercano por medio de la biblioteca FLANN *(Fast Library for Approximate Nearest Neighbors)*,
- Búsqueda en el espacio de la imagen de datos organizados, y
- Búsqueda indexada del espacio de datos organizados.

Para emparejamiento de características (no usando las coordenadas de los puntos si no ciertas características) solo existen los siguientes métodos:

- Emparejamiento por fuerza bruta y
- Arboles kd, búsqueda del vecino más cercano por medio de la biblioteca FLANN *(Fast Library for Approximate Nearest Neighbors)*.

En adición a la búsqueda, dos tipos de estimación de correspondencias son distinguidas:

- Estimación de correspondencias directa (default) busca las correspondencias en la nube B para cada punto de la nube A.
- Estimación de correspondencias recíproca, busca las correspondencias de la nube A en la nube B, y de la B a la A y usar solo la intersección.

3.3.1.4. Rechazo de correspondencias

Obviamente no todas las correspondencias son correctas. Debido a que las correspondencias incorrectas pueden afectar negativamente la estimación de la transformación final, estas necesitan ser rechazadas. Esto podría hacerse usando RANSAC [10] o limitando la cantidad y utilizar solamente un cierto porcentaje de las correspondencias encontradas.

Un caso especial es una o varias correspondencias donde uno de los puntos en el modelo corresponde a un número de puntos en la fuente. Este caso puede ser abordado usando solamente aquel con la menor distancia.

3.3.1.5. Transformación estimada

El último paso es calcular la transformación.

- Evaluar algún error de métrica basado en la correspondencia.
- Estimar una transformación (rígida) entre las posturas de la cámara (estimación de movimiento) y minimizar el error medido.
- Ejemplos: SVD para estimación de movimiento; Levenberg-Marquardt con diferentes kernels para estimación de movimiento;
- Usar la transformación rígida para rotar o trasladar la fuente hacia el objetivo, y potencialmente correr un ciclo interno de ICP [11] con todos o solo un subconjunto de puntos o los keypoints.
- Iterar hasta algún criterio de convergencia conocido.

3.4. Ejemplos de pipeline

A continuación se describirán los pasos dentro del pipeline de dos algoritmos diferentes para realizar el registro, los cuales son: ICP [11] y Registro basado en características.

3.4.1. Iterative Closest Point (ICP)

1. Buscar las correspondencias.
2. Rechazar las malas correspondencias.
3. Estimar una transformación usando las buenas correspondencias.
4. Iterar.

3.4.2. Registro basado en características

1. Utilizar los puntos característicos de SIFT [12].
2. Usar los descriptores de histogramas de puntos característicos (PFH) con los puntos característicos.
3. Obtener los descriptores de histogramas de puntos característicos (PFH) y estimar las correspondencias.
4. Rechazar las malas correspondencias usando uno más métodos.
5. Finalmente obtener la transformación.

Capítulo 4

Sistema propuesto y herramientas de desarrollo

En el presente trabajo se presentará la realización de un software que permite la construcción de un ambiente virtual, a través de información adquirida por medio de un sensor Kinect montado sobre el robot de servicio Justina, esto con la finalidad de tener una representación geométrica del mundo en el que se encuentra el robot, para posteriormente detectar las zonas transitables con el objetivo de poder diferenciar cuales son las zonas en las que dicho robot se podría situar. Además de poder probar algoritmos, haciendo uso de un simulador desarrollado en el laboratorio de Bio-Robótica, sin necesidad de usar al robot físico.

El sistema que se describe en este capítulo surgió como una primera solución a los problemas que se tienen en la construcción de ambientes virtuales y para ganar entendimiento de las dificultades que surgen. Es, por tanto, distante a los sistemas del estado del arte mencionados previamente.

El desarrollo del sistema lo haremos de manera modular con la finalidad de dar mayor versatilidad al programa y una gran robustez. Nuestro sistema está constituido por 5 módulos principales que corresponden al objetivo de esta tesis. El primer módulo se encarga de la adquisición de los datos de entrada del sistema (nubes de puntos) a través del sensor Kinect con el que cuenta el robot. El segundo módulo realiza el registro de varias nubes de puntos en una sola para obtener una imagen global de los datos adquiridos. El tercer módulo se encarga del muestreo de los datos para realizar los siguientes módulos de una manera más eficiente. El cuarto módulo se encarga de obtener el cierre cóncavo de los puntos formados por un plano paralelo al suelo y perpendicular al robot. Finalmente el quinto módulo se encargará de la visualización de las nubes de puntos como polígonos dentro del simulador.

4.1. Herramientas de desarrollo

Se utilizará C++ como lenguaje de programación y nos apoyaremos principalmente en las librerías de PCL, así como también se utilizará MATLAB.

4.1.1. Point Cloud Library

En la parte de la programación nos apoyaremos en librerías PCL, *Point Cloud Library* [28], es a gran escala un proyecto abierto para el procesamiento de nubes de puntos en 3D. Dichas librerías contienen numerosos algoritmos entre los que podemos distinguir algoritmos de filtrado avanzados, de estimación de funciones, reconstrucciones de superficies, registro de datos, ajustes de modelos y segmentación entre otros. En la Figura 4-1 podemos ver el logo de PCL.

Figura 4-1. Logo de PCL [28].

Estos algoritmos tienen diferentes usos como podrían ser el filtrado de valores atípicos, propios del error cometido por el hardware ante la captura de imágenes de un entorno; extracción de puntos clave de una imagen; crear superficies suavizadas del entorno o llevar a cabo una disminución de carga de imágenes reduciendo el número de puntos necesarios para formar dicha imagen sin perder información; entre otras finalidades.

PCL es un software de código abierto libre, además de ser gratuito para uso comercial e investigación.

PCL además es multiplataforma, ya que ha sido compilado y desplegado de manera correcta en Windows, Linux, MacOS, y Android/ iOS.

Para simplificar el desarrollo, PCL se divide en una serie de librerías de código más pequeño, para conseguir una fragmentación tal que permita simplificar el desarrollo de programas. Gracias a esta modularidad se logra una facilidad de compilación dentro de aquellas plataformas que no disponen de un potencial amplio para el compilado de programas complejos.

El formato de los datos con los que trabaja PCL son los archivos pcd. Este formato tiene intención de complementar los formatos existentes de nubes de puntos para llevar a cabo una mejor compatibilidad con PCL.

Cada archivo pcd (Figura 4-2) tiene una cabecera que identifica y declara ciertas propiedades de los datos de la nube de puntos contenido dentro de ese archivo. Dicho encabezado debe ser codificado en ASCII y contendrá la siguiente información:

- Versión: donde se especifica la versión del archivo .pcd.
- Campos: especifica el campo que cada punto debe contener. (XYZ, XYZRGB, etc.)
- Tamaño: especifica el tamaño de cada dimensión.
- Tipo: especifica el tipo de cada dimensión.
- Count: especifica cuantos elementos, tales como descriptores, tiene cada dimensión.
- Ancho: especifica el ancho del conjunto de datos.
- Alto: especifica el alto del conjunto de datos.
- Punto de toma de datos: especifica el punto de vista de adquisición del conjunto de datos.
- Número de puntos: especifica el número de puntos contenidos en la nube.

Las ventajas de este tipo de formato son principalmente su flexibilidad, su velocidad y la robustez que le da ser el archivo nativo de PCL, además de:

- La capacidad de almacenar y procesar conjuntos de datos organizados de nubes de puntos. Esto es de suma importancia para aplicaciones en tiempo real y áreas de investigación como la robótica, realidad aumentada, etc.
- Para binarios mmap / munmap tiene la forma más rápida de cargar y guardar datos en el disco.
- El almacenamiento de diferentes tipos de datos (todos los primitivos son soportados) permite que los datos de las nubes de puntos sean más flexibles y eficientes en lo que respecta a procesamiento y almacenamiento. Las dimensiones de puntos no validos se almacenan como NaN.
- Histogramas ND de descriptores de funciones (muy importantes para las aplicaciones de visión 3d de percepción.

A pesar de que PCD es el formato nativo de PCL, la biblioteca pcl_io ofrece la posibilidad de guardar y cargar archivos de otros formatos.

```
# .PCD v0.7 - Point Cloud Data file format
VERSION 0.7
FIELDS x y z
SIZE 4 4 4
TYPE F F F
COUNT 1 1 1
WIDTH 512
HEIGHT 1
VIEWPOINT 0 0 0 1 0 0 0
POINTS 512
DATA ascii
1.8117,2.6527,1.5551
1.7945,2.617,1.5896
1.7917,2.6211,1.6825
1.793,2.5618,1.6588
1.8021,2.6346,1.4359
1.7942,2.5729,1.4251
1.794,2.5485,1.5524
1.785,2.4815,1.5563
1.7916,2.6164,1.8361
1.7888,2.5643,1.8376
1.7851,2.5188,1.8352
1.7736,2.4626,1.8389
1.7831,2.4882,1.6869
1.773,2.4257,1.7157
1.7732,2.3906,1.6536
1.7748,2.3146,1.6742
1.8116,2.6562,1.1264
```

Figura 4-2. Ejemplo de archivo en formato PCD.

4.1.2. C++ y la programación orientada a objetos

En la realización de esta tesis utilizaremos el lenguaje de programación C++ diseñado en 1980 por Bjarne Stroustrup. La primera finalidad por la que se creó fue la extensión del exitoso lenguaje C para poder manipular objetos.

La programación orientada a objetos se creó buscando una programación estructurada cuya idea principal es separar las partes complejas de un programa en diferentes segmentos que sean ejecutados en función de las necesidades. Esto daría a una actividad modular diferente a la programación que generaba grandes cantidades de código en un solo bloque de manera lineal.

A diferencia de C, el cual se rige principalmente mediante funciones que se llaman unas a otras entre las que podemos intercambiar información, con C++ disponemos más versatilidad gracias a la existencia de clases. Dentro de estas clases podremos definir las propiedades y el comportamiento de un objeto en concreto. Los objetos son instancias de clases.

4.1.3. MATLAB

MATLAB (abreviatura de MATrix LABoratory, "laboratorio de matrices") [29] es una herramienta de software matemático con un lenguaje de programación propio (lenguaje M). MATLAB es multiplataforma y está disponible para las plataformas Unix, Windows y Mac OS X.

Algunas de sus funcionalidades básicas son: la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario (GUI) y la comunicación con programas en otros lenguajes y con otros dispositivos hardware.

Es un software muy usado en universidades y centros de investigación y desarrollo.

4.2. Desarrollo del sistema

En breve se describirán cada uno de los módulos mencionados al inicio de este capítulo y que conforman el trabajo medular de esta tesis.

4.2.1. Adquisición de las nubes de puntos

Como ya se ha mencionado en repetidas ocasiones, la adquisición de las nubes de puntos se hará mediante el sensor Kinect, con el que cuenta el robot Justina, En la Figura 4-3 se observa al robot Justina y donde se encuentra montado el sensor Kinect.

Para la adquisición de las nubes de puntos se debe de tener previamente tanto la posición del robot como la posición de la cabeza, que es donde se encuentra el sensor.

La adquisición de los datos se realiza por medio del software con el que ya cuenta el Robot Justina, el cual entrega un archivo rosbag, similar al archivo PCD descrito anteriormente. El software del robot de servicio Justina realiza internamente las transformaciones de las coordenadas locales a las coordenadas globales. Estas transformaciones serán descritas a continuación.

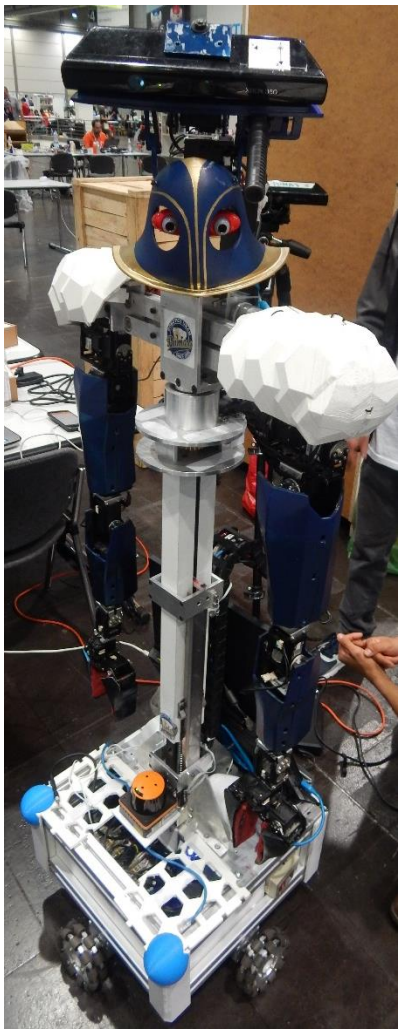


Figura 4-3. Posición del sensor Kinect sobre el robot Justina.

4.2.1.1. Transformación de la nube de puntos

Cada pixel dentro de la nube puntos contiene información en el espacio de 3 dimensiones. Esta posición está determinada en un sistema de referencia con origen O_0 en el centro del sensor Kinect. Los ejes ***X, Y*** y ***Z*** de este sistema de referencia apuntan hacia el frente, hacia la izquierda y hacia arriba, respectivamente (viendo al Kinect desde atrás), cuando los ángulos de la cabeza tienen valores de 0.

Debido a que el sensor está montado en la cabeza mecatrónica del robot, la cual tiene dos grados de libertad (tilt y pan) y ésta se mueve para apuntar hacia donde exista información relevante o de interés, el sistema de referencia de los puntos al sistema de referencia O_2 del robot, como se puede ver en la imagen 4-4.

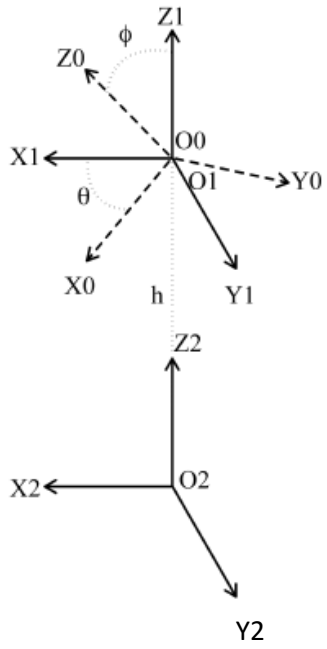


Figura 4-4. Sistemas de referencia del Kinect y del robot.

Para transformar un punto $p_{0i} = (x_{0i}, y_{0i,}, z_{0i})$, donde i representa su índice en una dimensión en la nube de puntos, del sistema $\boldsymbol{O_0}$, a su correspondiente p_{2i}, en el sistema $\boldsymbol{O_2}$, se utiliza la siguiente transformación homogénea, la cual involucra dos rotaciones seguidas de una traslación:

$$p_{2i} = \begin{pmatrix} cos\phi & 0 & sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -sin\phi & 0 & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & h \\ 0 & 0 & 0 & 1 \end{pmatrix} p_{0i} \qquad (4.1)$$

Si desarrollamos más la ecuación 4.1 tenemos:

$$\begin{cases} p_{2i}(x) = p_{0i}(x)cos\theta cos\phi - p_{0i}(y)sin\theta + p_{0i}(z)cos\theta sin\phi \\ p_{2i}(x) = p_{0i}(x)sin\theta cos\phi + p_{0i}(y)cos\theta + p_{0i}(z)sin\theta sin\phi \\ p_{2i}(x) = h - p_{0i}(x)sin\phi + p_{0i}(z)cos\phi \end{cases} \qquad (4.2)$$

Donde θ y ϕ son los ángulos de los movimientos de la cabeza, tilt y pan, respectivamente, y h es la altura a la que se encuentra la cabeza respecto al centro del robot. A partir de la ecuación 4.2 podemos establecer el algoritmo 4-1 para obtener el valor de cada punto $p_{2i}(x, y, z)$.

Algoritmo 4-1: Transformación de la nube de puntos

Entrada: Nube de puntos **P**

Salida: Nube de puntos **P'**

1. **Función** KINECT2ROBOT(P);
2. **for** all $p(x,y,z)\ \epsilon\ P$ **do**;
3. $p'(x) \leftarrow p(x)cos\theta cos\phi - p(y)sin\theta + p(z)cos\theta sin\phi$;
4. $p'(y) \leftarrow p(x)sin\theta cos\phi + p(y)cos\theta + p(z)sin\theta sin\phi$;
5. $p'(z) \leftarrow h - p(x)sin\phi + p(z)cos\phi$;
6. **return** P'

Algoritmo 4-1. Transformación de la nube de puntos.

Como se había mencionado con anterioridad, esta transformación de las nubes de puntos se realiza internamente en el software del robot de servicio Justina, por lo que cuando realizamos la adquisición de datos obtenemos un archivo con los datos necesarios que necesitamos para trabajar con los siguientes módulos.

Como se mencionó en el subtema anterior, el archivo obtenido es de tipo rosbag, pero para los siguientes módulos, por eficiencia nos conviene utilizar un archivo PCD, el cual contiene la misma información pero en un formato no tan pesado. Esto no significa ningún problema, ya que mediante un comando de ROS (Robot Operating System) que es el sistema operativo con el que trabaja el robot Justina, podemos transformar archivos rosbag a archivos PCD. Dicho archivo PCD será el archivo de entrada para los módulos posteriores que realizan el procesamiento de las nubes de puntos.

4.2.2. Registro de las nubes de puntos

Tal como se mencionó en el capítulo anterior, el registro de las nubes de puntos es un problema complejo y puede resolverse de varias formas, en nuestro caso nuestra solución utiliza el algoritmo ICP [11], descrito en capítulos anteriores, para incrementalmente hacer el registro de una serie de nubes de puntos de par en par. La implementación de este módulo utiliza la librería de PCL y el lenguaje C++; esta implementación toma un conjunto de nubes de puntos y las registra de par en par hasta obtener como resultado una sola nube de puntos global.

La idea principal es transformar todas las nubes de puntos en el cuadro de la primera nube de puntos. Esto se hace encontrando la mejor transformación entre cada nube consecutiva, y acumulando estas transformaciones sobre el conjunto total de las nubes.

En la figura 4-5, se muestra el emparejamiento de un par de nubes de puntos. Las nubes de puntos fuente a) y b) tienen cada una un total de 614400 puntos, mientras que en la imagen c) se puede observar su respectivo registro, con una cantidad de 1228800 puntos.

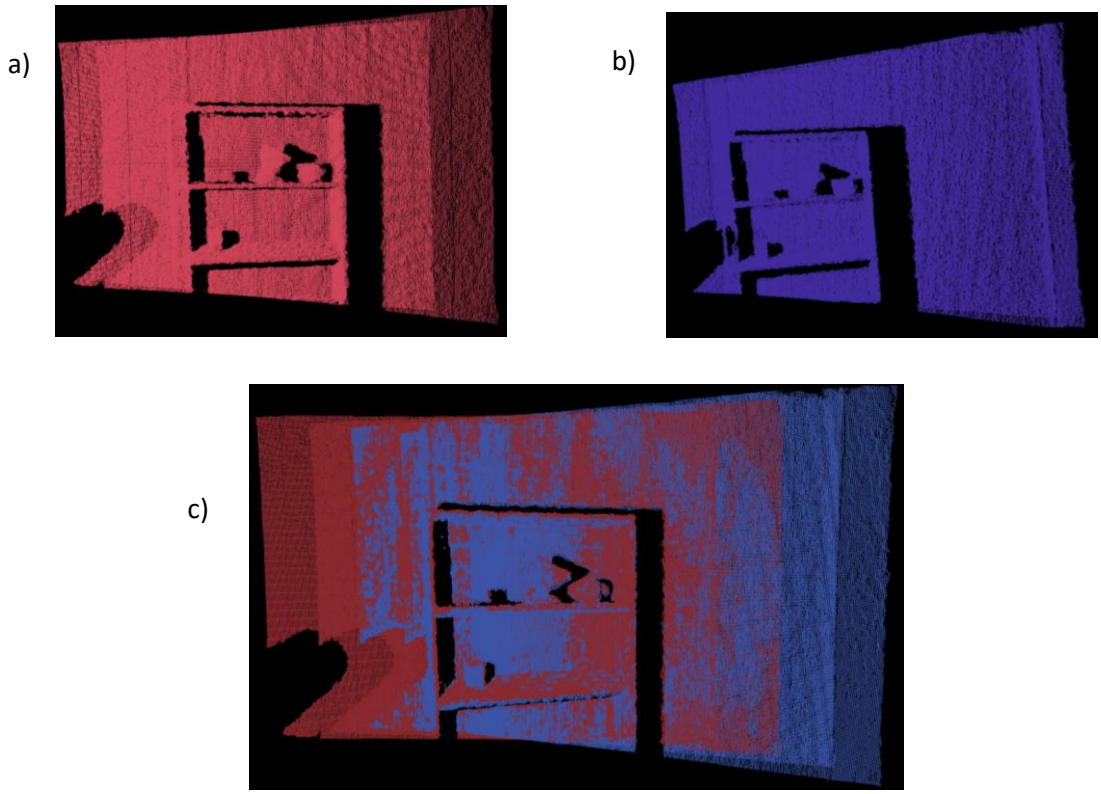


Figura 4-5. Ejemplo de registro de un par de nubes de puntos.

4.2.3. Muestreo de las nubes de puntos

Con la finalidad de optimizar la implementación del siguiente módulo, es factible realizar un muestreo de los datos y trabajar con una muestra significativa de los datos. Además, después de haber realizado el registro de las nubes de puntos, nos queda una nube de puntos de mucho mayor tamaño y por lo consiguiente con mayor cantidad de datos a procesar.

Para esté módulo hay dos métodos implementados, el primero es la voxelización, el cual es el método por excelencia, que se utiliza para realizar este tipo de muestreos, y el segundo es la cuantización vectorial, ambos descritos en capítulos anteriores.

2.3.1. Muestreo de las nubes de puntos por Voxelización

Para la implementación con este método utilizamos la librería de PCL y el lenguaje de C++. Esta implementación consiste básicamente en tomar la nube de puntos original y sobre de ella lanzar una cuadricula tridimensional, los puntos de la cuadricula que coincidan con los puntos de la nube de puntos original son los que formaran parte de la nueva nube de puntos muestreada. El tamaño del muestreo varía en función del tamaño del voxel deseado, es decir, el tamaño de cada uno de los cubos que forman la red, por lo que el tamaño final del muestreo es desconocido hasta la ejecución de la implementación.

Dentro de la implementación se dejó como un dato variable el tamaño del voxel, para así poder experimentar con varios muestreos. En la figura 4-6 podemos observar el comparativo entre la nube de puntos original obtenida con el Kinect a), la cual consta de 307200 puntos y entre un muestreo utilizando voxelización b), con una cantidad de 5544 puntos.

a)

b)

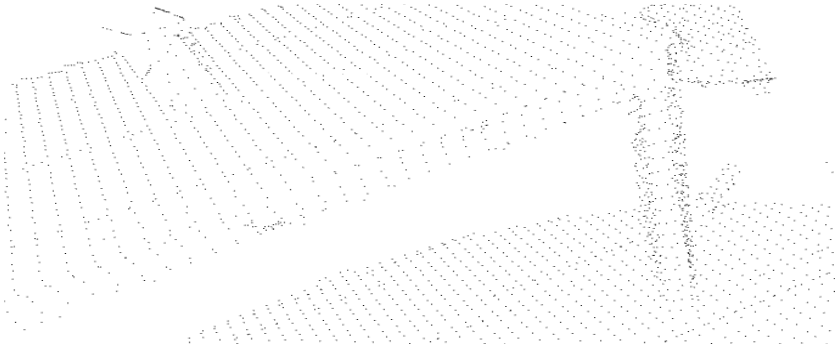


Figura 4-6. Muestreo con Voxelización.

4.2.3.2. Muestreo de las nubes de puntos por Cuantización Vectorial

Para implementar el muestreo de nubes de puntos por cuantización vectorial se optó por utilizar MATLAB y el algoritmo de Linde-Buzo-Gray descrito en el capítulo anterior. Esta implementación consiste en una técnica de clustering o agrupamiento de los puntos en regiones. Este agrupamiento se determina por la distancia euclidiana que existe entre los puntos y los centroides de las regiones. El tamaño del muestreo varía en función del número de regiones deseadas, este número de regiones solo puede ser un número potencia de 2, debido a la naturaleza del algoritmo. A diferencia del método anterior, el tamaño final de muestreo es totalmente conocido desde un inicio. Otra característica a mencionar es que, debido a la naturaleza del algoritmo, se espera una mejor distribución de los puntos muestreados con este método en comparación de la voxelización.

Dentro de la implementación se dejó como un dato variable el número de regiones en las que se muestreará la nube puntos, para así poder experimentar con varios muestreos. En la figura 4-7 podemos observar la comparación entre la nube de puntos original capturada con el Kinect a), la cual consta de 307200 puntos y un muestreo con voxelización de una nube de puntos con 1024 puntos en b).

a) b)

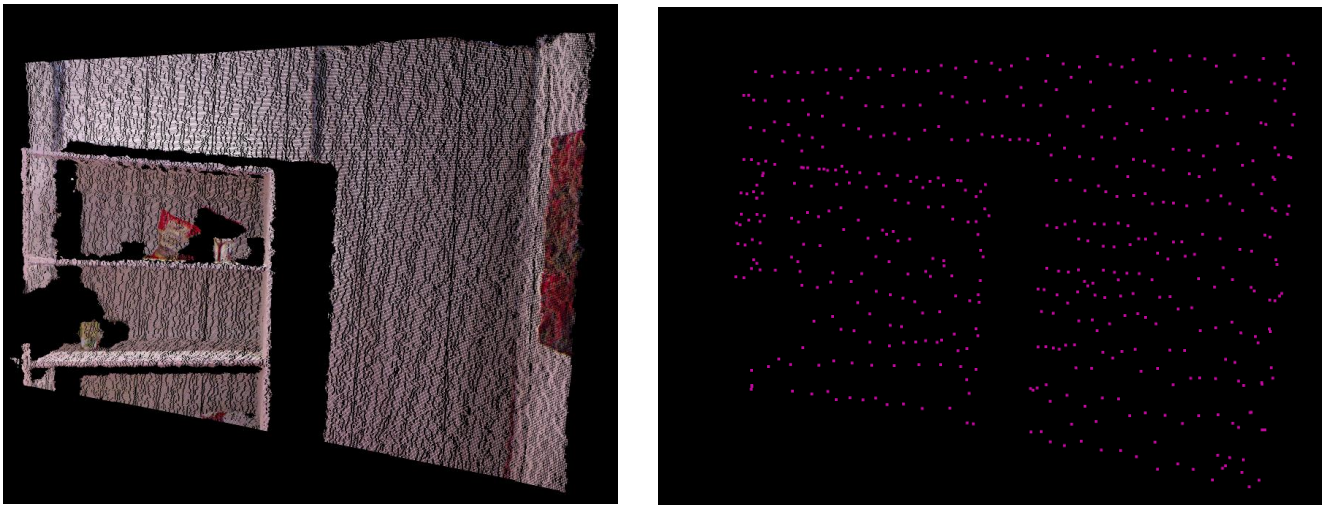


Figura 4-7. Cuantización vectorial con 1024 regiones.

4.2.4. Obtención del cierre cóncavo de la nube de puntos

Una vez que ya se tiene el muestreo de los datos, es necesario tener una representación simple de los datos que nos proporcione información sobre la forma de los objetos que se han capturado con el sensor Kinect, para posteriormente en el siguiente modulo tener una simulación con polígonos del mundo. Para ello se optó por obtener el cierre cóncavo de la nube de puntos, ya que la mayoría de las escenas en donde se desenvuelve el robot de servicio Justina genera un polígono no convexo. Para obtener dicho cierre cóncavo se llevarán a cabo los siguientes pasos, que fueron desarrollados en lenguaje C++ y con la ayuda de la librería de PCL.

El primer paso es proyectar la nube de puntos a un plano perpendicular al eje ***z***, por lo que hablamos del plano formado por "***x"*** y "***y",*** si recordamos el sistema de referencia en el que obtenemos las nubes de puntos, al momento de realizar dicha proyección lo que se obtendría es un plano paralelo al suelo, lo que nos daría como resultado es una especie de vista aérea de los objetos capturados por el sensor Kinect, en la que se podría apreciar el contorno de dichos objetos.

Para obtener el contorno descrito en el párrafo anterior, se deben de establecer los parámetros adecuados a la ecuación general de un plano dada por:

$$Ax + By + Cz + D = 0 \qquad (4.3)$$

Los valores de los parámetros de la ecuación quedarán establecidos de la siguiente manera:

$$\begin{cases} A = 0 \\ B = 0 \\ C = 1 \\ D = 0 \end{cases} \qquad (4.4)$$

Lo cual quiere decir que se proyectarán todos los puntos al plano ***XY.*** Es un método sencillo ya que solo se sustituirán las variables Z de los puntos por 0. La figura 4-8 muestra la proyección en el plano ***XY*** de una nube de puntos; en a) se observa la nube de puntos y en b) la proyección en el plano.

a) b)

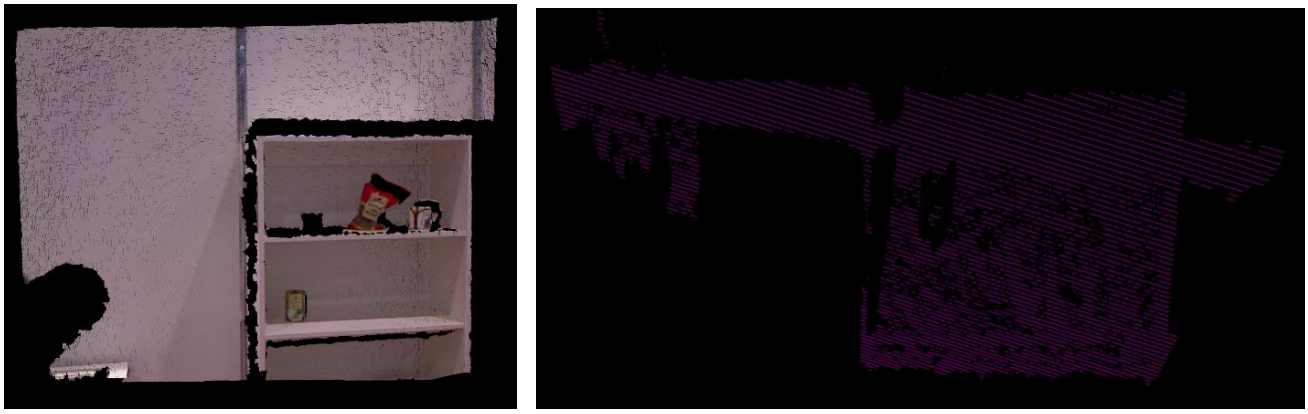


Figura 4-8. Proyección en el plano XY de una nube de puntos.

El siguiente paso es obtener el contorno de los puntos, mediante el cierre cóncavo de dichos puntos. Una vez que ya se cuenta con la proyección de los puntos en el plano ***XY***, obtener el cierre cóncavo se simplifica aún más, ya que la librería de PCL ya cuenta con una función que obtiene el cierre cóncavo. Dicha función tiene como entrada una nube de puntos; en nuestro caso será la nube de puntos proyectada en el plano, y está función devuelve el conjunto de puntos que forman el cierre cóncavo de los puntos que tomo como entrada.

4.2.5. Visualización de los datos en el simulador del robot de servicio Justina

Esté quinto y último modulo toma como entrada el conjunto de puntos que conforman el cierre cóncavo y mediante algunos algoritmos de graficación por computadora hace un renderizado tridimensional mediante polígonos, de esta forma los datos capturados por el Kinect (nubes de puntos) podrán ser visualizados como polígonos tridimensionales dentro del simulador con el que cuenta el robot de servicio Justina. El método para observar la visualización de los datos dentro del simulador del robot de servicio Justina será explicado brevemente a continuación.

La representación de los objetos que se consideren dentro del ambiente virtual debe de ser flexible en cuanto a su configuración, lo cual se refiere a que la distribución espacial de los objetos necesita estar dentro de un archivo que sirva para representar el mundo en el que el robot interactúa. Este archivo posee la extensión "wrl" y en él están contenidos los puntos que conforman el cierre cóncavo de las nubes de puntos. Estos puntos representan los vértices de los polígonos con los que se representan los objetos. Existen dos características principales para dichos polígonos, las cuales se describen a continuación:

- Los objetos están formados por polígonos simples, los cuales a su vez se encuentran formados por vértices interconectados entre sí mediante segmentos de línea.
- Con el objetivo de tener una visualización de los objetos en tres dimensiones, basta con establecer una altura en el eje Z.

Es preciso mencionar que los objetos se encuentran definidos por polígonos cóncavos en el plano z=0. En la figura 4-9 se muestra un ejemplo de cómo se visualizan los objetos en el plano z=0. El objetivo es crear una representación de estos aumentándoles una altura en el eje Z y así obtener un objeto tridimensional.

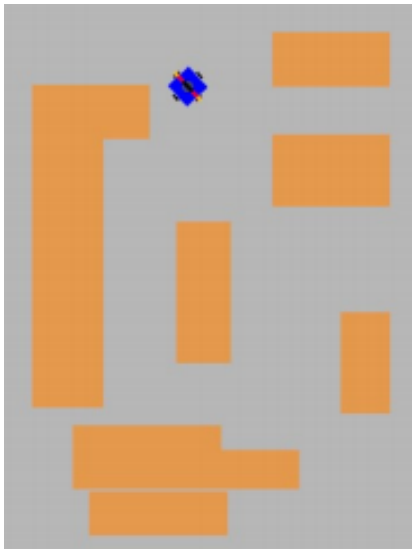


Figura 4-9. Vista de los objetos en el plano z=0.

Una vez que se cuenta con los vértices que forman a los objetos, lo siguiente es crear los triángulos que conforman la tapa inferior y superior de los objetos tridimensionales. Es importante mencionar que únicamente se cuenta con los vértices de los polígonos, por lo que se necesita realizar una triangulación a estos vértices para poder obtener sólidos en el simulador.

La triangulación se realiza con una implementación en C++ y con la ayuda de una librería llamada CGAL, la cual es una librería de C++ que proporciona algoritmos de geometría computacional [30].

Los triángulos obtenidos son utilizados para crear las tapas inferior y superior de los objetos, las tapas inferiores son las que se encuentran en el plano z=0, mientras que las superiores se encuentran en el plano que posee la altura del objeto.

Para crear los triángulos que forman las caras laterales de los objetos se asume que los vértices vecinos se encuentran ordenados de manera horaria. Los triángulos se crean de la siguiente manera: para cada polígono se recorre el arreglo de sus vértices creando dos triángulos en cada iteración, el primer triangulo se forma tomando el vértice en el que se encuentra la iteración, el segundo vértice corresponde a su vecino en el arreglo de vértices, el tercero es el vértice actual pero con la componente z igual a la altura del objeto.

El segundo triangulo se crea tomando como primer elemento el vértice vecino de la iteración actual, el segundo elemento corresponde a este mismo vértice pero su componente en z igual a la altura del objeto y el último elemento del triángulo es igual al vértice de la iteración actual con su componente z igual a la altura del objeto. Esta implementación es una especie de cosido entre los vértices de las caras superior e inferior de los objetos, en la figura 4-10 se puede observar la visualización de objetos tridimensionales en el simulador del robot de servicio Justina.

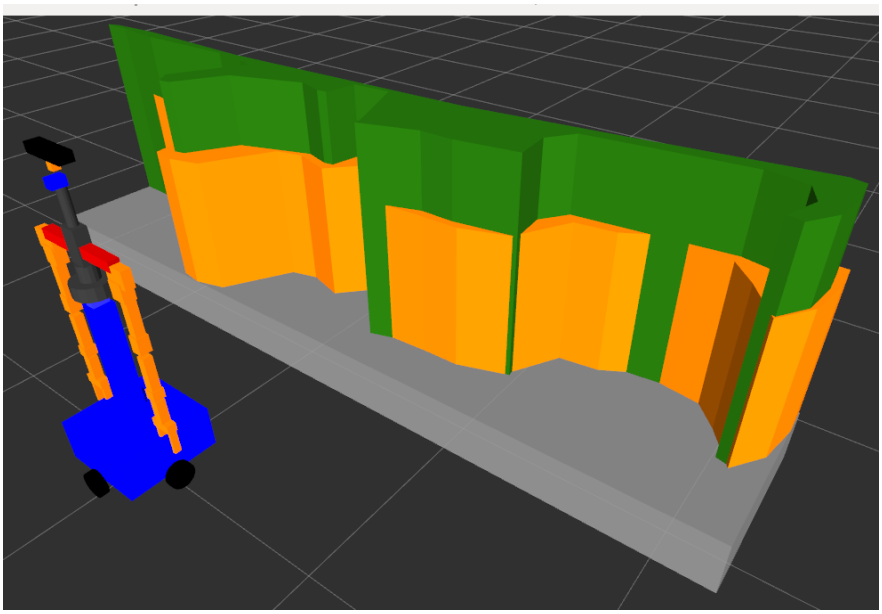


Figura 4-10. Visualización de objetos tridimensionales dentro del ambiente gráfico.

Capítulo 5

Experimentos y resultados

Para probar el sistema descrito en el capítulo anterior, realizamos el experimento que se describe a continuación y cuyos resultados también son reportados en este capítulo.

Con la ayuda del robot de servicio Justina y con el sensor Kinect que está montado sobre su cabeza, con dos grados de libertad, tilt (de arriba a abajo) y pan (de izquierda a derecha), se capturaron 10 nubes de puntos en un ambiente simulando una casa en la RoboCup 2016 llevada a cabo en Leipzig, Alemania. En la figura 5-1 se puede observar el ambiente que se capturó, en a) se muestra la parte izquierda del ambiente y en b) se muestra la parte derecha.

a)

b)

Figura 5-1. Ambiente de pruebas.

Las capturas de las nubes de puntos fueron de la siguiente manera: el robot se colocó en una posición fija y conocida; la cabeza, que posee dos grados de libertad (tilt y pan) con un rango de -1 a 1 radian en ambos movimientos, fue movida de la siguiente manera, el movimiento de tilt se decidió dejar en un valor fijo para cada una de las 10 capturas, el cual fue de -0.5 radianes, por otra parte el movimiento de pan fue disminuyendo la cantidad de 0.1 radian en cada captura, iniciando en 0.5 radian y terminando en -0.4 radian. La tabla 5-1 muestra los valores de la posición de la cabeza en cada una de las 10 capturas. La figura 5-2 muestra la captura número 1en a) y la captura número 10 en b) que se obtuvieron con el sensor Kinect en dichas posiciones.

Numero de captura	*Posición en tilt [radianes]*	*Posición en pan [radianes]*
1	-0.5	0.5
2	-0.5	0.4
3	-0.5	0.3
4	-0.5	0.2
5	-0.5	0.1
6	-0.5	0
7	-0.5	-0.1
8	-0.5	-0.2
9	-0.5	-0.3
10	-0.5	-0.4

Tabla 5-1. Resumen de las tomas utilizadas en el experimento.

a)  b) 

Figura 5-2. Capturas realizadas con el Kinect.

Con las capturas de las nubes de puntos antes descritas se realizó el registro de parejas de nubes de puntos, este registro como su nombre lo dice se realiza por parejas tal como se mencionó en capítulos anteriores, el emparejamiento se realiza por etapas, cada una de las etapas se realizó de la siguiente forma: la captura 1 con la captura 2, la captura 2 con la captura 3 y así sucesivamente hasta la captura 9 con la captura 10. Este emparejamiento arroja nuevas nubes de puntos que ahora van de la captura 1 hasta la captura 9, es decir el número de capturas se ha reducido en una unidad, este proceso se repite tantas etapas como sean necesarias para que al final solo se tenga una sola nube de puntos como resultado. La figura 5-3 ejemplifica las etapas que se llevaron a cabo para obtener la nube de puntos final donde la nomenclatura que se utiliza en la imagen es la siguiente: $\#\ de\ la\ captura_{\#\ de\ la\ etapa}$.

$$\left.\begin{matrix} Captura\ 1_1 \\ Captura\ 2_1 \end{matrix}\right\} Captura\ 1_2 \qquad \left.\begin{matrix} Captura\ 1_2 \\ Captura\ 2_2 \end{matrix}\right\} Captura\ 1_3$$

$$\left.\begin{matrix} Captura\ 2_1 \\ Captura\ 3_1 \end{matrix}\right\} Captura\ 2_2 \qquad \left.\begin{matrix} Captura\ 2_2 \\ Captura\ 3_2 \end{matrix}\right\} Captura\ 2_3$$

$$\left.\begin{matrix} Captura\ 3_1 \\ Captura\ 4_1 \end{matrix}\right\} Captura\ 3_2 \qquad \left.\begin{matrix} Captura\ 3_2 \\ Captura\ 4_2 \end{matrix}\right\} Captura\ 3_3$$

$$\left.\begin{matrix} Captura\ 4_1 \\ Captura\ 5_1 \end{matrix}\right\} Captura\ 4_2 \qquad \left.\begin{matrix} Captura\ 4_2 \\ Captura\ 5_2 \end{matrix}\right\} Captura\ 4_3$$

$$\left.\begin{matrix} Captura\ 5_1 \\ Captura\ 6_1 \end{matrix}\right\} Captura\ 5_2 \qquad \left.\begin{matrix} Captura\ 5_2 \\ Captura\ 6_2 \end{matrix}\right\} Captura\ 5_3 \qquad \ldots\ldots \left.\begin{matrix} Captura\ 1_9 \\ Captura\ 2_9 \end{matrix}\right\} Captura\ 1_{10}$$

$$\left.\begin{matrix} Captura\ 6_1 \\ Captura\ 7_1 \end{matrix}\right\} Captura\ 6_2 \qquad \left.\begin{matrix} Captura\ 6_2 \\ Captura\ 7_2 \end{matrix}\right\} Captura\ 6_3$$

$$\left.\begin{matrix} Captura\ 7_1 \\ Captura\ 8_1 \end{matrix}\right\} Captura\ 7_2 \qquad \left.\begin{matrix} Captura\ 7_2 \\ Captura\ 8_2 \end{matrix}\right\} Captura\ 7_3$$

$$\left.\begin{matrix} Captura\ 8_1 \\ Captura\ 9_1 \end{matrix}\right\} Captura\ 8_2 \qquad \left.\begin{matrix} Captura\ 8_2 \\ Captura\ 9_2 \end{matrix}\right\} Captura\ 8_3$$

$$\left.\begin{matrix} Captura\ 9_1 \\ Captura\ 10_1 \end{matrix}\right\} Captura\ 9_2$$

Figura 5-3. Etapas para realizar el registro de nubes de puntos.

La nube de puntos final después de haber hecho las 10 etapas del proceso descrito en el párrafo anterior se puede observar en la figura 5-4, en dicha imagen cada color representa a cada una de las capturas originales. Se puede observar que el registro se realizó con éxito, sin embargo el tamaño de la nube de puntos es bastante grande, ya que si cada una de las nubes de puntos a partir de las que se hizo el registro contiene una cantidad de 307200 puntos, al momento de realizar el registro esta cantidad crece de manera exponencial, hasta llegar a tener una cantidad de 314572800 puntos, lo cual representa una gran cantidad de datos, por lo que es necesario reducir su tamaño mediante un muestreo de tal forma que

obtengamos un archivo con menor información y que facilite su posterior procesamiento.

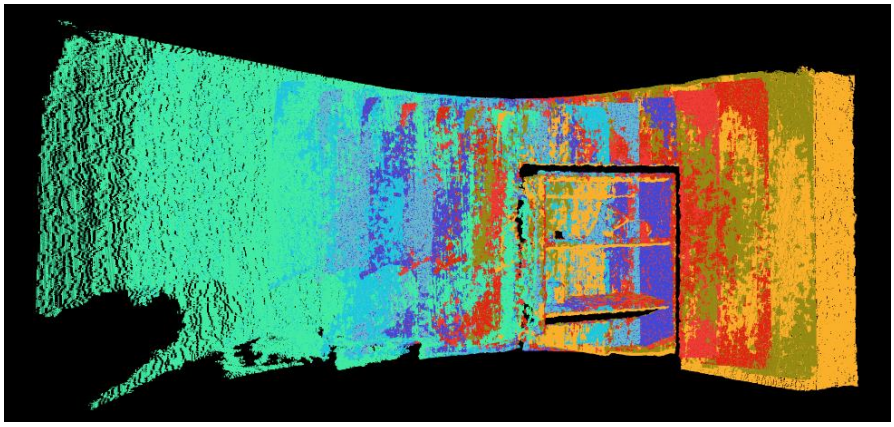


Figura 5-4. Registro de las nubes de puntos.

En el párrafo anterior mencionamos que era necesario hacer un muestreo de la nube de puntos, por lo que se realizaron varios muestreos con los dos métodos descritos en capítulos anteriores, los cuales son voxelización y cuantización vectorial, con la final de probar dichos métodos y decir cuál de ellos tiene un mejor desempeño.

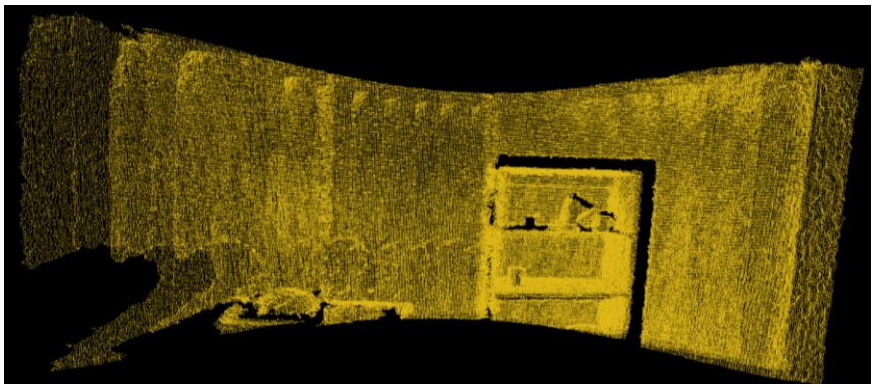
La prueba de los dos métodos de muestreo consistió en obtener muestreos de la nube de puntos en diferentes escalas, por ejemplo, en el caso de cuantización vectorial que podemos controlar el tamaño final de la nube de puntos se decidió variar el tamaño de dicha nube de la siguiente forma: 64, 128, 256, 512 y 1024 puntos; por otra parte en el método de voxelización al no poder controlar directamente el tamaño final del muestreo sino el tamaño de cada voxel, este tamaño fue el que fue variado en la implementación hasta obtener muestras que fueran similares en tamaño a las obtenidas con el método de cuantización vectorial. En la tabla 5-2 se puede observar un resumen de las pruebas mencionadas en los párrafos anteriores.

Número de experimento	*Cuantización vectorial [número de regiones]*	*Voxelización [número de voxeles]*	*Voxelización [tamaño de voxel en cm]*
1	64	67	0.57
2	128	129	0.37
3	256	225	0.25
4	512	505	0.18
5	1024	1077	0.12

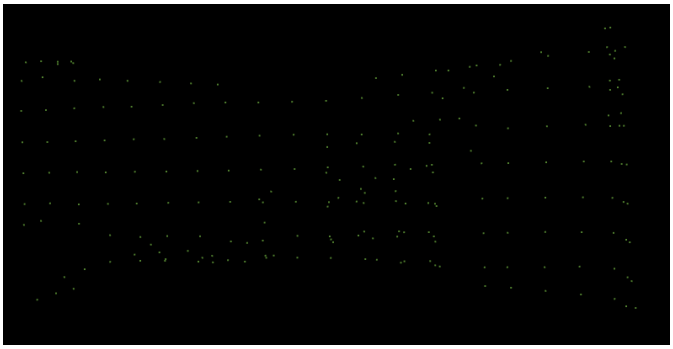
Tabla 5-2. Resumen de los experimentos de muestreo.

En la figura 5-5 en a) se muestra la comparación del registro obtenido contra el muestreo con voxelización en varias resoluciones b) 225. c) 505 y d) 1077 voxeles respectivamente.

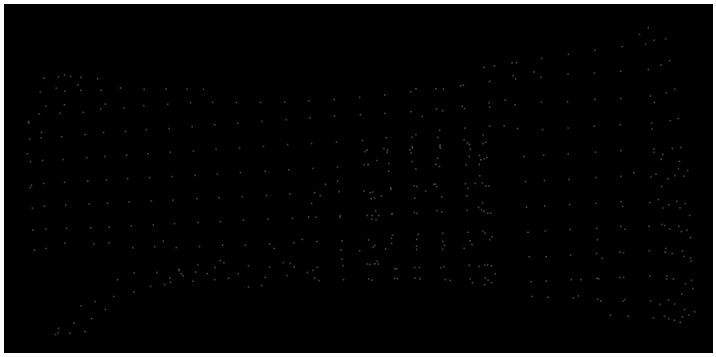
a)



b)



c)



d)

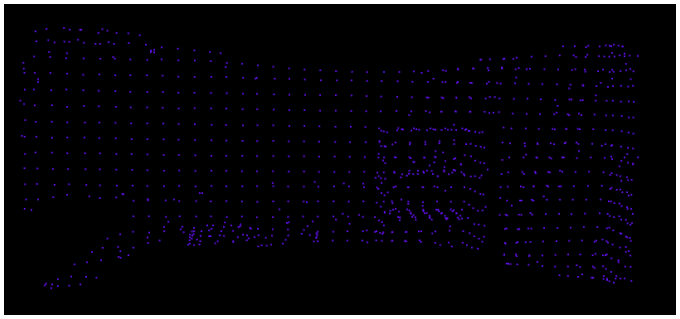
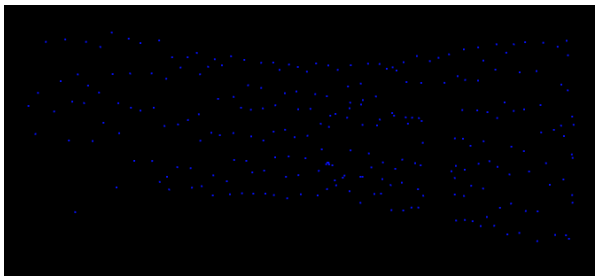


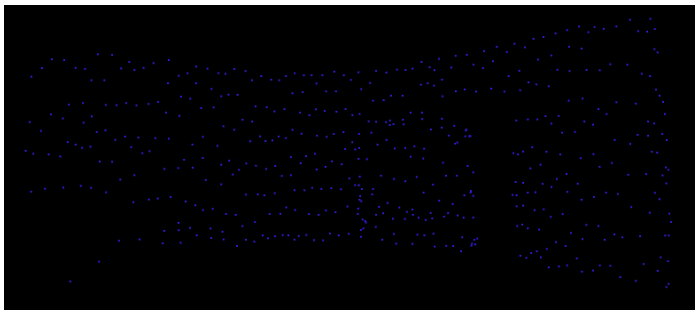
Figura 5-5. Muestreo por voxelización en distintas resoluciones.

En la figura 5-6 en d) se observa la comparación del registro obtenido contra los muestreos realizados con cuantización vectorial utilizando a) 256, b) 512 y c) 1024 regiones respectivamente, recordando que el ambiente que se eligió para realizar el registro es el que aparece en la imagen 5-1.

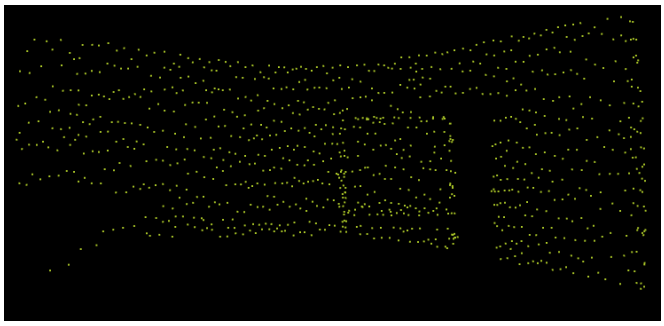
a)



b)



c)



d)

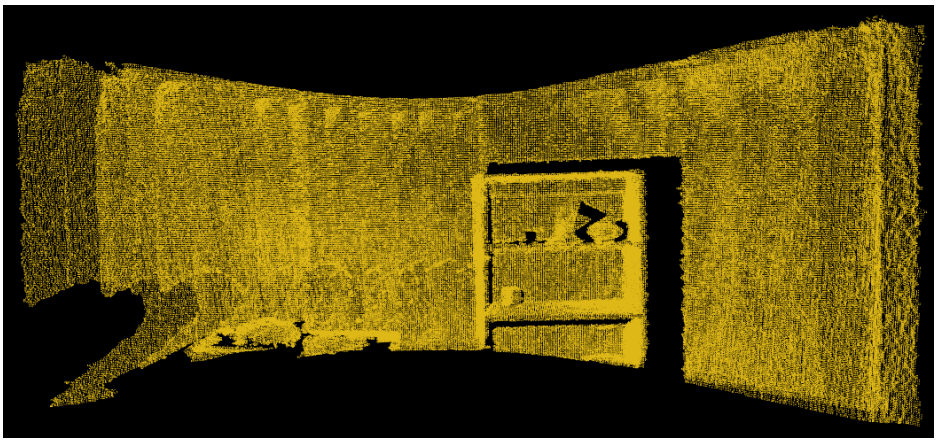


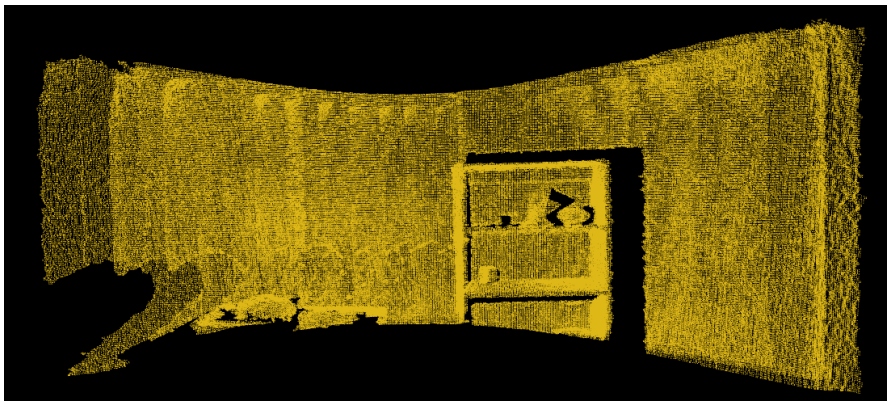
Figura 5-6. Muestreo por cuantización vectorial.

En las dos imágenes anteriores se puede apreciar que el muestreo de los datos fue exitoso con ambos métodos, ya que no se pierde la forma del ambiente capturado y la disminución de los datos es enorme, ya que pasamos del orden de millones de puntos al orden de miles o cientos.

Al observar los resultados de ambos métodos, el principal cambio que se nota es en la distribución de los datos, ya que en el método de voxelización se aprecia que el muestreo se realiza de una forma uniforme sobre toda la superficie, en cambio si se utiliza cuantización vectorial la distribución de los datos se realiza de manera diferente, mayor cantidad de puntos se observa en las zonas con objetos más grandes.

La siguiente fase del experimento consiste en obtener el cierre cóncavo de los muestreos realizados anteriormente, tanto para voxelización como para cuantización vectorial. El cierre cóncavo se obtuvo para cada uno de los experimentos resumidos con anterioridad en la tabla 5-2, así como para el registro original sin realizarle ningún tipo de muestreo, en la figura 5-7 en b) se muestra el cierre cóncavo de la nube de puntos original (registro) mostrada en a).

a)



b)

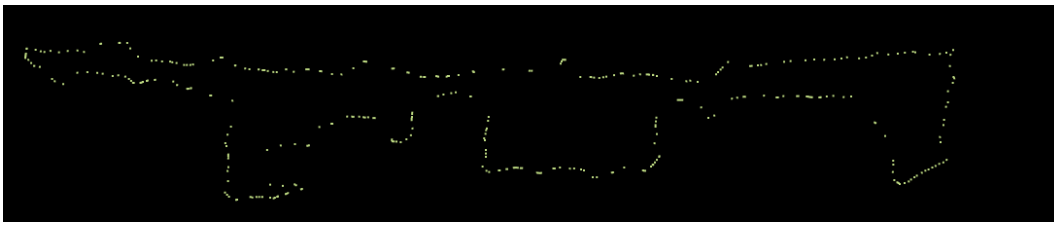


Figura 5-7. Cierre cóncavo de una nube de puntos.

En la figura 5-8 se muestra la comparación del cierre cóncavo de la nube de puntos (registro) contra el cierre cóncavo obtenido con las nubes de puntos muestreadas por voxelización en distintas resoluciones. El cierre cóncavo original es el que se puede observar en a), enseguida se pueden observar los cierres cóncavos de los muestreos por voxelización con las siguientes resoluciones: b) 225, c) 505 y d) 1077 voxeles respectivamente.

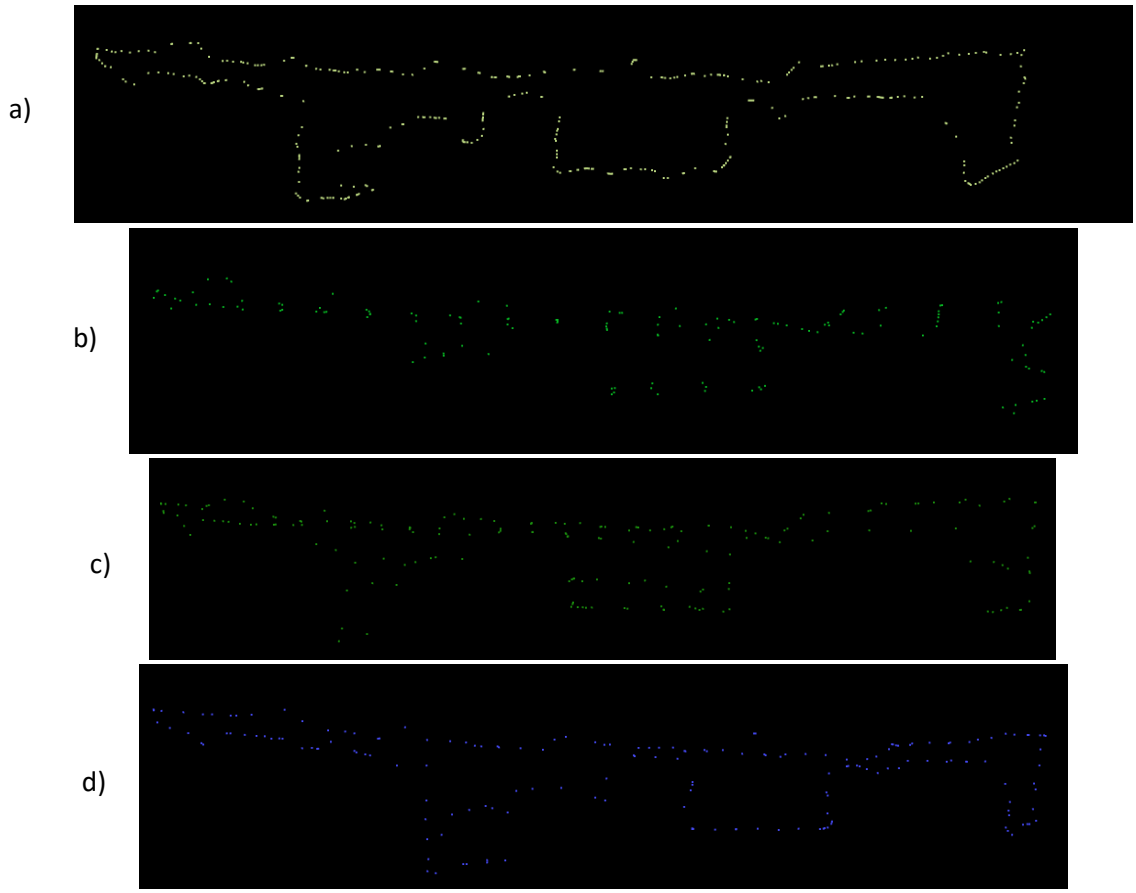


Figura 5-8. Comparativa entre el cierre cóncavo original y el cierre cóncavo de los muestreos por voxelización.

En la imagen se puede observar que conforme la resolución del muestreo aumenta, el resultado obtenido es mejor, ya que se puede apreciar que el cierre cóncavo obtenido con la voxelización con una resolución de 1077 voxeles mostrado en d) se asemeja mucho al cierre cóncavo original mostrado en a).

A continuación se mostrarán los resultados obtenidos al realizar el experimento anterior pero ahora utilizando el método de cuantización vectorial. En la figura 5-9 se muestra la comparación del cierre cóncavo de la nube de puntos (registro) contra el cierre cóncavo obtenido con las nubes de puntos muestreadas por cuantización vectorial en distinto número de regiones. El cierre cóncavo original es el que se puede observar en a), enseguida se pueden observar los cierres cóncavos de los muestreos por cuantización vectorial con el siguiente número de regiones: b) 256, c) 512 y d) 1024 regiones respectivamente.

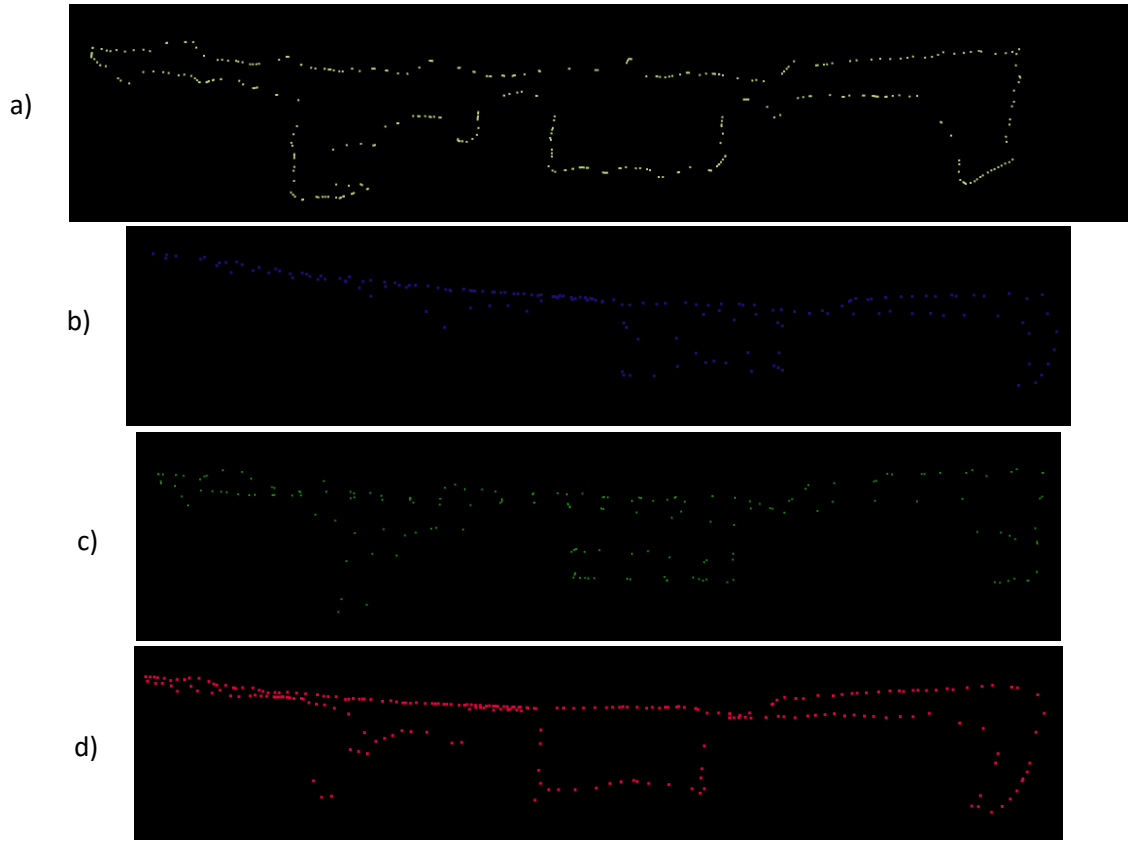


Figura 5-9. Comparativa entre el cierre cóncavo original y el cierre cóncavo de los muestreos por cuantización vectorial.

Al igual que con el método anterior, podemos decir que en la imagen se puede observar que conforme el número de regiones aumenta el resultado obtenido es mejor, ya que se puede apreciar que el cierre cóncavo obtenido con el muestreo por cuantización vectorial con un número de 1024 regiones mostrado en d) se asemeja mucho al cierre cóncavo original mostrado en a).

Hasta ahora se ha mostrado visualmente que con ambos métodos se obtiene un alto grado de parentesco en la forma obtenida, pero no solo es importante que se mantenga la forma del ambiente sino que además se obtenga una disminución considerable en la cantidad de datos necesarios para conservar ese parentesco. La tabla 5-3 muestra la cantidad de puntos que conforman a cada uno de los cierres cóncavos involucrados en los pasados experimentos. Los experimentos del 2 al 6 se refieren a experimentos realizados con el muestreo por voxelización, los experimentos del 7 al 11 se refieren a experimentos realizados con muestreo por cuantización vectorial; por ejemplo, para el experimento 1, el cual consta del registro sin aplicarle ningún tipo de muestreo se tiene una cantidad de 1568 puntos en su cierre cóncavo; para el experimento 6 que consta de aplicar una voxelización al registro original con 1077 voxeles., nos da como resultado un cierre cóncavo con 193 puntos; para el experimento 10 que consta de aplicar cuantización vectorial al registro original con 1024 regiones se obtienen 187 puntos en el cierre cóncavo.

Número de experimento	**Experimento**	**Cantidad de puntos del cierre cóncavo**
1	Registro original	1568
2	Voxelización [67 voxeles]	47
3	Voxelización [129 voxeles]	112
4	Voxelización [225 voxeles]	142
5	Voxelización [505 voxeles]	178
6	Voxelización [1077 voxeles]	193
7	Cuantización vectorial [64 regiones]	49
8	Cuantización vectorial [128 regiones]	119
9	Cuantización vectorial [256 regiones]	155
10	Cuantización vectorial [512 regiones]	180
11	Cuantización vectorial [1024 regiones]	187

Tabla 5-3. Resumen de los experimentos de la obtención del cierre cóncavo.

Si se toman en cuenta los experimentos de voxelización con una resolución de 1077 voxeles y el de cuantización vectorial con una cantidad de 1024 regiones, los cuales fueron los que visualmente tienen un mejor resultado, tenemos 8.12 y 8.38 veces menor cantidad de datos respectivamente. Con lo cual se puede establecer que el experimento ha sido exitoso hasta ahora.

Una vez establecido que la información muestreada realmente ha cumplido la meta de reducir una cantidad considerable la cantidad de datos y que esta disminución de datos no ha afectado en gran medida a la característica de la forma del ambiente capturado, ahora se mostrara el resultado final del experimento de este sistema, en cual consiste en mostrar en el simulador del robot Justina el ambiente capturado y procesado por el sistema presentado en esta tesis.

A continuación, en la figura 5-10 se mostrarán los resultados del experimento 1 de la tabla en comparación a los del experimento 6. Se renderizaron los dos polígonos en el simulador del robot de servicio de Justina, el de menor altura y en color amarillo es el resultado del experimento 1 (registro original) y el de mayor altura es el del experimento 6 (voxelización con 1077 puntos). Adicionalmente en la misma figura se puede apreciar la comparativa entre el ambiente real mostrado en d) y varias tomas del ambiente simulado en a), b) y c).

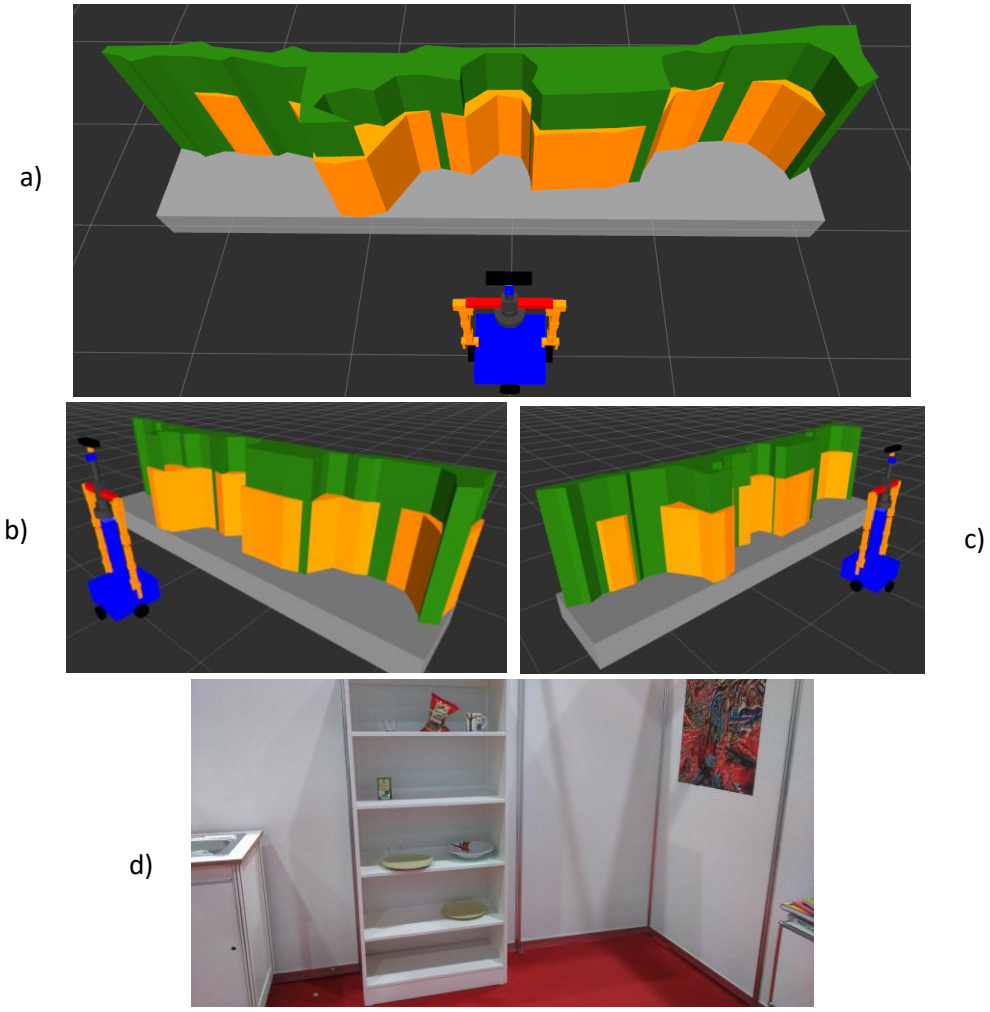


Figura 5-10. Comparativa entre el ambiente real y el ambiente simulado en el experimento 6 (voxelización con 1077 puntos).

A continuación en la figura 5-11 se muestra la comparativa entre los experimentos 4, 5 y 6, los cuales corresponden a experimentos muestreados con voxelización, con a) 225, b) 505 y c) 1077 puntos respectivamente. De igual forma que en la imagen anterior el polígono amarillo representa el registro original y el polígono verde representa a los experimentos 4, 5 y 6. Se puede observar que conforme aumenta la cantidad de puntos, la definición del polígono verde es mejor.

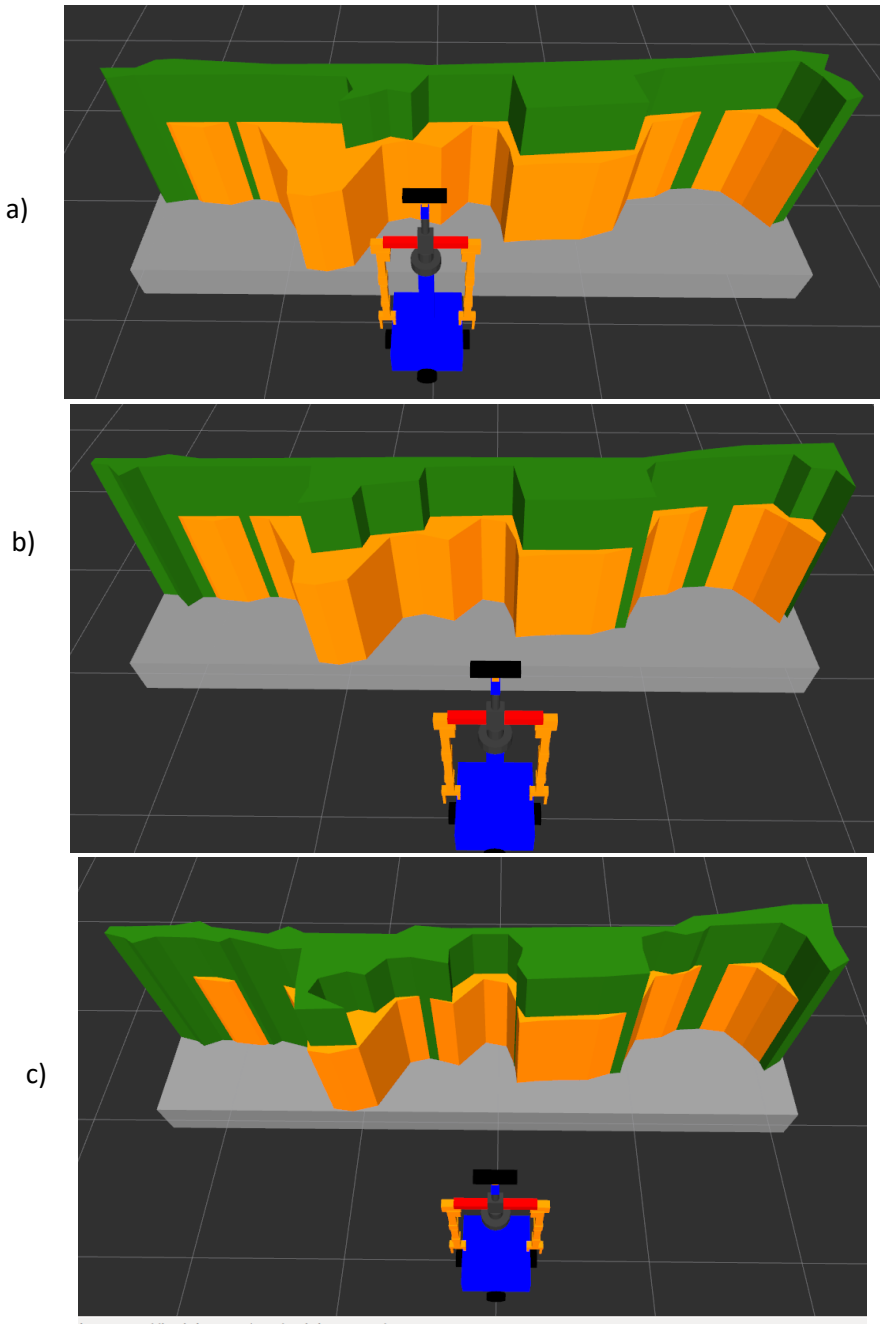
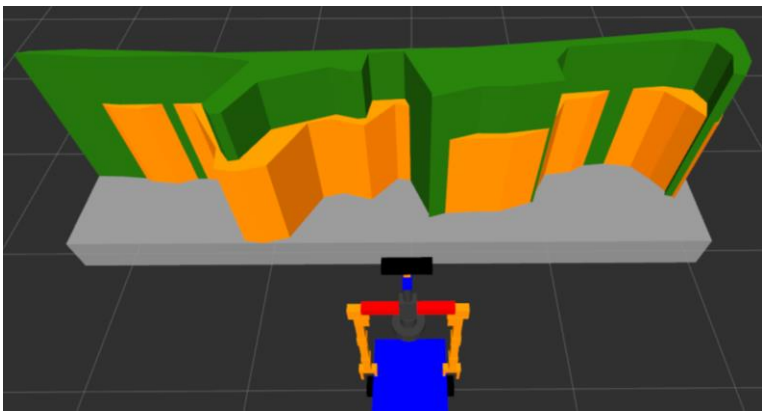


Figura 5-11. Comparativa entre los experimentos realizados con voxelización (a) 225, b) 505 y c) 1077 puntos respectivamente).

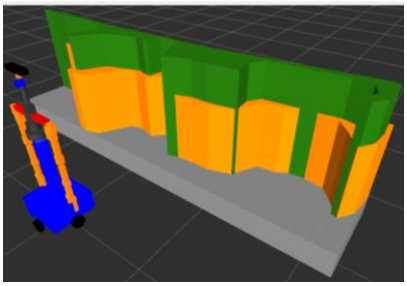
A partir de las figura 5-11 se puede establecer que el mejor resultado obtenido con el método de voxelización es el obtenido en el experimento 6, es decir utilizando el método de voxelización con una resolución de 1077 voxeles, ya que se puede observar que los polígonos casi se superponen en su totalidad.

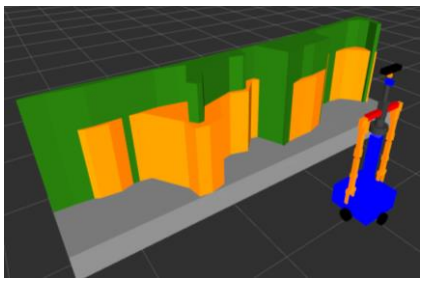
Ahora se procederá a mostrar los resultados obtenidos con el método de cuantización vectorial. En la figura 5-12 en a), b) y c) se muestra el resultado de los experimentos 1 y 11, donde el polígono amarillo representa al registro original y el polígono verde al experimento muestreado por cuantización vectorial con 1024 regiones; así mismo, en d) se muestra el ambiente real.

a)



b)





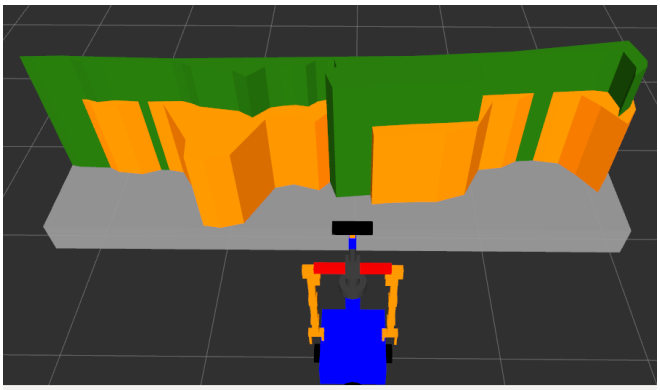
c)

d)

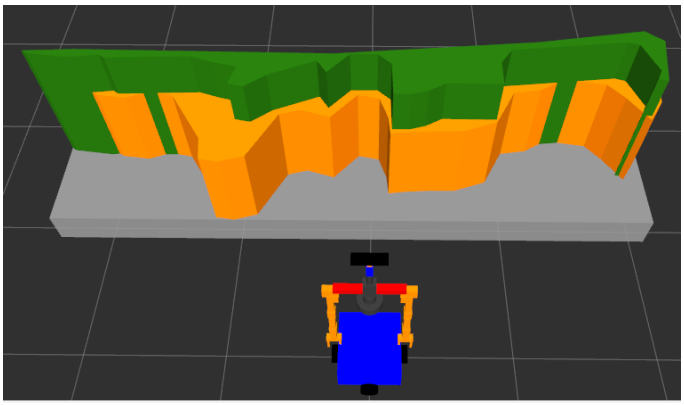
Figura 5-12. Comparativa entre el ambiente real d) y el ambiente simulado en el experimento 11 a), b) y c) (cuantización vectorial con 1024 puntos).

A continuación en la figura 5-13 se muestra la comparativa entre los experimentos 9, 10 y 11, los cuales corresponden a experimentos muestreados con cuantización vectorial, con a) 256, b) 512 y c) 1024 puntos respectivamente. De igual forma que en la imagen anterior el polígono amarillo representa el registro original y el polígono verde representa a los experimentos 9, 10 y 11. Se puede observar que conforme aumenta la cantidad de puntos la definición del polígono verde es mejor.

a)



b)



c)

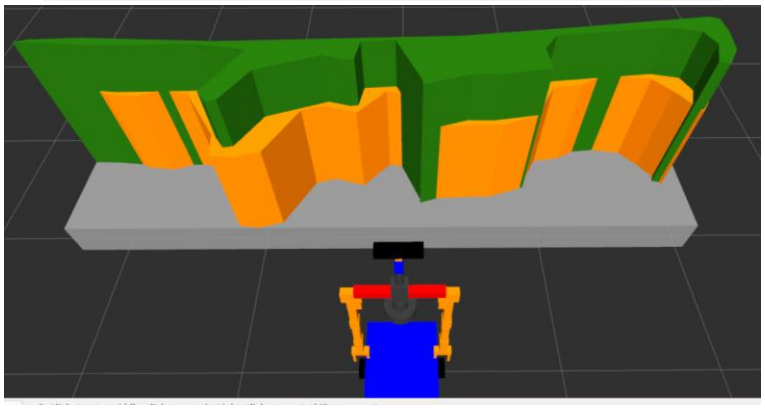
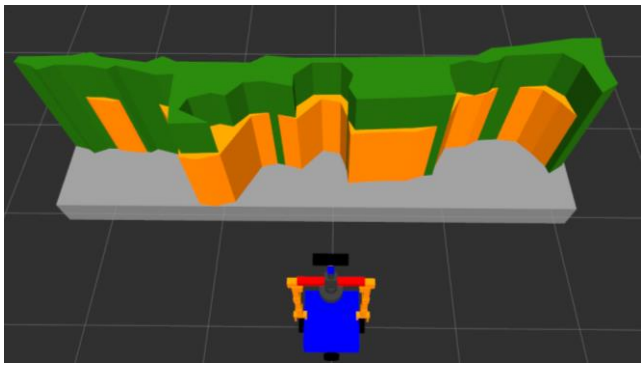


Figura 5-13. Comparativa entre los experimentos realizados con cuantización vectorial (a) 256, b) 512 y c) 1024 puntos respectivamente).

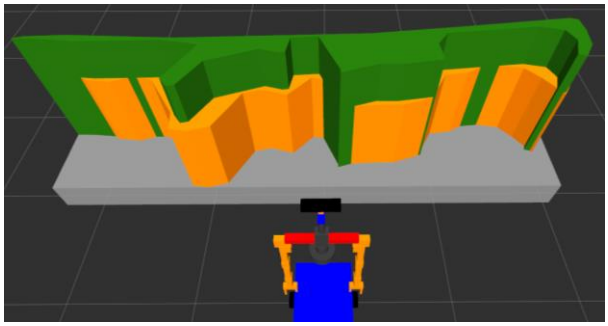
A partir de las figura 5-13 se puede establecer que el mejor resultado obtenido con el método de cuantización vectorial es el obtenido en el experimento 11, es decir, utilizando el método de cuantización vectorial con un numero de 1024 regiones, ya que se puede observar que los polígonos casi se superponen en su totalidad.

Entonces podemos decir que los experimentos 6 (voxelización con 1077 puntos) y 11 (cuantización vectorial con 1024 puntos) fueron los más exitosos de las pruebas realizadas y descritas en este capítulo, no solo porque se cumplió con el objetivo de reducir la cantidad de datos en un razón de 8 veces sino que también se logró conservar la forma casi en su totalidad, lo cual se pretendía demostrar en la presente tesis. En la figura 5-14 se puede observar la comparativa entre los experimentos 6 mostrado en a), 11 mostrado en b) y el ambiente real mostrado en c).

a)



b)



c)

Figura 5-14. Comparativa entre los experimentos 6 en a), 11 en b) y el ambiente real en c).

Capítulo 6

Conclusiones

La tesis presentada a lo largo de estos capítulos tuvo por objetivo principal desarrollar un sistema que permitiera la construcción de ambientes virtuales a partir de nubes de puntos capturadas con un sensor RGB-D (Kinect), montado sobre el robot de servicio Justina, para así tener una representación gráfica del mundo en el que se desenvuelve dicho robot.

A lo largo de este trabajo se expusieron diferentes metodologías para la construcción de modelos 3D, a partir de nubes de puntos. No obstante es difícil encontrar una metodología que obtenga resultados satisfactorios sin comprometer el tiempo de ejecución o sin encontrarse con una serie de adversidades que dificulten su realización. Es por ello que desde el principio del diseño del sistema desarrollado para esta tesis se quiso afrontar el problema de la construcción de ambientes virtuales en su forma más simple, de esta forma se podría entender bien la problemática, ganar entendimiento sobre el tema, dar una solución y descubrir empíricamente las dificultades que se presentan.

Con esta simplificación en mente, se optó por seguir una metodología sencilla, capaz de cumplir con el objetivo de esta tesis. El sistema desarrollado se dividió en los siguientes módulos: obtención de los datos mediante el sensor Kinect, realización del registro de los datos obtenidos con el Kinect, muestreo de los datos con dos diferentes métodos (cuantización vectorial y voxelización), obtención del cierre cóncavo de los datos muestreados y, finalmente, la visualización de los datos dentro del simulador.

En cuanto a realizar el registro de las nubes de puntos es donde quizás se tuvieron la mayor cantidad de problemas, ya que el algoritmo elegido e implementado (PCL) para realizar el registro de nubes de puntos requiere de parámetros que son muy variables dependiendo del objeto a registrar. Por ejemplo, las nubes de puntos de objetos simétricos solían empalmarse con el modelo anterior en vez de complementarse y los objetos con varias zonas planas tendían complementarse de una manera errónea. No obstante los resultados de esta implementación para los experimentos mostrados en esta tesis fueron buenos, ya que se logró tener un registro exitoso de las nubes de puntos utilizadas como entrada de este sistema, sin embargo se corría el riesgo de que no fuera así.

En lo referente a las implementaciones realizadas para el muestreo de las nubes de puntos, se desarrollaron dos métodos: cuantización vectorial y voxelización. Aunque ambos métodos tuvieron buenos resultados en las pruebas realizada cada método tiene sus ventajas y desventajas, las cuales describiremos en los siguientes párrafos.

La ventaja de utilizar el método de voxelización es que debido a que este método de muestreo tiene como principal objetivo precisamente muestrear nubes de puntos, que para nuestro caso fueron los datos de entrada, obtiene un mejor desempeño en cuanto a tiempo de ejecución se refiere. Por otra parte, la ventaja que tiene el método de cuantización vectorial es que siempre se puede establecer la cantidad de puntos que contendrá la nube de puntos muestreada.

La desventaja que tiene el utilizar voxelización es que al solo poder controlar el tamaño de voxel con el que se hará el muestreo no se puede establecer la cantidad de puntos que tendrá la nube de puntos muestreada, es decir, el tamaño dependerá de otras variables como lo pueden ser la propia distribución de los puntos en la nube original. Para el caso de cuantización vectorial su principal desventaja es que no es un método especializado en trabajar con nubes de puntos, por lo que su rendimiento en cuanto a tiempo de ejecución es muy bajo.

Algo que no se mencionó en el capítulo de resultados fue la comparación de tiempo de ejecución entre ambos métodos, siendo la razón principal para no hacerlo con anterioridad el que, al ser un sistema que no trabaja en tiempo real sino que todo el procesamiento se hace fuera de línea, esta información no era muy relevante para nuestro caso, sin embargo para futuros trabajos puede ser de gran utilidad tener un breve análisis, el cual se hará en el siguiente párrafo.

Para el siguiente análisis de rendimiento de tiempos de ejecución no debemos de olvidar la enorme cantidad de información con la que ambos métodos trabajan, teniendo esto en mente pasaremos al análisis. La implementación de voxelización en el peor de los casos experimentados se llevó 5 minutos, mientras que la de cuantización vectorial en el peor de los casos se tomaba 6 horas. La principal razón por la que los tiempos son tan diferentes es que la naturaleza del algoritmo de cuantización vectorial, al ser muy iterativo tarda mucho más en dar un resultado que el método de voxelización; aunado a esto, el incremento de tiempo también se debe a que para el caso de la voxelización nos ayudamos de la librería de PCL, mientras que para el caso de cuantización vectorial no se utilizó alguna librería que simplificara la implementación de este método.

En general, se puede decir que la hipótesis propuesta al inicio de esta tesis se validó completamente, ya que, mediante pruebas experimentales, se determinó que, utilizando el sistema propuesto, no es necesaria una gran cantidad de datos para obtener una representación virtual fidedigna del mundo real que se intenta

representar, solo basta contar con los indispensables, en base a las pruebas descritas en el capítulo anterior podemos decir que podemos reducir la cantidad de información hasta 8 veces para poder tener un resultado aceptable.

En cuanto a determinar que método de muestreo de entre los dos utilizados en esta tesis es mejor, se puede decir que no hay uno mejor que otro para nuestros fines; sin embargo, con el método de cuantización vectorial se obtuvieron menor cantidad de puntos, los cuales en pasos posteriores se convirtieron en un polígono más simple comparado al del método de voxelización, no obstante como mencionamos en párrafos anteriores el método de voxelización está diseñado para muestrear nubes de puntos; por lo que para trabajos a futuro sería recomendable seguir utilizando voxelización como método de muestreo, no obstante hay que mencionar que el método de cuantización vectorial no ha sido muy utilizado en este tipo de aplicaciones, por lo que con algunas adecuaciones para muestrear nubes de puntos podría significar una nueva alternativa para resolver este tipo de problemas de muestreo.

La conclusión final y personal, es que este trabajo aporta no solo entendimiento del problema de la construcción y simulación en ambientes virtuales, sino también, una primera aproximación de un sistema funcional que construya ambientes virtuales a partir de nubes de puntos con la finalidad de que en un futuro estos ambientes sean generados de manera más automática. La principal utilidad para los estudiantes del laboratorio de Bio-Robótica del uso de este sistema será que podrán generar ambientes virtuales 3D a partir de nubes de puntos y no a partir de mediciones manuales como se hacía con anterioridad.

6.1. Trabajo a futuro

En esta sección se pretende dar ideas que permitan continuar con el desarrollo de este trabajo, ideas que, por razones de tiempo, no pudieron ser probadas y estudiadas exhaustivamente.

En este trabajo, se concluyó que las mediciones del sensor Kinect suelen ser ruidosas, por lo que una posible mejora podría ser utilizar la segunda versión de este sensor, el cual, para obtener información de los objetos utiliza un sensor de “Tiempo de vuelo”. Dicho sensor proporciona considerablemente menos ruido que su predecesor y la resolución de este sensor es mayor. Dada la naturaleza de ciertos algoritmos utilizados en esta tesis, es de suponer que estos tengan un mejor desempeño utilizando este sensor.

Otra mejora que podría aplicarse a este sistema es implementar las soluciones mediante el uso de programación en GPU, ya que al realizar el procesamiento de los datos en paralelo se podrían reducir los tiempos de ejecución de estas implementaciones. Y poder realizar la ejecución en tiempo real. Como mencionamos en la sección de conclusiones, el sistema propuesto no funciona en tiempo real, debido a los altos tiempos que lleva el procesamiento de los datos.

También podría ser útil tratar de adecuar el presente sistema para ser utilizado como un sistema de RGB-D SLAM (Simoultaneous Localization and Mapping), ya que hasta el momento el proceso de la construcción de ambientes virtuales realizado en esta tesis es manual; se mueve al robot manualmente a posiciones previamente establecidas. La idea sería que el robot navegara autónomamente sobre el ambiente que se desea virtualizar, realizara capturas de dicho ambiente con el sensor Kinect y, conforme el robot se desplace dentro del ambiente real, fuera creando el ambiente virtual y localizándose al mismo tiempo. Esta idea suena un poco más compleja y tendría que primero tratar de implementarse el uso del cómputo en paralelo, para lograr una ejecución en tiempo real y así sea posible utilizar la filosofía de SLAM.

Si las sugerencias escritas anteriormente no son suficientes, existen otros campos para trabajar con sensores RGB-D, lo cual indica que hay mucho trabajo por hacer en relación a los sensores RGB-D.

Apéndice A
Código fuente

A.1. Muestreo utilizando Voxelización

```
int main (int argc, char** argv)
{
	//Create the pointcloud objects
	pcl::PCLPointCloud2::Ptr cloud (new pcl::PCLPointCloud2 ());
	pcl::PCLPointCloud2::Ptr cloud_sampled (new pcl::PCLPointCloud2 ());

	// Fill in the cloud data
	pcl::PCDReader reader;
	reader.read ("sample_pointcloud.pcd", *cloud);

	// Create the sampling object
	pcl::VoxelGrid<pcl::PCLPointCloud2> voxel_sampling;
	voxel_sampling.setInputCloud (cloud);
	//Set the size of the voxel
	voxel_sampling.setLeafSize (0.57f, 0.57f, 0.57f);
	voxel_sampling.filter (*cloud_sampled);

	//Save the pointcloud downsampled
	pcl::PCDWriter writer;
	writer.write ("downsampled.pcd", *cloud_sampled,
	Eigen::Vector4f::Zero(), Eigen::Quaternionf::Identity(), false);

	return (0);
}
```

A.2.1. Muestreo utilizando Cuantización Vectorial (parte 1 de 5)

```
%%% Create vectors
c_x = [];
c_y = [];
c_z = [];
centpert = [];
nc=64;

%%% read the PCD file and fill vectors
readF = loadpcd('pointcloud.pcd');
x = readF(1,:);
y = readF(2,:);
z = readF(3,:);


size_x=size(x);

points=(size_x(2));

c_x(1) = 0;
c_y(1) = 0;
c_z(1) = 0;


for n=1:points
    centpert(n) = 0;
    c_x(1) = c_x(1) + x(n);
    c_y(1) = c_y(1) + y(n);
    c_z(1) = c_z(1) + z(n);
end

%%% get the first centroid
c_x(1) = c_x(1)/points;
c_y(1) = c_y(1)/points;
c_z(1) = c_z(1)/points;
```

A.2.2. Muestreo utilizando Cuantización Vectorial (parte 2 de 5)

```
ep = 0.005;
r=2;
%%% iterate foreach new region
while r < nc + 1
    %%% compute the new centroid

    if mod(r/2,2) == 1
        for i=1 : r/2
            c_x_aux(i*2 -1) = c_x(i) + ep;
            c_y_aux(i*2 -1) = c_y(i);
            c_z_aux(i*2 -1) = c_z(i) + ep;
            c_x_aux(i*2) = c_x(i) - ep;
            c_y_aux(i*2) = c_y(i);
            c_z_aux(i*2) = c_z(i) - ep;
        end
    end

    if mod(r/2,2) == 0
        for i=1 : r/2
            c_x_aux(i*2 -1) = c_x(i);
            c_y_aux(i*2 -1) = c_y(i) + ep;
            c_z_aux(i*2 -1) = c_z(i);
            c_x_aux(i*2) = c_x(i);
            c_y_aux(i*2) = c_y(i)- ep;
            c_z_aux(i*2) = c_z(i);
        end
    end

    for i = 1 : r
        c_x(i) = c_x_aux(i);
        c_y(i) = c_y_aux(i);
        c_z(i) = c_z_aux(i);
    end

estab = 100;
d_aux_ant = 10000000000;

estab2=0;

%%% loop for stabilization
while estab > 0.3 && estab2<1000

estab = 0;
```

A.2.3. Muestreo utilizando Cuantización Vectorial (parte 3 de 5)

```
%%% compute the nearest centroid foreach point
for i = 1 : points
    for j = 1 : r
        d_aux_act = sqrt(((c_x(j)-x(i))^2) + ((c_y(j)-y(i))^2) + ((c_z(j)-
z(i))^2));
        if d_aux_act < d_aux_ant
            centpert(i) = j;
            d_aux_ant = d_aux_act;
        end

    end
    d_aux_ant = 10000000000;
end

%%initialize aux_vars
for i=1 : r
    c_x_aux(i) = 0;
    c_y_aux(i) = 0;
    c_z_aux(i) = 0;
    cont(i) = 0;
end

%% get the total elements per centroid
for i=1:points
            c_x_aux(centpert(i)) = c_x_aux(centpert(i)) + x(i);
            c_y_aux(centpert(i)) = c_y_aux(centpert(i)) + y(i);
            c_z_aux(centpert(i)) = c_z_aux(centpert(i)) + z(i);
            cont(centpert(i)) = cont(centpert(i)) + 1;
end

%% set the new centroids using the average
for i = 1 : r
    c_x_aux(i)=c_x_aux(i)/cont(i);
    c_y_aux(i)=c_y_aux(i)/cont(i);
    c_z_aux(i)=c_z_aux(i)/cont(i);
end

for i=1 : r
    estab = estab + sqrt((c_x(i)-c_x_aux(i))^2 + (c_y(i)-c_y_aux(i))^2 +
(c_z(i)-c_z_aux(i))^2);
end

estab = estab/r;
```

A.2.4. Muestreo utilizando Cuantización Vectorial (parte 4 de 5)

```
%%% compute the nearest centroid foreach point
for i = 1 : points
    for j = 1 : r
        d_aux_act = sqrt(((c_x(j)-x(i))^2) + ((c_y(j)-y(i))^2) + ((c_z(j)-
z(i))^2));
        if d_aux_act < d_aux_ant
            centpert(i) = j;
            d_aux_ant = d_aux_act;
        end

    end
    d_aux_ant = 10000000000;
end

%%initialize aux_vars
for i=1 : r
    c_x_aux(i) = 0;
    c_y_aux(i) = 0;
    c_z_aux(i) = 0;
    cont(i) = 0;
end

%% get the total elements per centroid
for i=1:points
            c_x_aux(centpert(i)) = c_x_aux(centpert(i)) + x(i);
            c_y_aux(centpert(i)) = c_y_aux(centpert(i)) + y(i);
            c_z_aux(centpert(i)) = c_z_aux(centpert(i)) + z(i);
            cont(centpert(i)) = cont(centpert(i)) + 1;
end

%% set the new centroids using the average
for i = 1 : r
    c_x_aux(i)=c_x_aux(i)/cont(i);
    c_y_aux(i)=c_y_aux(i)/cont(i);
    c_z_aux(i)=c_z_aux(i)/cont(i);
end

for i=1 : r
    estab = estab + sqrt((c_x(i)-c_x_aux(i))^2 + (c_y(i)-c_y_aux(i))^2 +
(c_z(i)-c_z_aux(i))^2);
end

estab = estab/r;
```

A.2.5. Muestreo utilizando Cuantización Vectorial (parte 5 de 5)

```
%%% set the centroids
for i=1 : r
    c_x(i) = c_x_aux(i);
    c_y(i) = c_y_aux(i);
    c_z(i) = c_z_aux(i);
end
estab2=estab2+1;

end %%end loop for estabilization
r=r*2;
end %%end the iteration foreach region

c_x=c_x';
c_y=c_y';
c_z=c_z';
%%% save the downsampled file
matrix=[c_x,c_y,c_z];
csvwrite('downsampling.pcd',matrix);
```

A.3. Proyección de una nube de puntos en el plano Z=0 y obtención del cierre cóncavo de dicha nube de puntos.

```
int main (int argc, char** argv)
{
  //Create the pointcloud objects
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_projected (new
pcl::PointCloud<pcl::PointXYZ>);

  // Fill in the cloud data
  pcl::PCDReader reader;
  reader.read ("sample_pointcloud.pcd", *cloud);

  // Create a set of planar coefficients with X=Y=0, Z=1
  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients ());
  coefficients->values.resize (4);
  coefficients->values[0] = coefficients->values[1] = 0;
  coefficients->values[2] = 1.0;
  coefficients->values[3] = 0;

  // Create the planar object
  pcl::ProjectInliers<pcl::PointXYZ> proj;
  proj.setModelType (pcl::SACMODEL_PLANE);
  proj.setInputCloud (cloud);
  proj.setModelCoefficients (coefficients);
  proj.filter (*cloud_projected);


  // Create a Concave Hull representation of the projected inliers
  pcl::PointCloud<pcl::PointXYZ>::Ptr concave_hull (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::ConcaveHull<pcl::PointXYZ> concave;
  concave.setInputCloud (cloud_projected);
  concave.setAlpha (0.1);
  concave.reconstruct (*concave_hull);

  //Save the concave hull
  pcl::PCDWriter writerConcave;
  writerConcave.write ("pointcloud_concavehull2d.pcd", *concave_hull,
false);

  return (0);
}
```

Bibliografía

[1] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. "Kinectfusion: Real-time dense surface mapping and tracking". In *IEEE ISMAR*. IEEE, October 2011.

[2] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. "Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera". In *ACM Symposium on User Interface Software and Technology*, October 2011.

[3] P. Henry, M. Krainin, E. Herbst, X. Ren and D. Fox. "RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments". In *International Journal of Robotics Research*, Vol 31, 2012.

[4] H. Hoppe, T. De Rose, T. Duchamp, J. McDonald and W. Stuetzle. "Surface reconstruction from unorganized points". In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '92), pages 71-78, 1992.

[5] P. Alliez, D. Cohen, Y. Tong and M. Desbrun. "Voronoi-based variational reconstruction of unoriented point sets". In *Proceedings of the fifth Eurographics symposium on Geometry processing,* pages 39-48, July 2007.

[6] Yang Chen and Gérard Medioni. "Object modeling by registration of multiple range images". In *Image and Vision Computing*, pages 145–155, 1992.

[7] Felix Endres, Jürgen Hess, Nikolas Engelhard et al. "An evaluation of the RGB-D SLAM system". In *IEEE International Conference on Robotics and Automation* (ICRA), May 2012.

[8] Microsoft. *Kinect for windows sensor components and specifications.* [en linea]: https://msdn.microsoft.com/en-us/library/jj131033.aspx. [Accedido: 2016].

[9] K. Khoshelham. "Accuracy Analysis of Kinect Depth Data". In *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume XXXVIII-5/W12, 2011 ISPRS*, Calgary, Canada, August 2011.

[10] Martin A. Fischler, Robert C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In *Communications of the ACM,* Vol. 24, pages 381-395, June 1981.

[11] Francois Pomerlau, Francis Colas, Roland Siegwart and Stéphane Magnenat. "Camparing ICP Variants on Real-World Data Sets". In *Autonomous Robots,* pages 133-148, April 2013.

[12] David G. Lowe, "Object recognition from local scale-invariant features,". In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, Vol.2, pages 1150-1157, 1999.

[13] G. Grisetti, S. Grzonka, C. Stachniss, P. Pfaff, and W. Burgard. "Estimation of accurate máximum likelihood maps in 3D". In *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (IROS), 2007.

[14] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. "A tree parameterization for efficiently computing maximum likelihood maps using gradient descent". In *Proc. of Robotics: Science and Systems* (RSS), 2007.

[15] K. Konolige. "Sparse sparse bundle adjustment". In *Proc. of the British Machine Vision Conference* (BMVC), 2010.

[16] M. Lourakis and A. Argyros. "SBA: A software package for generic sparse bundle adjustment". In *ACM Transactions on Mathematical Software* (TOMS), pages 1–30, 2009.

[17] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. "Bundle adjustment - a modern synthesis". In *Vision algorithms: theory and practice*, pages 153–177, 2000.

[18] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. "Surfels: Surface elements as rendering Primitives". In *ACM Transactions on Graphics* (Proc. of SIGGRAPH), 2000.

[19] O'Rourque. "Computational Geometric in C". *Cambridge University*, 1994.

[20] M. Sramek, A. Kaufman. "vxt: a C++ Class Library for Object Voxelization". In *Proc. International Workshop Volume Graphics,* 1999.

[21] M. Jones. "The Production of Volume Data from Triangular Meshes Using Voxelization". In *Computer Graphics Forum*, pages 157-167, 1996.

[22] D. Haumont. "Complete Polygonal Scene Voxelization". In *Jouranl of Graphics Tools*, pages 27-41, 2002.

[23] S. Fang and H. Cheng. "Hardware Accelerated Voxelization". In *Computer & Graphics* , pages 433-442, 2000.

[24] E. Karabassi et al. "A fast Depth-Buffer-Based Voxelization Algorith". In *ACM Journal of Graphics Tools*, pages 114-124, 2002.

[25] Y. Linde, A. Buzo; R. Gray. "An Algorithm for Vector Quantizer Design". In *IEEE Transactions on Communications*. Vol. 28, pages 84-95, January 1980.

[26] E. Rosten; T. Drummond. "Machine learning for high-speed corner detection". In *European Conference on Computer Vision*, Vol 1: pages 430–443, 2006.

[27] Steder et al. “NARF: 3D range image features for object recognition.” In *Intelligent Robots and Systems*, Taipei, Taiwan, 2010.

[28] PCL. *PCL.*[en linea]: http://pointclouds.org/documentation/ [Accedido: 2016].

[29] MATLAB. *MathWorks*. [en linea]: http://es.mathworks.com/help/index.html [Accedido: 2016].

[30] CGAL. *The Computacional Geoemtry Algorithms Library*. [en linea]: http://www.cgal.org/ [Accedido: 2016].