Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Detección e identificación de movimientos de un robot empleando nubes de puntos 3D

Autor: Adrián Campos Díez

Tutor: Manuel Ocaña Miguel

2020

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Detección e identificación de movimientos de un robot empleando nubes de puntos 3D

Autor: Adrián Campos Díez

Director: Manuel Ocaña Miguel

Tribunal:

Presidente: Ignacio Fernández Lorenzo

Vocal 1: Pedro Alfonso Revenga de Toro

Vocal 2: Manuel Ocaña Miguel

Calificación: .....

Fecha: .....

Agradecimientos

Quiero agradecer a todas las personas que me han acompañado y apoyado durante este largo camino y por las cuales no habría llegado tan lejos. A mi madre, a mi familia, a mis amigos. Gracias.

"Derrotados son los que dejan de luchar, y dejar de luchar es dejar de soñar"
José Mujica.

Resumen

El objetivo de este trabajo de fin de grado es la identificación de un robot y su entorno a partir de una nube de puntos. Para ello se emplean algoritmos disponibles en la librería Point Cloud Library(PCL, Librería de nube de puntos). La nube de puntos se proporciona por una cámara IntelRealSense ZR300. Se trabajará con imágenes en entornos interiores donde la interacción del robot con el entorno es más compleja. Se trabajará utilizando el entorno ROS y Matlab. **Palabras Clave:** ROS, Matlab, PCL.

Abstract

The objetive of this final degree proyect is to identify a robot and its environment from a point cloud. For this, algorithms available in the Point Cloud Library (PCL) are used. Point cloud is provided by an IntelRealSense ZR300 camera. We will work with images in indoor environments where the interaction between the robot and the environment is more complex. We will work using the ROS and Matlab environment. **Keywords:** ROS, Matlab, PCL.

Índice general

Índice de figuras

Índice de Extractos de Código

Palabras Clave

ROS	Robot Operating System (Sistema Operativo Robótico)
PCL	Point Cloud Library (Librería de Nube de Puntos)
SEGMENTATION	Segmentación
FILTERING	Filtrado

Capítulo 1

Introducción

Día a día la robótica va tomando un papel más relevante en la sociedad, dejando de ser un campo meramente para el estudio o para el ámbito militar, se va adaptando a las necesidades del día a día.

Por ello es muy importante la dedicación de nuestros recursos en este campo para crear nuevos puntos de vista en la creación de sistemas robóticos, ya sea para mejorar dichos sistemas, darle mayor funcionalidad, abaratar los costes para hacerlos más accesibles, etc.

Presentamos un objetivo muy concreto: la identificación de un robot y su entorno a partir de una nube de puntos y utilizando algoritmos de la librería Point Cloud Library. Trabajando con imágenes de entornos interiores proporcionados por una cámara Intel-RealSense ZR300.

1.1. Motivación

La motivación para la realización de este TFG se puede resumir en los siguientes puntos:

- Entender y profundizar en el mundo de la visión artificial como complemento de la robótica. Para ello será necesario estudiar los diferentes algoritmos empleados

para clasificar el entorno mediante las nubes de puntos, determinando cual será el más apropiado para trabajar en un entorno dinámico.

- Se trata de un campo de rápida expansión y que se puede aplicar en diferentes campos como la medicina, la conduccón autónoma, exploración del universo, etc.

1.2. Objetivos

Los objetivos a desarrollar en este TFG serán los siguientes:

- Extracción de la información no relevante del entorno del robot, como las paredes.
- Segmentación de la figura del robot llevando a cabo un estudio con diferentes algoritmos.
- Segmentación de las partes fundamentales de la figura del robot como los brazos.
- Identificación de los movimientos del robot.

1.3. Estructura del proyecto

La estructura que se va a llevar a cabo en este trabajo se explica a continuación:

- **Introducción:** Este capítulo presenta una visión general del mundo de la visión artificial, así como las inquietudes, motivos y objetivos por los que se ha realizado este proyecto.
- **Estado del arte:** Se buscarán y se analizarán en detalle los distintos métodos y algoritmos para detectar objetos móviles, figuras... (Se realizará un estudio de "papers" o artículos científicos sobre estos conceptos) y posteriormente seleccionar el método que mejor se adapte a este TFG.
- **Instalación del Sistema Operativo (Ubuntu) junto con el entorno ROS:** Se instalará el sistema Ubuntu con el entorno de desarrollo ROS, en el cual podremos programar los algoritmos para simular y verificar el éxito en la instalación. Se harán las pruebas y se obtendrán resultados de ellas para poder finalizar con las conclusiones y trabajos futuros del mismo.

- **Instalación de la librería PCL:** Se instalará la librería PCL en el entorno ROS para llevar a cabo las pruebas con el entorno real.
- **Implementación y resultados:** Se llevarán a cabo pruebas con diferentes algoritmos de la PCL y se realizará una comparación de su funcionamiento y de los resultados obtenidos por cada uno de ellos.
- **Conclusiones:** Se concluye cuáles de los algoritmos probados resuelven el problema de la manera más eficiente.

Capítulo 2

Estado del arte

2.1. Historia de la visión artificial

En este apartado se explicará cómo ha evolucionado la visión artificial llegando al estado actual, tratando los conceptos necesarios para el mejor rendimiento del proyecto.

La visión artificial o también llamada visión por computador, es una disciplina científica que incluye métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de reproducir las características visuales de objetos o espacios a través de la interpretación de los datos que puede realizar un software destinado a esta labor.

La evolución de la visión artificial ha ido estrechamente relacionada con el desarrollo de las cámaras fotográficas y la obtención de imágenes desde la perspectiva científica, como las radiografías.

Las nubes de puntos son un caso particular de la visión artificial, en la que las imágenes generadas no son planas sino que se añade información de profundidad, cosa que antes se conseguía con algoritmos de geometría epipolar que fusionaban información de dos cámaras.

Figura 2.1: Radiografía médica

Las primeras investigaciones en visión por computador tuvieron lugar en los años 50 al usar algunas de las primeras redes neuronales para descubrir los bordes de un objeto y clasificar objetos simples en categorías como círculos y cuadrados.

Como describió Lawrence Roberts en su tesis "Percepción mecánica de los sólidos tridimensionales", publicada en 1963 [7] y ampliamente considerada como uno de los precursores de la visión por computadora moderna.

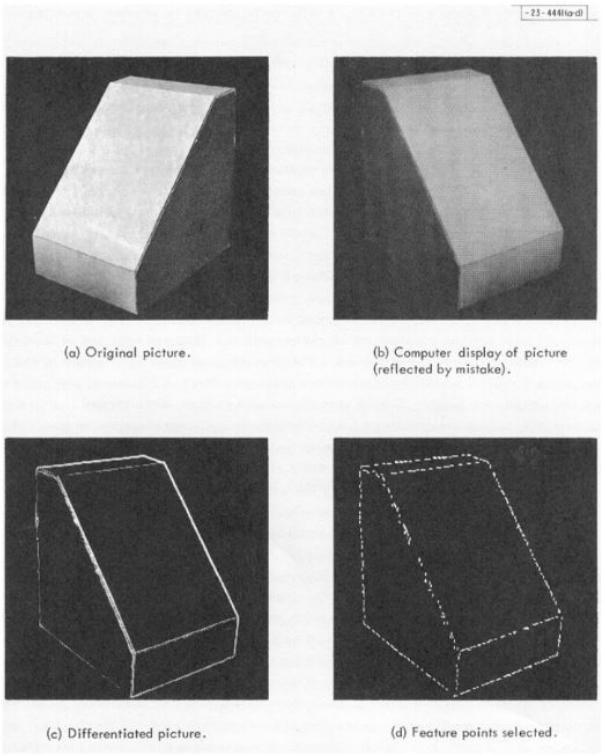

FIGURA 2.2: Reducción del mundo visual a imágenes geométricas

Debemos remontarnos a los años 60 para encontrarnos con los primeros modelos de automatización entre la adquisición de imágenes por cámaras de visión y el procesamiento de dichas imágenes con el objetivo de conocer su estructura interna a niveles en los que no se podría observar sólo con las imágenes originales.

En los años 70 se marcó el punto de partida de la visión por computador para fines comerciales, interpretando texto escrito a mano o mecanografiado utilizando el reconocimiento óptico de caracteres. Este avance se implementó para evaluar el texto escrito para ciegos.

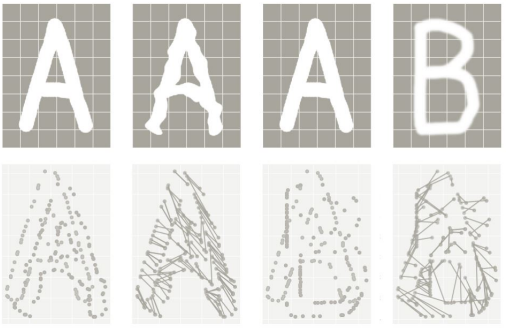

FIGURA 2.3: Técnicas de reconocimiento óptico de caracteres (OCR)

La visión artificial empieza a ser relevante en la década de los 80, debido a que la expansión de la ingeniería informática da lugar a la creación de microprocesadores más avanzados que permiten captar, procesar y reproducir imágenes tomadas por una cámara a la que podían estar conectada de forma remota.

En la década de los 90, el rápido crecimiento de internet permitió poner en disposición grandes conjuntos de imágenes para el análisis, lo cual ayudó a crear máquinas capaces de reconocer a personas específicas en fotos y vídeos, lo que se conoce como reconocimiento facial.

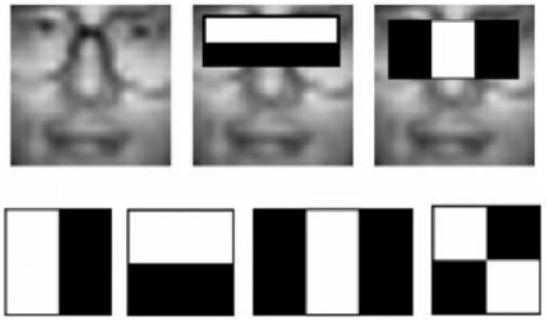

FIGURA 2.4: Reconocimiento facial

Las nubes de puntos, históricamente se han creado mediante escáneres de detección remota activos, como radares y lásers utilizadas en plataformas aéreas y terrestres en el ámbito militar.

Más adelante, cuando la complejidad de las imágenes que se querían procesar aumentaba y no eran suficientes los mecanismos implementados hasta el momento se empezaron a utilizar las nubes de puntos.

Como podemos observar en el trabajo de Marc Levoy y Turned Whitted en "El uso de puntos como primitiva de visualización de 1985", [5] se proponen soluciones ante el aumento de la comlejidad visual de las escenas generadas por computadora, en este caso, el uso de primitivas de modelado clásicas se vuelve menos atractivo. La customización de los algoritmos de visualización, el conflicto entre el orden de los objetos y la representación del orden en la imagen así como la poca utilidad de la coherencia para la identificación de objetos en entornos muy comlplejos son los factores que desencadenan la creación de un método de representación más eficiente. Se propone para ello el

desacople del modelado por geometría con el proceso de representación, introduciendo la noción de puntos como primitiva universal.

Se demuestra que con una matriz discreta de puntos desplazados arbitrariamente en el espacio se puede representar una superficie tridimensional continua. esto resolvía el antiguo problema de procucir bordes de silueta correctos para texturas con mapas de relieve

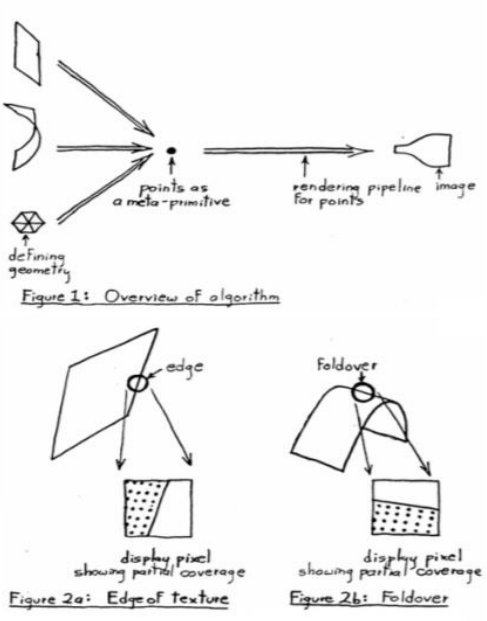

FIGURA 2.5: Uso de puntos como primitiva de visualización

A consecuencia de esto, una amplia clase de objetos geométricamente definidos, tanto planos como curvos, pueden ser representados por puntos.

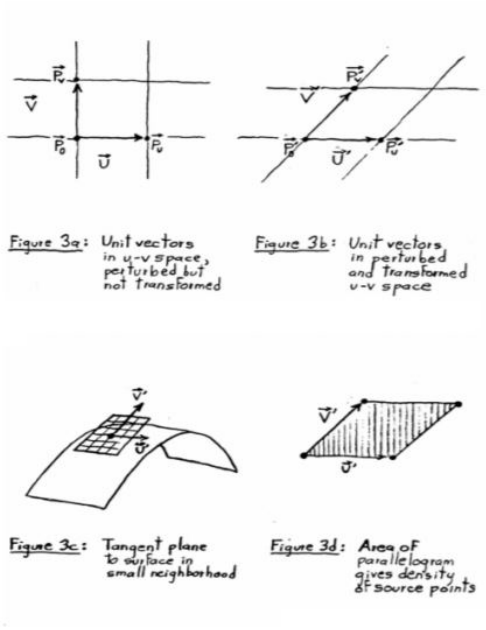

FIGURA 2.6: Uso de puntos como primitiva de visualización

Este algoritmo de representación es simple y no requiere de coherencia para ser eficiente. Además, el tratamiento de los puntos puede llevarse a cabo de manera aleatoria.

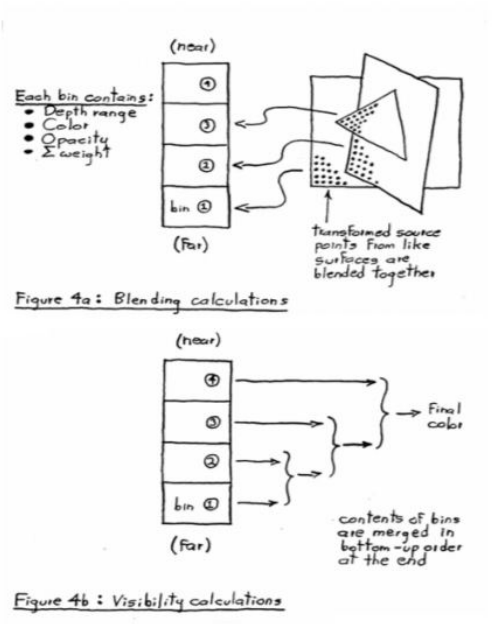

FIGURA 2.7: Uso de puntos como primitiva de visualización

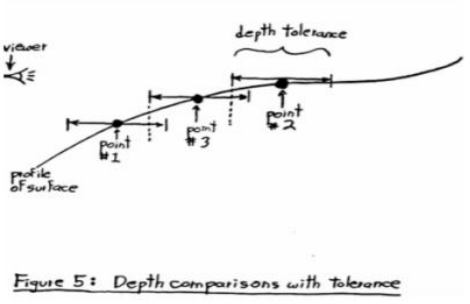

FIGURA 2.8: Uso de puntos como primitiva de visualización

Actualmente las aplicaciones con nubes de puntos están aumentando potencialmente, sin embargo esta expansión enfrenta limitaciones técnicas, principalmente por la falta de información semántica dentro de los conjuntos de puntos y por la gran cantidad de datos a procesar. La extracción de conocimiento dentro de la nube de puntos sigue siendo un proceso manual y lento que sufre de interpretaciones humanas propensas a cometer errores. Esto destaca una necesidad de crear sistemas de análisis de datos para crear una información coherente y estructurada. Se encuentran trabajos como "PointNet"[9] que aporta eficiencia y efectividad para solucionar los problemas mencionados anteriormente combinando las nubes de puntos con inteligencia artificial. El funcionamiento de dicho trabajo se basa en una red neuronal que consume directamente nubes de puntos.

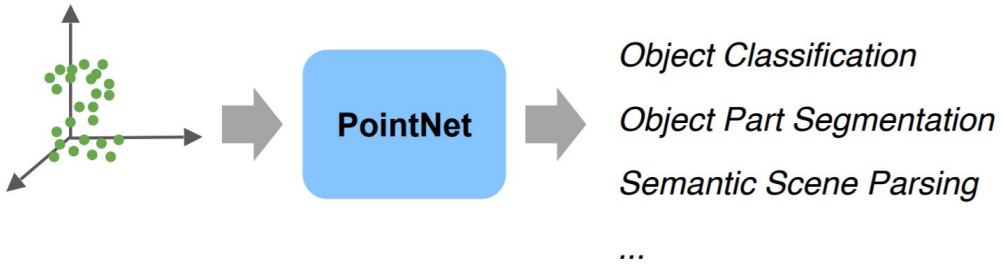

FIGURA 2.9: Entrada y aplicaciones de PointNet

A partir de una nube de puntos desordenada se pueden llevar a cabo tareas como clasificación, segmentación de objetos y segmentación semántica entre otras.

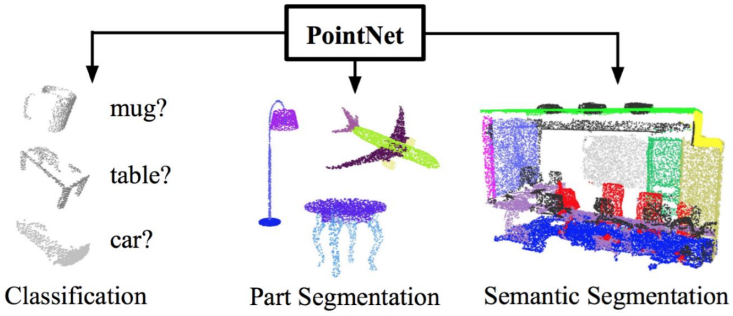

FIGURA 2.10: Entrada y aplicaciones de PointNet

Existen dos requerimientos para el correcto funcionamiento de dicha aplicación:

- El modelo debe ser invariante ante N! permutaciones. Para ello, los puntos de entrada desordenados se representan en un vector de dimension D. De esta forma

aunque los puntos se traten en diferente orden, si pertenecen al mismo conjunto se reconozca como tal.

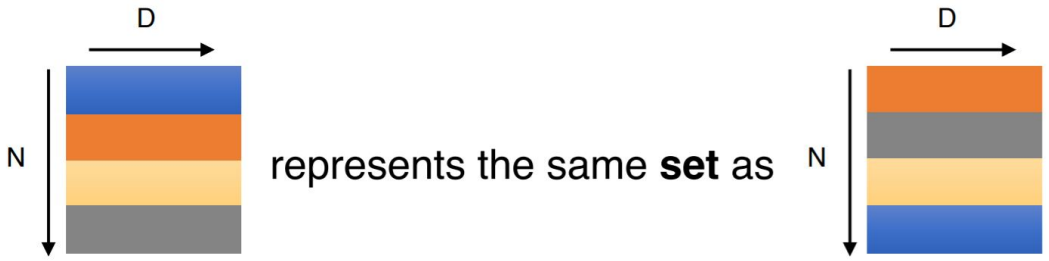

FIGURA 2.11: Vector de puntos de entrada

Este vector de puntos a su vez será tratado con una función simétrica del tipo:

$f(x_1, x_2, ..., x_n) \equiv f(x_\pi, x_{\pi 2}, ..., x_{\pi 3}), x_i \in \mathbb{R}^D$

Es posible construir una familia de funciones simétricas para redes neuronales

$f(x_1, x_2, ..., x_n) = \gamma \circ g(h(x_1), ..., h(x_n))$

La función anterior será simétrica si g es simétrica. Por lo tanto se puede construir una red de funciones simétricas cuyo funcionamiento empírico se basa en Perceptrón Multicapa (MLP) y Max Pooling. El Perceptrón multicapa es una red neuronal artificial formada por múltiples capas, de tal manera que tiene capacidad para resolver problemas que no son linealmente separables. Max pooling es un tipo de operación que normalmente se agrega a las redes neuronales convolucionales siguiendo capas convolucionales individuales. Cuando se agrega a un modelo, Max Pooling reduce la dimensión de las imágenes al reducir el número de pixeles en la salida de la capa convolucional anterior.

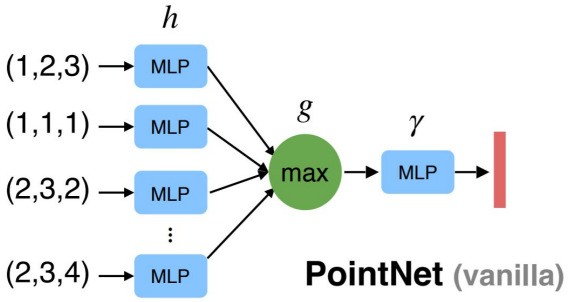

FIGURA 2.12: Obtención de funciones simétricas

- La rotación de la nube de puntos no debe afectar a los resultados de la clasificación. Para ello los puntos de entrada se procesan a través de una trasformada.

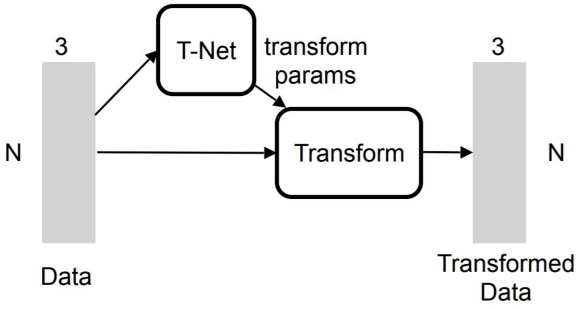

FIGURA 2.13: Transformada

La cual se trata solamente de una matriz multiplicadora.

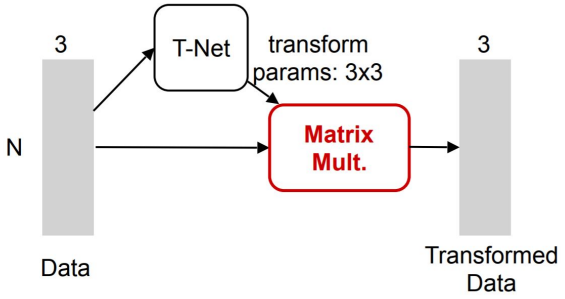

FIGURA 2.14: Transformada

2.2. Aplicaciones de la visión artificial

En este apartado explicaremos algunas de las aplicaciones más interesantes que tiene la visión artificial en una amplia gama de campos relacionados con el reconocimiento de objetos y representación de escenas. [3]

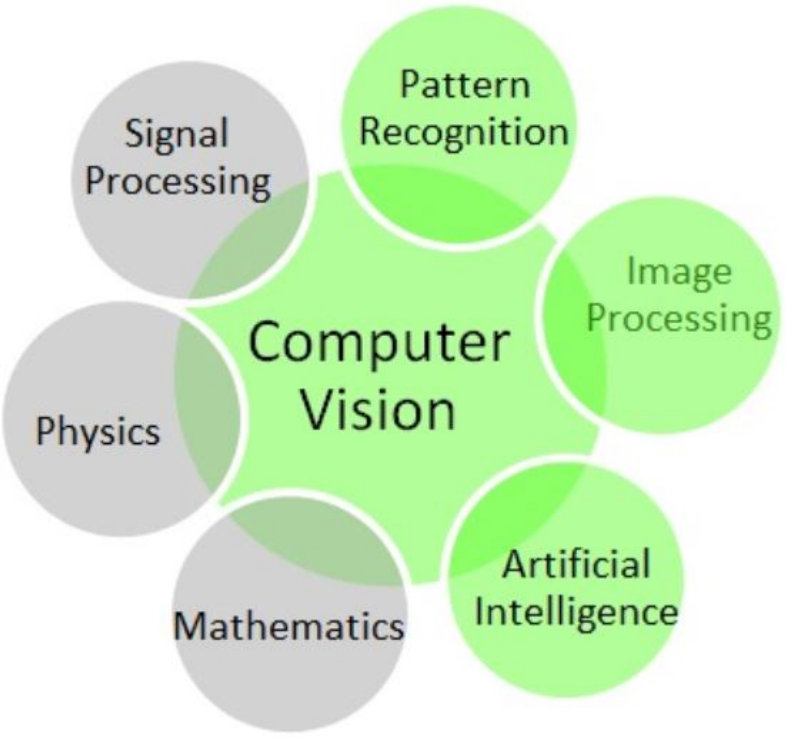

FIGURA 2.15: Aplicaciones de la Visión Artificial

- **Detección y reconocimiento de objetos:**

  Para este tipo de aplicaciones es necesaria la existencia de un modelo previo para cada uno de los objetos que se quieran detectar en la imagen. A partir de estos modelos, se deben buscar los métodos que sean capaces de generar la información que permitirá diferenciar unos puntos de otros en función de las características que presenten dichos puntos. Una vez asociados, se deberá ser capaz de calcular la correspondencia entre el modelo y la imagen a analizar. A partir de este emparejamiento de puntos el sistema deberá ser capaz de obtener la posición y orientación del objeto. [11]

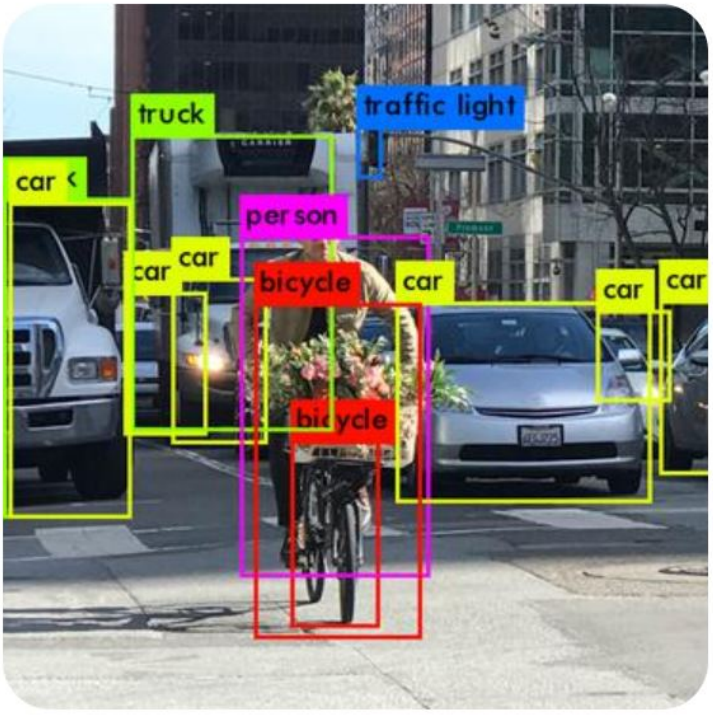

FIGURA 2.16: Detección de objetos

- **Reconocimiento facial:**

  Se trata de la aplicación de identificación de personas en imágenes digitales o fotogramas específicos de vídeo, en base a sus características faciales y comparación de estas con una base de datos.

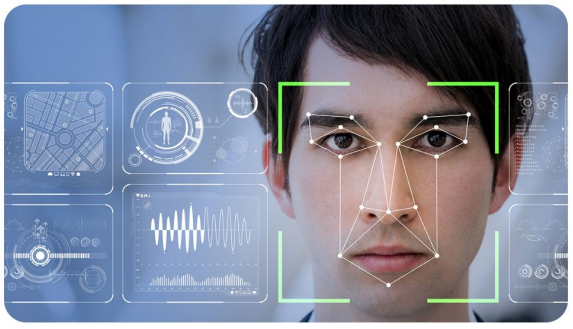

FIGURA 2.17: Reconocimiento Facial

Se utiliza en un gran número de aplicaciones de seguridad que requieren del reconocimiento del usuario como las que se muestran en la siguiente tabla.

Áreas	Aplicaciones específicas
Biometría	Licencia de Conducir, Programas de Derecho, Inmigración, DNI, Pasaportes, Registro de Votantes, Fraude
Seguridad de la información	Inicio de Sesión, Seguridad en Aplicaciones, Seguridad en Bases de Datos, Cifrado de Información, Seguridad en Internet, Acceso a Internet, Registros Médicos, Terminales de Comercio Seguro, Cajeros Automáticos
Cumplimiento de la ley y vigilancia	Videovigilancia Avanzada, Control CCTV, Control Portal, Análisis Post-event, Hurto, Seguimiento de Sospechosos, Investigación
Tarjetas inteligentes	Valor Almacenado, Autenticación de usuarios
Control de acceso	Acceso a Instalaciones, Acceso a Vehículos

FIGURA 2.18: Aplicaciones del reconocimiento facial

El proceso de reconocer una cara se basa principalmente en 4 etapas:

1. **Detección de la cara:** Detecta si hay una cara en la escena y proporciona la localización y proporción de dicha cara.
2. **Alineación de la cara:** En la cara detectada en el paso anterior, se identifican los componentes tales como ojos, boca, nariz, las distancias entre estos elementos, etc. Se normalizan dichas componentes respecto a transformaciones geométricas como el tamaño y la posición y, fotométricas como la iluminación.
3. **Extracción de características:** Se extraen las características más relevantes para poder distinguir entre caras de diferentes individuos según variaciones geométricas y fotométricas.
4. **Reconocimiento:** El vector de características extraido de los pasos anteriores es comparado con vectores de características almacenados en la base de datos. Si se encuentra una elevado porcentaje de similitud entre los pares obtentendremos una coincidencia con rostro existente en la base de datos, en caso contrario, estaremos ante una cara desconocida.

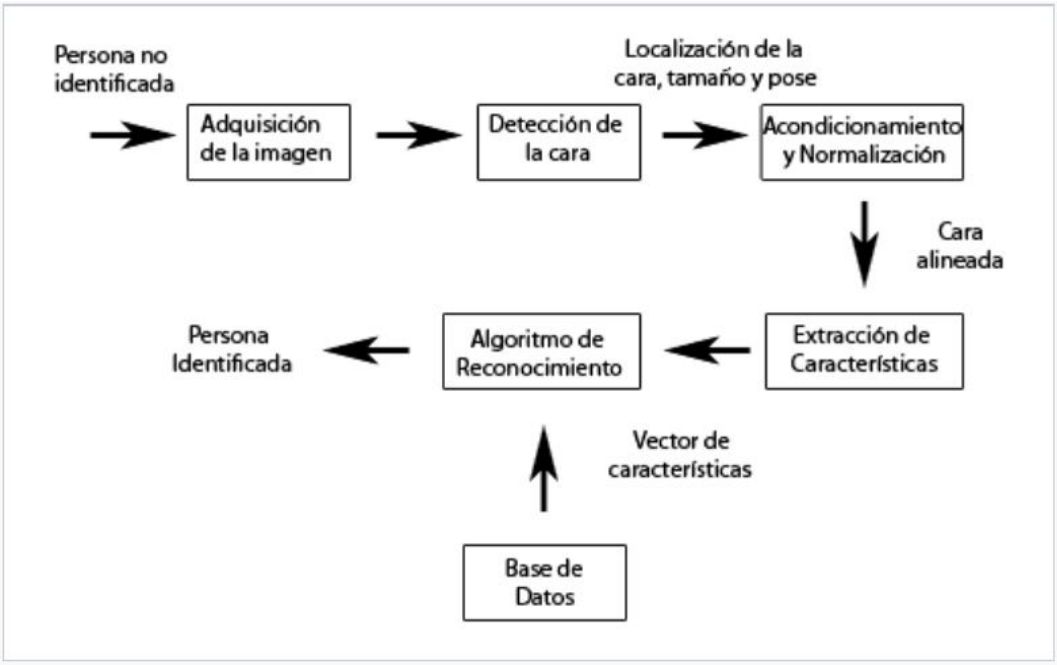

FIGURA 2.19: Proceso de reconocimiento facial

- **Visión artificial aplicada a la ganadería y agricultura:**

FIGURA 2.20: Dron analizando cultivos

La uso de la visión artificial ha facilitado la tarea de monitorear el ganado, identificar problemas de salud, como la cojera, que pueden afectar la producción de leche.

Esto permite una rápida detección de problemas en los animales, con mayor rapidez que con las técnicas humanas tradicionales y dota de mayor tiempo de reacción a la hora de aplicar tratamientos a dichos animales para que la producción no se vea afectada.

La visión artificial también ha encontrado su lugar en la agricultura y la horticultura, donde las cosechas pueden ser detectadas visualmente y clasificadas por

color, tamaño y condición, lo que conlleva un gran ahorro al no tener que utilizar personal para la realización de dicha tarea.

- **Automoción:**

  La visión artificial esta tomando un papel muy importante en el mundo de la automoción. El mejor ejemplo de ello es la marca de automóviles Tesla, la cual ocupa el trono de vehículos de conducción autónoma.

  Estos vehículos, dotados con cámaras que proveen 360º de visibilidad alrededor del coche y 250 metros de rango.

FIGURA 2.21: Rangos de las cámaras utilizadas por los vehículos autónomos de Tesla

Algunas de las aplicaciónes más relevantes de la visión artificial aplicada en la automoción son las siguientes:

- **Sistemas de detección de peatones** Los sistemas de detección de peatones permiten que el vehículo se anticipe a la propia percepción del conductor, detectando a los peatones que transitan en su entorno y previendo potenciales situaciones de peligro. Estos están basados en el funcionamiento combinado de un radar integrado en la rejilla frontal del vehículo con una cámara situada dentro del habitáculo, por detrás del espejo retrovisor. El análisis de los datos que ambos proporcionan se realizará a traves de una unidad electrónica de

control.

La misión del radar es la de detectar la distancia a la que se encuentran los objetos por delante del vehículo, la cámara, por su parte, es la encargada de establecer de qué tipo de objetos se trata, ya sea un peatón o otro coche. Además de establecer el patrón de movimiento del peatón, lo que permite calcular si la trayectoria de dicho peatón interferirá en la trayectoria que lleva el vehículo.

FIGURA 2.22: Reconocimiento de los patrones de los peatones

- **Control de crucero adaptativo** El control de crucero adaptativo es una evolución del control de crucero, el cual permite mantener una velocidad constante previamente fijada por el conductor y que se desconectará auntomáticamente cuando pisamos el freno. En esta versión más inteligente se tiene el cuenta el tráfico a la hora de mantener la velocidad con lo que el conductor no tendrá que estar acoplándose continuamente a las condiciones de la carretera. El control de crucero adaptativo cuenta con una serie de radares que se encargan de detectar el tráfico en la vía, de tal manera que si nos encontramos con un coche por delante a una velocidad inferior, automáticamente el sistema alerta al conductor del peligro y reduce la velocidad de nuestro vehículo actuando sobre el sistema de frenos, de forma que se mantiene la distancia de seguridad que haya sido predeterminada. Una vez que el carril por el que

circulamos queda libre, el sistema acelera el vehículo hasta la velocidad que hayamos programado.

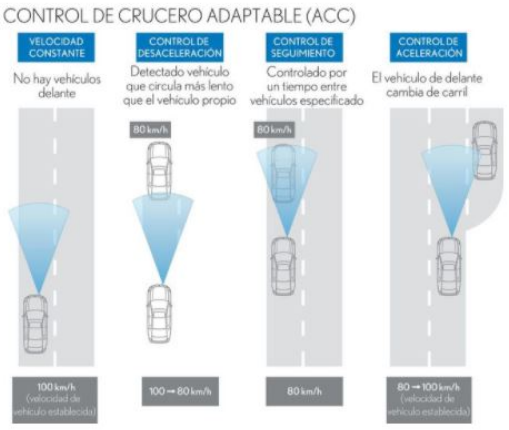

FIGURA 2.23: Control de crucero adaptativo

- **Sistemas de reconocimiento de señales:** Gracias a las cámaras con las que están equipados los vehículos es posible captar las imágenes y darles un tratamiento digital por el cual el sistema es capaz de leer las señales e interpretarlas para dar un aviso al conductor. Este tipo de tecnología recibe el nombre de Traffic Sign Recognition (TSR).

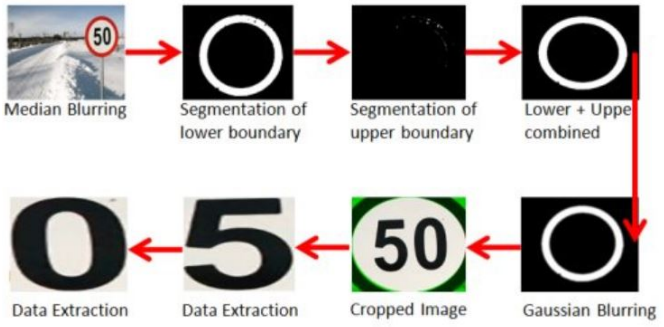

FIGURA 2.24: Diagrama de bloques del reconocimiento de señales

El funcionamiento de dicha aplicación se basa en dos fases:

- **Detección:** Es la fase en la cual la señal es detectada. Se deben llevar a cabo varias acciones para realizar este procreso de la mejor manera:
  - Desenfocado: La imagen sin procesar puede incluir ruido aleatorio o puntos negros debido a un dispositivo de captura menos eficiente. Además, debido a una iluminación inadecuada, alguna parte de esta imagen puede contener bordes nítidos o cambios de color abruptos

que pueden afectar al proceso de detección. Es por esto que necesitamos tener un efecto de desenfoque.

- ◇ Detección basada en color: El color es uno de los atributos más poderosos para la detección de objetos, el tono, la saturación y el valor juegan un papel importante en el procesamiento de imágenes. Por ejemplo es sencillo detectar una señal por sus bordes rojos.
- ◇ Detección basada en formas: Con la detección basada en formas, es posible detectar una información de forma particular, objetos circulares, cuadrados o triangulares tratámdose de señales.
- ◇ Extracción de características: Extracción de características dentro de los límites marcados por las acciones anteriores.
- ◇ Separación: Dentro de esta fase se utilizan métodos de contornos para separar unos dígitos de otros.

- ○ **Reconocimiento:** En esta fase se establece de que señal se trata. Hay multitud de técnicas para realizar esta tarea, uno de los más conocidos es OCR, que tiene la finalidad de poder diferenciar un texto de una imagen cualquiera. Para hacerlo se basa en cuatro etapas:
  - ◇ Binarización o caracterización: Proceso de convertir la imagen a blanco y negro donde quedan claramente marcados los contornos de los caracteres y símbolos que contiene la imagen.
  - ◇ Fragmentación o segmentación de la imagen: Este proceso implica la detección mediante procedimientos de etiquetado determinista o estocástico de los contornos o regiones de la imagen, basándose en la información de intensidad o información espacial.
  - ◇ Adelgazamiento de los componentes: Una vez aislados los componentes conexos de la imagen, se les tendrá que aplicar un proceso de adelgazamiento para cada uno de ellos. Este procedimiento consiste en ir borrando sucesivamente los puntos de los contornos de cada componente de forma que se conserve su tipología.
  - ◇ Comparación con patrones: En esta etapa se comparan los caracteres obtenidos anteriormente con unos teóricos (patrones) almacenados en una base de datos.

- **Conducción autónoma:** Existen diferentes niveles de conducción autónoma dependiendo del nivel de intervención del conductor, los niveles son los siguientes:

    1. Asistencia en la conducción: El vehículo cuenta con algún sistema de ayuda a la conducción y está pensado para brindar una conducción más cómo y mejorar la seguridad al volante.
    2. Automatización parcial: Se precisa conductor aunque este no realizará tareas relativas al movimiento. El vehículo cuenta con control de movimiento tanto longitudinal como lateral, aunque no tiene detección y respuesta ante objetos. El vehículo tendrá capacidad de actuar de forma independiente ya que pueden realizar una o varias tareas hasta ahora realizadas por en conductor.
    3. Automatización condicionada: Se precisa conductor y aunque la auntonomía sea más elevada, este deberá estar atento para intervenir. Tiene sistemas de automatización en lo referente al control de movimiento longitudinal y lateral; detección y respuesta ante objetos. El vehículo podrá decidir cuando cambiar de carril, frenar para evitar colisiones con otro vehículo, etc. El factor humano seguirá siendo clave ya que el sistema puede precisar de su intervención.
    4. Automatización elevada: No se precisará de la intervención humana en ningún momento ya que el coche será el propio vehículo quien controle el tráfico y las condiciones del entorno, definirá la ruta o alternativas y responderá ante cualquier situación. En el caso de existir algún fallo en el sistema principal, el vehículo contará con respaldo para actuar y seguir conduciendo.
    5. Automatización completa: El vehículo tendrá la capacidad, bajo demanda realizada a través de la interfaz por la que se introducen las órdenes, de ir a cualquier lugar sin necesidad de volante, pedales o mandos, ya que cuenta con sistemas de automatización a todos los niveles. En este nivel la figura del conductor no existe. Cuenta con un sistema de automatización que en caso de fallo se respaldará con otro sistema, por lo que él mismo solucionará cualquier imprevisto.

El nivel 2 es actualmente el más común en los concesionarios de diferentes marcas, mientras que el nivel 3 solo lo encontramos en algunas de ellas.

Los niveles 4 y 5 se encuentran en desarrollo y se prevee que a lo largo de esta década empiecen a implementarse en las carreteras.

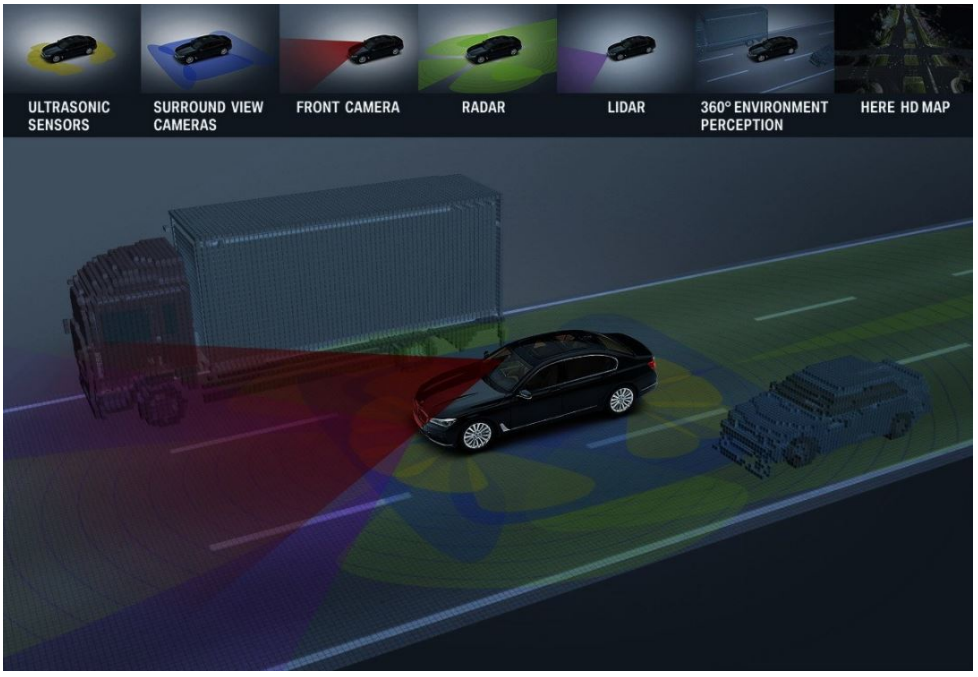

FIGURA 2.25: Sensores para la condución autónoma

2.3. Cámaras de nubes de puntos

Una nube de puntos 3D es el producto resultante del escaneo láser o fotogrametría digital. Se compone de millones de puntos posicionados tridimensionalmente en el espacio, formando con exactitud milimétrica una entidad física y representando su superficie externa. La nube de puntos 3D contiene una amplia información métrica sobre las superficies escaneadas, así como de su color y la reflectividad del material.

Las nubes de puntos se utilizan para muchos propósitos, tales como creación de modelos CAD en 3D para piezas prefabricadas, para metrología e inspección de calidad y para una multitud de aplicaciones de visualización, animación, renderización, personalización masiva creación de modelos digitales de elevación del terreno

Las cámaras de nubes de puntos generan un mapa de profundidad, con el cual se reproyecta cada pixel en el espacio 3D y se le asigna el color del mismo pixel desde el marco RGB utilizando las bibliotecas de nubes de puntos. [10]

Se utilizan numerosos algoritmos para: filtrar valores atípicos de datos ruidosos, unir nubes de puntos 3D, segmentar las partes relevantes de la escena y calcular descriptores para reconocer objetos en la escena en función de su apariencia geométrica, y crear superficies a partir de nubes de puntos.

Los dos tipos de nubes de puntos con los que se suele trabajar son:

- **Nubes de puntos clásicas:** Son nubes de puntos dotadas con la información cartesiana de los puntos y su valor de intensidad en escala de grises, calculada en función de la diferencia de intensidad de la señal laser para cada punto medido.
- **Nubes de puntos coloreadas:** Estas nubes de puntos están dotadas de un valor cromático, es decir, que para cada punto de la nube, además de la información de sus coordenadas X, Y, Z posee el valor del color en el rango RGB. Esta información es más dificil de procesar que con las nubes de puntos en escala de grises, pero replica a la realidad con exactitud y enorme realismo.

2.4. Point Cloud Library

PCL [6] es una librería de C++ desarrollada para el tratamiento de nubes de puntos en N-dimensiones desarrollado por Willow Garage con el objetivo de realizar procesamiento con numerosas técnicas. Esta librería tiene algoritmos para aplicar filtros, estimaciones de funciones, reconstruccion de superficies, ajustes de modelos, segmentación, entre otros. Además se encuentra liberada bajo licencia BSD, es decir, que es libre para uso comercial e investigativo, y sus códigos fueron desarrollados en lenguaje de programación C++. La financiación y el soporte de esta librería se presenta gracias a grandes empresas tales como Nvidia, Google, Toyota, Trimble, Urban Robotics, Honda Research Institute y Sandía Intelligent System and Robotics. Por otra parte, PCL funciona sobre varias plataformas como pueden ser Windows, Linux, MacOS y Android. Estas herramientas ofrecen el portencial necesario para procesar y recontruir una escena tridimensional de una manera sencilla.

El formato utilizado por la librería PCL, se conoce como PCD (Point Cloud Data). Este formato nace ante la necesidad de capturar datos obtenidos a partir de sensores y representarlos como una nube de puntos 3D.

PCL permite tratar las nubes de puntos de muchas formas, vamos a centrarnos en una de ellas que es la segmentación. [10]

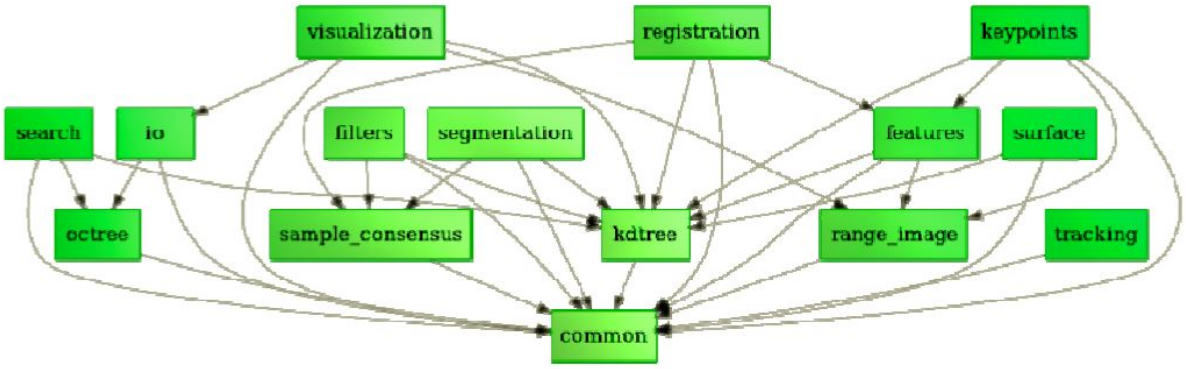

FIGURA 2.26: Organización de la biblioteca PCL

2.5. Robot Operating System (ROS)

Robot Operating System (*ROS*) [8] es una plataforma software, lo que se conoce como framework, para el desarrollo de software específico para robótica. Fue creado por el instituto de investigación Willow Garage en 2007, bajo licencia BSD. Todos sus programas son de código abierto (OSS), permitiéndose así su uso gratuito, tanto para la investigación como para fines comerciales. Su filosofía radica, en que todos los programas sean de uso libre, reutilizables, escalables según la aplicación deseada e integrables con todo el software de robótica existente y futuro.

ROS se compone de un conjunto de librerías de programación, aplicaciones, drivers y herramientas de visualización, monitorización, simulación y análisis, todas reutilizables para el desarrollo de nuevas aplicaciones para robots tanto simulados como reales.

Su estructura es modular, de forma que cada programa o aplicación, lo que en *ROS* se denomina nodo, se ejecuta dentro otro ejecutable (Master), que funciona de núcleo del

sistema y hace las veces de plataforma, por la que se comunican los diferentes nodos mediante topics y/o servicios.

En este proyecto se ha utilizado la distribución Kinetic y se ha instalado sobre Ubuntu 16.04.

2.5.1. Robot Operating System

Se ha elegido *ROS* (Robot Operating System) como entorno para el desarrollo de la solución inmersiva, que se encuentra completamente integrado y preparado para desarrollar aplicaciones compatibles con los requisitos propuestos para nuestro proyecto.

ROS es un marco flexible para escribir software para nuestro robot. Es una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas. *ROS* se creó desde cero para fomentar el desarrollo de software de robótica colaborativa. Por ejemplo, un laboratorio podría tener expertos en el mapeado de ambientes interiores, y podría contribuir con un sistema de clase mundial para producir mapas. Otro grupo podría tener expertos en el uso de mapas para navegar, y otro grupo podría haber descubierto un enfoque de visión por computadora que funciona bien para reconocer pequeños objetos en desorden. *ROS* fue diseñado específicamente para grupos de trabajo como estos para colaborar y construir sobre el trabajo de cada uno.

Uno de los principales motivos es la filosofía con la que se creo *ROS*, la de que todos los programas sean de uso libre, reutilizables, escalables según la aplicación deseada e integrables con todo el software de robótica existente y futuro. Por ello el sistema se creó teniendo en cuenta otras plataformas software abiertas ya existentes como PCL, existiendo paquetes para su uso en *ROS*. Se compone de un conjunto de librerías de programación, aplicaciones, drivers y herramientas de visualización, monitorización, simulación y análisis, todas reutilizables para el desarrollo de nuevas aplicaciones para robots tanto simulados como reales, de ello que nosotros nos basemos en utilizar la librería PCL.

2.5.1.1. Conectividad Matlab-ROS

Para implementar el funcionamiento para la representación de nuestro robot y el emtorno, utilizaremos *ROS* Distribuido. *ROS* presenta una arquitectura de programación distribuida, en la que diferentes nodos se comunican a través mensajes enviados a los topics. Además, *ROS* permite que estos nodos se distribuyan en diferentes máquinas comunicadas por una red, para ajustarse a los recursos disponibles. Sólo es necesario un Master de *ROS* (roscore), que se ejecutará en una de las máquinas.

Todas las máquinas deben conocer dónde se ejecuta el Master, lo cual se indica configurando en todas ellas (incluida la propia máquina en la que se ejecuta) la variable de entorno *ROS_MASTER_URI*, además si existe conectividad completa entre las máquinas a través de la red, los topics creados por cada una de ellas serán accesibles a todas las demás, para configurar todo esto es necesario que en el fichero .bashrc, se añada al final las sentencias del código 2.1

Listing 2.1: Código añadido en .bashrc para conseguir conectar ROS-Matlab.

```
export ROS_MASTER_URI=http://IP_ROSMASTER_MACHINE:11311
export ROS_IP=IP_LOCAL_MACHINE
```

En este proyecto utilizaremos una máquina virtual y una máquina nativa con *Matlab* que ejecutara el roscore de todo el proyecto. *Matlab* dispone de la toolbox "Robotics System Toolbox" que permite crear nodos en *Matlab/Simulink* con conectividad al sistema *ROS*. Por último, falta conectar *Matlab* con *ROS* para ello ejecutaremos en *Matlab* el siguiente código:

Listing 2.2: Código ejecutado con matlab en la maquina nativa para conseguir conectar ROS-Matlab.

```
rosinit('http://IP_ROSMASTER_MACHINE' , 'NodeHost' , 'IP_LOCAL_MACHINE');
%Siendo NodeHost maquina nativa(Matlab)
```

Una vez conectado *Matlab* con *ROS* debemos recoger la información del robot y subscribirnos a todos los topics de este para poder recoger información de ellos.

Para la visualización de la nube de puntos generada tras el proceso de segmentación debemos suscribirnos al siguiente topic:

Listing 2.3: Código ejecutado con matlab para suscribirse al topic de la nube de puntos generada.

```
sub = rossubscriber('/people_points_cloud');
```

Una vez que estamos conectados al suscriptor adecuado, esperamos a recibir información de él con una latencia de 10 segundos. Llevamos a cabo la conversión del tipo de mensaje pointCloud2 a un objeto del tipo nube de puntos para poder operar con Matlab. Cuando se trata de una nube de puntos sin color debemos extraer las coordenadas X, Y, Z. Si por el contrario se va a recibir una nube de puntos con color, deberíamos utilizar el código comentado para extraer las coordenas cartesianas además de la información RGB.

Listing 2.4: Recepción del mensaje desde ROS.

```
ptCloudRef=receive(sub,10);
ptCloudCurrent=receive(sub,10);

xyz = readXYZ(ptCloudRef);
%%rgb = readRGB(ptCloudRef);
ptCloud = pointCloud(readXYZ(ptCloudRef));

xyz = readXYZ(ptCloudCurrent);
%%rgb = readRGB(ptCloudCurrent);
%%ptCloud2 =
    pointCloud(readXYZ(ptCloudCurrent),'Color',uint8(255*readRGB(ptCloudCurrent)))
ptCloud2 = pointCloud(readXYZ(ptCloudCurrent));
```

Se debe llevar a cabo también un método de reducción de la resolución, para ello empleamos el siguiente código, dotando de una resolución a la cuadrícula 3D de (0,1 x 0,1 x 0,1). Este proceso no solo acelera el registro de los puntos, sino que también mejora la resolución.

Listing 2.5: Reducción de la resolución.

```
gridSize = 0.1;
fixed = pcdownsample(ptCloud, 'gridAverage', gridSize);
moving = pcdownsample(ptCloud2, 'gridAverage', gridSize);
```

Una vez que tenemos la resolución adecuada, llevamos a cabo una transformada para registrar una nube de puntos en movimiento en una nube de puntos fija. Para ello tomamos la primera nube de puntos como referencia y aplicamos la transformada sobre la segunda nube de puntos. Se fusiona la nube de puntos de la escena con la nube de puntos transformada para procesar los puntos superpuestos.

Listing 2.6: Transformada aplicada a la nube de puntos.

```
tform = pcregistericp(moving, fixed, 'Metric','pointToPlane','Extrapolate',
    true);
ptCloudAligned = pctransform(ptCloud2,tform);
```

Representamos la escena original.

Listing 2.7: Representación de la escena original.

```
mergeSize = 0.015;
ptCloudScene = pcmerge(ptCloud, ptCloudAligned, mergeSize);
ptCloudSr = pcdenoise(ptCloudScene);
```

Se lleva a cabo una segmentación de la nube de puntos en grupos, con una distancia euclidiana mínima de minDistance entre puntos de diferentes grupos. Pcsegdist asigna una etiqueta compuesta por número enteros para cada grupo de la nube de puntos.

Listing 2.8: Segmentación y etiquetas.

```
minDistance = 0.5;
[label,numClusters] = pcsegdist(ptCloudSr,minDistance);
```

Se visualiza la nube de puntos segmentada como se muestra a continuación, también se pueden añadir límites de respresentación por distancia en cada una de las coordenadas cartesianas como se muestra en el comentario para la coordenada Z.

Listing 2.9: Visualización de la nube de puntos segmentada.

```
figure

pcshow(ptCloudSr.Location,label)
colormap(hsv(numClusters))
title('Point Cloud Clusters')
xlabel('X (m)')
```

```
ylabel('Y (m)')
zlabel('Z (m)')
%%zlim([1 5])
drawnow
```

También es posible visualizar el histograma con cada cluster de puntos y el número de puntos contenidos en cada uno de ellos.

LISTING 2.10: Visualización del histograma.

```
figure
histogram(label)
```

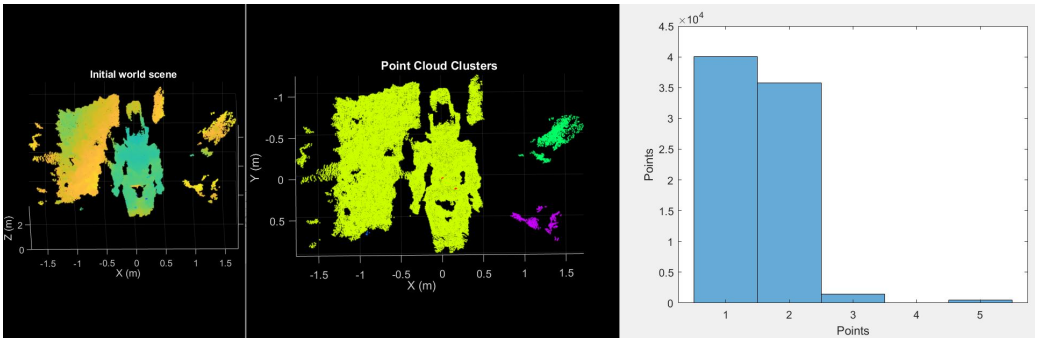

FIGURA 2.27: Proceso de segmentación con Matlab

2.5.2. Sistema de archivos

El sistema de archivos de *ROS* se divide básicamente en dos niveles:

- Paquetes: Son el nivel más bajo de organización del sistema de archivos de *ROS*. Pueden contener cualquier cosa: ejecutables (nodos), modelos, tipos de mensajes y servicios, herramientas o librerías.
- Pilas: Son agrupaciones de paquetes que forman una librería de alto nivel. Cada pila agrupa paquetes que son complementarios entre ellos.

Para compilar un paquete en *ROS* se utiliza CMake, una plataforma de código abierto para la generación, compilación y comprobación de paquetes de software. Su uso es bastante sencillo, basta con que el paquete de *ROS* contenga un archivo llamado CMakeLists.txt compuesto por una serie de macros según lo que se quiera crear y compilar.

2.5.3. Nodos

Un nodo es un módulo o proceso individual dentro del sistema de *ROS*, que realiza un cómputo que puede enviar y/o recibir información de otros nodos y usar y/u ofrecer servicios. Estos nodos no son más que los ejecutables de los paquetes de *ROS* ejecutándose.

Se muestra un ejemplo de dos nodos, talker y listener en el que, el primero está publicando en un topic, denominado chatter, y el segundo está subscrito a él.

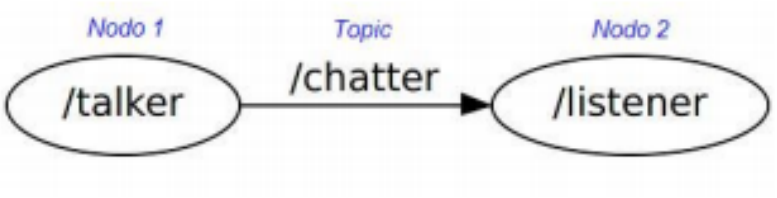

FIGURA 2.28: Nodos

Cada nodo posee un nombre único que lo identifica y lo distingue de todos los demás nodos en ejecución, los programas de los nodos pertenecientes a este proyecto se escribirán todos en C++.

2.5.4. Topic

Los topics son buses por los cuales los nodos envían o reciben mensajes. Simplemente un nodo debe comunicar al Master que desea publicar en él para enviar información o suscribirse en él para recibir su información. Pueden existir varios nodos publicando y varios nodos subscritos a la vez en un mismo topic. Todos los topics son unidireccionales, por lo cual un nodo que envía información por un topic nunca recibirá una respuesta directamente por él.

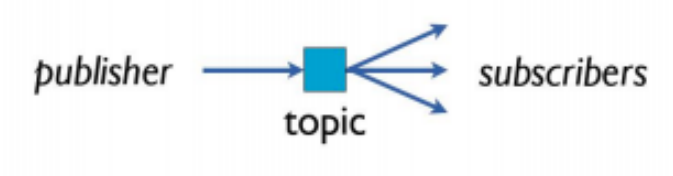

FIGURA 2.29: Topic

Todos los topics son unidireccionales, por lo cual un nodo que envía información por un topic nunca recibirá una respuesta directamente por el mismo topic, ni sabrá si sus mensajes están siendo recibidos. Si se desea enviar una información y recibir una respuesta, debe usarse el otro tipo de comunicación entre nodos, los servicios.

2.5.5. Servicios

Los servicios en *ROS* son la forma en que se envía un mensaje (request) desde un nodo a otro y éste le responde con otro mensaje (response)

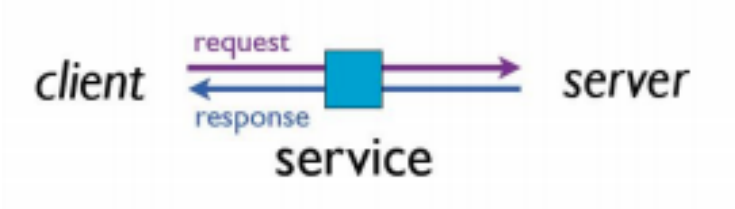

Figura 2.30: Servicio

Así un nodo puede ofrecer un servicio (server) y cualquier otro nodo puede utilizarlo (client) llamándolo junto con un mensaje y esperando una respuesta de éste, es decir, otro mensaje. El servicio ofrecido por un nodo es totalmente independiente de los clientes. Los servicios deben definir el tipo de servicio que se va a ofrecer, esto es el tipo de mensaje que se va a recibir como request y el tipo de mensaje que se va a devolver como response.

2.5.6. Master

El Master (o roscore) es un nodo que funciona de núcleo del sistema, proporcionando una plataforma mediante el registro de nombres, servicios y parámetros, sobre la cuál se hace posible la comunicación y el envío de datos entre los diferentes nodos individuales del sistema. Sin él los nodos no se “encontrarían” entre ellos.

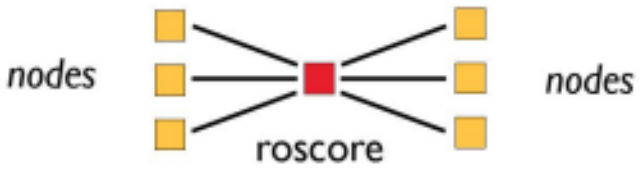

FIGURA 2.31: Master-Roscore

Realmente roscore es una herramienta que proporciona tres cosas:

- El Master, entendido como plataforma de comunicación para los nodos.
- El Servidor de Parámetros.
- Registro de salida al que todos los nodos envían información denominado /rosout.

Como podemos observar en la figura 4.4 , los dos nodos, uno (cámara) que envía imágenes y otro que las recibe, se dirigen antes al Master para crear y buscar, respectivamente, el topic antes de iniciar la comunicación entre ellos.

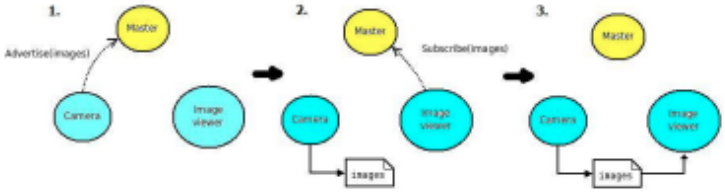

FIGURA 2.32: Funcionamiento básico Master-Roscore

Como ya se puede prever, la ejecución del Master es obligatoria antes de lanzar nodos que se comuniquen entre ellos o necesiten leer parámetros. Aunque algunos programas ya lo lanzan, al ser ejecutados, no siempre es así, por lo cual, se aconseja que siempre se haga roscore antes de empezar.

2.5.6.1. Lanzadores

En *ROS* es posible ejecutar nodos, llamara servicios y leer o publicar en topics directamente desde una terminal. Sin embargo, si lo que se desea es lanzar varios nodos a la vez, cargar una serie de parámetros o ejecutar varias herramientas, esa no sería la mejor

opción, ya que habría que hacerlo uno por uno. Por esta razón, existe en *ROS* una herramienta, que se ejecuta usando el comando roslaunch, que permite fácilmente lanzar múltiples nodos con sus argumentos, establecer una serie de parámetros en el servidor y otras opciones bastante interesantes, que se cargaran directamente en un archivo de configuración de extensión .launch.

Los lanzadores .launch son archivos de configuración escritos en XML en los que se especifican los nodos a lanzar, los argumentos de entrada y los parámetros necesarios para crear una aplicación completa. Como todo archivo XML, se compone de una serie de etiquetas (tags) que tienen una serie de opciones a rellenar.

Capítulo 3

Implementación y resultados

En este apartado se presenta el diseño de la aplicación desarrollada, que ha seguido las necesidades y especificaciones expuestas anteriormente. Se expone a continuación las decisiones de diseño tomadas y el diseño de la aplicación utilizado en este proyecto.

3.1. Introducción

El punto de partida de este proyecto es un conjunto de archivos .bag de ROS que almacenan los topics de nubes de puntos de interiores, donde se encuentra un robot humanoide de la NASA conocido como Valkyrie. Tenemos varios videos donde se puede comprobar al robot y diferentes objetos del entorno en diferentes situaciones. La versión de la librería que utilizaremos con ROS será la 1.7.0.

FIGURA 3.1: Valkyrie

La metodología seguida para la realización de este proyecto es la siguiente:

1. Partimos de un bag, en el cual se puede observar al robot y su entorno.
2. Se aplican diferentes filtros para eliminar las partes de la escena que no proporcionen información relevante.

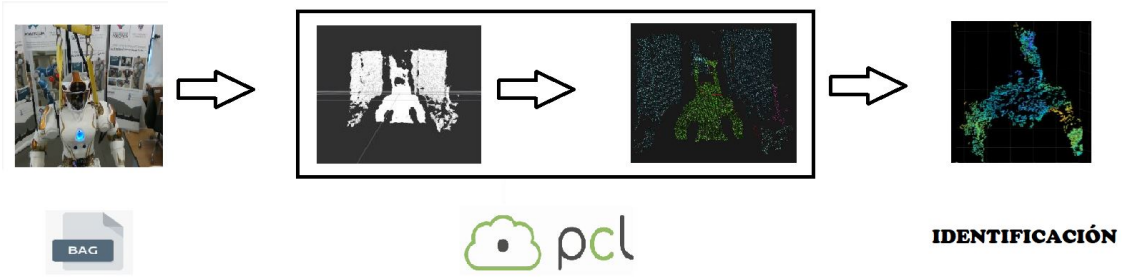

FIGURA 3.2: Metodología del proceso de segmentación

La cámara IntelRealSense ZR300 [4] va a ser el instrumento utilizado para la captura de imágenes.

Las cámaras pioneras en este ámbito fueron las Kinect, [12] las cuales comparten similitudes con las cámaras IntelRealSense, las cuales poseen las siguientes características:

Aplicaciones:

1. Reconocimiento de objetos, localización y tracking.

2. Reconocimiento de personas y gestos.

3. SLAM en tiempo real, mapeado y re-localización.

Descripción:

- Imagen de profundidad con visión estéreo.
- 6 grados de libertad en la unidad de visión inercial.
- Sensor óptico de ojo de pez en un único módulo con una interfaz USB3.0.
- Los datos son sincronización con un reloj con una marca de tiempo de 50 us.

Características:

- Imágenes ASIC.
- VGA 480x360 y resolución QVGA.
- Rango de capturas de profundidad 0.55 m a 2.8 metros.
- Sistema de proyección láser infrarrojo (clase 1).
- Unidad de visión inercial.
- Sincronización de reloj.
- Definición de color de 1080p RGB.
- Ojo de pez con salida monocromátrica VGA.

FIGURA 3.3: Cámara IntelRealSense zr300

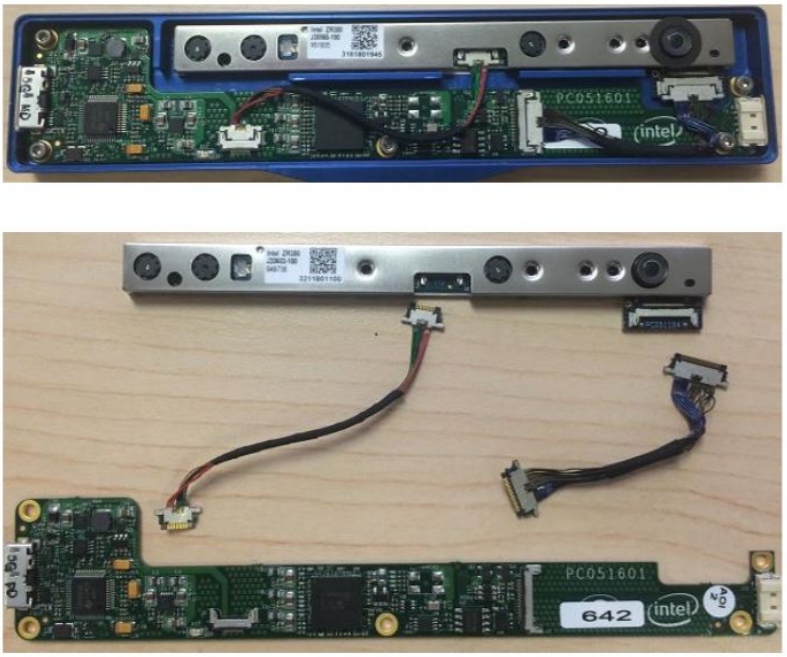

FIGURA 3.4: Componentes Cámara IntelRealSense zr300

Por otro lado, los topics generados por la cámara y utilizados en el desarrollo son los siguientes:

- /camera/points: Dicho topic proporciona una nube de puntos de las imágenes captadas por la cámara. El tipo de mensaje de este topic es del tipo sensor_msgs/PointCloud2, cuyos campos son los siguientes:

```
std_msgs/Header header
uint32 height
uint32 width
sensor_msgs/PointField[] fields
bool is_bigendian
uint32 point_step
uint32 row_step
uint8[] data
bool is_dense
```

FIGURA 3.5: Campos msgs_pointCloud2

- /camera/color/camera_info: Este topic proporciona meta información de la cámara para las imágenes publicadas en camera/image_raw. El tipo de mensaje es sensor_msgs/Camera_Info, el cual posee los siguientes campos:

```
std_msgs/Header header
uint32 height
uint32 width
string distortion_model
float64[] D
float64[9] K
float64[9] R
float64[12] P
uint32 binning_x
uint32 binning_y
sensor_msgs/RegionOfInterest roi
```

FIGURA 3.6: Campos msgs_Camera_Info

- /camera/color/image_raw: Este topic contiene las imágenes sin comprimir captadas por la cámara. El tipo de mensaje es sensor_msgs/Image, con los siguientes campos:

```
std_msgs/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

FIGURA 3.7: Campos msgs/Image

- /camera/color/image_raw/compressed: Este topic contiene las imágenes comprimidas captadas por la cámara. El tipo de mensaje es sensor_msgs/compressedImage, con los siguientes campos:

```
std_msgs/Header header
string format
uint8[] data
```

FIGURA 3.8: Campos msgs/compressedImage

- /camera/depth/image_raw: Este topic contiene la imagen de profundidad de las imágenes captadas por la cámara. Los mensajes son del tipo sensor_msgs/Image, cuya estructura se ha mostreado anteriormente.

- /camera/depth/image_raw/compressedDepth: Este topic contiene la imágen de profundidad captada por la cámara en un formato comprimido. Los mensajes son

del tipo sensor_msgs/compressedImage, cuya estructura se ha mostrado anteriormente.

Por otro lado, la forma de almacenar todos estos topics publicados por la cámara es utilizando un archivo.bag, dicho archivo almacena información sobre topic publicados. Es posible generar nuestro propio archivo .bag utilizando en comando rosbag record -a.

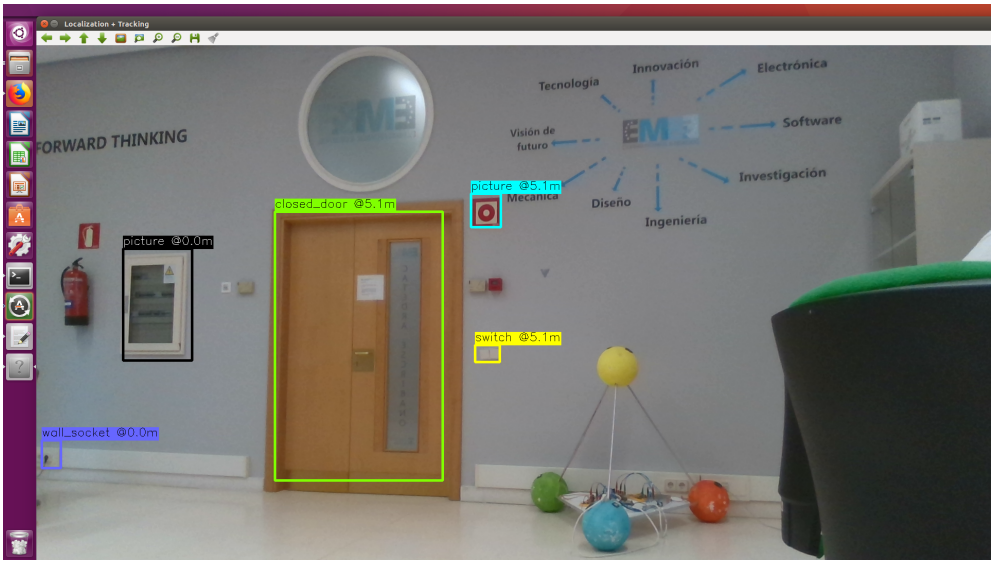

FIGURA 3.9

FIGURA 3.10: Captura del archivo .bag generado recogiendo los topics de imagen junto con la herramiena de tracking de la cámara.

Al grabar algunas pruebas con el comando anteriormente surgieron varios errores de conversión entre los formatos [8UC1] y [16UC1] con [bgr8].

Esto se debe a que [8UC1] y [16UC1] no son formatos de color, sino que indican que la imagen tiene un canal de 8 y 16 bits de tamaño respectivamente, por lo que la conversión no tiene sentido. Este problema se soluciona cambiando estos formatos por [mono8] y [mono16], los cuales sí son un formato de color monocromático de un tamaño de 8 y 16 bits de tamaño respectivamente.

Este cambio en el código se realiza en el archivo camera_node.cpp de la carpeta realsense_ros_camera.

También apareció otro error al intentar grabar con el comando *rosbag*, debido al transporte de imágenes de profundidad comprimida: *la compresión requiere imágenes de un solo canal de punto flotante de 32 bits o de 16 bits de profundidad bruta*. La posible solución a este problema es deshabilitar el topic *compressedDepth*.

La solución más sencilla y la que se ha adoptado ha sido la de ejecutar el comando *rosbag record -a -x "(.*)/compressed(.*)"* para llevar a cabo las grabaciones.

Podemos ver todos los topics almacenados en un archivo .bag en la siguiente imagen:

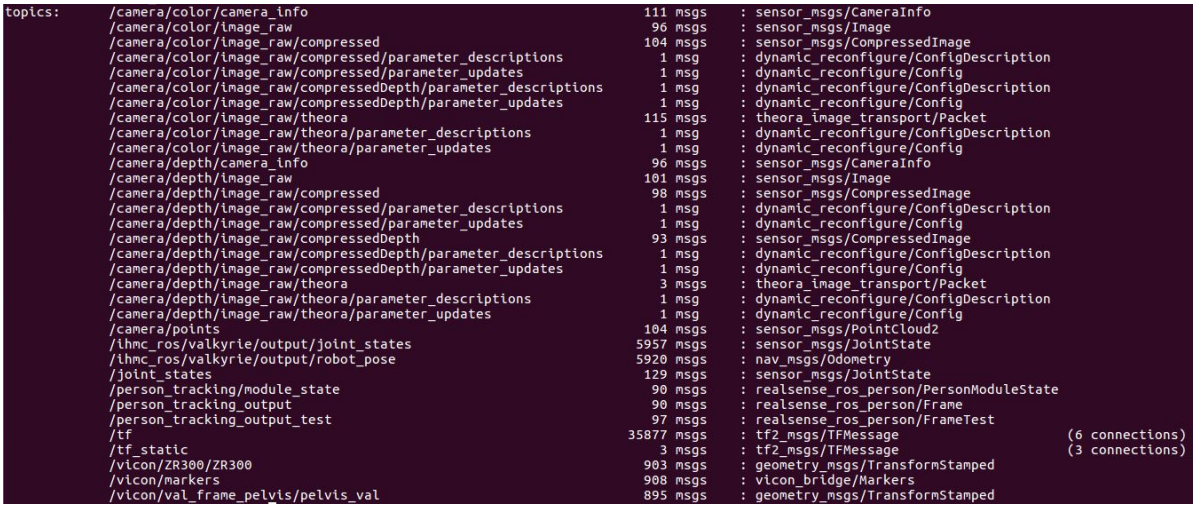

FIGURA 3.11: Topics almacenados en un archivo .bag

3.2. Algoritmo RANSAC

El algoritmo RANSAC (Random Simple Consensus) [2], Consenso de Muestra Aleatoria, es un método iterativo para calcular los parámetros de un modelo matemático de un conjunto de datos observados que contiene valores atípicos (outliers). Este modelo fue publicado por Fischler y Bolles en 1981. [1]

Una característica importante de este algoritmo es que para la realización de la estimación solo se tienen en cuenta los inliers (valores típicos del modelo que se esté tratando), a diferencia de otros modelos, como puede ser el caso de mínimos cuadrados, en el cual se toman en cuenta todos los puntos, tanto inliers como outliers.

Esta no diferenciación entre inliers y outliers puede provocar una distorsión del modelo, por eso RANSAC, en contraposición a estos, es considerado un método robusto en detección de características.

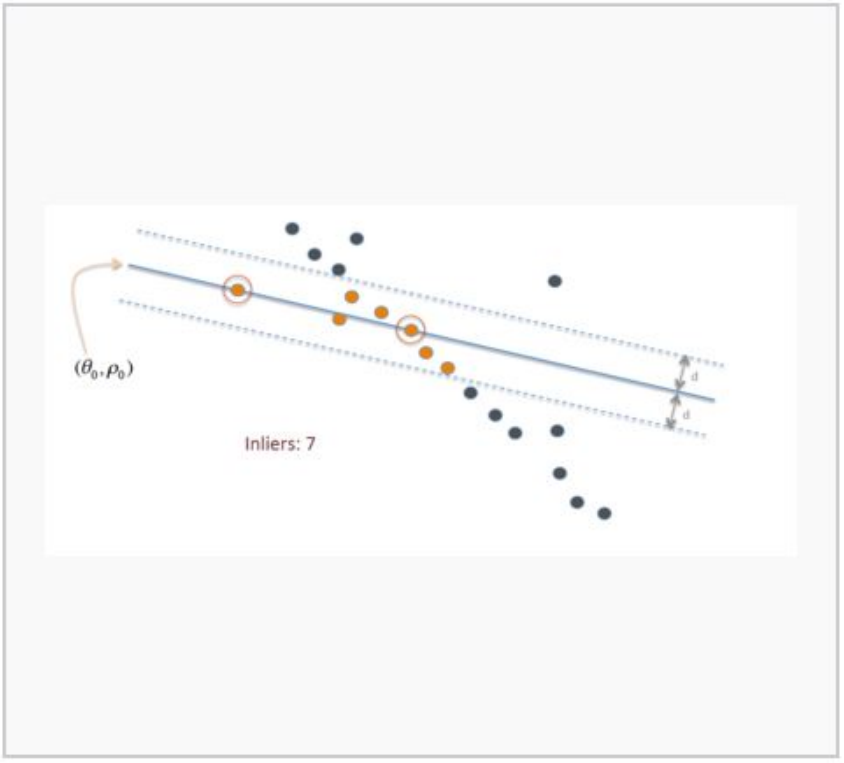

Figura 3.12: Algoritmo RANSAC

Existen dos tipos de errores frecuentes en deteción de características:

- **Errores de clasificacón:** Dichos errores tienen lugar cuando en un proceso de detección se identifica una zona como un inlier, es decir, una zona con unas características típicas del modelo que se está tratando, pero que en realidad no lo es, sino que es un outlier, es decir, una zona con unas características diferentes a las del modelo tratado. Estos errores no cumplen con ninguna determinada distribución.
- **Errores de medida:** Estos errores se producen cuando una determinada zona es detectada correctamente según las características del modelo, pero se produce un fallo a la hora de ubicarla en la imagen. Estos errores poseen una distribución normal, por lo que son facilmente evitables al tomar un gran número de muestras.

En nuestro caso los datos de entrada serán la nube de puntos, y el proceso iterativo para llevar acabo la detección es el siguiente:

1. Seleccionar un subconjunto aleatorio de los datos originales, llamados inliers hipotéticos.
2. Un modelo se monta en el conjunto de inliers hipotéticos.
3. Todos los demás datos se prueban contra el modelo ajustado. Esos puntos que se ajustan al modelo estimado, de acuerdo con alguna función de pérdida de modelos específicos, se consideran como parte del conjunto de consenso.
4. El modelo estimado es bueno si se han clasificado suficientes puntos como parte del conjunto de consenso .
5. Después, el modelo puede ser mejorado volviendo a estimar usando todos los miembros del conjunto de consenso.

Para un modelo RANSAC general encontramos las siguientes variables de entrada:

- Datos de entrada, en nuestro caso una nube de puntos.
- Modelo, que puede ser cilindro, esfera, plano, circunferencia, etc.
- n, el número mínimo de datos para ajustar el modelo.
- k, el número de iteraciones realizadas por el algorimo.
- t, un valor umbral que indica cuando los datos se ajustan al modelo.

- d, el número de valores de los datos necesario para afirmar que los datos de entrada se ajustan al modelo.

Las variables de salida serán las siguientes:

- Best-model los parámetros del modelo que mejor se ajustan a los datos.
- Best-consensus-set los puntos de los datos de entrada que han sido estimados como mejores frente al modelo.
- Best-error el error de este modelo relativo a la entrada.

3.3. Region Growing

Una Region Growing comprueba que el ángulo formado por las normales y la curvatura entre dos puntos no supere un valor máximo, considerando así que pertenecen a la misma superficie y por tanto al mismo cluster.

El funcionamiento de dicho algoritmo es el siguiente:

1. En primer lugar, se ordenan los puntos por su valor de curvatura, debido a que el algoritmo empieza por los puntos con valor de curvatura menor, estos son los puntos pertenecientes a las áreas planas. Comenzar por los planos permite tener un menor número de segmentos.
2. Una vez que estan todos los puntos etiquetados y la nube está ordenada, el proceso de agrupación se produce de la siguiente manera empezando por el punto de curvatura mínimo.
3. El punto seleccionado se agrupa al conjunto llamado semillas.
4. Para cada punto inicial, el algoritmo encuentra sus puntos vecinos.
5. Se comprueba el ángulo normal de cada vecino con el del punto inicial actual. Si el ángulo es menor que el valor umbral, el punto actual se agrega a la región actual.
6. Después de eso, cada vecino se comprueba con el valor de curvatura. Si la curvatura es menor que el valor umbral, este punto se agrega a las semillas.

7. La semilla actual se elimina del grupo de las semillas. Si el conjunto de semillas se vacía significa que el algoritmo ha hecho crecer la región y el proceso se repite desde el principio.

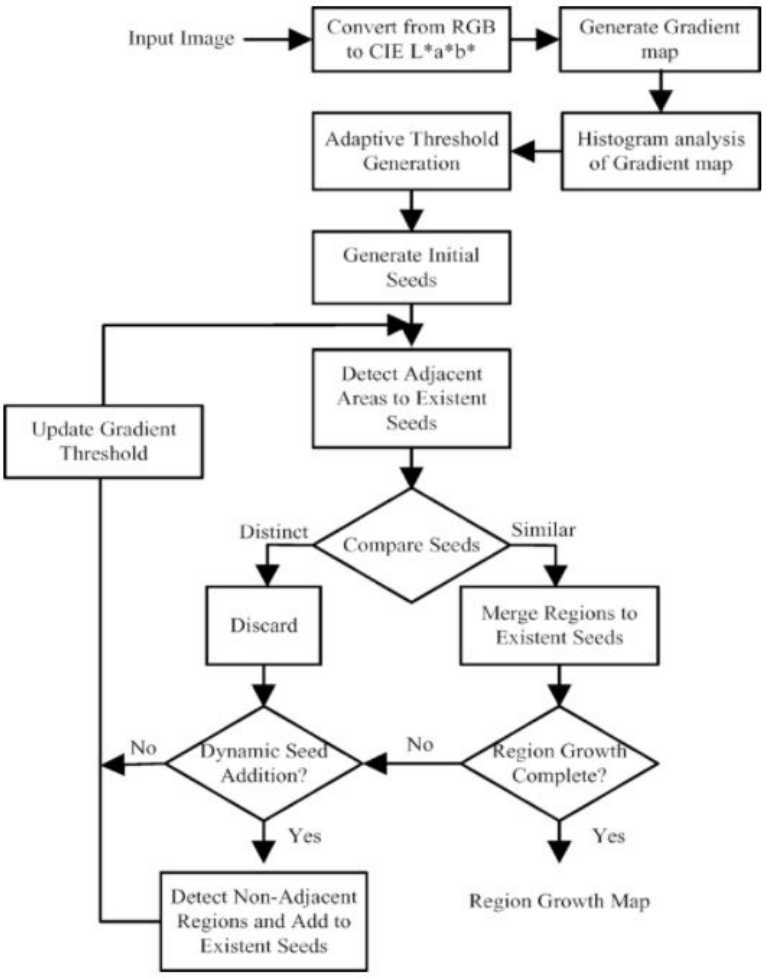

FIGURA 3.13: Diagrama de flujo del algoritmo Region Growing

Podemos encontrar varios aspectos a tener en cuenta a la hora de trabajar con dicho algoritmo.

- Es importante la correcta elección de los puntos semilla. Dependiendo de la parte de la imagen que queramos segmentar debemos elegir unos puntos con un determinado rango, por lo que la elección de estos puntos determina el resultado final de la segmentación.

- Cuanta más información tengamos de la imagen original mejor y más precisa será la segmentación, debido a que podremos definir los puntos semilla y umbrales con mayor precisión.
- Al llevar a cabo la segmentación, ningún resultado de este proceso puede ser menor que el valor ümbral de área mínima".
- El valor ümbral de similitud"tiene gran importancia a la hora de la segmentación, pues las regiones de pixeles que posean un valor menor, serán considerados pertenecientes al mismo grupo, por lo que no se hará una distinción entre ellas.

Algunas de las ventajas que posee este algoritmo son las siguientes:

- Poder separar regiones en torno a unas propiedades predefinidas.
- Buenos resultados de segmentación en imágenes con bordes claros.
- Fácil implementación, solo son necesarios un pequeño número de puntos semilla para representar la propiedad que marcará la región a segmentar, y su cultivo a posteriori.
- Posibilidad de tener varios criterios de segmentación al mismo tiempo.
- Eficiente al realizar la comprobación de cada pixel en un tiempo determinado.

Y con las siguientes desventajas:

- Es un método local, lo que no provee de una visión global sobre el problema.
- Sensible al ruido.
- A menos que se aplique una función umbral a la imagen, puede existir una ruta continua de puntos relacionados con el color que conecte dos puntos cualesquiera de la imagen.
- El acceso a memoria practicamente aleatorio ralentiza el algoritmo, por lo que puede ser necesaria una adaptación.

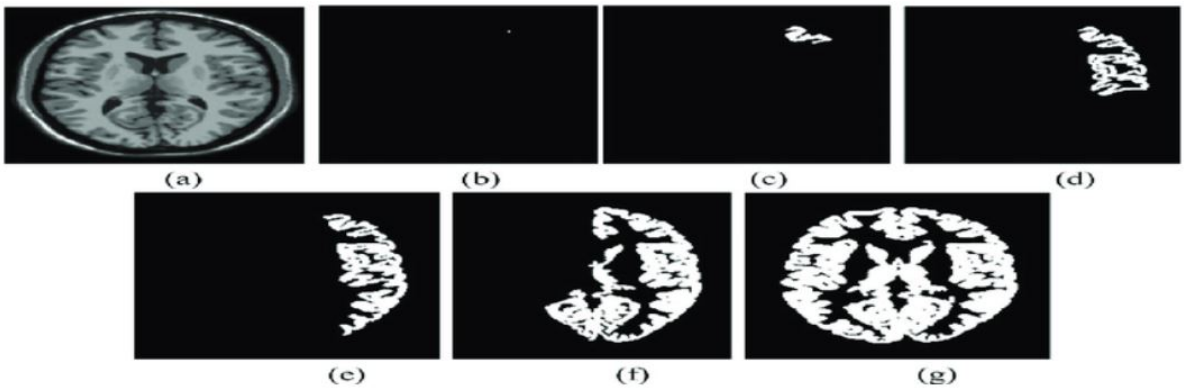

FIGURA 3.14: Proceso de selección de semillas. (a) Nube de puntos de muestra; (b) 1 punto semilla seleccionado; (c) 10 vecinos más cercanos añadidos; (d) 100 vecinos más cercanos añadidos; (e) 1000 vecinos más cercanos añadidos; (f) 2000 vecinos más cercanos añadidos; (g) Toda la región de puntos vecino es añadida, la región está completa

3.4. Segmentación

La biblioteca PCL segmentation contiene algoritmos para segmentar una nube de puntos en distintos grupos. Estos algoritmos son los más adecuados para procesar una nube de puntos que se compone de varias regiones aisladas espacialmente. En tales casos, la agrupación a menudo se usa para descomponer la nube en sus partes constituyentes, que luego se pueden procesar de forma independiente.

3.4.1. Clasificación de los filtros

Los filtros agrupados en esta librería se podrían clasificar en tres categorías:

a)Filtros que se aplican con el fin de eliminar datos erróneos o atípicos (outliers) medidos por el dispositivo de escaneo.

-Radius Outliers Removal: elimina puntos que no se encuentren en un radio dado un mínimo de puntos especificado.

-Conditional Removal: elimina puntos que no cumplan una condición previamente establecida.

-Statistical Outliers Removal: Elimina puntos dispersos de una zona local más densa. Se basa en suponer una distribución gaussiana de los puntos en esa zona.

-Remove NaN From Point Cloud: Elimina los puntos que no tiene valores numéricos como coordenadas. En su lugar tienen el valor NaN (not a number)

B)Filtros que se aplican para reducir el número de puntos de la nube(downsampling) quedándose con los más significativos o de interés para nuestros objetivos.

-Passthrough: Este filtro nos permite eliminar los puntos que estén dentro o fuera de un rango especificado. Este rango puede establecerse sobre cualquier coordenada XYZ, intensidad, color, pero solo un parámetro a la vez.

-Voxelgrid: Este filtro realiza un submuestreo de la nube de puntos, dividiéndola en cuadrículas (voxel), calcula el centroide de la cuadrícula y sustituye todos los puntos del voxel por el centroide calculado.

Permite configurar el número mínimo de puntos por voxel y el tamaño del voxel. Si un voxel no tiene ese número mínimo de puntos que se especifica, el voxel es descartado.

c)Filtros que no reducen el número de puntos de la nube sino que los modifica.

-Bilateral: Con este filtro no se eliminan los datos atípicos, sino que se corrigen en función de los valores de sus vecinos, bajo el supuesto de que la imagen varía lentamente en el espacio. Con este filtro se mantiene la estructura inicial que tuviese la nube.

- **Truncamiento por vóxel:** En primer lugar, antes de comenzar a tratar una nube de puntos, debemos reducir el tamaño de la muestra con motivo de simplificar el tratamiento de la imagen pero sin perder detalle de la misma.

  Un vóxel es una cuadrícula 3D en el espacio. Esta cuadrícula se aproxima sobre los datos de la nube de puntos de entrada. Entonces, todos los puntos que se encuentren dentro de cada voxel serán aproximados a su centroide y, por lo tanto, se reducirá el número de puntos.

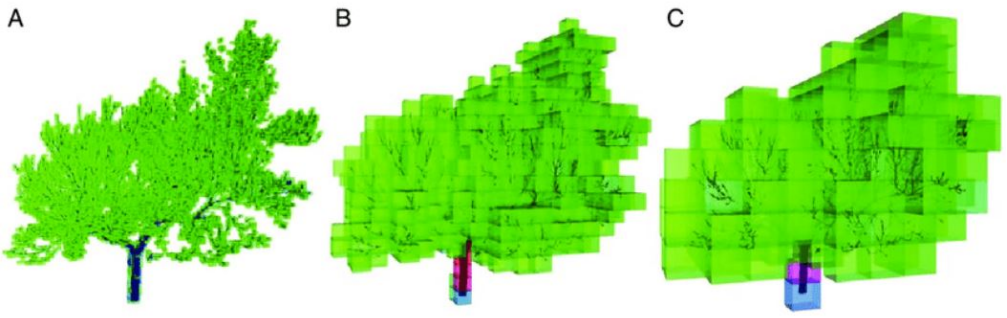

FIGURA 3.15: Ejemplos de voxelización utilizando vóxeles de resolución 0.1 m (A), 0.5 m (B), 1 m (C)

- **Octree:** Un Octree es una estructura de datos de tipo árbol jerárquico utilizada para subdividir nubes de puntos en conjuntos para realizar el procesamiento de forma eficiente. Para esta estructura se utilizan algortimos de búsqueda tales como "Búsqueda de vecinos vóxel","Búsqueda del vecino más cercano determinado por un radio K", cuya ventaja es la de ajustarse automaticamente al tamaño de la nube de puntos.

FIGURA 3.16: Octree

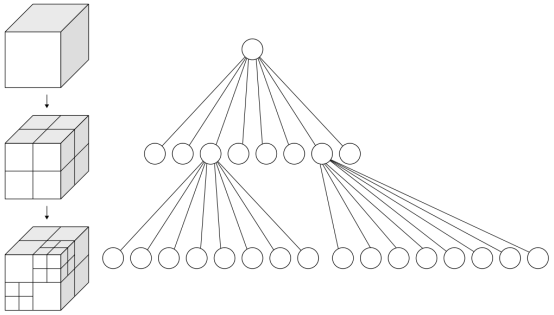

FIGURA 3.17: Representación Octree

- **KdTree:** Es una estructura de tipo árbol que almacena un conjunto de puntos k-dimensional con el fin de realizar búsquedas eficientes del vecino más cercano. Puede ser usado para encontrar las correspondencias entre grupos de puntos para los métodos de características geométricas, o en descriptores para definir la zona local alrededor de un punto o puntos.

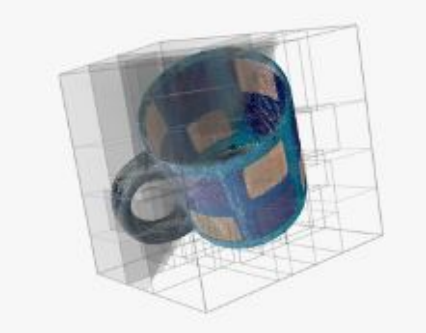

FIGURA 3.18: Kdtree

- **Segmentación con colores:** Este tipo de segmentación hace uso del algotitmo *region growing* explicado anteriormente, pero existe hay dos diferencias en el algoritmo basado en color respecto al algoritmo region growing.

  1. La primera de ellas es que usa colores en lugar de normales.
  2. La segunda es que hace uso del algoritmo de fusión para el control de la segmentación por encima y por debajo de los umbrales establecidos.

  Después de la segmentación se intentan fusionar clústeres con colores cercanos. Dos grupos vecinos que tengan una pequeña diferencia ante el color promedio se fusionarán. Entonces el segundo paso de fusión tiene lugar. Durante este paso cada uno de las nubes es verificada por el número de puntos que contiene. Si este número

de puntos está por debajo del valor establecido el cluster actual se fusionará con el cluster vecino más cercano.

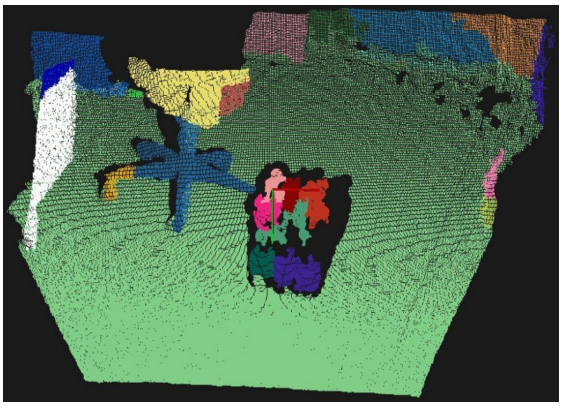

FIGURA 3.19: Segmentación por colores

- **Segmentación con normales para obtener cilindros:** En este filtro, se emplea un estimador RANSAC (Random Sample Consensus),que es un algoritmo iterativo que comprueba si los puntos de una nube se corresponden o pertenecen a un modelo geométrico definido (plano, esfera, cilindro, etc).

- **Segmentación con normales:** En este filtro el objetivo del algoritmo es extraer los planos de la nube de puntos capturada, trabajando con vecindades de puntos sobre los cuales se ajusta un plano para calcular la normal en cada punto de la nube.

  Luego, examinando las distancias entre los puntos y procediendo, en orden creciente de distancias, se evalúan las normales y según estas son similares y no muy separadas en distancias, se agrupan en segmentos.

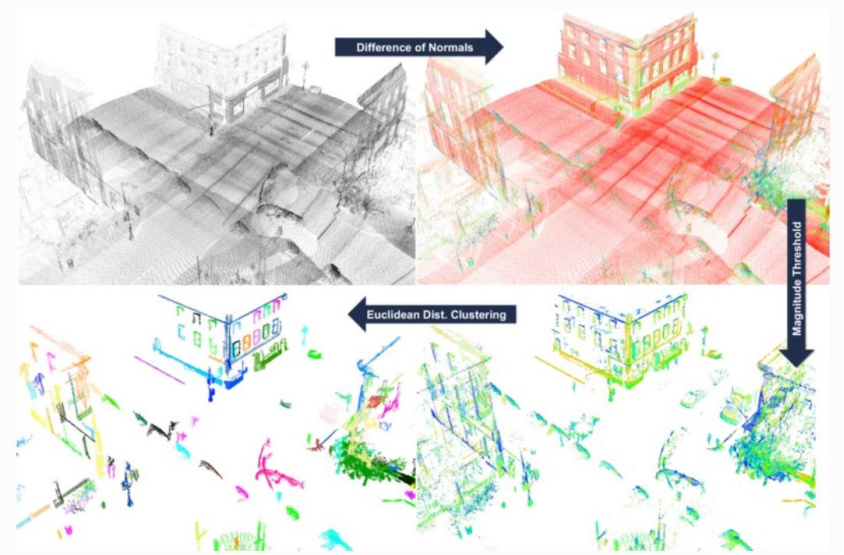

FIGURA 3.20: Segmentación con normales

- **Filtrado por distancia:**

  Dentro de las librerías de la cámara RealSense, encontramos funcionalidades para el tracking de personas.

  A partir de los datos obtenidos de las funciones centerOfMassWorld.x, centerOfMassWorld.y y centerOfMassWorld.z, se puede llevar a cabo una segmnetación por distancia. El valor de x básicamente da la distancia entre la cámara y el valkyrie y el valor de z indica la dirección relativa del usuario desde el centro del encuadre.

  Este tipo de filtrado es muy práctico a la hora de programar al robot para que se mueva a fin de mantener una distancia contante de las personas detectadas.

- **Euclidian Cluster Extration:**

  Una parte principal de este algoritmo es la segmentación de planos, la cual ya se ha explicado anteriormente, por lo que solo se va a explicar la parte no comentada.

  Un método de agrupación debe dividir un modelo de nube de puntos no organizado P en partes más pequeñas para que el tiempo de procesamiento total de P se reduzca significativamente.

  Se puede implementar un enfoque de agrupamiento de datos simple en un sentido euclidiano haciendo uso de una subdivisión de cuadrícula 3D del espacio utilizando cuadros de ancho fijo, o más generalmente una estructura de datos de árbol octal.

Esta representación particular es muy rápida de construir y es útil para situaciones donde se necesita una representación volumétrica del espacio ocupado, o los datos en cada caja 3D resultante (u hoja de octárbol) se pueden aproximar con una estructura diferente.

Sin embargo, en un sentido más general, podemos hacer uso de los vecinos más cercanos e implementar una técnica de agrupamiento que es esencialmente similar a un algoritmo de relleno de inundación, que lo que busca es determinar el área formada por elementos contiguos en una matriz multidimensional.

Supongamos que hemos dado una nube de puntos con un plano y una persona delante del plano (o un plano y objetos encima de él), queremos encontrar y segmentar los grupos de puntos de objetos individuales que se encuentran en el plano. Suponiendo que usamos una estructura de árbol Kd para encontrar los vecinos más cercanos, los pasos algorítmicos a seguir serían los siguientes:

1. Crear una representación de árbol Kd para el conjunto de datos de la nube de puntos de entrada P;
2. Establecer una lista vacía de racimos C, y una cola de los puntos que necesitan ser comprobados Q;
3. Luego, para cada punto p_i en P, realizar los siguientes pasos:
   - Agregar p_i a la cola actual Q;
   - Por cada punto $p_i \in Q$ hacer:
     - Buscar el conjunto P_i^k de puntos vecinos de p_i en una esfera con radio $r < d_{th}$;
     - Para cada vecino $p_i^k \in P_i^k$, verificar si el punto ya ha sido procesado y, si no, agregarlo a Q;
   - Cuando la lista de todos los puntos de Q se ha procesado, añadir Q a la lista de grupos C, y restablecer Q a una lista vacía.
4. El algoritmo termina cuando todos los punto $p_i \in P$ han sido procesados y ahora forman parte de la lista de grupos de puntos C.

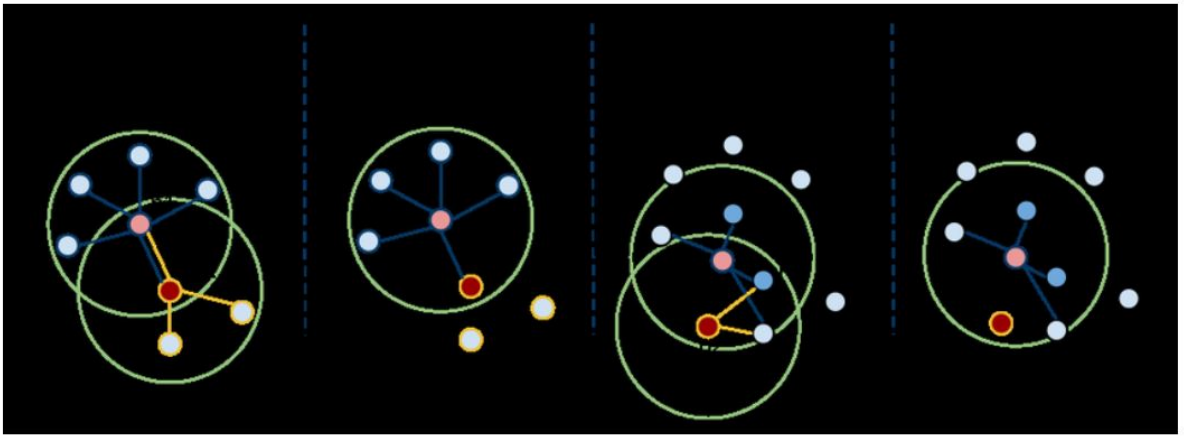

FIGURA 3.21: Representación de la extracción Kdtree

3.5. Pruebas

A partir de los filtros estudiados anteriormente se van a llevar a cabo las pruebas con el robot Valkyrie.

3.5.1. Resultado de aplicar truncamiento por vóxel

Este filtro crea una cuadrícula de vóxeles 3D(es parecido a un conjunto de diminutas cajas 3D en el espacio) sobre los datos de la nube de puntos de entrada. Luego, en cada vóxel todos los puntos presentes se aproximarán con su centroide. Este enfoque es un poco más lento que aproximarlos con el centro del vóxel, pero representa la superficie subyacente con mayor precisión.

Tenemos que crear las nubes de puntos de entrada(*cloud*) y de salida(*cloud_filtered*).

LISTING 3.1: Creamos el filtro para el vóxel

```
pcl::VoxelGrid<pcl::PCLPointCloud2> sor
```

El método setLeafSize() asigna el tamaño de los vóxel.. Asignando por parámetro 3 valores que serán los tamaños respecto a las coordenadas x, y, z. Probando diferentes tamaños comprobamos que asignando un valor más pequeño, el filtrado queda de una forma más uniforme, lo que se explica por qué le estamos asignando una distancia muy pequeña entre vóxeles. Al hacer lo contrario y aumentar el tamaño obtenemos mucha más separación entre voxeles y el filtrado no queda tan uniforme.

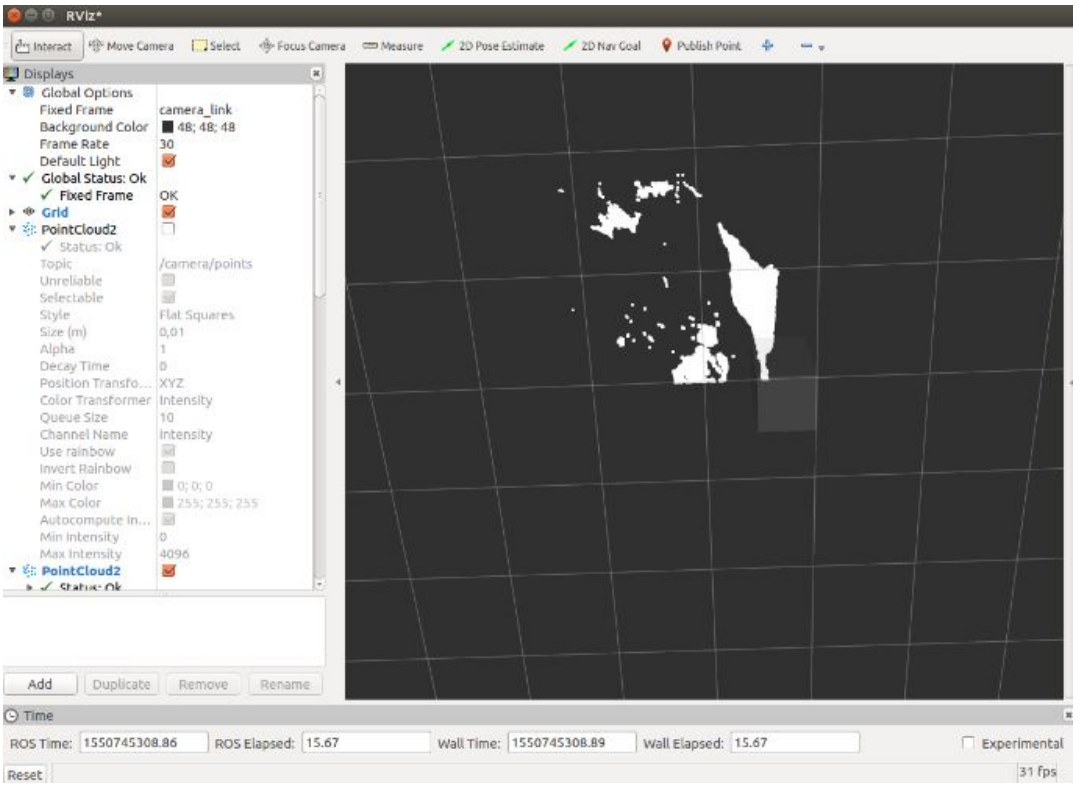

FIGURA 3.22: Cuadrículas voxel

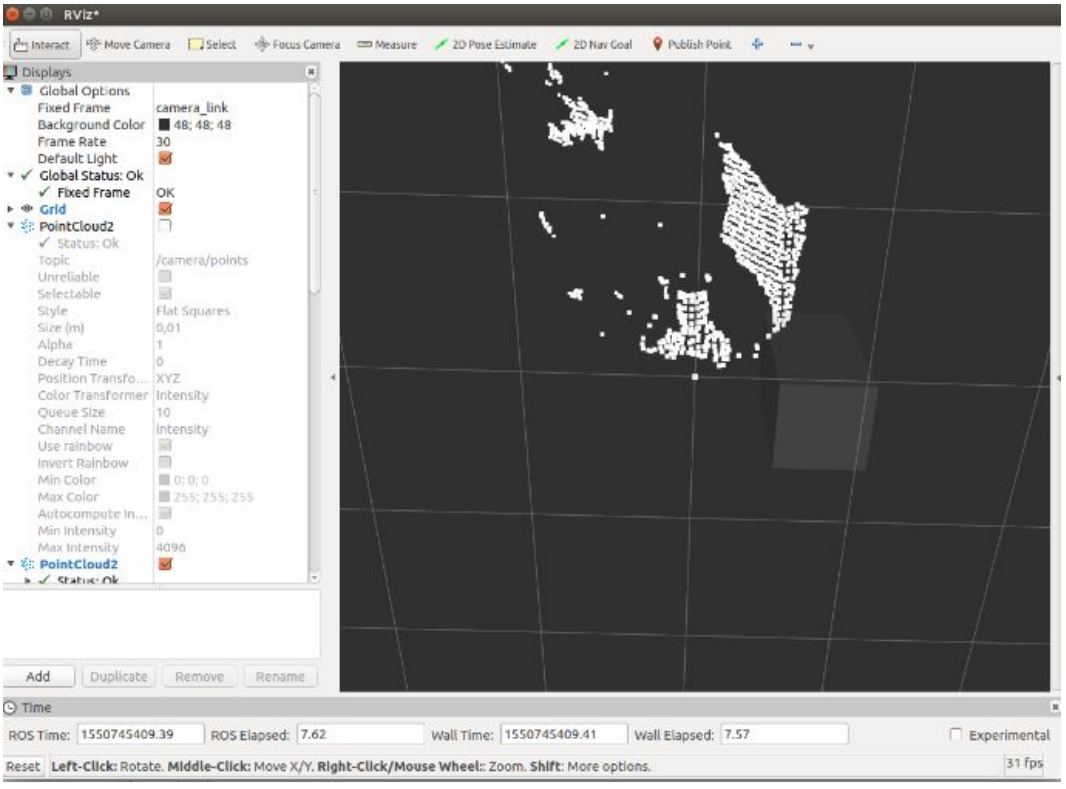

FIGURA 3.23: Cuadrículas voxel

3.5.2. Resultado de aplicar búsqueda con árbol octal

Un octree es una estructura de datos basada en un árbol para administrar datos 3D dispersos. Cada nodo interno cuenta con exactamente 8 hijos.

Primero debemos crear una instancia de octree que se inicializa con su resolución

Listing 3.2: Instanciación del Octree

```
pcl::octree::OctreePointCloudSearch<pcl::PointXYZ> octree(resoluction);
```

Este octree guarda un vector de índices de los puntos dentro de sus nodos hojas.
El parámetro de la resolución describe la longitud de los voxels más pequeños en el nivel más bajo del octree.
La profundidad del octree es, por tanto, una función de la resolución, así como la dimensión espacial lo es de la nube de puntos.
Si se conoce un cuadro delimitador de la nube de puntos, debe asignarse al octree utilizando el método *defineBoundinhBox*. Luego asignamos un puntero a la nube de puntos y agregamos todos los puntos al octree.

Una vez que la nube de puntos se ha asignado al octree, podemos realizar operaciones de búsqueda.

El primero de ellos es *neighbors within voxel search*, el cual asigna el punto de búsqueda al voxel del nodo hoja correspondiente y devuelve un vector de índices de puntos. Estos índices se relacionan con puntos que caen dentro del mismo voxel. La distancia entre el punto de búsqueda y el resultado de la búsqueda depende, por tanto, del parámetro de resolución del octree.

Listing 3.3: Neighbors within voxel search

```
std::vector<int> pointIdxVec;
 if (octree.voxelSearch (searchPoint, pointIdxVec))
    {
    for (size_t i = 0; i < pointIdxVec.size (); ++i)
       std::cout << "    " << cloud_pcl->points[pointIdxVec[i]].x << " " <<
  cloud_pcl->points[pointIdxVec[i]].y << " " <<
  cloud_pcl->points[pointIdxVec[i]].z << std::endl;
    }
```

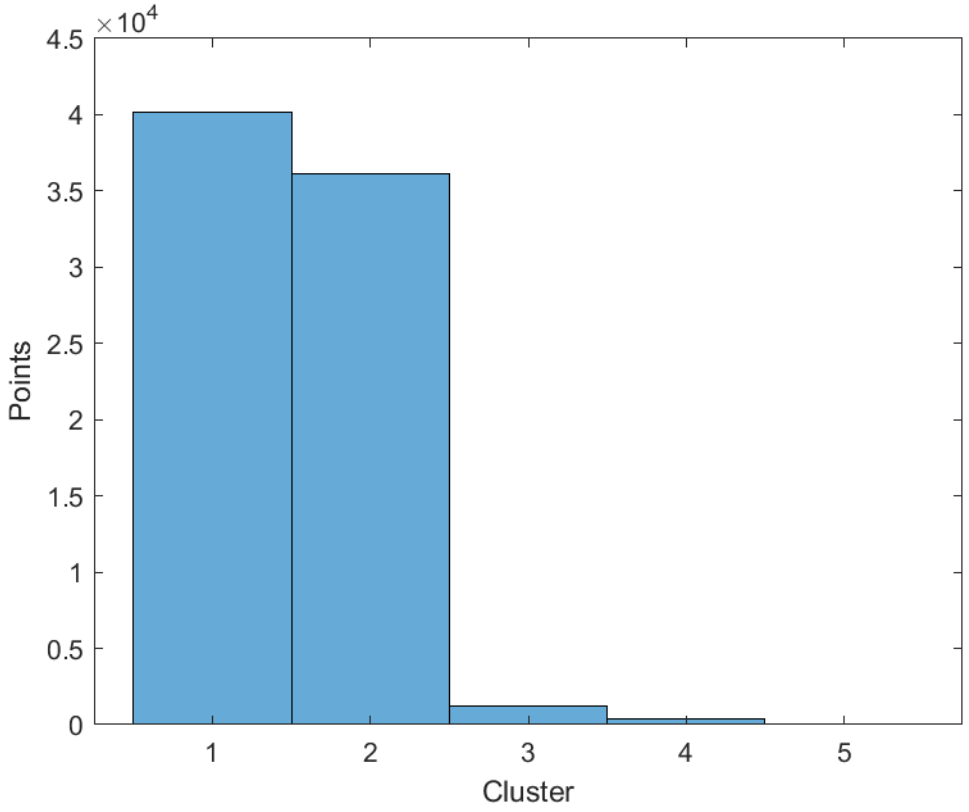

FIGURA 3.25: Conjuntos de nubes de puntos tras la segmentación

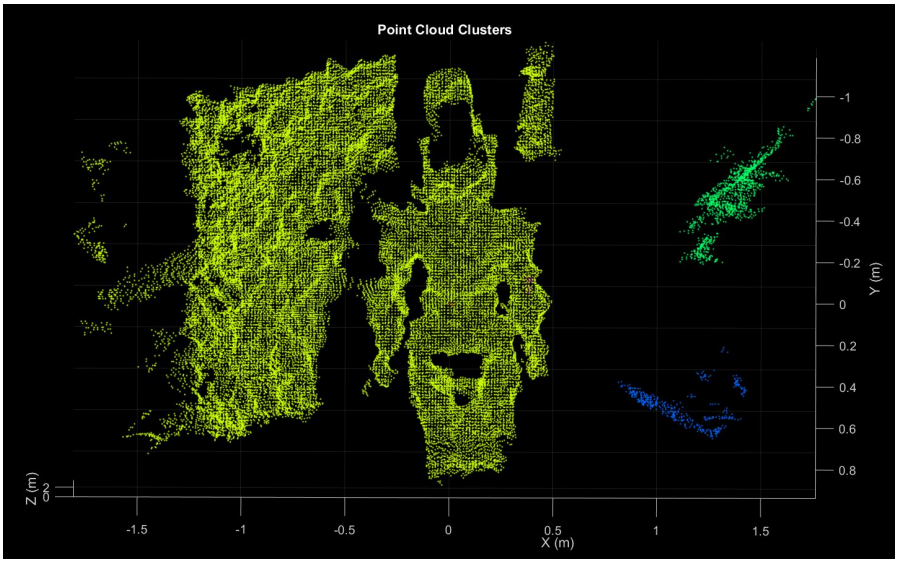

FIGURA 3.24: Búsqueda por voxel

A continuación, se muestra una K-búsqueda del vecino más cercano. Se van a llevar a cabo dos pruebas, con 10 y 20 K vecinos. El método de "Búsqueda de vecinos más cercanos" escribe los resultados de la búsqueda en dos vectores separados:

Listing 3.4: Neighbors within voxel search

```
int K = 20;
 std::vector<int> pointIdxNKNSearch;
 std::vector<float> pointNKNSquaredDistance;
 std::cout << "K nearest neighbor search at (" << searchPoint.x << " " <<
  searchPoint.y << " " << searchPoint.z << ") with K=" << K << std::endl;
 if (octree.nearestKSearch (searchPoint, K, pointIdxNKNSearch,
  pointNKNSquaredDistance) > 0)
    {
    for (size_t i = 0; i < pointIdxNKNSearch.size (); ++i)
       std::cout << "    "  <<   cloud_pcl->points[ pointIdxNKNSearch[i] ].x
  << " " << cloud_pcl->points[ pointIdxNKNSearch[i] ].y << " " <<
  cloud_pcl->points[ pointIdxNKNSearch[i] ].z << " (squared distance: " <<
  pointNKNSquaredDistance[i] << ")" << std::endl;
    }
```

1. El primero (pointIdxNKNSearch) contendrá el resultado de la búsqueda (los índices se refieren al conjunto de datos de PointCloud asociado).

2. El segundo vector contiene las distancias cuadradas correspondientes entre el punto de búsqueda y los vecinos más cercanos.

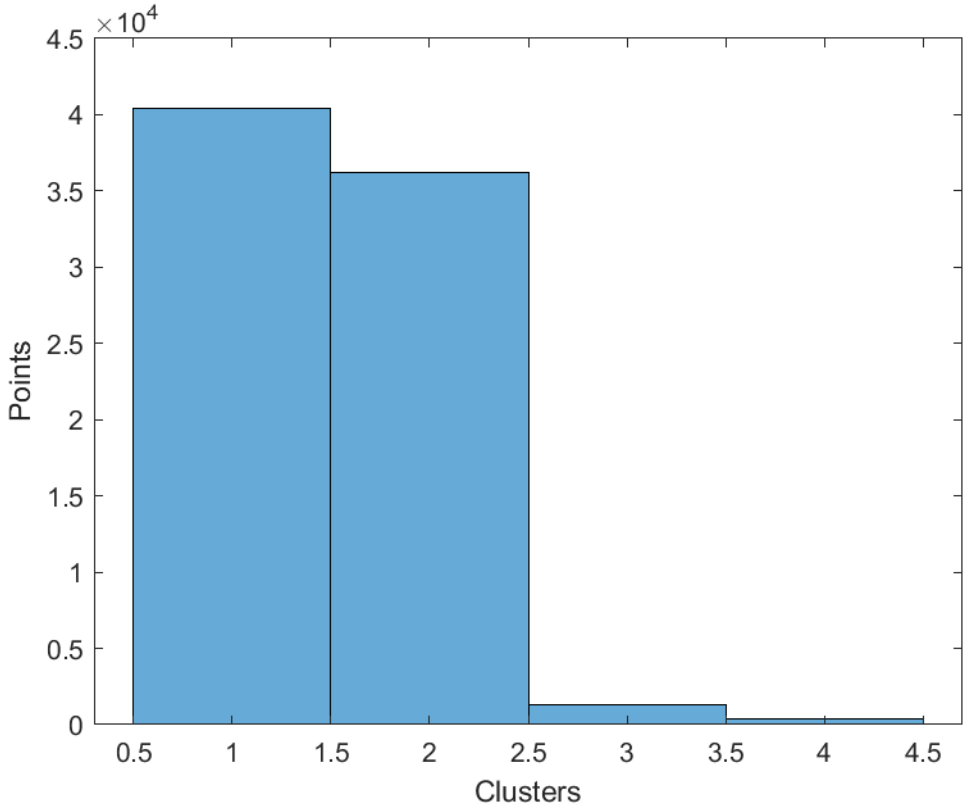

FIGURA 3.27: Conjuntos de nubes de puntos tras la segmentación

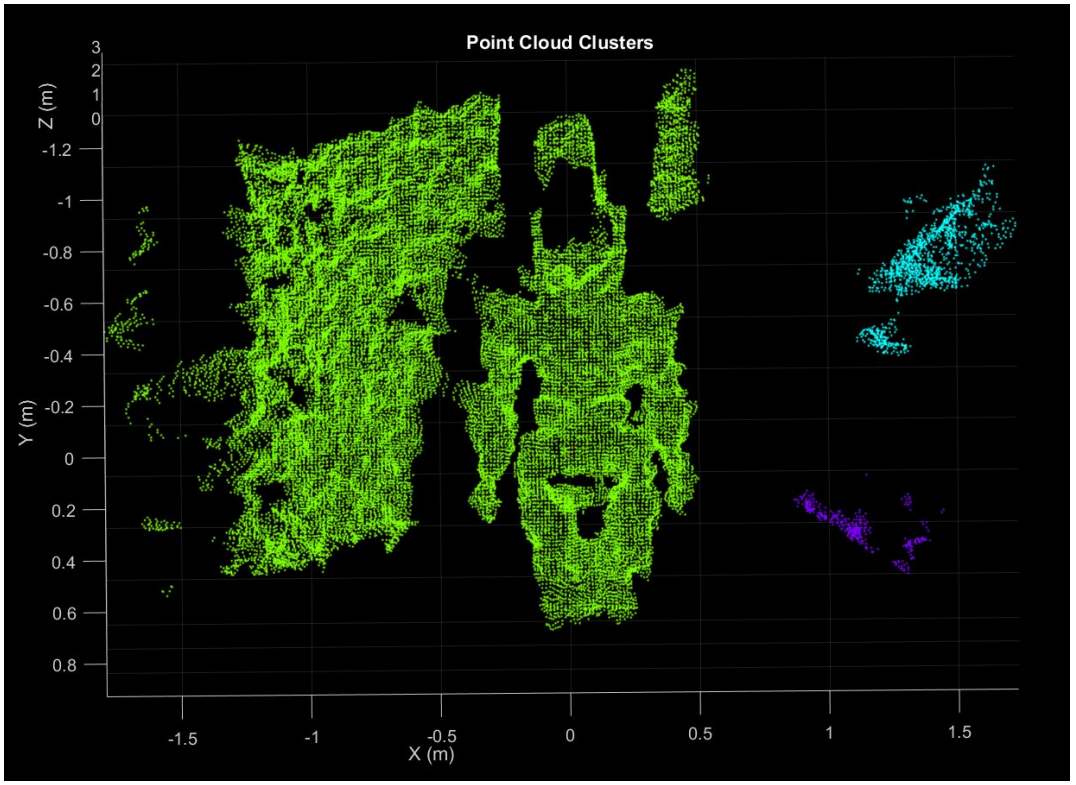

FIGURA 3.26: K = 10

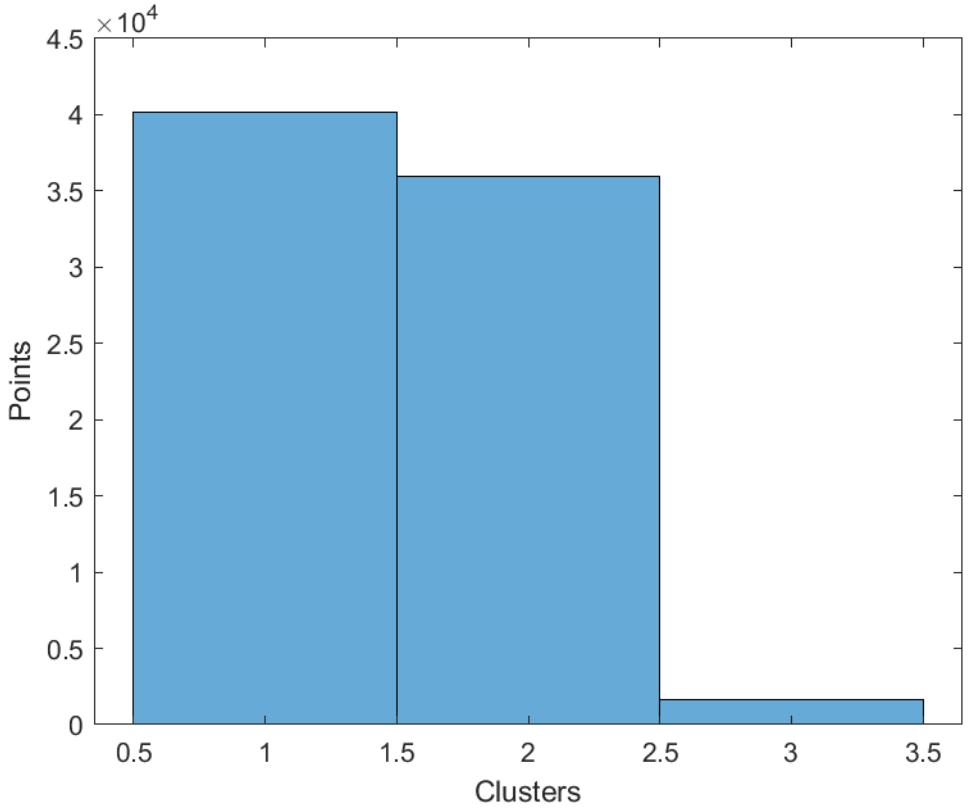

FIGURA 3.29: Conjuntos de nubes de puntos tras la segmentación

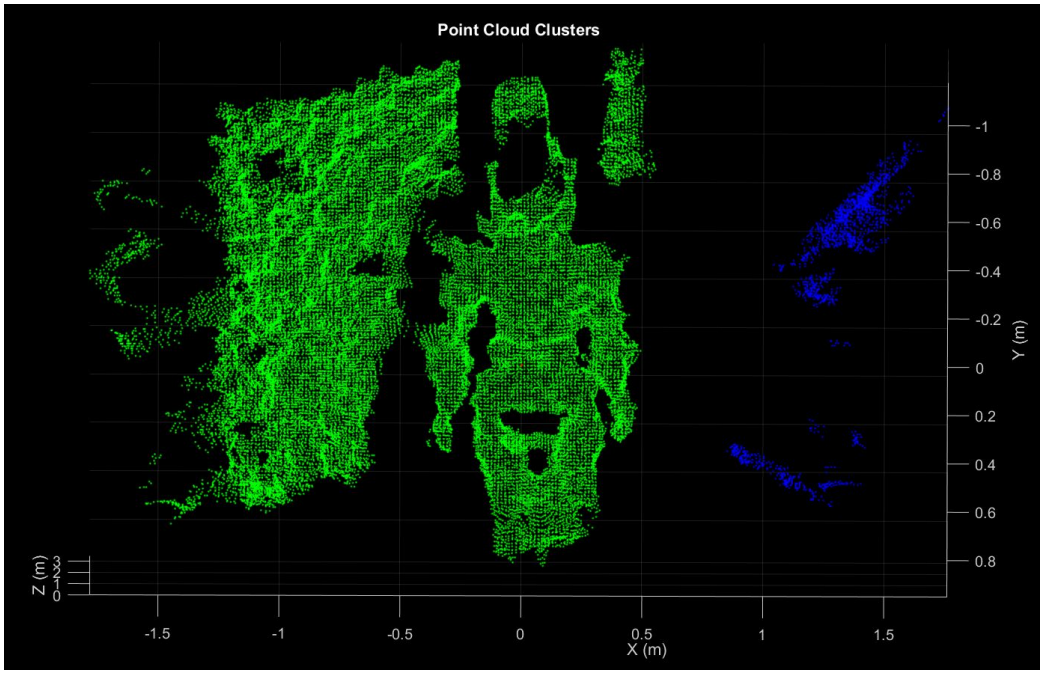

FIGURA 3.28: K = 20

Si se aumenta el número de k vecinos, no se observan variaciones en el modelo generado.

La "Búsqueda de vecinos dentro del radio"funciona de manera muy similar a la "Búsqueda de vecinos más cercanos K". Sus resultados de búsqueda se escriben en dos vectores separados que describen índices de puntos y distancias de puntos de búsqueda de cuadrados.

LISTING 3.5: Neighbors within radio search

```
std::vector<int> pointIdxRadiusSearch;
 std::vector<float> pointRadiusSquaredDistance;
 float radius = 256.0f * rand () / (RAND_MAX + 1.0f);
 std::cout << "Neighbors within radius search at (" << searchPoint.x << " "
  << searchPoint.y << " " << searchPoint.z << ") with radius=" << radius <<
  std::endl;
 if (octree.radiusSearch (searchPoint, radius, pointIdxRadiusSearch,
  pointRadiusSquaredDistance) > 0)
    {
    for (size_t i = 0; i < pointIdxRadiusSearch.size (); ++i)
       std::cout << "    "  <<   cloud_pcl->points[ pointIdxRadiusSearch[i]
 ].x << " " << cloud_pcl->points[ pointIdxRadiusSearch[i] ].y << " " <<
 cloud_pcl->points[ pointIdxRadiusSearch[i] ].z << " (squared distance: "
 << pointRadiusSquaredDistance[i] << ")" << std::endl;
    }


    }
```

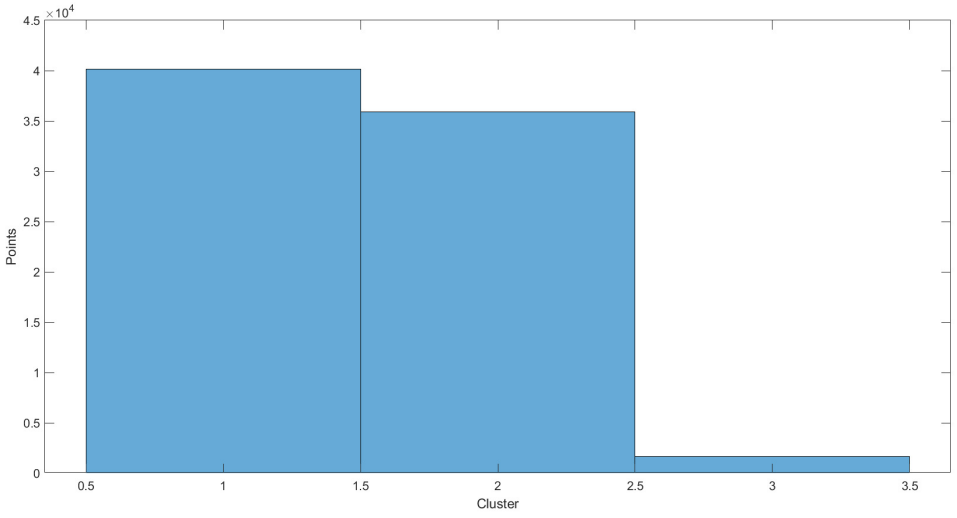

FIGURA 3.30: Conjuntos de nubes de puntos después de la segmentación

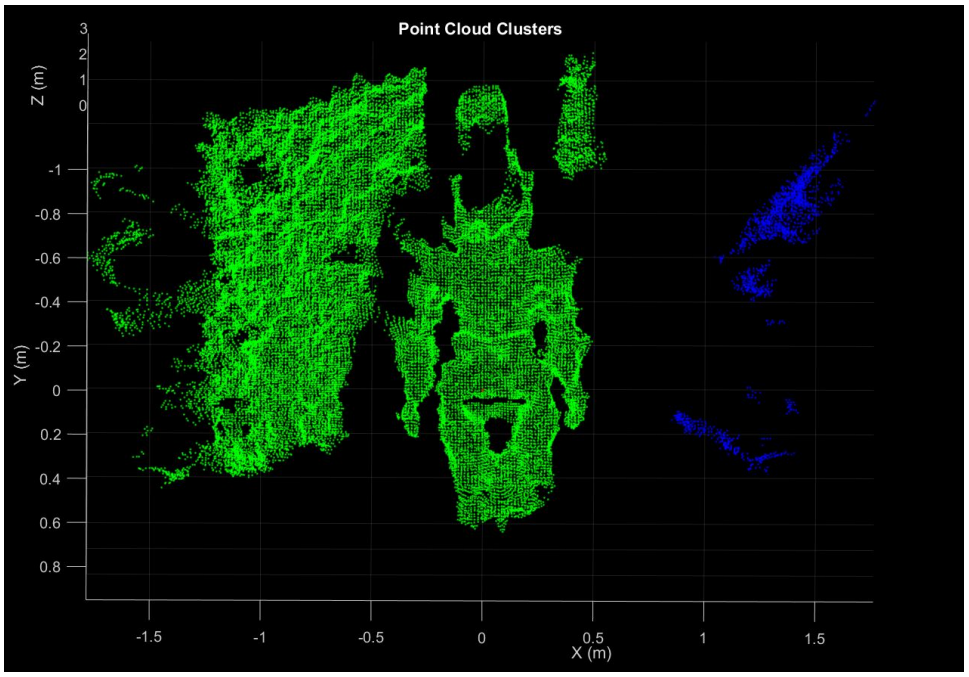

FIGURA 3.31: K = 20

Si aumentamos el tamaño del radio de búsqueda de 256 a 512 obtenemos unos resultados similares.

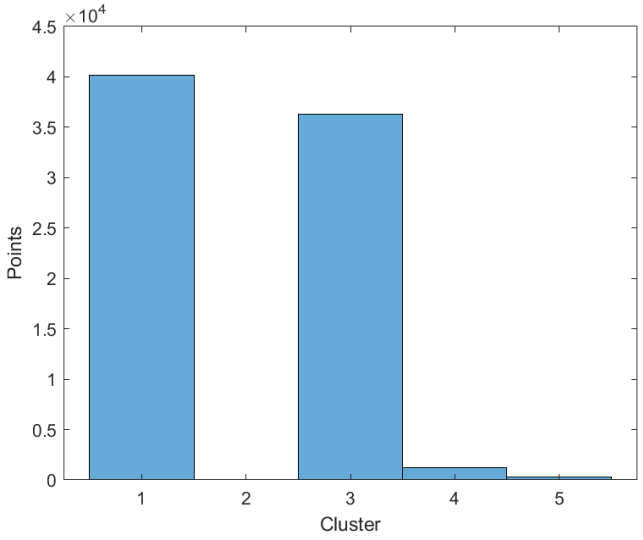

FIGURA 3.32: Conjuntos de nubes de puntos tras la segmentación

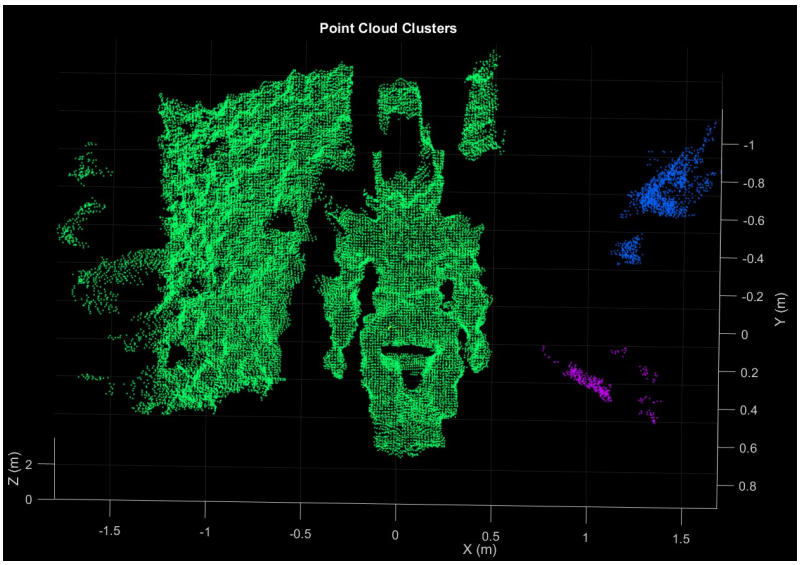

FIGURA 3.33: Nube de puntos generada tras aumentar el radio de búsqueda

Ocurre prácticamente lo mismo si en vez de aumentar reducimos el tamaño del radio a la mitad, 128.

Se han probado los 3 algoritmos mencionados anteriormente de forma separada y también de forma conjunta.

Todos ellos se ajustan correctamente al modelo del entorno, pues se pueden observar dos grandes agrupaciones, el fondo plano que representan las paredes y la figura del robot en cada uno de los algoritmos. Se pueden observar también pequeñas agrupaciones de nubes de puntos que representan otras partes de la imagen con menos puntos.

3.5.3. Resultado de aplicar segmentación con colores

Es necesaria la instanciación de la clase

LISTING 3.6: Instanciación RegionGrowing

```
pcl::RegionGrowingRGB<PointT> reg;
```

Una Region Growing comprueba que el ángulo formado por las normales y la curvatura entre dos puntos no supere un valor máximo, considerando así que pertenecen a la misma superficie y por tanto al mismo cluster.

A dicha clase se le debe de introducir la nube de puntos de entrada, los índices y el método de búsqueda, que en este caso será un árbol de búsqueda

Listing 3.7: Instanciación RegionGrowing

```
reg.setInputCloud(cloud_pcl);
reg.setIndices(indices);
reg.setSearchMethod(tree);
```

También se ajustan los umbrales del color para probar los colores de los puntos para saber a qué región pertenece.

Listing 3.8: Instanciación RegionGrowing

```
reg.setPointColorThreshold()
```

Se establece el umbral del color para los grupos, este valor es similar al anterior pero se usa cuando tiene lugar el proceso de fusión.

Listing 3.9: Instanciación RegionGrowing

```
regionColorThreshold()
```

Establecemos que si el clúster tiene menos puntos que los establecidos se fusionará con el vecino más cercano.

Listing 3.10: Instanciación RegionGrowing

```
reg.setMinClusterSize()
```

A continuación, tiene lugar la parte donde se lanza el algoritmo, devolverá la matriz de clústeres cuando finalice el proceso de segmentación:

Listing 3.11: Instanciación RegionGrowing

```
std::vector <pcl::PointIndices> clusters;
reg.extract (clusters);
```

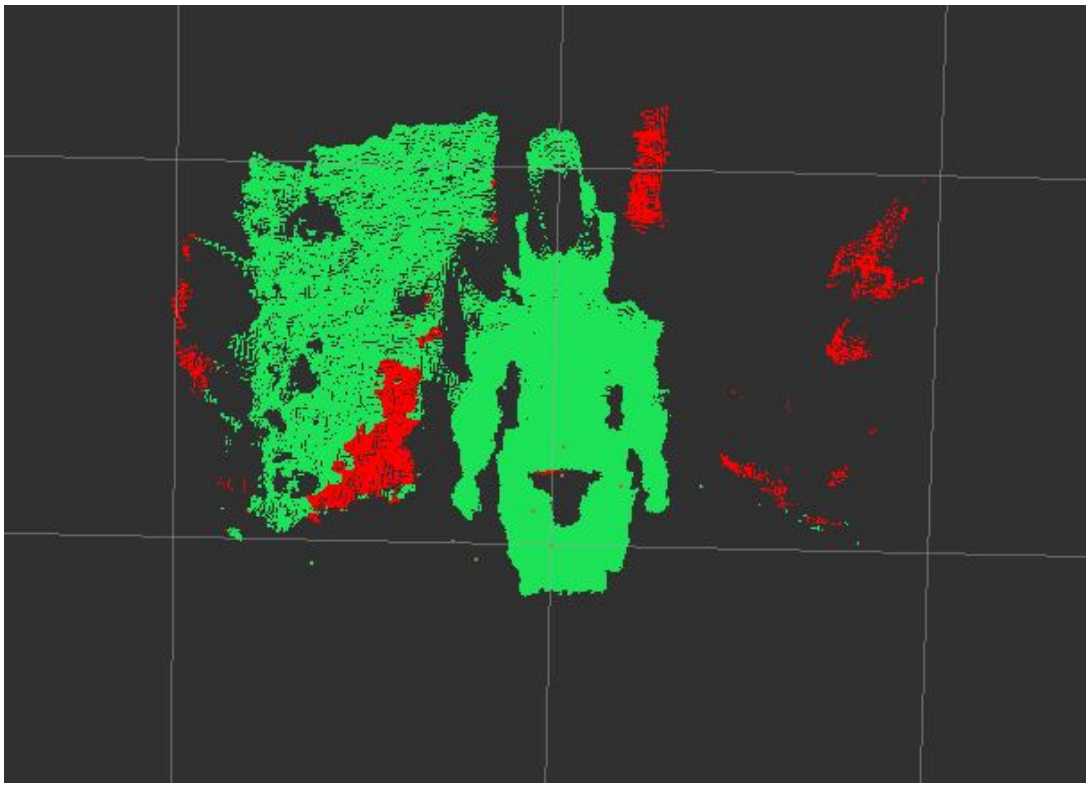

FIGURA 3.34: Segmetación por color

3.5.4. Resultado de aplicar segmentación con normales para obtener cilindros

Para llevar a cabo la segmentación por normales, lo primero que debemos hacer es instanciar la clase para métodos y modelos de muestra de consenso que requieren el uso de superficies normales para la estimación.

El filtro a crear debe ser del tipo NormalEstimation, el cual recibe por parámetro la nube de puntos de entada.

LISTING 3.12: Instanciación SAC

```
pcl::PassThrough<PointT> pass;
  pcl::NormalEstimation<PointT, pcl::Normal> ne;
  pcl::SACSegmentationFromNormals<PointT, pcl::Normal> seg;
  pcl::ExtractIndices<PointT> extract;
  pcl::ExtractIndices<pcl::Normal> extract_normals;
  pcl::search::KdTree<PointT>::Ptr tree (new pcl::search::KdTree<PointT> ());
```

Debemos reducir el ruido de la escena, en este caso realizamos un filtrado a lo largo de una dimensión específica,la coordenada z, con una distancia de 0.5 m.

Listing 3.13: Filtro passthrouh

```
pass.setInputCloud (cloud_pcl);
pass.setFilterFieldName ("z");
pass.setFilterLimits (0, 5);
pass.filter (*cloud_filtered);
std::cerr << "PointCloud after filtering has: " <<
  cloud_filtered->points.size () << " data points." << std::endl;
```

Realizamos la estimación de las normales de la escena, para diferenciar unas nubes de puntos de otros dependiendo del grado de los planos.

Para ello utilizamos el modelo de árbol de búsqueda y asignamos el número de vecinos que compondrán cada uno de los grupos de las nubes de puntos a diferenciar.

setSearchMethod() proporciona un puntero al objeto de búsqueda, el cual recibe por parámetro un tree que representa un grafo con estructura de árbol cuyos nodos se irán uniendo en función de los criterios de la segmentación.

setKSearch() asigna el número de K vecinos a usar para la estimación.

setNumberOfNeighbours() permite asignar el número de vecinos a la región.

Listing 3.14: Estimación de las normales

```
ne.setSearchMethod (tree);
ne.setInputCloud (cloud_filtered);
ne.setKSearch (50);
ne.compute (*cloud_normals);
```

En este filtro, usamos un estimador RANSAC, que como explicamos anteriormente, es un algoritmo iterativo que comprueba si los puntos de una nube se corresponden o pertenecen a un modelo geométrico definido (plano, esfera, cilindro, etc). Lo empleamos para

obtener los coeficientes del plano en este caso y posteriormente del cilindro e imponemos un umbral de distancia desde cada punto interno al modelo que no supere los 5 cm.

Además, establecemos la influencia normal de la superficie en un peso de 0.1 y limitamos la distancia al umbral del modelo en 0.1.

Si disminuimos este último valor hacemos que sea más complicado que los puntos se asemejen al modelo del plano, y por consiguiente serán menos puntos los que conformen el plano.

LISTING 3.15: Creación del objeto de segmentación para el modelo del plano y asignación de los parámetros

```
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_NORMAL_PLANE);
seg.setNormalDistanceWeight (0.1);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations (100);
seg.setDistanceThreshold (0.1);
seg.setInputCloud (cloud_filtered);
seg.setInputNormals (cloud_normals);
// Obtain the plane inliers and coefficients
seg.segment (*inliers_plane, *coefficients_plane);
```

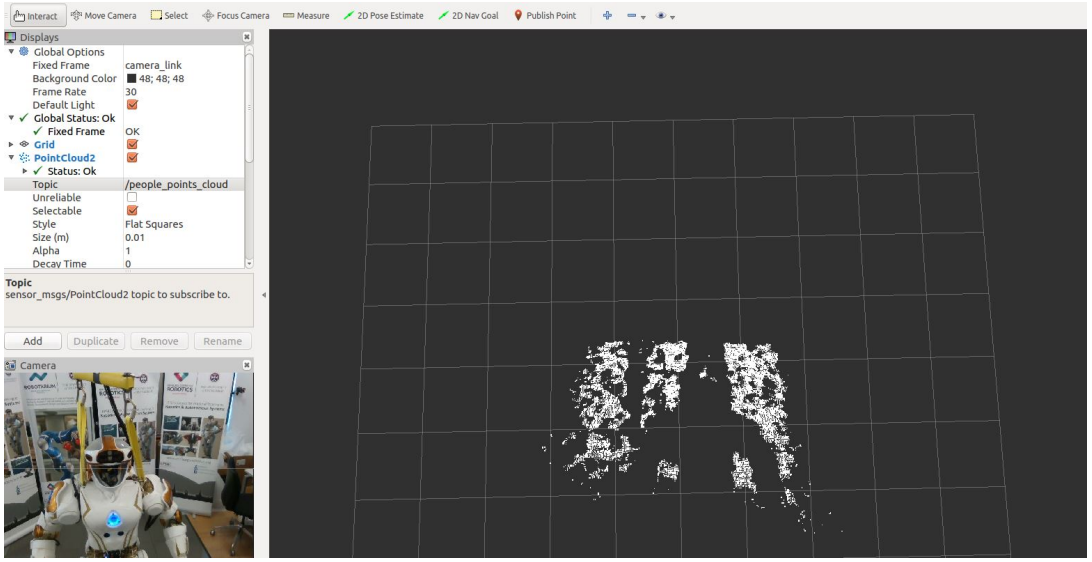

FIGURA 3.35: Formas planas en la escena

En la imagen anterior podemos comprobar como la segmentación del modelo del plano se realiza correctamente, vemos como la única nube de puntos existente es la del fondo de la escena, mientra que la figura del robot que no tiene partes planas no aparece en ella.

Una vez que hemos obtenido los conjuntos de puntos que cumplen los requisitos de pertenencia con el modelo del plano los extraemos de las escena.

Listing 3.16: Extracción de los inliers del plano de la nube de puntos de entrada

```
extract.setInputCloud (cloud_filtered);
extract.setIndices (inliers_plane);
extract.setNegative (false);
```

Guardamos los inliers del plano en el disco.

Listing 3.17: Escribimos en disco los inliers del plano

```
// Write the planar inliers to disk
pcl::PointCloud<PointT>::Ptr cloud_plane (new pcl::PointCloud<PointT> ());
extract.filter (*cloud_plane);
//std::cerr << "PointCloud representing the planar component: " <<
  cloud_plane->points.size () << " data points." << std::endl;
//writer.write ("table_scene_mug_stereo_textured_plane.pcd", *cloud_plane,
  false);
```

Una vez que hemos obtenido los conjuntos de puntos que cumplen los requisitos de pertenencia con el modelo del plano los extraemos de las escena.

Listing 3.18: Eliminamos los inliers del plano y extraemos el resto

```
extract.setNegative (true);
extract.filter (*cloud_filtered2);
extract_normals.setNegative (true);
extract_normals.setInputCloud (cloud_normals);
extract_normals.setIndices (inliers_plane);
extract_normals.filter (*cloud_normals2);
```

Creamos el objeto de segmentaxción para el modelo del cilindro y le asignamos los parámetros.

Establecemos la influencia normal en la superficie en un peso de 0.1.

Y establecemos el umbral del modelo en plano en 0.05.

Con un mayor peso conseguiríamos filtrar un modelo del cilindro de mayor tamaño.

LISTING 3.19: Creamos el objeto de segmentación para la segmentación del modelo del cilindro y asignamos los parámetros

```
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_CYLINDER);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setNormalDistanceWeight (0.1); //0.1
seg.setMaxIterations (10000);
seg.setDistanceThreshold (0.05); //0.05
seg.setRadiusLimits (0, 0.1); //0.1
seg.setInputCloud (cloud_filtered2);
seg.setInputNormals (cloud_normals2);
```

Obtenemos los inliers y coeficientes del modelo del cilindro.

LISTING 3.20: Obtenemos los inliers y coeficientes del modelo del cilindro

```
// Obtain the cylinder inliers and coefficients
seg.segment (*inliers_cylinder, *coefficients_cylinder);
//std::cerr << "Cylinder coefficients: " << *coefficients_cylinder <<
  std::endl;
```

Guardamos dichos inliers en disco.

LISTING 3.21: Escribimos los inliers del cilindro en disco

```
extract.setInputCloud (cloud_filtered2);
extract.setIndices (inliers_cylinder);
extract.setNegative (false);
pcl::PointCloud<PointT>::Ptr cloud_cylinder (new pcl::PointCloud<PointT> ());
extract.filter (*cloud_cylinder);
```

En la siguiente imagen podemos ver las formas cilíndricas encontradas en el Valkirye por parte del algoritmo, esto tiene una gran importancia es la realización de nuestro trabajo.

Esto es debido a que un objetivo fundamental de dicha investigación es, a parte de la localización de dicho robo en el entorno, también la localización de sus partes móviles como pueden ser su brazos, para facilitar la interacción de estos con los objetos del entorno.

Debido a que los brazos del robot se aproximan a un modelo cilíndrico podemos utilizar el modelo del cilindro para segmentarlos del resto del cuerpo.

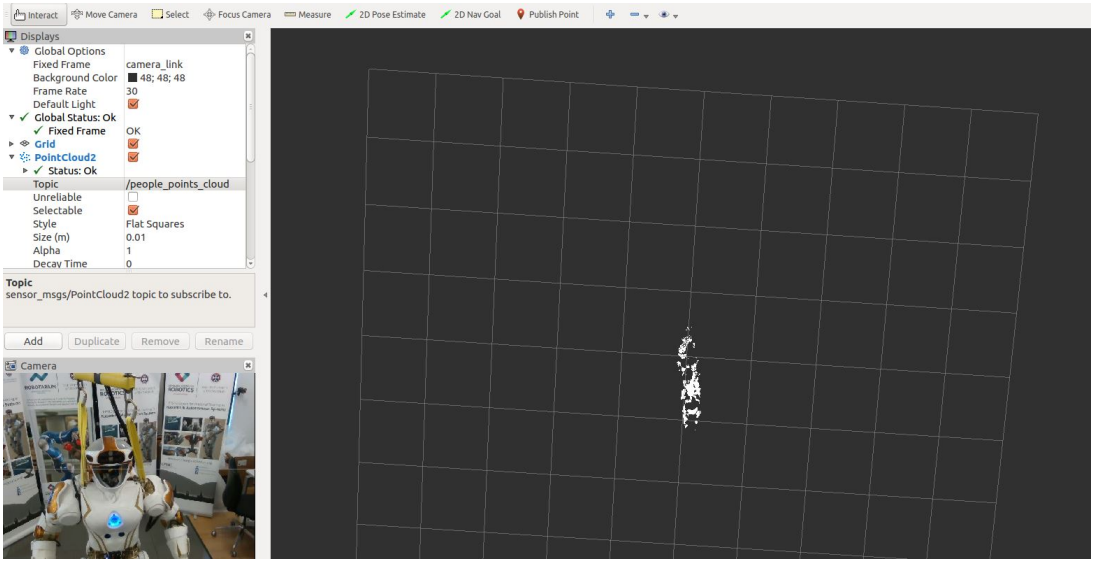

FIGURA 3.36: Formas cilíndricas en el Valkirye

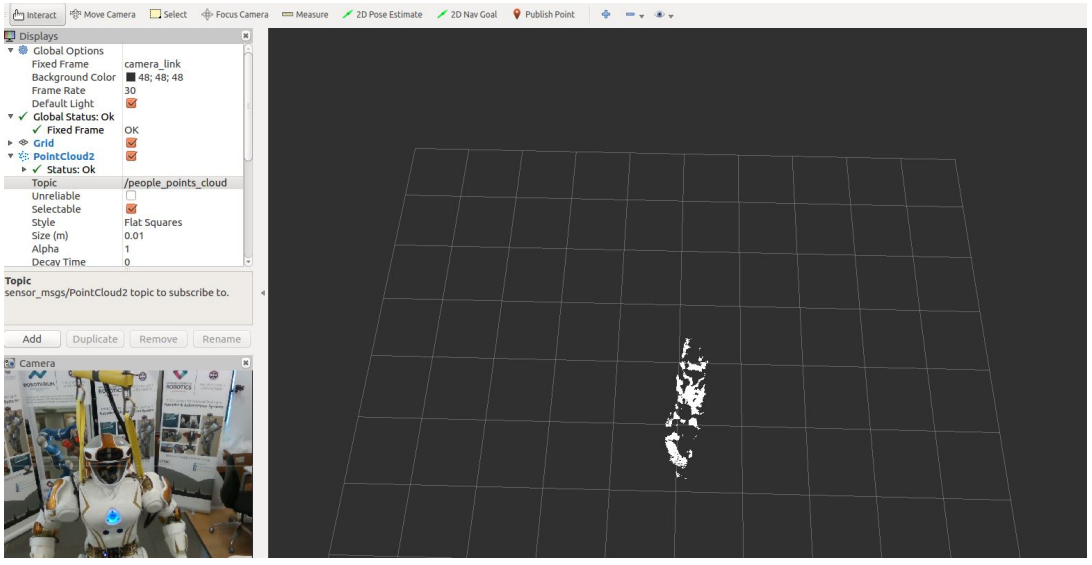

FIGURA 3.37: Formas cilíndricas en el Valkirye

Si aumentamos la tolerancia de los valores del tamaño del radio podemos aproximar modelos con un radio más grande, como la parte del torso del robot, que pese a no

tratarse de una figura cilíndrica perfecta puede aproximarse como se muestra en las siguientes imágenes.

Las formas ya no son tan precisas, se muestra desde una visión superior y sobre la transformada Z para mas claridad.

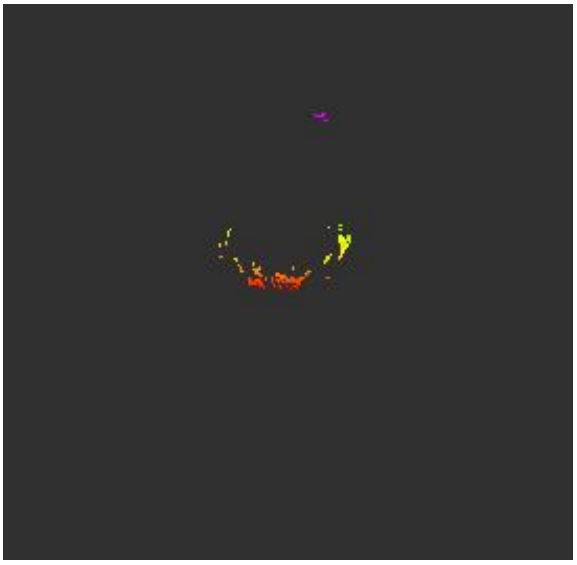

FIGURA 3.38: Formas cilíndricas en el Valkirye

3.5.5. Resultado de aplicar segmentación con normales

En este filtro el objetivo del algoritmo es extraer los planos de la nube de puntos capturada, trabajando con vecindades de puntos sobre los cuales se ajusta un plano para calcular la normal en cada punto de la nube. Luego, examinando las distancias entre los puntos y procediendo, en orden creciente de distancias, se evalúan las normales y según estas son similares y no muy separadas en distancias, se agrupan en segmentos.

LISTING 3.22: Escribimos los inliers del cilindro en disco

```
pcl::search::Search<pcl::PointXYZ>::Ptr tree =
  boost::shared_ptr<pcl::search::Search<pcl::PointXYZ> > (new
  pcl::search::KdTree<pcl::PointXYZ>);
pcl::PointCloud <pcl::Normal>::Ptr normals (new pcl::PointCloud
  <pcl::Normal>);
pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> normal_estimator;
 normal_estimator.setInputCloud (cloud_pcl);
normal_estimator.setKSearch (50);
```

```
normal_estimator.compute (*normals);

pcl::IndicesPtr indices (new std::vector <int>);
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setInputCloud (cloud_pcl);
pass.setFilterFieldName ("z");
pass.setFilterLimits (0.0, 1.0);
pass.filter (*indices);

pcl::RegionGrowing<pcl::PointXYZ, pcl::Normal> reg;
reg.setMinClusterSize (50);
reg.setMaxClusterSize (1000000);
reg.setSearchMethod (tree);
reg.setNumberOfNeighbours (30);
reg.setInputCloud (cloud_pcl);
////reg.setIndices (indices);
reg.setInputNormals (normals);
reg.setSmoothnessThreshold (3.0 / 180.0 * M_PI);
reg.setCurvatureThreshold (0.85);

std::vector <pcl::PointIndices> clusters;
reg.extract (clusters);
```

El filtro que crearemos será del tipo NormalEstimation, al cual le introduciremos la nube de puntos de entrada.

El método setSearchMethod() proporciona un puntero al objeto de búsqueda, en este caso le metemos por parámetro un tree que representa un grafo con estructura de árbol cuyos nodos se irán uniendo en función de los criterios de la segmentación.

El método setKSearch() asigna el número de K vecinos a usar para la estimación.

Se llevan a cabo varias pruebas para comprobar cual es el número de vecinos que mejor se adapta.

Un número de vecinos demasiado pequeño, como por ejemplo 10, no permite una diferenciación clara de las zonas por su diferencia de normales, como se muestra en la siguiente imagen.

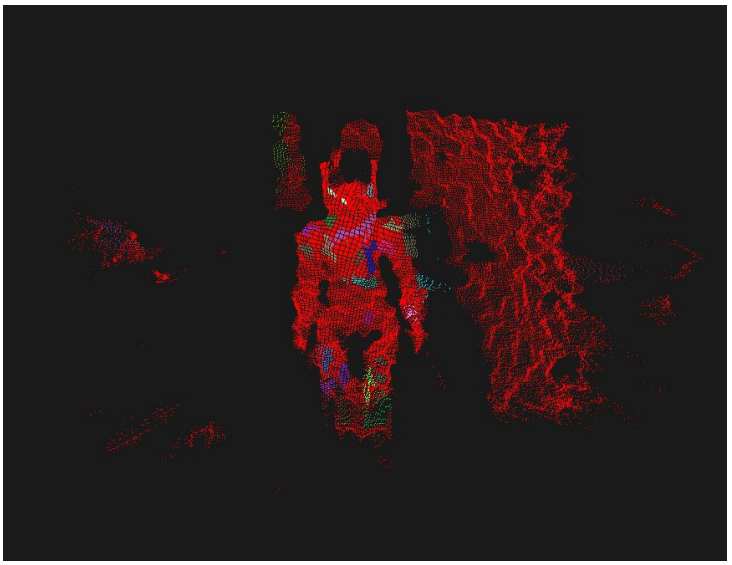

FIGURA 3.39: Nubes de puntos creadas en las segmentación

Si aumentamos el número de vecinos a 50, obtenemos una diferenciación más clara como vemos en la siguiente imagen.

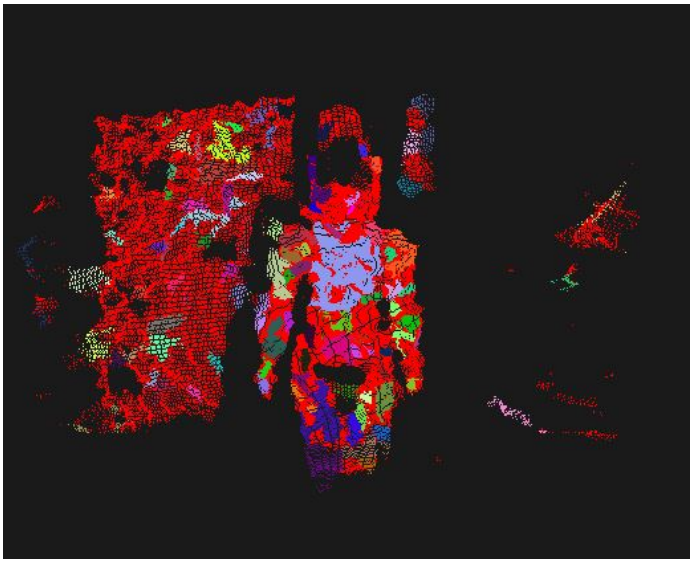

FIGURA 3.40: Segmetación por normales

```
Number of clusters is equal to 97
First cluster has 101 points.
These are the indices of the points of the initial
cloud that belong to the first cluster:
40108, 40136, 40137, 40138, 40139, 40140, 40141, 40142, 40143, 40144,
40197, 40198, 40199, 40200, 40201, 40237, 40238, 40239, 40240, 40241,
40242, 40243, 40244, 40245, 40246, 40247, 40248, 40249, 40250, 40251,
40252, 40253, 40254, 40255, 40256, 40257, 40258, 40259, 40260, 40261,
40262, 40263, 40264, 40265, 40266, 40267, 40268, 40269, 40270, 40271,
40272, 40273, 40274, 40275, 40276, 40277, 40278, 40279, 40280, 40281,
40282, 40283, 40284, 40285, 40286, 40287, 40289, 40290, 40291, 40292,
40293, 40294, 40295, 40296, 40297, 40298, 40299, 40300, 40301, 40302,
40303, 40304, 40305, 40306, 40307, 40308, 40309, 40310, 40311, 40312,
40313, 40314, 40315, 40316, 40317, 40318, 40319, 40320, 40321, 40331,
51775,
```

FIGURA 3.41: Índices de las nubes de puntos creadas en las segmentación

Como podemos ver se crean 97 cluster de puntos, para cada uno de las zonas con normales similares.

Se debe establecer también un umbral sobre el número máximo de puntos que compondrá la nube y otro con el valor mínimo.

La recomendación es establecer un valor mínimo lo más próximo a 1 y un valor máximo lo más grande posible.

En este caso se establecen los umbrales en 50 como valor mínimo y 1000000 como valor máximo.

El método setNumberOfNeighbours() permite asignar el número de vecinos a la región.

Con un número muy grande(1000) de kSearch, y un número muy pequeño de vecinos(10), se tiene una distinción muy clara entre objetos grandes, debido a que la media de la estimación de la normal se hace con muchos puntos, por lo que resulta muy general y solo los que difieren en mucho de esa normal son diferenciados.

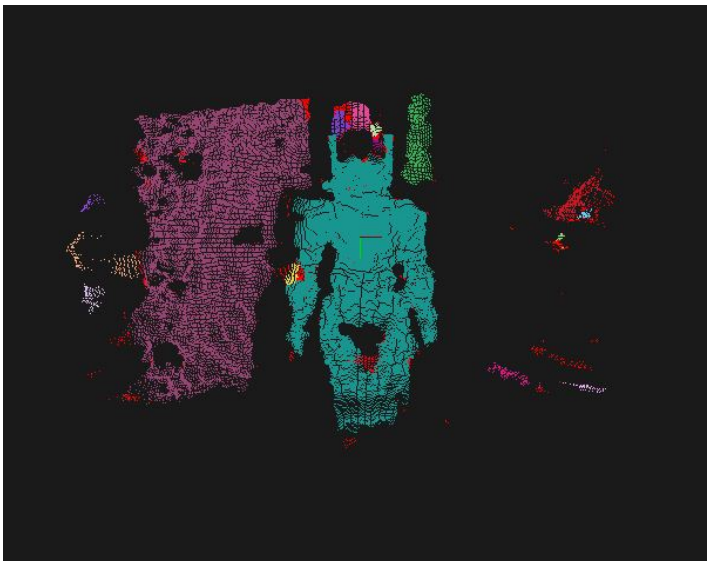

FIGURA 3.42: Segmetación por normales

Esto se puede comprobar en la imagen anterior, en la cual se crean 17 clusters de puntos, a diferencia de los 97 que se creaban en la prueba anterior, lo que indica la distinción más clara para objetos de gran tamaño.
La segmentación por este procedimiento resulta muy eficiente.

```
Number of clusters is equal to 17
First cluster has 55 points.
These are the indices of the points of the initial
cloud that belong to the first cluster:
3057, 3377, 3697, 4017, 4337, 4338, 4342, 4658, 4659, 4660,
4661, 4662, 4665, 4978, 4979, 4980, 4981, 4982, 4985, 5298,
5299, 5300, 5301, 5302, 5305, 5618, 5619, 5620, 5621, 5622,
5625, 5938, 5939, 5940, 5941, 5942, 5943, 5944, 5945, 6258,
6259, 6260, 6261, 6262, 6263, 6264, 6265, 6579, 6580, 6581,
6582, 6583, 6585, 6902, 6903,
```

FIGURA 3.43: Segmetación por normales

3.5.6. Resultado de aplicar segmentación por distancia

El filtrado por distancia consiste en filtrar la imagen de tal forma que dependiendo de unos límites de distancia podamos visualizar objetos que se encuentren en ese rango definido y obviar aquellos que se encuentren fuera de él.

En el filtrado por distancia definimos las nubes de puntos de entrada y de salida, las cuales serán del tipo PCLPointCloud2. Definimos un filtro de tipo PassThrough (es un filtro al cual se le define un parámetro y se le establecen unos límites por los cuales es

filtrado) al cual se le debe de introducir la nube de puntos de entrada (en realidad se le introduce el puntero a dicha nube(cloudPtr)). Se asignan tres campos a dicho filtro, estos serán los campos x, y, z que hacen referencia a dichas coordenadas. Para cada uno de ellos se declara una distancia mínima y máxima que hará referencia al rango visible. Se filtra y se asigna el resultado a la nube de puntos de salida (en este caso es igual que la entrada y es el puntero a dicha nube(cloudFilteredPtr)).

LISTING 3.23: Segmentación por distancia

```
ptfilter.setInputCloud (cloudPtr);
        ptfilter.setFilterFieldName ("z");
        min=float(person_data.usersData[j].centerOfMassWorld.z)-0.3;
        max=float(person_data.usersData[j].centerOfMassWorld.z)+0.5;
        ptfilter.setFilterLimits (min,max);

        ptfilter.filter (*cloud_filtered);
        ptfilter.setInputCloud (cloud_filteredPtr);

        ptfilter.setFilterFieldName ("x");
        min=float(person_data.usersData[j].centerOfMassWorld.x)-0.25;
        max=float(person_data.usersData[j].centerOfMassWorld.x)+0.25;
        ptfilter.setFilterLimits (min,max);

        ptfilter.filter (*cloud_filtered);
        ptfilter.setInputCloud (cloud_filteredPtr);

        ptfilter.setFilterFieldName ("y");
        min=float(person_data.usersData[0].centerOfMassWorld.y)-0.7;
        max=float(person_data.usersData[0].centerOfMassWorld.y)+0.7;
        ptfilter.setFilterLimits (min,max);

        ptfilter.filter (*cloud_filtered);
```

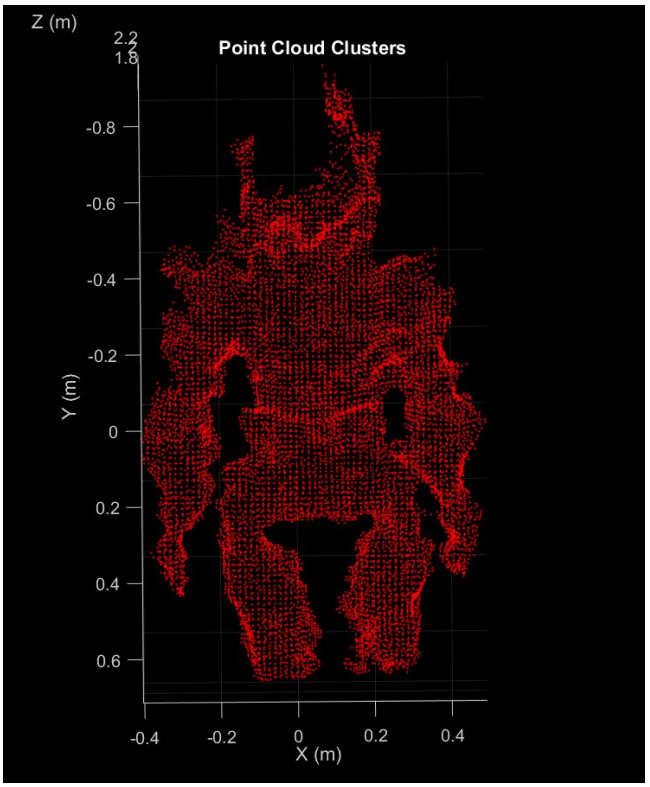

FIGURA 3.44: Segmetación por distancia

Como podemos observar en la imagen, realizamos una segmnetación con unos rangos de distancia que nos permiten quedarnos solamente con el cluster del robot. El nivel de similitud con la figura real es muy alto en esta segmentación, debido a que no hay otros objetos que interfieran con la nube de puntos del robot por unos rangos de distancia determinados.

3.5.7. Resultado de aplicar Euclidian Cluster Extration

Para la realización de este algoritmo es necesaria en primera instancia la segmentación de los planos, que, como ya la hemos explicado anteriormente no vamos a entrar en ella.

En el supuesto de que ya se haya realizado la segmentación de los planos correctammente debemos crear el objeto KdTree para el método de búsqueda y extracción además de añadir los parámetros.

Dichos parámetros como se muestra en el código posterior sirven para determinar la tolerancia de la nube de puntos en el espacio euclidiano, asignar un tamaño mínimo y un tamaño máximo de la nube.

Listing 3.24: Escribimos los inliers del cilindro en disco

```
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
  pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud (cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (0.02);
ec.setMinClusterSize (100);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);
```

Listing 3.25: Escribimos los inliers del cilindro en disco

```
int j = 0;
 pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
  pcl::PointCloud<pcl::PointXYZ>);
for (std::vector<pcl::PointIndices>::const_iterator it =
  cluster_indices.begin (); it != cluster_indices.end (); ++it)
{

  for (std::vector<int>::const_iterator pit = it->indices.begin (); pit !=
  it->indices.end (); ++pit)
    cloud_cluster->points.push_back (cloud_filtered->points[*pit]); //*
  cloud_cluster->width = cloud_cluster->points.size ();
  cloud_cluster->height = 1;
  cloud_cluster->is_dense = true;

 std::cout << "PointCloud representing the Cluster: " <<
  cloud_cluster->points.size () << " data points." << std::endl;
  //std::stringstream ss;
 //ss << "cloud_cluster_" << j << ".pcd";
  //writer.write<pcl::PointXYZ> (ss.str (), *cloud_cluster, false); //*
  j++;
}
```

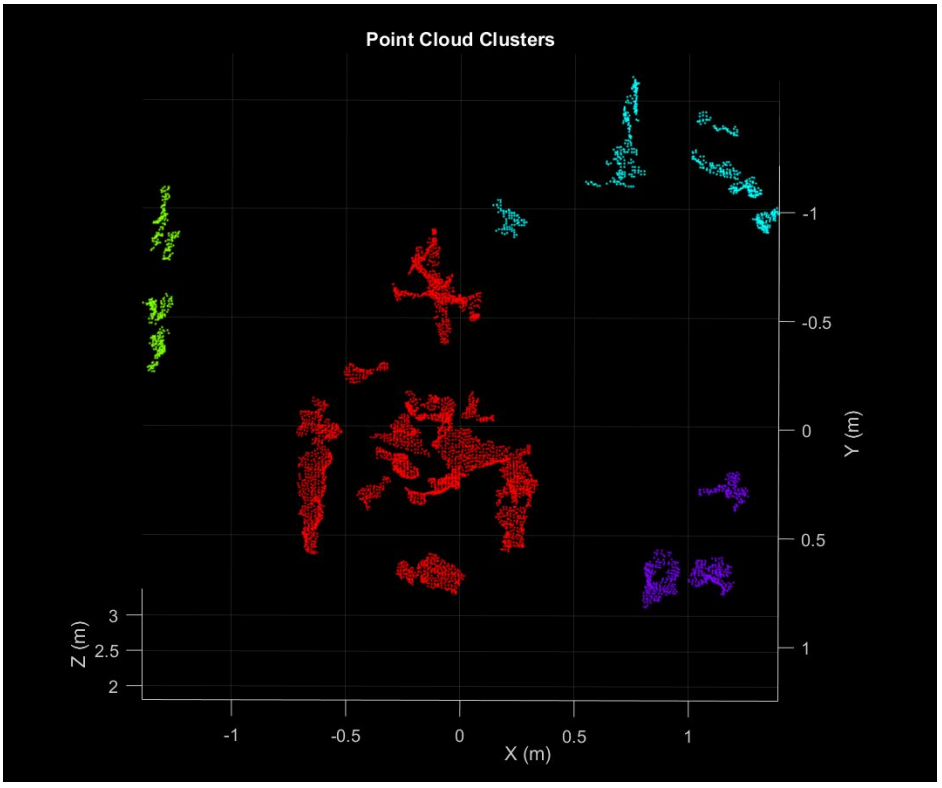

FIGURA 3.45: Vista de frente

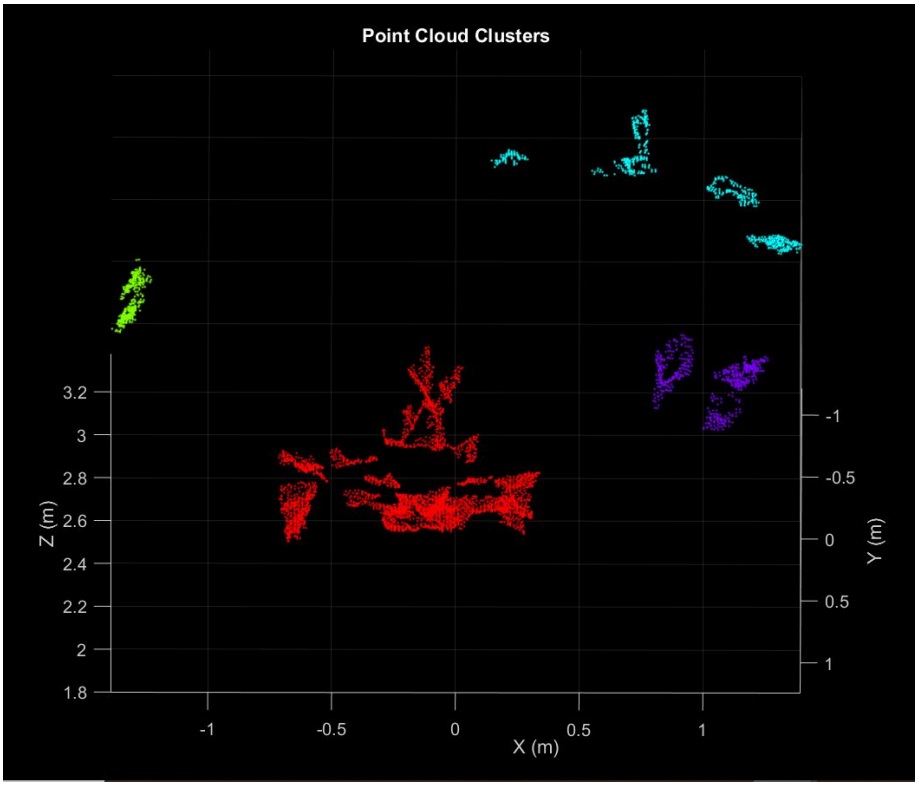

FIGURA 3.46: Vista superior

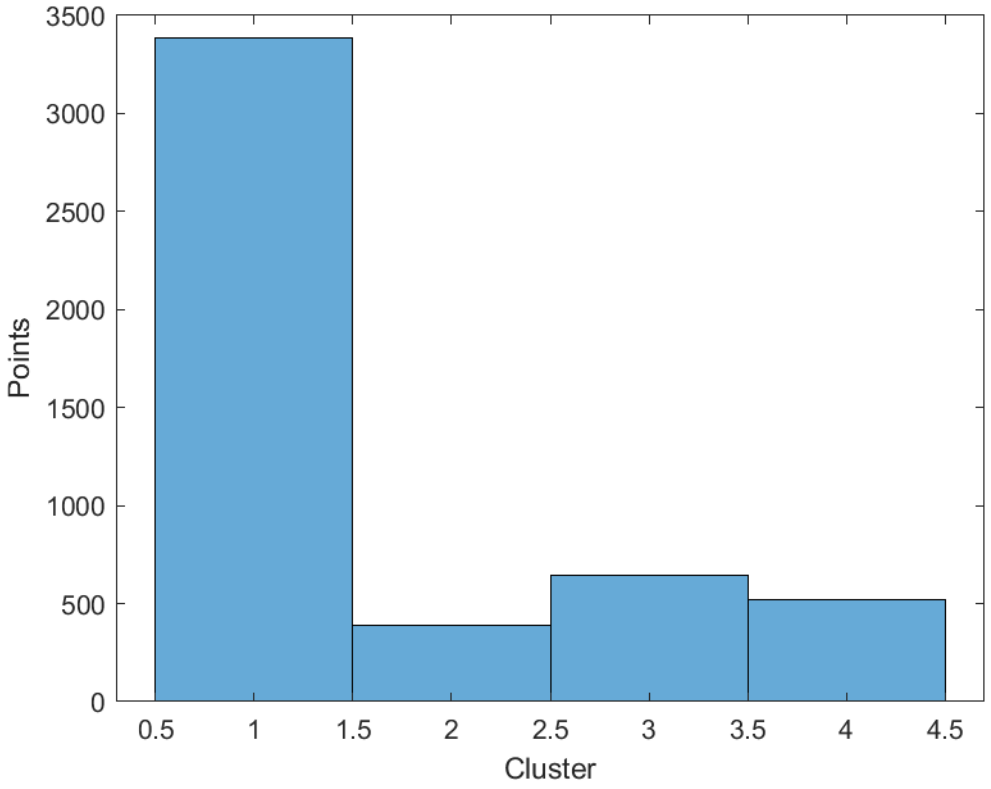

FIGURA 3.47: Conjuntos de nubes de puntos después de la segmentación

Como se puede ver en las imágenes, y el en gráfico se identifica la mayor nube de puntos como el robot así como nubes de puntos más pequeñas correspondientes a algunas partes de las paredes.

Capítulo 4

Conclusiones

Este Trabajo de Fin de Grado ha permitido conocer en profundidad el sistema operativo robótico (*ROS*) como herramienta para la implementación de multitud de proyectos en el ámbito de los robots así como las diversas posibilidades que ofrece para el desarrollo software y las ventajas que tiene respecto a otras tecnologías similares.

Así como la librería PCL y su amplia funcionalidad en el procesamiento de nubes de puntos y procesamiento de geometría 3D.

Las principales dificultades en el inicio fueron familiarizarse con el entorno ROS y el lenguaje C++ de la librería PCL.

Se cumplen los siguientes objetivos:

- Se consigue extraer información del entorno como las paredes, por medio de la extracción de los planos.
- Los algoritmos usados para identificar las formas del robot se han basado en métodos de búsqueda y modelos matemáticos RANSAC y Region Growing.
- El modelo de segmentación con normales para obtener cilindros ha sido utilizado para la segmentación de los brazos del robot por su semejanza de estos con dicha forma geométrica.

No se cumple el objetivo:

- Reconocimiento de los movimientos del robot.

No se ha llevado a cabo la implementación de dicho objetivo pero el trabajo realizado es el primer paso para la realización de dicha tarea.

Tabla 4.1: Algoritmos de segmentación

Algoritmo	Método de búsqueda	Algoritmo de cálculo
Bísqueda con árbol octal	Octree	X
Segmentación con colores	tree	Region growing
Segmentación con normales para obtener cilindros	Tree	RANSAC
Segmentación con normales	Tree	region growing
Segmentación por distancia	Rango Z	X
Euclidian cluster extration	Kdtree	Euclidian cluster extration

Diferentes algoritmos pueden ser usados para los mismos fines, pero en algunos de ellos se encuentran ventajas o desventajas dependiendo de la nube de puntos.

Como en el caso de la segmentación con Kdtree, el cual necesita de una cantidad determinada de normales para realizar la segmentación y en algunas determinadas nubes de puntos sin paredes o suelos puede producir fallos.

También se ha comprobado la viabilidad de Matlab como herramienta de apoyo para la representación de nubes de puntos así como su posible uso para llevar a cabo procesos de segmentación de forma independiente.

Este proyecto ha tenido un enfoque en el ámbito de la robótica pero hay una amplia gama de campos en los que la visión por computador tendrá gran relevancia en el futuro y en nuestra vida cotidiana.

Bibliografía

[1] Fischler y Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". En: *Tesis* (1981). URL: http://dx.doi.org/10.1145/358669.358692.

[2] Pablo Flores. "Algoritmo RANSAC, fundamento teórico". En: *Tesis* (2011). URL: http://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2011/keypoints/FundamentoRANSAC.pdf.

[3] Derek Hoiem. "Representation and techniques for 3D object recognition and scene interpretation". En: *Tesis* (2011). URL: https://pdfs.semanticscholar.org/e456/5133627bfba6afad49bf2be311944be477bc.pdf.

[4] *Intel Realsense Camera ZR300*. Intel. 2017. URL: https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/ZR300-Product-Datasheet-Public.pdf.

[5] Marc Levoy. "The Use of Points as a Display Primitive". En: *University of North Carolina* (1985). URL: https://graphics.stanford.edu/papers/points/point-with-scanned-figs.pdf.

[6] *Point Cloud Library*. URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/.

[7] Lawrence Gilman Roberts. "Machine Perception of Three-Dimensional Solids". En: *Massachusetts Insitute od Technology* (1961). URL: https://dspace.mit.edu/bitstream/handle/1721.1/11589/33959125-MIT.pdf?sequence=2.

[8] *Robot Operating System*. URL: http://wiki.ros.org/ROS/Tutorials.

[9] Charles R.Qi. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". En: *Stanford University* (2017). URL: https://arxiv.org/pdf/1612.00593.pdf.

[10] Andrés Felipe Carlo Salcedo. "Procesamiento de nubes de puntos por medio de la librería PCL". En: *Tesis* (2012). URL: https://dialnet.unirioja.es/servlet/articulo?codigo=4271775.

[11] Amaia Santiago. "Sistema de detección de objetos mediante cámaras. Aplicación en espacios inteligentes". En: *Tesis* (2007). URL: http://www.geintra-uah.org/system/files/private/Proyecto.pdf.

[12] Juan Toribios. "Modelado 3D de objetos a partir de la Kinect". En: *Tesis* (2012). URL: https://core.ac.uk/download/pdf/29404176.pdf.

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR

Universidad de Alcalá