DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

PROYECTO FIN DE GRADO

ANÁLISIS Y RECONOCIMIENTO DE OBJETOS MEDIANTE TECNOLOGÍA TIME OF FLIGHT

Autor: Borja Martín Valverde

Director: Luis Moreno Lorente

Tutor: Jorge García Bueno

LEGANÉS, MADRID

JUNIO 2012

"La sabiduría consiste en saber cuál es el siguiente paso;
la virtud, en llevarlo a cabo."

David Starr **Jordan, 1931.**

Hay tantas personas a las que agradecer que hoy me encuentre escribiendo este apartado de agradecimientos, que no se muy bien por donde empezar.

Primeramente debo dar las gracias a mi ***familia*** *por facilitarme las cosas para realizar esta carrera y por el apoyo que me han dado durante estos cuatro años tan largos. En especial a mi* ***Madre*** *que siempre me ha dado ciertos consejos que me han venido muy bien en ciertos momentos, a mi* ***Hermana*** *y a mi* ***Padre*** *por apoyar lo que hago y animarme para continuar.*

También quiero dar las gracias a ***Alejandro*** *por ese empujón inicial que necesitaba para realizar este proyecto, a* ***Rober*** *por estar ahí realizando el proyecto compartiendo Kinect y a* ***Santi*** *por la ayuda ofrecida.*

A mi tutor ***Jorge*** *darle las gracias por confiar en mí cuando era necesario y por darme consejos para realizar este proyecto.*

Y dar las gracias a ***SARA*** *por aguantarme durante estos años de carrera que en ocasiones parecían ser demasiado largos, y por darme su apoyo incondicional y a mi segunda* ***familia*** *por entender que hiciera estas cosas tan raras.*

GRACIAS *a todos por estar ahí en los momentos buenos y en los momentos más duros.*

Índice general

Lista de Figuras

Resumen

El objetivo que persigue este proyecto es asentar las bases para el reconocimiento de objetos y posterior manipulación de estos por parte del robot Manfred. Obteniendo primeramente la segmentación del entorno real en 3 dimensiones, mediante algoritmos sobre la nube de puntos, que representa los datos extraídos del sensor Kinect de Microsoft en el entorno de visualización que nos brindan las bibliotecas de PCL(Point Cloud Library).

Hemos realizado la segmentación del entorno en subconjuntos o subunidades pertenecientes a la nube total de los datos extraídos, pasaremos a su clasificación, discriminando por tamaño y forma determinaremos si el subconjunto pertenece a un plano o a un objeto clasificándolos de esta forma.

En este proyecto se muestra el gran potencial de las librerías PCL, no solo para los temas de los que trata este proyecto sino para temas relacionados con los trabajos futuros que se pueden desarrollar a partir de este proyecto y muchas otras aplicaciones de gran interés para la investigación en visión artificial.

Palabras clave:

robótica, OpenCV, PCL, visión artificial, segmentación, clasificación, detección objetos.

Abstract

The objective of this project is to lay the foundations for object recognition and subsequent handling of these by the robot Manfred, first we obtain segmentation in real environment with 3 dimensions, we use algorithms on the point cloud that representing the data extracted from sensor Microsoft's Kinect in PCL (Point Cloud Library).

We've performed segmentation of the environment in subunits or subgroups belonging to the total cloud, where we extracted the data, we turn to their classification, discriminating by size and shape that will determine if the subset belongs to a plane or an object and we classify them this way.

This project shows the great potential of the PCL's libraries , not only for the issues addressed in this project, also for issues related to future work that can be developed from this project and many other applications of high interest for research in artificial vision.

Keywords:

robotics, OpenCV, PCL, computer vision, segmentation, clasification, objects detection.

Abreviaturas

PCL **P**oint **C**loud **L**ibrary

OpenCV **O**pen **C**omputer **V**ision

OpenNI **O**pen **N**atural **I**nteraction

GPU **G**raphics **P**rocessing **U**nit

CPU **C**ontrol **P**rocess **U**nit

RGB **R**ed **G**reen **B**lue

HSV **H**ue **S**aturation **V**alue

ROS **R**obot **O**perative **S**ystem

OS **O**perative **S**ystem

TOF **T**ime **O**f **F**light

VFH **V**ery **F**ast **H**istogram

GNU **G**enuinely **N**ot **U**nix

KinFu **Kin**ect **Fu**sion

VTK **V**isualization **T**ool**K**it

DOF **D**egrees **O**f **F**ree

RANSAC **R**andom **S**ample **C**onsensus

Capítulo 1

Introducción

,

1.1. Historia de la robótica

El término $''robot''$ proviene del la expresión Checa $''robota''$ que significa servidumbre, fué asignado por primera vez en un libro llamado **"Robots Universales de Rossum"** publicado en 1921 por **Karel Capek**. En este libro los robots eran máquinas con inteligencia limitada diseñadas para realizar los trabajos más duros. Hoy en día este término ha evolucionado hasta tener como significado de robot, cualquier sistema autónomo capaz de realizar un trabajo por sí solo y reaccionar a estímulos externos.

La Robótica es un campo multidisciplinar, ya que es un nexo de unión entre diferentes disciplinas, por ejemplo en la ingeniería en sus ramas de la mecánica, electrónica, eléctrica, control, informática, telemática, etc..., centrándose en la percepción, manipulación, inteligencia artificial o incluso medicina, que es uno de los campos en los que se está avanzando mucho.

La robótica se podría definir como el arte de percibir el entorno con sensores y tomar decisiones mediante un ordenador o máquina de manera autónoma (parcialmente o totalmente).

Podríamos decir que el objetivo que tiene la existencia de robots, es hacernos la vida más fácil a los seres humanos pero también mejorar la calidad de esta.

Los robots se pueden clasificar según sus características:

- Eje del brazo: rectangular, polar o cilíndrico.
- Espacio de trabajo: secuencia limitada, punto a punto o continuo.
- Locomoción y cinemática: Base fija, con ruedas, con piernas...
- Tipo de control: distribuido o centralizado.
- Tipo de alimentación: eléctricos, neumáticos, hidráulicos.
- Tamaño: nano, pequeña, grande o enorme
- Tipo y número de articulaciones: lineal o rotativo
- Tarea que realiza: industrial, doméstico, servicios médicos, militares...
- Tipo de movimiento: movimiento de giro, conjunta-interpolación, en línea recta o interpolación circular.

1.2. Percepción en robótica

Desde antes del inicio de la robótica propiamente dicha, el ser humano ha intentado crear sistemas que reproduzcan algunas de las funciones que puede realizar el propio ser humano. Este fué el motivo por el que se crearon autómatas (no en el sentido en el que lo solemos conocer ahora), en el sentido de la palabra que significa máquina capaz de imitar la figura y los movimientos de un ser animado. Esto era así dado el anhelo que sentía el ser humano por conocerse a sí mismo, morfológicamente y funcionalmente, pero también para delegar el trabajo en estas máquinas.

Al intentar que una máquina reproduzca nuestros movimientos o nuestro comportamiento, es decir, que actuen como nosotros, en cierto modo, vemos la necesidad de otorgarle los mismos sentidos para que puedan realizar los mismo actos. La percepción en robótica se basa en esto, en otorgar, analizando y estudiando cada una de las posibilidades, algo parecido a los sentidos humanos, que nos permiten realizar una retroalimentación con el entorno que nos

rodea en cuanto a lo que a información se refiere. A esto se le llama percepción, y dado que en el ser humano la visión supone más de un 75 % de la información que recibe el cerebro del entorno, podriamos decir que la visión artificial es el método de percepción del cual se puede extraer más información.

Es curioso el hecho de que las máquinas de hoy en día sean capaces de procesar información mucho más rápido que el cerebro humano y sin embargo los sistemas de visión artificial se encuentran muy lejos de ser como la visión biológica.

Algunas de las tareas que puede realizar un robot por medio de la visión artificial serían:

- Reconocimiento de objetos: para evitar obstáculos estáticos y móviles o para seguimiento o manipulación de los mismos.
- Construcción de modelos: de objetos y lugares para reconocerlos y localizarlos, o como mapeado del entorno.
- Caracterizar su propia posición y movimiento y la de los objetos que le rodean.

1.3. Objetivos

En esta sección explicaremos los objetivos del proyecto, dando un breve repaso a lo que luego iremos introducciendo en este documento:

Lo primero que haremos será dar una base teórica de los precedentes en visión artificial y sobre lo que viene siendo la parte del estado del arte en visión artificial, con la base de los algoritmos utilizados en la programación de la aplicación y la base técnica tanto del software como del hardware utilizado, dando justificaciones acerca de la elección de estas herramientas.

Después nos centraremos en explicar la plataforma utilizada, es decir, el sistema operativo, lenguaje de programación y herramientas sobre las que se sientan las bases de toda la programación implicada en la realización del proyecto.

Figura 1.1: Robot Q.bo Modelo-T de la empresa española TheCorpora y desarrollado por Francisco Paz, es un ejemplo de percepción en robótica no solo por las cámaras que tiene incorporadas, sino por los 2 micrófonos omnidireccionales y 1 unidireccional, sus sensores de proximidad que le otorgan capacidad de interacccionar con el entorno que le rodea, moverse en él y actuar sobre él con sus actuadores

Más tarde realizaremos un breve repaso a los algoritmo que hemos utilizado, y a la implementación de estos en la aplicación, asi como el funcionamiento de la misma. Y daremos una idea de como hemos realizado los experimentos y pruebas que luego nos han hecho tomar decisiones alternativas para cumplir los objetivos.

Capítulo 2

Aspectos teóricos

Comenzaremos este capítulo con una introducción sobre la visión artificial convencional, con la definición de imagen bidimensional, los espacios de color utilizados en dichas imágenes, y daremos un breve repaso a algunos de los métodos de análisis que se utilizan en dichas imágenes.

En contraposición explicaremos que es una nube de puntos, el software que se utiliza habitualmente para la adquisición de datos por parte del dispositivo PrimeSense (en nuestro caso es el sensor Kinect de Microsoft), explicaremos también que es PCL (Point Cloud Library) y por último daremos una explicación teórica de los algoritmos utilizados en este proyecto.

2.1. Visión artificial en 2D

Antes de que sucediera esta gran revolución, no tanto en la tecnología, que ya existía, sino en el precio de los sensores que permiten la adquicisión de datos en 3 dimensiones, los algoritmos estaban pensados para analizar, procesar, umbralizar y segmentar imágenes en 2 dimensiones. En este apartado explicaremos algunos de los algoritmos más utilizados en este campo.

2.1.1. ¿Qué es una imagen?

Una imagen digital se podría definir como una matriz M x N cuya mínima expresión sería el pixel, cada uno de estos píxeles tienen asociados 3 valores comprendidos entre 0 y 255 que

representan el color (será distinto dependiendo del espacio de color con el que trabajemos). En la siguiente expresión se muestra el modelo que representa una imagen:

$$f(x,y) = \begin{pmatrix} f(0,0) & f(0,1) & \cdots & f(0,N-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N-1) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0) & f(M-1,1) & \cdots & f(M-1,N-1) \end{pmatrix}$$

Donde cada elemento $f(x,y)$ representa 1 pixel, el cual contiene información de color para conformar entre todos ellos una imagen. Esta información puede ser diferente dependiendo del espacio de color en el cual estemos trabajando.

2.1.2. Espacios de color

Como ya hemos dicho los píxeles tienen valores asociados para cada elemento $f(x,y)$, que contiene la información referente al color de dicho elemento o pixel que representa una muestra discreta del color del entorno real. Existen distintos criterios para definir el color con estos 3 valores numéricos que son los canales de color, algunos de estos criterios son los que se muestran a continuación.

2.1.2.1. Espacio de color RGB

Este criterio se basa en el concepto de que todos los colores se pueden obtener por mezcla de los colores rojo, verde y azul, de ahí RGB (Red Green Blue), dando distintos pesos a cada color. Este modelo es cartesiano y se compone de 8 vértices que corresponden al color Rojo (1,0,0), Verde (0,1,0), Azul (0,0,1), Negro (0,0,0), Blanco (1,1,1) y el resto son mezcla de estos anteriores, como se muestra en la figura 2.1.

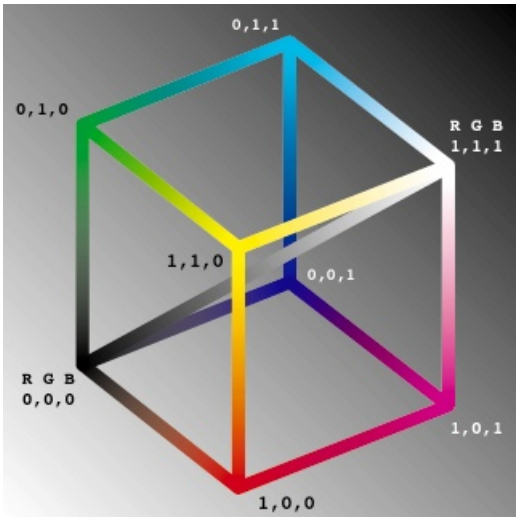


Figura 2.1: Espacio de color RGB

Nota: las componentes (R,G,B) cuando se trabaja en el ordenador comprende los valores entre 0 y 255 aunque en el modelo lo indiquemos en el rango entre 0 y 1.

Este espacio de color es muy estable, pero en ocasiones es demasiado simple para realizar ciertas operaciones de segmentación o clasificación de objetos y por ello se utilizan otros espacios de color.

2.1.2.2. Espacio de color HSV

Este espacio de color está compuesto de 3 parámetros que nos sirven para diferenciar los distintos colores, que son H (Hue) es el matiz del color, cada uno de los 3 colores Rojo, Verde y Azul dispone de 120º siendo el máximo valor de Hue 360º, S (Saturation) en la saturación del color y V (Value) es el valor de intensidad. Este espacio de color es más robusto que el anterior modelo, pero tiene el incoveniente de que es inestable para los colores cercanos al blanco o al negro, es decir, es inestable para los grises. Se muestra una imagen de este modelo en la Figura 2.2.

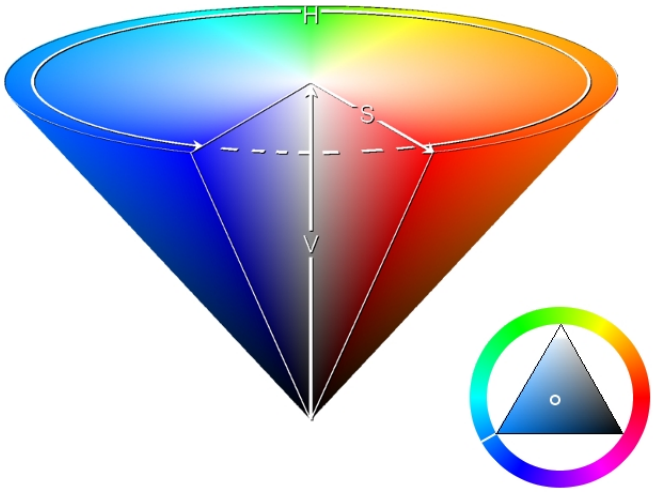
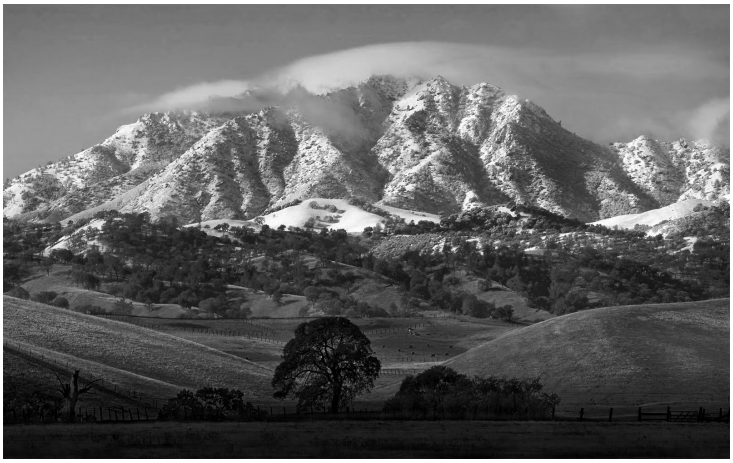


Figura 2.2: Espacio de color HSV

2.1.2.3. Escala de grises

Este modelo es una variante simplificada a un canal de color del espacio de color RGB, en este modelo los tres colores Rojo, Verde y Azul tienen los mismos pesos, es decir, cada color tiene un peso de 1/3 y además los tres tienen el mismo valor, que va desde 0 hasta 255, de esta forma se aumenta la velocidad de cualquier algoritmo que realicemos sobre la imagen. No obstante hay que tener en cuenta que al utilizar este modelo estamos asumiendo una pérdida significativa de información y que esta pérdida de información puede ser crucial para realizar ciertos algoritmos.

(a) Imagen RGB



(b) Imagen en escala de grises

Figura 2.3: Pérdida de información por cambio en espacio de color

2.1.3. Análisis de imágenes digitales

Para el ser humano no supone apenas esfuerzo el acto de reconocer y clasificar objetos con el fin o no de manipularlos, y solo con ver estos objetos sabemos como tenemos que colocar la mano para poder agarrarlos. Pero para una máquina este acto, que el ser humano realiza de manera prácticamente inconsciente, no es fácil, y se necesitan realizar una serie de algoritmos relativamente complejos para conseguir algo parecido a lo que realizamos los seres humanos solo que de manera más simple y con más restricciones. Cabe destacar que este hecho es curioso ya que la velocidad de cómputo de una máquina es mucho mayor que la velocidad de cómputo de un ser humano.

En cuanto a la parte de reconocimiento y clasificación de objetos, al igual que el ser humano, que obtiene gran parte de la información por medio del sentido de la vista, se realiza con visión artificial.

Una vez hemos tomado los datos de la imagen hay que realizar una serie de transformaciones para poder extraer la información de manera fiable y óptima por parte de un ordenador. Estas operaciones siguen un orden determinado para conseguir optimizar este proceso:

- **Preprocesamiento.**

  En el cual se realizan tareas o transformaciones como el filtrado de la imagen para eleminación de ruido o para resaltar ciertas características. Explicaremos algunos de los algoritmos que se utilizan con esta finalidad.

  Modificación del histograma: Un histograma es una gráfica que muestra el número de píxeles que tienen el mismo valor, estos valores están ordenados desde 0 hasta 255. Lo más habitual es realizar una umbralización a partir de un valor determinado.

$$f_{filtrada}(x,y) = \left\{ \begin{array}{ll} V & \text{si } f(x_n, y_n) \geq \text{T} \\ 0 & \text{si } f(x_n, y_n) < \text{T} \end{array} \right\}$$

  También puede darse la operación dual:

$$f_{filtrada}(x,y) = \left\{ \begin{array}{ll} V & \text{si } f(x_n, y_n) < \text{T} \\ 255 & \text{si } f(x_n, y_n) \geq \text{T} \end{array} \right\}$$

Ó bien una umbralización bilateral que engloba los dos anteriores que sería el más completo y en teoría con el que deberiamos eliminar más ruido si los valores de umbralización son los adecuados:

$$f_{filtrada}(x,y) = \left\{ \begin{array}{ll} 255 & \text{si } f(x_n, y_n) > T_2 \\ V & \text{si } T_1 \leq f(x_n, y_n) \leq T_2 \\ 0 & \text{si } f(x_n, y_n) < T_1 \end{array} \right\}$$

Donde $f_{filtrada}(x,y)$ sería la imagen después del proceso de filtrado o operación que se realice en el preprocesamiento, $f(x_n, y_n)$ es el valor del enésimo pixel, V es el valor que tenía el pixel en la imagen original y T es el valor de umbralización.

Esta umbralización se basa en mantener el valor que tenía el pixel según unas condiciones de umbralización, si no cumple la condición de umbralización diremos que es ruido y debe ser eliminado de alguna forma.

A continuación se muestra la imagen del histograma de una imagen:

Figura 2.4: Ejemplo del histograma de una imagen

Eliminación de ruido: Cabe destacar en esta parte, el filtro lineal de la Media (Ecuación 2.1) en el cual lo que hacemos es pasar una máscara 3 x 3 que sume todos los valores de color de los píxeles por los que pasamos la máscara y los divida entre el número de ellos,

en ocasiones con este filtro no es posible realizar un filtrado satisfatorio o mejor dicho fructífero como sucede con el ruido impulsional, para la eliminación de este ruido solo nos queda utilizar la Mediana que lo que hace es poner el valor que este en el medio en una sucesión en la que tenga el mismo número de valores por encima que por debajo, es evidente que este filtro no es lineal, en cuanto al ruido Gaussiano también conocido como desenfoque Gaussiano utilizaremos un filtro Gaussiano.

Media:

$$\bar{a} = \frac{1}{n} \sum_{k=0}^{k=n} a_k \tag{2.1}$$

Mediana:

$$Mediana = x_i 1 + \frac{(N_m/2) - N_{i-1}}{f_i}(x_{i2} - x_{i1}) \tag{2.2}$$

Filtro Gaussiano(Máscara):

$$w(x,y) = \frac{e^{\frac{-(x^2+y^2)}{4\sigma^2}}}{\sum_{s=-a}^{a} \sum_{s=-b}^{b} e^{\frac{-(s^2+t^2)}{4\sigma^2}}} \tag{2.3}$$

Extracción de características.

Se podría considerar a este paso como una preparación de la imagen, o al menos como un estudio previo de la imagen ya filtrada, para su posterior segmentación. Cuando hablamos de características nos referimos por ejemplo al nivel de Gris, o Color, o incluso a la posición que tienen los distintos elementos en la imagen, para realizar una umbralización y posible binarización de la imagen, con el fin de acotar el problema y por tanto simplificarlo.

En la imagen 2.5 se muestra una representación del espacio de características de una imagen, los ejes X, Y y Z representan 3 características, y los puntos representan los píxeles de la imagen.

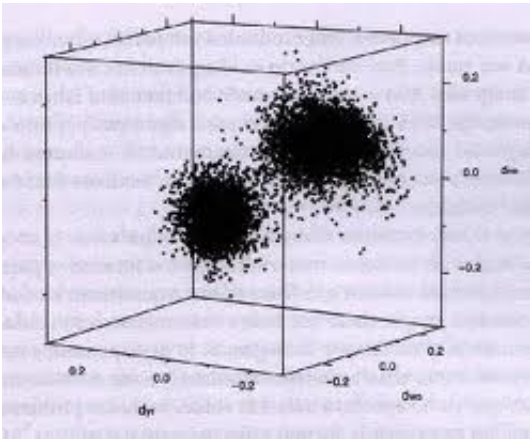


Figura 2.5: Gráfica representativa de las características

Es la persona que está programando la que decide que características definen mejor a un objeto determinado, y que diferencian unos objetos de otros de la manera más eficiente posible. Para tomar la decisión a la hora de elegir dichas características, que luego serán determinantes, el programador se suele apoya en una serie de estudios previos que realiza con otros algoritmos y funciones, en este apartado explicaremos algunos de ellos.

Uno de los problemas más frecuentes que nos encontramos a la hora de analizar una imagen es saber donde empieza y acaba un objeto, para obtener esta información solemos recurrir a una serie de algoritmos que resaltan los cambios bruscos en el color, la textura o incluso la forma.

- Detección de bordes. Existen varias formas de detectar o resaltar el borde de un objeto, por medio de operadores que hacen de máscara, de tal forma que vamos pasando dicha máscara por todos y cada uno de los píxeles:

  Gradiente, la expresión matemática del gradiente es la siguiente:

$$\nabla f(x,y) = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} & \frac{\partial f(x,y)}{\partial y} \end{pmatrix}$$

En el caso de las imágenes digitales este operador se simplifica y queda de la siguiente forma:

$$\nabla f(x,y) = \left(\frac{\Delta f(x,y)}{\Delta x} \quad \frac{\Delta f(x,y)}{\Delta y}\right)$$

La máscara más sencilla en el caso del operador Gradiente sería:

-1	1

-1
1

Existen otras máscaras para el operador gradiente del tipo 3 x 3, pero la idea es la misma, que es restar de alguna forma los píxeles para observar en cual de las restas se produce un cambio brusco. Hay que tener cuidado ya que si restamos un valor demasiado elevado a un valor suficientemente bajo puede producirse valores de borde que sean negativos o por el contrario que sean mayores de 255 y esto el ordenador no lo puede mostrar por medio de una imagen, por lo que se suelen utilizar otros métodos, que normalizan de alguna forma la solución o evitan directamente este problema. Debemos decir que a este problema se le suma otro ya que este operador es muy sensible al ruido.

Detector de bordes Canny, utiliza la derivada de una gaussiana ya que es un operador optimizado y umbraliza con 2 valores los bordes de tal forma que queden solon los bordes reales.

- Puntos clave o Keypoints.

  Uno de los algoritmos más importantes que se utilizan en esta parte es un algoritmo que te encuentra puntos de interes o Keypoints en la imagen, como esquinas u otros. En la figura 2.6 se muestra un ejemplo de este proceso.

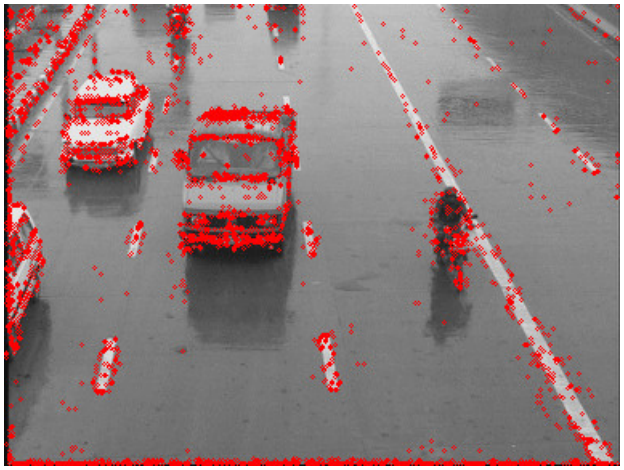


Figura 2.6: Keypoints

Lógicamente antes de llegar a esto hemos tenido que filtrar el proceso.

- **Segmentación.**

  Entramos ya en la parte del proceso en la cual desarrollamos algoritmos, que deben servir para separar en subgrupos los elementos de la imagen preprocesada y con la extracción de características realizada. Por ejemplo, para separar objetos de lo que llamamos fondo de nuestra imagen y los propios objetos entre sí. Solemos basarnos en condiciones más o menos restrictivas para considerar que un elemento reúne las características antes extraídas para ser un candidato a pertenecer a un grupo determinado.

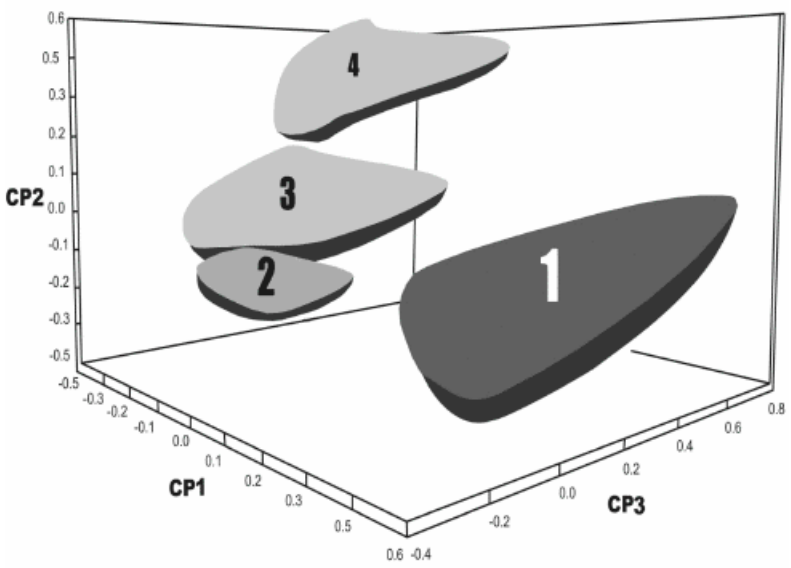


Figura 2.7: Segmentación de los píxeles en grupos

Las segmentaciones más habituales se realizan por color ya que es la característica más descriptiva de un objeto en las imágenes digitales, lo que se hace es binarizar la imagen a partir de estas características, por ejemplo, lo que sea rojo en la imagen pasa a ser blanco y lo que no sea rojo pasa a ser negro.

$$f_{transform}(x,y) = \left\{ \begin{array}{ll} 1 & \text{si } f(x_n, y_n) \geq \text{T} \\ 0 & \text{si } f(x_n, y_n) < \text{T} \end{array} \right\}$$

Siendo $f_{transform}(x,y)$ la imagen binarizada, T es el umbral elegido (en este caso seria el umbral que consideramos como rojo), y $f(x_n, y_n)$ sería el pixel que estamos estudiando. El valor de $f_{transform}(x,y) = 1$ representa el color blanco, y $f_{transform}(x,y) = 0$ el color negro. Hay que tener en cuenta que para una imagen a color tendríamos un umbral para cada componente de color, en el caso del espacio de color RGB tendríamos un umbral para el componente R (Red), otro para el G (Green) y otro para el B (B), con el espacio de color HSV seguiriamos el mismo procedimiento teniendo en cuenta como se definen los colores en él.

(a) Imagen digital

(b) Imagen binarizada por color

Figura 2.8: Binarización de imagen por color rojo

Pero aunque para los seres humanos sea fácil definir que es rojo y que no lo es, con gran exactitud y prácticamente para cualquier variación en la iluminación, para una máquina no es tan sencillo por eso dependiendo del objeto o sistema a reconocer elegiremos un espacio de color u otro (RGB o HSV por ejemplo).

Existen otros métodos para segmentar como por ejemplo el crecimiento de regiones, que

utiliza puntos que se dispersan por la imagen inicialmente de manera aleatoria, son los llamados puntos semilla, después por vecindad va creciendo por las regiones uniformes hasta que llega a un sitio que ya no es igual a esta región. Los resultados de este algoritmo son siempre distintos aunque lo hagamos en la misma imagen, ya que los puntos iniciales son aleatorios.

Otro algoritmo parecido es el de Split and Merge (Separar y Unir), consiste en dividir la imagen horizontal y verticalmente por la mitad es decir en 4 trozos del mismo tamaño, si 2 de las regiones tienen las mismas características se unen, y se vuelven a dividir cada uno de estos trozos en otros 4, y así sucesivamente, el número de iteraciones las elige el usuario, y dependiendo de las iteraciones se obtendrán mejores o peores resultados.

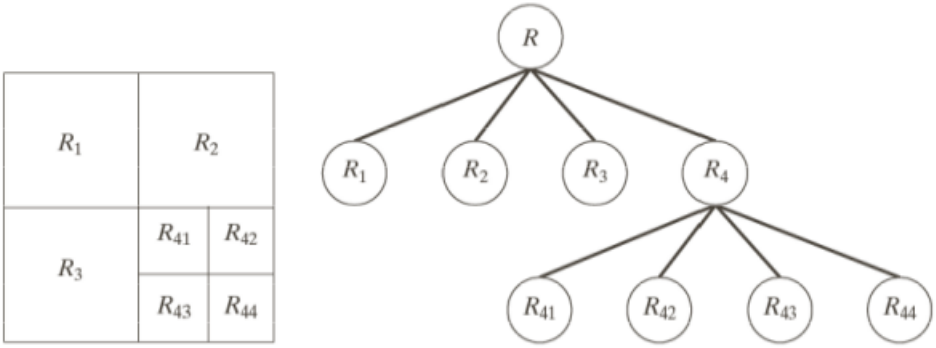


Figura 2.9: Método Split Merge

- **Transformaciones morfológicas.**

Llegados a este punto lo que pretendemos es apoyarnos en instrumentos o herramientas que nos simplifiquen la búsqueda para el próximo proceso. Se suelen utilizar tranformaciones como:

Dilatación-erosión: La dilatación consiste en hacer crecer regiones que sean de color blanco o similar, y la erosión es la operación inversa, es decir, la degradación de las regiones que sean de color negro o similar, estos algoritmos cobran gran importancia también fil-

trado una vez hemos obtenido una imagen binarizada. En la figura 2.10 se muestra un ejemplo de dilatación de una imagen.

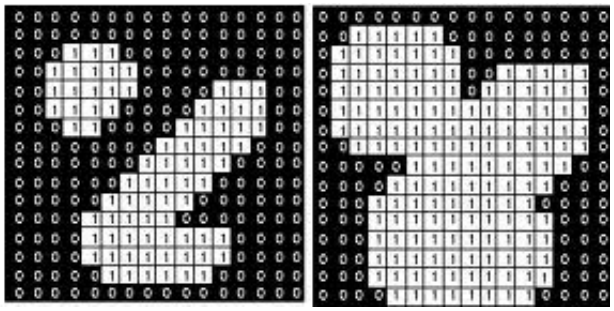


Figura 2.10: Dilatación de una imagen binarizada

Esqueletización: Consiste en obtener la mínima expresión del objeto sin llegar a eliminarlo, un tipo de esqueletización es MAT (Medial Axis Transformation) se dice que si inscribimos un círculo en el objeto y este toca por dos lados su borde, el centro de dicho círculo pertenece al esqueleto de este.

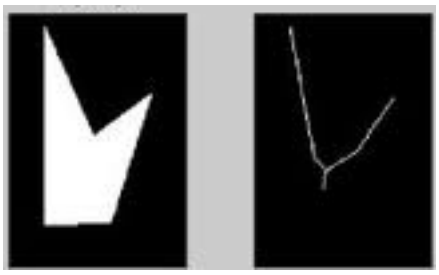


Figura 2.11: Esqueletización de una imagen binarizada

Cerco convexo (Convex hull): Lo que pretende este algoritmo es inscribir el objeto en una figura cuyos ángulos siempre sean menores de 180º, de tal forma que esta figura en la cual queremos inscribir nuestro objeto es un polígono convexo.

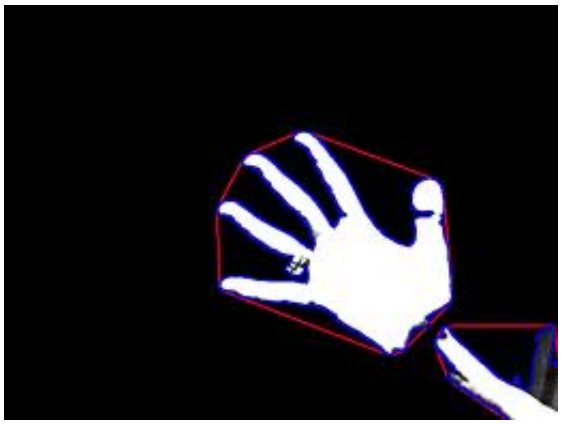


Figura 2.12: Cerco convexo en una imagen binarizada

Nota: El cerco convexo se muestra en color rojo.

- **Descripción.**

La descripción sería como el segundo obstáculo que debemos superar en cuanto a consideraciones de diferenciación en grupos de elementos u objetos. En este proceso suele realizarse un estudio del sistema, que suele ser eurísticos y recopilatorios. Por medio de estos métodos inductivos conseguimos realizar modelados de los elementos u objetos que nos encontramos en el entorno, estos modelos son los que llamamos descriptores. Un ejemplo de descriptores en dos dimensiones serían los lagos y bahías provenientes del convex hull, otro descriptor que merece la pena nombrar es el denominado compacticidad que se refiere a la relación entre el perímetro del objeto y su área de la forma:

$$\frac{P^2}{A}$$

es muy habitual utilizar este descriptor como característica de una región, también se usan la posición, orientación y otro tipo de descriptores referentes a la forma del objeto como la signatura/firma que es el resultado de realizar una función basada en el giro desde el

centro geométrico con respecto al borde del objeto e ir midiendo la distancia entre el centro y el borde en función del giro, o los descriptores de Fourier que se basan en los armónicos, explicándolo un poco más en profundidad, consiste en ir tomando los valores del borde en sentido antihorario, a cada uno de los puntos se transforman a números complejos para después realizarles la transformada de Fourier de la siguiente forma:

$$h(k) = x_k + y_k j \tag{2.4}$$

$$H(\omega) = \frac{1}{n} \sum_{k=0}^{k=n-1} h(k) e^{\frac{-2\omega\pi kj}{n}} \tag{2.5}$$

Los coeficientes para cada una de las frecuencias es decir los ármonicos son los descriptores de Fourier. Si análizamos los primeros ármonicos de la función que describe el borde del objeto veremos que la forma es muy parecida a la real y más parecida según aumentamos el número de ármonicos o descriptores de Fourier.

Reconocimiento o clasificación.

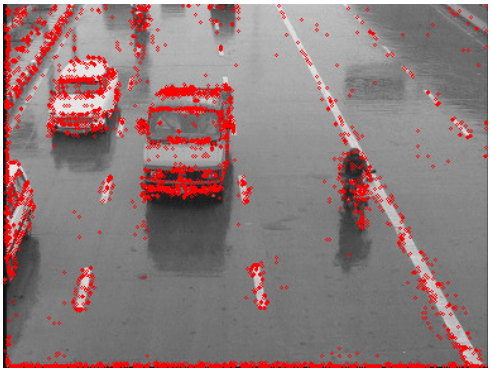
Es la última etapa de todo análisis de imágenes ya sea en dos o en tres dimensiones, y es el objetivo final que se persigue cuando realizamos el estudios en visión artificial. En esta parte lo que se pretende es realizar el reconocimiento de patrones dados por la fase anterior y así saber a ciencia cierta, de que objeto u elemento se trata. Un ejemplo de clasificador sería el clasificador Bayesiano en el cual lo que se hace es introducirle los datos en una fase de entrenamiento y es en la segunda fase, que es la de test, en la que se reconocen los patrones. Este clasificador se basa en el teorema de Bayes, con la diferencia de que se le añaden algunas condiciones o hipótesis que simplifican dicho proceso, en su ejecución, entrenamiento y fase de prueba encontramos grandes parecidos con otro método de clasificación que se basa en redes neuronales. El clasificador bayesiano se basa en el teorema de Bayes que es un método probabilístico de pertenencia a un grupo en el espacio de características que se ha definido previamente y dice así:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{2.6}$$

Donde $P(B|A)$ es la probabilidad de que el objeto de vector de características A pertenezca a la clase o grupo B y se conoce como probabilidad a posteriori ya que es el dato que queremos obtener, $P(A|B)$ es la probabilidad de que en la clase o grupo B se dé un vector de características A, $P(A)$ es la probabilidad de que el vector de características sea A y $P(B)$ es la probabilidad de que un grupo del espacio de características sea el grupo B, a estas dos últimas probabilidades se les denomina probabilidades a priori, ya que son los datos que obtenemos directamente del espacio de características y son los datos que utilizaremos para calcular $P(B|A)$.

Otro de los algoritmos más usados hasta ahora en visión artificial es SIFT (Scale Invariant Feature Transform), utiliza los puntos de interés o keypoints que hemos descrito anteriormente. SIFT es un algoritmo que se usa para detectar y describir las características locales de las imágenes. Las aplicaciones de este método incluyen el reconocimiento de objetos, mapeado y navegación en robótica, modelado en 3D, reconocimiento de gestos, seguimiento en vídeo, etc.

El algoritmo primero encuentra los puntos de interés en la imagen y estos puntos nos proporcionan una "descripción"del objeto (descriptor). Esta descripción, se puede utilizar para identificar un objeto en una imagen que contenga muchos otros objetos. Para realizar el reconocimiento fiable, es importante que las características extraídas de la imagen sean detectables incluso para cambios en la escala, aumento del ruido o cambios en la iluminación.



(a) Detección de Keypoints

(b) Reconocimiento de los puntos de interés en nueva imagen

(c) Resultado del reconocimiento

Figura 2.13: Pasos del algoritmo SIFT

2.1.4. OpenCV

Se puede decir que OpenCV (Open Computer Vision) es una herramienta con múltiples bibliotecas y funciones para tratamiento de imágenes en 2 dimensiones. Es multiplataforma, ya que es compatible con GNU/Linux, Microsoft Windows y Apple Mac OS. Su código es totalmente abierto y libre para su utilización comercial y para fines de investigación.

OpenCV contiene funciones preestablecidas capaces de realizar reconocimiento de objetos y reconocimiento facial. Es una herramienta muy potente pero el hecho de que este pensado para imágenes en 2 dimensiones, ha hecho que no utilicemos esta herramienta directamente, aunque hay que decir que en muchas de las bibliotecas de la herramienta que hemos usado (PCL) están basadas en una modificación y adaptación a 3 dimensiones de las bibliotecas de OpenCV.

Toda la informacion sobre OpenCV puede consultarse en su página[1].

2.2. Estado del arte en Visión Artifial

2.2.1. ¿Qué es una Nube de Puntos?

Podemos definir una nube de puntos como una matriz tridimensional cuya mínima expresión es un punto (véase la diferencia con el píxel), cada punto puede contener información relativa al color, ya sea RGB, HSV o escala de grises, las coordenadas del punto representan su posición en el entorno real en X, Y y Z. Sin embargo, por las características en cuanto a la adquisición de los datos por parte del dispositivo Primesense, a efectos prácticos, podemos simplificar la definición de nube de puntos, como una malla bidimensional compuesta de puntos, la cual se encuentra deformada en profundidad, de tal forma que para un valor de X e Y solo le corresponde un valor de Z (profundidad). Esta simplificación no puede atribuirse a una nube de puntos cuando estemos hablando de mapeado, ya que en este caso por el método progresivo de adquisición de datos, para un mismo valor de X e Y si que pueden existir varios puntos cuya única diferencia en posición sería la profundidad (lo mismo puede aplicarse a todos los pares de ejes de coordenadas).

[1] http://opencv.willowgarage.com

En nuestro caso en particular, una nube de puntos no es más que una representación que se realiza por medio de las bibliotecas de PCL de los datos que se obtienen del sensor de infrarrojos y de la cámara RGB, de tal forma que cada uno de estos datos tiene integrados datos relativos a la localización espacial en las coordenadas X, Y y Z, entendiendo que el eje de coordenadas es la propia cámara, y de color, ya sea RGB, HSV o en escala de grises, y que se visualizan en un entorno que utiliza internamente VTK y OpenGL para la interacción mediante eventos con dichos datos por parte del usuario. Entendiendo esto, podemos afirmar que el concepto de resolución en la imagen ha cambiado con respecto a las imágenes convencionales bidimensionales, ya que se ha tenido que equilibrar dos resoluciones, la de RGB y la del sensor de infrarrojos que es inferior, pudiendo obtenerse un máximo de 307200 puntos.

Si tenemos encuenta la simplificación que hemos propuesto antes, la forma de la función que define una nube de puntos sería la siguiente:

$$f(x,y,z) = \begin{pmatrix} f(0,0,Z_{(0,0)}) & f(0,1,Z_{(0,1)}) & \cdots & f(0,N-1,Z_{(0,N-1)}) \\ f(1,0,Z_{(1,0)}) & f(1,1,Z_{(1,1)}) & \cdots & f(1,N-1,Z_{(1,N-1)}) \\ \vdots & \vdots & \ddots & \vdots \\ f(M-1,0,Z_{(M-1,0)}) & f(M-1,1,Z_{(M-1,1)}) & \cdots & f(M-1,N-1,Z_{(M-1,N-1)}) \end{pmatrix}$$

2.2.2. Primesensor versus Time Of Flight

Como ya hemos adelantado anteriormente Microsoft ha revolucionado en estos últimos años el campo de la visión artificial con el sensor, pensado en un principio para la consola XBox 360, Kinect, gracias a la reducción en el precio de esta tecnología. Esta reducción en el precio hace que los desarrolladores que antes usaban cámaras Time Of Flight se decanten a comprar estas cámaras 3D con tecnología similar. No obstante estas nuevas cámaras tienen un error significativamente superior a las cámaras TOF. En la figura 2.14 se muestran las distintas tecnologías que se han venido utilizando hasta el momento para la detección de mapas de profundidad, para distintas aplicaciones.

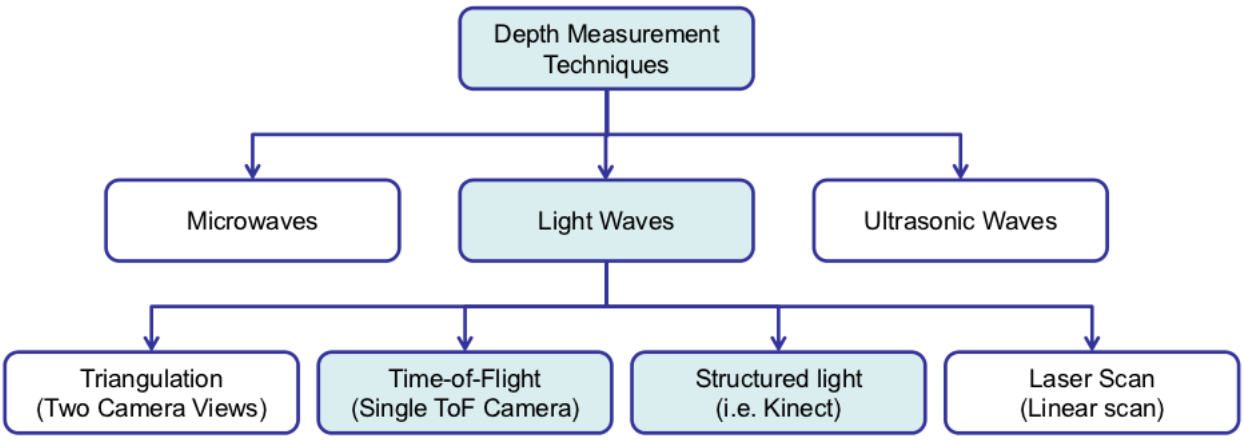


Figura 2.14: Técnicas de medición de profundidad

2.2.2.1. Time Of Flight

Las cámaras con tecnología Time Of Flight se basan en la extracción del entorno tridimensional por medio del proceso emitir un haz de luz, normalmente infrarroja, y cuantificar el tiempo que tarda en regresar ese haz de luz al receptor de dicho dispositivo, de hay Time Of Flight (Tiempo de vuelo), este tiempo que cuantificamos se podría decir que es el tiempo que está volando el haz de luz desde su emisión hasta su recepción. Este sistema consigue perfilar los objetos de la escena al tomar varias medidas consecutivas. El error en la localización en posición de un punto de un objeto puede llegar a ser relativamente pequeño, del orden de milímetros, dependiendo del dispositivo TOF que utilicemos.

En la figura 2.15 se muestra un esquema con las diferentes partes que componen el sistema TOF y su funcionamiento.

En el caso de estar tomando medidas consecutivas (que es lo habitual), es más eficiente medir el desfase entre la señal emitida y la señal recibida, como hemos señalado en la figura 2.15. En este caso las ecuaciones que utilizaremos para medir dicho desfase serían las siguientes:

Señal emitida:

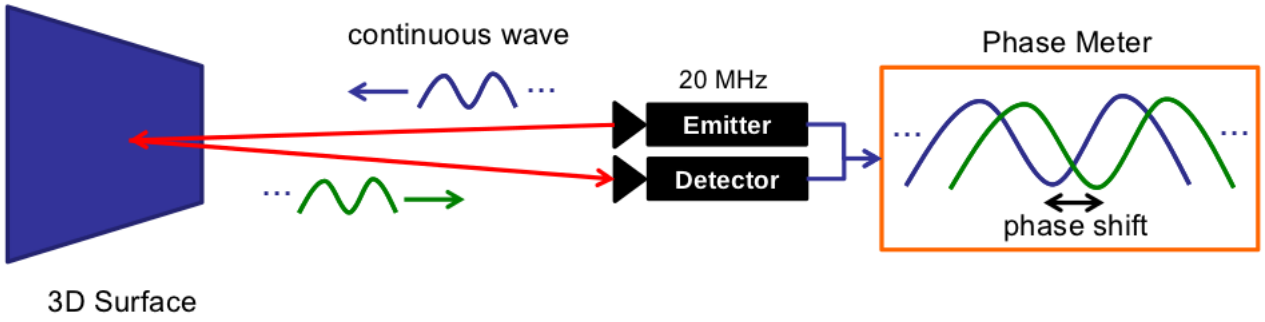


Figura 2.15: Esquema de funcionamiento de un sistema TOF

$$g(t) = \cos(\omega t) \tag{2.7}$$

Señal recibida tras rebotar en una superficie de la escena:

$$s(t) = b + a\cos(\omega t + \phi) \tag{2.8}$$

Correlación cruzada de las dos señales:

$$c(\tau) = s \otimes g = \int_{-\infty}^{\infty} s(t)g(t+\tau)\mathrm{d}t \tag{2.9}$$

Siendo:

ω = frecuencia de modulación.

b = constante bias.

a = amplitud.

ϕ = desfase entre la señal de entrada y de salida.

τ = offset.

La función de correlación cruzada se simplifica a:

$$c(\tau) = \frac{a}{2}\cos(\omega t + \phi) + b \tag{2.10}$$

Tenemos que cada punto detectado por el sensor se refleja en los objetos cuatro veces por período en intervalos desde A_0 a A_3. Estos cuatro valores son suficientes para recuperar la señal sinusoidal. El desfase (ϕ) entre la luz emitida y la señal recibida es por tanto:

$$\phi = \frac{A_3 - A_1}{A_2 - A_0}; \quad A_i = (c.\frac{\pi}{2}) \tag{2.11}$$

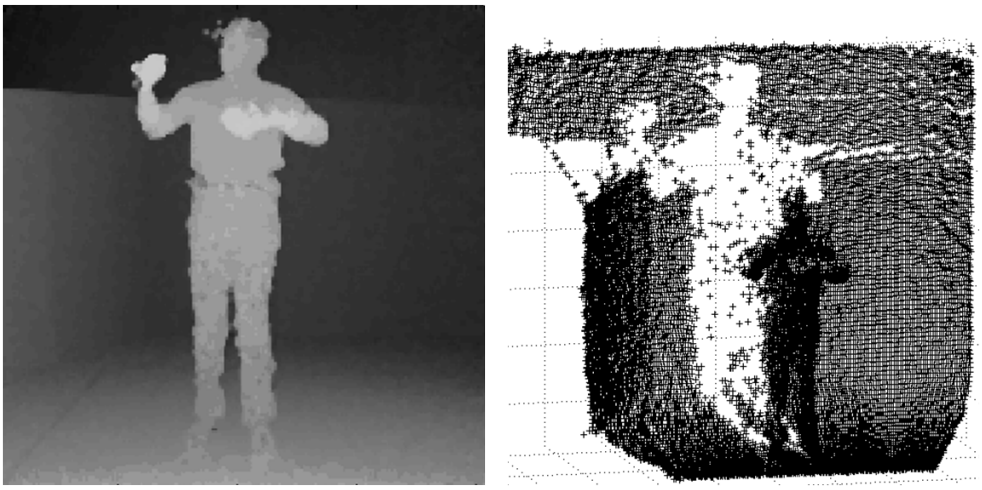
El valor de la ampitud a sería el siguiente:

$$a = \frac{1}{2}\sqrt{(A_3 - A_1)^2 + (A_0 - A_2)^2} \tag{2.12}$$

Y la distancia entre dispositivo y el objeto del que hemos tomado la medida se rige por la siguiente ecuación:

$$d = \frac{c}{4\pi\omega}\phi \tag{2.13}$$

En la figura 2.16(a) se muestra el mapa de profundidad del fondo extraído a partir de una cámara con tecnología Time Of Flight, y en la figura 2.16(b) podemos observar una nube de puntos tal y como la definimos anteriormente, extraída con el mismo sensor TOF.



(a) Imagen mapa de profundidad (b) Imagen de nube de puntos tecnología ToF

Figura 2.16: Muestra de la tecnología ToF

Podemos observar un ejemplo de sensor o cámara TOF en la figura 2.17.

Figura 2.17: PMDVision CamCube

2.2.2.2. Primesensor

La tecnología Primesense es similar en cuanto a los resultados que se obtienen del mapa de profundidad, no obstante el método por el cual se obtienen estos datos es distinto. La adquisición de los datos se realiza mediante la proyección de un patrón de puntos infrarrojos conocido, como una especie de malla compuesta por puntos infrarrojos, y lo que medimos en este caso es la deformación de esa malla, es decir la distorsión del patrón en cuanto a lo que se refiere a la profundidad de los puntos que lo componen.

Figura 2.18: Patrón de puntos infrarrojos

Con un Primesensor la codificación es directa y tiene que ser única en cada posición con el fin de reconocer cada punto en el patrón. El patrón que utiliza Kinect es pseudo aleatorio.

Se puede observar el funcionamiento de los sensores que se basan en la tecnología Primesense en la figura 2.19

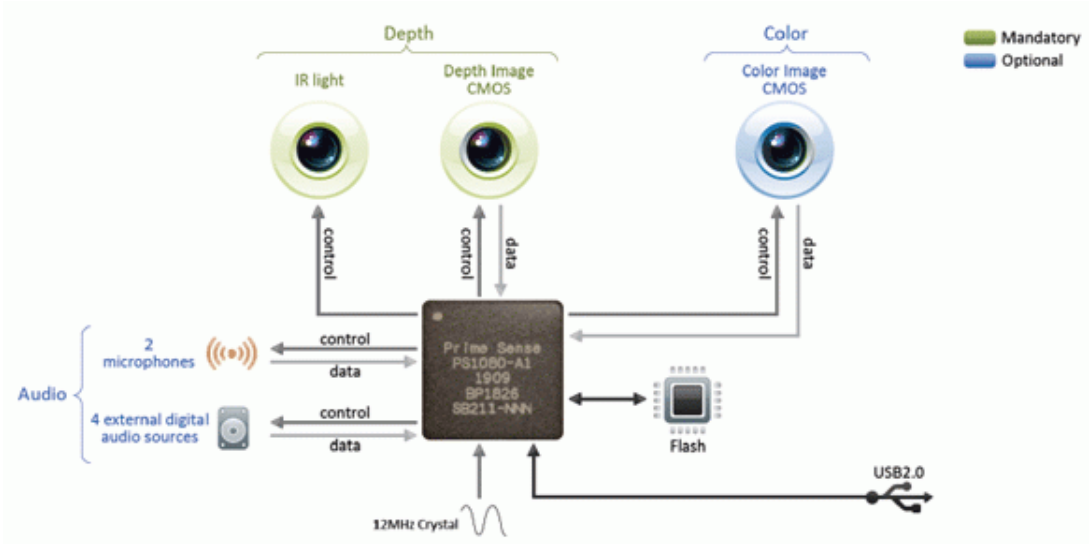


Figura 2.19: Sistema Primesense

¿Cómo calcular los datos de profundidad?

- Triangulación de cada punto entre la imagen distorsionada del patrón de puntos y el mo-

delo del patrón.

- Cada punto tiene su correspondencia en el patrón.
- Se calcula el mapa de profundidad.
- Se calibra en el momento de su fabricación. Tomando un conjunto de imágenes de referencia en diferentes lugares, y se almacenan en la memoria.

Una vez hecho esto ya se puede utilizar el sensor para tomar datos.

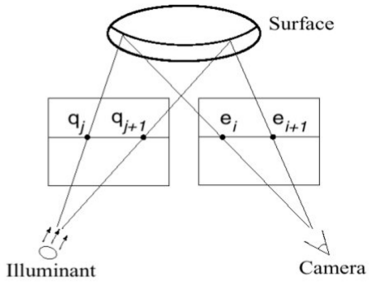


Figura 2.20: Desfase entre el emisor y el receptor

Hay que tener en cuenta que el tamaño de los puntos de la nube depende de la distancia al sensor y de la orientación de este. Así pues se delimitan 3 zonas de funcionamiento para el sensor.

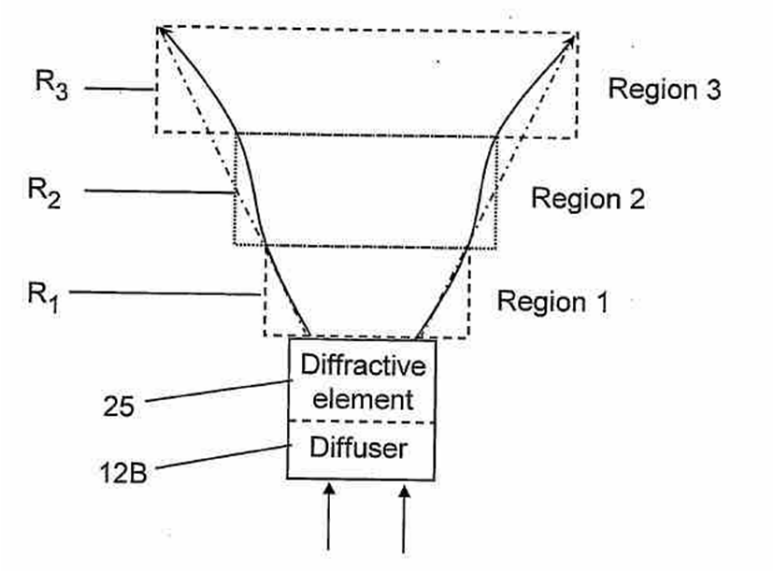
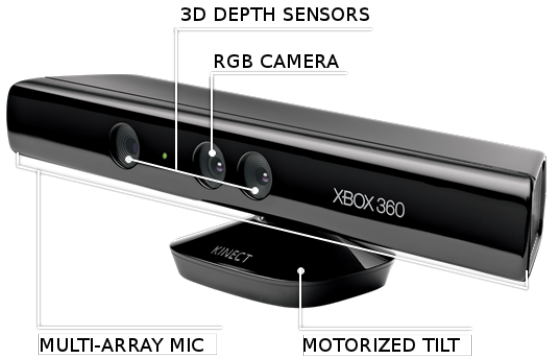


Figura 2.21: Zonas de funcionamiento del Primesensor

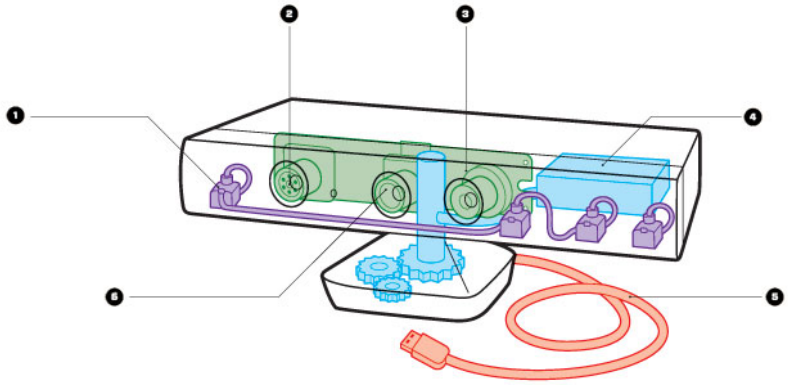
- Primera zona o región (R1): En esta zona la precisión de las medidas tomadas es alta, además que el número de muestras para un mismo objeto dado es mayor, por lo que obtendremos mayor información sobre la forma del objeto en esta zona. Comprende entre los $0,8m$ hasta $1,2m$ aproximadamente.

- Segunda zona o región (R2): En esta zona se aprecia una pérdida significativa de información aunque siguen siendo aceptables los datos extraídos, el error empieza a ser del orden de 1 centímetro. El rango de esta zona esta comprendido entre $1,2m$ y $2m$ aproximadamente.
- Tercera zona o región (R3): En esta zona se distinguen las formas y superficies pero la medida ya no es del todo fiable. Comprende desde los $2m$ hasta los $6m$ aproximadamente, pero a partir de los $3,5m$ la pérdida de precisión es muy grande.

Centrándonos en el Primesensor que hemos utilizado para la realización de este proyecto que es el sensor Kinect de Microsoft, tenemos los datos técnicos:



(a) Partes del Sensor Kinect de Microsoft



(b) Sensor Kinect de Microsoft internamente

Figura 2.22: Sensor Kinect de Microsoft

- Resolución de la imagen de profundidad: 640x480 px.
- Resolución de la imagen RGB: 1600x1200 px.
- Es capaz de captar 60 FPS.
- Rango de operación aceptable: 0,8m 3,5m.
- Error en la resolución espacial X-Y: 3mm a 2m de distancia.
- Error en la resolución de profundidad: 1cm a 2m de distancia.

Otras características del sensor Kinect que la diferencian de la Xtion Pro de Asus y PSDK Reference, es el motor que tiene situado en su base, que le permite cierta movilidad vertical de unos 60º (entre -30º y 30º), también la matriz de micrófonos que tiene instalada alrededor de su borde, y aunque menos importante también lleva instalado un led que puede cambiar de color (rojo, verde, naranja, parpadeo o alternancia de colores), que le confieren funcionalidades que los otros sensores no pueden realizar.

En la figura 2.2.2.2 se muestran los otros 2 sensores que se utilizan además de la Kinect de Microsoft.

(a) Xtion Pro de Asus (b) PSDK Reference

Figura 2.23: Primesensors

2.2.3. Adquisición de los datos

La entrada de datos provenientes de nuestra cámara, antes de poder ser visualizado y analizado con las librerías de PCL, tenemos que adquirirlos por medio de una plataforma. Nosotros hemos utilizado 2 de estas plataformas, una de ellas es OpenNI y la otra es Libfreenect (u OpenKinect).

2.2.3.1. OPENNI

OpenNI (Open Natural Interaction) se puede definir como una recopilación de APIs de distinta índole (multi-idioma) para escribir aplicaciones que utilizan Interacción Natural en un marco multiplataforma. La API de OpenNI se compone de un conjunto de interfaces para la creación de aplicaciones de NI.

Con interacción Natural (NI) se refiere a la interacción que tiene el ser humano con el dispositivo de manera natural con sus sentidos, centrándose principalmente en la audición y visión.

OpenNI ha sido escrito y distribuido bajo la Licencia Pública General de GNU esto significa que su código puede ser distribuido libremente. Es decir, se puede redistribuir y modificar bajo los términos de la Licencia Pública General de GNU. Toda la información sobre la licencia se encuentra en la página http://www.gnu.org/licenses/.

El objetivo principal de OpenNI es formar una API estándar que permite la comunicación entre:

- Visión y sensores de audio, serían los dispositivos o sensores encargados de la adquisición de los datos del entorno (visuales y auditivos).
- Visión y la percepción de middleware de audio, serían los componentes de software que analizan el audio y los datos visuales que se introducen desde los sensores).

OpenNI proporciona un conjunto de APIs para ser implementadas por los dispositivos con sensor/es, y un conjunto de APIs que son implementados por los componentes de middleware. Al romper la dependencia entre el sensor y el middleware, la API de OpenNI permite escribir

aplicaciones y portarlas sin esfuerzo adicional para operar en la parte superior de diferentes módulos middleware (Una vez escrito, lo desplegamos por todas partes).

La API de OpenNI también permite a los desarrolladores de middleware escribir algoritmos de formatos de datos en bruto, sin importar para qué dispositivo/sensor fue pensado. OpenNI es una API de código abierto que está disponible públicamente en http://www.OpenNI.org.

OpenNI permite el reconocimiento de voz, los dispositivos pueden recibir instrucciones a través de comandos vocales. También reconoce gestos de las manos, existen gestos de las manos predefinidos que son reconocidos e interpretados para activar y controlar los dispositivos.

Otro de los avances de esta tecnología es que permite el tracking del cuerpo, analizándolo e interpretándolo. Todo esto en un principio tenía fines comerciales (videojuegos), pero con el código abierto se abren las puertas a la investigación de la visión artificial en 3D.

OpenNI proporciona la interfaz para los dos dispositivos físicos y los componentes de middleware. Estos componentes se denominan módulos, y se utilizan para padquirir y procesar los datos sensoriales. Los módulos que actualmente están soportados por OpenNI son:

- Sensor 3D
- Cámara RGB
- Cámara de infrarrojos
- Dispositivo de audio (un micrófono o una matriz de micrófonos)

En cuanto a los componentes middleware podemos decir que se pueden utilizar para analizar:

- El cuerpo: procesa los datos sensoriales y genera la información relacionada con el cuerpo (por lo general la estructura de datos que describe las articulaciones, la orientación, centro de masas, etc).
- El punto de la mano: procesa los datos sensoriales y genera la ubicación de un punto de la mano (normalmente la palma de la mano).

- Detección de gestos: identifica los gestos predefinidos (por ejemplo, una mano que saluda) y alerta a la aplicación.
- Análisis de la escena o entorno: analiza la imagen de la escena con el fin de producir información (como la separación entre el primer plano de la escena (es decir, las figuras) y el fondo, las coordenadas del plano del suelo, la identificación individual de las figuras en la escena, ect).

En la figura 2.24 se muestra un esquema del funcionamiento de OpenNI.

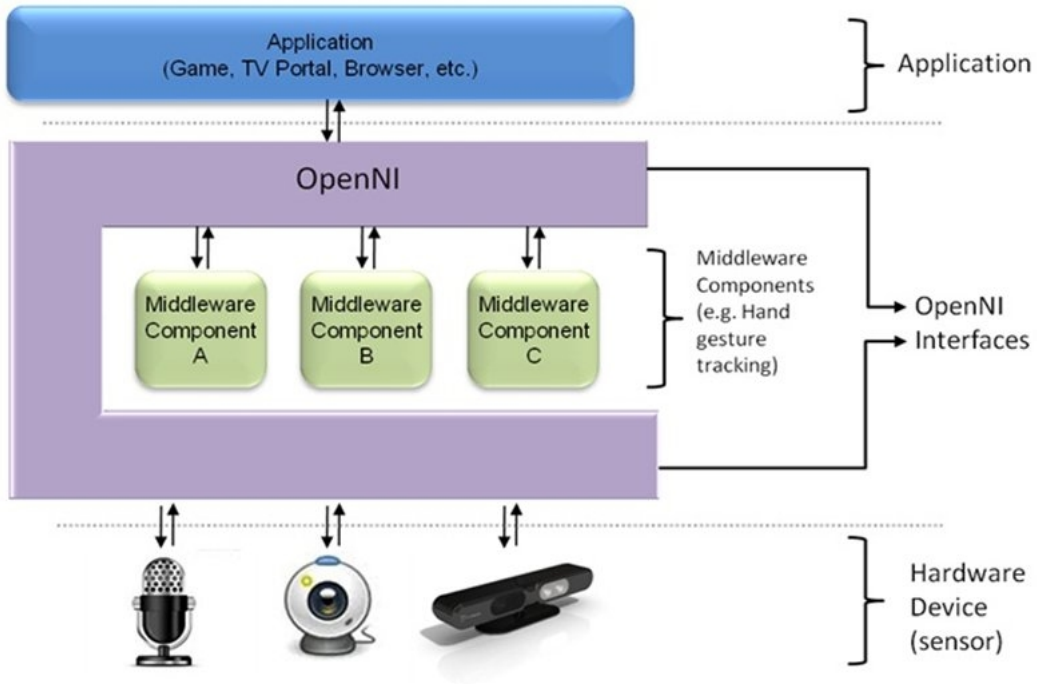


Figura 2.24: Funcionamiento de OpenNI

Nodos de producción

Los datos significativos en 3D se definen como datos que pueden comprender, entender y traducir la escena. La creación de los datos significativos en 3D es una tarea compleja. Típicamente, se inicia mediante el uso de un dispositivo/sensor que produce datos de salida. A menudo, estos datos son un mapa de profundidad, donde se representa cada píxel por su distancia al sensor. La middleware especializada se utiliza para procesar esta salida en bruto, y producir una salida de nivel superior, que puede comprender y utilizar la aplicación.

OpenNI define nodos de producción , que son un conjunto de componentes que tienen un papel productivo en el proceso de creación de los datos necesarios para las aplicaciones de interacción (NI). Cada nodo de producción encapsula la funcionalidad que se refiere a la generación del tipo de datos específico. Estos nodos de producción son los elementos fundamentales de la interfaz de OpenNI que establece las solicitudes. Sin embargo, en la API de los nodos de producción solo se define el lenguaje. La generación de la programación lógica de datos debe ser implementada en los módulos que se conectan a OpenNI.

Cada nodo de producción es una unidad independiente que genera un tipo específico de datos, y puede atender a cualquier objeto, ya sea otro nodo de producción, o la propia aplicación. Sin embargo, por lo general algunos nodos de producción utilizan siempre los nodos de producción más bajos que representan los datos a nivel de tipo, el hecho de analizar estos datos de nivel inferior y produce más datos a nivel de la aplicación. Por ejemplo, supongamos que que queremos seguir el movimiento del cuerpo de una persona mediante una aplicación en una escena 3D. Para ello necesitamos un nodo de producción que proporcione los datos del cuerpo, o, en otras palabras, un generador de usuario. Este generador de usuario obtiene sus datos de un generador de profundidad. Un generador de profundidad es un nodo de producción que se ejecuta por un sensor, que toma los datos en bruto de un sensor de profundidad.

A continuación se muestran los ejemplos más comunes de alto nivel utilizando el middleware Primesense Nite:

- **I**dentificación de un gesto con la mano (por ejemplo, saludar). La salida es una alerta a la

aplicación de que un gesto específico se ha producido.

- **U**bicación de la mano de un usuario. La salida puede ser el centro de la palma (a menudo denominado como "punto de mano ") o las puntas de los dedos.

Figura 2.25: Ejemplo PointViewer de OpenNI con Nite

- **D**etección de usuarios. La salida es la ubicación actual del/los usuario/s.
- **T**racking del cuerpo humano dentro de la escena. La salida es la ubicación actual y la orientación de las articulaciones de esta cifra (a menudo denominado como "datos del cuerpo ").

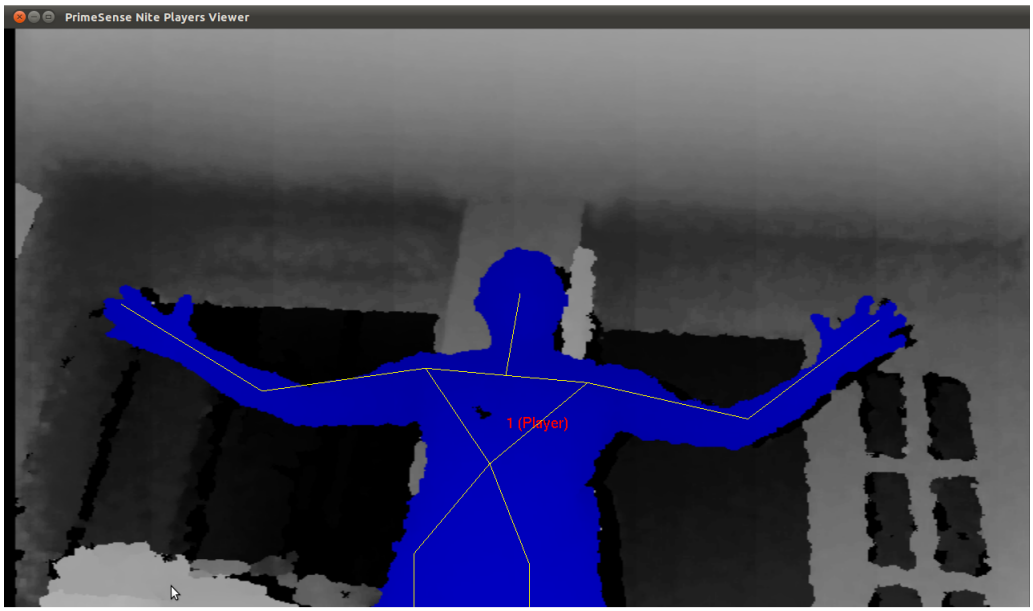


Figura 2.26: SamplePlayers de OpenNI con Nite

OpenNI/Primesense NITE

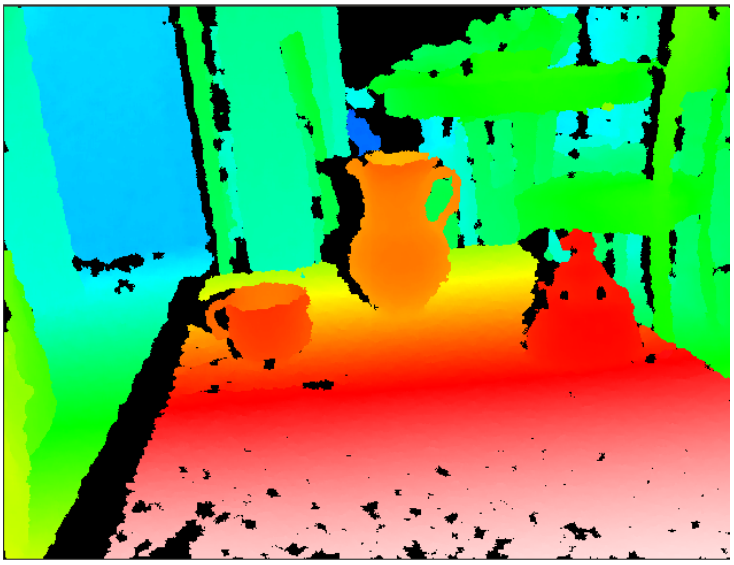
NITE es un middleware que se utiliza dentro de OpenNI y se podría decir que es el cerebro del proceso que lleva la máquina para comprender los movimientos naturales del cuerpo humano. Lo que hace es traducir los movimientos humanos en entradas para la aplicación, y responde a ellas según la programación que lleve interna.

LIBFREENECT

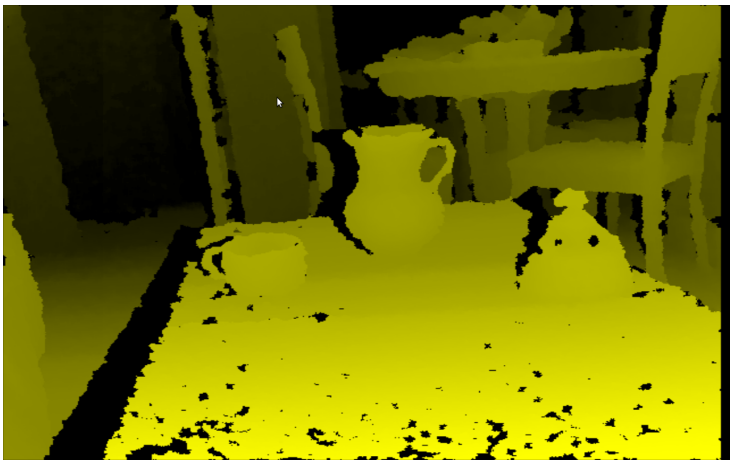
Libfreenect pertenece al proyecto OpenKinect que es una comunidad que trabaja con código abierto y que además contribuye al desarrollo de este, implementando software de cara al nuevo hardware de Microsoft para la consola Xbox, Kinect, aunque también es compatible con la Xtion Pro de Asus y la PSDK Reference.

Figura 2.27: Sensores compatibles con OpenNI y Libfreenect

Las funcionalidades que tiene esta plataforma son en su inmensa mayoría las mismas que OpenNI, aunque el mapa de profundidad se presenta de manera diferente, ya que se muestra con 3 canales de color, presentando así mayor facilidad para que el usuario pueda tener una mayor resolución al menos desde el punto de vista visual para distinguir las distintas zonas del mapa de profundidad, en la figura 2.2.3.1 se muestra una comparación entre el mapa de profundidad de OpenNI y el de LibFreenect.



(a) Mapa de profundidad de Libfreenect



(b) Mapa de profundidad de OpenNI

Figura 2.28: Comparativa de Imágenes RGB

Puede observarse que la diferencia entre ambos sistemas no es muy grande, es más los datos que recibe OpenNI y Libfreenect para pintar el mapa de profundidad son los mismo, por lo que aunque aparentemente nos parezca que da más información uno que otro, en realidad no es así presentan los mismos datos representándolos de manera un poco distinta.

Libfreenect permite al igual que OpenNI mostrar la imagen RGB (Imagen 2.2.3.1), la imagen de Infrarrojos (Imagen 2.30), y otras funcionalidades como las que utilizan la matriz de micrófonos del sensor de Microsoft Kinect.

(a) Imagen RGB Libfreenect

(b) Imagen RGB de OpenNI

Figura 2.29: Comparativa de Imágenes RGB

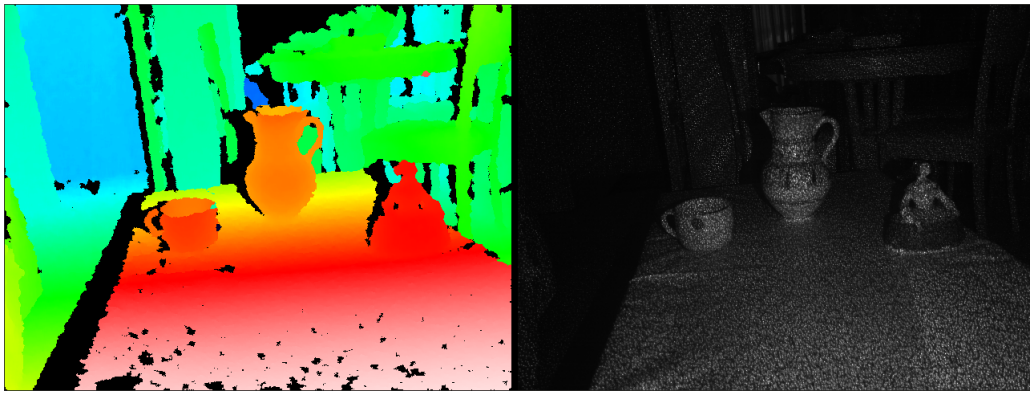


Figura 2.30: Imagen de cámara IR de Libfreenect

En la imagen de infrarrojos puede observarse la cantidad de puntos que lanza el sensor Kinect para la adquisición de los datos de profundidad. Como curiosidad podemos decir que este proceso genera 2 sombras desfasadas entre sí una propia del emisor de infrarrojos que generan los objetos al tapar la luz infrarroja y otra al rebotar hacia el receptor que se encuentra unos centímetros a la izquierda de este y es este hecho precisamente (que no se encuentren localizados emisor y receptor en el mismo sitio) el que provoca que las sombras estén desfasadas.

2.3. ¿Cómo reconocer objetos en 3D?

2.3.1. Problemas básicos

El reconocimiento de objetos en 3 dimensiones es relativamente más sencillo que en 2 dimensiones, ya que tenemos más información que nos aporta más características distintivas entre los objetos. A pesar de esto el reconocimiento de objetos no es sencillo, y requiere de un estudio previo de las características de los objetos que mejor los definan. Y, al igual que hacíamos en las imágenes digitales, hay que realizar una serie de operaciones sobre la nube de puntos que resalten las características distintivas y que eliminen el ruido u otros desperfectos que nos molesten a la hora de analizar los objetos de la nube.

Existe un problema añadido cuando el reconocimiento se realiza en tiempo real, que es la velocidad de cómputo, es decir, la velocidad que tarda nuestro ordenador en realizar todos los

algoritmos necesarios para la ejecución del programa. Este problema lleva consigo la contraposición entre velocidad y cantidad de datos, con esto queremos decir que al aumentar el número de datos que introducimos en los algoritmos, disminuye la velocidad, pero la calidad del resultado del algoritmo es directamente proporcional al número de datos que le introduzcamos, por lo que si disminuimos mucho el número de datos a introducir en los algoritmos, aumentamos la velocidad, pero a costa de disminuir la fiabilidad del proceso.

2.3.2. Solución. Algoritmos

En este proyecto se ha realizado la detección de objetos y planos, y para ellos hemos tenido que realizar un filtrado de la nube de puntos, una búsqueda de los subgrupos de puntos en la nube, el reconocimiento de estos subgrupos como plano u objeto, el etiquetado de los distintos objetos de la imagen y la reconstrucción superficial de los objetos.

2.3.2.1. Filtrado de la nube

En todo análisis que se realice siempre existen muestras erróneas, a estas muestras erróneas las llamamos ruido. Este ruido puede entorpecer la tarea de los algoritmos de reconocimiento o que realicen cualquier otra operación, ya que supondría estar operando con datos erróneos y no queda otra que dada esta situación el resultado del algoritmo sea en mayor o menor medida erróneo.

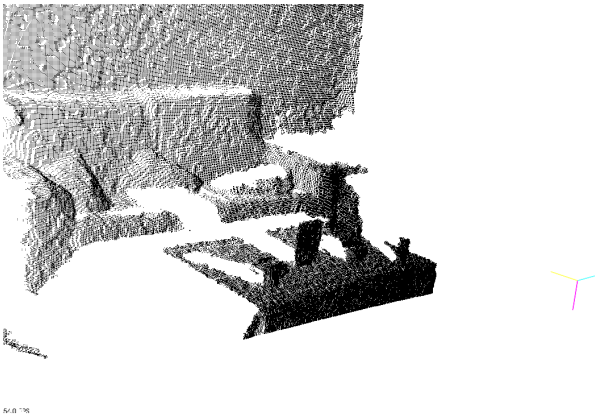
Por lo tanto, lo correcto es realizar una serie de operaciones que eliminen o atenúen el peso de estos datos erróneos en nuestras muestras. Estas operaciones son algoritmos de filtrado, y lo que hacen es intentar discriminar aquellos valores que estén fuera de rango o que no cumplan las características que cumplen los datos correctos.

En nuestro caso hemos utilizado 3 filtros cuyo efecto combinado, en el momento adecuado, elimina los datos que considera incorrectos o que estorben para la realización de los siguientes algoritmos.

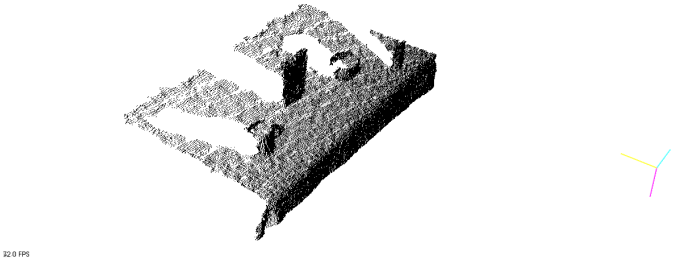
- Umbralización o Threshold (PassThrough): Hemos decidido utilizar este algoritmo por 2 razones la primera es que nuestro sensor Kinect tiene un rango de profundidad en el que las muestras son fiables y si nos pasamos de este rango ya no son tan fiables, y esto perjudica a los resultados obtenidos por los algoritmos de reconocimiento de los planos y

objetos, la segunda razón es que al utilizar este filtro eliminamos un montón de datos y de esta forma aumentamos la velocidad del proceso. La ecuación que rige este filtro es la siguiente:

$$f(x,y,z) = \left\{ \begin{array}{ll} f(x,y,z) & \text{si } Z \geq \text{T} \\ 0 & \text{si } Z < \text{T} \end{array} \right\}$$



(a) Imagen sin umbralizar



(b) Umbralización de la profundidad

Figura 2.31: Filtro Threshold para Umbralización

- Filtro VoxelGrid: Este algoritmo lo que hace es reducir la resolución de la nube, esto nos sirve para aumentar la velocidad de los algoritmos, aunque conlleva una pérdida de información significativa y más cuanto más filtremos. El filtro VoxelGrid crea una red de Voxel que son como pequeñas cajas tridimensionales encima de los puntos de la nube entonces los puntos que han quedado encerrados en cada Voxel son sustituidos por su centroide.

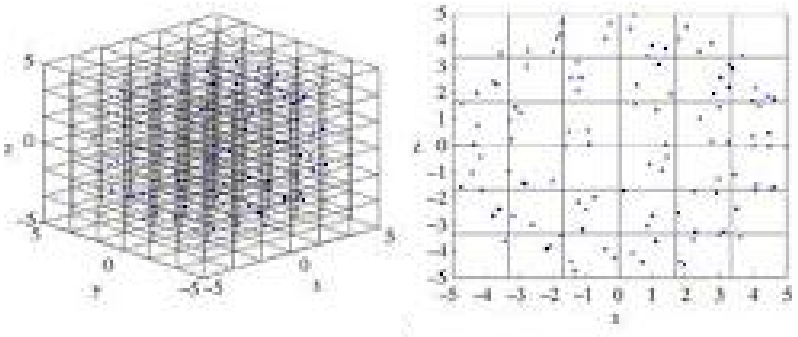
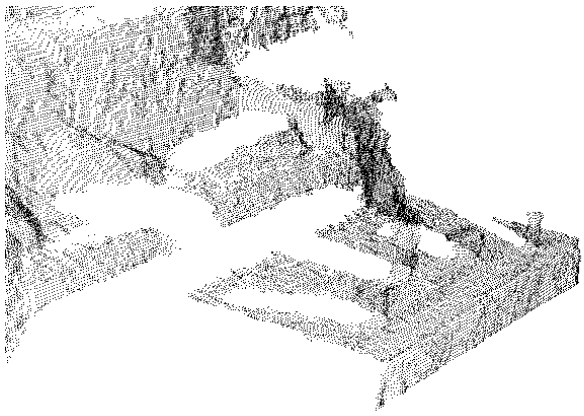
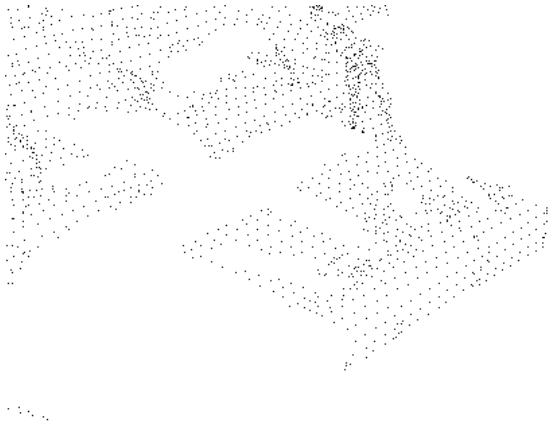


Figura 2.32: Voxelgrid



(a) Imagen sin filtrar



(b) Filtrado con un tamaño de voxel de 0,5

Figura 2.33: Filtro Voxelgrid sobre nube de puntos

- Stadistical Outlier Removal: Este algoritmo estudia la posibilidad estadística de que un punto pertenezca a un grupo de puntos o no. El algoritmo itera con todos los datos de entrada en dos ocasiones: En la primera iteración, se calculará la distancia media que cada punto tiene con sus K vecinos más cercanos. El valor de K se puede establecer mediante $setMeanK()$. A continuación, se calculan la media y la desviación estándar de todas estas distancias con el fin de determinar un umbral distancia. El umbral de distancia será igual a: $\bar{d} = M.\sigma$ donde $\bar{d}$ es la media de la distancia, M es el multiplicador de la desviación estándar, σ es la desviación estándar de la probabilidad de que la distancia sea igual a la media de las distancias. El multiplicador de la desviación estándar se puede ajustar mediante $setStddevMulThresh()$. En la segunda iteración los puntos se clasifican como inlier si la distancia a sus vecinos está por debajo del umbral o como outlier si la distancia

está por encima de este umbral.

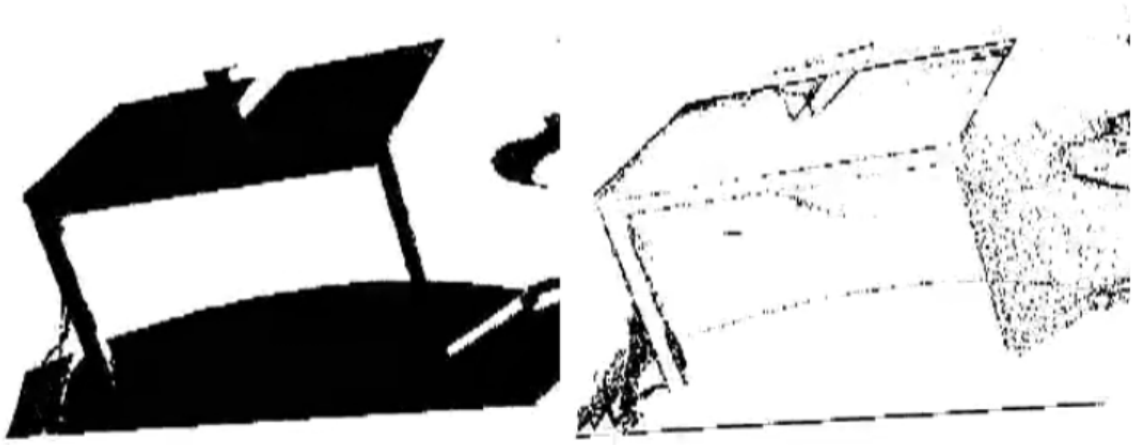


Figura 2.34: A la izquierda la imagen filtrada y a la derecha el ruido que hemos filtrado

2.3.2.2. Kdtree. Como método iterativo de búsqueda

kdtree es la abreviatura de $k - dimensional\ tree$, que es un árbol de decisión que según las condiciones que le introduzcamos va particionando el espacio de la estructura de datos y organiza los puntos del espacio Euclídeo de k dimensiones.

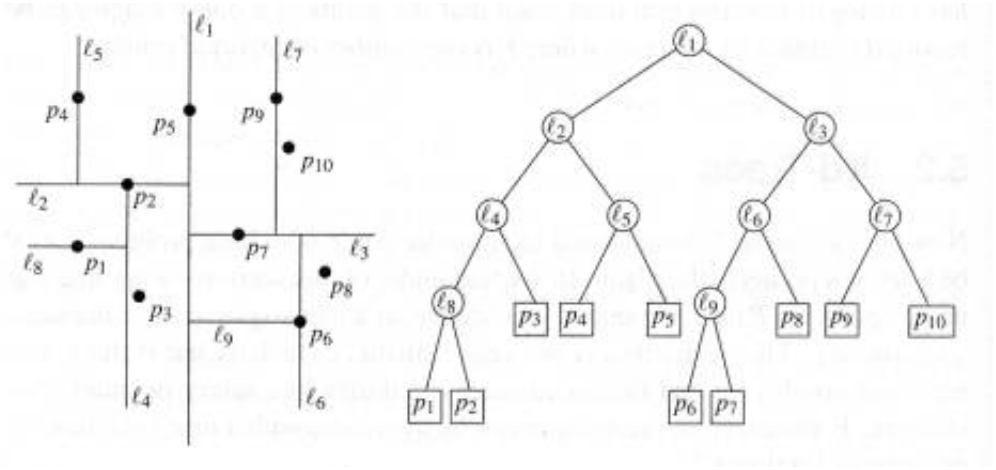


Figura 2.35: kdtree en 2 dimensiones

Los kdtree se usan para múltiples aplicaciones que trabajan con estructuras de datos, y

sirven para realizar una búsqueda multidimensional, estas búsquedas suelen realizarse para encontrar ciertas condiciones en el espacio de datos, como por ejemplo búsqueda del vecino más proximo (esto es la búsqueda del punto más cercano al punto estudiado).

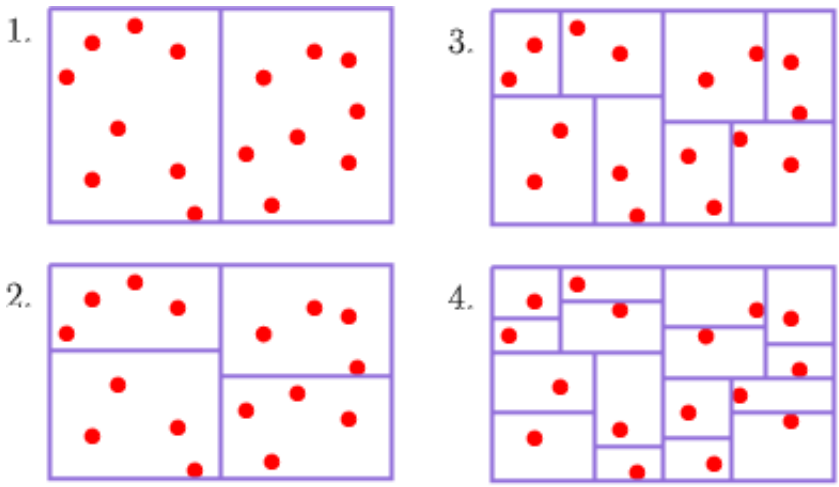


Figura 2.36: Proceso de particionado del espacio

La letra k hace refierencia al número de dimensiones de nuestro espacio de datos. Por tanto un $kdtree$ tridimensional técnicamente es un árbol de decisión en 3D. Aunque es más descriptivo llamarlo kdtree tridimensional, ya que un árbol tridimensional puede referirse a otro tipo de árbol, pero el término kdtree se refiere a un tipo concreto de árbol de decisión.

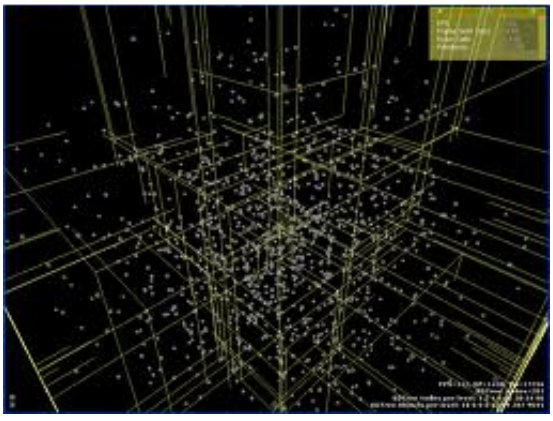


Figura 2.37: kdtree en 3 dimensiones

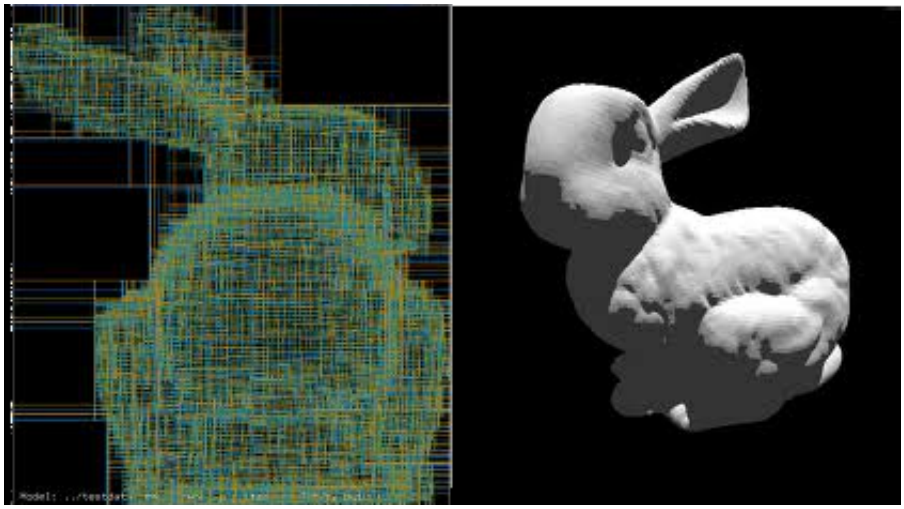


Figura 2.38: Ejemplo de kdtree en 3 dimensiones

2.3.2.3. Euclidean Cluster Extraction

Se basa en considerar subgrupos de puntos en función a una distancia Euclídea entre los puntos, si se cumple la condición en la distancia entre los puntos se considera que forman parte del mismo Cluster, y si no se considera que son puntos de Clusters distintos. La distancia Euclídea entre dos puntos $P_1 = (x_1, y_1, z_1)$ y $P_2 = (x_2, y_2, z_2)$ viene dada por la siguiente ecuación:

$$d_E(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{2.14}$$

2.3.2.4. RANSAC. Random Sample Consensus

Este algoritmo se utiliza para detección de figuras en un espacio de puntos. En el caso de las bibliotecas de PCL nos permite reconocer líneas, esferas, conos, cilindros y planos, en el entorno tridimensional de las nubes de puntos.

Antes de explicar en que consiste este algoritmo explicaremos que son los puntos que se consideran como ”inliers” y cuales se consideran como outliers. Un punto se considera inlier cuando se encuentra dentro del rango preestablecido en el cual se considera dentro del modelo que hemos considerado, es decir si la distancia entre el punto real y el que le correspondería en el modelo es suficientemente pequeña como para ser considerado aceptable, y un punto se considera como ”outlier” cuando el punto se encuentra significativamente distanciado del punto que se corresponde con él en el modelo. Por ejemplo supongamos que nuestro modelo es una recta, y que tenemos una nube de puntos como dispersión de datos como se muestra en la figura 2.39, la distancia es la que importa a la hora de considerar un punto como ”inlier” o no, ya que la distancia es el error que depende de la posición del punto y de los parámetros del propio modelo.

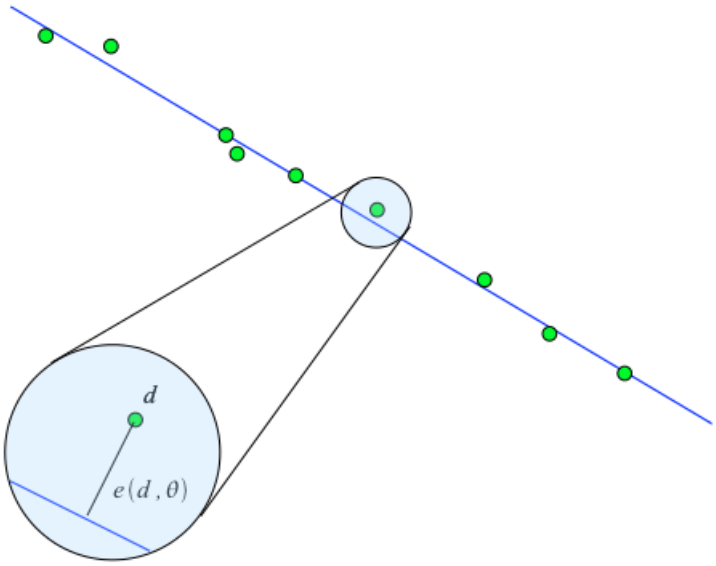


Figura 2.39: Modelo RANSAC de una línea

El error viene dado por la siguiente expresión:

$$e_{\mathcal{M}}(d,\theta) = \frac{\theta_1 x_1 + \theta_2 x_2 + \theta_3}{\sqrt{\theta_1^2 + \theta_2^2}} \tag{2.15}$$

Donde $\theta_1 x_1 + \theta_2 x_2 + \theta_3$ es el modelo de una recta y $\sqrt{\theta_1^2 + \theta_2^2}$ es la distancia euclídea entre cada punto y el modelo.

El algoritmo RANSAC consiste en definir un modelo de una figura y con la ecuación de este modelo y el error que calculamos en cada iteración con los puntos del entorno encajar dicho modelo en el espacio de puntos de manera que se minimice el error al máximo. Al tratarse de un método iterativo, cuantas más iteraciones hagamos, mayor optimización en la minimización del error obtendremos y por tanto mejores resultados. También debemos decir que este algoritmo realiza la búsqueda de los puntos para calcular el error de manera aleatoria, de esta forma es más probable abarcar grandes dimensiones en el espacio de puntos y por tanto encontrar antes el espacio que se corresponde mejor con el modelo, aumentando la velocidad del proceso, aunque al ser aleatorio esto no es del todo seguro. También debemos decir, que por las características de este algoritmo cuantos más elementos contenga la muestra original de puntos, mejor será el reconocimiento.

El modelo $\mathcal{M}$ se define como:

$$\mathcal{M}(\theta) = \{d \in \mathbb{R}^n : f_{\mathcal{M}}(d;\theta) = 0\} \tag{2.16}$$

donde θ es el vector de parámetros del modelo, $f_{\mathcal{M}}$ es una función suave, cuyo nivel Set contiene todos los conjuntos de puntos que se ajustan al modelo M y d es la distancia euclídea entre el punto real y el modelo para un espacio n-dimensional.

2.3.2.5. Etiquetado de objetos

Este algoritmo lo que pretende es separar visualmente unos objetos de otros para hacer el proceso más atractivo de cara al usuario, además de extraer información relativa a la posición física de cada uno de los objetos.

Primeramente hemos utilizado un algoritmo llamado $getMinMax3D()$ que lo que hace es buscar, en cada uno de los objetos ya segmentados, las coordenadas máximas y mínimas absolutas de la nube que representa al objeto estudiado en X,Y y Z.

Con esta información podemos calcular el centro geométrico del objeto de la siguiente forma:

$$X_{centro} = X_{min} + \frac{X_{max} - X_{min}}{2}; \quad Y_{centro} = Y_{min} + \frac{Y_{max} - Y_{min}}{2}; \quad Z_{centro} = Z_{min} + \frac{Z_{max} - Z_{min}}{2} \tag{2.17}$$

Siendo las coordenadas mínimas X_{min}, Y_{min} y Z_{min}, y las coordenadas máximas X_{max}, Y_{max} y Z_{max}.

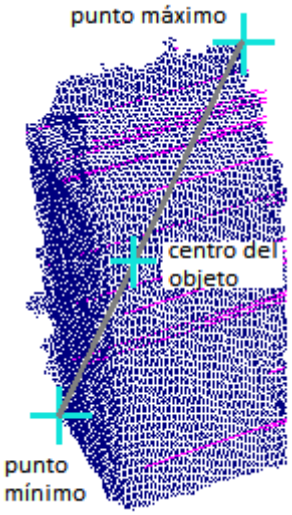


Figura 2.40: Algoritmo MinMax con cálculo del centro del objeto

Sabiendo cual es la posición del centro y calculando la altura, anchura y profundidad del objeto de la siguiente forma:

$$Altura = X_{max} - X_{min}; \quad Anchura = Y_{max} - Y_{min}; \quad Profundidad = Z_{max} - Z_{min} \tag{2.18}$$

Podemos definir un cubo en el cual esté inscrito el objeto, si introducimos dichos datos como coeficientes del cubo. El resultado sería algo parecido a lo que mostramos en la siguiente figura:

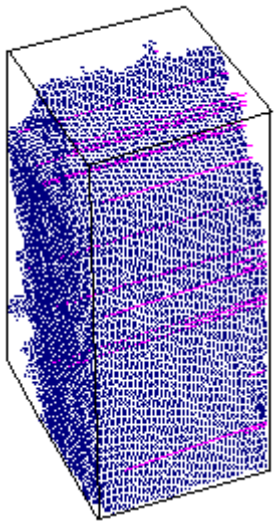
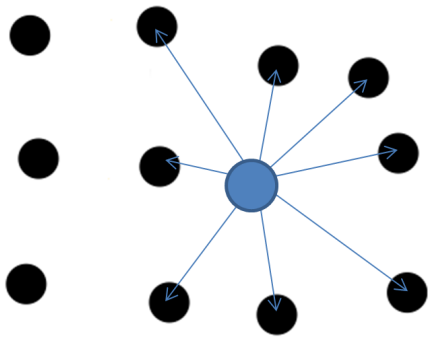


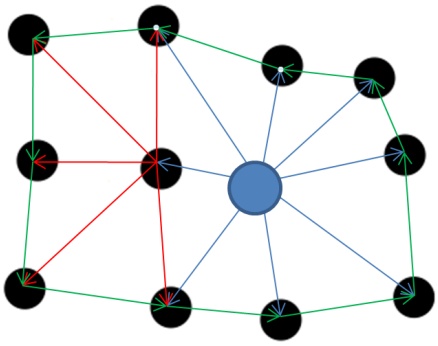
Figura 2.41: Cubo que inscribe al objeto

2.3.2.6. Reconstrucción superficial con Fast triangulation

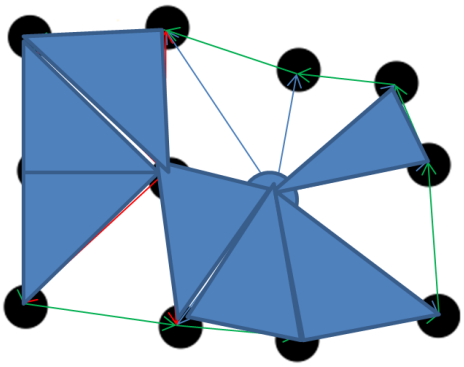
Antes de realizar el algoritmo de triangulación como tal, se necesitan una serie de datos previos. Primero tomamos la información de las normales de los puntos (que se han calculado previamente), se eligen un número máximo de vecinos por cada punto de la nube, y una distancia máxima a la cual se considera que los puntos son vecinos, también se definen los ángulos máximos y mínimos para la generación de los triángulos. Con esta información se generan los triángulos uniendo un punto con cada uno de sus vecinos más cercanos en el rango de distancias y ángulos del triángulo definidos, y se rellenan estos triángulos utilizando la información de las normales de los vértices del triángulo, de tal forma que se reconstruye la superficie del objeto. No obstante si el número de puntos es pequeño la reconstrucción será inexacta, pero merece la pena utilizar este algoritmo ya que es muy rápido, y se puede utilizar en tiempo real.



(a) Búsqueda y unión con los k-vecinos más próximos del punto



(b) Búsqueda y unión con los k-vecinos del vecino K_n del primer punto



(c) Reconstrucción superficial con los datos de las normales de los puntos y las líneas de unión entre puntos

Figura 2.42: Esquema del Algoritmo Fast Triangulation

Capítulo 3

Plataforma de desarrollo

En este capítulo explicaremos la plataforma de desarrollo que hemos utilizado para la realización de este proyecto. Comenzaremos con una pequeña introducción sobre el Sistema Operativo que hemos elegido y sobre las herramientas de programación usadas así como las plataformas..

3.1. Sistema Operativo

El Sistema Operativo que hemos elegido es Linux, la razón para elegir este entorno son las características que tiene, que lo hacen más rápido y estable que otros sistemas operativos. La razón más importante para la elección de este sistema es su compatibilidad con las herramientas y aplicaciones que necesitabamos para cumplir las especificaciones que debía cumplir este proyecto.

En concreto nos hemos decantado por Ubuntu 11.04 de 32 bits, que se trata de una distribución GNU/Linux, cuya licencia es totalmente libre. Otras ventajas con las que cuenta este sistema operativo son:

- Multiplataforma, multitarea, multiprocesador y multiusuario.
- Carga selectiva de programas según la necesidad.
- Protección de la memoria, haciéndolo más estable frente a caídas del sistema.
- Uso de bibliotecas enlazadas estática y dinámicamente.

3.2. Lenguaje C++

Este proyecto se ha realizado utilizando código en Lenguaje C++ ya que es un lenguaje de nivel superior, el cual es intuitivo y relativamente fácil de programar, aunque la principal razón de esta elección es que las herramientas que hemos utilizado se programan con dicho lenguaje, tanto PCL (Point Cloud Library), como OpenNI (Open Natural Interaction), como VTK para la visualización.

Algunas de las características que hacen que resulte atractiva la programación en este lenguaje son:

- Programación orientada a objetos.
- Portabilidad.
- Brevedad.
- Programación modular.
- Compatibilidad con C.
- Velocidad.

El lenguaje C surgió en los laboratorios Bell de AT& T ha sido asociado con el sistema operativo UNIX. Su eficiencia y facilidad de uso han hecho que el lenguaje ensamblador apenas haya sido utilizado en UNIX. Este lenguaje ha evolucionado paralelamente a UNIX. Una muestra de esto es que en 1980 se añaden al lenguaje C nuevas funcionalidades como clases, conversión de tipo y chequeo del tipo de argumentos de una función, entre otras, a este desarrollo del lenguje C se le denominó lenguaje C con Clases.

En 1983, el lenguaje C con Clases sufrió una evolución y con ella fue rediseñado, extendido y nuevamente implementado. A esta nueva evolución se le denominó Lenguaje C++. Ahora las extensiones principales eran funciones virtuales, funciones sobrecargadas (un mismo identificador puede representar distintas funciones), y operadores sobrecargados (un mismo operador puede utilizarse en distintos contextos y con distintos significados).

3.3. Point Cloud Library (PCL)

Point Cloud Library (PCL) es un proyecto independiente a gran escala y de código abierto para el análisis y procesado de nubes de puntos 3D. PCL cuenta con numerosos algoritmos que se encuentran en la vanguardia o estado del arte de la visión artificial, que incluye algoritmos de filtrado, reconstrucción superficial, modelado, segmentación, etc.... Pero los algoritmos de más alto nivel son sin duda los que se centran en el mapeado y el reconocimiento de objetos.

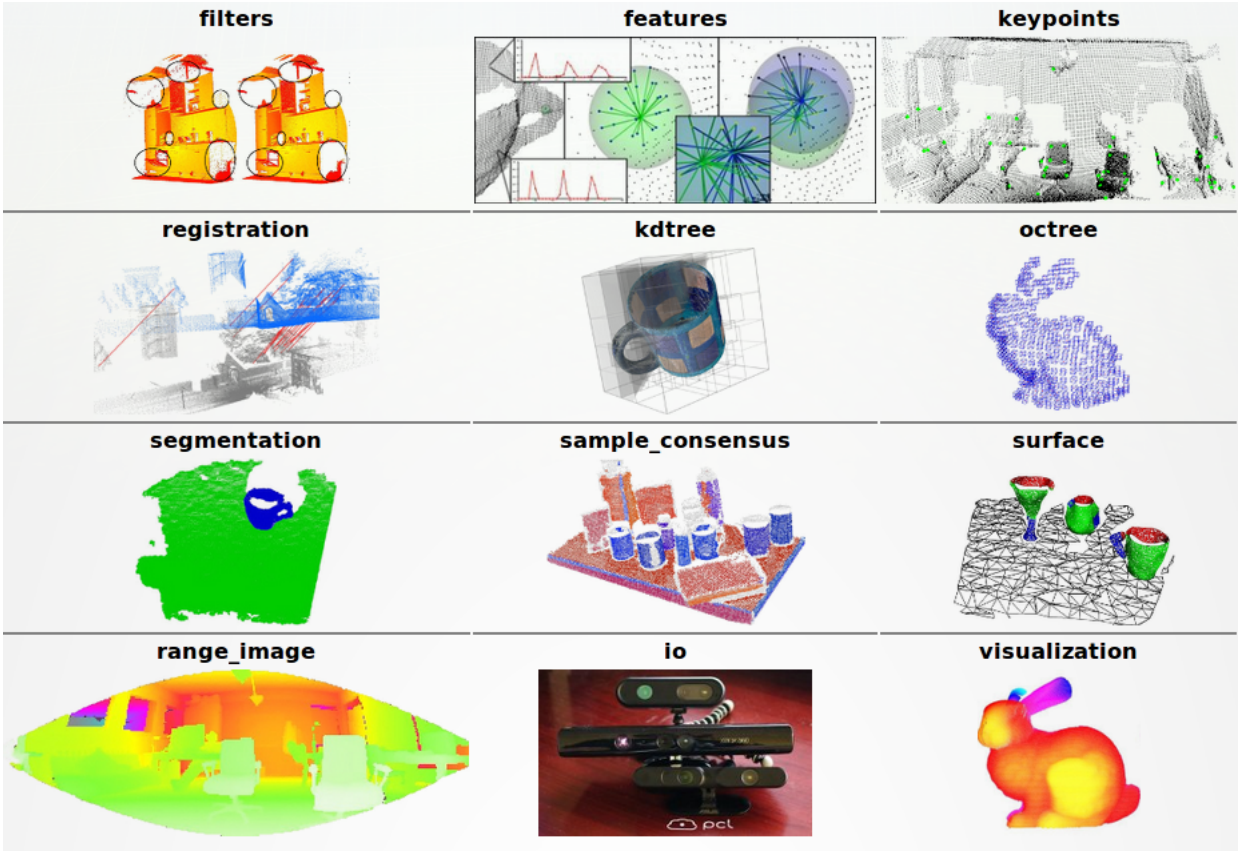


Figura 3.1: Bibliotecas de Point Cloud Library

PCL se encuentra bajo los términos de la licencia BSD. Esto quiere decir que es totalmente libre su comercialización y uso para investigación. La página oficial es http://pointclouds.org/.

En cuanto a la página oficial de PCL, podemos decir que se encuentra distribuida de tal forma que resulta atractiva para el usuario, ya que cuenta con una serie de tutoriales dedicados a cada una de las bibliotecas de PCL. Además es un software que está en continua evolución ya que la comunidad PCL está integrada por investigadores, además de otros usuarios que generan, modifican y mejoran tutoriales, código o establecen las bases para generar nuevos algoritmos u optimizaciones de estos.

Es multiplataforma ya que es compatible con los sistemas operativos de Linux, Microsoft Windows y Apple Mac OS, además se puede implementar en Qt Creator, Visual Studio, CodeBlocks y otros. También es totalmente compatible con ROS (Robot Operating System), que explicaremos en el siguiente apartado.

PCL utiliza OpenNI internamente para la adquisición de los datos por parte del dispositivo Primesense, y engloba también VTK con algunas modificaciones específicas de PCL para la visualización de las nubes de puntos 3D.

Figura 3.2: Visualizador de PCL

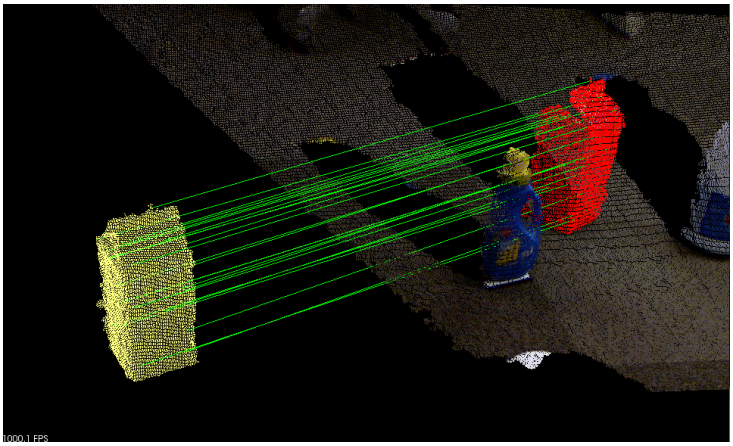
Un hecho que cabe destacar es que PCL junto con NVidia CUDA y Itseez han conseguido reproducir el algoritmo Kinect Fusion (Figura 3.3) de Microsoft con herramientas totalmente libres (de código abierto).

Figura 3.3: Kinect Fusion con código abierto

En esta imagen se muestra la imagen RGB arriba a la izquierda, la imagen de Profundidad de OpenNI con PCL abajo a la derecha y la reconstrucción superficial con mapeado en tiempo real arriba a la derecha. Kinect Fusion(Kinfu) es una muestra del enorme potencial que tienen las bibliotecas de PCL.

Otro de los logros destacables es la implementación del algoritmo SIFT (Scale-Invariant Feature Transform), que explicamos anteriormente, para nubes de puntos. Este algoritmo es mucho más potente que el algoritmo en 2D ya que es totalmente invariante a la iluminación (excepto a la luz infrarroja si lo utilizamos en exterior) y es totalmente invariante al color (si lo deseamos) ya que los datos de profundidad son independientes de los datos de color. Este algoritmo todavía no se ha implementado de forma oficial en tiempo real, pero funciona para reconocimiento estático de objetos.

(a) Nube de puntos del entorno



(b) SIFT en 3D

Figura 3.4: Reconocimiento por SIFT en 3D

3.4. ROS (Robot Operating System)

ROS fue desarrollado originalmente en 2007 por la Universidad de Stanford en el laboratorio de Inteligencia Artifial (IA). Actualmente cuenta con una comunidad en constante crecimiento con más de 100 paquetes, 200 pilas y 50 repositorios. ROS se encuentra bajo los términos de la licencia BSD.

ROS es un metasistema de nodos que ofrece bibliotecas y herramientas como software para generar aplicaciones robóticas. En ROS se proporciona también una guía del hardware que se puede usar, así como los controladores de dispositivos, visualizadores, paso de mensajes, gestión de paquetes... por medio de nodos. ROS cubre la mayor parte de la programación de

bajo nivel (hilos, protocolos de comunicación, mapas de memoria, etc) proporcionando a sus usuarios un desarrollo del código rápido y específico.

En 2009, fue asignado el desarrollo a Willow Garage, que ha continuado desarrollando y actualizando las bibliotecas de forma gratuita y abierta. Se puede descargar y modificar en https:/code.ros.org/gf/project/ros (la última visita de junio de 2012).Como sus propios creadores dicen, ROS se puede definir como: Un conjunto de bibliotecas y herramientas para ayudar a los desarrolladores de software a crear aplicaciones robóticas. Se proporciona una guía del hardware y los controladores de dispositivos, las bibliotecas, visualizadores, paso de mensajes, gestión de paquetes, etc... en código abierto.

ROS se puede resumir con tres características fundamentales:

- Meta-Sistema operativo
- Marco de software de robótica
- Arquitecturas robóticas

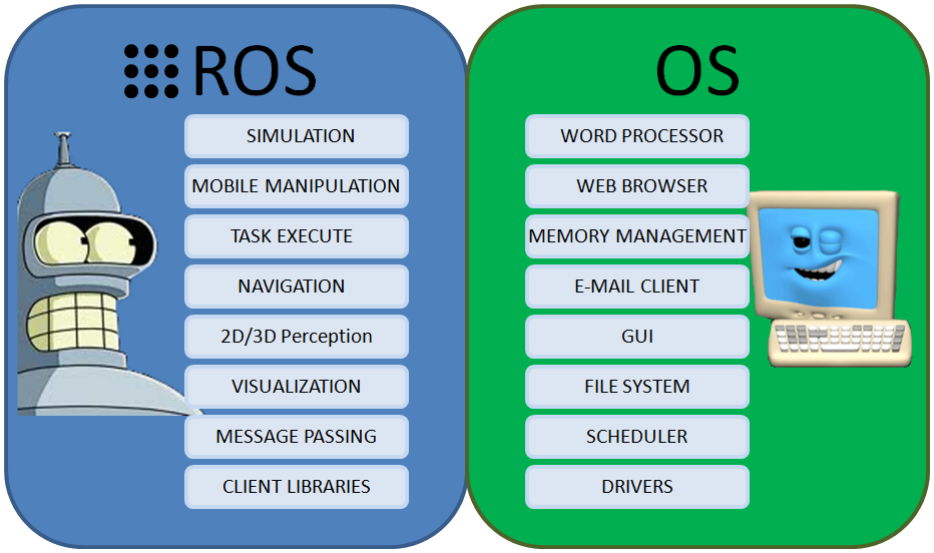


Figura 3.5: ROS Vs OS. ROS está diseñado para trabajar a bajo nivel en consonancia con el Sistema Operativo. Las operaciones de bajo nivel se realizan por ROS para que los usuarios sólo tengan que pensar en aplicaciones de alto nivel

ROS es totalmente compatible con PCL y nos permite realizar los tutoriales de PCL al modo de programación de ROS de tal forma que se puedan generar posteriormente nodos con diferentes funciones.

3.5. Manfred

La idea de este proyecto y otros que se están desarrollando en paralelo, era realizar la programación del sistema Primesense para instalarlo posteriormente en el robot "MANFRED" de la Universidad. En lo que compete a este proyecto se ha realizado una parte de la visión artificial de dicho robot, en concreto la parte de reconocimiento del entorno, es decir, de objetos y planos, en estos momentos también se está programando la mano artificial que se instalará en el robot, y la parte de "tracking" de la mano con visión artificial y con herramientas software similares a las que se han empleado en este proyecto.

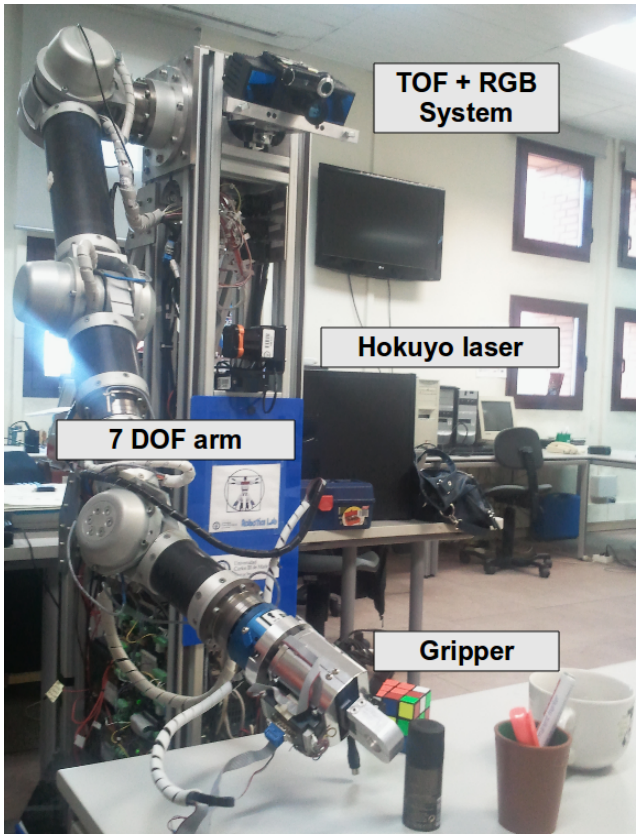


Figura 3.6: Robot MANFRED de la Universidad Carlos III de Madrid

”MANFRED” es un manipulador autónomo antropomórfico, avanzado, fiable y seguro. ”MANFRED” nació en 2001 en el departamento de Sistemas y Automática de la Universidad Carlos III de Madrid, convirtiéndose en un robot pionero en España. Primeramente se realizó el diseño y ensamblado mecánico, centrándose en el brazo, a esta parte se le dedicó 2 años. A partir del año 2003 los trabajos realizados se han centrado más en la parte de software del robot en diferentes ámbitos.

El diseño de ”MANFRED” le permite desenvolverse por entornos controlados (instalaciones como pasillos, salas o despachos) de manera autónoma. En dicho entorno, puede encontrase ciertos obstáculos como puertas, objetos, etc...Y por eso el robot tiene un diseño que integra todas las capacidades básicas de un robot móvil para moverse de forma segura y autónoma, una coordinación motora entre la base y el brazo manipulador, y la coordinación sensorial para manipular objetos.

Para la integración de la programación de la visión artificial en el robot ”MANFRED” será necesario o al menos conveniente, utilizar la herramienta ROS, por la fiabilidad, seguridad y funcionalidades que nos aporta dicho sistema.

Capítulo 4

Experimentación

Para la realización de nuestro sistema hemos seguido los siguientes pasos que son necesarios para el correcto funcionamiento de la aplicación, y que son operaciones que procuran optimizar la manera de reconocer los planos y los objetos:

4.1. Adquisición de datos

Lo primero que hace nuestro sistema es tomar la adquisición de los datos por parte del sensor Kinect, este proceso se realiza con PCL utilizando internamente la herramienta OpenNI. OpenNI nos aporta los datos de profundidad y RGB pero es PCL con VTK el que los ordena de tal forma que los datos se pueden visualizar como nube de puntos y trabajar de esta forma con las herramientas propias de PCL.

4.2. Filtrado

Tras la adquisición de datos lo que se hace es filtrar estos, para mejorar los resultados de los algoritmos que vienen después, pero antes de filtrar hemos decidido desechar los datos RGB de la nube para trabajar con independencia de la iluminación y también aligerar el proceso al manejar menos datos.

4.2.1. Filtro Threshold

El primer filtro que hemos utilizado es el Passthrough que no es más que un filtro que umbraliza la dimensión o dimensiones que le introduzcas, en nuestro caso hemos umbralizado la profundidad, para eliminar datos que están fuera del rango fiable y aumentar también la velocidad del proceso.

```
   //threshold
pcl::PassThrough<pcl::PointXYZ>pass;//Declaramos el filtro para el tipo
de nube utilizamos
pass.setInputCloud (cloud);//Nube de entrada
pass.setFilterFieldName ("z");//Variable que umbralizamos, en nuestro caso
la profundidad
pass.setFilterLimits (0.5, 1.5);//threshold
pass.filter (*cloud_thr);//Nube filtrada
```

4.2.2. Filtro Voxelgrid

El segundo filtro se hace sobre la nube de puntos ya filtrada por el Passthrough o filtro de umbralización, y es el filtro Voxelgrid, en nuestro caso los Voxel tienen un tamaño de 0.005 esto quiere decir que los Voxel que luego hagan el centroide de los puntos que engloban es pequeño pero lo sufucientemente grande como para poder filtrar los puntos y en parte normalizarlos, además de que lleva consigo también un aumento de la velocidad del proceso.

```
   //Voxelgrid
pcl::VoxelGrid<pcl::PointXYZ>sor;//Declaramos el filtro con el tipo de
nube que utilizamos
sor.setInputCloud (cloud_thr);//Introducimos la nube que esta umbralizada
por el anterior filtro
sor.setLeafSize (0.005f, 0.005f, 0.005f);//Elegimos el tamaño de los Voxels
sor.filter (*cloud_filtered);//Nube filtrada
```

4.2.3. Filtro Stadistical Outlier Removal

El tercer filtro lo realizamos justo antes de la segmentación en planos y objetos, y lo realizamos sobre la nube de puntos a la cual se le han realizado los otros 2 filtros, y es Stadistical Outlier Removal. Lo que hace este filtro es descartar los puntos que se salen de una distribución normal de puntos, es decir, puntos atípicos y fuera de norma.

```
   //StatisticalOutlierRemoval
pcl::StatisticalOutlierRemoval<pcl::PointXYZ>sor;//Declaración del filtro
con el tipo de nube
sor.setInputCloud (cloud);//Nube de entrada
sor.setMeanK (50);//Número de vecinos que búscamos
sor.setStddevMulThresh (1.0);//multiplicador de la desviación estándar
sor.filter (*cloud);//nube filtrada
```

4.3. Segmentación y extracción de planos y objetos

Lo siguiente que hacemos es el algoritmo RANSAC que lo que hace es comparar un modelo preestablecido de plano por toda la nube de manera iterativa, y si cumple las características requeridas es un plano. Si el algoritmo dice que es un plano, le introducimos una restricción de tamaño, es decir, que un plano pequeño no será considerado como plano, y pasará al algoritmo siguiente. Si no lo detecta como plano pasa al algoritmo siguiente.

```
   //Segmentación RANSAC
pcl::SACSegmentation<pcl::PointXYZ>seg;
seg.setOptimizeCoefficients (true);
seg.setModelType (pcl::SACMODEL_PLANE);//modelo del plano en RANSAC
seg.setMethodType (pcl::SAC_RANSAC);//método de segmentación
seg.setMaxIterations (1000);//número de iteraciones
seg.setDistanceThreshold (0.01);//distancia máxima permitida entre el punto
y el modelo
```

```
   //Extracción de planos con los inliers
pcl::ExtractIndices<pcl::PointXYZ>extract;
extract.setInputCloud (cloud);
extract.setIndices (inliers);
extract.setNegative (false);
extract.filter (*cloud_p);//Nube con el plano

   //Extracción de lo que no es un plano
extract.setNegative (true);//elegimos que realice el negativo de la nube
de los planos.
extract.filter (*cloud_f);//nube con los puntos que no son planos.
```

Es entonces cuando se realiza una búsqueda de los puntos con kdtree para separarlos en Clusters atendiendo a que los puntos componentes de dicho cluster estén a una distancia Euclidea determinada.

```
pcl::EuclideanClusterExtraction<pcl::PointXYZ>ec;//método de segmentación
de objetos
ec.setClusterTolerance (0.02);//máxima distancia entre puntos para ser
de la misma nube 2cm
ec.setMinClusterSize (100);//mínimo número de puntos de un cluster para
ser considerado objeto
ec.setMaxClusterSize (3000);//máximo número de puntos de un cluster para
ser considerado objeto
ec.setSearchMethod (kdtree);//método de búsqueda elegido kdtree
ec.setInputCloud(cloud);
ec.extract (cluster_indices);//extracción de los índices de los objetos
```

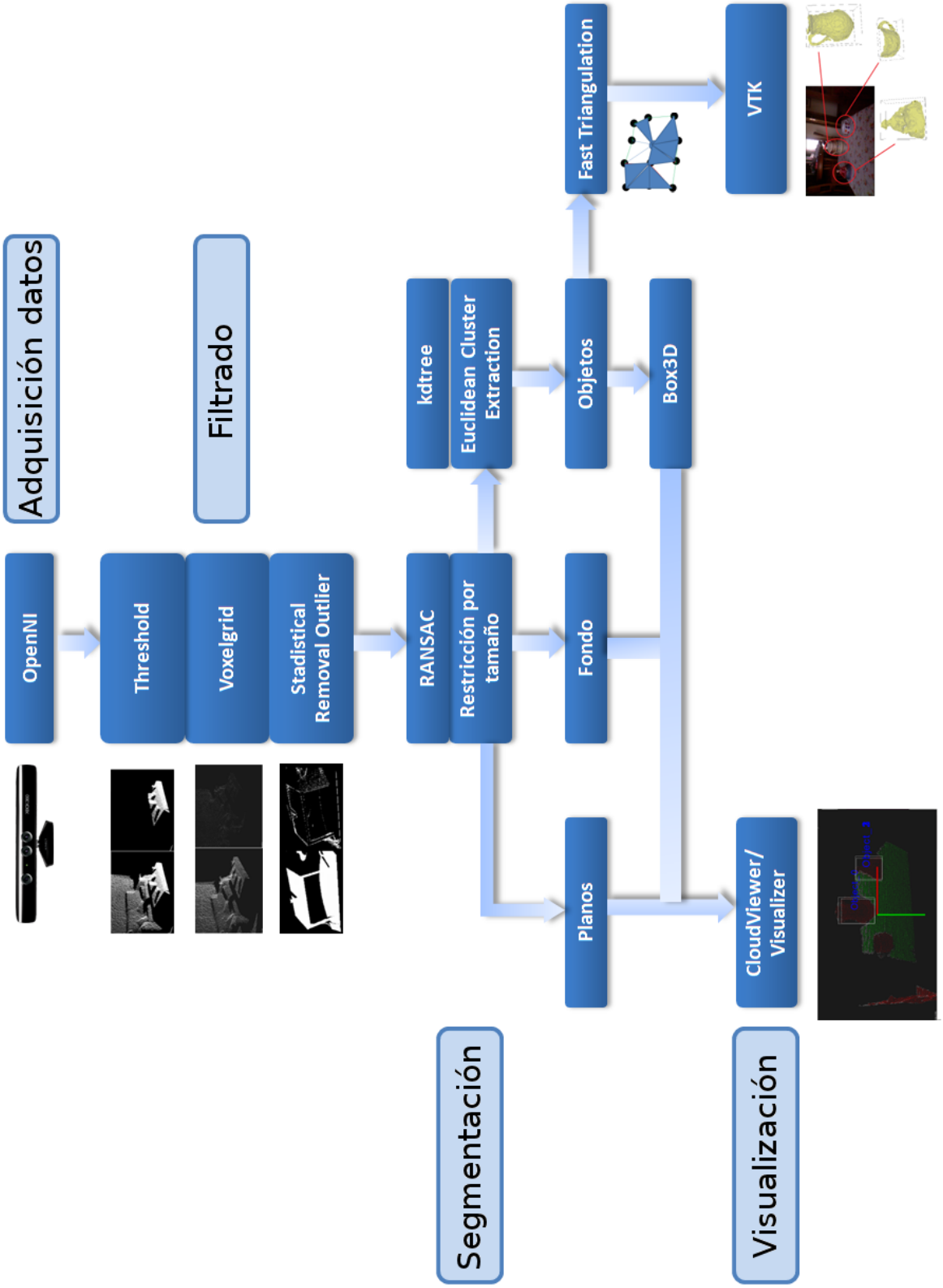
Una vez detectados los objetos se realiza el cálculo de los coeficientes para generar los cubos de etiquetado. Una en vez englobado los objetos en un cubo cada uno (con ciertas restricciones en tamaño del objeto para ser englobado), se realiza el etiquetado por nombre con un texto3D.

Hecho esto se realiza la visualización en PCL de la nube segmentada por color, siendo lo verde los planos y lo rojo los objetos. Para la visualización utilizamos las herramientas PCL_Visualizer junto con CloudViewer.

Por otra linea de ejecución se realiza la triangulación de los puntos y unión por segmentos para su posterior reconstrucción superficial, que visualizaremos con VTK de manera independiente al visualizador de PCL.

```
gp3.setInputCloud (cloud_with_normals);//Introducimos la nube con los datos
de las normales a los puntos
gp3.setSearchMethod (kdtree);//búscamos los puntos con kdtree
gp3.reconstruct (triangles);//reconstruimos la superficie del objeto
```

El proceso que sigue nuestro sistema de detección de planos y objetos se puede visualizar en el siguiente esquema:


Adquisición datos
Filtrado
OpenNI
Threshold
Voxelgrid
Stadistical Removal Outlier
RANSAC
Restricción por tamaño
kdtree
Euclidean Cluster Extraction
Objetos
Box3D
Fast Triangulation
VTK
Fondo
Planos
CloudViewer / Visualizer
Segmentación
Visualización

Capítulo 5

Análisis de los Resultados

En los inicios de este proyecto la velocidad de ejecución era demasiado lenta como para llevarlo a cabo en tiempo real, por ello empezamos a realizar el proyecto en nubes de puntos estáticas con la extensión PCD (Point Cloud Data) utilizada en PCL, el código solo nos permitía la detección de un plano y un objeto a la vez, completándose la detección múltiple, tanto de planos como de objetos, mediante lentas iteraciones de un mismo algoritmo que almacenaba las detecciones anteriores en archivos PCD.

La optimizacón de los algoritmos que llevaban a cabo todo el proceso de ejecución del proyecto en nubes de puntos PCD (Point Cloud Data), nos permitió primero la detección múltiple de planos y objetos a la vez sin necesidad de almacenar los datos anteriores en archivos PCD que, dicho sea de paso, ralentizaban mucho el proceso, y después la visualización de los cubos, no sin ciertos problemas que explicaremos más en detalle a continuación.

La simplificación de algunas partes de nuestros algoritmos nos llevo a comenzar con pruebas en tiempo real, de las que obtuvimos buenos resultados, excepto por el hecho de que no nos permitía el movimiento del sensor Kinect sin salirse de la ejecución.

Cambiando el código del algoritmo de Inscripción de los Objetos en cubos, conseguimos aumentar la velocidad del proceso permitiéndonos por primera vez el movimiento del sensor Kinect, siempre y cuando dicho movimiento no sea demasiado brusco.

5.1. Experimento 1. Detección de planos y objetos

La detección de todos y cada uno de los planos y objetos, que se encontran en una nube de puntos correspondiente a un entorno de una complejidad moderada, llevaba una carga de proceso que ralentizaba el grueso del programa.

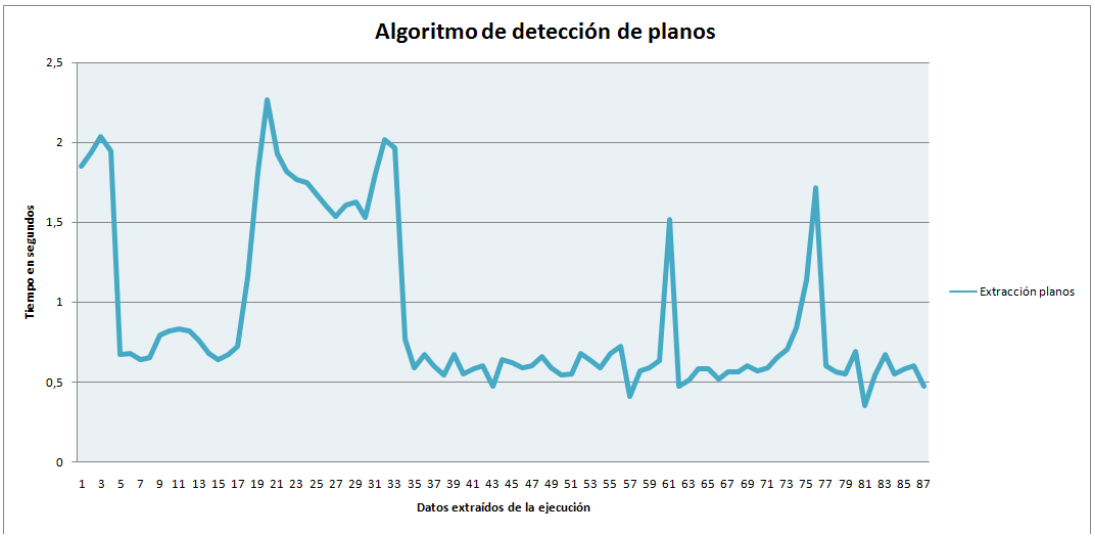
Este motivo nos obligó a estudiar la posibilidad de reducir los datos de entrada pasando la nube por filtros y de este modo el algoritmo tardaría menos en realizar la detección de los planos y objetos, debido a que tendría que procesar menos puntos. No obstante el hecho de procesar menos puntos conlleva otros problemas como la disminución de la uniformidad en la adquisición de datos y como consecuencia de esto, el aumento de la sensibilización a la pérdida de puntos, es decir, que perder un punto en una nube de puntos muy filtrada puede ser crítico para su clasificación.

Un apoyo importante y decisivo para la elección en el porcentaje de puntos filtrados fue la utilización de métodos eurísticos, es decir, mediante prueba y error, por los cuales tomamos las decisiones pertinentes para buscar un equilibrio entre la velocidad de procesamiento y la sensibilidad ante el filtrado de puntos.

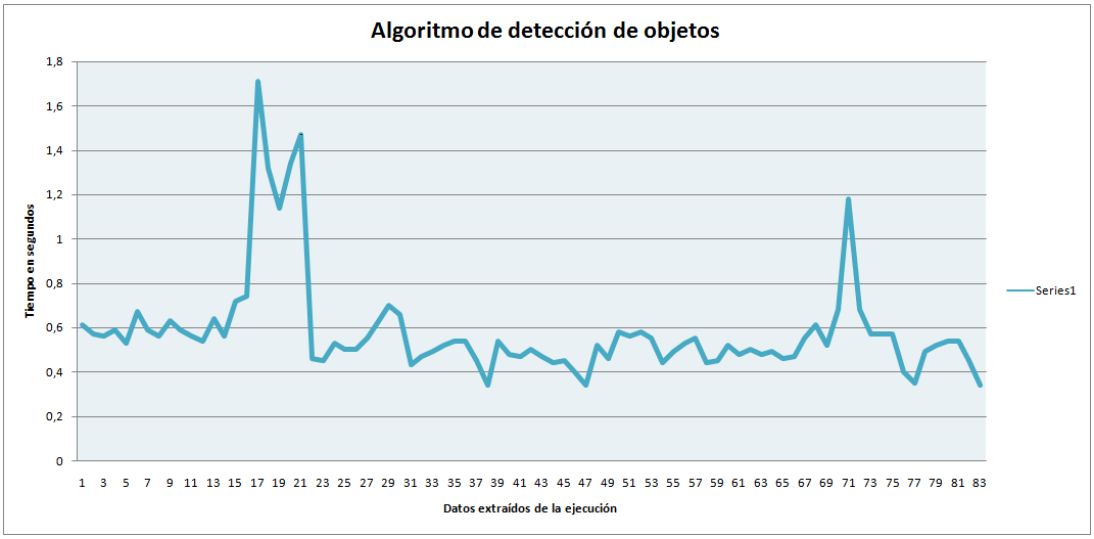
Además del filtro de puntos citado anteriormente decidimos introducir dos filtros más que a pesar de aumentar la velocidad de procesamiento, no comprometían la sensibilidad, aumentando también la fiabilidad de la detección, que eran un filtro Estadístico de ”Outliers” para eliminación del ruido que proviene de la toma de datos de la Kinect y un filtro de profundidad o mejor dicho una umbralización de profundidad.

La umbralización de profundidad decidimos introducirla porque el error en la detección de planos y objetos aumenta con la profundidad, y porque consideramos que por las características de este proyecto, cuyo fin último es la manipulación de objetos por parte de un robot, no era necesario detectar planos y objetos que por distancia no estuvieran al alcance del robot en ese momento.

El tiempo que consumen estos algoritmos se muestran en la imagen 5.1(a) y 5.1(b).



(a) Tiempo que tarda en realizar las iteraciones del algoritmo de detección de planos



(b) Tiempo que tarda en realizar las iteraciones del algoritmo de detección de objetos

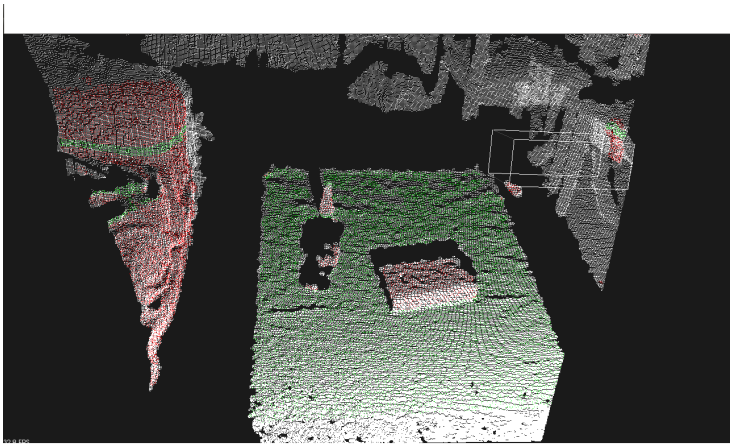
Figura 5.1: Tiempos de los algoritmos de segmentación en planos y objetos

5.2. Experimento 2. Visualización de la segmentación

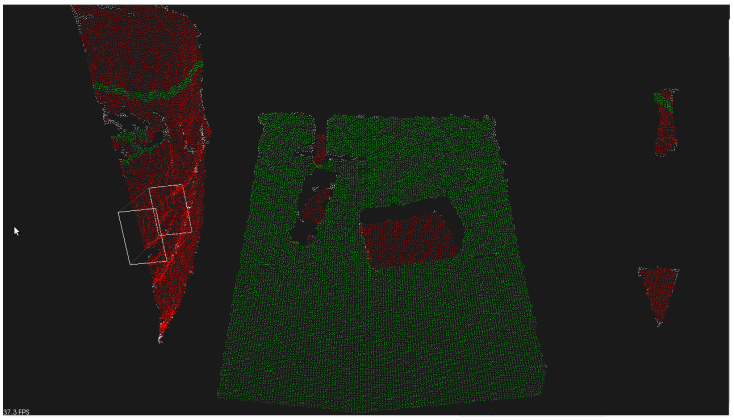
Cuando convertimos el código para ejecutar el programa en tiempo real, uno de los problemas surgió en la visualización del color asociado a la clasificación en planos (de color verde) y objetos (de color rojo) debido a que distintos algoritmos trabajaban a la vez sobre los mismos datos, esto fue relativamente fácil solucionarlo clonando la nube de puntos y asociando la nube clonada al algoritmo de visualización, si no se hace esto se genera ruido en forma de ondas concéntricas en la visualización del color.

Si algo hemos aprendido en esta parte del proyecto es que no se debe nunca realizar la visualización de una nube que esta siendo recorrida por otro algoritmo sea el que fuere.

En la visualización en la cual ya habíamos solventado el problema que hemos descrito anteriormente, realizamos ciertas mejoras, ya que la visualización de los colores era tenue, y por motivos no solo de estética decidimos aumentar el número de puntos coloreados en el algoritmo, no sin antes haberlo optimizado para compensar la velocidad de ejecución (figura 5.2).



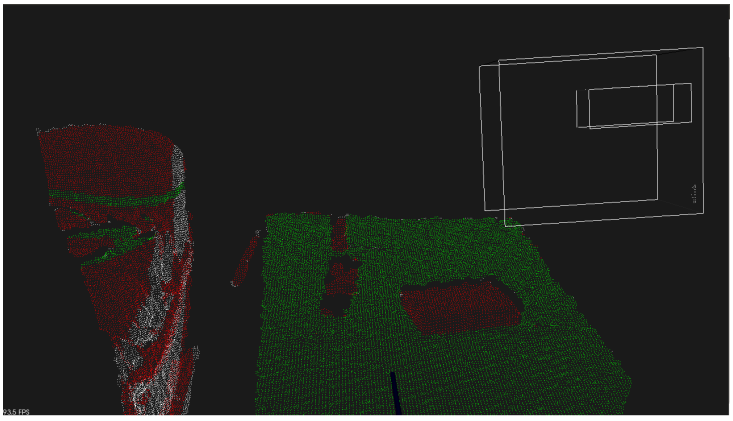
(a) Imagen con problemas de visualización



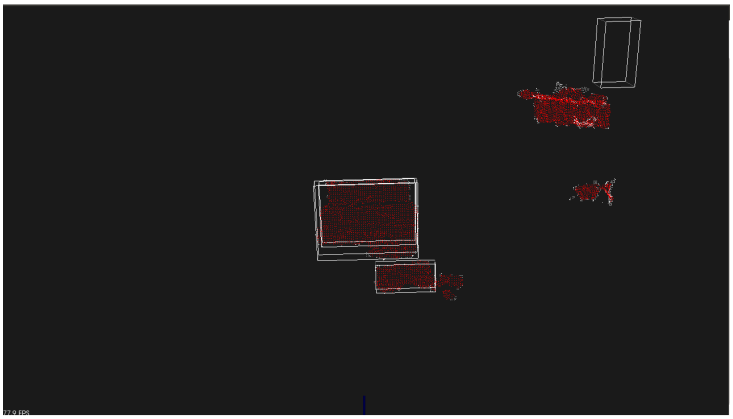
(b) Imagen filtrada y mejorada

Figura 5.2: Mejora y filtrado en el algoritmo de visualización

Otro de los problemas que surgieron en la visualización era la asignación de los coeficientes necesarios para la creación de los cubos en tiempo real. La elección de cuantos y que tamaño de objetos vamos a inscribir, lo decidimos mediante pruebas experimentales (figura 5.2).



(a) Cubos erróneos

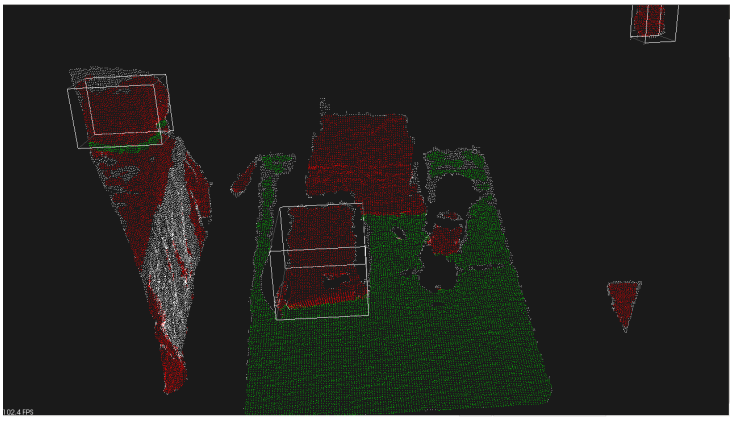


(b) Cubos lentos y limitados

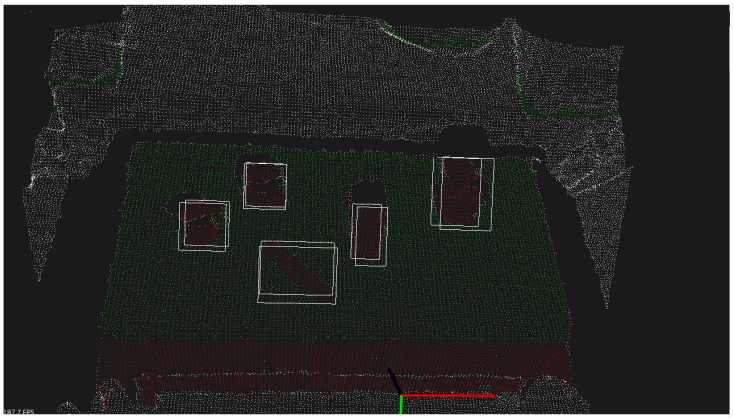
Figura 5.3: Error en el algoritmo que calcula los coeficientes del cubo

En una primera aproximación se consiguió que los cubos englobaran a lo que el programa consideraba como objetos, pero era un poco lento y tenía un número pequeño de objetos que podía reconocer con sus coeficientes incluidos.

Más tarde conseguimos mejorar el algoritmo además de simplificarlo y de esta forma no solo aumentamos su velocidad, sino que además conseguimos que englobara más objetos en el cubo, conciertas restricciones de tamaño, como se muestra en la figura 5.2.



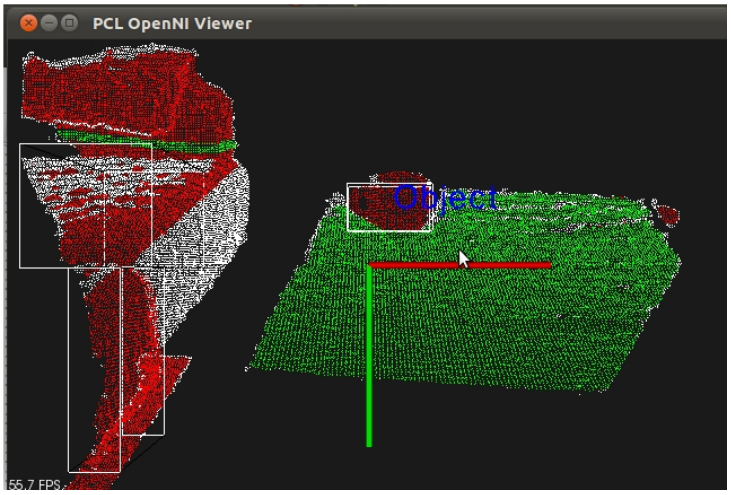
(a) Prueba 1 con el algoritmo mejorado



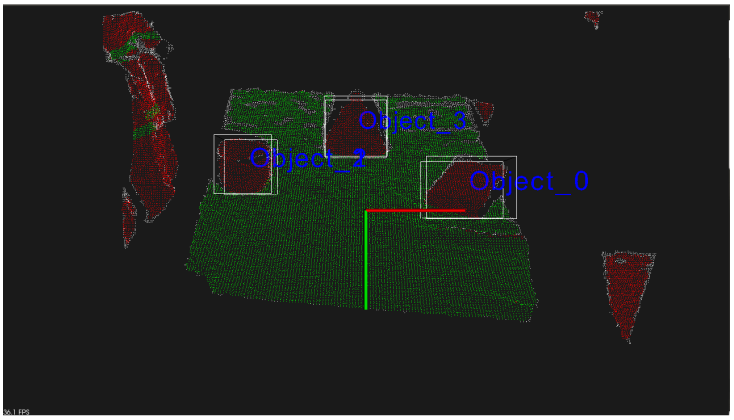
(b) Prueba 2 con el algoritmo mejorado

Figura 5.4: Algoritmo que calcula los coeficientes del cubo mejorado

Tras mejorar el algoritmo que calculaba los coeficientes de los cubos y el que los dibujaba, introdujimos en el algoritmo de visualización un extra que no es otro que el etiquetado numerando los objetos con una función que permite crear un texto en 3D. Aprovechando los coeficientes de los cubos, posicionamos los textos en 3D de la visualización como se muestra en la figura 5.2, no sin antes provar dicho algoritmo en nuestra visualización como se muestra en la siguiente figura.



(a) Prueba 1 con texto 3D



(b) Prueba 2 con texto 3D

Figura 5.5: Etiquetado de los objetos con texto 3D

Un problema genérico en la segmentación que cabe mencionar, es el margen que tenemos en cuenta a la hora de considerar la continuidad en los puntos de un cluster o subgrupo de puntos, es decir, la máxima separación entre dos puntos para considerarlos pertenecientes al mismo cluster. En nuestro caso la máxima distancia permitida entre dos puntos para ser considerados pertenecientes a un mismo cluster, varía dependiendo de si es un plano o un objeto. En el caso de los planos somos más restrictivos y consideramos que si la distancia entre 2 puntos es mayor a 1cm no forman parte del mismo cluster, con los objetos sin embargo somos un poco más permisivos, ya que la forma puede variar mucho de un objeto a otro, y no sabemos como van a estar distribuídos sus puntos, en este caso toleramos una distancia de 2 cm. Este hecho lleva consigo ciertas limitaciones, como que basta que un punto de un objeto determinado este a 2 o menos centímetros de un punto de otro objeto distinto, para que los considere el mismo objeto. Este ejemplo que acabamos de poner no es del todo cierto, ya que para que dos objetos distintos sean considerados como uno solo tienen que suceder otras condiciones que burlen en parte el algoritmo de clustering. A continuación se muestra el error producido en la figura 5.6.

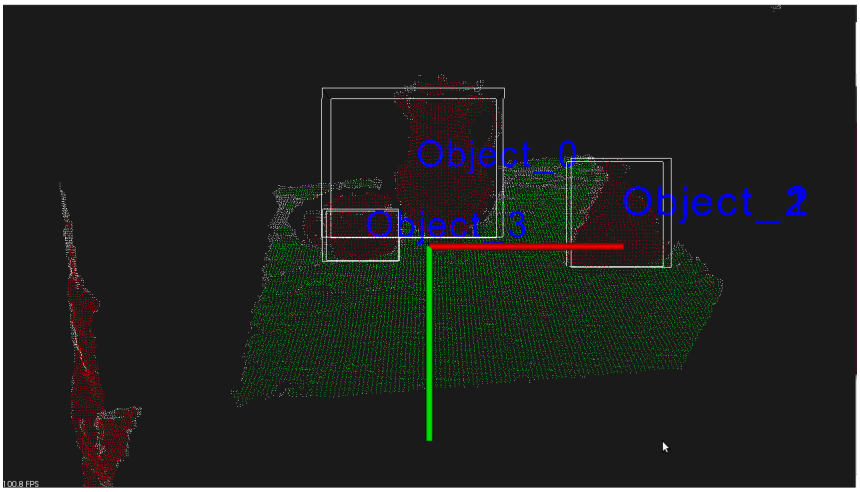
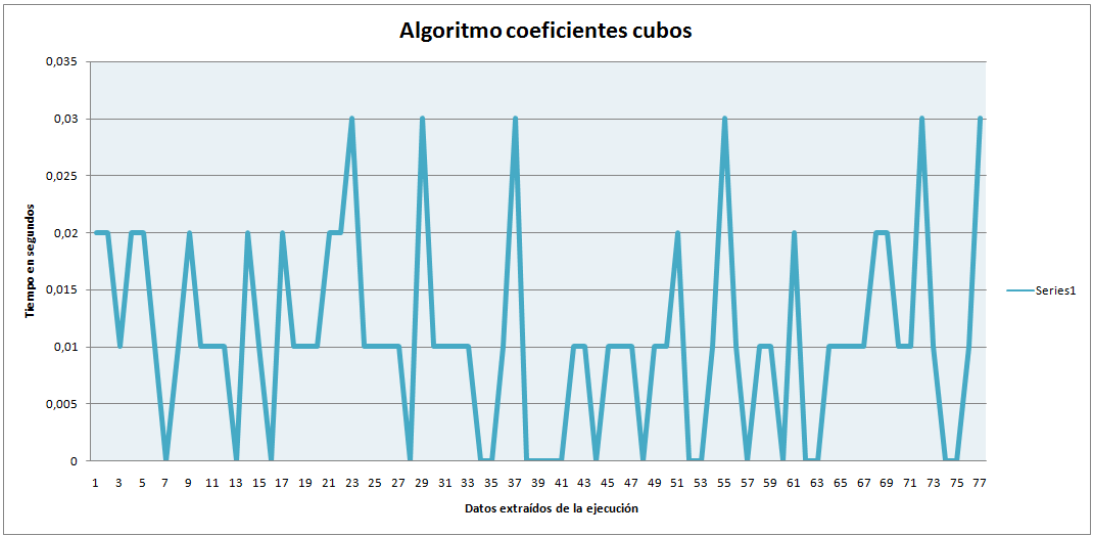
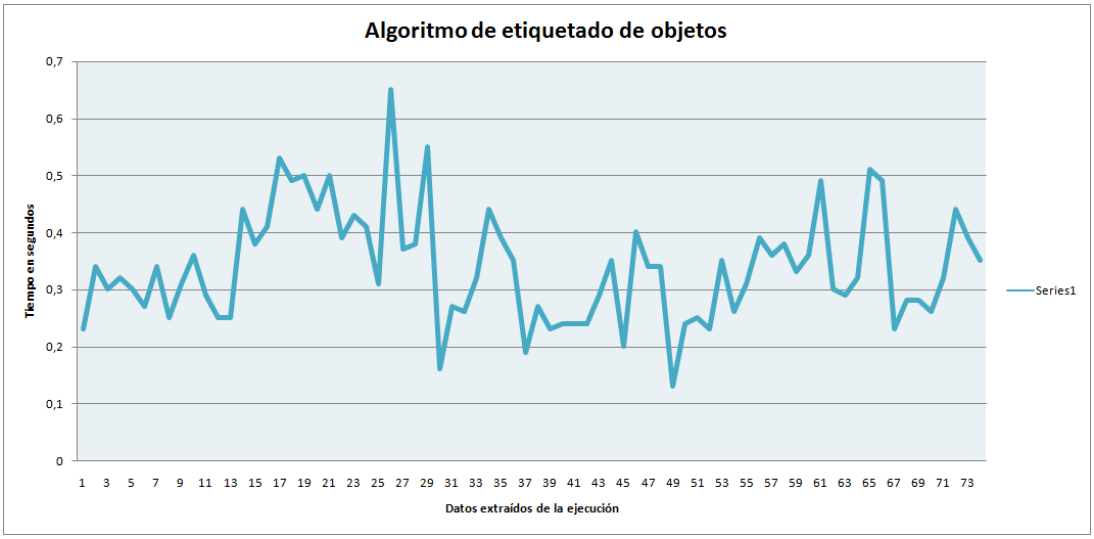


Figura 5.6: Error por violación de condición de proximidad entre objetos

El tiempo que consumen los algoritmos de cálculo de los coeficientes de los cubos y de etiquetado de objetos se muestran en las siguientes figuras 5.7(a) y 5.7(b):



(a) Tiempo que tarda en calcular los coeficientes de los cubos



(b) Tiempo que tarda en realizar el etiquetado de los objetos

Figura 5.7: Tiempos de los algoritmos de etiquetado de objetos

5.3. Experimento 3. Movilidad del sensor Kinect

El origen de este problema residía, de manera genérica,en que, a pesar de haber solucionado los problemas anteriores que nos impedían utilizar el programa en tiempo real, aún era demasiado lento como para asumir cambios demasiado bruscos en el entorno, que son los que se producen si movemos el sensor Kinect. Ya que, por motivos internos del algoritmo de detección de planos, si realizamos un movimiento del sensor, producimos como consecuencia, un cambio relativamente brusco de los puntos de la nube y no le damos tiempo a realizar todas las iteraciones que hemos predispuesto en el frame actual, ya que si tarda menos el programa en cambiar de frame que en realizar las operaciones pertinentes en dicho frame, se produce un error que nos hace abortar la ejecución del programa.

Aparentemente es fácil de solucionar, en teoría bastaría con reducir el número de iteraciones que realiza el algoritmo, pero una vez más nos encontramos inmersos en un tira y un afloja entre la velocidad del algoritmo y la fiabilidad de los resultados de este. Por esto nos vemos obligados de nuevo a recurrir a métodos eurísticos, que por lo general suelen dar muy buenos resultados.

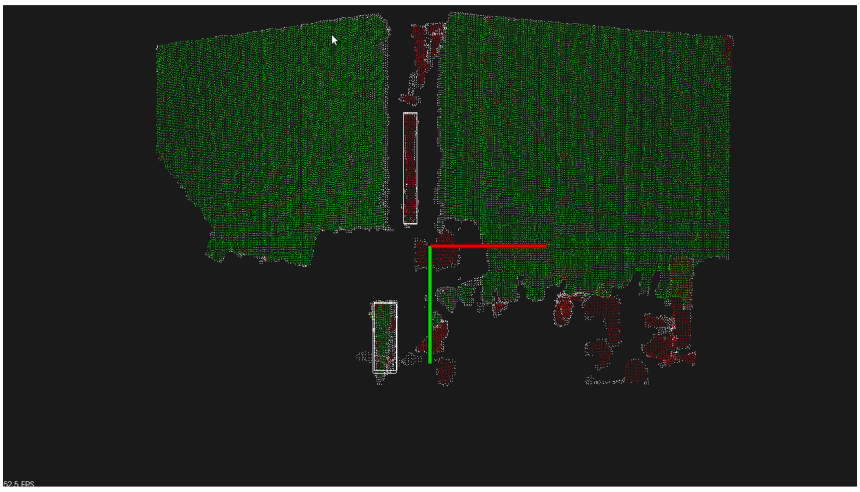


Figura 5.8: Fallo en la detección por movilidad del sensor

Si observamos los datos de tiempos que hemos extraído, podemos observar que hay una zona en la cual todos los algoritmos sufren un retraso temporal significativo con respecto al resto de datos, esto se debe a que en ese instante se estaba realizando un movimiento del sensor Kinect y esto aumenta la carga de proceso de manera importante.

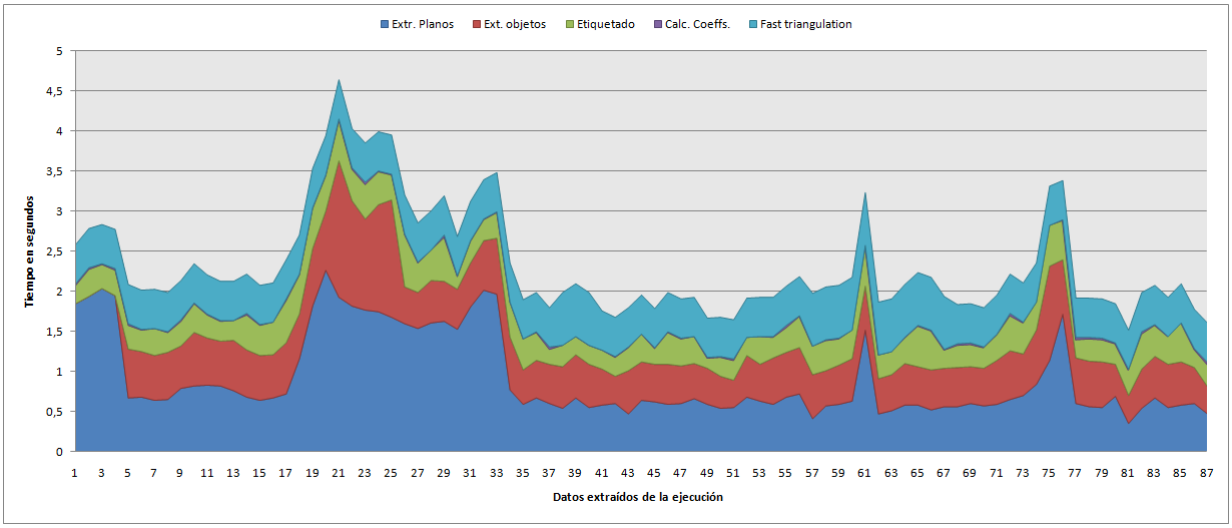


Figura 5.9: Tiempos de todos los algoritmos de nuestro sistema

5.4. Experimento 4. Fast triangulation

La introdución de este algoritmo aumenta significativamente la carga de proceso del programa total y por tanto aumenta el tiempo de ejecución de este. No obstante, dadas las características de este proyecto que tiene como finalidad última el reconocimiento de los objetos, consideramos que era importante introducir dicho algoritmo.

En las siguientes imágenes mostramos los problemas con los que nos hemos encontrado en este algoritmo y algunos procedentes de los algoritmo anteriores, como el error de proximidad que produce un error derivado a la hora de generar los descriptores.

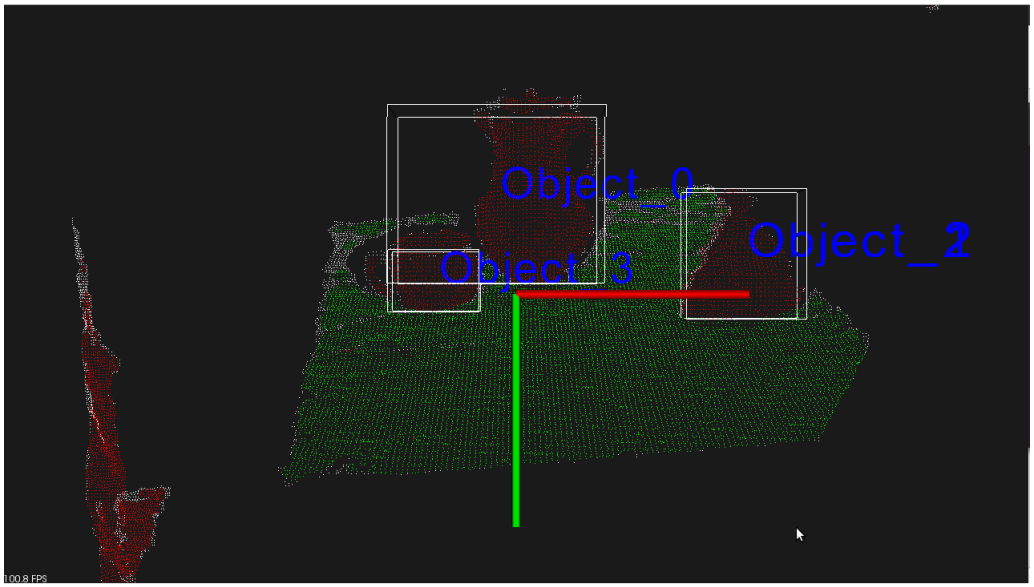
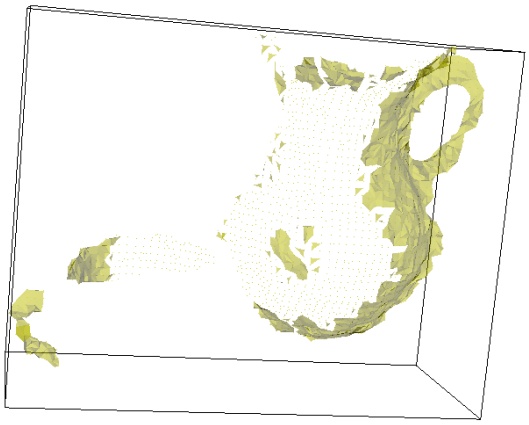
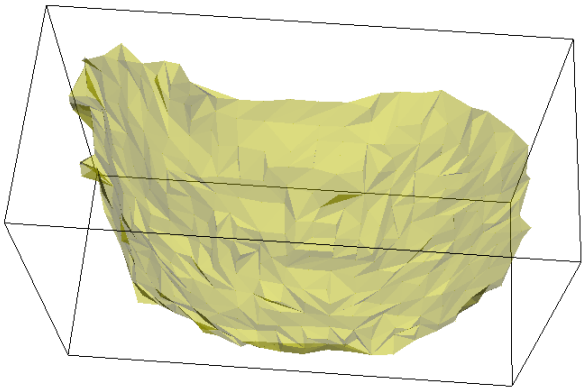


Figura 5.10: Ejemplo de error de proximidad



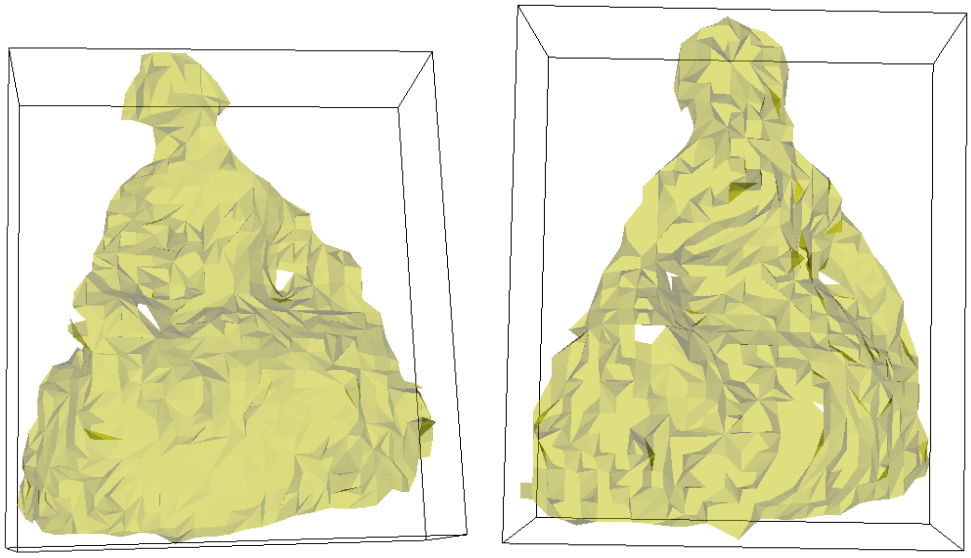
(a) Descriptor de la jarra erróneo



(b) Descriptor de taza parcial debido a la proximidad a la jarra

Figura 5.11: Error al saltarse la restricción de proximidad entre objetos de 2cm

En la figura 5.4 se muestra un error procedente de las limitaciones del sensor Kinect y de las restricciones de los algoritmos. Primeramente debemos decir que el error del sensor Kinect (como ya hemos explicado en las condiciones técnicas del sensor) es de 1 cm a los 2 metros de distancia, a esto debemos sumarle que en el algoritmo de reconocimiento se ha dado una restricción de proximidad para ser reconocidos 2 puntos como del mismo Cluster de 2cm.



(a) Objeto a 1,5 metros del sensor Kinect

(b) Objeto a más de 2 metros de distancia del sensor Kinect

Figura 5.12: Pérdida de precisión en el modelo cuando el objeto sobrepasa los 2 metros de distancia del sensor Kinect

En la figura 5.4 se muestra la pérdida de puntos y por tanto de resolución que se produce al alejar los objetos del sensor Kinect. Para obtener un buen modelo, el objeto debe estar a una distancia de entre 0,8 metros y 2,5 metros, y además tener suficientes puntos como para poder definir bien la aproximación de la reconstrucción superficial por lo que se debe tener precaución a la hora de filtrar si lo que interesa es un buen modelado.

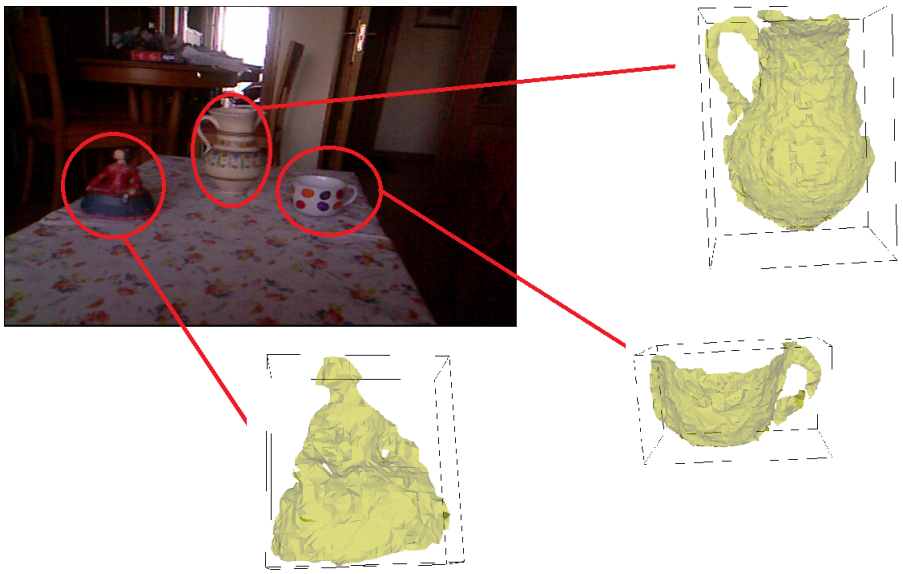
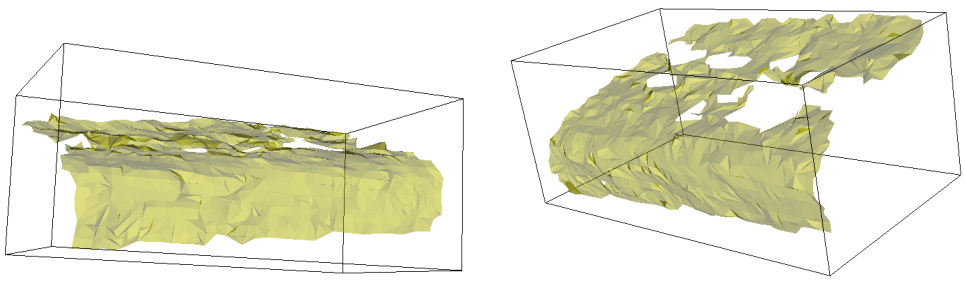


Figura 5.13: Descriptores con Fast triangulation y VTK

En la figura 5.13 se muestra el resultado de aplicar el algoritmo de fast triangulation a partir de los datos del entorno.

En la siguiente figura se muestra uno de los errores que tienen los sensores Primesense y es que en superficies reflectantes o pulidas rebotan los infrarrojos y sufrimos una pérdida de datos importante. También sucede que para superficies transparentes los infrarrojos atraviesan esa superficie o se modifica su trayectoria de tal forma que sufrimos pérdida de datos. En las esquinas o superficies puntiagudas también sucede este hecho.



(a) Descriptor de un libro

(b) Descriptor de un libro con pérdida de puntos

Figura 5.14: Rebote y absorción de infrarrojos sobre superficies

Los datos de tiempo que hemos extraído de este algoritmo son los siguientes:

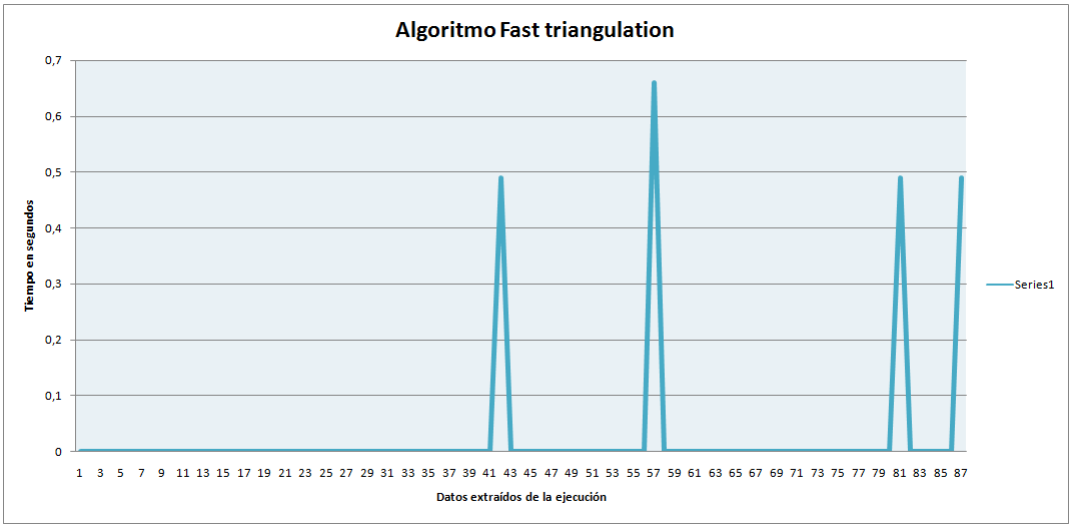


Figura 5.15: Carga de proceso temporal del algoritmo fast triangulation

A continuación se puede observar una gráfica con el tanto por ciento de la carga de proceso que llevan cada uno de los algoritmos utilizados en el proceso. Puede observarse que el algoritmo que más tarda en realizarse es el de extracción de planos, aunque este algoritmo solo se realiza si el plano que ha sido detectado sufre una pérdida de puntos superior al 70 %. El segundo algoritmo que más tarda es el de reconocimiento de objetos ya que hace una búsqueda iterativa con kdtree de los puntos de los cluster y los selecciona por distancia Euclídea. Puede reducirse el tiempo de procesamiento filtrando más puntos, pero empeoraría el algoritmo de Fast Triangulation.

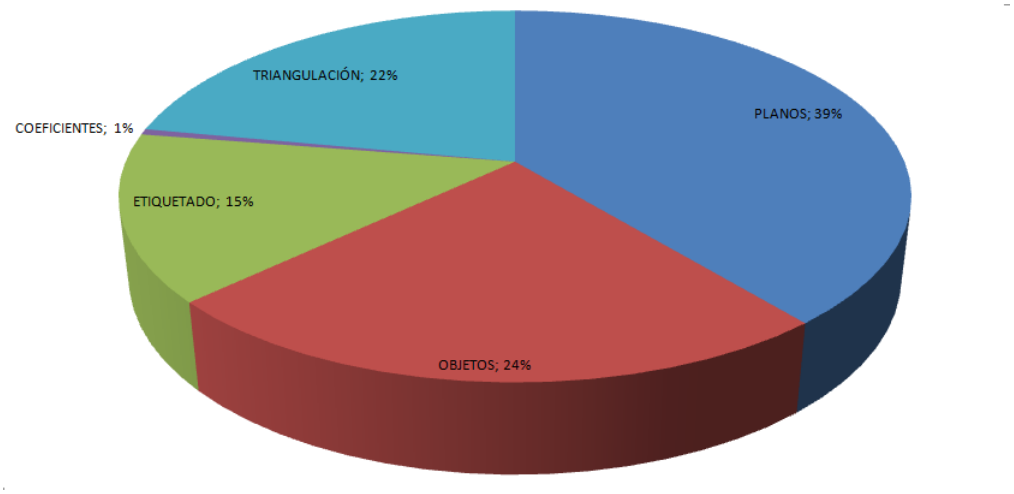


Figura 5.16: Distribución de carga del proceso entre los distintos algoritmos

Capítulo 6

Conclusiones

Hemos desarrollado un sistema capaz de:

- Filtrar una nube de puntos sin perder información relevante, eliminando los puntos demasiado alejados del sensor, normalizando los puntos y filtrando un tanto por ciento de estos, y encontrando y quitando aquellos puntos denominados como outliers.
- Segmentar el entorno siendo capaz de reconocer aquellas partes que forman parte del entorno y aquellas que son planos u objetos, realizando varias búsquedas con diferentes métodos iterativos. Tardando un tiempo medio de 1 segundo para los planos, y de 0,6 segundos para los objetos, pero en el caso de los planos para movimientos no demasiado bruscos el algoritmo solo se realiza 1 vez, ya que está diseñado para reconocer los planos anteriormente detectados hasta para una pérdida del 70 % de los puntos que tenía el plano originamente.
- Etiquetar objetos envolviéndolos en cubos, cuyos parámetros hemos calculado, y asignándoles un nombre que los diferencie de los demás. Tardando un tiempo medio de 0,3 segundos para todos los objetos que detecte en la fase anterior.
- Realizar la visualización de todo lo anterior de una forma amigable, ya que el usuario puede distinquir a simple vista los objetos y los planos, por el color, y los objetos entre sí por las etiquetas.
- Reconstruir la superficie de los objetos en el momento que se lo digamos con un algoritmo que se maneja en tiempo real. Tardando un tiempo medio de 0,5 segundos para todos los objetos detectados anteriormente.

- Visualizar la reconstrucción superficial con VTK, para hacernos una idea de que objeto se trata realmente.

Y todo esto se realiza en tiempo real y a diferencia de otros sistemas que son a cámara fija, este sistema permite movimientos del sensor Kinect mientras sigue realizando los algoritmo que hemos descrito anteriormente.

Capítulo 7

Trabajos futuros

En esta parte vamos a explicar algunos desarrollos que se pueden implementar para aumentar las funcionalidades de este proyecto a medio plazo:

- Como trabajo futuro y como medida para reducir el tiempo de ejecución del código se podría implementar en el código del proyecto la utilización de la GPU para la parte de segmentación en planos y con la CPU se haría la parte de reconocimiento de objetos en paralelo, esta medida la ibamos a tomar en un principio pero decidimos no hacerlo debido a que las bibliotecas involucradas en este proceso de PCL se encuentran en la versión de desarrollo que aún es inestable, no obstante en poco tiempo se podrá utilizar con la versión PCL 1.6.
- Como mejora hemos pensado que sería interesante introducir el mapeado y modelado del entorno real con Kinect Fusion de PCL, esta medida reduciría el tiempo de ejecución significativamente, ya que solo tendría que detectar los planos o imagen base una vez, y nos centraríamos en la detección, clasificación y reconocimiento de los objeto, las razones de no haber implementado esto en nuestro proyecto son varias, las más importantes son que la tarjeta gráfica del ordenador en el que se ha realizado el proyecto no soporta Kinect Fusion, y la otra es que Kinect Fusion de PCL aún está en versión de desarrollo y es inestable.
- Para la parte de reconocimiento de patrones o pattern recognition se podría utilizar el algoritmo Cluster Recognition and 6DOF Pose Estimation using VFH(Very Fast Histogram) descriptors, el único incoveniente es que se necesitan gran cantidad de datos previamente

extraídos y clasificados para entrenar el sistema.

- Y para la integración en el Robot **MANFRED** se podría utilizar ROS para aumentar las prestaciones y mejorar la comunicación entre los distintos sistemas.

Bibliografía

[1] Duda, R.O. and Hart, P.E. and Stork, D.G. Pattern classification. *Pattern Classification and Scene Analysis: Pattern Classification*, Wiley, 2001.

[2] Bishop, C.M. *Pattern Recognition And Machine Learning*. Springer, 2006.

[3] Bradski, G. and Kaehler, A. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, 2008.

[4] Szeliski, R. *Computer vision: Algorithms and applications*. Springer-Verlag New York Inc, 2010.

[5] Hemant M. Kakde. *Range Searching using Kd Tree, 2005.*

[6] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, Pat Hanrahan *Interactive k-D Tree GPU Raytracing.*

[7] Marco Zuliani *RANSAC for Dummies, 2012.*

[8] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges and Andrew Fitzgibbon *KinectFusion: Real-Time Dense Surface Mapping and Tracking.*

[9] Anónimo *Initial alignment of point clouds, 2011.*

[10] Ranga Rodrigo *Image Processing and Computer Vision.*

[11] T. Rabbani , F. A. van den Heuvel and G. Vosselman *Segmentation of point clouds using smoothness constraint.*

[12] Rusu, R.B. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments* Institut für Informatik der Technischen Universität München, 2009.

[13] T. Rabbani , F. A. van den Heuvel and G. Vosselman *Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera.*

[14] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison and Andrew Fitzgibbon. *KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera.*