Universitat d'Alacant
Universidad de Alicante

Sistemas de percepción
Grado Ingeniería Robótica

Práctica 2

Detección de objetos tridimensionales en nubes de puntos

Carmen Ballester Bernabeu

Sheila Sánchez Rodríguez

Índice

Carmen Ballester Bernabeu
Sheila Sánchez Rodríguez

Introducción al problema

El problema planteado consiste en el desarrollo de un pipeline de reconocimiento de objetos 3D dentro de una escena. Para ello, se dispone de una nube de puntos que contiene dicha escena y una nube de puntos para cada modelo del objeto, es decir, el objeto segmentado.

El reconocimiento de objetos dentro de una escena, tanto en visión tradicional 2D como en sistemas de visión 3D, es uno de los problemas más comunes y más estudiados dentro de la ciencia de la computación y de la visión artificial. Por lo tanto, se trata de un problema acerca del cual existen numerosos artículos y publicaciones científicas, además de una gran variedad de métodos para realizar el procedimiento y, lo más importante en relación a nuestro desarrollo, muchas librerías que implementan estos métodos y que constan con una documentación y el soporte de la comunidad en foros. En nuestro caso, utilizamos las librerías de la ***Point Cloud Library*** (**PCL**), que implementan todos los métodos necesarios y más para poder completar satisfactoriamente la tarea.

El problema de que exista una gran variedad de métodos para resolver la misma tarea es que algunos funcionan mejor para algunos casos que otros. Además, los métodos suelen tener una gran variedad de parámetros que se tienen que ajustar de forma empírica. Por lo tanto, la parte más compleja del problema, y la que se suele perfeccionar con la experiencia, es la de seleccionar el método y los parámetros adecuados para llevar a cabo una tarea dentro del programa global de reconocimiento.

Podemos descomponer el problema del reconocimiento en los siguientes bloques:
1. **Eliminar** todos los **planos dominantes** de la escena.
2. **Extraer puntos característicos** de la escena y de los objetos.
3. **Calcular los descriptores** para los puntos característicos de la escena y de los objetos.
4. **Computar los emparejamientos** entre los descriptores de la escena y de los objetos.
5. **Rechazar los emparejamientos incorrectos**.
6. **Calcular la transformación** que hay entre la nube de puntos del objeto y el objeto en la escena.
7. **Refinar el resultado** con ICP.

Estado del arte

Una de las principales aplicaciones de la visión por computador es la de reconocer los objetos que nos rodean. Si somos capaces de enseñar a los robots a interpretar el entorno y reconocer cada uno de los objetos que componen una escena, habremos dotado a la máquina de una gran independencia para poder tomar decisiones por sí misma. Por ese motivo hay gran cantidad de estudios entorno al reconocimiento de objetos.

Existen dos tendencias:
- Detección basada en **marcas artificiales**.
- Detección basada en **marcas naturales**.

De estas dos propuestas, vamos a utilizar la detección por marcas naturales, pues aprovecha la información intrínseca que aporta el objeto al ser proyectado en el plano imagen. El inconveniente que presentan respecto a un sistema basado en marcas artificiales es que hay que determinar qué puntos del objeto presentan ciertas características que los convierte en candidatos idóneos para ser detectados en otras imágenes. En esta línea se diferencian 3 métodos principalmente: métodos basados en detección de bordes, métodos basados en ajuste de plantilla y métodos basados en extracción de puntos y sus descriptores invariantes.

Una de las líneas de investigación que se está siguiendo actualmente en este aspecto, está relacionado con las bases de datos. Podemos ver un ejemplo de una gran base de datos en la publicación de Lai [1]. Esta gran base de datos tiene 300 objetos organizados en 51 categorías diferentes.
Tang describe en su investigación como crea una gran base de datos con multitud de datos y posiciones de diferentes objetos cotidianos [2]. Gracias a la enorme cantidad de datos de cada objeto consigue una detección altamente eficaz. El sistema es capaz de manejar correctamente obstrucciones, cambios de iluminación, múltiples objetos y diferentes casos dentro del mismo objeto.
El método propuesto por Kasper habla de la creación de una gran base de objetos para su posterior utilización en reconocimiento y manipulación de objetos [3].

Ahn usa un enfoque para el reconocimiento de objetos cotidianos dentro de una escena [4]. Utiliza un sistema de visión para robots conocido como SLAM (Simultaneous localization and mapping). En este caso lo utiliza para el reconocimiento de objetos. Para aplicar bien el método SLAM debe seguir tres pasos. Primero debe extraer las características invariantes. Después aplica un agrupamiento RANSAC (RANdom SAmple Consensus) para el reconocimiento de objetos en presencia de puntos externos. Por último calcula información métrica precisa para actualizar el SLAM.

Fundamento teórico

Detectores

Para reconocer objetos dentro de una escena es necesario definir los puntos característicos del objeto, que serán los mismo en la escena y en el modelo del objeto. De esta forma, será posible detectar en qué parte de la escena está el objeto. Los punto característicos o ***keypoints*** son puntos de la escena o el objeto que se encuentran en una zona de especial interés como esquinas, bordes, etc.

Por definición, un keypoint tiene que ser **repetible**, es decir, que pueda ser detectado desde cualquier punto de vista que pueda ser capturada la nube de puntos; y **distinguible**, es decir, que pueda diferenciarse del resto de puntos de la escena.

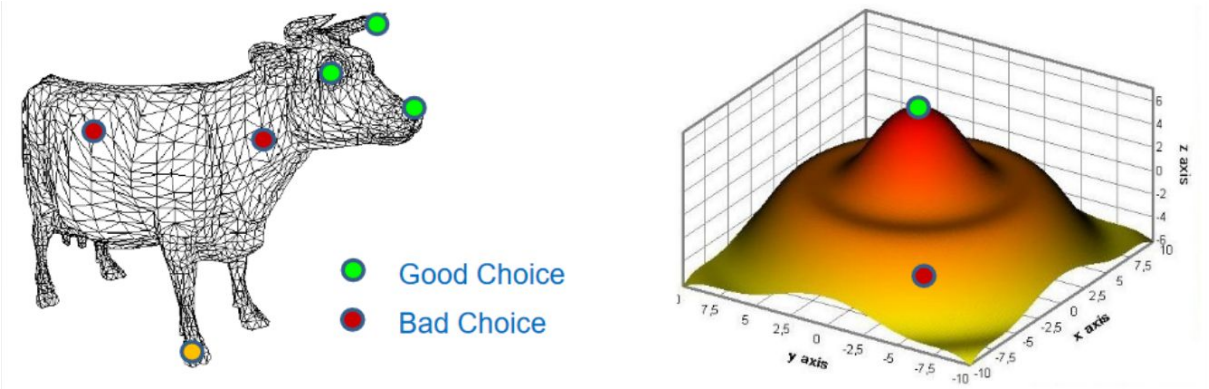


Figura 1. Ejemplos de keypoints

Los **detectores** son métodos de extracción de keypoints. La escala en la que se detecta un keypoint es muy importante. Esta escala se conoce como **escala característica** del keypoint. Hay dos posibles enfoques acerca de la escala característica:

- **Escala fija**: el parámetro de vecindad del keypoint es fijo. Tiene problemas porque normalmente detecta o bien puntos esparcidos y ni representativos o bien puntos cercanos y redundantes.
- **Escala adaptativa**: se detectan los keypoints en varias escalas realizando el análisis de la nube espacialmente. La escala en la que se detecta el keypoint se le asocia. Permite obtener puntos no redundantes que den información adecuada del entorno.

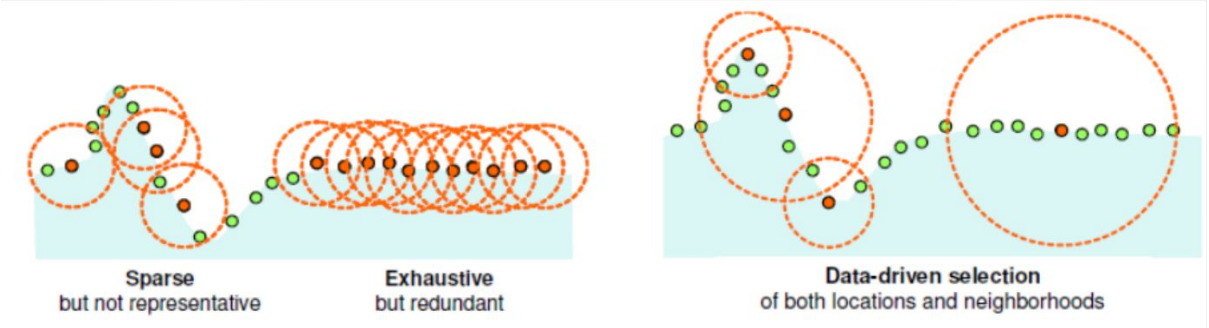


Figura 2. Escalas en la detección de keypoints

La mayoría de detectores requieren del cálculo de las normales en cada punto de la imagen. Para calcularla se utiliza **PCA**, que requiere de la definición de un radio de vecindad. Este parámetro es muy sensible porque depende de la escala y de la dispersión de la nube de puntos. Una vez obtenida la **matriz de covarianza**, que indica la variabilidad del entorno en dicho punto en cada una de las tres direcciones, podemos determinar que un keypoint es aquel punto que presenta mucha variabilidad en dos direcciones. Sin embargo, esto detecta un gran número de puntos característicos, por lo que hay muchos métodos que los filtran utilizando el concepto de **activación del keypoint**.

Algunos de los detectores más utilizados son:

- **Harris 3D**: se basa en la detección de esquinas y aristas, que son las zonas caracterizadas por grandes cambios de intensidad en los ejes. Una vez calculadas todas

las normales suprime el exceso de keypoints similares. Para ello se calcula un valor ***r***, **parámetro de activación**, correspondiente a cada keypoint dentro de una zona determinada. Se escoge el keypoint que mayor parámetro de activación tiene dentro de ese clúster, de forma que los keypoints se distribuyen de forma global y obtenemos mejor información de la nube.

- **SIFT 3D**: está basado en el concepto de curvatura. Se utiliza la **matriz hessiana** con los valores de la curvatura principal. A la nube de puntos se le aplica una serie de convoluciones de filtros gaussianos y se crea el espacio de escala gaussiano para detectar los keypoints cómo diferencia de gaussianas con sus vecinos.
- **SUSAN**: utiliza una técnica genérica de procesamiento de imagen que consiste en examinar una vecindad esférica alrededor del punto y estimar la diferencia entre el núcleo y cada punto de dicha vecindad. Si la diferencia es mayor que un umbral se pone ese punto a valor 1 y si no a 0, para obtener el **área USAN** de la vecindad. Luego este área se compara con un umbral geométrico y se establecen como keypoints los máximos o mínimos de la función de comparación.

Descriptores

Los descriptores son estructuras que codifican información de la geometría de la nube de puntos. Hay una gran cantidad de descriptores, y cada uno es más adecuado para una tarea específica. Por definición, para poder obtener la correspondencia entre dos nubes, los descriptores tienen que ser robustos frente a transformaciones y ruido e indiferente a la resolución.

Los descriptores pueden ser **locales**, si codifican información individual de cada keypoint, o **globales**, si codifican información de la geometría de un objeto en general.

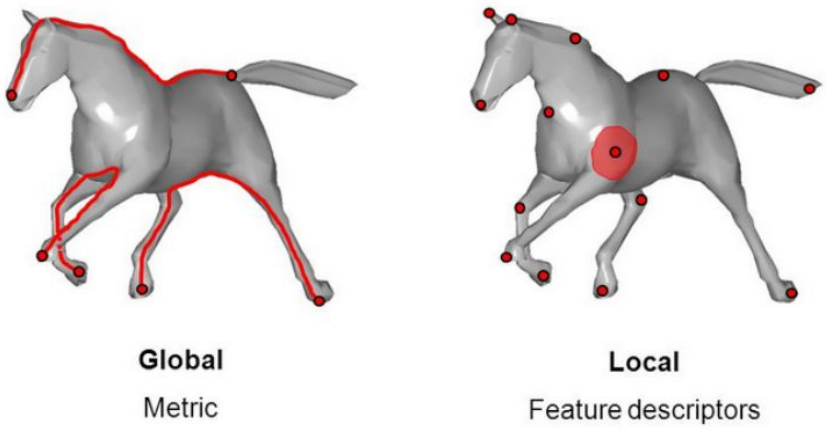


Figura 3. Descriptores de características

Para este problema se utilizan los descriptores locales, que describen la geometría alrededor del keypoint, permitiendo una correspondencia 1:1 entre los puntos de las dos nubes. Normalmente los descriptores se someten a procesos de reducción de tamaño para mejorar la velocidad del algoritmo en tiempo de ejecución.

Los descriptores locales más usados son:

- ***Point Feature Histogram* (PFH)**: analiza la diferencia entre las direcciones de las normales de la vecindad del punto. Empareja el punto central con cada punto de su vecindad y los puntos vecinos entre sí y se calculan los tres valores angulares de la diferencia entre las normales de los puntos y la distancia euclídea. Estos valores se agrupan de forma separada en un histograma, por lo que el descriptor final son cuatro histogramas unidos.
- ***Fast Point Feature Histogram* (FPFH)**: es una variación del algoritmo anterior que permite la ejecución en tiempo real. Considera únicamente las conexiones entre el keypoint y sus vecinos. Para compensar la pérdida de precisión se calculan los descriptores FPFH de los vecinos y se añaden al del keypoint ponderados según la distancia a la que se encuentran.
- ***Intrinsic Shape Features* (ISF)**: es un método de detección de keypoints y extracción de características que se basa en medidas de calidad por regiones. Se calcula la característica de forma del keypoint como un histograma ponderado de las características de forma de la vecindad, definiendo el vector de características 3D como un vector que recoge la forma 3D del punto.
- **3D Shape Context**: crea una malla esférica en el keypoint, que se subdivide equitativamente. El descriptor se define como un histograma que contiene en cada valor la suma ponderada de puntos que caen en el sector asociado a dicho valor.
- **Unique Shape Context**: es una extensión del descriptor anterior que define un marco de referencia local para cada keypoint, de forma que proporcionan direcciones principales que orientan de forma única la malla 3D de cada descriptor, evitando las descripciones múltiples del mismo keypoint.
- **SHOT**: define una estructura esférica que se subdivide equitativamente como soporte de forma que el descriptor se calcula como los histogramas locales de la información geométrica de los putos dentro de un sector. Para un sector, se crea un histograma unidimensional que recoge la acumulación de la cuenta del ángulo entre la normal del keypoint y la de cada punto que pertenece a ese sector.

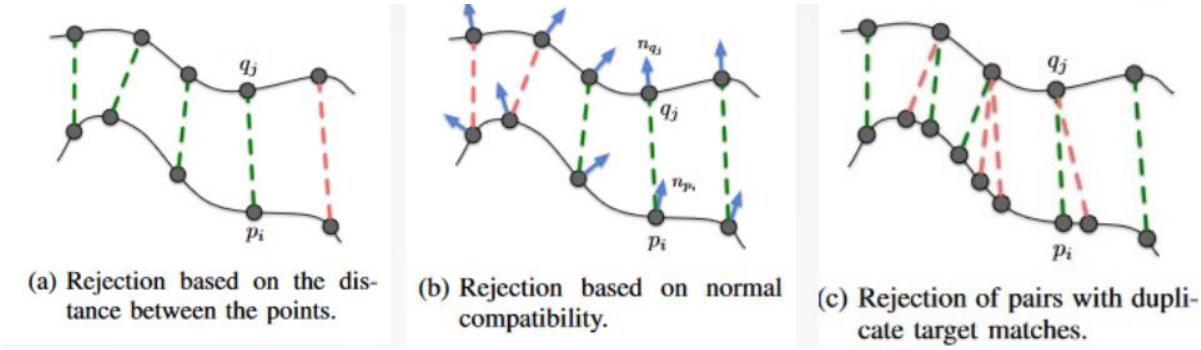
Correspondencias

Una vez obtenidos los vectores descriptores de cada keypoint, tanto en la escena como en el modelo del objeto, hay que definir las correspondencias entre ellos para poder saber qué keypoint del modelo corresponde a qué keypoint de la escena.

Al trabajar con vectores, se utilizan **medidas de distancia entre vectores**, como la distancia euclídea. Para cada descriptor de la escena buscamos el más cercano en el modelo del objeto. El problema es que este método produce falsas correspondencias que es necesario filtrar Utilizando otros métodos.

Rechazo de correspondencias

Existen una serie de filtros para determinar si la correspondencia entre dos puntos es correcta, basados en la coherencia semántica o geométrica, cómo rechazar pares cuya distancia sea mayor que un umbral o aquellos que no tengan correspondencia biyectiva.



(a) Rejection based on the distance between the points.

(b) Rejection based on normal compatibility.

(c) Rejection of pairs with duplicate target matches.

Figura 4. Ejemplos de rechazo de correspondencias

Sin embargo, el método más utilizado para rechazar correspondencias es aquel que descarta los puntos que no son coherentes con la transformación rígida que mejor se aplique a la mayoría de emparejamientos. El mejor método para estimar esta transformación es ***Random Sample Consensus*** **(RANSAC)**.

RANSAC es un **método iterativo** que permite obtener de forma robusta los parámetros de un modelo, en nuestro caso, la transformación entre las nubes de puntos. El algoritmo recibe como entradas:

- El número mínimo de datos requerido para el modelo, *s*.
- El número de iteraciones, *N*.
- El umbral de distancia para considerar si un dato encaja en el modelo, *U*.
- El número de puntos para soportar un modelo, *T*.
- El conjunto de datos, *D*.

El algoritmo consiste en elegir un número *s* de datos del conjunto *D* aleatoriamente y encontrar el modelo *M* que mejor se ajusta a ellos. Para cada dato *d* que pertenece a *D* y no está en *s*, se comprueba si su distancia a *M* está por encima o debajo del umbral *U*. En cada iteración obtenemos dos conjuntos: los ***inliers***, que son los puntos *d* cuya distancia a *M* es menor que *U*, y los ***outliers***, cuya distancia es mayor. Luego se refina el resultado M recalculando *M* solo con los inliers.

En este problema, los datos de entrada son los **pares de características** y la salida es una **matriz de transformación** con seis parámetros: tres coordenadas y tres rotaciones. Para encontrar la transformación, por tanto, hay que encontrar el modelo que minimice la función de error entre el par de puntos, que no es más que la función de mínimos cuadrados:

$$E(R,t) = \frac{1}{N_p}\sum_{i=1}^{N_p} \|x_i - R \cdot p_i - t\|^2$$

Para calcular el modelo se calcula el centro de masas de cada conjunto, que es la media, y se establece a traslación como la **diferencia entre ambos centros**. Para la rotación se calcula la matriz de covarianza cruzada y se le aplica **descomposición SVD**, de forma que la matriz resultante es la rotación:

$$W = \sum_{i=1}^{Np} x_i' p_i'^T \qquad W = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T \qquad R = UV^T$$

Una vez obtenida la transformación utilizando RANSAC, ya podemos descartar los puntos que no son coherentes con la transformación de los emparejamientos.

Figura 5. Descarte de emparejamientos tras aplicar RANSAC

Refinamiento del resultado

Normalmente, las escena y el objeto provendrán de dos capturas diferentes y eso no asegura que exactamente los puntos de una, estén en la otra. Lo normal es que la transformación que nos devuelve RANSAC sea buena, pero no la mejor que se puede obtener. Para ello se hace uso de técnicas de refinamiento como el ICP, el cual se aplica sobre una estimación inicial considerada muy buena, que es la que nos ha proporcionado RANSAC.

El **ICP** (Iterative Closest Point) es un algoritmo empleado para minimizar la diferencia entre dos nubes de puntos . El **ICP** a menudo se usa para reconstruir superficies 2D o 3D a partir de diferentes escaneos, para localizar robots y lograr una planificación de ruta óptima, etc.

Una nube de puntos, la referencia, se mantiene fija, mientras que la otra, la fuente, se transforma para que coincida mejor con la referencia. El algoritmo revisa iterativamente la transformación (combinación de traducción y rotación) necesaria para minimizar una métrica de error.
Debemos proporcionarle como entrada las nubes de puntos de referencia y fuente, la estimación inicial de la transformación para alinear la fuente con la referencia, criterios para detener las iteraciones; y nos proporcionará como salida la transformación refinada.

Los pasos que sigue el algoritmo a grandes rasgos son:
1. Para cada punto de la nube de puntos origen, hacer coincidir el punto más cercano en la nube de puntos de referencia.

2. Estimar la combinación de rotación y traslación utilizando una técnica de minimización métrica de la distancia cuadrática media de punto a punto que alineará mejor cada punto de origen con su coincidencia encontrada en el paso anterior.
3. Transforme los puntos de origen utilizando la transformación obtenida.
4. Iterar hasta que se produzca la convergencia o el criterio de finalización.

Figura 6. Resultados de aplicar ICP

Codificación del problema

En la implementación de este problema hemos usado librerías de la ***Point Cloud Library*** (**PCL**), que implementan todos los métodos que necesitamos para llevar a cabo todos los pasos de la detección de objetos 3D citados anteriormente.

Toda la práctica está implementada en un único fichero llamado *objectDetection.cpp*, cuya cabecera se encuentra en *objectDetection.h*. Para compilar el código se utiliza el sistema de *CMake*.

ObjectDetection.cpp

El objetivo de la práctica es comprobar varios métodos para detección y descripción de keypoints, por lo que lo más sencillo es implementar una función principal muy modulada que permita cambiar una función por otra para realizar las pruebas. El pseudocódigo de esta función principal es muy parecido a los pasos para la detección de objetos citados anteriormente:

```
Main:
  Cargar las nubes de puntos
  Eliminar los planos dominantes
  Para la escena y cada uno de los objetos
    Extraer los detectores
    Extraer los descriptores
  fin Para
  Para cada objeto
```

```
    Matching entre objetos y escena
    Corregir malos emparejamientos
    Calcular la transformación
    Refinar el resultado con ICP
  fin Para
```

Las funciones que auxiliares que se utilizan para modular el código son las siguientes:

- ***loadImages()***: esta función recibe como parámetros la referencia de la nube de puntos declarada en el main para la escena llamada *scene* y de un vector de nubes de puntos, también declarado en el main, llamado *objects*. Dentro de la función cargamos las nubes de puntos de la escena y de cada uno de los objetos que queremos detectar. En una carpeta llamada *scene* se encuentra la escena que, tras ser cargada, se guardará en la nube de puntos pasada como referencia. En otra carpeta llamada *objects*, se encuentran todos los objetos numerados, los cuales son cargados y almacenados en el vector de nubes de puntos pasado por referencia.

- ***removePlane()****:* esta función recibe como parámetros la referencia de la nube de puntos de la escena, *scene*, y el vector de umbrales que creamos en el *main* para establecer los distintos umbrales para eliminar los planos. Lo primero que hacemos es definir el objeto para eliminar los puntos. A continuación, para cada uno de los umbrales que hemos definido, copiamos la escena a una nube puntos XYZ para posteriormente obtener los inliers y extraerlos de la escena. La nube resultante será guardada en la nube de puntos *scene* que le hemos pasado por referencia.

- ***removeCajonera()****:* esta función es la encargada de eliminar la parte inferior de la mesa para conseguir eliminar completamente aquellos planos innecesarios. Para ello, recorremos todos los puntos de la escena eliminando aquellos que tengan una coordenada "*y"* mayor que la coordenada *"y"* del plano de la mesa.

- ***getModel()***: esta función recibe como parámetros la referencia de la nube de puntos de la escena y del vector que va a contener los inliers, además del umbral necesario para aplicar RANSAC. Esta función obtiene el modelo del plano dominante en la imagen utilizando RANSAC, por lo que eliminando los inliers de la escena estamos eliminando el plano dominante.

- ***keypoints()***: esta función es la encargada de realizar la detección en las nubes de puntos. Dependiendo del método empleado, recibirá un tipo diferente de parámetros, y realizará procesos distintos.

- ***features()***: esta función es la encargada de extraer los descriptores a partir de las detecciones de la función anterior. Además, al igual que esta, recibirá un tipo diferente de parámetro según el descriptor empleado.

- ***matching()***: esta función recibe el vector de correspondencias y por referencia los descriptores de los objetos, el descriptor de la escena y la distancia máxima. Para obtener las correspondencias utilizamos la clase *CorrespondenceEstimation* de la librería *registration* y nos creamos un objeto de esa clase. A este objeto le indicamos la fuente, que será cada uno de nuestros objetos y la referencia la escena. Nos devolverá un vector de correspondencias, basado en la distancia máxima que regulamos, que guardaremos en el vector que le pasamos como parámetro a la función.

- ***rejection():*** esta función recibe cuatro parámetros: el vector de correspondencias obtenido en el paso anterior, el vector de correspondencias filtradas donde guardaremos los resultados de esta función, los keypoints del objeto y los de la escena. Creamos un objeto de la clase *CorrespondenceRejectorDistance*, al cual le indicamos que la fuente es el objeto y la referencia la escena, le proporcionamos como entrada las correspondencias que tenemos actualmente y, basándose en un parámetro configurable que le indicamos, nos devuelve las correspondencias filtradas, que serán guardadas en el vector de correspondencias filtradas que le hemos pasado por parámetro.

- ***transformation():*** esta función recibe el vector de correspondencias filtradas, obtenido en la función anterior, los keypoints del objeto del que se quiere obtener la transformación, los keypoints de la escena, el objeto y la escena. Creamos un objeto de la clase *TransformationEstimationSVD* donde se guardarán las matrices de transformación (rotación y traslación). Para ello, la función necesita los tres parámetros que le llegan a la función. Una vez obtenida la transformación, mostramos cómo queda el objeto en la escena después de ser transformado. Además en esta misma función llevamos a cabo el refinamiento del resultado con ICP (iterative closest ponit). Creamos un objeto de la clase *IterativeClosestPoint* llamado *icp*, al que debemos indicarle, la fuente, que será el objeto, la referencia, que será la escena, la distancia máxima, número máximo de iteraciones,etc y devuelve la transformación refinada.

Además de estas funciones, utilizamos varias funciones auxiliares para mostrar en una ventana el proceso de detección de los objetos paso a paso. Estas funciones utilizan un objeto *viewer*, mediante el cual muestran la escena, los objetos, los keypoints de ambos, las correspondencias entre ellos, el objeto tras la transformación, etc.

Keypoints

Cómo se ha explicado anteriormente, existen una gran variedad de métodos de detección de keypoints implementados en la PCL. Dependiendo de la tarea unos funcionan mejor que otros, por lo que de primeras no se puede saber qué método es mejor para el problema planteado. Se ha realizado la detección de keypoints con los métodos siguientes:

- **Uniforming Sampling**: este método consiste en reducir la dimensionalidad de la nube de puntos realizando un muestreo uniforme. Divide la nube de puntos original en *voxels*, que se pueden definir como cubos que delimitan espacios en la nube, y sustituye todos los puntos contenidos en un voxel por su centroide.

Para utilizarlo hay que incluir la librería ***<pcl/filters/uniform_sampling.h>***, y luego simplemente definimos el objeto de la clase ***UniformSampling***, inicializamos los parámetros de entrada y llamamos al método que realiza el proceso. Todos estos detalles de implementación se pueden encontrar en la documentación de la clase de la PCL.

- **ISS**: realmente es un descriptor que se basa en describir la forma alrededor del punto detectado, aunque se incorporó un sistema de detección de keypoints para mejorar los resultados. Este sistema escanea la superficie y elige los puntos que presentan mayor variabilidad en la dirección principal.

  Para utilizarlo se incluye la librería ***<pcl/keypoints/iss_3d.h>***, se define el objeto de la clase ***ISSKeypoint3D*** y un objeto *Kdtree* para llevar a cabo la búsqueda. Luego simplemente se llaman a los métodos que inicializan los parámetros necesarios y al método que computa el resultado. Son métodos que además requieren de la inicialización de parámetros que se obtienen de forma empírica. Toda la documentación relativa al uso de esta clase puede encontrarse en la página de la PCL.

- **Harris**: como se ha explicado anteriormente, este método calcula las normales en cada punto y utiliza los puntos detectados como keypoints tras el cálculo de las normales y calcula un valor de activación del keypoint para considerarlo como tal.

  Para utilizarlo hay que incluir la librería ***<pcl/keypoints/harris_3d.h>***, definir el objeto de la clase ***HarrisKeypoint3D*** y llamar a los métodos correspondientes para inicializar los parámetros necesarios y computar los keypoints detectados. Toda la documentación necesaria sobre los métodos se encuentra disponible en la documentación de la clase en la página de la PCL.

Tal y como se puede comprobar la forma de detectar los keypoints es muy similar independientemente del método que se esté utilizando, hay que incluir la librería necesaria y tener presente la documentación de la clase para poder llamar a los métodos adecuados.

Descriptores

Al igual que con los detectores, en la PCL están implementadas muchas clases de descriptores de keypoints. Lo importante en estos casos es saber qué clase usar en función de la tarea que se vaya a realizar. En este proyecto hemos utilizado como descriptores:

- **SHOT**: este método codifica información acerca de la forma alrededor de la superficie del keypoint utilizando como soporte una estructura esférica subdividida. Para utilizarlo es necesario calcular en primer lugar las normales de la nube de puntos. Los parámetros utilizados para el cálculo de las normales influyen directamente en la calidad del detector.

  Para calcular las normales incluimos la librería ***<pcl/features/normal_3d_omp.h>***, definimos el objeto de la clase ***NormalEstimationOMP*** y llamamos a los métodos que inicializan los parámetros y computan el resultado. Para obtener los descriptores

incluimos la librería ***<pcl/features/shot_omp.h>***, definimos la clase ***SHOTEstimationOMP*** y llamamos a los métodos que inicializan los parámetros iniciales y que computan el resultado. Estos métodos pueden consultarse en la documentación de la PCL.

- **PFH**: este método codifica la información del keypoint utilizando la diferencia entre las normales del punto con su vecindad y entre sí. Es un método preciso aunque lento, pero en esta práctica no tenemos restricciones de tiempo real ni una cantidad de keypoints demasiado alta, por lo que es adecuado.

  Para utilizarlo incluimos la librería ***<pcl/features/pfh.h>***, definimos la clase ***PFHEstimation*** y llamamos a los métodos necesarios para inicializar los parámetros y obtener los resultados. Estos métodos y cómo se utilizan se encuentran en la documentación de la clase de la PCL.

Pruebas y experimentación

Eliminar planos dominantes

Una vez cargadas las nubes de puntos, es necesario extraer los planos dominantes para mejorar la precisión de la detección de los objetos. Como se ha explicado anteriormente, se utiliza RANSAC para obtener el modelo del plano dominante y extraer los inliers de la escena. Sin embargo, hay dos problemas principales al utilizar este método:

1. No se conoce el número de planos dominantes de la escena, por lo que no sabemos las veces que tenemos que aplicar RANSAC.

2. El umbral para determinar si un dato pertenece al modelo o no es muy sensible. Además, lo más seguro es que este valor de umbral varíe en función del plano dominante que se busque eliminar.

La solución para los dos problemas es la misma: prueba y error. Establecer estos dos parámetros depende de la escena y de los objetos que se quieran detectar.

Para el **primer plano dominante** detectado, que es el de la pared de fondo, el mejor umbral **oscila entre 0.1 y 0.05**. Podemos comprobar que con un umbral más grande la pared no se elimina del todo porque al ser más permisivo con los datos que pertenecen al modelo, realmente no se está detectando el plano dominante. Sin embargo, con un umbral más pequeño sí se detecta correctamente el plano dominante, pero es tan restrictivo que no lo elimina del todo.

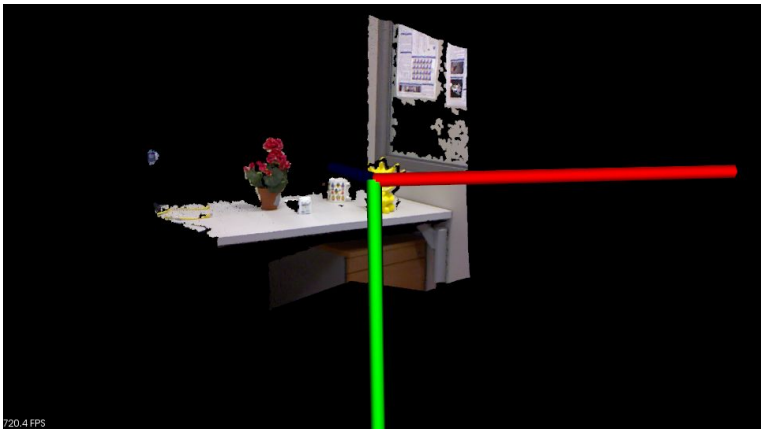


Figura 7. Umbral de 0.08, dentro de los márgenes correctos

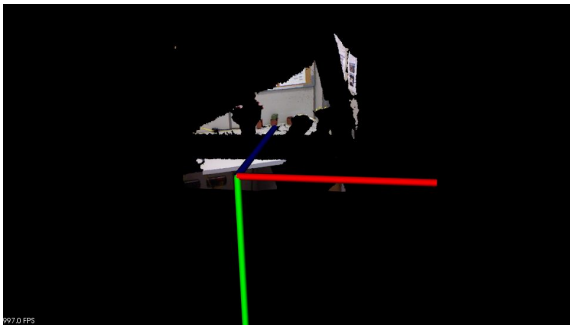


Figura 8. Umbral de 0.2

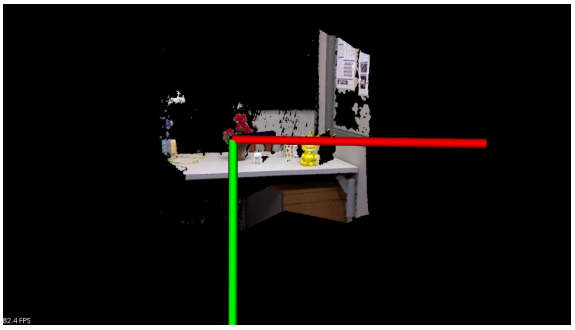


Figura 9. Umbral de 0.02

Una vez establecido el valor anterior, a partir de él se calcula el valor del **segundo plano dominante**. El valor adecuado aproximadamente **oscila entre 0.1 y 0.04**. Podemos comprobar que para valores muy bajos, el plano dominante detectado no es la otra pared, sino la mesa, aunque no la elimina perfectamente. Si el valor es muy alto, al igual que antes, no encuentra el modelo correcto de un plano dominante de la imagen.

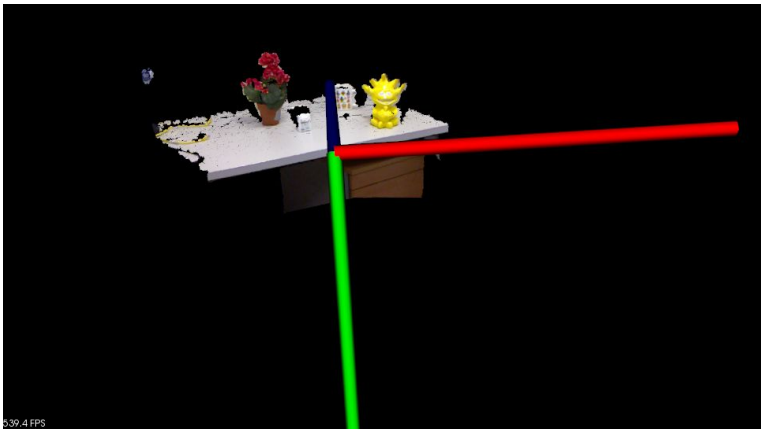


Figura 10. Umbral de 0.06, dentro de los márgenes correctos

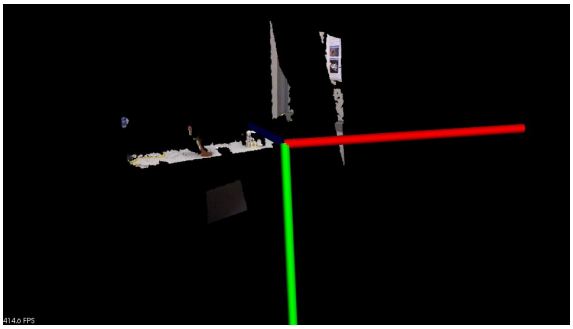


Figura 11. Umbral de 0.2

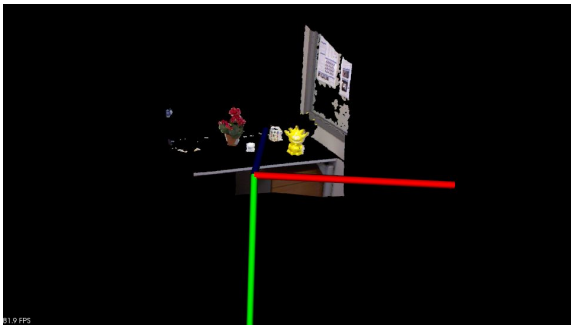


Figura 12. Umbral de 0.01

Podemos observar que para este caso particular, el **tercer plano dominante** es el último. Este plano corresponde al plano de la mesa según los valores de umbral que se han aplicado. Este umbral es más sensible, ya que es el que interactúa directamente con los objetos. Si elegimos un umbral muy grande, se elimina de la escena parte de los objetos. Por el contrario, si elegimos uno muy pequeño, la mesa no se elimina bien. El umbral correcto es **aproximadamente 0.02**, que a pesar de no eliminar la mesa totalmente, mantiene los objetos prácticamente intactos y solo queda de la mesa una fina línea del borde.

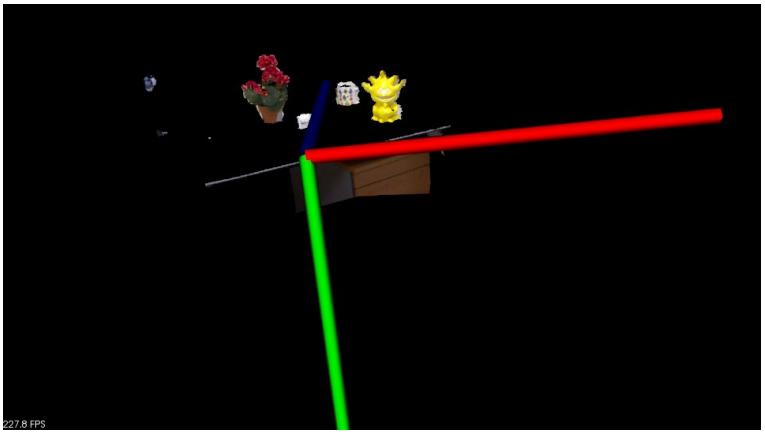


Figura 13. Umbral de 0.03

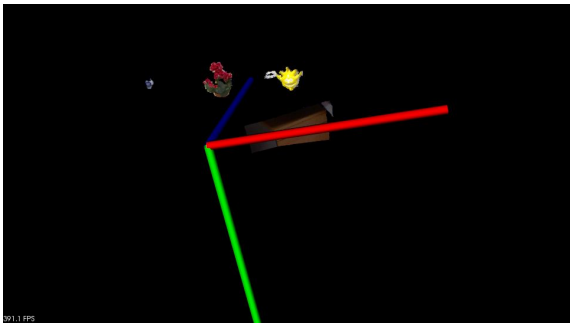


Figura 14. Umbral de 0.6

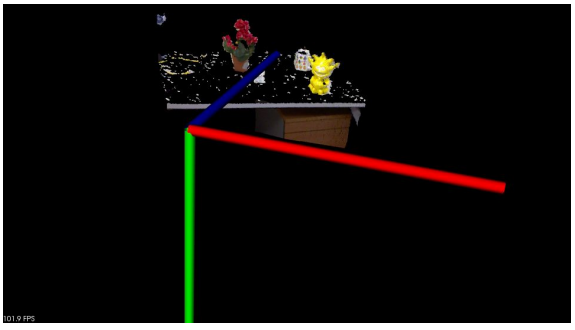


Figura 15. Umbral de 0.005

Después de hacer pruebas y experimentación con estos resultados, nos dimos cuenta que la cajonera que sigue quedándose por debajo de la mesa es molesta, ya que muchos de los keypoints de los objetos se emparejan con ella. Por lo tanto, utilizamos la función

removeCajonera tal y como se ha explicado anteriormente para eliminarla y conseguir mejores resultados.

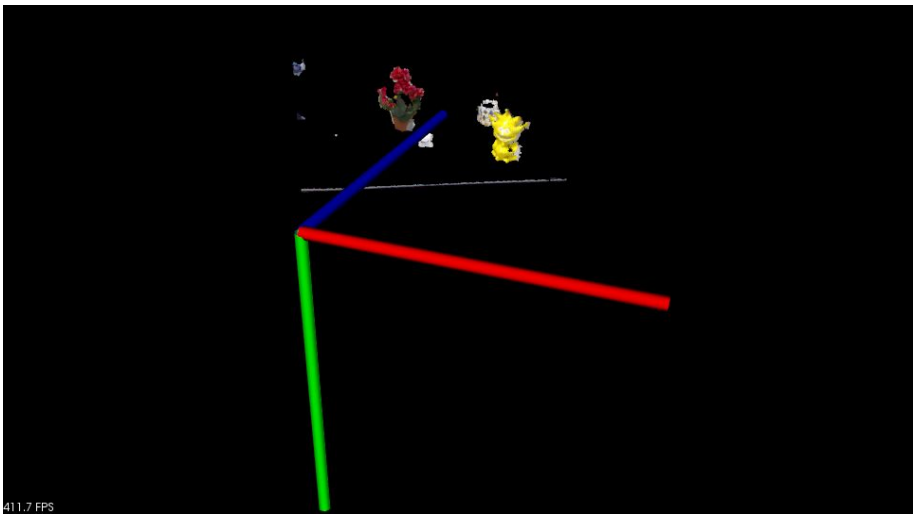


Figura 16. Resultado final tras eliminar los planos dominantes

Detectores

Para poder comprobar los keypoints detectados lo más útil es añadir a la función que muestra una nube de puntos el método del objeto viewer para insertar una esfera en un punto indicado.

ISS

Al detectar los keypoints con ISS y visualizar los resultados podemos comprobar que se detectan keypoints en zonas que no son de interés: la cajonera y el borde la mesa. Además, no hay una gran cantidad de puntos en los objetos. Por ejemplo, en un objeto únicamente detecta un keypoint. Es una cantidad insuficiente de puntos.

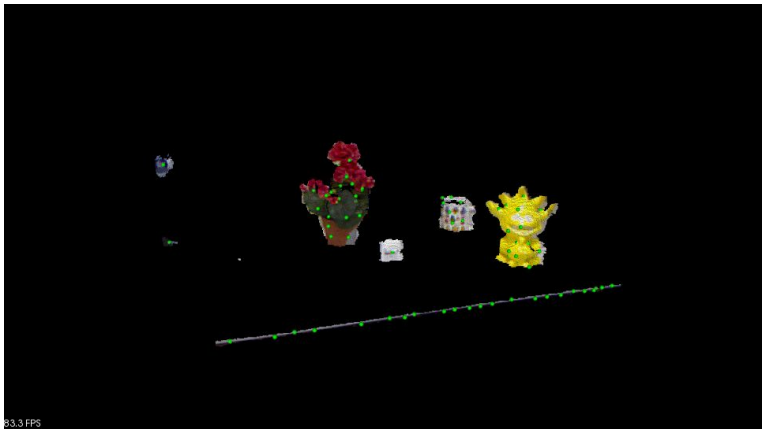
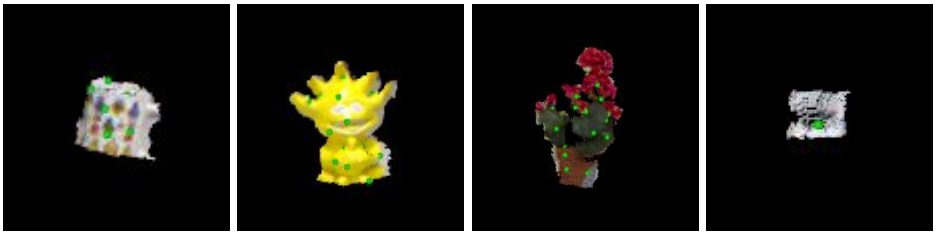


Figura 17. Keypoints de la escena usando ISS - **67**



Figuras 18-21. Keypoints de los objetos usando ISS - **6, 14, 20, 1**

Harris

El método de Harris nos devuelve una gran cantidad de keypoints, pero a pesar de detectar más puntos en los objetos respecto al método anterior, encontramos algunos de ellos en zonas sin objetos y los detectados correctamente, no se encuentran uniformemente distribuidos, ya que la mayoría se centran en los contornos.

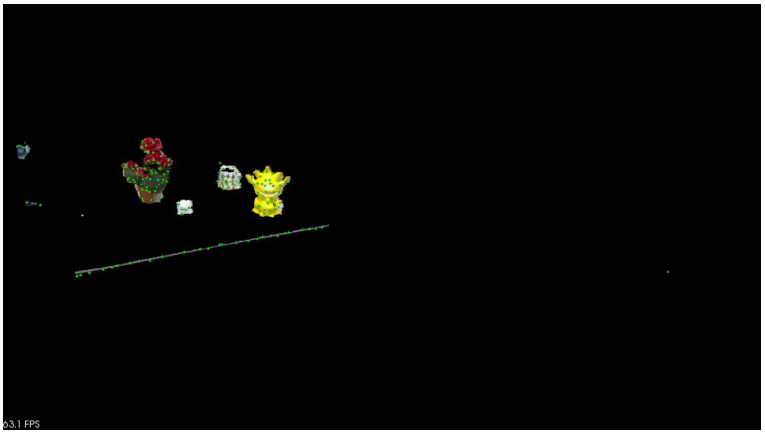
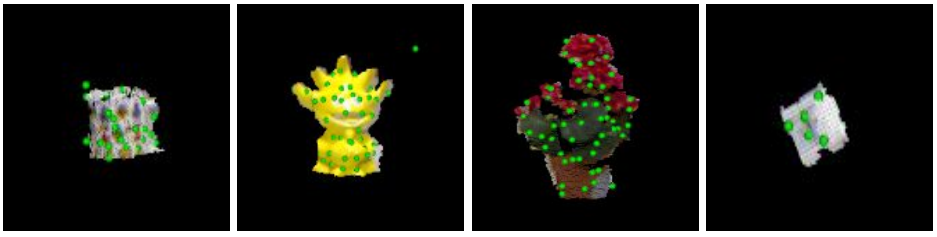


Figura 22. Keypoints de la escena usando Harris - **176**



Figuras 23-26. Keypoints de los objetos usando Harris - **23, 39, 61, 5**

Uniform Sampling

Este método nos da la mejor solución en relación al número de puntos detectados y lo descriptivos que son. Detectamos puntos suficientes en los objetos para poder describirlos correctamente y además se reparten más uniformemente por todo el objeto. Por lo tanto, es el **método escogido** para detectar los keypoints debido a que es el que mejor funciona.

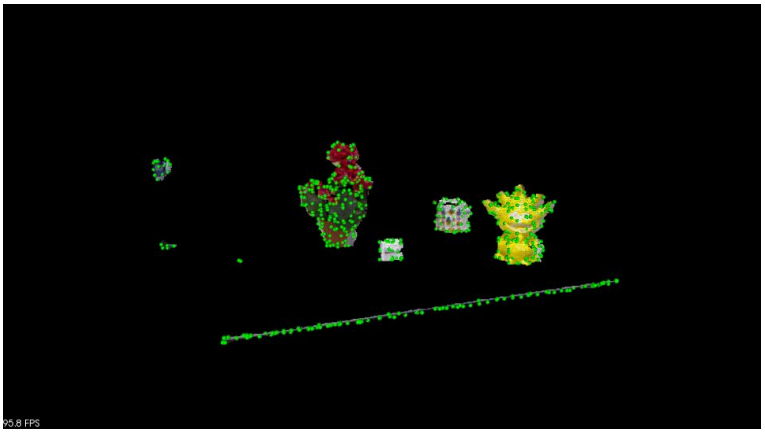
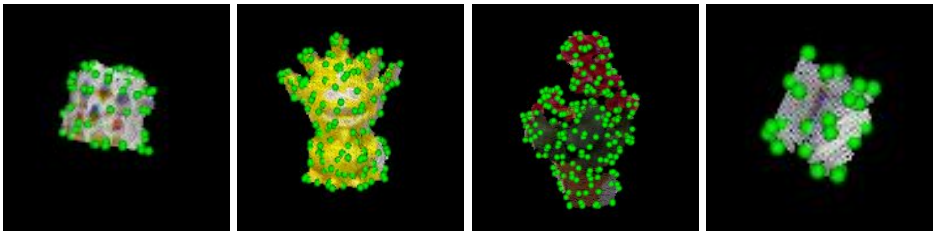


Figura 27. Keypoints de la escena usando US - **542**



Figuras 28-31. Keypoints de los objetos usando US - **51, 141, 209, 22**

Descriptores

PFH

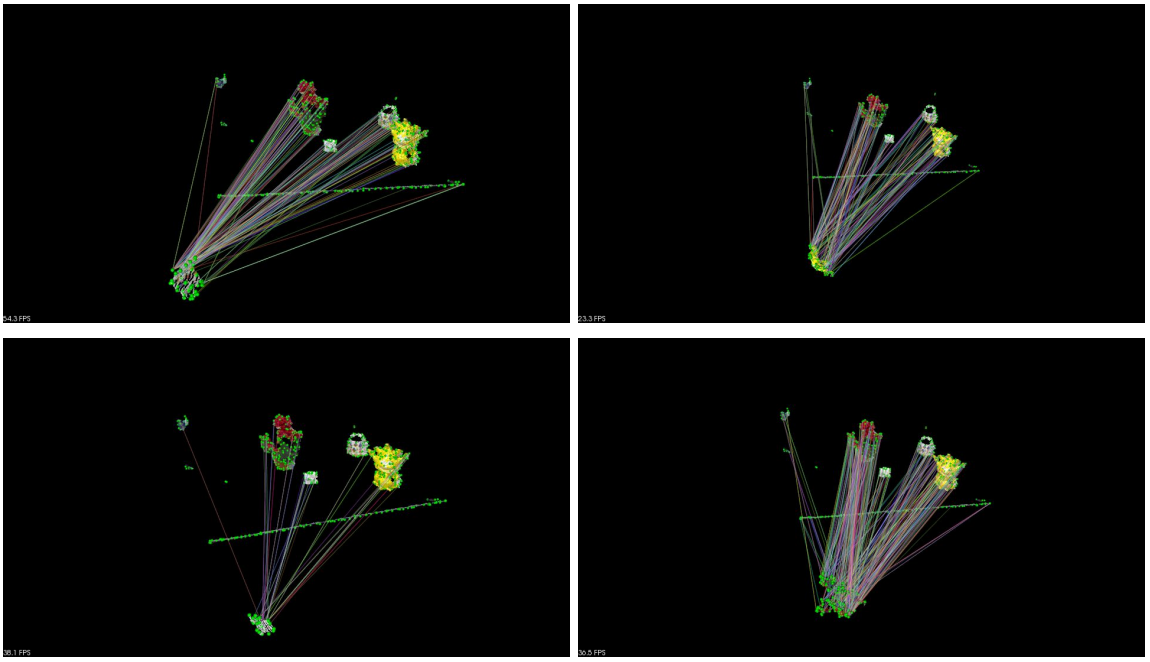
PFH es un descriptor muy preciso, pero a pesar de no tener restricciones de tiempo real en la práctica, necesitamos ajustar muchas cosas al azar y probarlas en la ejecución, por lo que finalmente este método lo hemos descartado por la cantidad de tiempo que tarda en ejecutarse.

SHOT

SHOT es un descriptor robusto y rápido que consigue describir correctamente todos los puntos detectados como keypoints a excepción de un punto en la escena. Aún así, son muy buenos resultados para un tiempo de ejecución muy rápido, por lo que es el **método escogido** para describir los keypoints detectados.

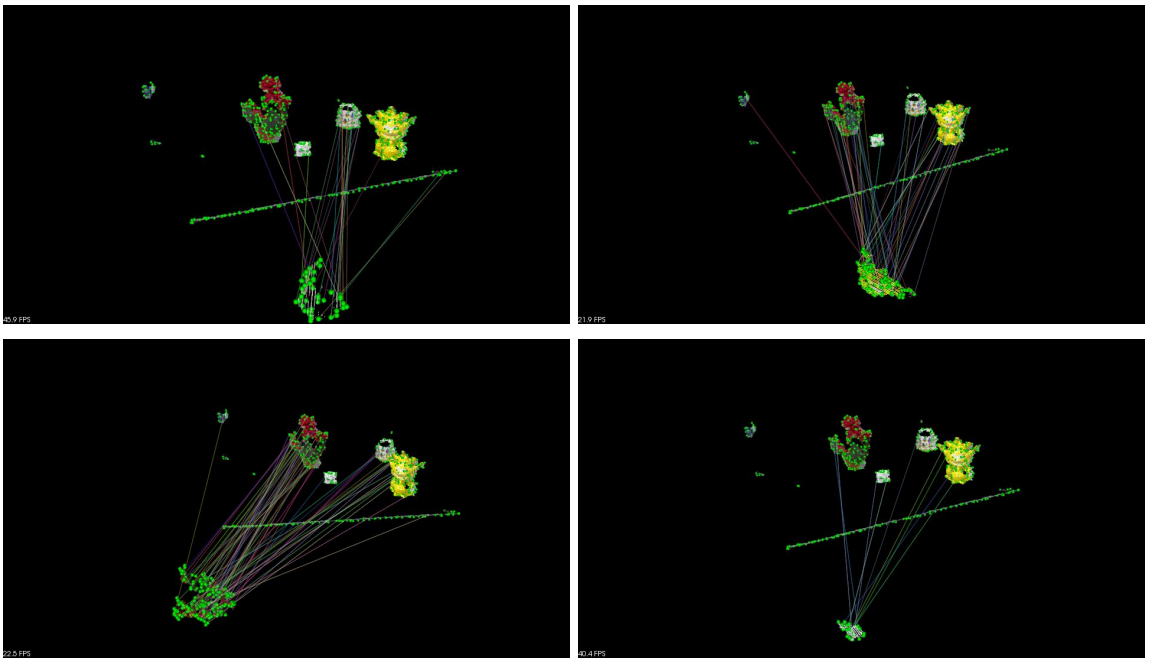
Correspondencias

En primer lugar para obtener las correspondencias entre los objetos y la escena intentamos usar ***KdTree*** de la librería **FLANN**. Para cada punto característico descrito en el objeto, buscaba el vecino más cercano en el descriptor de la escena y si la distancia era menor de cierto parámetro proporcionado, lo añadía al vector de correspondencias. Sin embargo, al mostrar las correspondencias pudimos observar que es un método impreciso y, como se puede ver en las siguientes imágenes, encontraba más correspondencias con otros objetos de la escena que con el objeto correcto.

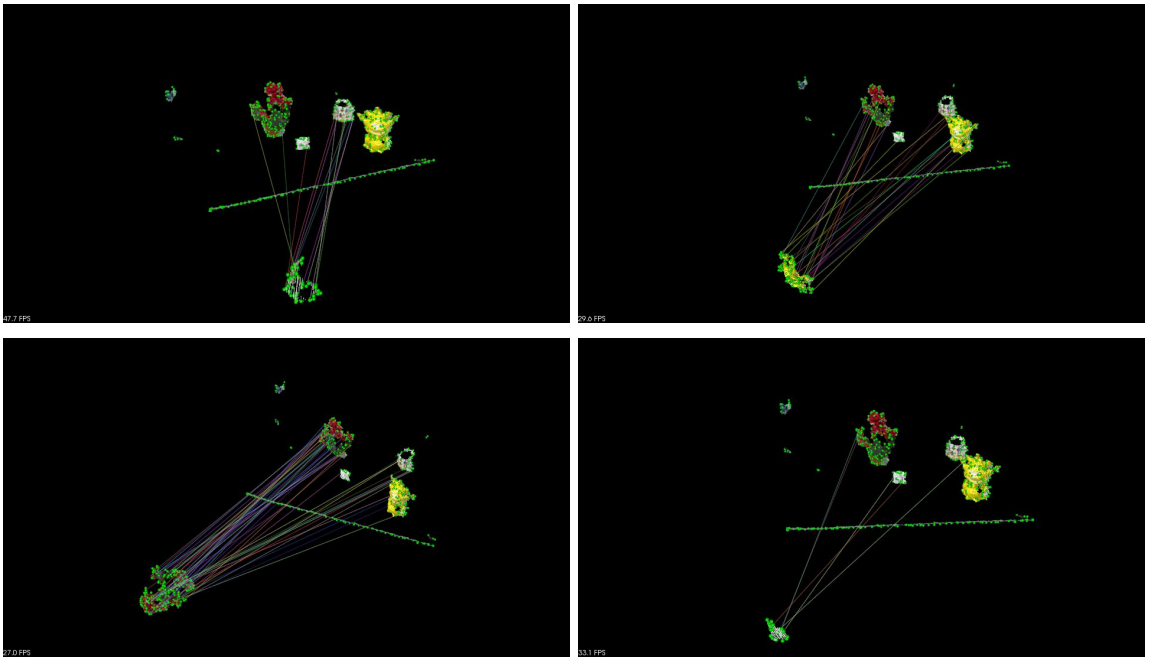


Figuras 32 - 35. Correspondencias utilizando *KDTreeFLANN* - **153, 154, 286, 33**

Por esta razón, decidimos utilizar otro método que nos proporcionará mayor precisión. Este método es ***CorrespondenceEstimation***, el cual es mucho más sencillo de implementar y con el cual obtenemos mejores resultados como se puede observar en las siguientes imágenes. Es necesario ajustar el parámetro de la distancia máxima entre los puntos para considerarlos como emparejamiento, ya que este parámetro es clave para la calidad de los mismos. Si el parámetro es muy grande, habrán muchas falsas correspondencias que perjudicarán al resultado; y si es muy pequeño, perdemos información de puntos del objeto. El parámetro elegido es de **0.8.**



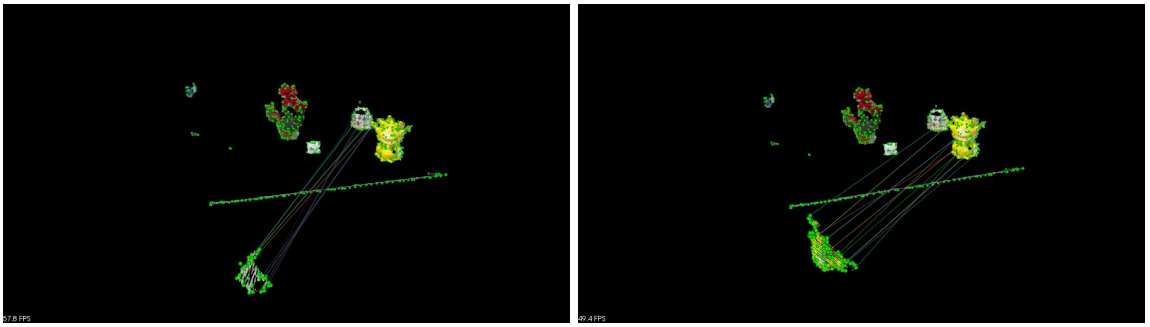
Figuras 36 - 39. Correspondencias utilizando *CE* y parámetro 1.5 - **22, 44, 62, 9**

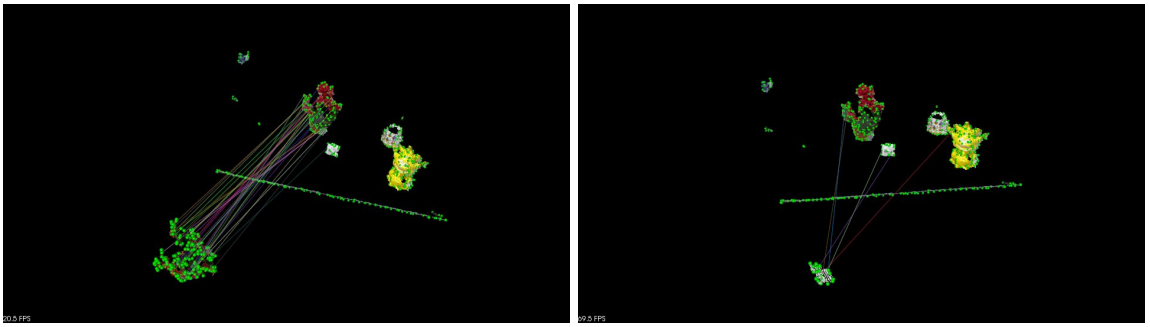


Figuras 40 - 43. Correspondencias utilizando *CE* y parámetro 0.8 - **13, 23, 46, 5**

Rejection

A pesar de estimar el parámetro de distancia máxima para obtener los emparejamientos lo más acertados posibles, hay que filtrar las malas correspondencias. Para ello, utilizamos la clase ***CorrespondenceRejectorDistance***, que utiliza RANSAC para eliminar los malos emparejamientos. También hay que definir una serie de parámetros para eliminar los emparejamientos malos sin eliminar información de emparejamientos correctos. Esta distancia máxima es diferente para cada objeto, por lo que se define en el main de forma empírica y se pasa como parámetro a la función.

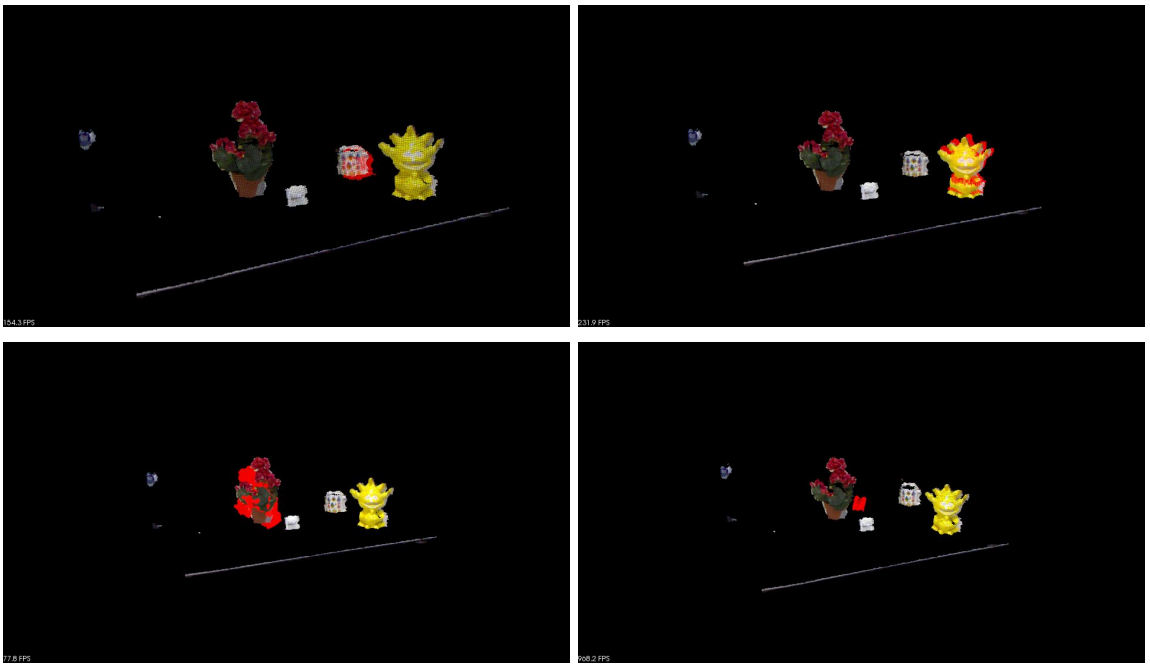




Figuras 44 - 47. Eliminar malas correspondencias- **10, 10, 31, 5**

Transformación

Una vez obtenidas las correspondencias, necesitamos obtener la transformación que las explica. Como hemos utilizado RANSAC anteriormente, podríamos utilizar el método que proporciona para estimar la transformación. Finalmente, decidimos utilizar la clase ***TransformationEstimationSVD***, que nos devuelve la transformación rígida a partir de los keypoints y el vector de correspondencias.

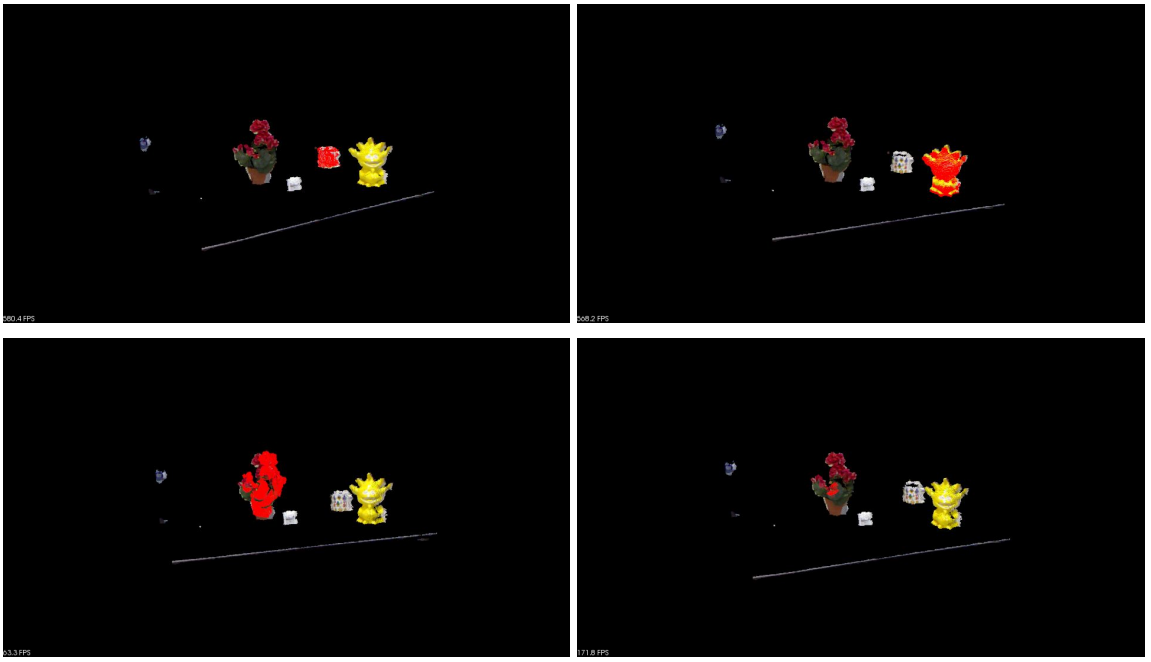


Figuras 48 - 51. Transformaciones de los objetos

Iterative Closest Points

La transformación obtenida en el paso anterior, no es perfecta, por eso necesitamos refinar el resultado. Para llevarlo a cabo, utilizamos la clase ***IterativeClosestPoint***, que intenta encontrar la mejor transformación minimizando la distancia, durante las iteraciones que le indiquemos. Podemos comprobar que los resultados mejoran en los objetos donde la transformación tenía buenos resultados, pero en el cuarto objeto no obtenemos un mejor resultado con ICP porque la transformación de la que partimos no es buena.

Los errores obtenidos para cada uno de los objetos son: 0.000257674, 0.000232589, 0.00162732 y 0.000295561. El último error no es correcto, ya que como se ha mencionado, al tener mal las correspondencias, el error del ICP lo calcula respecto a estas y se cree que está bien calculado.



Figuras 52 - 55. Alineación de las nubes de objetos tras aplicar ICP

Conclusiones

Durante la realización de esta práctica hemos visto que hay gran variedad de métodos para extraer puntos característicos y sus descriptores. Cada uno de ellos tiene una forma distinta de hacerlo, incluso algunos de ellos necesitan puntos de tipos muy específicos para funcionar. Es difícil saber cuál de ellos proporcionará mejores resultados y muy tedioso probar todos ellos. El uso de un método u otro, depende del tipo de problema a realizar y no existe uno que pueda proporcionar resultados óptimos ante cualquier tipo de problema.

En cuanto a la obtención de correspondencias, hemos tenido que invertir mucho tiempo en probar muchos valores en el parámetro de máxima distancia para eliminar todos los malos y que quedarán correctamente filtradas. De hecho, el último objeto no conseguimos filtrarlas correctamente ya que con una disminución mínima del parámetro, eliminaba todas las correspondencias establecidas.

Si quedan correspondencias erróneas la transformación no puede llevarse a cabo, como se observa en el último objeto y si no son suficientes, la transformación obtenida será errónea. Además si no se refina nunca será perfecta.

Como conclusión general, para conseguir un resultado óptimo, tenemos que escoger los métodos de detección específicamente para esta tarea y ajustar los parámetros en las correspondencias específicamente para cada objeto que detectamos. Por tanto, el proceso

de detección de objetos sería muy difícil realizarlo en tiempo real y generalizarlo para cualquier objeto.

El problema que se ha comentado anteriormente de que es muy difícil que estos algoritmos trabajen en tiempo real debido a la cantidad de parámetros que hay que ajustar y métodos que hay que elegir es uno de los principales motivos por los que el uso de redes neuronales en los sistemas de visión por computador está siendo estudiado, a pesar de que todavía se requiere muchos avances en este campo.

Bibliografía y referencias

[1] K. Lai, L. Bo, X. Ren, and D. Fox. A large-scale hierarchical multi-view rgbd object dataset. In Robotics and Automation (ICRA), 2011 IEEE International
Conference on, pages 1817–1824. IEEE, 2011.

[2] J. Tang, S. Miller, A. Singh, and P. Abbeel. A textured object recognition pipeline for color and depth image data.

[3] A. Kasper, Z. Xue, and R. Dillmann. The kit object models database: An object model database for object recognition, localization and manipulation in service robotics. The International Journal of Robotics Research, 31(8):927–934, 2012.

[4] S. Ahn, M. Choi, J. Choi, and W.K. Chung. Data association using visual object recognition for ekf-slam in home environment. In Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, pages 2588–2594. IEEE, 2006.

[5] PCL/OpenNI tutorial 4: 3D object recognition (descriptors)

[6] PCL/OpenNI tutorial 5: 3D object recognition (pipeline)

[7] Registration with the Point Cloud Library - A Modular Framework for Aligning in 3D

[8] Point Cloud Library (PCL): PCL API Documentation