Visión Artificial y Robótica
Práctica 1

Alejandro Galán Cuenca
Jorge Donis del Álamo

14 de mayo de 2021

Índice

1. Reconstrucción 3D

1.1. Propósito

El problema que se plantea es el siguiente. Supongamos una colección de n nubes de puntos $P = P_1, P_2, P_i \dots P_n$. Cada P_i contiene m puntos p_j, todos ellos en el sistema de coordenadas local de P_i. Para obtener una representación global de P, es necesario **transformar** todos los puntos de P_i $\forall\, i > 1$ al sistema de coordenadas de P_1. Esta transformación consiste en una traslación y una rotación de cada punto. Para calcular esta transformación, es necesario conocer puntos **comunes** a cada pareja de nubes. Dichos puntos deberán de ser transformados desde P_i hasta P_1 para que caigan en la misma posición.

1.2. Metodología

El procedimiento es simple. Simplemente cabe **alinear** parejas de nubes de puntos consecutivas (en el orden en que fueron tomadas[1]). Cada pareja par_i es igual a la tupla (P_i, P_{i+1}). Una vez se encuentra la transformación t que traslada P_{i+1} a P_i, podemos acumularla en una transformación global T de la siguiente manera: $T = T \times t$. Esta transformación global T es la que transforma P_{i+1} al sistema de coordenadas de P_1;

Así que, en definitiva, el problema consiste en **alinear dos nubes de puntos**. Para alinear dos nubes de puntos P_1, P_2, se ha de obtener una transformación t.

$$t = \begin{bmatrix} R_{1,1} & R_{1,2} & R_{1,3} & v_1 \\ R_{2,1} & R_{2,2} & R_{2,3} & v_2 \\ R_{3,1} & R_{3,2} & R_{3,3} & v_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde R es la matriz de rotación y v es el vector de traslación.

Para obtener una t que alinee P_1 con P_2 cabe encontrar características **comunes** a las dos nubes. Dichas características o keypoints (puntos clave), represente el mismo punto en el espacio global. Como representan el mismo punto, han de estar situados en la misma posición. El algoritmo es el siguiente:

[1] Esto es con el propósito de que compartan características.

```
1: procedure ALIGN_CLOUDS(P1, P2)
2:     k1 = keypoints(P1)
3:     k2 = keypoints(P2)
4:     correspondences = Ø
5:     for all k1_i ∈ k1 do
6:         for all k2_j ∈ k2 do
7:             if is_simmilar(k1_i, k2_j) then
8:                 correspondences ← (k1_i, k2_j)
9:     return transformation_estimation(correspondences)
```

La función `is_simmilar()` utilizará información relativa a cada uno de los keypoints para compararlos. Esta información es la que almacenan los **descriptores**. De esta manera, habrá que calcular los desriptores para todos los keypoints.

La función `transformation_estimation` es la que realmente calcula t. En este caso, usamos una estimación basada en la **Descomposición de Valores Singulares (SVD)** de la matriz de covarianza.

Con todo esto, siguen faltando dos pasos fundamentales. Primero, es necesario **rechazar** muchas correspondencias que van a ser erróneas. Eso es porque las correspondecias son una estimación. Las nubes rara vez van a compartir keypoints con la misma localización exactamente. Cada vez que se calculan los keypoints, se pueden obtener resultados diferentes[2]. De manera que `is_simmilar()` no deja de ser una estimación. Para rechazar correspondencias erróneas, emplearemos RANSAC[2].

Finalmente, este alineado de las nubes P_1 y P_2 es un alineado **grueso**. Es una buena estimación inicial poco costosa, pero **no siempre va a funcionar bien**. El problema fundamental es la detección de keypoints y su coherencia. Si estamos mirando un plano, va a ser muy complicado calcular muchos keypoints. A lo sumo se podrían calcular los puntos esquina, pero no serían suficientes. Las nubes pequeñas y con poca variabilidad van a representar un problema.

Para terminar la alineación de manera **fina**, emplearemos ICP[3].

1.3. Implementación

1.3.1. Grabación de datos

El primer paso es la obtención de las nubes de puntos consecutivas que serán usadas en el proceso de registro. Es importante que la distancia entre las distintas capturas **no ha de ser muy grande**. A menor distancia, menor diferencia entre las nubes, más keypoints en común y más sencillo será el alineado.

Para el captado de datos usamos el paquete `rosbag`:

[2] La intención es que el detector de keypoints sea lo más consistente posible por este motivo.

```
rosbag::Bag pc_bag("point_cloud_messages.bag", mode);
pc_bag.write("point_cloud", ros::Time::now(), msg);
```

El fichero `point_cloud_messages` acaba pesando 3.2GB y contiene **329** nubes de puntos.

El nodo `grabador_datos` sirve para manejar al robot mientras se capturan (continuamente) nubes de puntos. Para que las nubes estén lo más juntas posibles se usa una velocidad lineal de 0.8 y una velocidad angular de 0.6.

1.3.2. Detección de keypoints

El detector de keypoints ha de tener las siguientes características

1. Ha de ser rápido (no más de 2 segundos por nube de puntos)[3].
2. Ha de ser **consistente**, es decir, para nubes distintas, ha de detectar los mismos puntos. Éstos pueden ser puntos esquina o puntos que formen un plano, puntos uniformes...

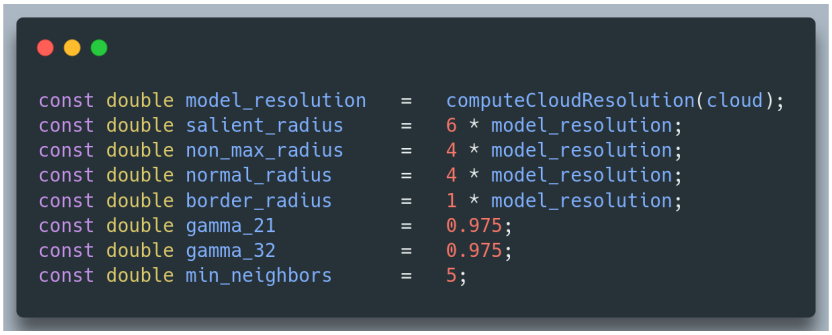
El primer detector que hemos utilizado ha sido SIFT3D[1]. Éste emplea diferencias de Gaussianas a diferentes escalas para cácular máximos y mínimos locales. Dichos máximos y mínimos son los puntos de interés. Es importante ver los parámetros de los tamaños de las escalas y la descomposición de las mismas. Al final empleamos estos:

```
0.001,  //SIFT3D minimmum scale
8,      //SIFT3D nº octaves
11,     //SIFT3D number of scales per octave
0.001,  //SIFT3D minimmum contrast
```

El segundo detector de keypoints que hemos usado es **Intrinsic Shapes Signatures** (ISS)[5].

Iss se basa en los autovalores $\lambda_1, \lambda_2, \lambda_3$ de la matriz de covarianza de la región al rededor de un cierto punto. Calcula dos ratios: $\frac{\lambda_2}{\lambda_1}$ y $\frac{\lambda_3}{\lambda_2}$. Estos ratios han de ser menores que un umbral definido por el usuario. La relevancia de los puntos se calcula después de una etpa etapa de Non Maxima Suprression (NMS). Los puntos con mayor relevancia son los puntos clave.

[3]Para agilizar tiempo de cálculo, primero se aplica un filtro de downsampling (VoxelGrid) a cada nube de puntos.



```
const double model_resolution   =   computeCloudResolution(cloud);
const double salient_radius     =   6 * model_resolution;
const double non_max_radius     =   4 * model_resolution;
const double normal_radius      =   4 * model_resolution;
const double border_radius      =   1 * model_resolution;
const double gamma_21           =   0.975;
const double gamma_32           =   0.975;
const double min_neighbors      =   5;
```

Figura 1: Parámetros de detector de keypoints ISS

`model_resolution` es una medida de la "densidad" de la nube de puntos. Para cada punto, se calcula la distancia a su vecino más cercano[4] dentro de la propia nube. Al final se divide por el número de puntos. En definitiva, es la distancia media (euclídea y cuadrada) de cada punto a su vecino más cercano.

Finalmente, tratamos de usar **Uniform Sampling**, pero no conseguimos que funcionara con la PCL. Así que decidimos hacer algo parecido, `RandomSample`. En este caso, simplemente se escogen puntos aleatorios[5] de la nube. Probamos con distintos número de puntos. Intentamos hacerlo proporcional (escoger más keypoints cuantos más puntos originales hubiera); pero al final lo mejor fue dejar un valor constante. En definitiva, se escogían 17000 puntos de manera aleatoria de cada nube como keypoints.

1.3.3. Extracción de características

Las características nos darán la información necesaria para **emparejar** pares de puntos similares. Estas correspondencias serán porque los puntos compartirán vecinos similares o a la misma distancia, o normales parecidas...

El primer tipo de característica que usamos fue **Point Feature Histogram**. En concreto la implementación FPFH (Fast Point Feature Histogram), que nos brinda complejidades lineales en vez de polinómicas.

PFH crea un historgrama para cada punto que puede ser comparado con otros histogramas para establecer correspondencias. El algoritmo usa la vecindad del punto P_q que queremos analizar. Para cada pareja de puntos de dicha vecindad, se establece un origen p_s y un destino p_t. Con ayuda de la información de las normales, se calcula la curvatura de dicha pareja. Finalmente se aumenta la frecuencia de dicha pareja respecto a la curvatura en el historgrama de P_q.

Los histogramas FPFH en PCL tienen 33 valores. Es importante revisar que ninguna de las frecuencias es `NaN`. Estos valores pueden aparecer cuando la estimación de las normales es errónea. En caso de que un punto tenga valores `NaN`, se eliminará tanto el propio keypoint como el histograma asociado.

[4] En este caso, se calcula la distancia a los dos vecinos más cercanos, ya que el más cercano es el propio punto.

[5] El generador de números aleatorios siempre se inicializa con las misma semilla.

El segundo descriptor que utilizamos fue SHOT[4] (Signature of Histograms of Orientations). Éste codifica la información sobre la topología de la esfera que rodea al keypoint. Esta esfera se divide en 32 volúmenes. Cada volumen tiene su propio histograma. Dicho histograma se actualiza con la información del ángulo[6] de la normal del keypoint con el volumen.

SHOT usa un sistema de referencia local, así que es invariable respecto a la rotación y además, es robusto al ruido.

Tanto SHOT como PFH necesita una variable que determine el rango de búsqueda máximo de los vecinos más cercanos. A mayor radio, más vecinos. Al final usamos un valor de $R = 0{,}21$.

1.3.4. Cálculo de correspondencias

Una vez tenemos los descriptores de los keypoints, hay que emparejar los keypoints con descriptores (histogramas) similares. Esto se hace con las clase `CorrespondenceEstimation` de PCL. Sin embargo, que dos puntos tengan características similares no significa que represente el mismo punto en el sistema de coordenadas global. Por lo tanto, es necesario realizar un proceso de **filtrado**.

RANSAC es un método aleatorio iterativo para detectar *outliers* dentro de un modelo matemático. Nos sirve para rechazar correspondencias erróneas (que no se ajustan a una transformación). Para hacer esto, RANSAC selecciona un subconjuntos aleatorio de las correspondencias iniciales y estima una transformación. La transformación es evaluada con el resto de correspondencias, manteniendo aquellas que se enceuntra debajo de de cierto umbral de error. Si el número de correspondencias es superior a cierto umbral, el algoritmo converge. También se puede establecer un número máximo de iteraciones. En definitiva, el algoritmo maximiza el número de *inliers*.

En PCL, usamos la clase `CorrespondenceRejectorSampleConsensus`. Esta clase nos proporciona la transformación que mayor número de inliers proporciona. Sin embargo, es recomendable volver a calcular la transformación usando únicamente los inliers (es decir, las correspondencias después del filtrado de RANSAC). Para determinar esta transformación final, usamos las clase de PCL `TransformationEstimationSVD`.

1.3.5. ICP

Con la anterior transformación, obtenemos una estimación "gruesa" del alineamiento de las nubes de puntos P_1 y P_2. Pero es necesario refinarala. Para ello usamos **Iterative Closest Points**. En definitiva, ICP hace algo parecido a lo que habíamos hecho anteriormente. Primero establece unas correspondencias entre los puntos, calcula una transformación y repite el proceso. Existen varias condiciones de convergencia, siendo una de ellas un número máximo de iteraciones.

Tal y como se ve en el apartado 1.4, ICP mejora sustancialmente la calidad de la reconstrucción 3D.

[6] En concreto, el coseno del ángulo.

1.4. Resultados

Experimentalmente, analizamos los distintos detectores de keypoints, detectores de características y el uso de ICP o no.

Es complicado determinar la **bondad** de una reconstrucción, ya que no deja de ser un parámetro subjetivo. Sólo nosotros sabemos si dos keypoints realmente se corresponden en la realidad. Sin embargo, hemos establecido una métrica que puede estimar la bondad de la reconstrucción. La hemos llamado **distancia**. Se calcula con dos nubes de puntos P_1, P_2. Ambas contienen únicamente los keypoints. Además, P_2 ya ha sido alineada previamente (es el resultado del alineamiento).

```
1: procedure DISTANCIA(P1, P2)
2:     distancia = 0
3:     for all p_i ∈ P_1 do
4:         distancia += distancia_vecino_mas_cercano(p_i, P_2)
5:     return distancia
```

Esta métrica "distancia" tiene un problema y es que **depende** del detector de keypoints. Aunque los keypoints acaben estando a mayor distancia después de la transformación, esto no significa que el alineamiento sea peor. Además, simplemente aumentando el número de keypoints, aumentaría la distancia. Cabe destacar que esta métrica es una estimación.

	FFPH				**SHOT**			
	ICP		No ICP		ICP		No ICP	
	t	d	t	d	t	d	t	d
SIFT	1209	22.76	1246	20.60	2236	20.26	2255	20.30
ISS	1064	27.68	1113	26.49	753	26.09	751	26.18
RANDOM	1176	24.21	1081	26.38	13200	24.73	12725	28.88

Cuadro 1: Resultados del registro

t es el tiempo (en segundos) de la reconstrucción total (329 nubes). d es la métrica "**distancia**" que comentábamos anteriormente. En la carpeta `reconstrucciones` del proyecto se pueden encontrar todos los ficheros `.pcd` con las nubes finales y capturas de las mismas. Todas las reconstrucciones se han ejecutado sobre un downsample de las nubes originales de VoxelGrid con tamaño de hoja = 0,03.

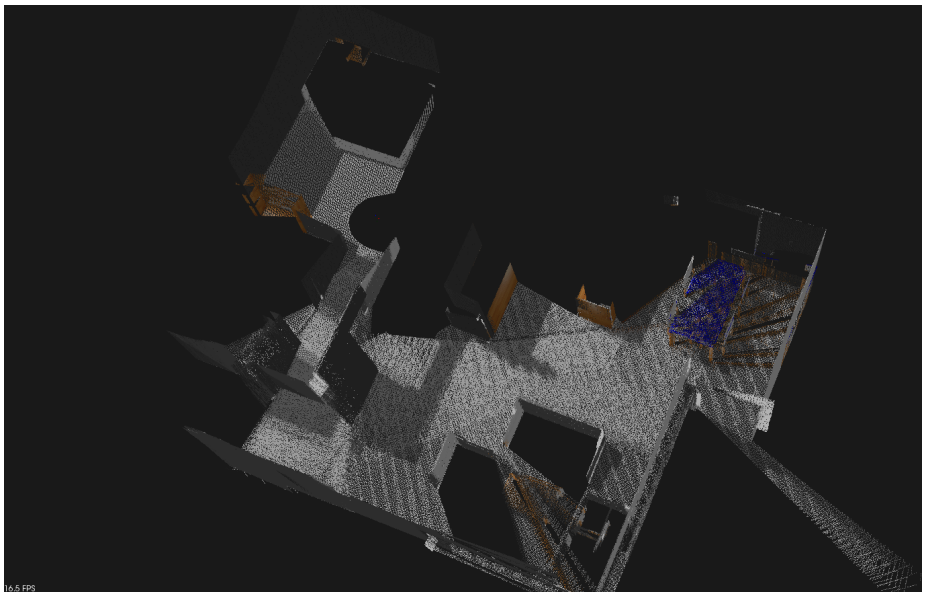


Figura 2: SIFT + FFPH + ICP

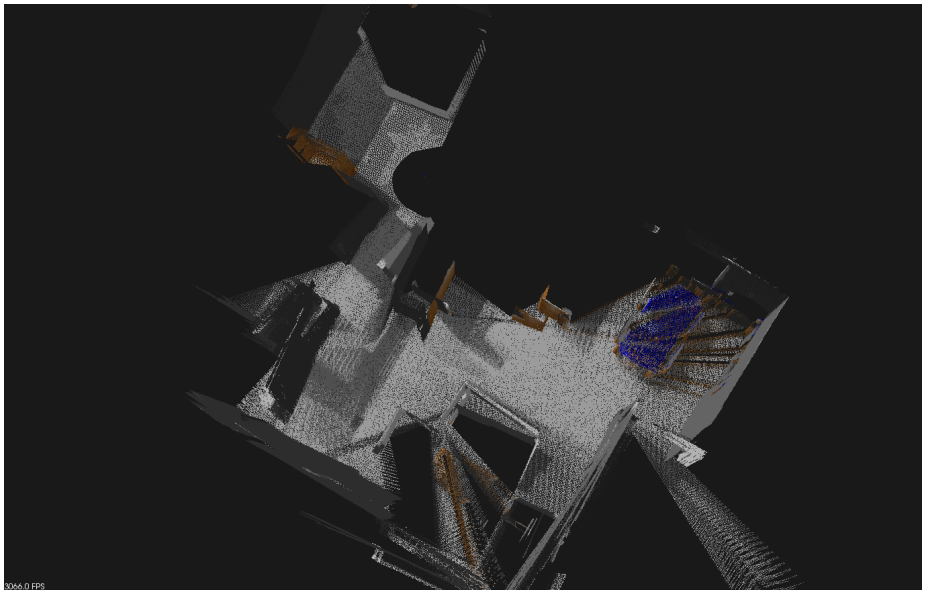


Figura 3: SIFT + FFPH

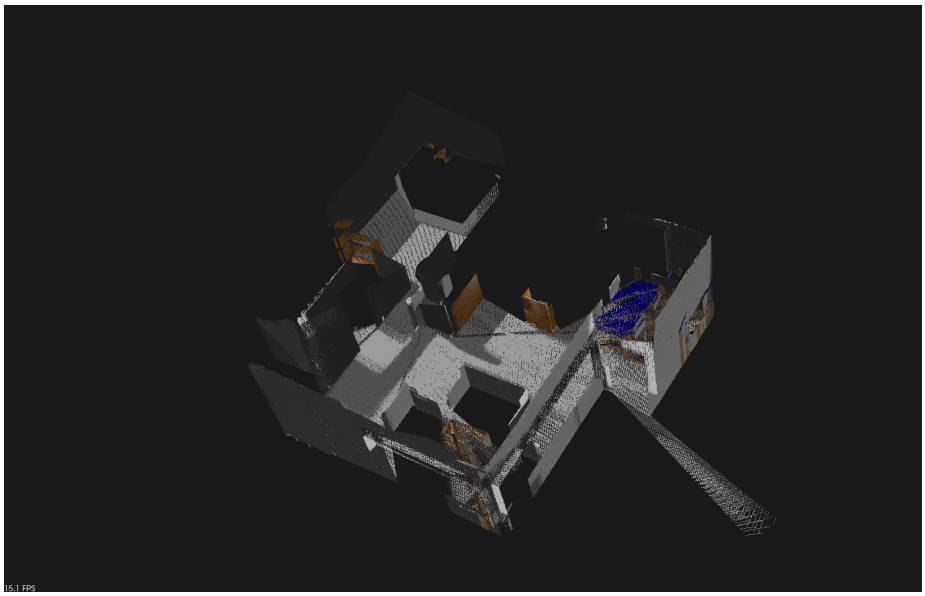


Figura 4: SIFT + SHOT + ICP

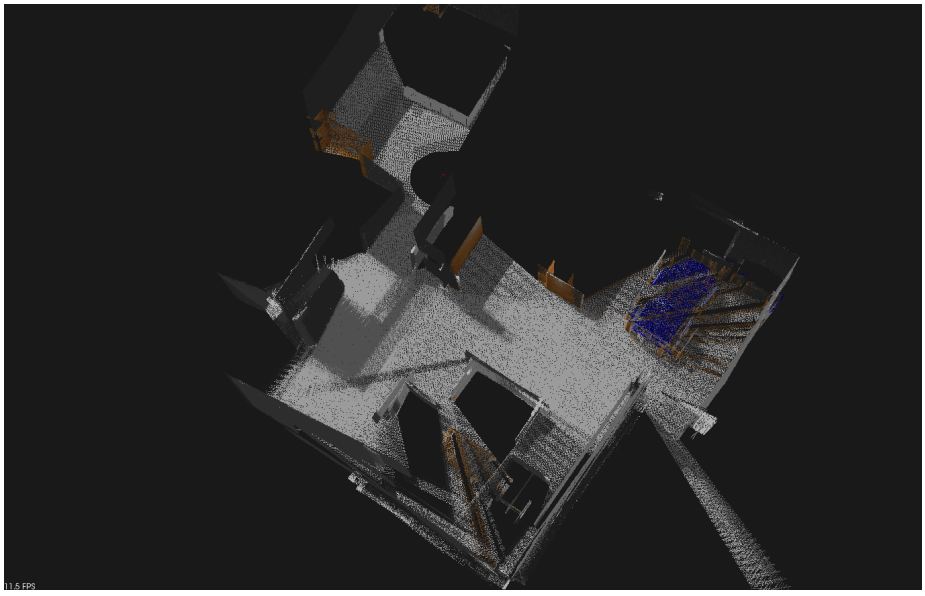


Figura 5: SIFT + SHOT

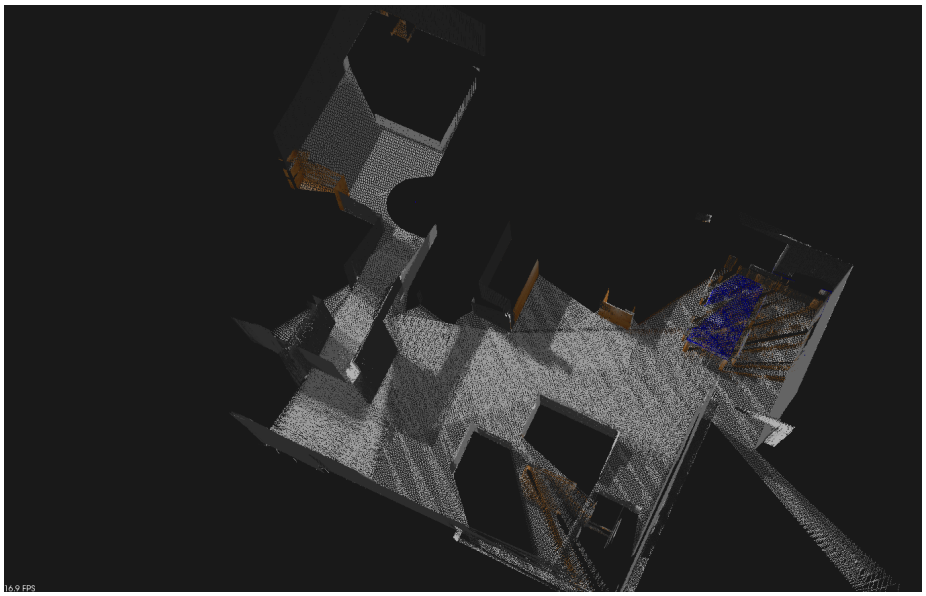


Figura 6: ISS + FPFH + ICP

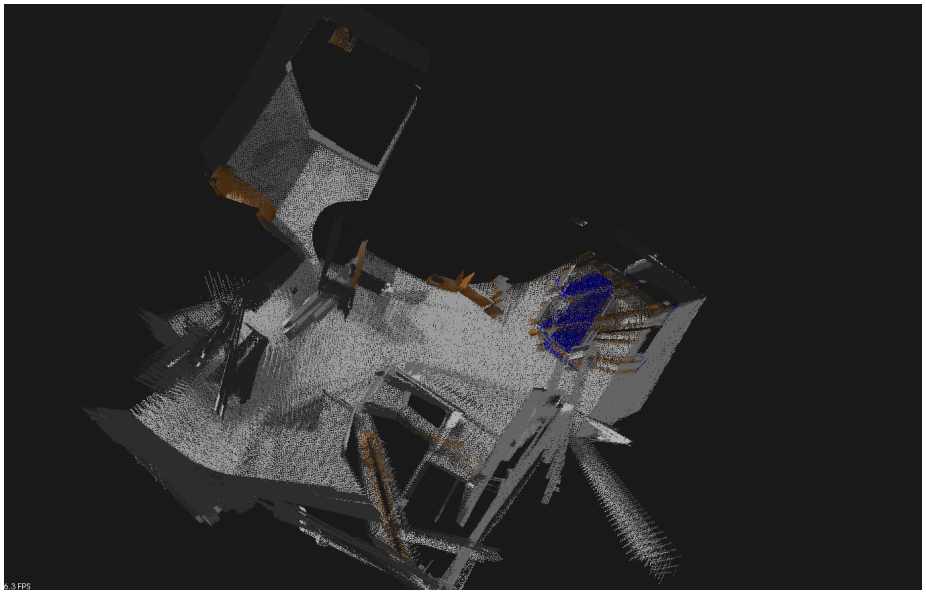


Figura 7: ISS + FPFH

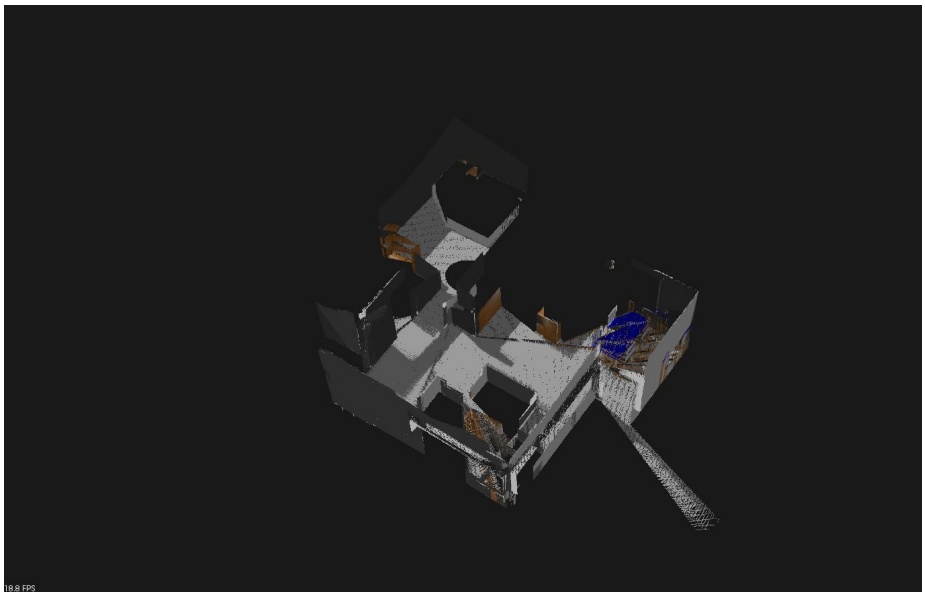


Figura 8: ISS + SHOT + ICP

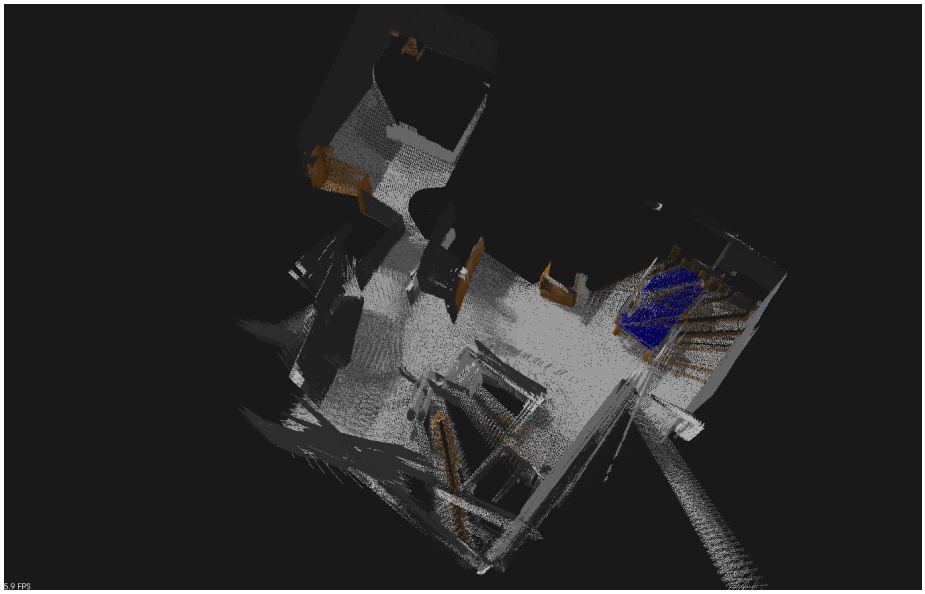


Figura 9: ISS + SHOT

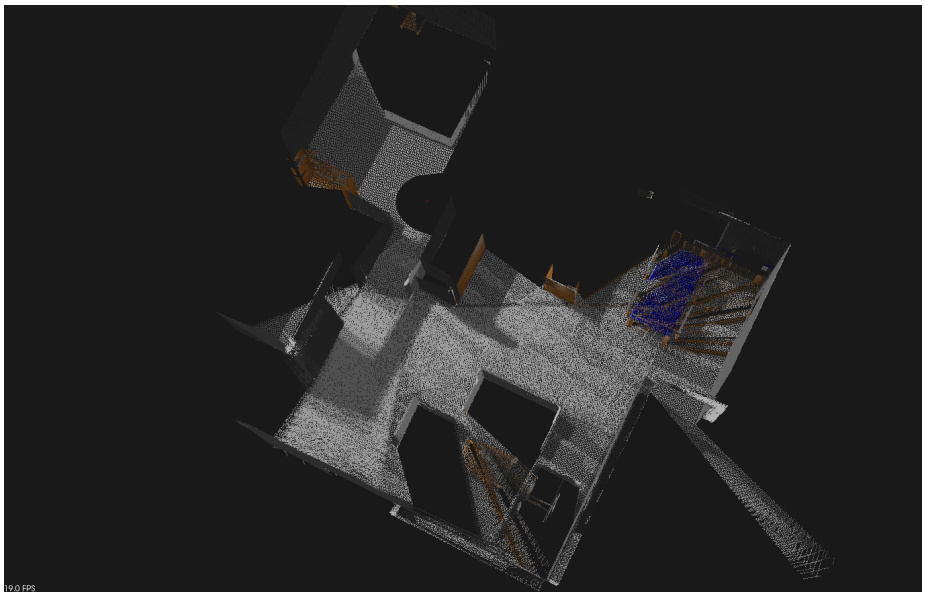


Figura 10: RANDOM + FPFH + ICP

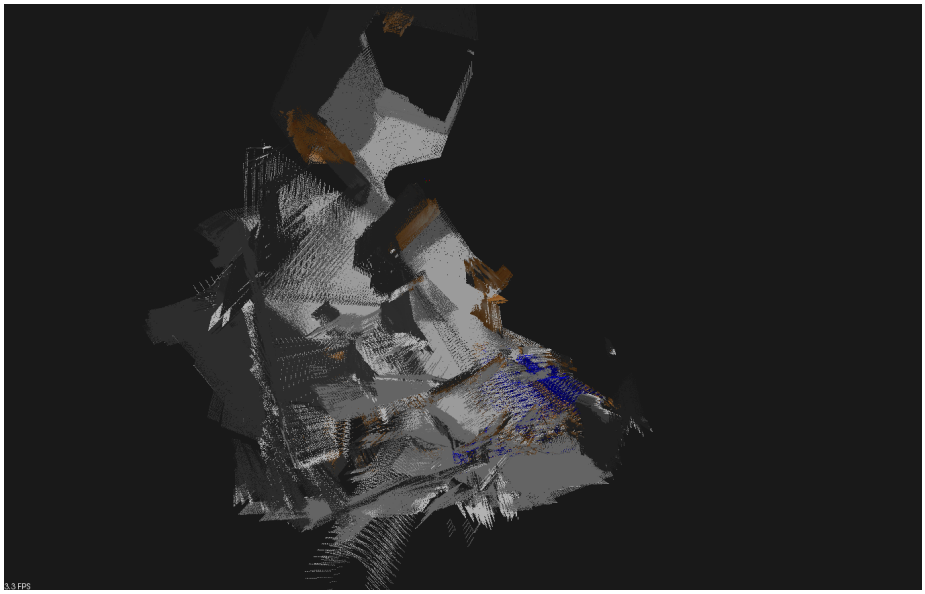


Figura 11: RANDOM + FPFH

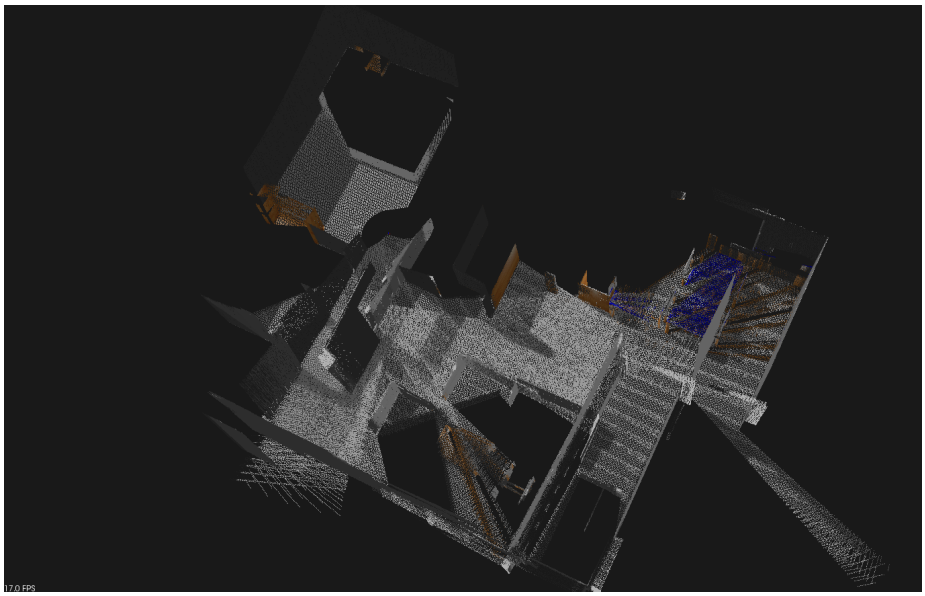


Figura 12: RANDOM + SHOT + ICP

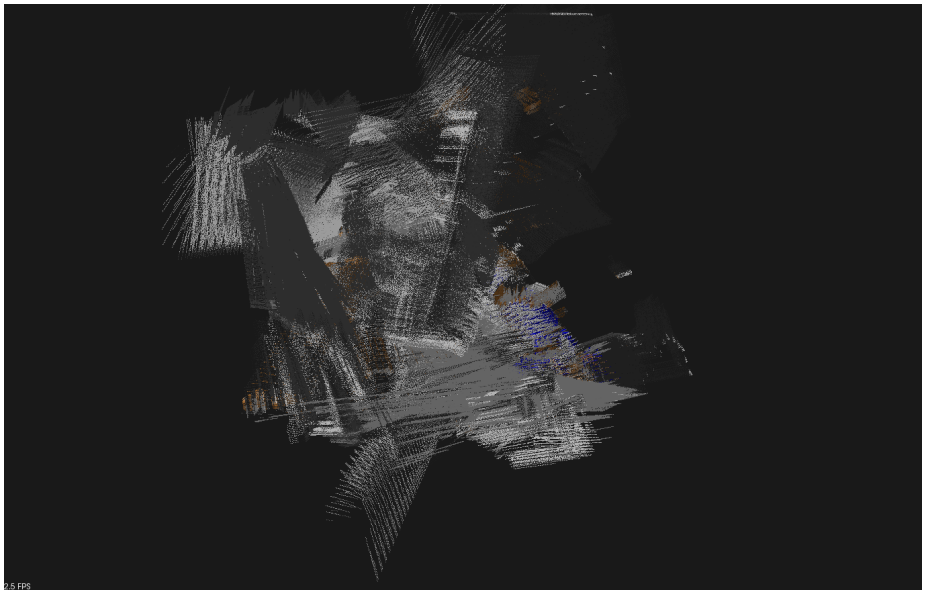


Figura 13: RANDOM + SHOT

En general, podemos ver como **ICP mejora sustancialmente los alineamientos**. Esto se da para todas las combinaciones. Además, no añade demasiado

tiempo de calculo (unos 0.23 segundos por iteración).

La combinación con menor distancia es **SIFT3D + SHOT + ICP**, pero también es de las más lentas (descartando los RANDOM). La combinación con menor tiempo es **ISS + SHOT** (sin ICP).

En general, podemos ver como los detectores de keypoints RANDOM funcionan muy mal. Además, se puede comprobar cómo la métrica de la "distancia" nos da una idea aproximada sobre la bondad de la reconstrucción (RANDOM sin ICP tiene las distancias más elevadas y las peores reconstrucciones).

2. Sigue líneas

2.1. Propósito

Se plantea el problema del seguimiento de una línea por medio de la visualización de una imagen. Se nos pasaron por la cabeza varias formas de abordar este problema, bien por redes neuronales o con PID como se explica en la documentación web del ejercicio. No obstante, consideramos que era más sencillo el método explicado en clase, este consiste en detectar si la línea está en el centro (vamos bien) o si se encuentra en uno de los laterales y tratar de corregirlo, de modo que se pretenda tener centrada siempre esta línea.

2.2. Metodología

Para realizar el seguimiento, primero se capturará la imagen y se clasificarán sus puntos en base al color (como la línea es roja, se clasifica todos los puntos en rojo/no rojo, poniéndolos a "negro" o "blanco". Como nos indica la web de Robotics-Academy).

A continuación, dividiremos el espectro horizontal en 9 zonas y realizamos un conteo de cuántos puntos rojos hay en cada una. Como asumimos que puede haber cierto error o que la línea puede estar entre varias zonas y queremos la o las más predominantes en las que se encuentre, se le hace pasar por un umbral, ya que nosotros solo querremos saber si la línea está en esa zona. Los umbrales son diferentes para cada zona, ya que con que haya 8 puntos en una zona del extremo significaría que estamos cerca de perder la línea y nos interesa corregir cuanto antes, pero si hay 25 en la zona central (de 60 posibles) este no va a pasar el umbral, ya que seguramente esté también en otras zonas contiguas, y esto significa que no tenemos la línea totalmente de frente, bien estamos en una curva o bien venimos de hacer "zig-zag" y nos conviene más hacer caso a las otras regiones correspondientes, ya que toca corregir el rumbo.

Finalmente, estos datos recogidos se pasan por un "switch" en el que se decidirá cómo de rápido avanzar y qué velocidad angular se devolverá. Los estados son excluyentes, priorizando los cercanos al centro. Así mismo, cuánto más cercana sea la zona al punto central, más velocidad lineal se tendrá y menos velocidad angular, ya que se asume que son las más cercanas a estar en el camino correcto. Por otro lado, las secciones más extremas tienen menos velocidad lineal y más angular, buscando un giro muy pronunciado para volver cuando antes a la línea y pasar a otro estado más seguro.

Al haber subclasificado en varios intervalos el entorno, el sistema se puede permitir utilizar la velocidad máxima gran parte del tiempo y seguir las curvas a una velocidad más moderada, sin sacrificar su trayectoria respecto a la recta.

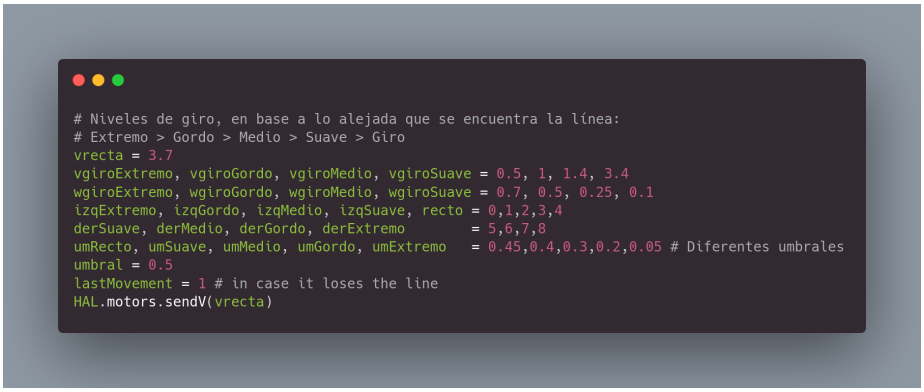
Como excepción, el estado central no se tiene en cuenta como excluyente, y este no modificará la velocidad angular, únicamente aumentará la velocidad lineal (si cambia el ángulo de giro a 0 y el coche no está paralelo a la línea la desviación será mayor, y será un impedimento constante por volver a la línea, sobre todo en curvas).

Otro aspecto a reseñar es que no haya ningún estado que cumpla con el umbral, esto significaría que el estado extremo no ha sido suficiente para volver a la línea. Para solucionar esto, se avanza linealmente lento, y se gira con gran ángulo pero menor que el estado extremo, para evitar ser un giro demasiado brusco y que vuelva a la línea de forma perpendicular, o incluso en sentido contrario. Para saber hacia qué dirección debe girar, se guarda una variable con el último estado que ha visitado, si este se encontraba a la izquierda o a la derecha de la línea. Este último estado ha sido muy útil en las pruebas realizadas, aunque en la versión final no es necesario, porque no llega a salirse de la línea en ningún momento.

2.3. Implementación

2.3.1. Código secuencial

En esta zona se declaran las constantes respectivas a las diferentes 9 zonas y se inicializa la velocidad a la de la recta. De modo que nada más arrancar no va a tener velocidad angular y no va a "virar" sobre la línea.



```
# Niveles de giro, en base a lo alejada que se encuentra la línea:
# Extremo > Gordo > Medio > Suave > Giro
vrecta = 3.7
vgiroExtremo, vgiroGordo, vgiroMedio, vgiroSuave = 0.5, 1, 1.4, 3.4
wgiroExtremo, wgiroGordo, wgiroMedio, wgiroSuave = 0.7, 0.5, 0.25, 0.1
izqExtremo, izqGordo, izqMedio, izqSuave, recto = 0,1,2,3,4
derSuave, derMedio, derGordo, derExtremo        = 5,6,7,8
umRecto, umSuave, umMedio, umGordo, umExtremo   = 0.45,0.4,0.3,0.2,0.05 # Diferentes umbrales
umbral = 0.5
lastMovement = 1 # in case it loses the line
HAL.motors.sendV(vrecta)
```

Figura 14: Código secuencial

2.3.2. Clasificación en subzonas

Se asigna la cantidad de puntos que se encuentran en cada respectiva sección.

```
blancos = np.empty(9)
for i in range(640): # 9 subzonas
    if pantalla[i] == 255:
        if i < 64:
            blancos[0] += 1
        elif i < 128:
            blancos[1] += 1
        elif i < 256:
            blancos[2] += 1
        elif i < 320:
            blancos[3] += 1
        elif i < 384:
            blancos[4] += 1
        elif i < 448:
            blancos[5] += 1
        elif i < 512:
            blancos[6] += 1
        elif i < 576:
            blancos[7] += 1
        elif i < 640:
            blancos[8] += 1
```

Figura 15: Subzonas

2.3.3. Aplicación de umbrales

Se aplican los umbrales dependiendo de la subzona a tratar.

```
for i in range(9):
    blancos[i] /= 64

    if (i == 0 or i == 8):
        umbral = umExtremo
    elif (i == 1 or i == 7):
        umbral = umGordo
    elif (i == 2 or i == 6):
        umbral = umMedio
    elif (i == 3 or i == 5):
        umbral = umSuave
    elif (i == 4):
        umbral = umRecto

    if blancos[i] > umbral:
        blancos[i] = 1
    else:
        blancos[i] = 0
```

Figura 16: Aplicación de umbrales

2.3.4. Toma de decisiones

Finalmente el conjunto de condiciones aplicadas a los datos reunidos anteriormente para decidir las velocidades a aplicar al coche de carreras.

```
    if( blancos[izqSuave] == 1 ): # priorizar los suave
        HAL.motors.sendV(vgiroSuave)
        HAL.motors.sendW(wgiroSuave)
        lastMovement = 1
    elif( blancos[derSuave] == 1 ):
        HAL.motors.sendV(vgiroSuave)
        HAL.motors.sendW(-wgiroSuave)
        lastMovement = -1
    elif( blancos[izqMedio] == 1 ):
        HAL.motors.sendV(vgiroMedio)
        HAL.motors.sendW(wgiroMedio)
        lastMovement = 1
    elif( blancos[derMedio] == 1 ):
        HAL.motors.sendV(vgiroMedio)
        HAL.motors.sendW(-wgiroMedio)
        lastMovement = -1
    elif( blancos[izqGordo] == 1 ):
        HAL.motors.sendV(vgiroGordo)
        HAL.motors.sendW(wgiroGordo)
        lastMovement = 1
    elif( blancos[derGordo] == 1 ):
        HAL.motors.sendV(vgiroGordo)
        HAL.motors.sendW(-wgiroGordo)
        lastMovement = -1
    elif( blancos[izqExtremo] == 1 ):
        HAL.motors.sendV(vgiroExtremo)
        HAL.motors.sendW(wgiroExtremo)
        lastMovement = 1
    elif( blancos[derExtremo] == 1) :
        HAL.motors.sendV(vgiroExtremo)
        HAL.motors.sendW(-wgiroExtremo)
        lastMovement = -1
    else:   # Se ha perdido y no encuentra la línea
        HAL.motors.sendV(vgiroExtremo)
        HAL.motors.sendW(lastMovement*wgiroGordo)
    if ( blancos[recto] == 1 ): # El momento de giro que llevase anteriormente
        HAL.motors.sendV(vrecta)
```

Figura 17: Decisiones

2.4. Resultados

Enlace a drive con el vídeo demostración de una vuelta al circuito

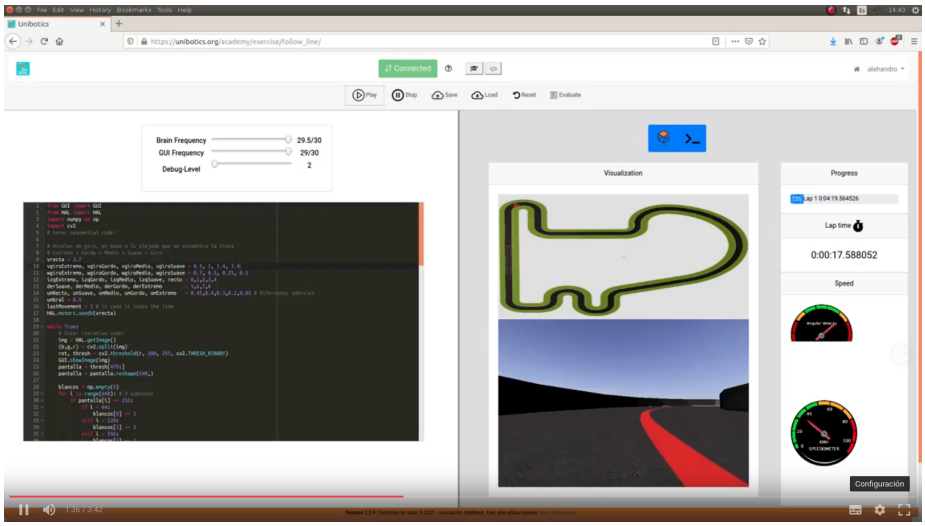


Figura 18: Vídeo demo

2.5. Feedback sobre Unibotics

Hemos experimentado ciertos problemas respecto a la plataforma. A parte del problema de la versión que se solucionó por medio de los foros. Presentaba ciertos bugs, como que la cámara visualice a veces al propio coche de fórmula 1 y esto sea un ruido casi más grande que la propia línea, que la cámara se coloque un poco más alta y la línea se vea más delgada (la ejecución va a ser muy diferente). Este en concreto sucede cuando el robot se ha chocado contra la pared y reiniciamos, de hecho si además de chocarse sigue avanzando, se sale del circuito y no va a volver a su posición original, imposibilitando la cámara o se queda en el césped, obligando a reiniciar el pc. (Entiendo que con gazebo se podría recolocar el coche, pero mi máquina no era suficiente para utilizar ese gazebo web cómodamente).

Otro aspecto es la terminal, con un tamaño muy reducido y no modificable. Que por "noVNC" al menos en mi caso no ha sido posible utilizarla en pantalla completa, igual que gazebo.

A la hora de realización de la práctica, en mi máquina personal (Ubuntu 16 donde corría ROS y Gazebo) no era posible realizarlo porque a los segundos perdía la conexión y tocaba reiniciar la máquina para que volviese a pasar. Para arreglar este error sí que viene la mayor bondad del Unibotics, trabaja en un entorno docker. Afortunadamente disponía de un portátil fedora en el que creo que no tenía ni python, pero únicamente instalando docker (y sus 7gb de contenedor) me fue posible utilizar Unibotics.

El aspecto negativo de este ejercicio es que depende de las capacidades de la máquina en la que lo estés corriendo. Quiero decir, yo inicialmente lo desarrollé en un ordenador con capacidades muy limitadas, y el circuito se completaba en 14 minutos (para hacer debug y comprobar ciertas curvas se hizo bastante tedioso, además que las más problemáticas están al final, y Gazebo no funcionaba correctamente en esta máquina). Claro, cuando conseguí que me fuese en mi máquina personal que es más potente, tuve que cambiar un par de parámetros porque el coche se chocaba, pero con casi el mismo código se pasaba el circuito en a penas 3 minutos...

Además, una vez terminado, para grabar el vídeo de demostración, como capturar la pantalla consume más recursos este mismo código iba más lento, hace más "eses" y se acababa chocando. Sí que es verdad que si realizas el ejercicio con una velocidad prudencial, por ejecutarse en un entorno con menos capacidades va a seguir pasándose el circuito, no era mi caso porque sí que buscaba el límite de velocidad y seguir la propia línea.

A pesar de los inconvenientes que he comentado, tengo en cuenta que está todavía en desarrollo y podría ser una buena opción en un futuro. Teniendo en cuenta que ROS nos ha dado más problemas, de más tiempo e inexplicables, con esta opción se evitan.

Referencias

[1] David G. Lowe. “Distinctive Image Featuresfrom Scale-Invariant Keypoints”. En: (5 de ene. de 2004).

[2] Robert C. Bolles Martin A. Fischler. “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography”. En: (jun. de 1981).

[3] Neil D. McKay Paul J. Besel. “A method for registration of 3-D shapes”. En: (feb. de 1992).

[4] Luigi Di Stefano Samuele Salti Federico Tombari. “SHOT: Unique Signatures of Histograms for Surface and Texture Description”. En: (ago. de 2014).

[5] Yu Zhong. “Intrinsic Shape Signatures: A Shape Descriptor for 3D Object Recognition”. En: (nov. de 2009).