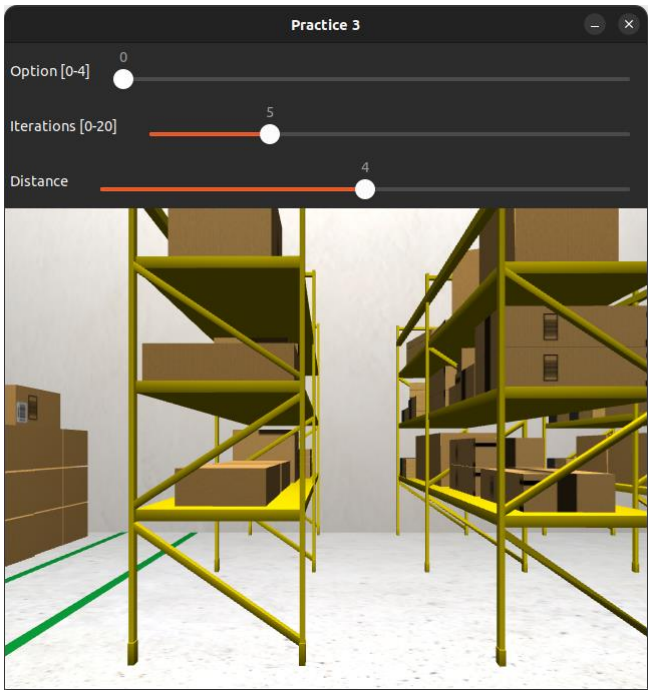
Práctica 3 – Morfología, Geometría y 3D

Este ejercicio tiene como objetivo trabajar en OpenCV con operaciones morfológicas, geometría 2D-3D y 3D-2D, y PointCloud. Todo ello visto en el **Tema 6: Operaciones morfológicas**, **Tema 7: Geometría y calibración** y **Tema 8: Visión 3D**.

Puntos totales posibles del ejercicio: 10

Instrucciones

Utilizando el simulador con Tiago y el escenario de **aws_warehouse**, se pide crear un programa que trabaje con la imagen y muestre en la parte superior varios controles deslizantes (sliders) como los de la figura:



Se pide que en cada una de las **5 opciones** se haga lo siguiente con la imagen **BGR** del Tiago:

- **Opción 0:** Mostrar la imagen en formato de color **RGB**. En caso de haber puntos pintados en la opción 3, borrar los puntos y dejar únicamente la imagen original sin procesar.
- **Opción 1:** Identificar el **esqueleto** de las líneas amarillas y negras del suelo.
- **Opción 2:** Proyectar puntos de **3D a 2D** a diferentes distancias.
- **Opción 3:** Calcular las proyecciones de **2D a 3D** de los puntos seleccionados con el ratón teniendo en cuenta la imagen de profundidad.
- **Opción 4:** De la nube de puntos, **extraer los planos** asociados a las baldas de las estanterías utilizando Point Cloud Library

Para añadir un evento que controle los clicks de ratón, es necesario crear un Callback:

```
// create mouse callback
cv::setMouseCallback( "window_name", on_mouse, 0 );
```

Donde “window_name” es el nombre de la ventana que controlará este evento, y la función on_mouse es la función a la que se llama cuando ocurra una pulsación, la cual se define como sigue:

```
// create mouse callback
void on_mouse(int event, int x, int y, int, void*)
{
    switch (event){
      case cv::EVENT_LBUTTONDOWN:
        std::cout << "Left button" << std::endl;
        break;
      case cv::EVENT_RBUTTONDOWN:
        std::cout << "Right button" << std::endl;
        break;
      default:
        std::cout << "No button" << std::endl;
        break;
    }
}
```

Esqueleto

Hay que identificar las cintas amarillas y negras, para ello, se puede aplicar cualquier técnica vista en clase para identificar únicamente aquellos objetos que pertenecen a las mismas.

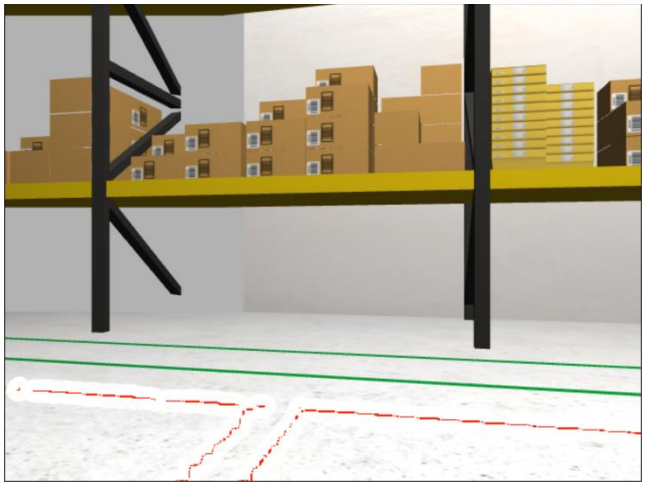
Una vez identificados los objetos en la zona deseada, se pide crear el esqueleto de la imagen a partir de operaciones morfológicas.

El algoritmo que hay que implementar para conseguir un esqueleto a partir de operaciones morfológicas es el siguiente:

1. Crear una imagen **esqueleto** vacía.
2. Realizar una **apertura** de la imagen. La llamaremos **open**.
3. **Restar** esta nueva imagen open a la imagen original. La llamaremos **temp**.
4. **Erosionar** la imagen original, y redefinir la imagen esqueleto calculando la **unión** entre el **esqueleto** actual y la imagen **temp**.
5. **Repetir los pasos 2-4** tantas veces como se haya elegido en el slider **Iterations** (0-20).

El **esqueleto** se mostrará sobre la imagen original con las líneas del apartado anterior, y el **esqueleto** calculado superpuesto será de color **rojo**.

Con el fin de visualizar mejor el esqueleto, se valorará positivamente que se eliminen del fondo las cintas en la imagen original, mostrando únicamente el esqueleto.

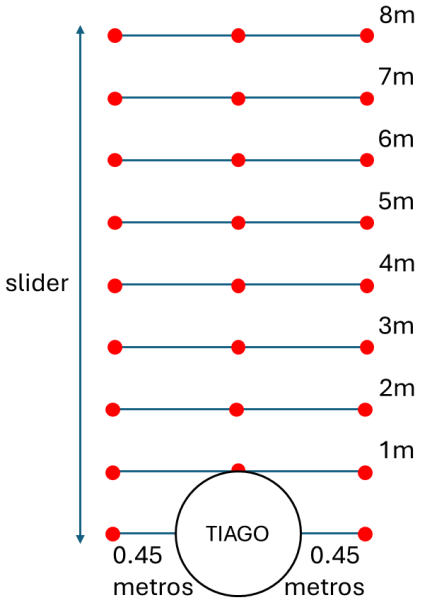


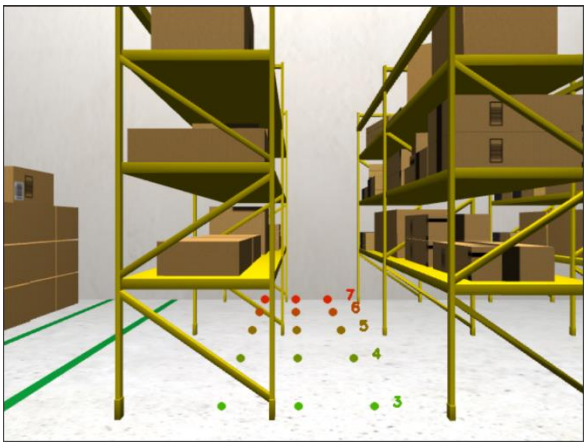
Proyección 3D a 2D

Se pide que se proyecten puntos en la base del robot (**basefootprint**) de diferentes colores, que estarán distanciados 1 metro entre ellos en la realidad (mundo 3D), hasta el valor máximo seleccionado en el slider de **Distance**. Este slider irá de **0 a 8**, y junto a uno de los puntos exteriores se indicará el valor de la distancia en los puntos.

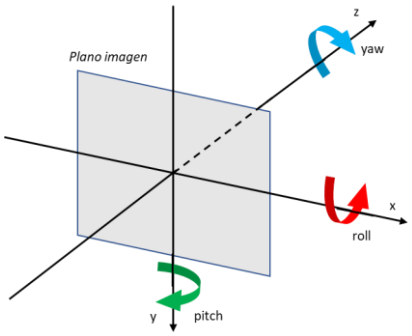
Se generarán puntos tanto en la línea del eje X del robot, como en una zona lateral, que tendrá 0.9 metros de ancho, es decir, 0.45 metros a cada lado del Tiago, y estará situada a los pies de este, por lo que cada punto nos indicará la distancia de los objetos situados en el suelo en dicho punto.

Para ello, puede verse el esquema del mundo 3D en la siguiente figura, junto al resultado final.

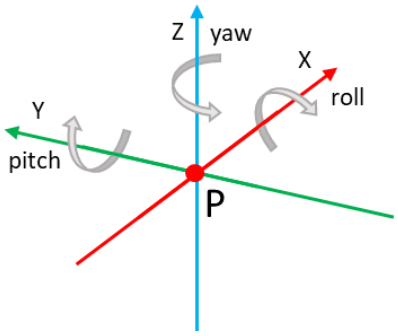




Hay que tener en cuenta que el sistema de coordenadas de la cámara difiere del sistema de coordenadas del simulador Gazebo (ver imagen), y que una unidad en el eje z de la imagen (X/Y en Gazebo), equivale a 1 metro en la realidad.



Ejes en la imagen



Ejes en Gazebo

Para calcular la transformada de 3D (X,Y,Z) a 2D (x,y), necesitamos conocer los **parámetros intrínsecos** y **extrínsecos** del sistema para generar las matrices cv::Mat correspondientes.

Para valores de los **parámetros intrínsecos**, hay que observar la **matriz de proyección** que nos facilita el simulador a través de los siguientes comandos vistos en clase:

1. Ver el tipo del mensaje enviado en ROS para la información de la cámara:

```
ros2 topic info /head_front_camera/rgb/camera_info
```

2. Una vez sabemos el tipo del mensaje, utilizaremos el siguiente comando para ver la ayuda del mensaje y observar los parámetros que nos interesaría ver:

```
ros2 interface show <msg>
```

3. Observadas las variables que necesitamos, tenemos que ejecutar el siguiente comando para ver qué valores tiene la cámara del simulador y así poder generar la **matriz *K*** correspondiente a los **parámetros intrínsecos** del sistema.

```
ros2 topic echo <topic>
```

Para la obtención de los **parámetros extrínsecos**, es necesario conocer la transformada desde la base del robot a la cámara. Para ello, podemos utilizar las tfs que nos proporciona el modelo del robot. Y ejecutando el siguiente comando, conocer la traslación y rotación que sufre la cámara con respecto a la base del robot.

```
ros2 run tf2_ros tf2_echo <frameid_from> <frameid_to>
```

Una vez generadas las matrices, se utilizará la función **projectPoints** de OpenCV para realizar la multiplicación de las matrices y los puntos. Esta función necesita las tres matrices (rotación, traslación y parámetros intrínsecos) en formato cv::Mat, y un vector de puntos 3d que contiene los puntos a proyectar en coordenadas del mundo, y un vector de puntos 2d que devolverá los puntos en coordenadas de la imagen.

Dado que la transformada da la rotación como un quaternion, es posible convertirla a tf2::Matrix3x3 de forma sencilla antes de obtener la cv::Mat de la siguiente manera:

```
tf2::Matrix3x3 mat(tf2::Quaternion{rotx, roty, rotz, rotw});
```

Se valorará positivamente que la **creación de las matrices** de parámetros intrínseca y extrínseca se haga de forma **automática**.

Para ello, se obtiene la información del modelo de la cámara del atributo **camera_model_**. El cuál contiene tanto los parámetros individuales como la **matriz de parámetros intrínsecos**.

En el caso de la **matriz de parámetros extrínsecos**, es necesario obtener de forma automática los valores de la transformada del punto 4 mediante las funciones de **lookupTransform**.

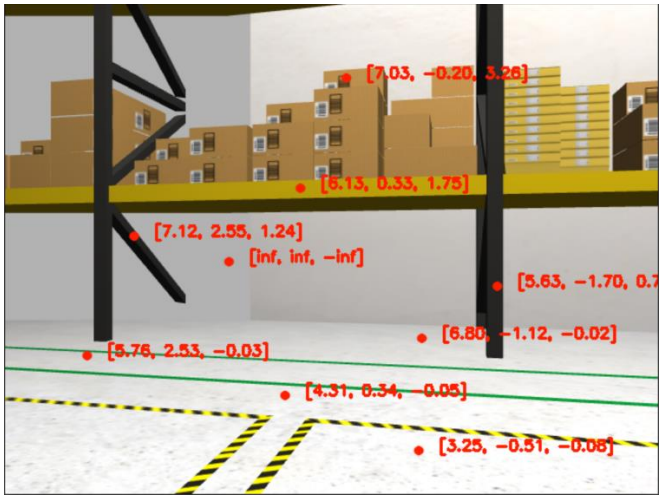
Proyección 2D a 3D

En la opción 2 es necesario calcular las proyecciones de los puntos seleccionados en la imagen. Dado que al transformar de 3D a 2D la distancia en profundidad no se puede calcular, es necesario utilizar la imagen de profundidad que nos proporciona el Tiago.

Hay que tener en cuenta que esta imagen al tratarse de una imagen de profundidad, los valores que son distancias, de modo que, si el formato obtenido es en UINT16, esta estará expresada en milímetros, mientras que si es FLOAT32 será en metros.

Así pues, haciendo uso de los parámetros intrínsecos de la cámara y de la distancia en profundidad obtenida de dicha imagen, se puede calcular la proyección de un punto 2D (x,y) de la imagen en el mundo real 3D (X,Y,Z).

Sobre la imagen se mostrarán los puntos seleccionados con el ratón, cuyas **distancias** estarán en metros con respecto a **basefootprint**. Si no es posible obtener la profundidad, se indicará como inf (infinito).



Extracción de planos con PCL

1. **Filtro de color**

Utilizando el PointCloud de entrada en formato **pcl::PointXYZRGB**, se pide crear un filtro de color que únicamente muestre el color de los elementos a identificar.

En caso de necesitar hacer un cambio del espacio de color, existen múltiples funciones implementadas dentro de PCL que nos facilitan el trabajo. Para más detalles, ver:

https://pointclouds.org/documentation/namespacepcl.html

Dependiendo de la precisión del filtrado, puede que haya valores anormales (outliers).

2. **Eliminación de valores outliers**

Para eliminar los valores outliers, existen una serie de filtros, como el estadístico. El uso de este filtro puede verse aquí:

https://pcl.readthedocs.io/projects/tutorials/en/latest/statistical_outlier.html

Este programa genera dos nubes de puntos con los inliers y outliers detectados por el filtro estadístico. Nosotros nos quedaremos únicamente con los inliers, cuyo resultado será la nube de puntos obtenida tras el filtro de color sin outliers.

3. **Detección de planos**

En este apartado, se pide detectar tantos planos como baldas haya en las estanterías, y de cada una extraer los coeficientes que pertenecen al mismo.

Una vez detectados, se dibujará un plano que englobe todos los puntos detectados. Para ello, habrá que detectar los valores máximo y mínimo de la nube de puntos donde se haya extraído el plano.

En la detección de los planos, se quiere utilizar **RANSAC**. Para ello hay varias formas de hacerlo, bien utilizando RandomSampleConsensus, o bien mediante extracción de índices de forma genérica SACSegmentation.

Un ejemplo de **RandomSampleConsesus**, donde se obtiene la nube de puntos que mejor se ajusta a un modelo tanto de plano, como de esfera, puede verse aquí:

https://pcl.readthedocs.io/projects/tutorials/en/latest/random_sample_consensus.html

En el caso de la extracción de índices con **SACSegmentation**, en el tutorial de PCL tienes un ejemplo donde **se extraen planos** hasta que la nube de puntos original es reducida a menos del 30% de la nube de puntos original:

https://pcl.readthedocs.io/projects/tutorials/en/latest/extract_indices.html

En dicho ejemplo, se utiliza un objeto **seg** de la clase **pcl::SACSegmentation<pcl::PointXYZ>** para la segmentación del plano. El resultado de la segmentación **seg.segment()** son los coeficientes de la ecuación del plano y una lista de índices a los puntos de la nube que pertenecen a dicho plano.

Observa que para recuperar dichos puntos a partir de la lista de índices, se utiliza un filtro **extract** del tipo **pcl::ExtractIndices<pcl::PointXYZ>**. Aunque en nuestro caso, todos los puntos serán **pcl::PointXYZRGB**.

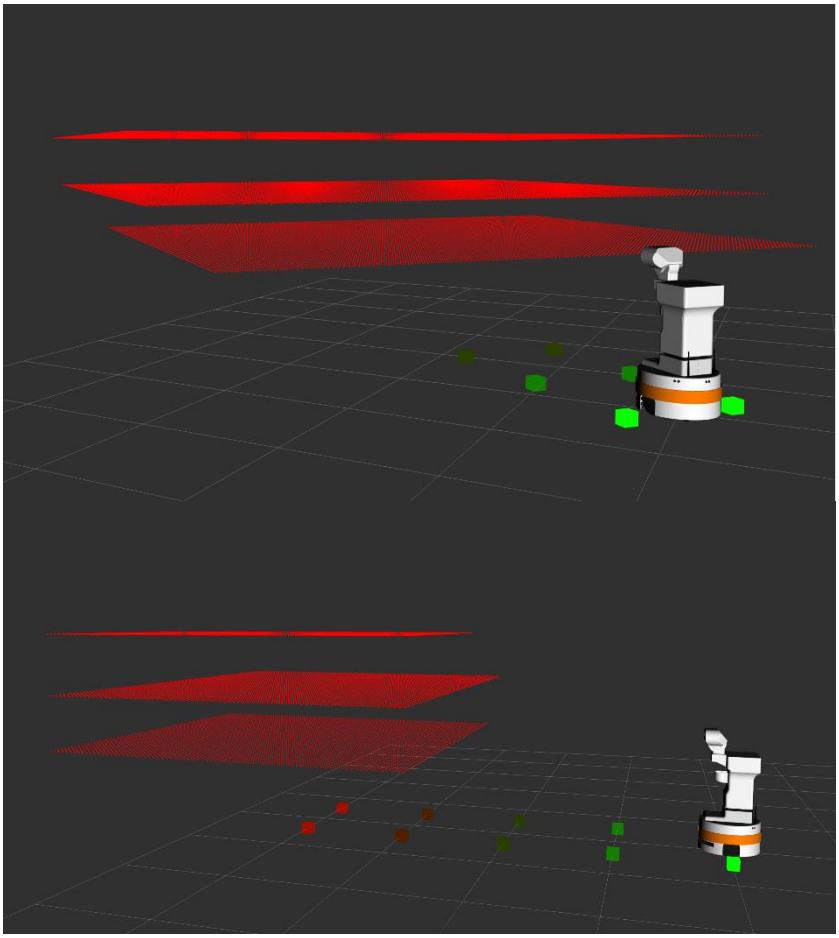
En el caso de necesitar ajustar a otros modelos, los proporcionados por PCL pueden verse aquí:

https://pointclouds.org/documentation/namespacepcl.html#a801a3c83fe807097c9aded5534df1394

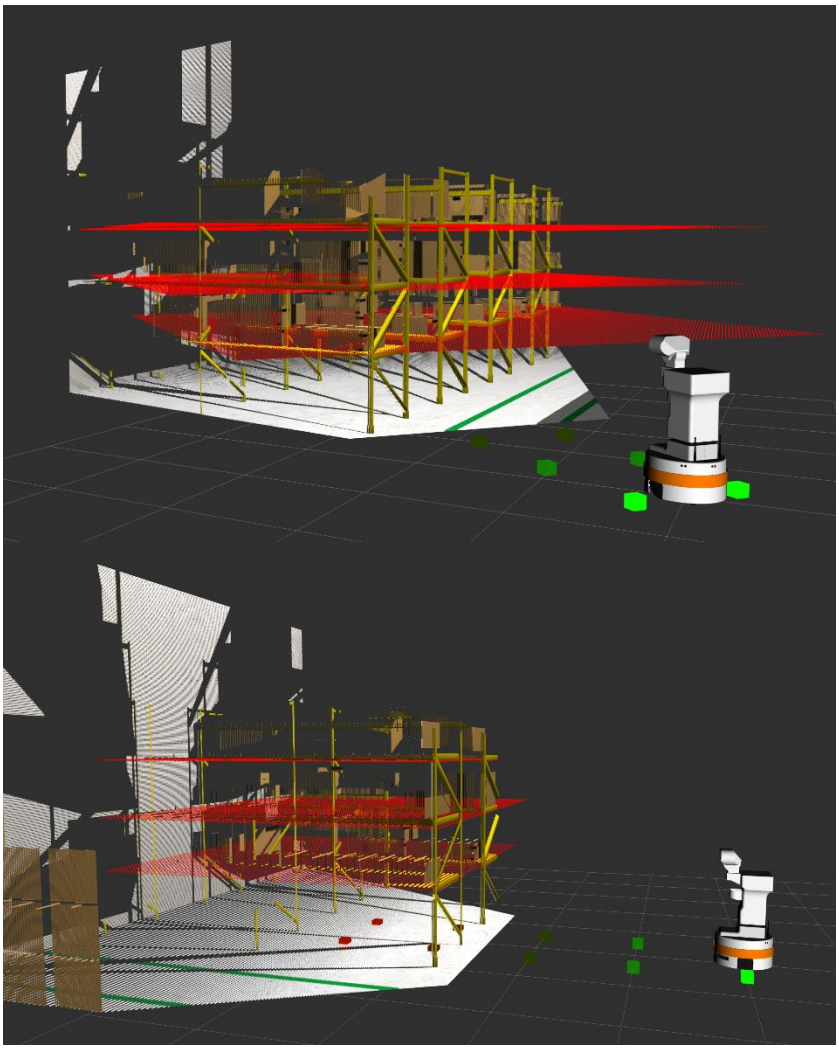
4. Incluir cubos

Finalmente, se pide incluir cubos en las posiciones 3D que tenían los puntos utilizados en el apartado 2, los usados para generar proyecciones. En este caso, se dibujarán únicamente un cubo por cada punto de los exteriores, los cuales estarán distanciados **0.45 metros a cada lado del robot**, e irán a una distancia frontal del robot desde los 0 metros hasta los 8 metros, separadas 1 metro entre sí en función del valor del slider **Distance**. Además, los cubos tendrán colores según su distancia (misma línea, mismo color).

Unas imágenes del resultado final pueden verse a continuación:



Si se superpone el PointCloud original, puede verse lo siguiente:



Ayuda

lookupTransform:
https://docs.ros.org/en/foxy/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Listener-Cpp.html

PinHole model Camera:
https://docs.ros.org/en/api/image_geometry/html/c++/classimage__geometry_1_1PinholeCameraModel.html

Stack de procesamiento de imágenes en ROS
https://docs.ros.org/en/rolling/p/image_pipeline/index.html

OpenCV projectPoints
https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga1019495a2c8d1743ed5cc23fa0daff8c

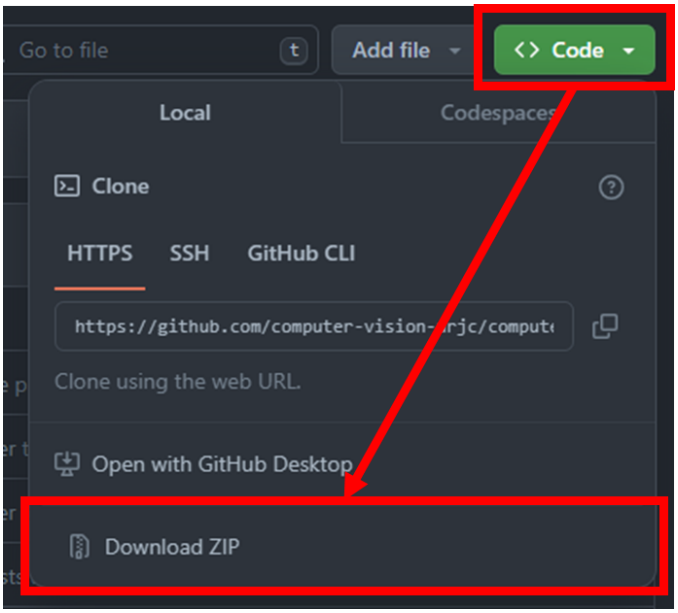
Memoria

Deberá modificarse el README y añadir un apartado para responder los siguientes puntos:

1. En el apartado 2, se han generado una serie de puntos cuya distancia es con respecto a basefootprint. ¿Con respecto a qué frame de la cámara has realizado los cálculos para la obtención de los parámetros extrínsecos? Justifica tu respuesta.
2. Una vez aplicado RANSAC en PCL. ¿Cómo has identificado que el plano obtenido pertenece en realidad a una de las baldas de las estanterías?

Entrega

La **entrega** consistirá en subir al **Aula Virtual** el **archivo .zip** generado a través del repositorio de GitHub Classroom.



Para ello, será necesario que cada integrante del grupo esté dentro del curso, si no lo está, tiene que entrar en el siguiente enlace, y asociarse con el usuario creado a partir de su email de la URJC. Una vez hecho esto, deberá entrar en el grupo de la asignatura, si no existe, uno de los miembros tendrá que crearlo, siendo el nombre del grupo el mismo que figura en el Aula Virtual.

La plantilla con el nodo ROS 2 proporcionada, deberá ser modificada para que el **nombre del paquete** sea **practica3-grupoX**, donde X es el número de grupo del Aula Virtual. En este caso, es posible **modificar el archivo cabecera (.hpp)** de la plantilla que se proporciona para incluir únicamente los atributos necesarios que permiten obtener las transformaciones que darán lugar a los parámetros extrínsecos.

Si no se cumplen los criterios anteriores, o se entrega un paquete que no compila, la calificación será 0.