Universidad Miguel Hernández de Elche

MASTER UNIVERSITARIO EN ROBÓTICA

“Detección, segmentación y cálculo de la profundidad de objetos a partir de la información proporcionada por una cámara RealSense”

Trabajo de Fin de Máster

2021 - 2022

Autor: Jorge Francisco Aznar Ortega
Tutor/es: Óscar Reinoso García

ÍNDICE

AGRADECIMIENTOS

En primer lugar, quiero agradecer a Óscar Reinoso que ofertara este tema como Trabajo Fin de Máster, así como que me permitiera realizarlo. Durante el desarrollo del mismo he aprendido acerca de temas sobre los que no tenía un gran conocimiento, extendiendo de este modo las competencias adquiridas durante el máster. También he de agradecerle la ayuda que me ha brindado en los momentos en los que he tenido dudas acerca de cómo avanzar en el trabajo.

En segundo lugar, mi reconocimiento a mi familia por apoyarme durante mis años de formación universitaria. Ellos me animaron a seguir adelante en los momentos difíciles y contribuyeron a que el camino fuese más fácil. Debo mencionar también a mis amigos, por hacerme más llevaderos los cursos académicos y ayudarme cuando lo he necesitado.

Finalmente, dar las gracias a todos los profesores que he tenido durante mi etapa en la universidad por brindarme durante estos cinco años los conocimientos necesarios para completar mis estudios sin grandes dificultades.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

En este trabajo se lleva a cabo el desarrollo de un *software* que utiliza los datos de una cámara RGB-D para detectar y segmentar los objetos captados, así como hallar la distancia a la que se encuentra cada uno de ellos.

Durante el transcurso del trabajo se han testado varios métodos para obtener las mencionadas distancias. Todos ellos se exponen con sus pros y sus contras.

El documento está estructurado en siete capítulos y un anexo. El primer capítulo es una introducción. En el segundo se hablará de las cámaras de profundidad y sus usos.

En el tercer capítulo se comentan las características específicas del modelo utilizado. En el cuarto se exponen los prerrequisitos necesarios para poner en funcionamiento el programa creado. El quinto capítulo muestra y explica el código desarrollado.

En el sexto se presentan imágenes de los resultados. En el séptimo capitulo se mencionan algunas posibles mejoras para corregir carencias del programa. Para finalizar, se incluye un anexo con las referencias bibliográficas consultadas.

ABSTRACT

In this work, the development of a software that uses the data from an RGB-D camera to detect and segment the captured objects, as well as to find the distance at which each of them is located, is carried out.

During the course of the work, several methods have been tested to obtain these distances. All of them are presented with their pros and cons.

The document is structured in seven chapters and an annex. The first chapter is an introduction. In the second chapter, depth cameras and their uses are discussed.

The third chapter describes the specific characteristics of the model used. Next, the prerequisites necessary to put the created program into operation are presented. The fifth chapter shows and explains the code developed.

In the sixth chapter, images of the results are presented. The seventh chapter mentions some possible improvements to correct the program's shortcomings. Finally, an appendix is included with the bibliographical references consulted.

1. INTRODUCCIÓN

El aumento del uso de robots móviles y autónomos ha propiciado el desarrollo y mejora de los sistemas de captación y sensores, con el propósito de que estos sean capaces de proporcionar la mayor cantidad y calidad de información posible a dichos robots. Con esos datos serán capaces de ubicarse en el entorno que les rodea y actuar correctamente ante las complicaciones que se presenten. Existe en el mercado una gran cantidad de tipos de sensores que ayudan a la localización de los robots, tales como GPS, lídares, ultrasónicos, etc. El dispositivo utilizado en este trabajo es una cámara RGB-D, cuya particularidad reside en que capta la profundidad de las escenas adicionalmente a las imágenes propiamente dichas.

1.1 OBJETIVOS

El propósito de este Trabajo Fin de Máster es el desarrollo de un programa que sea capaz de detectar y segmentar los objetos presentes y hallar la distancia entre el robot y cada uno de ellos. Las imágenes y la información de profundidad se capturan mediante una cámara RealSense D435i de Intel montada sobre un robot móvil. La detección y la segmentación son realizadas por un código en Python basado en *Detectron2*, una biblioteca de inteligencia artificial desarrollada por Facebook. Una vez detectados los elementos de cada escena, se utilizan la información de profundidad y las cajas de detección o máscaras de segmentación (según el modo que se elija) para asignar las distancias.

2. ESTADO DEL ARTE

La finalidad de esta sección es profundizar en el conocimiento acerca de las cámaras de profundidad, los tipos de tecnologías que existen y sus diferencias con las cámaras normales. También se comentan algunos usos de estos dispositivos.

2.1 CÁMARAS DE PROFUNDIDAD

De acuerdo a lo explicado en [1], las cámaras digitales estándar proporcionan imágenes en la forma de una matriz 2D de píxeles. Cada uno de estos píxeles posee tres valores numéricos asociados, comprendidos entre el 0 y el 255. Agrupando un gran conjunto de píxeles se crean las fotografías con las que estamos familiarizados. Por otro lado, una cámara de profundidad genera píxeles que adicionalmente contienen el valor de la distancia al objeto (profundidad). En nuestro caso, la cámara dispone de un sistema RGB y otro de profundidad, que combinados proporcionan píxeles con los cuatro valores (o RGBD).

Existen diferentes tecnologías para calcular la profundidad. A continuación se exponen varios tipos de cámaras y sus principios de funcionamiento.

2.1.1 Luz estructurada y luz codificada

Se basan en la proyección de luz, normalmente infrarroja, desde un emisor hacia la escena. Se conoce el patrón de la luz proyectada, por lo que la forma en la que el sensor observa dicho patrón proporciona la información de profundidad.

Utilizando la disparidad entre la imagen esperada y la realmente vista por la cámara, se puede calcular la distancia desde el dispositivo hasta cada píxel.

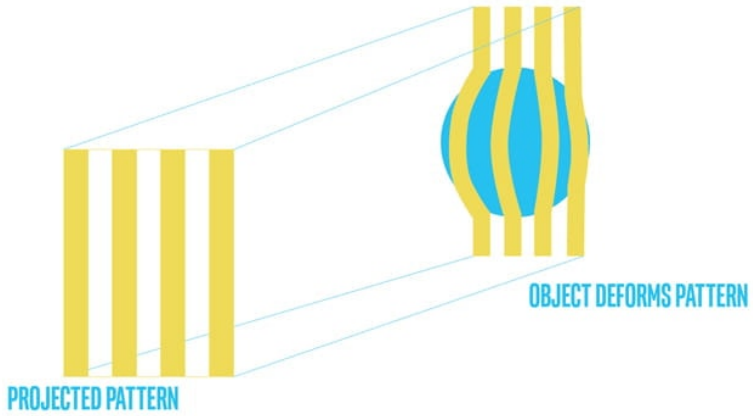

Figura 1: *Luz codificada y luz estructurada.*

2.1.2 Profundidad estéreo

Las cámaras de profundidad estéreo también suelen proyectar luz (en este caso, de cualquier tipo) sobre la escena para medir la profundidad. Estos aparatos disponen de dos sensores separados por una pequeña distancia conocida. Con cada uno de ellos se toma una imagen y se comparan entre sí para obtener la profundidad. Además, la separación entre los dos sensores determina la máxima distancia que se puede capturar: a mayor línea de base, mayor distancia de visualización.

Estos dispositivos funcionan adecuadamente en la mayoría de condiciones de iluminación, incluso en exteriores.

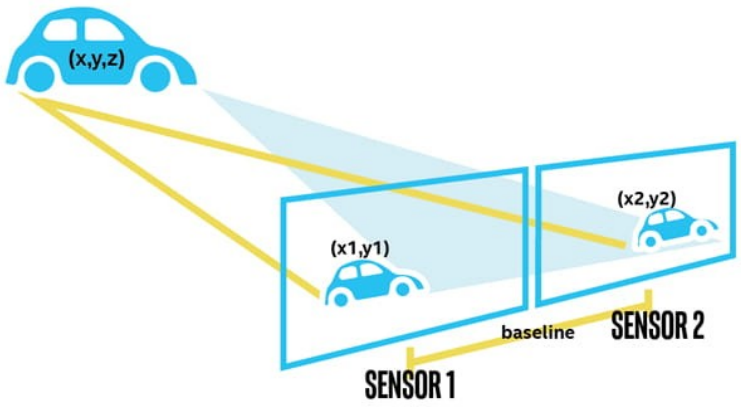

Figura 2: *Profundidad estéreo.*

2.1.3 Tiempo de vuelo Y LiDAR

En las cámaras de tiempo de vuelo, la velocidad de la luz es la variable utilizada para calcular la profundidad. Todos los dispositivos de esta clase emiten algún tipo de luz, hacen un barrido sobre la escena y cronometran cuánto tarda esa luz en volver al sensor de la cámara. En el caso de los LiDAR, utilizan luz láser para su funcionamiento.

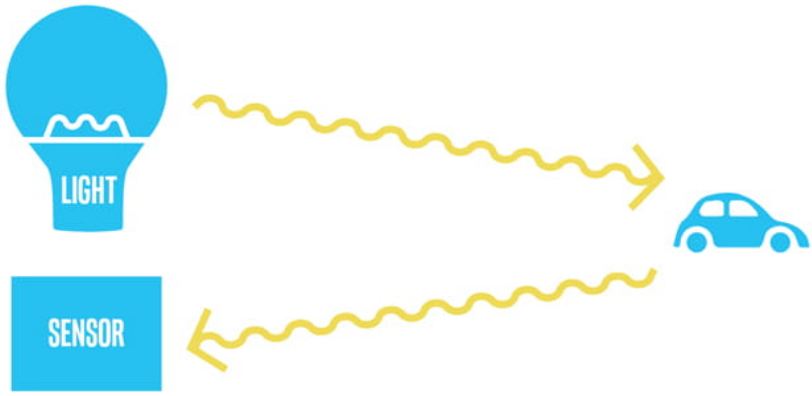

Figura 3: *Tiempo de vuelo y LiDAR.*

2.2 USOS

Las cámaras de profundidad proporcionan comprensión adicional sobre la escena, lo que en ocasiones permite que no sea necesaria la intervención humana para determinadas tareas.

Estos dispositivos son muy útiles en el campo de la robótica y los dispositivos autónomos. En [2] se exponen algunos usos concretos, como los siguientes:

2.2.1 Robótica

Mientras la visión artificial siga restringida a imágenes planas, limitando la interacción y la comprensión del entorno, las cámaras de profundidad tienen la capacidad de interpretar mejor el contexto y responder ante imprevistos.

2.2.2 Escaneo 3D

Estas cámaras permiten escanear una persona, un objeto o un lugar y crear modelos precisos de una forma rápida y sencilla.

2.2.3 Reconocimiento facial

Al agregar profundidad a un sistema de reconocimiento facial, no es posible burlarlo utilizando fotografías. Además, pueden identificar a una persona desde múltiples ángulos, sin necesidad de una alineación específica con el sensor.

2.2.4 Drones

Este tipo de cámaras contribuye a la mejora de la seguridad, la prevención de colisiones y el reconocimiento de obstáculos. Esto cobra gran importancia ante la creciente popularidad de los drones.

2.2.5 Medición de objetos

Las cámaras de profundidad permiten obtener datos de alta precisión que se pueden usar para medir con facilidad objetos de cualquier tamaño.

3. HARDWARE

Como ya se ha mencionado, el dispositivo utilizado para la captación de las imágenes y la información de profundidad ha sido la cámara RealSense D435i de la empresa Intel. De acuerdo a los datos proporcionados por el fabricante, [3], este modelo es adecuado para aplicaciones en las que la cámara deba estar en movimiento, tales como la navegación robótica o el reconocimiento de objetos. Además, por su amplio campo de visión, produce pocos puntos ciegos. Por último, la cámara posee una gran sensibilidad con poca luz, lo que permite a los robots moverse por espacios con escasa o nula iluminación.

A continuación se muestra una imagen de este dispositivo:

Figura 4: *Cámara RealSense D435i.*

La estructura de la cámara se puede observar de forma detallada en la siguiente figura:

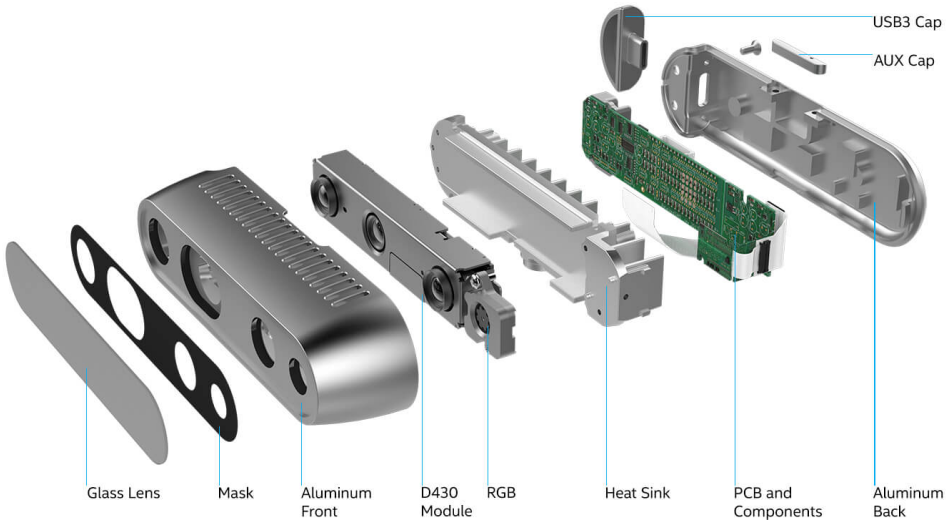

Figura 5*: Estructura de la cámara.*

Las características de la misma se muestran en la siguiente tabla:

Características	- Entornos de uso: interior/exterior. - Tecnología del sensor de imagen: *global shutter*. - Rango ideal: de 0,3 a 3 metros.
Profundidad	- Tecnología de profundidad: estereoscópica. - Mínima distancia de profundidad a la máxima resolución: ~28 cm. - Campo de visión: 87° × 58°. - Resolución de salida: hasta 1280 × 720. - Tasa de *frames*: hasta 90 fps.
RGB	- Resolución de la cámara RGB: 1920 × 1080. - Tasa de *frames*: 30 fps. - Tecnología del sensor RGB: *rolling shutter*. - Campo de visión del sensor RGB (H × V): 69° × 42°. - Resolución del sensor RGB: 2 MP.
Componentes principales	- Módulo de la cámara: Intel RealSense Module D430 + cámara RGB. - Placa del procesador de visión: procesador Intel RealSense Vision D4.
Características físicas	- Factor de forma: cámara periférica. - Longitud × anchura × altura: 90 mm × 25 mm × 25 mm. - Conectores: USB-C 3.1 Gen 1.

Tabla 1: *Características cámara RealSense.*

Fuente: *[3].*

4. REQUERIMIENTOS

Todas las pruebas y el desarrollo de los programas se han llevado a cabo en un ordenador con el sistema operativo Linux, concretamente con la distribución Ubuntu 20.04. También podría haberse realizado sobre macOS, aunque con algunas diferencias en la instalación de paquetería. La versión de Python debe ser posterior a la 3.7, habiéndose utilizado la 3.9. También se debe disponer de los compiladores `gcc` y `g++` en una versión igual o superior a la 5.4, contando el equipo de trabajo con la 9.4.0.

Para descargar las librerías es necesario el gestor de paquetes `pip3`. Si no se dispone de él, se puede obtener mediante el comando `apt-get install python3-pip`.

Cumpliendo estos prerrequisitos, a continuación se deben descargar las librerías necesarias para poder ejecutar correctamente el código desarrollado en el presente trabajo, que se listan a continuación:

- Librería *PyRealSense*, que se puede descargar mediante el comando `pip3 install pyrealsense2`.

- Librería *OpenCV*, necesaria para ejecutar la demo de *Detectron2*, visualizar y guardar los resultados y otras funciones como la media aritmética. Se puede descargar mediante el comando `pip3 install opencv-python`.

- De manera opcional se puede instalar *Ninja* para una construcción mas rápida de *Detectron2*, con el comando `pip3 install ninja`.

- La librería *Numpy* es un prerrequisito de *Pytorch* y también será utilizada para otros fines. Para instalarla, `pip3 install numpy`.

- Librería *PyTorch* en su versión 1.8 o superior, así como una versión de la librería *torchvision* que sea acorde a la de *PyTorch*.

La forma recomendada de instalar las dos librerías de *PyTorch* es descargarlas de forma conjunta desde [4]. En esta página aparecen una serie de opciones seleccionables que permiten obtener el comando de instalación más adecuado para cada entorno. Las alternativas disponibles se muestran en la siguiente tabla:

PyTorch Build	Stable (1.12.0)		Preview (Nightly)		LTS (1.8.2)	
Your OS	Linux		Mac		Windows	
Package	Conda	Pip	LibTorch		Source	
Language	Python		C++ / Java			
Compute Platform	CUDA 10.2	CUDA 11.3	CUDA 11.6	ROCm 5.1.1	CPU	

Tabla 2: *Alternativas seleccionables para la instalación de Pytorch.*

Fuente: *[4].*

Para este trabajo, las opciones seleccionadas han sido respectivamente: *Stable (1.12.0)*, *Linux*, *Pip*, *Python* y *CPU*. Al seleccionarlas, la web proporciona el siguiente comando:

```
pip3 install torch torchvision torchaudio --extra-index-url
https://download.pytorch.org/whl/cpu
```

A continuación se puede proceder con la instalación de *Detectron2* ejecutando los dos comandos siguientes [5]:

```
git clone https://github.com/facebookresearch/detectron2.git
python3 -m pip install -e detectron2
```

Una vez ejecutadas las instrucciones anteriores debemos disponer de una instalación de *Detectron2* completamente funcional, que podemos verificar lanzando el programa *demo.py* que incluye.

En el supuesto de que el *script* de prueba funcione de forma correcta, el código desarrollado en este trabajo debería ejecutarse sin inconvenientes.

5. CÓDIGO DESARROLLADO

En este apartado se explica en detalle el código que se ha desarrollado para este trabajo. En primer lugar cabe destacar que, pese a que en el repositorio de *Detectron2* [6] se puede encontrar una versión de prueba para testear la biblioteca, dicho código no se ha empleado como punto de partida para el presente desarrollo. En su lugar se ha optado por utilizar como referencia el código de *TheCodingBug* [7] para la parte de detección y segmentación, añadiendo y modificando algunas partes. Después se ha extendido la funcionalidad del programa, incorporando todo el nuevo código de profundidad.

La biblioteca de *Detectron2* está desarrollada en Python, así que el resto de código también se ha escrito en este lenguaje. Esto retrasó el inicio del desarrollo puesto que fue necesario ampliar los conocimientos sobre este, sucediendo lo mismo con *OpenCV*. Como punto favorable, se debe destacar la disponibilidad de una gran cantidad de funciones en el repositorio de *Realsense* [8] para diferentes lenguajes de programación, entre ellos Python. En lo referente a *OpenCV* se halló toda la información necesaria en la documentación disponible en su web, así como en foros.

El cálculo de la profundidad ha requerido de la mayor parte del tiempo dedicado a este TFM. A pesar de que se podían encontrar algunos códigos en Internet que cumplían con la finalidad buscada, ninguno funcionaba correctamente en las pruebas realizadas. Incluso tras llevar a cabo modificaciones sobre ellos no se consiguieron resultados adecuados en tiempo real o con ejecuciones periódicas. Es necesario remarcar que los procesos necesarios para realizar la tarea pretendida son muy intensivos computacionalmente. Se pretendía utilizar/desarrollar un programa diseñado para ordenadores con requerimientos medios, sin grandes capacidades gráficas, tanto por eficacia como por compatibilidad con una mayor cantidad de dispositivos.

El *software* creado para la tarea se compone de dos partes diferenciadas: la de detección y segmentación y la de cálculo de profundidad. Tras experimentar con diversas opciones durante la búsqueda de una base apropiada y realizar algunas modificaciones sobre esta, la primera parte concluyó con relativa celeridad, obteniendo un rendimiento admisible en ejecuciones cada pocos segundos (a partir de 3 aproximadamente).

Para la segunda parte, tras una búsqueda infructuosa en la Red, se decidió llevar a cabo un análisis de los códigos de profundidad disponibles en el repositorio de Intel [8]. A partir de ellos y con la ayuda de la documentación de *Detectron2* referente a los datos devueltos por el predictor, se averiguó cómo asignar la profundidad a cada objeto.

En lo que concierne a *Detectron2,* este dispone de varios modos de detección y segmentación. Este trabajo se centrará en *object detection* e *instance segmentation*. El primero consiste en situar una caja de detección sobre cada objeto identificado, además de su nombre y el porcentaje de certeza de la predicción. En cuanto al segundo, proporciona todos los datos del caso anterior añadiendo una máscara de segmentación sobre cada objeto detectado.

A continuación se muestran ejemplos (fuente: [9]) del resultado devuelto por cada modelo:

Figura 6: *Imagen de entrada 1.*

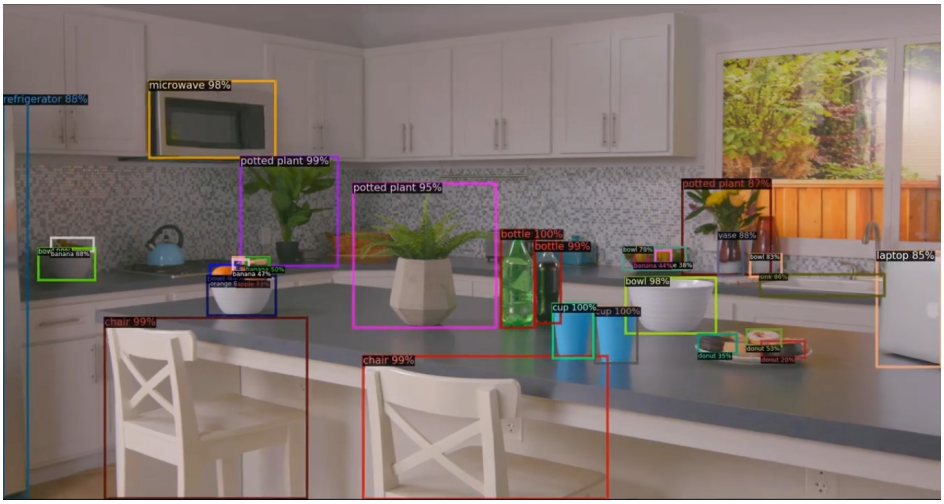

Figura 7: *Imagen de salida con el modelo 'object detection'.*

Figura 8: *Imagen de entrada 2.*

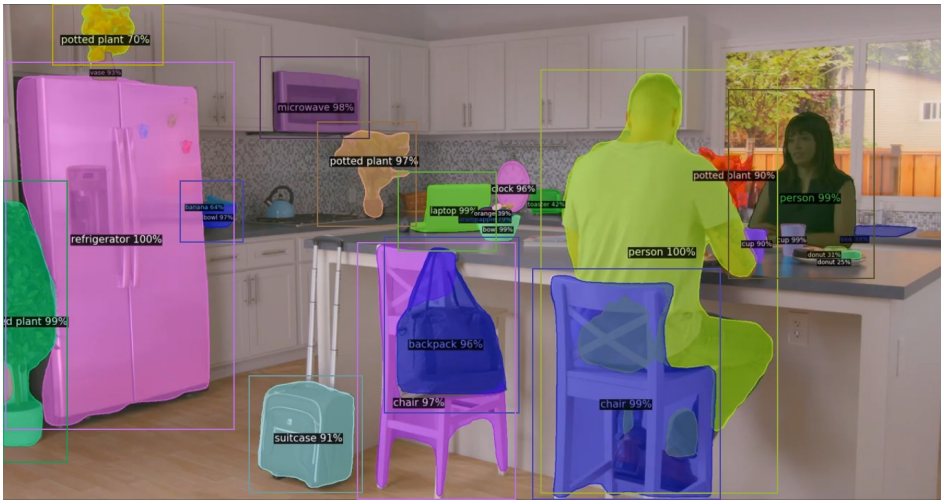

Figura 9: *Imagen de salida con el modelo 'instance segmentation'.*
Fuente: *Versión modificada de [9].*

Acto seguido se procederá a explicar y mostrar el *software* desarrollado, dividido en dos ficheros. El primero de ellos se llama *main.py.* Para ejecutarlo es necesario proporcionarle un parámetro que indica cada cuántos segundos se realizará la detección o segmentación, así como la profundidad de los objetos de la escena. Un ejemplo de ejecución por terminal para obtener resultados cada cinco segundos sería el siguiente:

```
user@user:~$ python3 main.py 5
```

Figura 10: *Ejecución desde la terminal.*

Dicho esto, se pasa a explicar el código. La mayor parte posee comentarios explicando su función, que también se comentará en este documento.

Al principio se cargan las bibliotecas necesarias e importan las clases y funciones desde el otro archivo desarrollado (*Detector_profundidad.py*, sobre el cual se profundizará más adelante):

```
1
2   # main.py
3
4   from Detector_profundidad import *              # Importación de las funciones y clases del archivo Detector_profundidad.py.
5   import pyrealsense2 as rs                       # Importación de las bibliotecas necesarias.
6   import numpy as np
7   import time
8   import sys
9   import cv2
10
```

Figura 11: *Imagen 1 del archivo main.py.*

Después, el código de la Figura 12 crea una instancia de un objeto de tipo *Detector_profundidad*, clase creada en el otro fichero, y las siguientes líneas habilitan la configuración y asignan valores a algunas variables.

```
10
11  detector = Detector_profundidad(model_type="IS")    # Creación de una instancia de la clase Detector_profundidad.
12
13  RESOLUTION_X = 1280                                 # Resolución horizontal de la imagen que capturará la cámara.
14  RESOLUTION_Y = 720                                  # Resolución vertical de la imagen que capturará la cámara.
15
16  config = rs.config()                                # Setup
17
18  # Habilitación de los streams de profundidad y color.
19  # El formato rs.format.bgr8 indica que los datos de color son tres canales de ocho bits.
20  # Hay que tener en cuenta el formato bgr (azul, verde, rojo) para usar OpenCV.
21  # El formato rs.format.z16 indica que la información de profundidad es un número entero de 16 bits sin signo.
22
23  config.enable_stream(rs.stream.color, RESOLUTION_X, RESOLUTION_Y, rs.format.bgr8, 30)  # El número 30 representa la tasa de frames.
24  config.enable_stream(rs.stream.depth, RESOLUTION_X, RESOLUTION_Y, rs.format.z16, 30)   # El número 30 representa la tasa de frames.
25
26  pipeline = rs.pipeline()                                                # Creación del pipeline.
27  profile = pipeline.start(config)                                        # Comienzo del streaming.
28  align = rs.align(rs.stream.color)                                       # Creación del objeto de alineación.
29  depth_scale = profile.get_device().first_depth_sensor().get_depth_scale()  # Cálculo de la escala de profundidad.
30
31  #Los 'frames' iniciales no son útiles porque durante los primeros segundos la cámara ajusta la exposición.
32  print("Inicio del ajuste de auto-exposición")
33  for x in range(30):
34      pipeline.wait_for_frames()                  # Mediante este bucle descartamos los primeros 30 frames.
35  print("Ajuste de auto-exposición finalizado")
36
```

Figura 12: *Imagen 2 del archivo main.py.*

Se debe tener en cuenta que no todas las resoluciones pueden ser combinadas con todas las tasas de *frames*. Los ajustes definidos en las líneas 23 y 24 pueden producir errores de configuración si no se utilizan duplas soportadas. Sin salir de estas configuraciones, cabe mencionar la posibilidad de utilizar una menor resolución de imagen a cambio de poder capturar más *frames* por segundo. Para la tarea encomendada, es recomendable utilizar como mínimo una tasa de 30 fps. De otra forma, es probable que la imagen capturada en movimiento esté borrosa.

El proceso de ajuste automático de la exposición de las líneas 32-35 es necesario, puesto que los resultados de los primeros *frames* capturados por la cámara tras ser encendida proporcionan información inexacta. Este fenómeno se produce porque el sensor debe adaptarse a las condiciones lumínicas del entorno.

En la siguiente tabla se recogen las imágenes en color y los mapas de profundidad capturados en los primeros instantes de funcionamiento de la cámara:

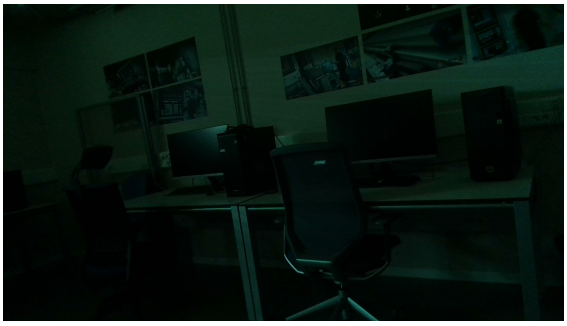

Figura 13: *Imagen RGB inicial.*

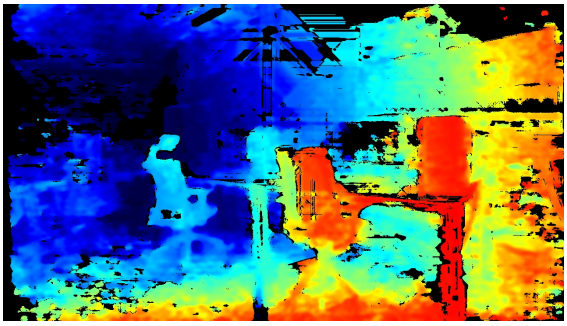

Figura 14: *Mapa de profundidad inicial.*

Figura 15: *Imagen RGB intermedia.*

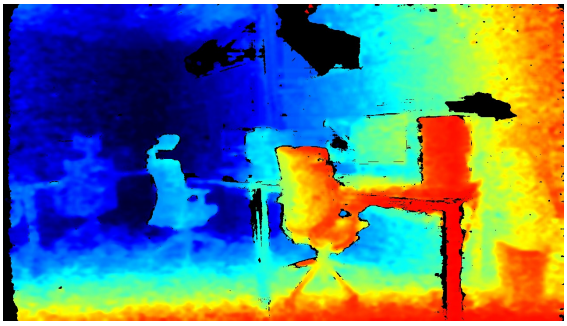

Figura 16: *Mapa profundidad intermedio.*

Figura 17: *Imagen RGB final.*

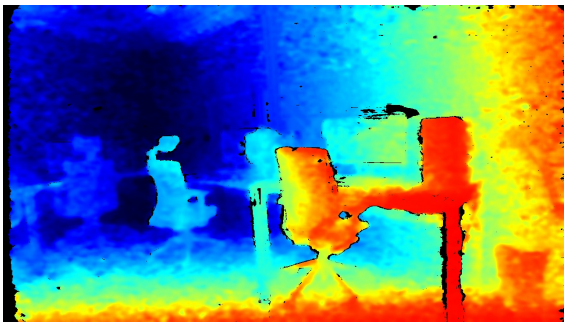

Figura 18: *Mapa de profundidad final.*

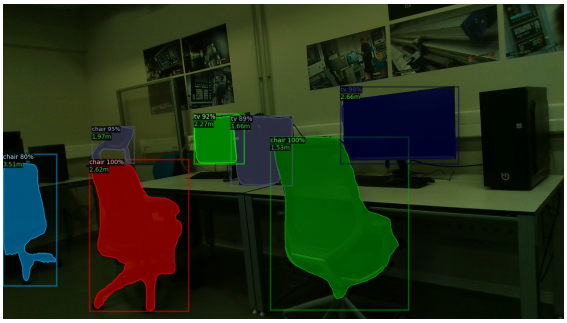

Figura 19: *Primeras segmentaciones.*

Figura 20: *Segmentaciones finales.*

Tabla 3: *Comparaciones.*

Al utilizar esta información inicial como entrada para el código, se detectan menos objetos y las profundidades distan de las reales.

Avanzando en el archivo, se dispone de una sentencia condicional para verificar que se pasa un parámetro por terminal. En caso contrario, se utiliza un valor de cinco segundos por defecto. Independientemente de esto, como el tiempo de cálculo del resultado del programa es de alrededor de dos segundos, se debería proporcionar como periodo un valor mínimo de 3.

```
36
37  if len(sys.argv) == 2:              # Si se pasa un único parámetro desde la terminal, se usará como periodo.
38      periodo = float(sys.argv[1])
39  else:                               # En otro caso, se utilizará un valor de cinco segundos como periodo.
40      periodo = 5
41
```

Figura 21: *Imagen 3 del archivo main.py.*

Por último, en este fichero se crea un bucle que lee los valores de profundidad y color de los datos capturados por la cámara, que se pasan como argumento a una función definida en el otro archivo. Esta información se alinea para que no haya incoherencias entre los datos de profundidad y los de color si las dos fuentes tuviesen distinta resolución.

```
41
42  try:                            # El código se ejecutará hasta que reciba el comando 'Ctrl + C'.
43      while True:
44
45          # Obtención de un conjunto de frames de color y profundidad y alineación de estos.
46          frames = pipeline.wait_for_frames()
47          aligned_frames = align.process(frames)
48
49          # Obtención de los frames alineados anteriormente.
50          depth_image = np.asanyarray(aligned_frames.get_depth_frame().get_data())
51          color_image = np.asanyarray(aligned_frames.get_color_frame().get_data())
52
53          detector.onImage(color_image, depth_image, depth_scale)  # Se llama al método onImage de la instancia detector.
54          time.sleep(periodo)                                      # Se detiene temporalmente el programa para que solo
55                                                                   # se ejecute cada determinados instantes.
56  except KeyboardInterrupt:
57      pass
58
59  pipeline.stop()      # Se detiene la cámara.
60
```

Figura 22: *Imagen 4 del archivo main.py.*

El segundo fichero que compone el programa se muestra en las siguientes capturas. Al igual que en *main.py*, lo primero es cargar las bibliotecas e importar funciones y clases (Figura 23).

```
1
2   # Detector_profundidad.py
3
4   from detectron2.engine import DefaultPredictor      # Importación de funciones y clases de otros archivos.
5   from detectron2.config import get_cfg
6   from detectron2.data import MetadataCatalog
7   from detectron2.utils.visualizer import ColorMode, Visualizer
8   from detectron2 import model_zoo
9   import time                                         # Importación de las bibliotecas necesarias.
10  import cv2
11  import numpy as np
12  import pyrealsense2 as rs                           # API de código abierto multiplataforma Intel RealSense.
13
14  import warnings
15  warnings.filterwarnings("ignore")
16
```

Figura 23: *Imagen 1 del archivo Detector_profundidad.py.*

Seguidamente se muestra la clase *Detector_profundidad*:

```
16
17 class Detector_profundidad:                          # Creación de una clase.
18   def __init__(self, model_type = "OD"):
19     self.cfg = get_cfg()
20     self.model_type = model_type
21
22     # Carga de la configuración del modelo y del modelo preentrenado.
23     if model_type == "OD": #object detection
24       self.cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml"))
25       self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml")
26     elif model_type == "IS": #instance segmentation
27       self.cfg.merge_from_file(model_zoo.get_config_file("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
28       self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
29     elif model_type == "KP": #keypoint detection
30       self.cfg.merge_from_file(model_zoo.get_config_file("COCO-Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml"))
31       self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Keypoints/keypoint_rcnn_R_50_FPN_3x.yaml")
32     elif model_type == "LVIS": #LVIS Segmentation
33       self.cfg.merge_from_file(model_zoo.get_config_file("LVISv0.5-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_1x.yaml"))
34       self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("LVISv0.5-InstanceSegmentation/mask_rcnn_X_101_32x8d_FPN_1x.yaml")
35     elif model_type == "PS": #Panoptic Segmentation
36       self.cfg.merge_from_file(model_zoo.get_config_file("COCO-PanopticSegmentation/panoptic_fpn_R_101_3x.yaml"))
37       self.cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-PanopticSegmentation/panoptic_fpn_R_101_3x.yaml")
38
39     confidence_threshold = 0.7         # Se establece el umbral de confianza en las predicciones.
40
41     self.cfg.MODEL.RETINANET.SCORE_THRESH_TEST = confidence_threshold
42     self.cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = confidence_threshold
43     self.cfg.MODEL.PANOPTIC_FPN.COMBINE.INSTANCES_CONFIDENCE_THRESH = confidence_threshold
44     self.cfg.MODEL.DEVICE = "cpu"      #CUDA en caso de disponer de tarjeta gráfica.
45
46     self.predictor = DefaultPredictor(self.cfg)
47     self.f = 0
48
```

Figura 24: *Imagen 2 del archivo Detector_profundidad.py.*

La primera vez que se usa cada tipo de modelo de detección o segmentación el programa descarga el modelo de Internet. Este proceso depende del tipo de conexión a la Red, aunque suele tardar menos de un minuto.

La función *onImage*, que es invocada en *main.py*, se comentará a continuación:

```
48
49   def onImage(self, color_image, depth_info, scale):    #Creación de una función.
50     start_time = time.time()
51     image = color_image
52     depth = depth_info
53     depth_scale = scale
54
55     if self.model_type != "PS":
56
57       pred = self.predictor(image)
58       predictions = pred['instances'].to("cpu")
59
60       viz = Visualizer(image[:,:,::-1], metadata = MetadataCatalog.get(self.cfg.DATASETS.TRAIN[0]), instance_mode = ColorMode.IMAGE)
61
62       output = viz.draw_instance_predictions(predictions)
63
64       boxes = predictions.pred_boxes                       # Se almacenan en variables los valores de la estructura 'predictions'.
65       area = boxes.area().numpy()
66       boxes_list = predictions.pred_boxes.tensor.numpy()
67       class_list = predictions.pred_classes.numpy()
68       mask_array = predictions.pred_masks.numpy()
69       num_instances = mask_array.shape[0]
70       mask_array = np.moveaxis(mask_array, 0, -1)
71       mask_array_instance = []
72       out = np.zeros_like(image) #Imagen en negro
73
```

Figura 25: *Imagen 3 del archivo Detector_profundidad.py.*

En la línea 58, *predictions* contiene toda la información necesaria para obtener las máscaras, las cajas de detección y otros datos de interés. El código de las líneas 60 y 62 habilita la visualización de los resultados. De requerir más velocidad en la ejecución y no necesitar la visualización de los datos se pueden eliminar o comentar estas dos líneas. Cabe reseñar que la función *Visualizer* no está diseñada para tareas en tiempo real. Una posible solución sería reemplazarla por llamadas a *OpenCV*.

La imagen inferior detalla el código que obtiene la profundidad mediante los tres métodos soportados por este programa.

```
73
74      for i in range(0, num_instances):
75        out = np.zeros_like(image)                                    # Obtención de los puntos de la máscara de segmentación.
76        mask_array_instance.append(mask_array[:, :, i:(i+1)])
77        out = np.where(mask_array_instance[i] == True, 255, out)
78        coords = np.column_stack(np.where(out > 100))
79        dist = 0
80        suma = 0
81
82        for j in range(0, coords.shape[0]):
83              suma = suma + depth[coords[j][0]][coords[j][1]]       # Método 1
84
85        profundidad = suma*depth_scale
86        profundidad = suma/(coords.shape[0])
87        profundidad = profundidad/1000
88
89        xmin_depth = int(boxes_list[i][0])
90        ymin_depth = int(boxes_list[i][1])
91        xmax_depth = int(boxes_list[i][2])
92        ymax_depth = int(boxes_list[i][3])
93
94        x1 = int(xmin_depth + (xmax_depth-xmin_depth)/5)
95        y1 = int(ymin_depth + (ymax_depth-ymin_depth)/5)
96        x2 = int(xmax_depth - (xmax_depth-xmin_depth)/5)
97        y2 = int(ymax_depth - (ymax_depth-ymin_depth)/5)
98
99        depth1 = depth[ymin_depth:ymax_depth,xmin_depth:xmax_depth].astype(float)    # Método 2
100       dist1,_,_,_ = cv2.mean(depth1)
101       dist1 = dist1 * depth_scale
102
103       depth2 = depth[y1:y2,x1:x2].astype(float)                                    # Método 3
104       dist2,_,_,_ = cv2.mean(depth2)
105       dist2 = dist2 * depth_scale
106
```

Figura 26: *Imagen 4 del archivo Detector_profundidad.py.*

Se debe considerar la ubicación de la cámara sobre el robot móvil para aumentar o reducir la distancia al objeto detectado. Es decir, dependiendo de si la cámara se monta en la parte delantera, central o trasera del robot será necesario realizar ajustes para mejorar la precisión del resultado devuelto. Además, por seguridad se puede aplicar una ligera reducción al dato real de distancia a los objetos. Así, el robot actuará suponiendo que estos están más próximos a él.

```
106
107         # En áreas peqeñas, la profundidad se superpone con la predicción. Para estas
108         # áreas, menores de 3000, se cambia la ubicación de la profundidad.
109         # En ocasiones sigue habiendo solapamientos.
110
111         if area[i] >= 3000:
112           viz.draw_text("{:.2f}m".format(profundidad),(boxes_list[i][0]+22,boxes_list[i][1]+17))
113         else:
114           viz.draw_text("{:.2f}m".format(profundidad),(boxes_list[i][0]+22,boxes_list[i][3]+13))
115
116     else:
117       predictions, segmentInfo = self.predictor(image)["panoptic_seg"]
118       viz = Visualizer(image[:,:,::-1], MetadataCatalog.get(self.cfg.DATASETS.TRAIN[0]))
119       output = viz.draw_panoptic_seg_predictions(predictions.to("cpu"), segmentInfo)
120
121     img_name = 'output{0}.png'.format(self.f)
122     cv2.imwrite("/home/arvc/Escritorio/resultados/" + str(img_name), output.get_image()[:,:,::-1])
123     self.f = self.f + 1
124
125     print("elapsed time:", format((time.time()-start_time), ".3f"))
126
127
```

Figura 27: *Imagen 5 del archivo Detector_profundidad.py.*

Finalmente, se guardan en un directorio las imágenes obtenidas como resultado, cada una de ellas con un nombre numerado de forma consecutiva.

Estudiado el código, apliquémoslo sobre la Figura 28:

Figura 28: *Imagen RGB.*

El mapa de profundidad de la escena se puede observar en la Figura 29. Las imágenes de este y los posteriores mapas se han tomado antes de aplicar la alineación de *frames*. Aunque las imágenes RGB y de profundidad parezcan ligeramente distintas, para el cálculo de la distancia sí se ha tenido en cuenta la información alineada.

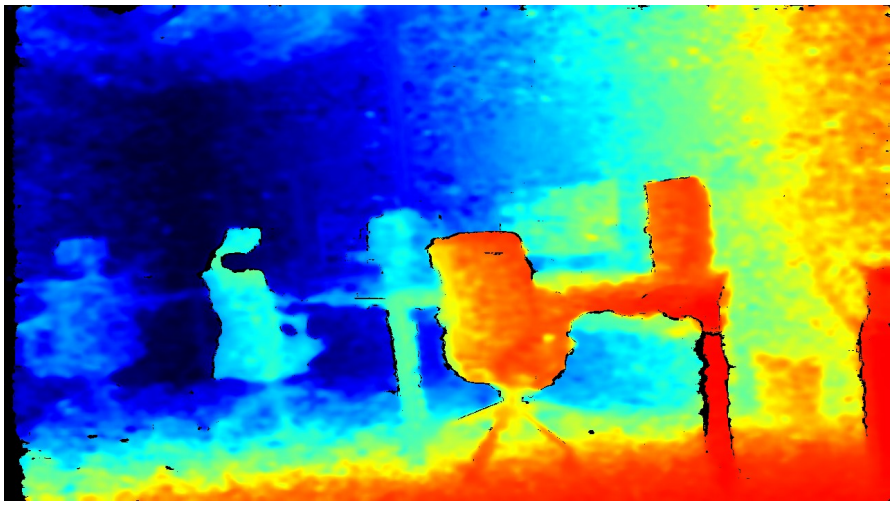

Figura 29: *Imagen de profundidad.*

El resultado proporcionado por el código Python se observa en la Figura 30:

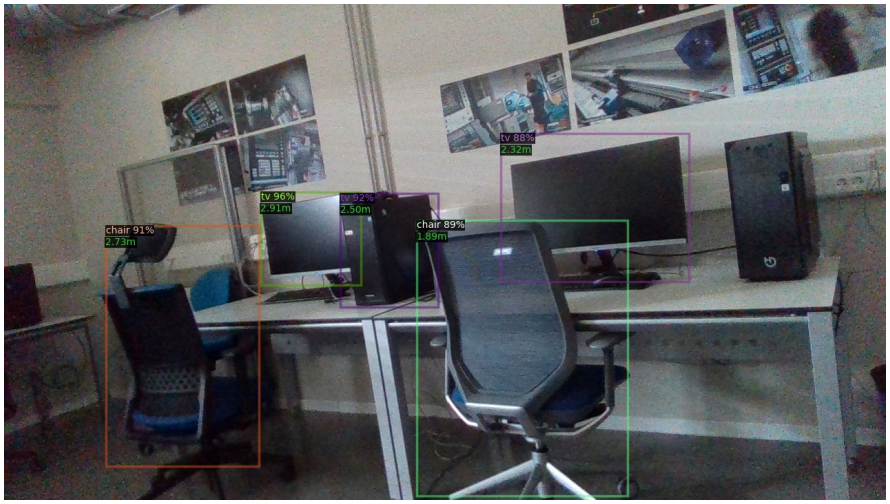

Figura 30: *Imagen de salida tras usar el modelo 'object detection'.*

La salida obtenida es la esperada: una imagen con objetos detectados y la profundidad de cada uno de ellos.

Como ya se ha indicado, el valor de la profundidad de cada objeto se puede obtener de tres maneras distintas. La primera de ellas es la media aritmética de las profundidades que se hallan en el interior de la caja que delimita cada uno de dichos objetos (Figura 31). La segunda es similar a la anterior, pero utiliza una versión reducida del delimitador para disminuir el error provocado al tomar en consideración profundidades que están en la caja pero no propiamente en el objeto (Figura 32). La diferencia en la superficie procesada entre ambos métodos se observa en la (Figura 33). La proporción de reducción de la caja se ha elegido específicamente para la escena representada en la imagen. No obstante, sería conveniente hallar unos porcentajes de reducción idóneos para la mayoría de situaciones.
Por último, la forma más precisa pero menos eficiente en cuanto a tiempo de computación es el cálculo de la profundidad mediante la media aritmética de las profundidades de los puntos incluidos en la máscara de segmentación de cada objeto (Figura 34).

Figura 31: *Cajas de detección por defecto.*

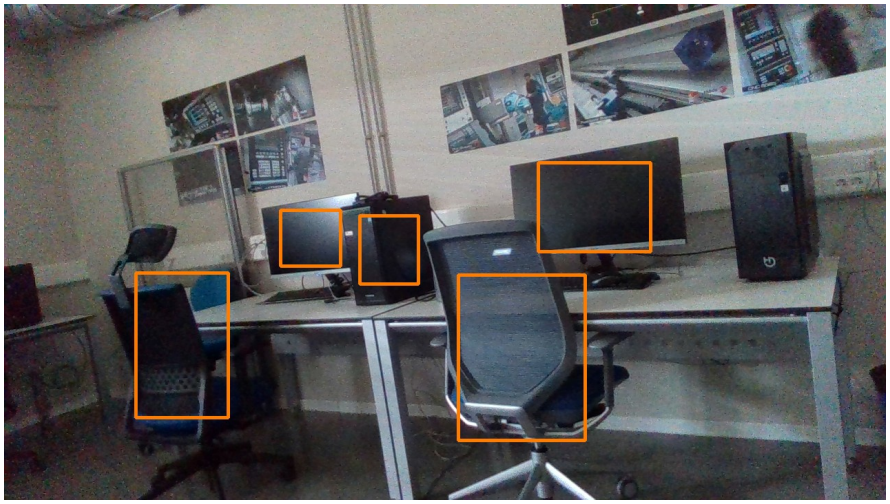

Figura 32: *Cajas de detección reducidas.*

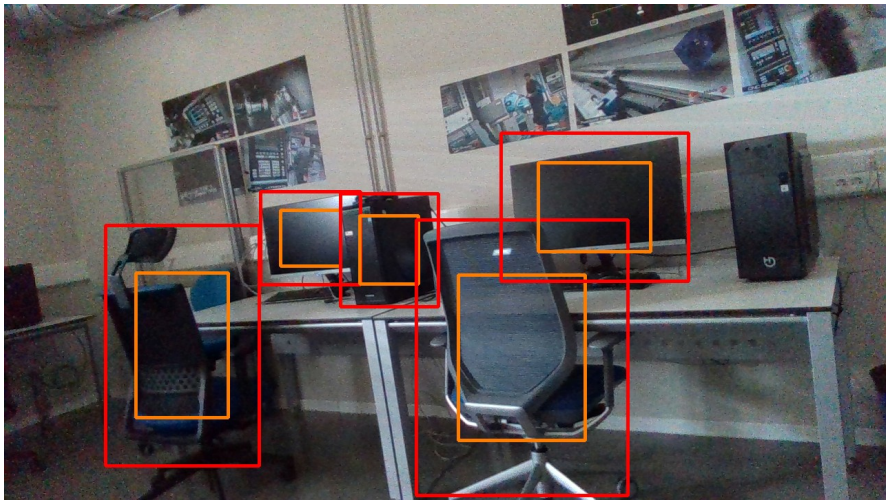

Figura 33: *Comparación del tamaño de las cajas de detección.*

Figura 34: *Máscaras de segmentación.*

Entrando más en detalle en el tercer método, que utiliza la máscara de segmentación, se explica seguidamente el proceso de obtención de la profundidad de cada objeto.

Partiendo de la Figura 35, utilizamos el modelo de *instance segmentation* y obtenemos la Figura 36, que incorpora las máscaras.

Figura 35: *Imagen de entrada 3.*

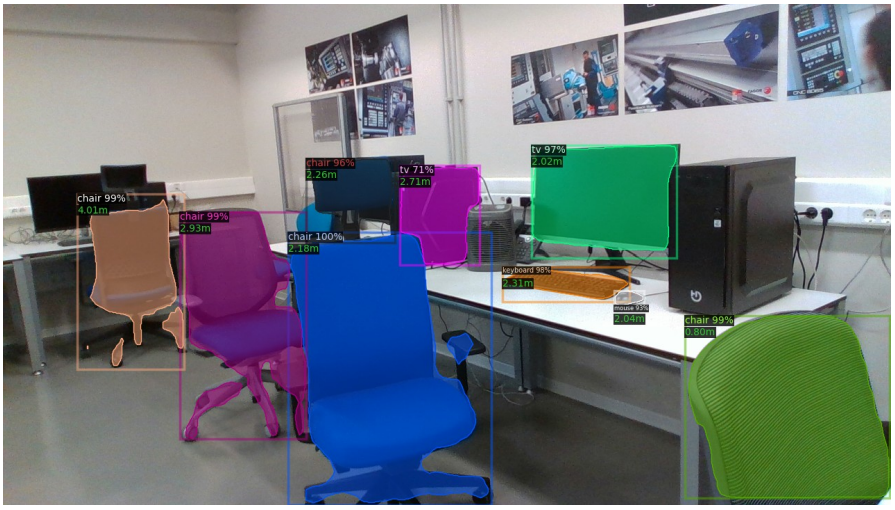

Figura 36: *Imagen con máscaras de segmentación.*

Las máscaras extraídas de la escena son las siguientes:

Figura 37: *Máscaras de segmentación de la escena.*

Para calcular la profundidad de los objetos, se obtienen todas las coordenadas de los puntos incluidos en la máscara de cada uno de ellos. Acto seguido, se lee la distancia de cada uno de esos puntos en el mapa de profundidad y se realiza la media aritmética. En las siguientes imágenes se muestra individualmente cada máscara:

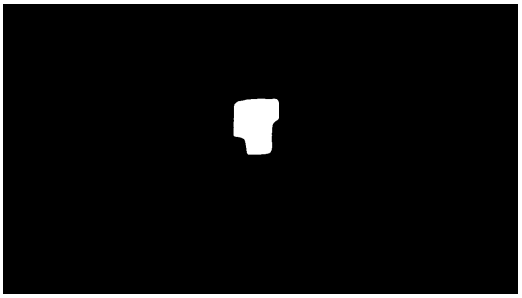

Figura 38: *CPU.*

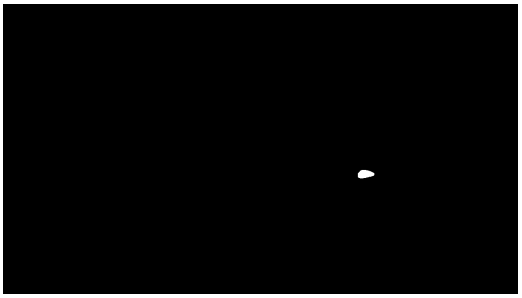

Figura 39: *Ratón.*

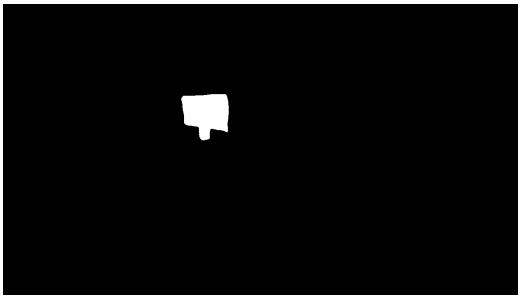

Figura 40: *Monitor 1.*

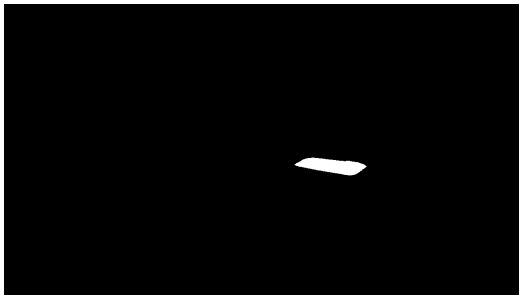

Figura 41: *Teclado.*

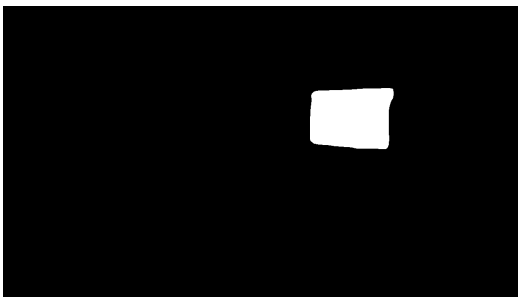

Figura 42: *Monitor 2.*

Figura 43: *Silla 1.*

Figura 44: *Silla 2.*

Figura 45: *Silla 3.*

Figura 46: *Silla 4.*

Tabla 4: *Máscaras de segmentación.*

5.1 COMPARACIÓN ENTRE MÉTODOS

Se ha llevado a cabo una batería de pruebas para comparar la velocidad de ejecución y la precisión de cada método de obtención de la profundidad.

En primer lugar, se exponen los resultados de velocidad de ejecución. Se han realizado diez pruebas sobre las mismas escenas y con la misma carga de trabajo en el ordenador para que los resultados sean equiparables.

	'Object detection' con la caja de detección por defecto	**'Object detection' con la caja de detección reducida**	**'Instance segmentation' con la caja de detección por defecto**	**'Instance segmentation' con la caja de detección reducida**	**'Instance segmentation' con la profundidad de las máscaras individuales**
Test 1	1,550	1,530	1,254	1,236	1,602
Test 2	1,518	1,527	1,247	1,248	1,609
Test 3	1,533	1,524	1,248	1,276	1,606
Test 4	1,529	1,560	1,255	1,308	1,631
Test 5	1,540	1,535	1,251	1,266	1,610
Test 6	1,566	1,541	1,248	1,246	1,672
Test 7	1,527	1,572	1,265	1,285	1,616
Test 8	1,528	1,542	1,271	1,239	1,692
Test 9	1,525	1,529	1,285	1,258	1,620
Test 10	1,563	1,563	1,254	1,280	1,629
Media	1,537	1,542	1,258	1,264	1,628

Tabla 5: *Tiempos de ejecución en segundos con distintos modelos y métodos de cálculo.*

A la vista de los resultados, la elección de una u otra caja genera variaciones irrelevantes. El tiempo de cálculo adicional de las coordenadas de la caja reducida se ve compensado por la menor cantidad de valores de los que obtener la media aritmética.

Los mejores resultados han sido obtenidos con el modelo *instance segmentation*, aunque al utilizar las máscaras de segmentación ha sido el más lento. En el modelo *object detection* no se puede realizar el cálculo de forma exacta al no delimitarse los objetos con máscaras.

Para comparar las diferencias detectadas en la profundidad se utilizan las cinco imágenes inferiores, cada una con un modelo y un método de cálculo. Los resultados más precisos son los mostrados en la Figura 47, así que se ha tomado esta imagen como referencia.

Figura 47: *Modelo 'instance segmentation' con cálculo exacto.*

Figura 48: *Modelo 'instance segmentation' con caja por defecto.*

Figura 49: *Modelo 'instance segmentation' con caja reducida.*

Figura 50: *Modelo 'object detection' con caja por defecto.*

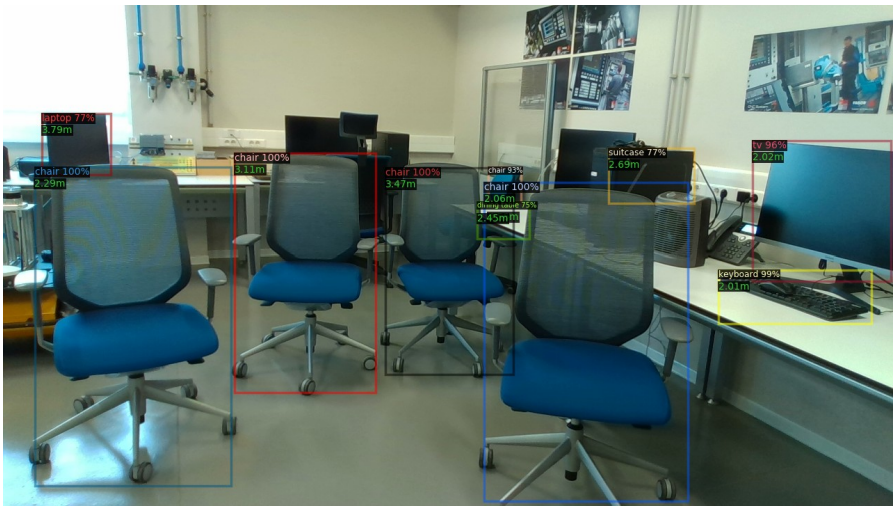

Figura 51: *Modelo 'object detection' con caja reducida.*

La comparación entre los resultados de profundidad se muestra en la siguiente tabla:

Modelo	Silla izquierda	Silla centro izquierda	Silla centro derecha	Silla derecha	Monitor	Teclado
'Instance segmentation' con la profundidad de las máscaras individuales (referencia)	2,24	3,14	3,38	2,09	1,92	2,00
'Object detection' con caja de detección por defecto	2,41	3,28	3,34	2,19	1,93	1,94
Error respecto la referencia (%)	7,59	4,46	1,18	4,78	0,52	3,00
'Object detection' con caja de detección reducida	2,29	3,11	3,47	2,06	2,02	2,01
Error respecto la referencia (%)	2,23	0,96	2,66	1,44	5,21	0,50
'Instance segmentation' con caja de detección reducida	2,49	3,28	3,39	2,17	1,85	1,98
Error respecto la referencia (%)	11,16	4,46	0,30	3,83	3,65	1,00
'Instance segmentation' con caja de detección reducida	2,29	3,08	3,39	2,09	2,00	2,01
Error respecto la referencia (%)	2,23	1,91	0,30	0,00	4,17	0,50

***Tabla 6**: Errores cometidos por las aproximaciones.*

5.2 LIMITACIONES DEL CÓDIGO

Un aspecto a tener en cuenta es que la profundidad devuelta en objetos oscuros, que absorben la luz o con reflejos es deficiente en ciertos casos.

En la escena de la Figura 52, la profundidad en las partes de la pantalla y de la pared en las que incide luz y se producen reflejos es detectada como nula (Figura 53).

Figura 52: *Primera imagen RGB con reflejo.*

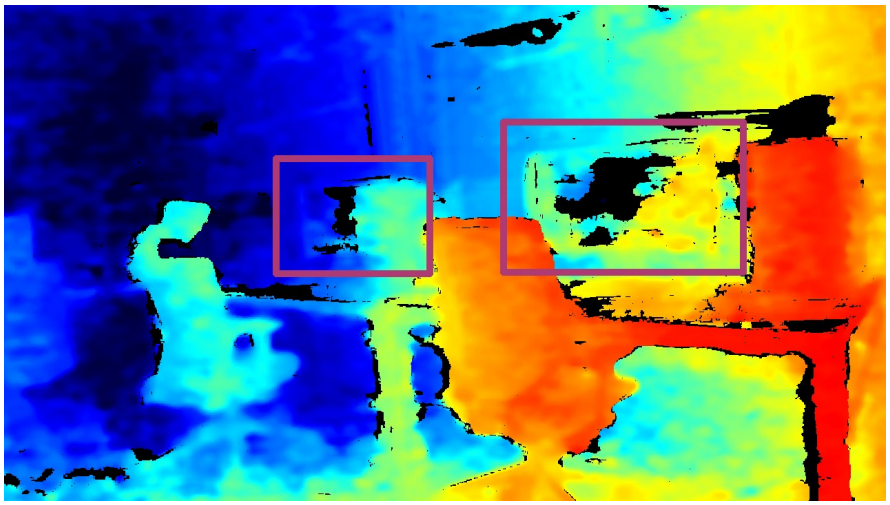

Figura 53: *Primera imagen de profundidad con reflejo.*

En este otro caso (Figura 54), la luz reflejada en los armarios hace que no se capture correctamente la distancia entre estos y la cámara, dando como resultado manchas oscuras en el mapa de profundidad (Figura 55). Además, la luz que entra por la ventana y las fluorescentes de la estancia producen errores análogos.

Figura 54: *Segunda imagen RGB con reflejo.*

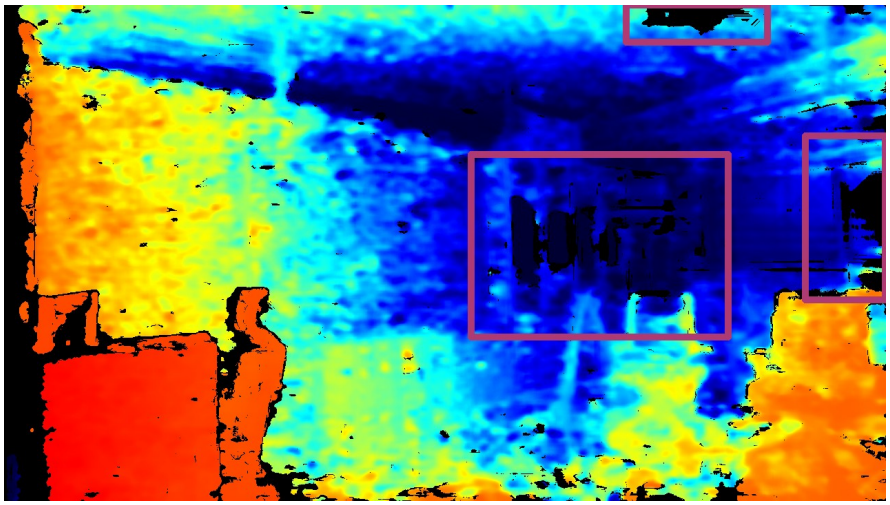

Figura 55: *Segunda imagen de profundidad con reflejo.*

Al respecto de los errores provocados por la manera en la que se realiza el cálculo de las distancias en cada método desarrollado, se ha podido observar que el área de los elementos identificados es casi siempre menor a la de su caja delimitadora. Los puntos delimitados pero fuera del elemento contribuyen a falsear la medición.

En objetos con formas cuadradas o rectangulares, la profundidad calculada es más fiel a la realidad. Si tienen otras formas, especialmente si estas son irregulares (farolas, personas, sillas…), la medición será problemática. Lo mismo ocurre en el caso de objetos captados parcialmente. En la Figura 56 se observan ejemplos de estas casuísticas.

Figura 56: *Situación que produce error al usar la caja de detección.*

Se ha medido la profundidad de los tres elementos detectados en la figura anterior con los tres métodos. Los resultados son los siguientes:

Objeto	**'Instance segmentation' con la profundidad de las máscaras individuales**	**'Instance segmentation' con la caja de detección por defecto**	**Error respecto la referencia (%)**	**'Instance segmentation' con la caja de detección reducida**	**Error respecto la referencia (%)**
Persona	1,115	2,37	112,556	1,839	64,933
Silla verde	0,766	1,574	105,483	1,524	98,956
Silla morada	3,855	3,918	1,634	3,819	0,934

Tabla 7: *Errores entre métodos.*

A la vista de estos resultados, saltan a la vista las carencias en algunos casos y las razones por las que quizás fuese conveniente utilizar métodos más precisos.

Otro inconveniente de las cajas de detección es que en ocasiones se superponen entre sí, como se aprecia en la Figura 57. Esto provoca que el vector de profundidades del elemento etiquetado como 'refrigerator' (máscara verde) contenga valores del objeto 'person' (máscara azul), ubicado a una distancia menor.

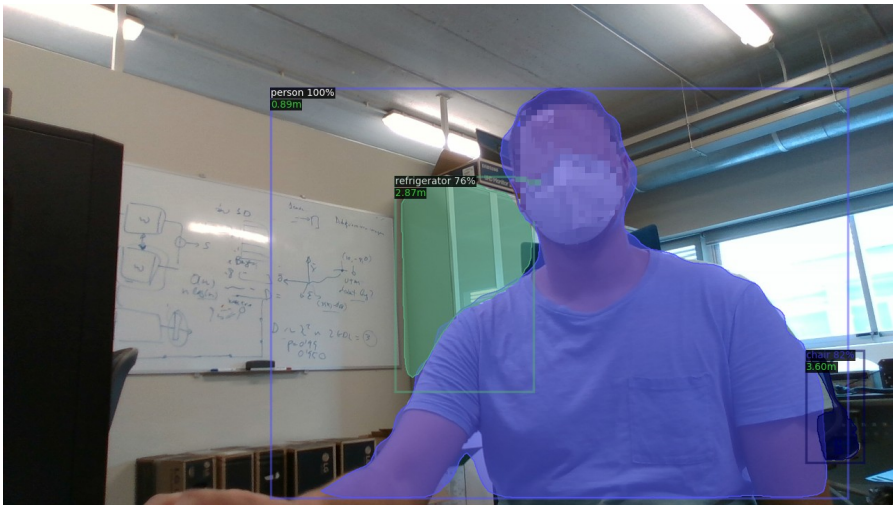

Figura 57: *Solape de cajas.*

Sin embargo, los métodos basados en caja de detección (sea reducida o por defecto) cuentan con un punto a su favor: son válidos tanto para el modelo de *instance segmentation* como para el de *object detection*, mientras que el basado en máscara solo lo es para el primero.

En la aplicación considerada en el presente documento, un error en la información de profundidad puede ocasionar colisiones y otros problemas no deseados. Por ello, sería preferible utilizar el método más preciso.

Si nos encontramos en exteriores, además de personas, el robot podría hallar objetos como los mostrados a continuación:

Figura 58: *Elemento exterior 1.*

Figura 59: *Elemento exterior 2.*

Figura 60: *Elemento exterior 3.*

Figura 61: *Elemento exterior 4.*

Por último reseñar que, durante las pruebas, los objetos negros (tales como la CPU del ordenador) en ocasiones no fueron detectados, por lo que sería necesario comprobar si otros elementos oscuros como los pivotes de la figura inferior serían también ignorados.

Figura 62: *Elemento exterior 5.*

6. RESULTADOS

Tras la explicación del funcionamiento del código, se exponen en las imágenes inferiores los resultados devueltos por el programa en determinados puntos temporales durante una ejecución en tiempo real. Se debe prestar más atención a la capacidad de clasificar objetos y a la coherencia entre las distancias que a la presencia de algún elemento mal clasificado, dado que se está usando un modelo genérico.

Se muestran cuatro secuencias extraídas de distintas ejecuciones del código.

6.1 RESULTADO 1

Secuencia de imágenes extraída durante un giro de izquierda a derecha:

Figura 63: *Resultado 1. Imagen 1.*

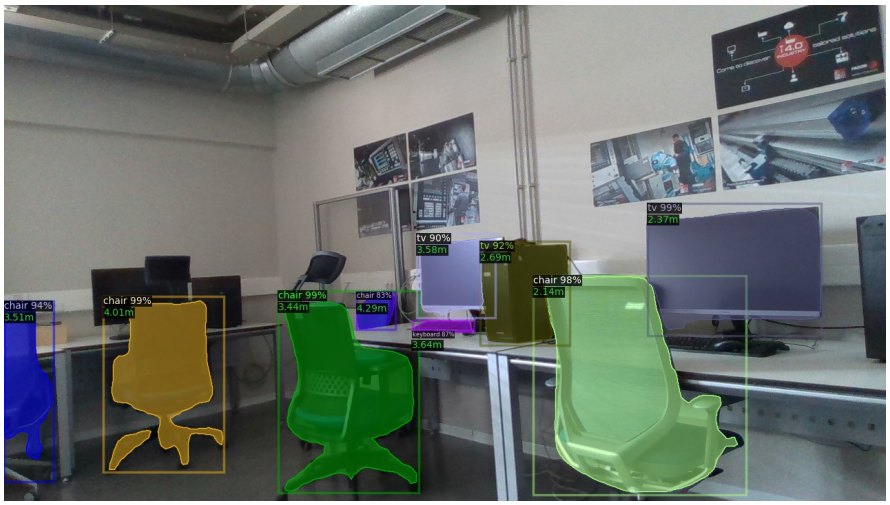

Figura 64: *Resultado 1. Imagen 2.*

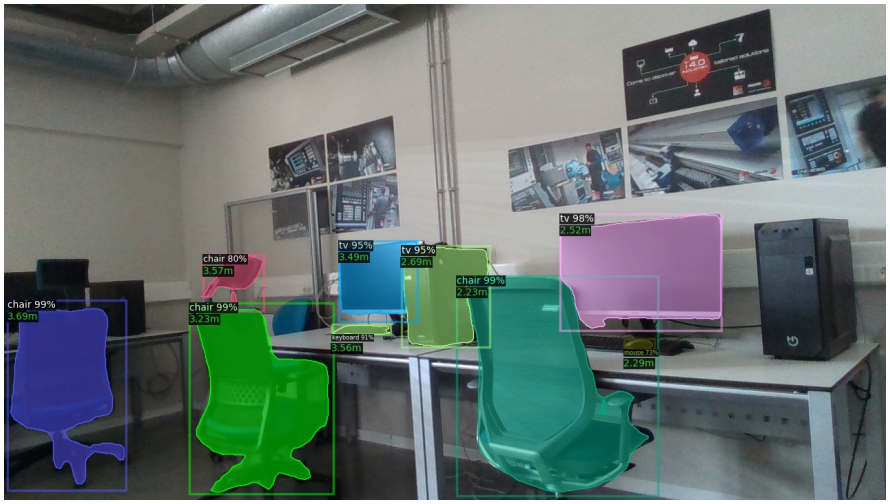

Figura 65: *Resultado 1. Imagen 3.*

Figura 66: *Resultado 1. Imagen 4.*

Figura 67: *Resultado 1. Imagen 5.*

Figura 68: *Resultado 1. Imagen 6.*

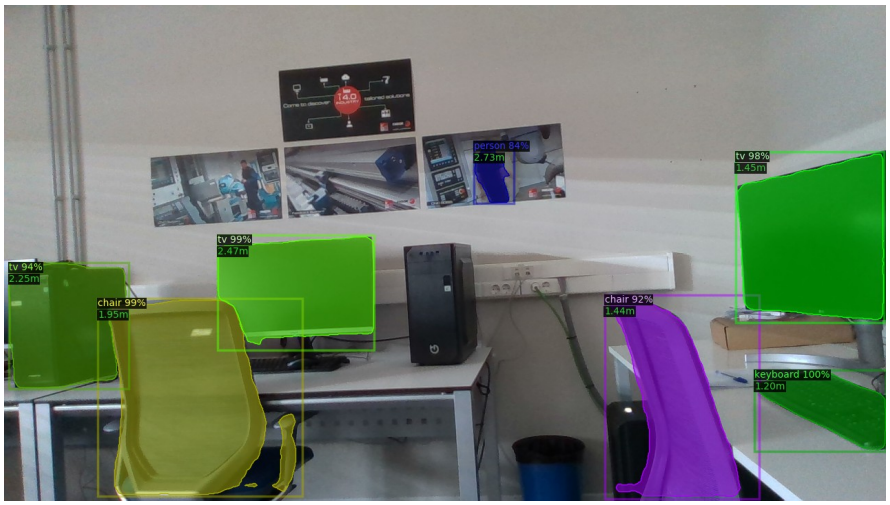

Figura 69: *Resultado 1. Imagen 7.*

Figura 70: *Resultado 1. Imagen 8.*

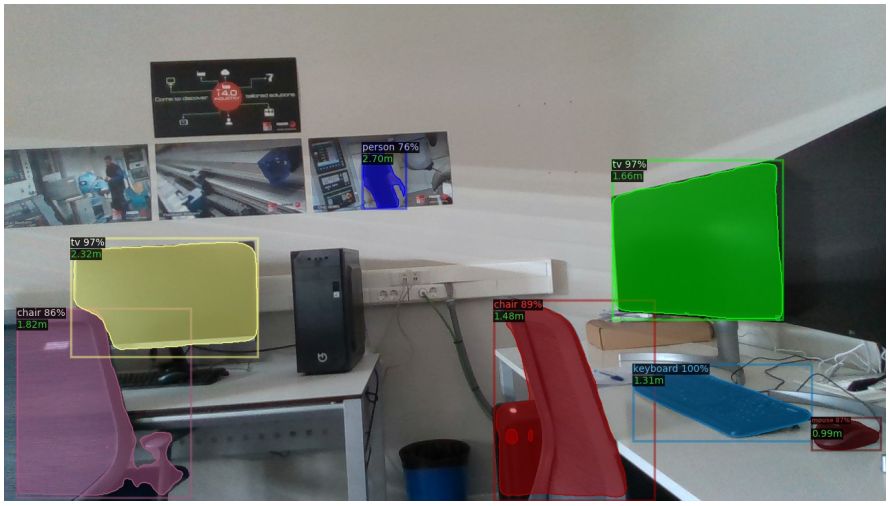

Figura 71: *Resultado 1. Imagen 9.*

Figura 72: *Resultado 1. Imagen 10.*

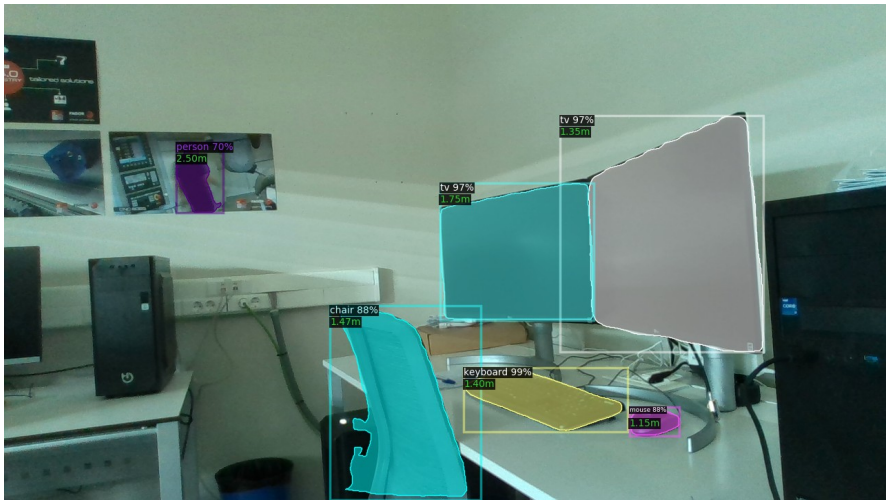

Figura 73: *Resultado 1. Imagen 11.*

6.2 RESULTADO 2

A continuación se muestra otra secuencia de imágenes, en esta ocasión de un giro de derecha a izquierda:

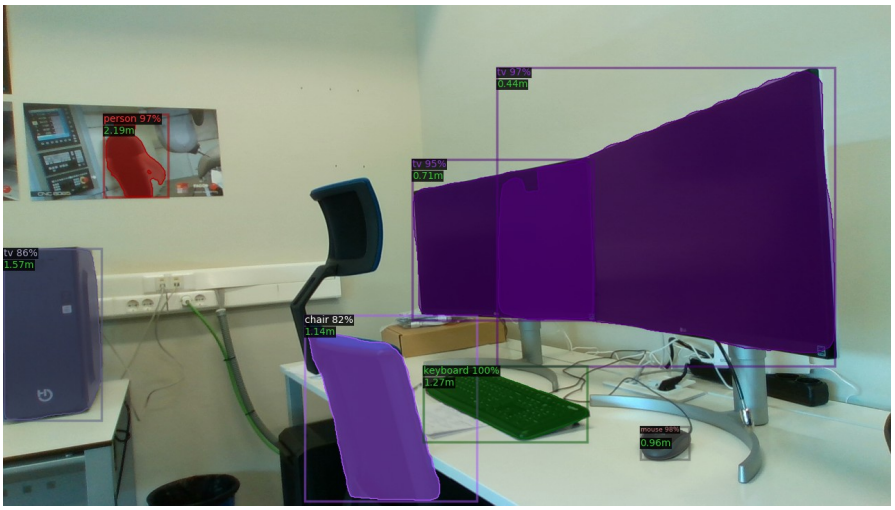

Figura 74: *Resultado 2. Imagen 1.*

Figura 75: *Resultado 2. Imagen 2.*

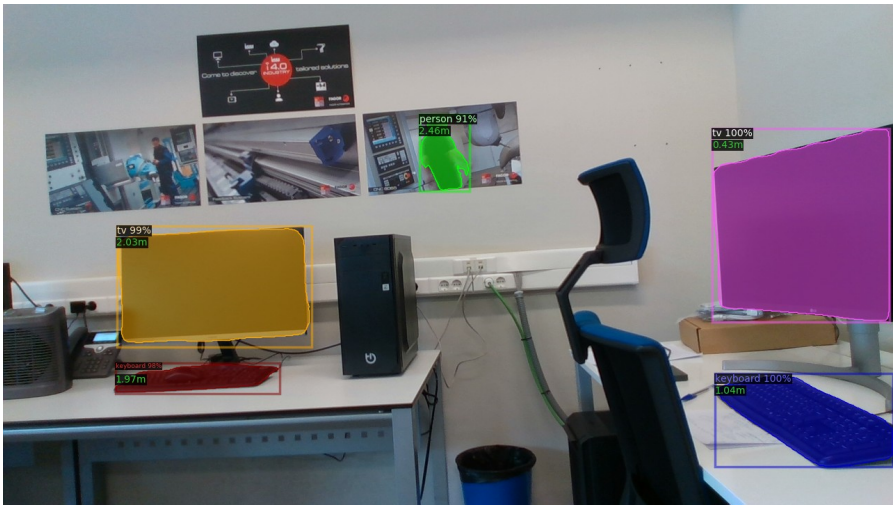

Figura 76: *Resultado 2. Imagen 3.*

Figura 77: *Resultado 2. Imagen 4.*

Figura 78: *Resultado 2. Imagen 5.*

Figura 79: *Resultado 2. Imagen 6.*

Figura 80: *Resultado 2. Imagen 7.*

Figura 81: *Resultado 2. Imagen 8.*

6.3 RESULTADO 3

Nueva secuencia de giro sobre una escena distinta:

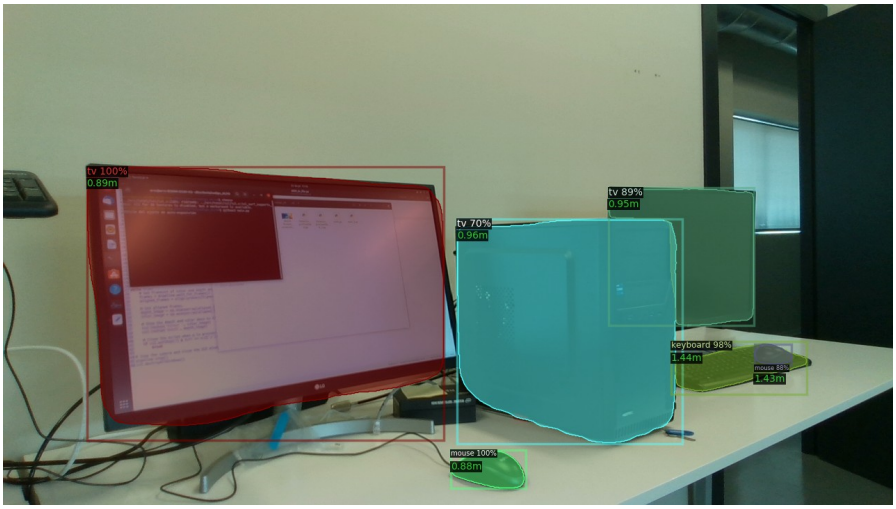

Figura 82: *Resultado 3. Imagen 1.*

Figura 83: *Resultado 3. Imagen 2.*

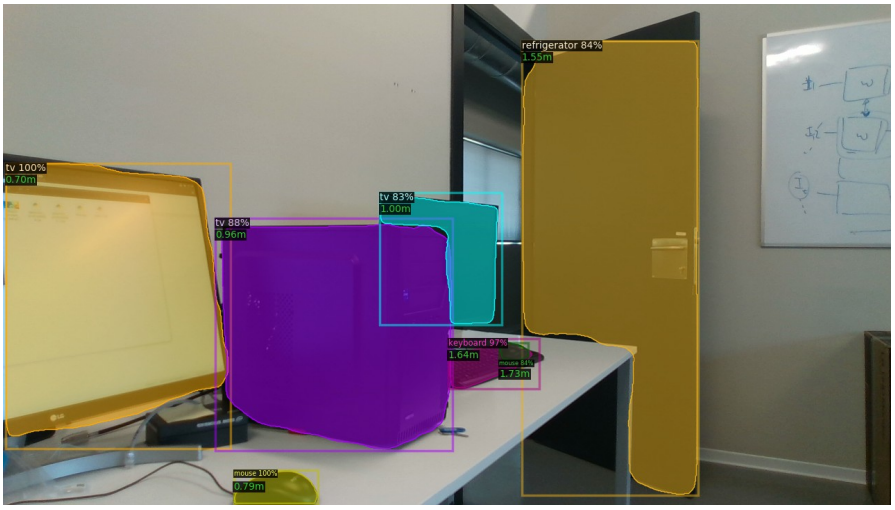

Figura 84: *Resultado 3. Imagen 3.*

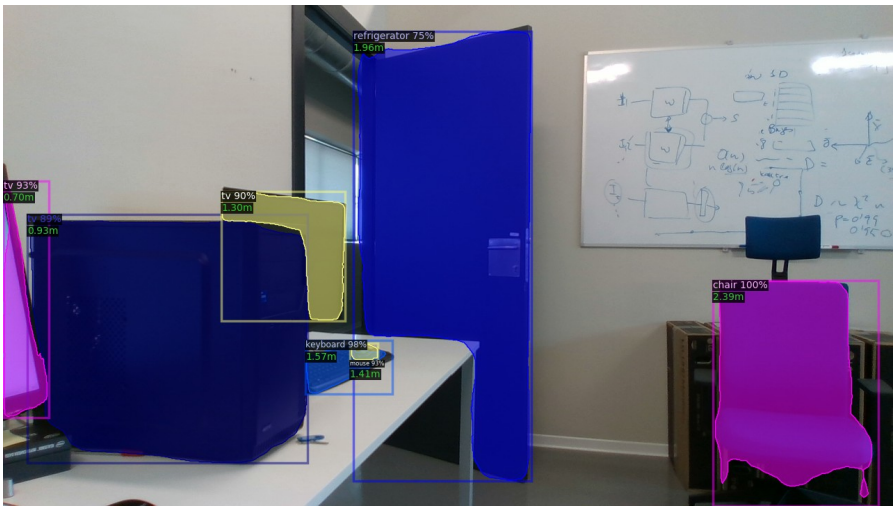

Figura 85: *Resultado 3. Imagen 4.*

Figura 86: *Resultado 3. Imagen 5.*

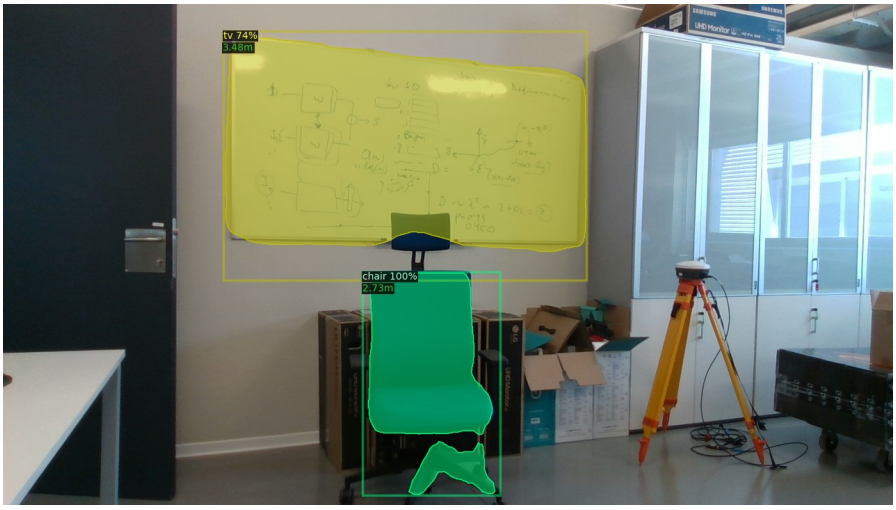

Figura 87: *Resultado 3. Imagen 6.*

Figura 88: *Resultado 3. Imagen 7.*

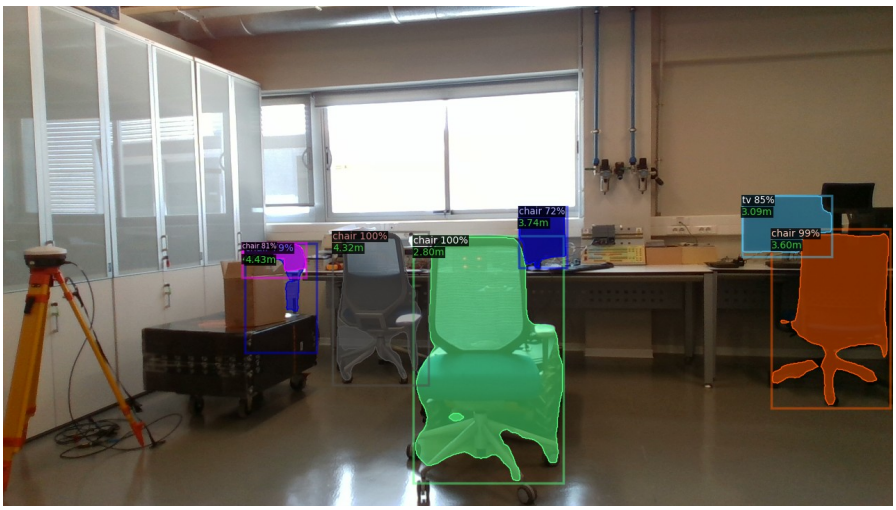

Figura 89: *Resultado 3. Imagen 8.*

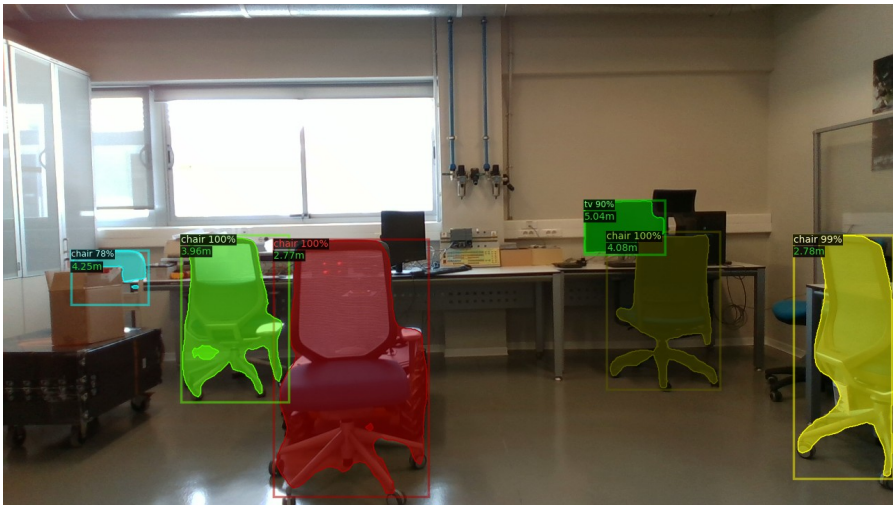

Figura 90: *Resultado 3. Imagen 9.*

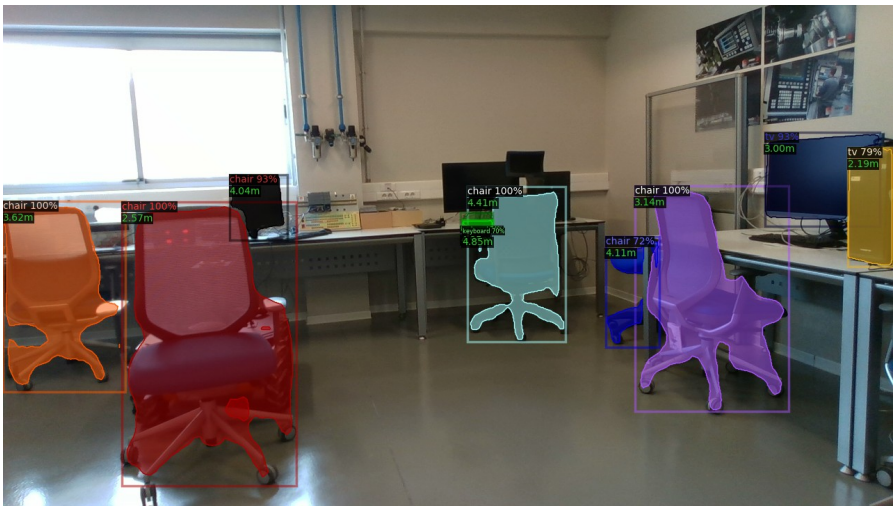

Figura 91: *Resultado 3. Imagen 10.*

Figura 92: *Resultado 3. Imagen 11.*

Figura 93: *Resultado 3. Imagen 12.*

Figura 94: *Resultado 3. Imagen 13.*

Figura 95: *Resultado 3. Imagen 14.*

6.4 RESULTADO 4

Imágenes capturadas durante el acercamiento de la cámara a un objetivo.

Figura 96: *Resultado 4. Imagen 1.*

Figura 97: *Resultado 4. Imagen 2.*

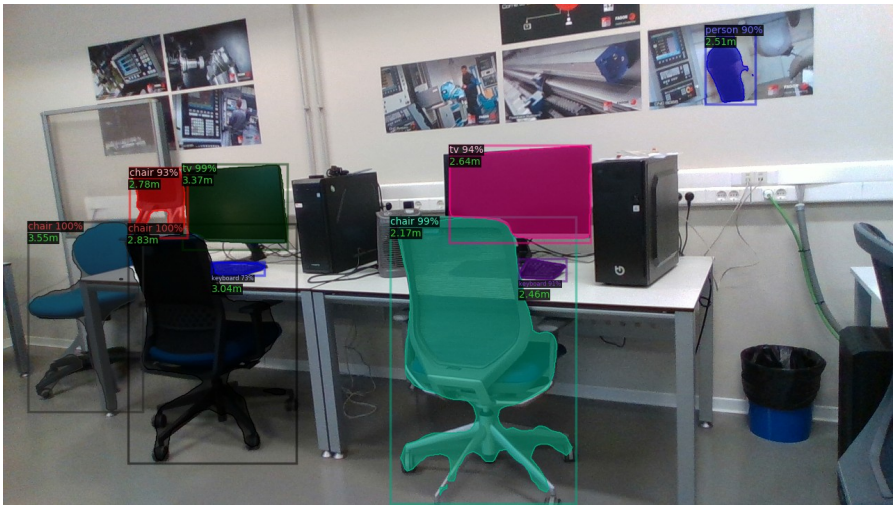

Figura 98: *Resultado 4. Imagen 3.*

Figura 99: *Resultado 4. Imagen 4.*

Figura 100: *Resultado 4. Imagen 5.*

Figura 101: *Resultado 4. Imagen 6.*

Figura 102: *Resultado 4. Imagen 7.*

Figura 103: *Resultado 4. Imagen 8.*

Figura 104: *Resultado 4. Imagen 9.*

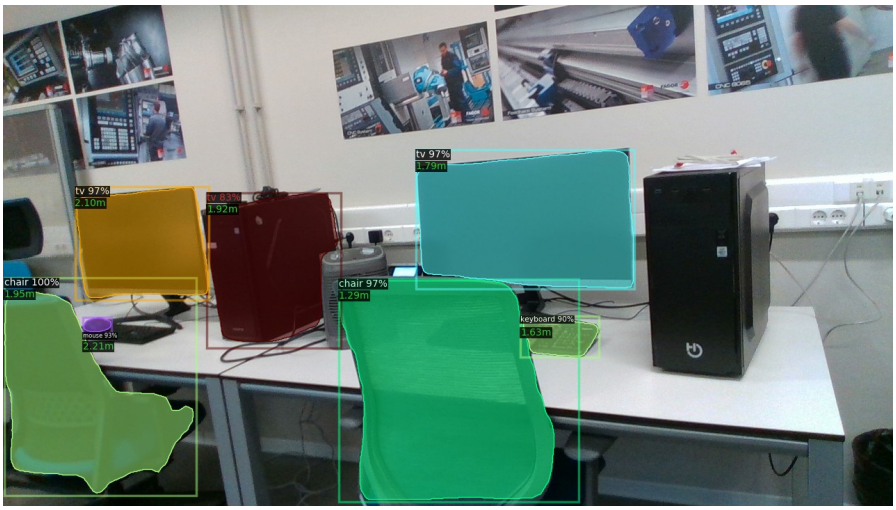

Figura 105: *Resultado 4. Imagen 10.*

Figura 106: *Resultado 4. Imagen 11.*

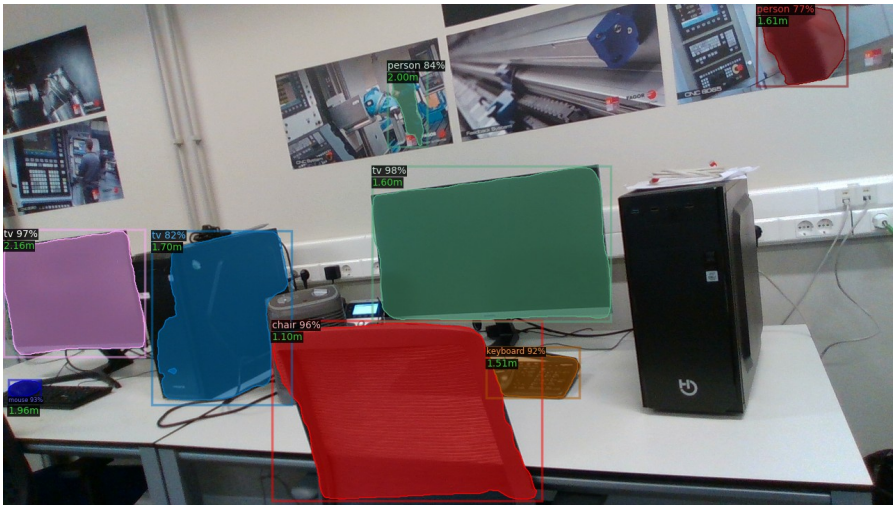

Figura 107: *Resultado 4. Imagen 12.*

7. CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se comentarán las tareas realizadas y los resultados obtenidos, así como las mejoras que se pueden llevar a cabo sobre el *software* desarrollado.

El objetivo del trabajo era la creación de un programa que expandiera la funcionalidad de la biblioteca *Detectron2* para que incluyese la profundidad de cada objeto en las sobreimpresiones de las imágenes obtenidas como resultado. A la vista de lo expuesto en el presente documento, ha quedado demostrado que esta tarea se ha cumplido.

Pese a haber conseguido el resultado esperado, se exponen a continuación algunos puntos que puede ser mejorados:

- Integración del código en ROS. La intención es que la cámara se sitúe sobre un robot móvil, probablemente un Husky (Figura 108). Sobre este ya se han montado otros sensores que funcionan a través de órdenes de ROS, por lo que sería preferible que nuestro programa desarrollado en Python fuese migrado a dicha plataforma.

En el repositorio de Github de la cámara utilizada encontramos algunos programas en ROS [10]. Entre otros, hay un paquete para lanzar la cámara, que publica *topics* (Figura 109). Se deberá acceder a los mensajes de los *topics* adecuados y utilizarlos como entrada del programa/paquete en ROS para que este proporcione como salida el mismo resultado que ofrecería el *software* en Python.

Figura 108: *Robot Husky.*
Fuente: *[11].*

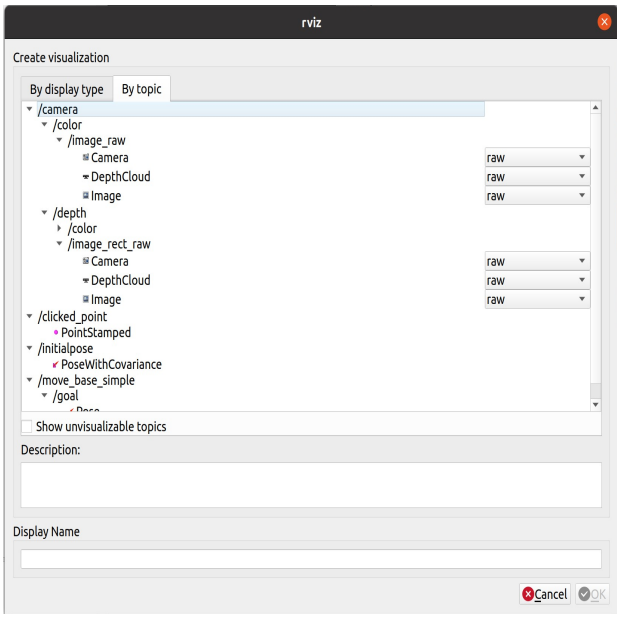

Figura 109: *ROS topics.*

- Crear o reentrenar un modelo con elementos que el robot puede encontrar durante su recorrido (bancos, farolas, plantas de la zona…) para que sea capaz de reconocerlos si no están incluidos en la lista de clases del modelo base. Los objetos que puede detectar actualmente son los que vienen en el archivo de configuración que cargamos en el *script Detector_profundidad.py*. Con él se pueden detectar ochenta clases repartidas entre elementos tales como personas, microondas, perros, bicicletas etc. En [12] se explica cómo entrenar a un modelo existente de *Detectron2* sobre un conjunto de datos personalizado.

- Optimizar el código para que pueda funcionar en tiempo real sin consumir demasiados recursos. Ya se ha expuesto en el capítulo correspondiente que el *software* tarda aproximadamente dos segundos en procesar cada imagen.

Las imágenes con los porcentajes de uso de la CPU para modo continuo y modo cada 5 y 10 segundos se muestran a continuación:

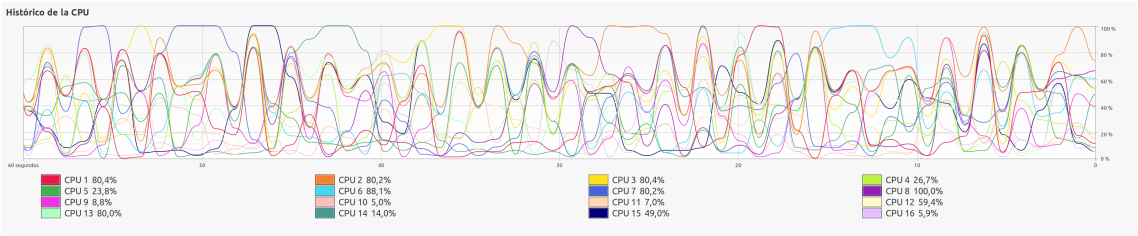

Figura 110: *Consumo de la CPU con funcionamiento continuo.*

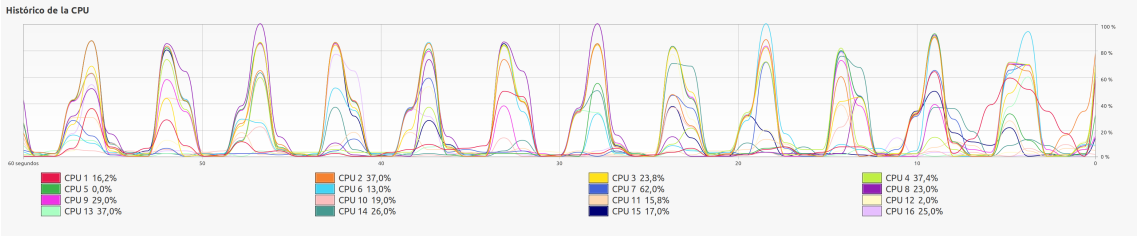

Figura 111: *Consumo de la CPU con funcionamiento cada cinco segundos.*

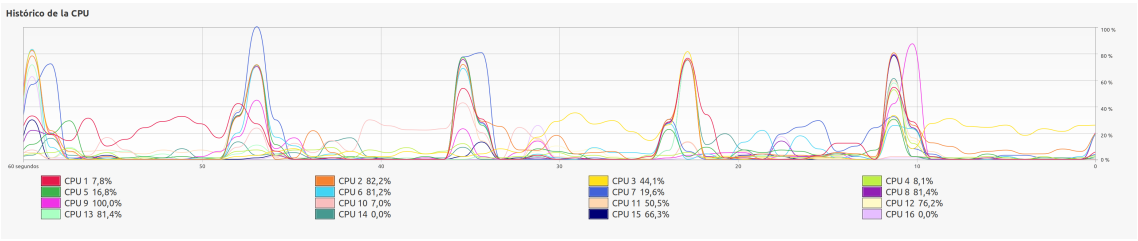

Figura 112: *Consumo de la CPU con funcionamiento cada diez segundos.*

- Tratar de mejorar el orden de dibujado de las profundidades y hacer que su color sea acorde al de la caja de detección o máscara de segmentación del objeto en cuestión. El visualizador de *Detectron2* trata de evitar que la etiqueta del objeto clasificado se solape con el propio elemento si este es muy pequeño. Para añadir la profundidad, se ha tratado de imitar este comportamiento, aunque en ocasiones se siguen produciendo este tipo de errores. Además, la profundidad de todos los objetos aparece con el mismo color (verde), haciendo que en algunos casos sea complicado determinar a cuál de las detecciones se asocia. Esto se debe a que, por coherencia, se ha utilizado el mismo código que asigna el nombre de la clase, pero por defecto se usa el color verde y no uno aleatorio.

ANEXOS

ANEXO A: REFERENCIAS BIBLIOGRÁFICAS

[1] https://www.intelrealsense.com/beginners-guide-to-depth/

[2] https://www.intelrealsense.com/use-cases/

[3] https://www.intelrealsense.com/depth-camera-d435i/

[4] https://pytorch.org/

[5] https://detectron2.readthedocs.io/en/latest/tutorials/install.html

[6] https://github.com/facebookresearch/detectron2

[7] https://youtu.be/Pb3opEFP94U

[8] https://github.com/IntelRealSense/librealsense

[9] https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/

[10] https://github.com/IntelRealSense/realsense-ros

[11] https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

[12] https://colab.research.google.com/drive/16jcaJoc6bCFAQ96jDe2HwtXj7BMD_-m5