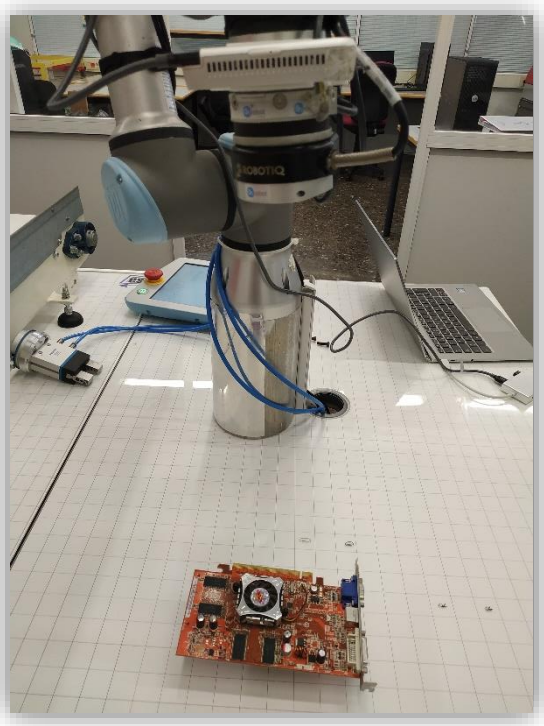
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

TRABAJO FINAL DE GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

VISIÓN TRIDIMENSIONAL EN UN BRAZO ROBOT PARA LA AUTOMATIZACIÓN DE ESCANEOS



AUTOR: SANTOS NAVARRO, EDUARDO

TUTOR: IVORRA MARTÍNEZ, EUGENIO

Curso Académico: 2019-20

RESUMEN

El escaneo 3D mediante fotogrametría es una herramienta muy utilizada en la actualidad. Esta tecnología permite la obtención de modelos 3D de objetos a los que se le ha realizado una serie de fotos. Pero esta técnica se ve muy afectada por qué tipo objeto se va a escanear. Es decir, si se escanea un objeto con pocas texturas, será mucho más complicado obtener un modelo en 3D que represente la realidad.

Este trabajo se centra en el diseño de un sistema de toma de imágenes para facilitar esta tarea de escaneado 3D mediante fotogrametría. Para ello, se conocerá de forma exacta desde qué posiciones y orientaciones se ha tomado imágenes y con ello se podrá realizar una mejor reconstrucción en 3D de los objetos con pocas texturas.

Se ha desarrollado el software necesario para la automatización de la toma de imágenes. Programando funciones para seleccionar las posiciones donde se harán las fotos, el movimiento del robot y la toma de imágenes con la cámara. Además, se ha diseñado un adaptado para acoplar la cámara a la muñeca del brazo robot UR3.

Con esto, se ha obtenido un sistema de toma de imágenes muy versátil. Ya que puede ser utilizado para otros fines, como control de calidad y detección de fallos en piezas. Pero además, cuenta con la importante ventaja de conocer con exactitud las posiciones y orientaciones desde las que se toma las imágenes, lo que aumenta en gran medida la calidad los modelos 3D obtenidos.

RESUM

L'escaneig 3D per mitjà de fotogrametria és una ferramenta molt utilitzada en l'actualitat. Esta tecnologia permet l'obtenció de models 3D d'objectes a què se li ha realitzat una sèrie de fotos. Però esta tècnica es veu molt afectada per quin tipus objecte es va a escanejar. És a dir, si s'escaneja un objecte amb poques textures, serà molt més complicat obtindre un model en 3D que represente la realitat.

Este treball se centra en el disseny d'un sistema de presa d'imatges per a facilitar esta tasca d'escanejat 3D per mitjà de fotogrametria. Per a això, es coneixerà de forma exacta des de quines posicions i orientacions s'ha pres imatges i amb això es podrà realitzar una millor reconstrucció en 3D dels objectes amb poques textures.

S'ha desenrotllat el programari necessari per a l'automatització de la presa d'imatges. Programant funcions per a seleccionar les posicions on es faran les fotos, el moviment del robot i la presa d'imatges amb la cambra. A més, s'ha dissenyat un adaptat per a acoblar la cambra a la nina del braç robot UR3.

Amb açò, s'ha obtingut un sistema de presa d'imatges molt versàtil. Ja que pot ser utilitzat per a altres fins, com a control de qualitat i detecció de fallades en peces. Però a més, compta amb la important avantatge de conéixer amb exactitud les posicions i orientacions des de les que es pren les imatges, la qual cosa augmenta en gran manera la qualitat els models 3D obtinguts.

ABSTRACT

3D scanning using photogrammetry is a widely used tool today. This technology allows obtaining 3D models of objects that have had a series of photos taken. But this technique is greatly affected by what kind of object is to be scanned. That is, if an object with few textures is scanned, it will be much more difficult to obtain a 3D model that represents reality.

This work focuses on the design of an imaging system to facilitate this task of 3D scanning using photogrammetry. For this, it will be known exactly from which positions and orientations images have been taken and so, a better 3D reconstruction of the objects with few textures can be made.

The software for the automation of image taking has been developed. Programming functions to select the positions where the photos will be taken, the movement of the robot and the taking of images with the camera. In addition, an adapter has been designed to attach the camera to the wrist of the UR3 robot arm.

With this, a very versatile imaging system has been obtained. Since it can be used for other purposes, such as quality control and fault detection in parts. But in addition, it has the important advantage of knowing exactly the positions and orientations from which the images are taken, which greatly increases the quality of the 3D models obtained.

MEMORIA

CONTENIDO

1 INTRODUCCIÓN

1.1.1 Objeto

El objeto de este proyecto es el diseño de un sistema de toma de imágenes para la obtención de modelos 3D mediante fotogrametría. Este sistema utilizará el brazo colaborativo UR3 para posicionar una cámara de profundidad en poses exactas desde las que se tomarán las imágenes.

Es muy importante conocer la posición exacta desde la que se toma cada imagen, pues es la base para una optimización de los algoritmos de reconstrucción de los programas de fotogrametría. Los softwares de fotogrametría tienen algoritmos para descubrir la posición desde donde se ha tomado cada imagen, pero estos algoritmos suelen fallar en objetos 3D con poca textura, obteniendo por tanto malas reconstrucciones 3D.

Por lo que, el objeto de este proyecto es desarrollar un sistema más robusto para el escaneo de objetos 3D, permitiendo reconstruir fielmente en 3D objetos con poca textura.

1.2 ESPECIFICACIONES

A continuación se van a explicar las especificaciones que se han de cumplir en este proyecto:

- Los puntos donde se toman las imágenes no deben ser siempre iguales. Se debe desarrollar una función con la que se pueda elegir los puntos donde se tomará imágenes adaptándolos al tamaño del objeto a fotografiar o las circunstancias del entorno.

- Se debe poder controlar el funcionamiento del sistema mediante un ordenador. Por ello, se debe implementar un sistema de comunicación socket ordenador-UR3.

- Se debe tomar imágenes después de cada movimiento del robot. Todo el sistema debe esperar mientras se hacen las fotografías y bajo ningún concepto se debe mover el brazo robot durante este proceso.

- Se debe tomar una imagen en color y una de profundidad. Estas imágenes se deben guardar en una carpeta y en sus nombres debe aparecer de qué tipo es y el número de la imagen. La numeración debe ser de cuatro dígitos, rellenándose los dígitos necesarios con ceros a la izquierda, por ejemplo: *DephImage0005.png.*

- En el diseño del adaptador de la cámara se debe realizar la sujeción principal de la cámara mediante un tornillo diámetro 1/4 de pulgada y rosca tipo Whithworth.

1.3 MARCO TEÓRICO

En este apartado se va a realizar una breve explicación de los conceptos teóricos que se han utilizado en el desarrollo de este proyecto.

1.3.1 Comunicación Socket

La comunicación por Sockets es un tipo de comunicación entre un cliente y un servidor. Para establecerse estas conexiones utiliza el protocolo de comunicación TCP/IP.

Para poder realizar una conexión es necesario indicar qué dispositivo hará las veces de servidor y cuál de cliente. Después de esto, se debe indicar en el cliente a qué dirección IP se desea realizar la conexión y por qué puerto se realizará. Mientras tanto, al servidor se le puede indicar la IP del dispositivo con el que aceptará la conexión, además del puerto por el que se realizará la conexión. Esto se puede observar de forma simplificada en el siguiente esquema (Imagen 1):

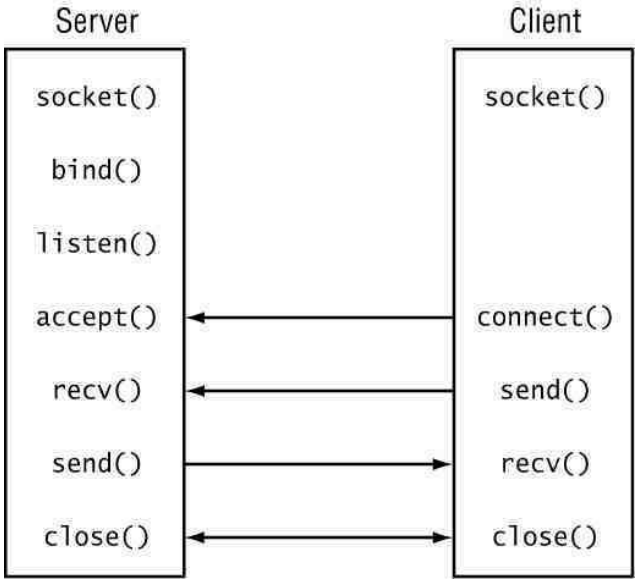


Imagen 1: Esquema de comunicaciones socket. (Imagen obtenida de: Referencia 1)

1.3.2 Fotogrametría

La fotogrametría es una técnica que utiliza las posiciones y mediciones de varias imágenes para obtener información sobre la forma y dimensiones de un objeto con el fin de crear modelos informáticos fieles a la realidad.

Esta técnica es comúnmente utilizada en la topografía, pero actualmente se está utilizando en mayor medida para el escaneado 3D de diferentes objetos. Cabe destacar que debido a los cambios de magnitud en la dimensiones entre la topografía y el escaneado 3D de piezas se puede encontrar cambios en la técnica, pero la base sobre la que se sustenta es la misma.

Para hacer un correcto uso de la fotogrametría un paso fundamental es la adquisición de las imágenes. Para realizar un escaneado 3D obteniendo una gran calidad en el resultado es necesario conocer la posición y orientación en la que fue tomada cada imagen. Además de que a mayor cantidad de imágenes mayor resolución tendrá el resultado final.

A continuación se muestra un ejemplo, en él se observa la reconstrucción 3D de un busto de Valle-Inclán (Imagen 2). Cada rectángulo azul indica una posición desde la que se tomó una imagen.

Imagen 2: Ejemplo de modelo obtenido mediante fotogrametría. (Imagen obtenida de Referenci 2)

Cabe destacar que la mayoría de programas de fotogrametría disponen de un algoritmo para obtener la posición de la cámara a partir de las imágenes. Este depende de las imágenes tomadas el algoritmo puede realizar una composición del modelo 3D más o menos fiable. Para evitar esta variabilidad lo óptimo es introducir las posiciones reales en las que había sido tomada cada imagen. Es por ello que en este proyecto se desarrolle un sistema de toma de imágenes muy exacto.

1.3.3 Geometría y cinemática en robótica

En este apartado se va a explicar los conceptos teóricos necesarios para facilitar el entendimiento de la solución planteada. En primer lugar se va a explicar un concepto que se repetirá a lo largo de la memoria, que es el concepto de pose. Y en segundo lugar, se hará una breve explicación de la cinemática inversa que se utiliza en la solución y los problemas que puede presentar.

Pose

Cuando en esta memoria se mencione durante la memoria la palabra pose se estará hablando de una posición y orientación. Es decir, los datos necesarios para ubicar un objeto en un espacio tridimensional, respecto de un sistema de coordenadas inicial. Esto es de gran importancia, dado que será necesario indicar al efector final del brazo robot la pose exacta donde se deberá mover.

Hay varias formar de representar las poses. Pero durante este proyecto se representará de dos formas:

- **Como una matriz de transformación:** esta es una matriz 4x4 formada por cuatro submatrices. La primera de ellas es la matriz de rotación 3x3, que representa la rotación del sistema de referencia final respecto al inicial. Después, se encuentra la matriz de traslación, que es 3x1 y representa un desplazamiento respecto al sistema de coordenadas inicial. Por último, se encuentra la matriz de perspectiva y la de escalado, pero estas permanecen invariantes en la robótica, ya que no hay cambios de perspectiva y se trabaja siempre a escala 1.
  A continuación, se muestra un ejemplo de una matriz de transformación completa y de la matriz de transformación utilizada en robótica (Imagen 3 y 4):

  Matriz de transformación general:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{p}_{3\times1} \\ \mathbf{f}_{1\times3} & \mathbf{w}_{1\times1} \end{bmatrix} = \begin{bmatrix} \text{Rotación} & \text{Traslación} \\ \text{Perspectiva} & \text{Escalado} \end{bmatrix}$$

Imagen 3: Esquema de la matriz de transformación expresada de forma general.

  Matriz de transformación en robótica:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{p}_{3\times1} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \text{Rotación} & \text{Traslación} \\ 0 & 1 \end{bmatrix}$$

Imagen 4: Esquema de la matriz de transformación utilizada en robótica.

- **Como tipo de variable en URscript:** esta es la forma de representar la pose en la programación de robots de Univeral Robots. Es un tipo de variable, en el que los tres primeros valores son los de la posición respecto al origen de coordenadas. Y los tres valores siguientes indican la rotación en X, Y, Z respectivamente. El formato de las rotaciones que se debe de introducir para que el programa funcione correctamente es de *axis-angle*. Para realizar conversiones entre una representación de la pose y otra existe una función de Matlab para transformar de matriz de rotación a *axis-angle* y otra función para la conversión en la otra dirección. Estas funciones son *rotm2axang y axang2rotm.*
  Se puede encontrar más información sobre el tipo de variable y cómo utilizarla en el manual de URscript (Referencia 2).

Cinemática inversa en robótica

La cinemática inversa en robótica consiste en el cálculo de los ángulos que debería tener cada articulación del brazo robot para alcanzar una pose objetivo. Este cálculo tiene más de una solución posible, además de poder tener singularidades.

Los brazos robot de Universal Robots ya tienen programado el cálculo de la cinemática inversa del robot, junto con funciones para elegir la solución de la cinemática inversa deseada.

Cabe destacar que en distintas soluciones de la cinemática inversa puede haber un gran cambio en cómo está colocado el robot, pero nunca hará diferencia en la pose del efector final. Esto se puede observar en el siguiente ejemplo (Imagen 5):

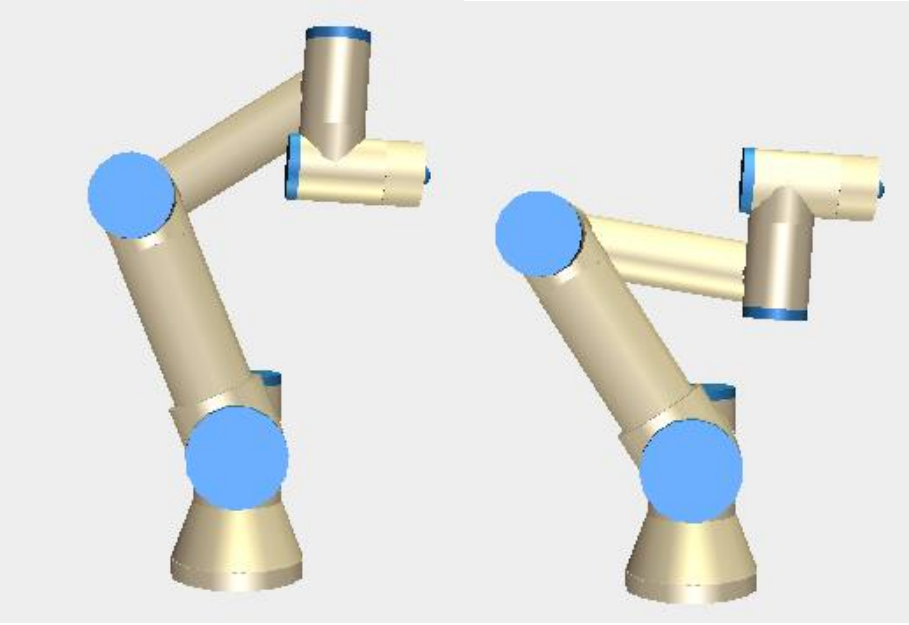


Imagen 5: Ejemplo del resultado de tomar dos soluciones de la cinemática inversa distintas.

1.3.4 Programación en URscript

En este apartado se va a explicar las bases de la programación de brazos robot de Universal Robots. Esta programación se puede realizar de dos formas.

La primera de ellas es mandando cada comando que se quiera realizar desde un equipo externo. Esto quiere decir que se controlará el robot completamente desde el equipo externo y cada vez que se quiera que actúe el brazo robot se mandará le mandará el comando al robot y a su vez el robot puede mandar datos de fin de tarea al ordenador. En la Imagen 6 se muestra un esquema de como fluye la comunicación en esta solución de la programación:

Imagen 6: Esquema de programación UR desde ordenador.

Por otro lado, en el otro método de programación posible sí que se desarrolla un script para ser leído en el brazo robot. De esta forma, el brazo robot no tiene dependencia de un equipo externo, pero aun así se puede establecer comunicaciones para la trasmisión de datos entre el robot y el equipo externo. Normalmente esta programación se realiza desde el *flexpendant* del brazo robot, que es la pantalla de control que tienen los brazos robots. En este caso la programación del brazo robot seguiría el siguiente esquema (Imagen 7):

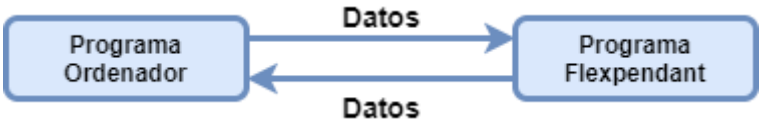


Imagen 7: Esquema de prgramación dividida

Para ambos métodos de programación se deben utilizar los comandos de URscript. Todos estos comandos están recogidos en el manual de URscript indicado en el apartado de *Referencias 2.*

Cabe destacar que en este proyecto se ha utilizado el segundo método de programación utilizando el programa URsim para simular la programación y comportamiento del robot.

2 ESCENARIO

En este proyecto se ha dispuesto de dos escenarios, en primer lugar el laboratorio donde se ha implementado y comprobado el correcto funcionamiento del proyecto (Imagen 8). En segundo lugar se encuentra el entorno de simulación, donde se ha desarrollado y hecho las comprobaciones del proyecto (Imagen 9).

Escenario del laboratorio

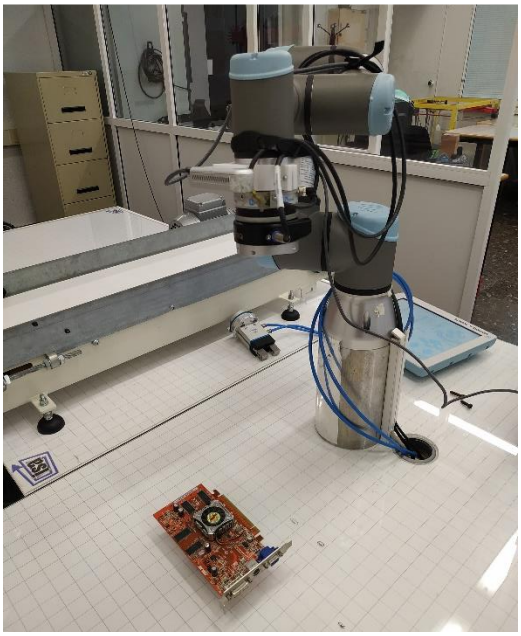


Imagen8: escenario utilizado en las pruebas en el laboratorio. En él se encuentra el brazo robot UR3 con la cámara D415 acoplada a su muñeca. Se está tomando imágenes a una placa GPU.

Escenario del simulador

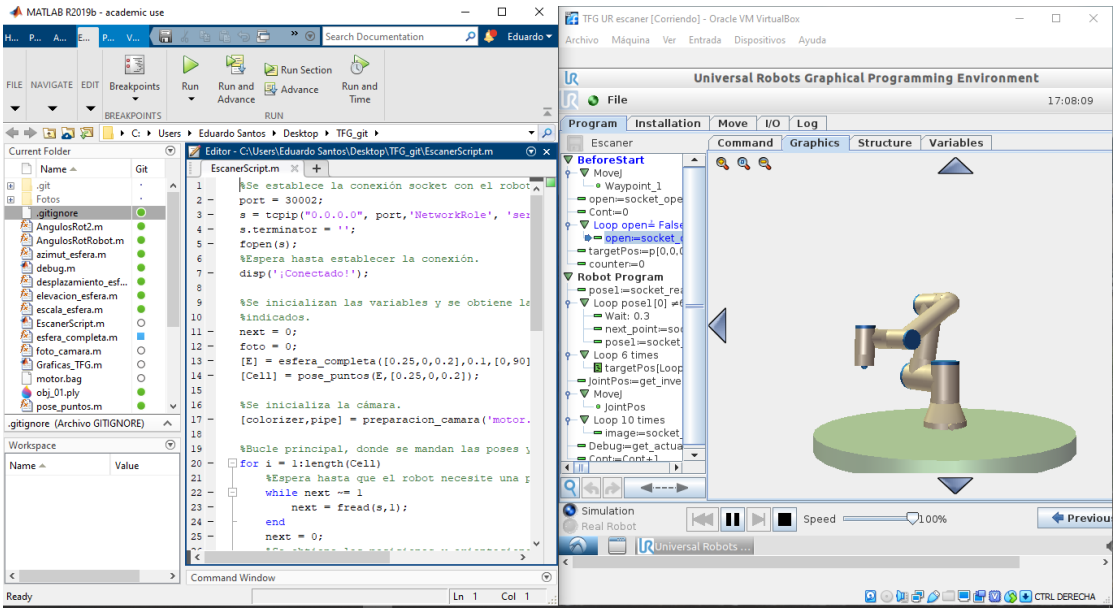


Imagen 9: Escenario utilizador para las comprobaciones en simulador. A la izquierda se encuentra el script de Matlab. A la derecha se encuentra el simulado URSim, donde se ha programado el robot y se observa su comportamiento

2.1 ROBOT UR3

Para este proyecto se ha utilizado el brazo robot colaborativo UR3 de Universal Robots. Se ha elegido un brazo robot colaborativo para reducir al mínimo los daños que se pudieran hacer durante el desarrollo del proyecto, tanto a los instrumentos como a humanos. Aunque cabe destacar, que finalmente se ha desarrollado la totalidad del proyecto vía simulador y antes de probar cualquier programa en el robot real se ha comprobado el correcto funcionamiento en el simulador.

Por otro lado, este brazo robot tiene una longitud de brazo de $50\ cm$, esto puede llegar a ser un problema dependiendo del objeto que se quiera escanear y de la cantidad de fotos que se requiera. Por lo que se va a tener que desarrollar estrategias para evitar los problemas por falta de longitud de brazo. Estas estrategias se explicarán en el apartado 5 Problemas encontrados y Soluciones Aportadas.

En el apartado de Referencias 3 se puede encontrar el manual de usuario del robot UR3 con información adicional útil para el uso de estos brazos robot.

2.2 SIMULADOR UR3

El desarrollo del proyecto se ha realizado utilizando un simulador del robot UR3. Este simulador es el URSim y funciona en un entorno de Linux, por lo que para poder utilizarlo desde un equipo con Windows se debe utilizar una máquina virtual.

El simulador permite realizar la programación del robot como si se estuviese utilizando el *flexpendant* además de simular por pantalla los movimientos que realizaría el robot.

2.3 CÁMARA

Para la toma de imágenes se va a utilizar la cámara de profundidad D415 de Real Sense (Imagen 10). Se va a utilizar una cámara de profundidad debido a las ventajas que aporta a la hora de realizar la recontrucción 3D de los objetos fotografiados. Por ejemplo, gracias a utilizar una cámara de profundidad no es tan crítica la iluminación del sistema y tampoco se capturará las sombras.

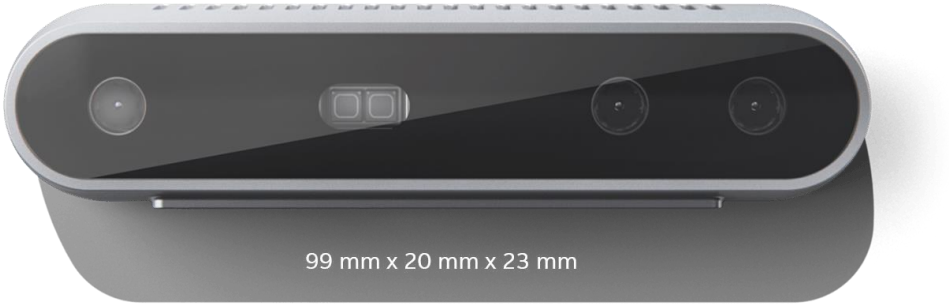


Imagen 10: cámara de profundidad D415 de RealSense. Utilizada en el proyecto.

La conexión de esta cámara con el ordenador se hace mediante un cable USB 3.0 tipo C. Lo que facilita su conexión con el ordenador, pero no permite realizar conexiones a distancias mayores de cinco metros.

Se ha escogido esta cámara en particular debido a sus pequeñas dimensiones y reducido peso, lo que la hace ideal para incorporarla en el brazo robot UR3. Otro atributo importante de esta cámara es la tecnología de estéreo activo que tiene para calcular la profundidad.

Es muy importante la especificación del peso de esta cámara, que es de $72\ g$. Al que hay que añadirle el peso del adaptador de la cámara al brazo robot, que es de $30\ g$. El conjunto de cámara+adaptador pesa $102\ g$, que al ser tan reducido no tiene un impacto significativo a las prestaciones del robot. Esto abre la posibilidad de equipar herramientas en el efector final del robot siempre y cuando no tapen el field of view de la cámara.

Las especificaciones que se han mencionado son las principales por las que se ha elegido esta cámara. Aunque en el apartado de Referencias 4, se puede encontrar el datasheet de esta cámara, donde se puede profundizar en la información de este apartado.

2.4 ADAPTADOR DE LA CÁMARA Y SOPORTE DE OBJETOS

En el entorno también se encontrará el adaptador de la cámara al brazo robot, el cual se ha diseñado para este proyecto y se ha impreso en 3D para la implementación final del proyecto.

Por otro lado, también será necesario utilizar un soporte para elevar el objeto que va a ser escaneado. Es necesario conocer de forma exacta el tamaño y la posición del soporte para asegurar el correcto escaneo de la pieza.

3 DESARROLLO DE LA SOLUCIÓN

Debido a lo complejo que es el problema propuesto a resolver se ha dividido el desarrollo de la solución en distintos apartados individuales. Con ello, se ha podido trabajar en paralelo en cada apartado y comprobar su correcto funcionamiento por separado.

La solución adoptada consiste en desarrollar un sistema de toma de fotos con el brazo robot UR3 para hacer un escaneado de piezas 3D mediante fotogrametría. El desarrollo del proyecto se realizó mediante simulaciones, que posteriormente han sido implementadas en el entorno físico. Para conseguir el desarrollo en el simulador se han seguido los siguientes pasos:

- **Obtener la pose de los puntos objetivo:** se ha desarrollado un Script en Matlab con el que se obtiene las poses de los distintos lugares donde se debe hacer una foto. Las poses obtenidas están en forma de matriz de transformación, lo que nos da información de la posición y orientación de cada punto.
- **Programar el movimiento del robot UR3:** en este paso se desarrollado programas para comprobar el movimiento del robot. El primero de ellos mueve el efector final del robot a una pose determinada, mientras que el segundo lee de un fichero distintas poses y alcanza esas posiciones.
- **Programación de la comunicación por Sockets:** en este paso se desarrolló el sistema de comunicación por sockets entre Matlab y el simulador del UR3. En primer lugar se comprobó que era posible esta comunicación mediante el programa *SocketTest*, con el que se consiguió actuar sobre el simulador. Después se desarrolló los programas en URScript y Matlab que permiten la comunicación vía socket.
- **Implementación de la toma de imágenes:** para la implementación de la toma de imágenes, se ha utilizado un wrapper de Matlab para la utilización de cámaras de *IntelRealSense* (Referencia 5). En este wrapper se encuentran las funciones necesarias para la toma de imágenes en color y profundidad, necesarias para el escaneo.
- **Coordinación de todos los apartados:** en este paso se ha juntado todos los apartados obteniendo un script general que cumple todo el funcionamiento descrito en el planteamiento. Se obtendrán las poses y se mandarán una a una vía socket al UR3. Cuando el robot alcance una posición, este mandará una señal y se tomará una foto en color y otra en profundidad y en acabar, se volverá a mandar una nueva pose.
- **Calibración de la cámara:** este es un paso fundamental para una correcta toma de imágenes en la implementación con el robot real. Aquí se explica por qué debe ser calibrada la cámara y que técnica se ha escogido para la calibración.
- **Diseño del adaptador de la cámara:** este paso se ha ido desarrollando en paralelo con el resto de pasos. Consiste en el diseño en SolidWorks de un adaptador de la cámara D415 al brazo robot UR3.

3.1 SCRIPT EN MATLAB PARA OBTENER LAS POSES DE LOS PUNTOS OBJETIVO

Este es el primer apartado de programación que se comenzó a desarrollar en el proyecto. Consiste en desarrollar un script en Matlab en el que se obtenga la pose de los puntos necesarios para realizar el proceso de escaneado. Estas poses corresponderán con los puntos de una superficie esférica y estarán orientados hacia un punto indicado.

A la hora de realizar la toma de imágenes el diseño debe ser robusto para distintas circunstancias, como seleccionar en qué áreas de puntos se tomarán las imágenes y adaptase a distintos tamaños de objeto. A continuación, se muestra una imagen ejemplo donde el tamaño de la esfera es de $0.1\ m$ y se ha tomado los puntos correspondientes al intervalo de azimut de $-90º$ a $180\ º$ (Imagen 11):

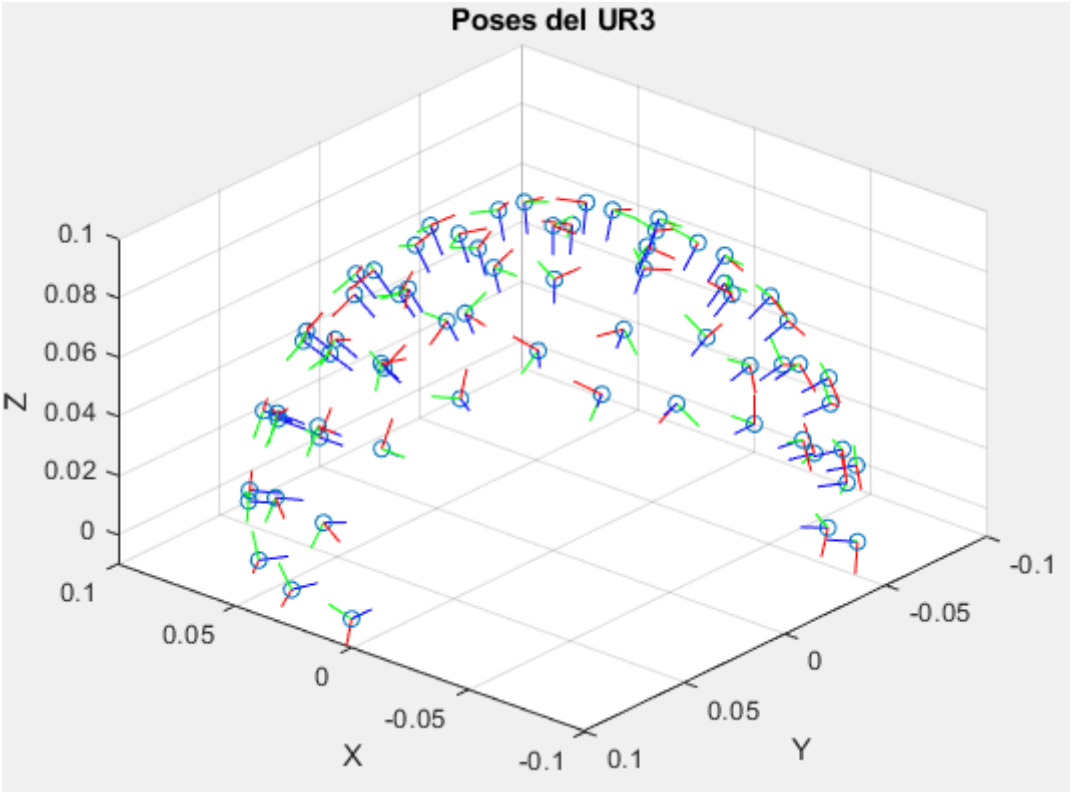


Imagen 11: Ejemplo de poses donde el robot hará una foto. En azul se muestra el eje Z, en rojo el X y en verde el Y.

Para obtener este resultado se ha desarrollado una serie de funciones que modifican una matriz de los puntos correspondientes a una esfera de radio $1\ m$. Después, se ha desarrollado otra función que calcula la pose en cada uno de los puntos para que el eje Z mire hacia el centro, como se puede observar en la Imagen 11, donde el eje Z es el de color azul.

Para el cálculo de las matrices de orientación en las poses se ha resuelto un sistema de ecuaciones formado por la última fila de una matriz de rotación genérica, formada por tres rotaciones, que son en Y, Z y X (Imagen 12). De aquí saldrán tres incógnitas, que son los ángulos de las rotaciones. Y como término independiente se encuentra el vector unitario que apunta de la posición de la que se calcula la orientación, al centro. Obteniendo el siguiente sistema de ecuaciones (Ecuación 1):

$$\begin{bmatrix} C\phi C\theta & S\phi S\alpha - C\phi S\theta C\alpha & C\phi S\theta S\alpha + S\phi C\alpha \\ S\theta & C\theta C\alpha & -C\theta S\alpha \\ -S\phi C\theta & S\phi S\theta C\alpha + C\phi S\alpha & C\phi C\alpha - S\phi S\theta S\alpha \end{bmatrix}$$

Imagen 12: Composición de rotaciones en Y, Z y X.

```
(-sin(y))*cos(z)                          = Xvec;

(sin(y)*sin(z)*cos(x))+(cos(y)*sin(x))    = Yvec;

(cos(y)*cos(x))-(sin(y)*sin(z)*sin(x))    = Zvec;
```

Ecuación 1: Sistema de ecuaciones para la obtención de la rotación en Y, Z y X.

Se utiliza este sistema ya que la última fila de las matrices de transfomación corresponde con las componentes del nuevo eje Z tras la rotación. Y como este valor es conocido, ya que se quiere que apunte hacia el centro, se puede obtener los valores de estas componentes a partir de la posición del punto y del centro. Todo esto se explica en detalle en la función *pose_puntos.*

3.1.1 Funciones utilizadas en el script

Estas son las funciones utilizadas para obtener el resultado mostrado en la Imagen 11:

escala_esfera

Esta función se encarga de aplicarle una escala a los puntos de la superficie de la esfera de radio $1\ m$, obteniendo a la salida una esfera de puntos de radio igual al factor de escala. Esta función es necesaria para adaptar las posiciones utilizadas a los distintos tamaños de objeto escaneados. A continuación se muestra un ejemplo del uso de esta función, en la que se aplica una escala de 2 a la esfera original.

```
%Se toma el valor de los puntos de la superficie de la esfera.
EsferaRadio_1 = csvread('sphere2.csv');

%Se aplica la escala.
EsferaRadio_Escala = escala_esfera(2,EsferaRadio_1);
```

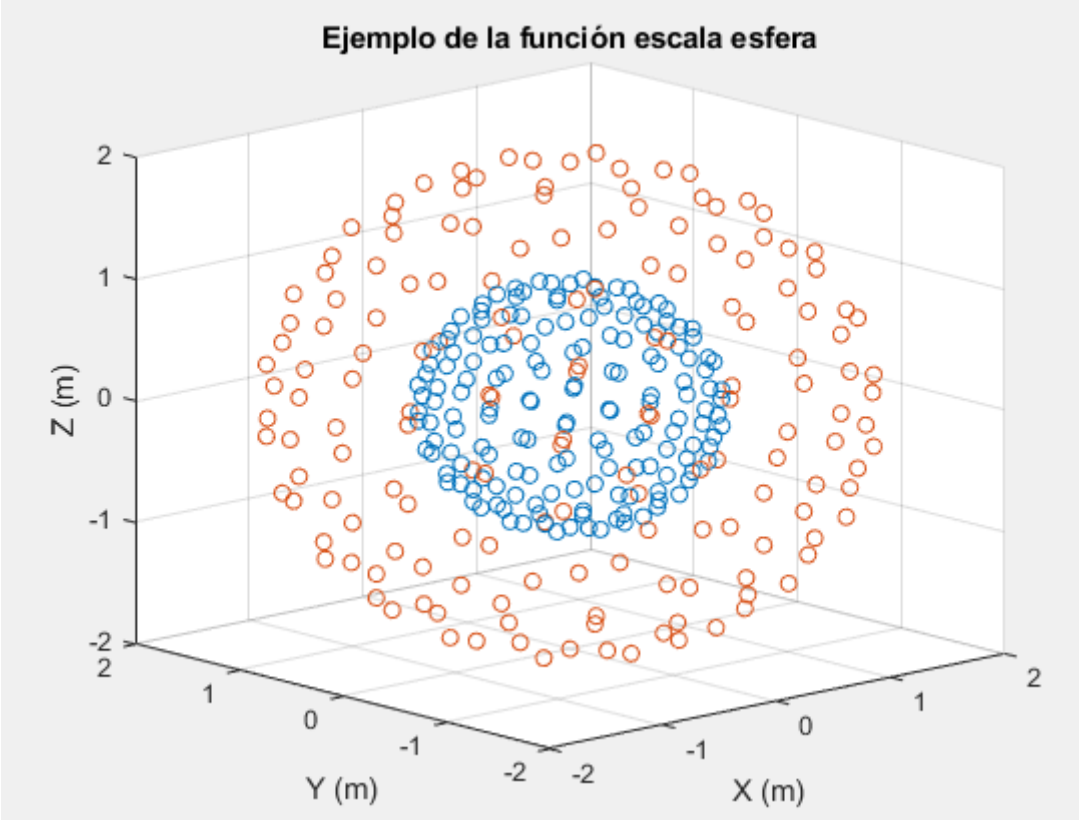


Imagen 13: Esferas de puntos antes y después de utilizar la función escala_esfera. En azul, la esfera de radio 1 y en naranja la esfera de radio 2.

elevacion_esfera

Esta función se encarga de quitar de la matriz los puntos que no se encuentren dentro del intervalo de elevación indicado. Por lo que, tan solo se tomará los puntos que se encuentren dentro del intervalo de alturas indicado. Es necesario utilizar esta función para tomar tan solo puntos a los que pueda acceder el robot sin problemas, pudiendo evitar puntos que estén por debajo del suelo o demasiado altos.

A la entrada de esta función se le deberá de indicar el intervalo de ángulo de elevación, la escala que se le ha aplicado a la matriz de puntos y por último, la matriz que será procesada. Los valores pueden oscilar de $-90º$ a $90º$, siendo el intervalo que toma todos los puntos $[-90º, 90º]$. A continuación, se muestra un ejemplo donde se toma el intervalo de puntos de $0º$ a $90º$ y radio uno de la esfera:

```
%Se toma el valor de los puntos de la superficie de la esfera.
Esfera= csvread('sphere2.csv');

%Se aplica el criterio de la elevación.
%Se ha de introducir primero el ángulo menor y después el mayor, con
%valores entre -90 y 90.
PuntosFinal = elevacion_esfera([0,90],1,Esfera);
```

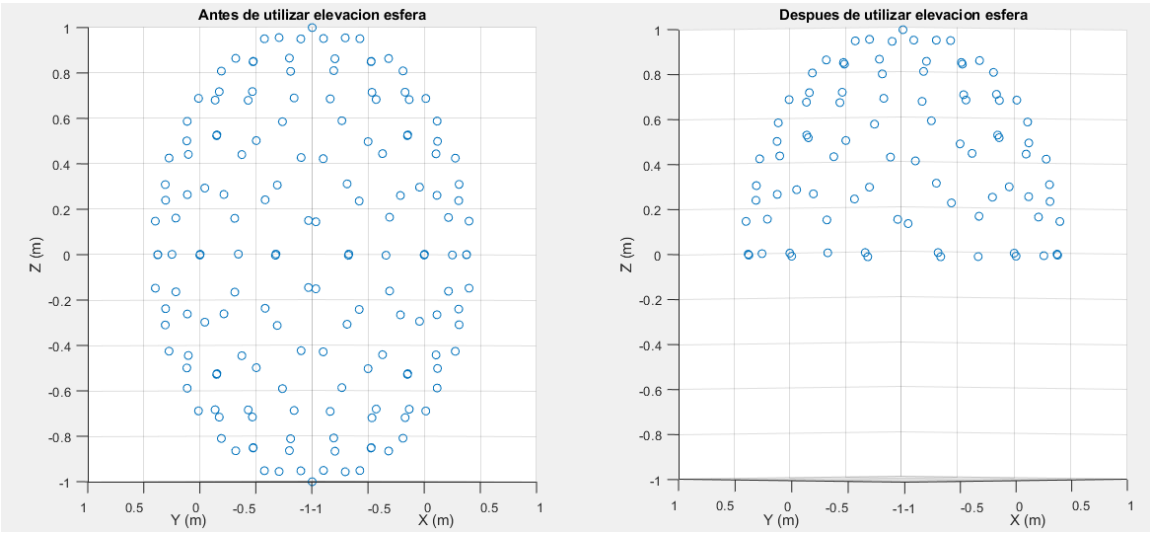


Imagen 14: Ejemplo de uso de la función elevación_esfera, para un intervalo de elevación de [0,90]º y escala = 1

azimut_esfera

Esta función tiene un comportamiento parecido a la función *elevacion_esfera*, pero en este caso se quitarán los puntos que no se encuentren dentro del intervalo indicado de azimut. Cabe destacar que en el desarrollo de esta función aparecieron una mayor cantidad de problemas. Esto se debía a que para distintos puntos podía existir la misma solución trigonométrica, por lo que hay que distinguir en todo momento que solución se está tomando. También se centró el desarrollo en poder utilizar intervalos de ángulos más flexibles. Esto se ve reflejado en que se tomará siempre el área comprendida desde el primer ángulo al segundo ángulo del intervalo. Esto significa que los puntos obtenidos por un intervalo de azimut $[-90,90]º$ no será igual que con $[90,-90]º$, esto se puede observar en la *Imagen 15.*

A continuación, se muestra un ejemplo del funcionamiento donde el intervalo de azimut en el primer caso es de$[-30,90]º$ y en el segundo caso de $[90,-30]º$:

```
%Se toma el valor de los puntos de la superficie de la esfera.
Esfera= csvread('sphere2.csv');

%Se aplica el criterio del azimut.
%Se tomará el área recogida desde el primer ángulo que se introduzca al
%segundo y los valores deben encontrarse entre -180 y 180.
%De escoger el mismo valor para las dos entradas se tomará el azimut
%completo.
%azimut = array(1x2).
%E = array de puntos.(:x3), siendo el orden X,Y,Z.
PuntosAzimut_1 = azimut_esfera([-30,90],Esfera);

PuntosAzimut_2 = azimut_esfera([90,-30],Esfera);
```

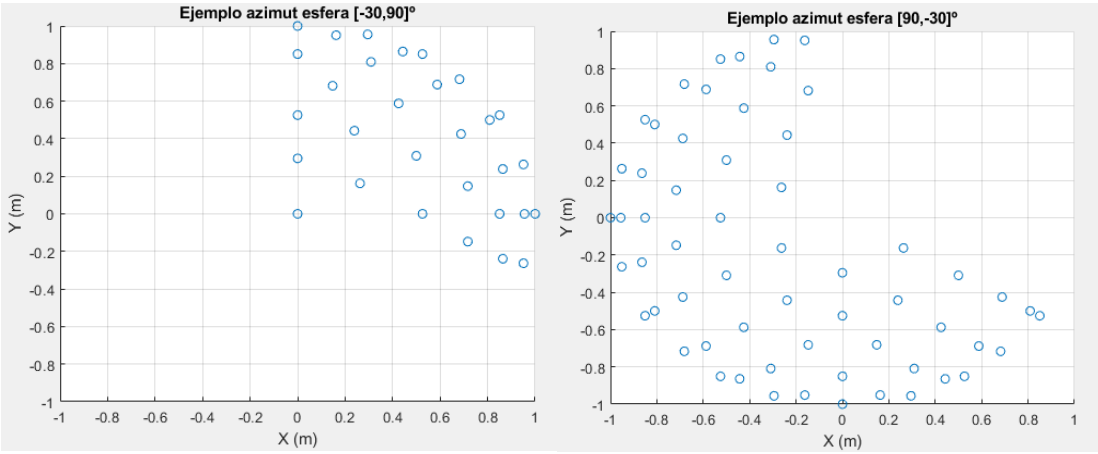


Imagen 15 :Ejemplo de la función azimut_esfera. Con valor del intervalo de azimut [-30,90]º (izquierda) y [90,-30] (derecha).

desplazamiento_esfera

Esta función se encarga de desplazar el centro de la superficie esférica de puntos. Esto es necesario debido a que el objeto a escanear no se va a encontrar en la coordenada $(0,0,0)$, ya que corresponde con la base del robot, si no que habrá que colocarlo en el punto donde le vaya a ser más fácil al robot alcanzar todas las posiciones. Para conseguir esta funcionalidad se ha sumado a cada punto de la matriz el valor del nuevo centro de la esfera de puntos.

A continuación, se muestra un ejemplo de uso de esta función, en la que se muestra la superficie esférica original centrada en el origen y la superficie esférica con el centro desplazado al punto $(2,0,0)$ (Imagen 16):

```
%Se toma el valor de los puntos de la superficie de la esfera.
Esfera= csvread('sphere2.csv');

%Se aplica el desplazamiento para centrarlo en el objeto.
%Despl = [X,Y,Z];
EsferaDesp = desplazamiento_esfera([2,0,0],Esfera);
```

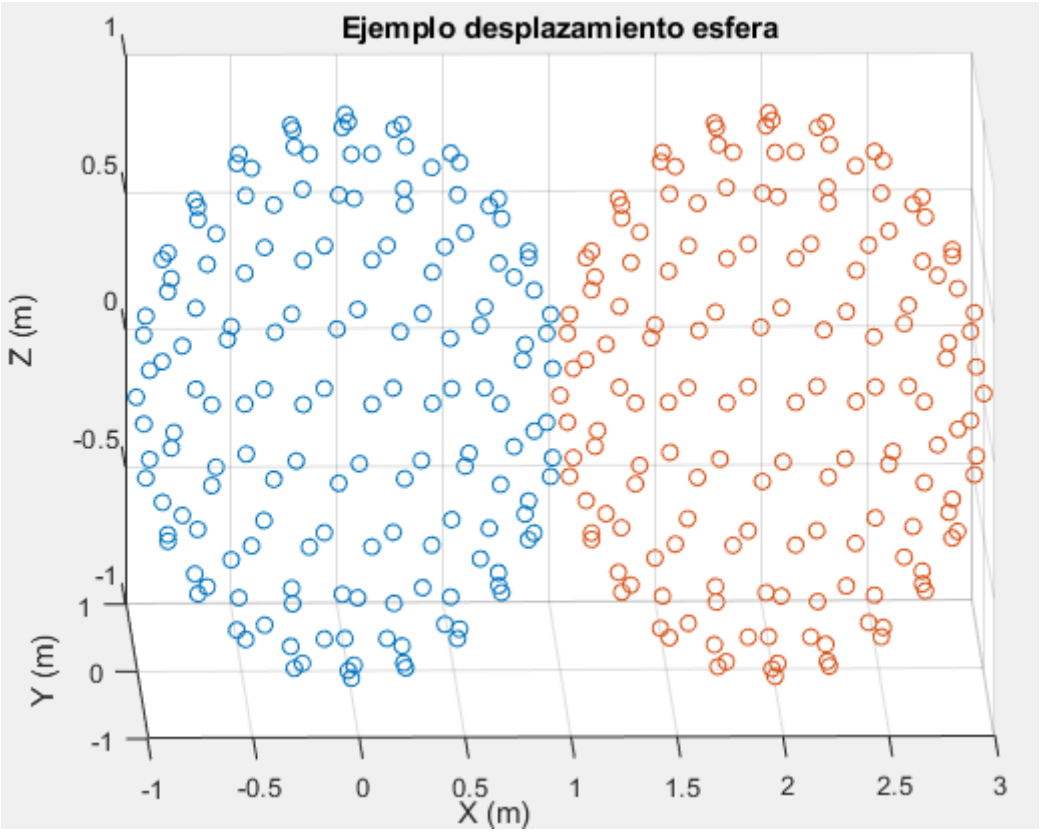


Imagen 16: Ejemplo de la función desplazamiento_esfera. La superficie esférica azul es la original, con centro en (0,0,0). La superficie esférica naranja es la obtenida con la función, con centro en (2,0,0).

posiciones_escaner

En esta función se llama a todas las funciones mencionadas anteriormente para obtener una superficie esférica que cumpla los requisitos de escala, elevación, azimut y desplazamiento introducidos por el usuario. La síntesis de todas estas funciones permite al usuario obtener de forma sencilla los puntos por lo que tiene que pasar el robot para realizar el escaneo de una pieza, evitando posiciones extremas.

A continuación, se muestra un ejemplo del uso de esta función, en la que se muestra una superficie esférica de radio $0.1\ m$, una elevación de $[0,90]º$, sin restricción en el azimut y un desplazamiento de $[0.2,0,0]\ m$ (Imagen 17):

```
%En esta función se obtiene los puntos de una superficie esférica
%cumpliendo unos parámetros de desplazamiento escala, elevación y
%azimut.
%desp       = [X,Y,Z]
%escala     = valor de la escala.  (Valores > 0)
%elevacion  = array(1x2).          (Entre -90 y 90. 1° Menor 2° Mayor)
%azimut     = array(1x2).          (Entre -180 y 180).

desp        = [0.2,0,0];
escala      = 0.1;
elevacion   = [0,90];
azimut      = [-180,180];

Esfera = posiciones_escaner(desp,escala,elevacion,azimut);
```

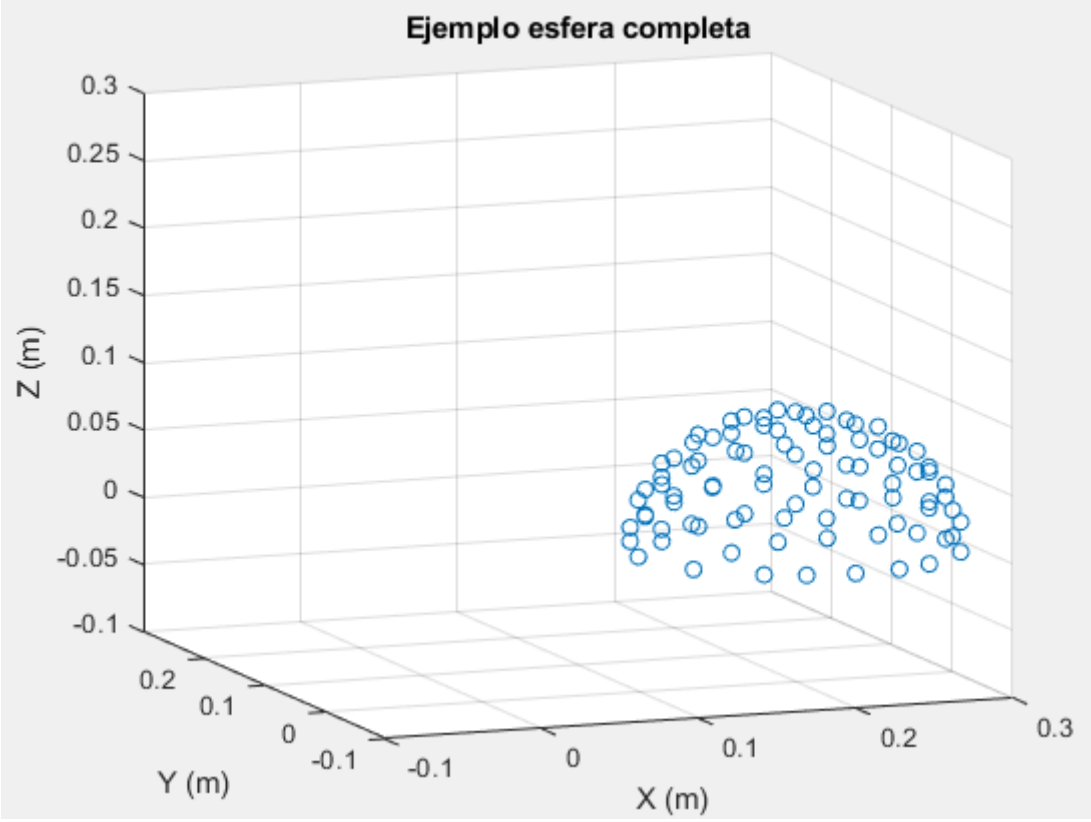


Imagen 17: Ejemplo de la función posiciones_escaner. Con parámetros: desp = [0.2,0,0], escala = 0.1, elevación = [0,90] y azimut = [-180,180].

Cabe destacar también el orden en el que se ejecuta cada función en *posiciones_escaner*, ya que en último lugar se hace el desplazamiento de los puntos. Por lo tanto, al valor del desplazamiento no se le aplica el factor de escala, por lo que el valor que indique el usuario será el desplazamiento final. Esta decisión se tomó ya que, de aplicar la escala después del desplazamiento el funcionamiento es menos intuitivo. También se utiliza antes las funciones *azimut_esfera* y *elevación_esfera*, ya que haciendo antes el desplazamiento puede llegar a dar problemas en la solución del problema trigonométrico.

pose_puntos

Con la función esfera completa se ha conseguido solucionar el problema de la selección de posiciones donde el robot debe hacer fotos. Ahora es necesario resolver la segunda parte del problema, en la que se debe obtener la orientación del robot en cada punto. Es decir, con la función *pose_puntos* se va a conseguir que para cada posición dada en *posiciones_escaner* se obtenga una matriz de rotación en la que el eje Z apunte hacia el centro de la superficie esférica.

Esta función ha presentado bastantes dificultades. Se ha tenido que investigar en profundidad los conceptos de matriz de rotación y la composición de rotaciones. Además, para comprobar cada una de las soluciones se ha tenido que diseñar un sistema de testeo.

El sistema de testeo consiste en representar tres puntos en el espacio. El primero es un punto aleatorio de la superficie esférica de puntos, el segundo es un punto en $(0,0,0)$ y el tercero es el obtenido al dibujar un punto a una distancia $(0,0,1)$ del sistema de coordenadas de la matriz de rotación a comprobar. La prueba indicará que la matriz de rotación es correcta cuando los tres puntos estén alineados (Imagen 18).

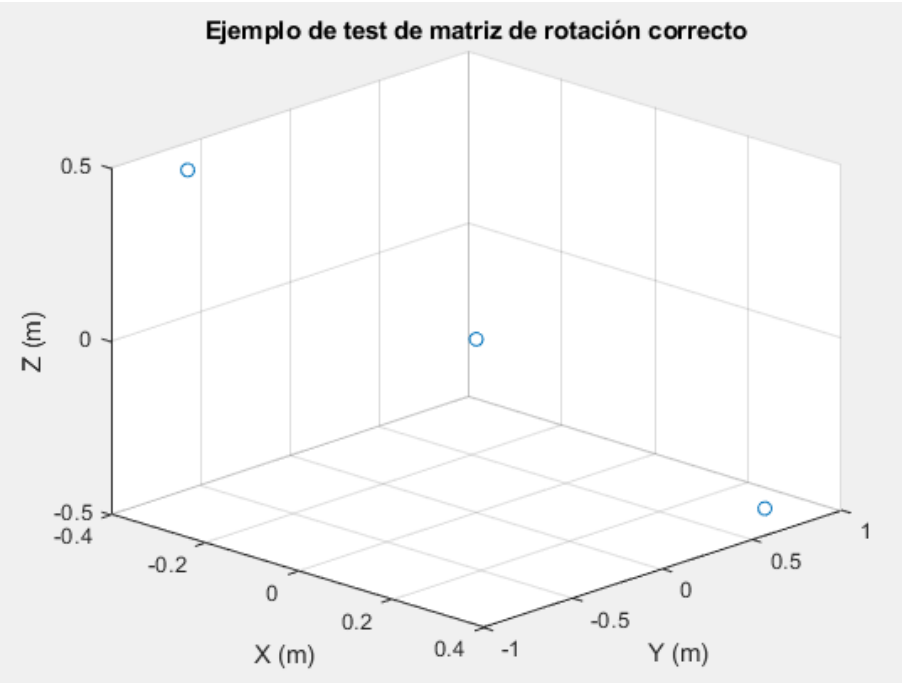


Imagen 18: Ejemplo en el que el test de la matrices de rotación es positivo.

Después de desarrollar el sistema de testeo se comenzó el desarrollo de la función *pose_puntos.* En primer lugar, se estudió en profundidad la composición de rotaciones y se intentó calcular por relaciones trigonométricas qué rotación en X, Y y Z era necesaria para conseguir que el eje Z del nuevo sistema de coordenadas apunte al centro de la esfera. Se intentó distintas composiciones de rotaciones, pero no se obtuvo ningún resultado positivo en el test, por lo que se cambió de estrategia para la resolución de este problema. A continuación, se muestra un extracto del código donde se obtiene los ángulos y se hace la composición de rotaciones para una solución en la que se utilizaba una rotación en Z y una en Y:

```
%Obtención del ángulo de rotación en Z mediante la función atan2.
angulo_z = atan2(0-Esfera(20,2),0-Esfera(20,1))
angulo_z_deg = angulo_z*180/pi

%Matriz de rotación en Z
Rz =[cos(angulo_z) -sin(angulo_z) 0;
    sin(angulo_z) cos(angulo_z) 0;
    0              0            1]
%Obtención del ángulo de rotación en Y mediante la función atan2.
angulo_y = atan2(0-Esfera(20,3),0-Esfera(20,1))
angulo_y_deg =angulo_y*180/pi

Ry = [cos(angulo_y) 0 sin(angulo_y);
        0           1       0;
      -sin(angulo_y) 0 cos(angulo_y)]

%Composición de rotaciones.
Rt = Rz*Ry

%Obtención del punto para el sistema de testeo.
Comp = Rt' *[0;0;1]
```

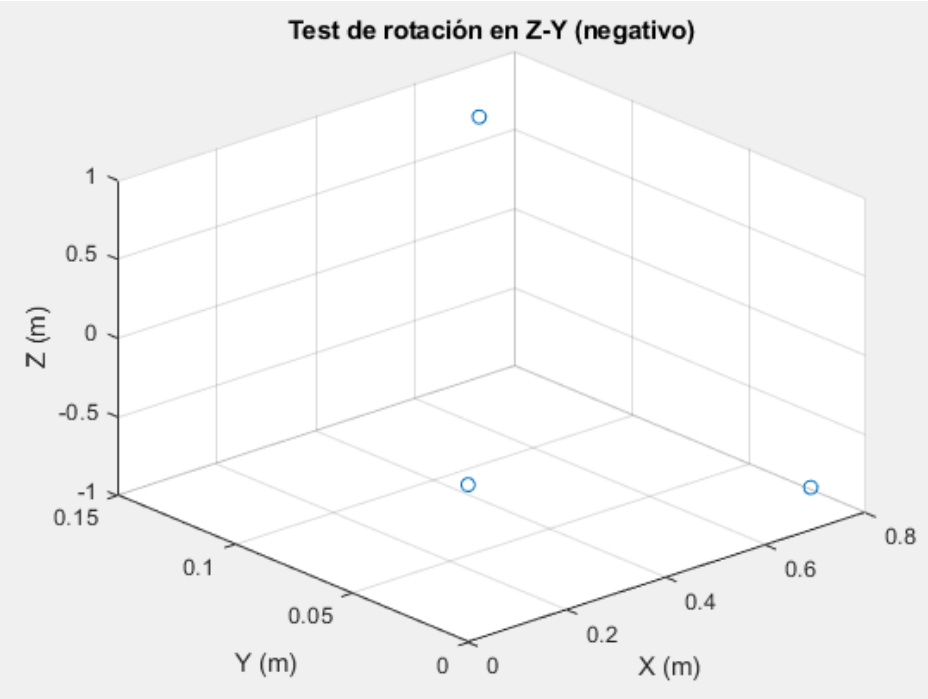


Imagen 19: Test de la solución obenida con una rotación en Z y una en Y. La matriz de rotación obtenida no apunta en la dirección correcta.

La siguiente estrategia de resolución del problema parte de comprender cada elemento de la matriz de rotación. En una matriz de rotación, cada fila representa el eje del sistema de referencia final respecto al sistema de referencia inicial. Esto significa que, a cada fila le corresponderá un vector unitario (con las medidas del sistema de referencia inicial) que representará la dirección del eje del nuevo sistema de referencia.

Con esta información, se puede plantear la solución del problema, ya que el vector unitario del eje Z de nuestro nuevo sistema de coordenadas es conocido. Este valor se puede obtener con el punto de la esfera donde se esté calculando los ejes y con el punto central de la superficie esférica, ya que el eje Z apuntará del primero al segundo.

Una vez calculado el vector unitario del eje Z, es necesario tomar una expresión general de composición de rotaciones. En este caso, se tomó la expresión correspondiente a una composición de rotaciones formada por: una rotación respecto al eje OX, una rotación respecto del eje OZ y por último una rotación respecto al eje OY. En la siguiente imagen, se muestra de forma detallada cómo es esta composición de rotaciones:

$$\mathbf{R} = \mathbf{R}_{y,\phi}\mathbf{R}_{z,\theta}\mathbf{R}_{x,\alpha} = \begin{bmatrix} C\phi & 0 & S\phi \\ 0 & 1 & 0 \\ -S\phi & 0 & C\phi \end{bmatrix}\begin{bmatrix} C\theta & -S\theta & 0 \\ S\theta & C\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \\ 0 & C\alpha & -S\alpha \\ 0 & S\alpha & C\alpha \end{bmatrix} =$$

$$= \begin{bmatrix} C\phi C\theta & S\phi S\alpha - C\phi S\theta C\alpha & C\phi S\theta S\alpha + S\phi C\alpha \\ S\theta & C\theta C\alpha & -C\theta S\alpha \\ -S\phi C\theta & S\phi S\theta C\alpha + C\phi S\alpha & C\phi C\alpha - S\phi S\theta S\alpha \end{bmatrix}$$

Imagen 20: Composición de rotaciones OXYZ. (Apuntes de Sistemas Robotizados - Geometría)

Como conocemos el valor de cada uno de los elementos de la última fila, ya que corresponde con el vector unitario calculado anteriormente, es posible calcular el valor de los distintos ángulos de las rotaciones. Para ello, se ha planteado desarrollado la función *AngulosRot2* que se encarga de resolver el sistema de ecuaciones formado por la última fila de la matriz y el vector unitario calculado. Con el valor de las rotaciones calculado, ya se puede formar la matriz de rotación completa y comprobar si es correcto con el sistema de testeo.

Esta estrategia sí que funcionó y se obtuvo el resultado mostrado en la Imagen 18. Tras esto, se formó la función *pose_puntos*, en la que se realizaba este procedimiento para todos los puntos que tenga en la matriz de entrada y a la salida se obtiene un cell array de matrices 4x4 con la pose de las vistas.

puntos_ejes

Esta es la última función que se desarrolló y tiene como finalidad facilitar la representación de las orientaciones en gráficos. Para ello, se generan tres puntos, uno en cada dirección de un eje. Con estos puntos ya se puede representar las rectas correspondientes a cada eje. A continuación, se muestra un ejemplo de uso de esta función:

```
%Comprobacion de la pose de los puntos.
%El valor que divide a la escala se utiliza para controlar el tamaño de
los
%ejes.
long_ejes = escala/5000;
[X,Y,Z] = puntos_ejes(Cell,long_ejes);
Z = Z + E;
Y = Y + E;
X = X + E;

%Eje Z de cada punto.Dibujar
plot3([E(:,1)';Z(:,1)'],[E(:,2)';Z(:,2)'],[E(:,3)';Z(:,3)'],'b')
%Eje Y de cada punto.Dibujar
plot3([E(:,1)';Y(:,1)'],[E(:,2)';Y(:,2)'],[E(:,3)';Y(:,3)'],'g')
%Eje X de cada punto.Dibujar
plot3([E(:,1)';X(:,1)'],[E(:,2)';X(:,2)'],[E(:,3)';X(:,3)'],'r')
```

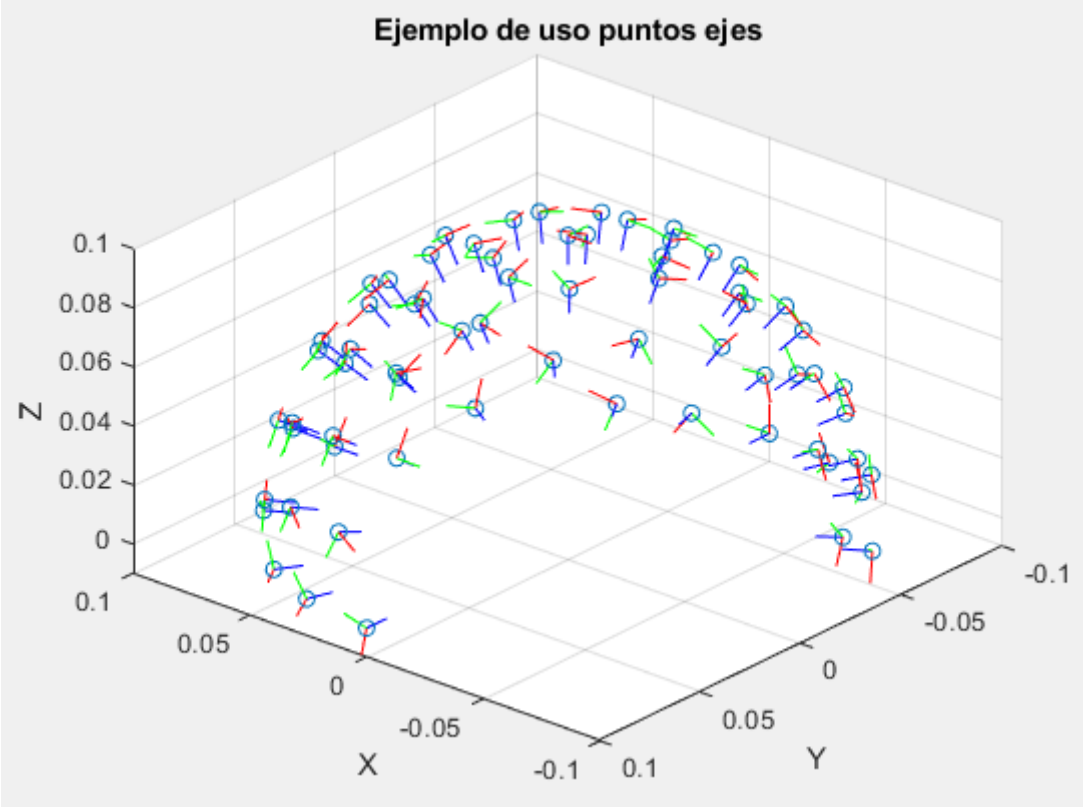


Imagen 21: Ejemplo de uso de la funcion "puntos_ejes". En color azul se muestra el eje Z, en rojo el X y en verde el Y.

A esta función también hay que indicarle la longitud que tendrán los ejes. Este valor se suele calcular a partir del parámetro de la escala de la superficie esférica.

3.1.2 Resultados de la obtención de poses en el script de Matlab.

Como resultado final de este apartado se ha conseguido obtener las poses de las vistas donde tendrá que pasar el robot llamando a tan solo dos funciones, *posiciones_escaner* y *pose_puntos.* Esto permite tener la funcionalidad requerida en funciones muy compactas y fáciles de utilizar.

Por otro lado, también se ha hecho un último test en el que se carga una imagen 3D y se tiene que generar una cúpula de puntos su alrededor. Esto simula la situación real de tener que adaptar los parámetros para el objeto que se va a escanear. Tras este test se obtuvo el siguiente resultado:

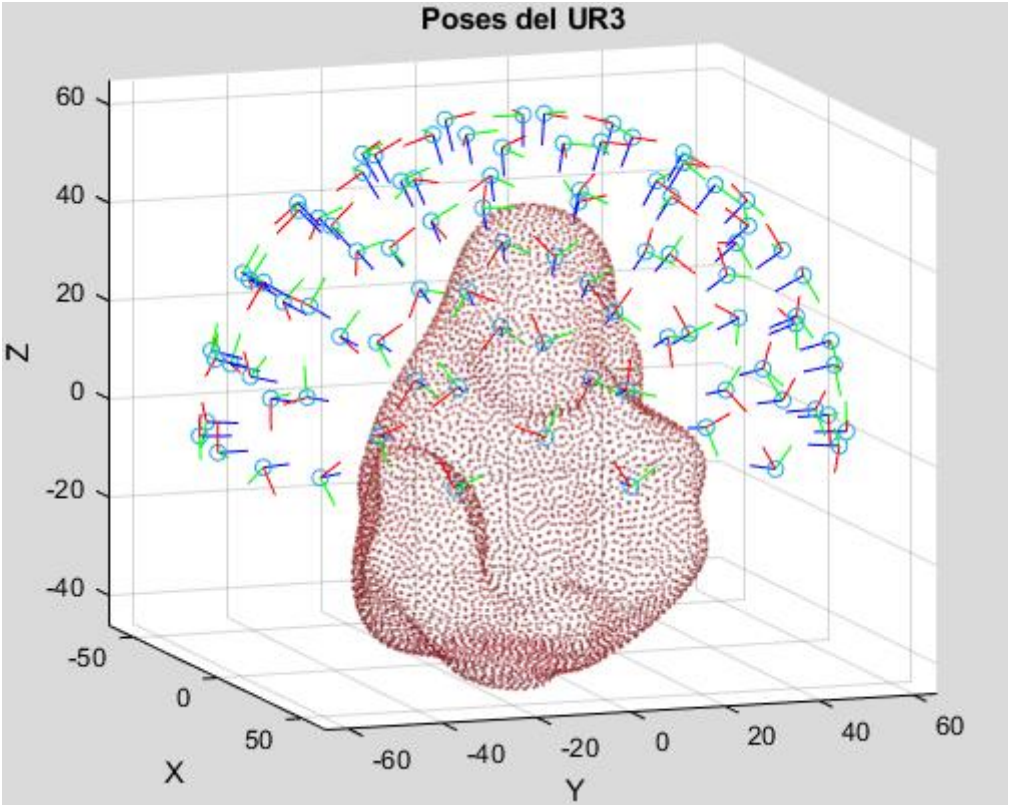


Imagen 22: Imagen de la prueba con un objeto 3D.

En este ejemplo, se ha cargado una imagen de una pieza en forma de mapa de puntos. Para hacer que la cúpula encaje con el tamaño y forma de la figura se ha aplicado una escala a la superficie esférica de 60. En las pruebas que se realicen con el robot UR3 se deberá tomar en todo momento escalas menores que 1, ya que los datos que lee el robot están en metros y los límites de distancia del robot están a menos de un metro.

3.1.3 Generación de trayectorias según los puntos obtenidos.

Este es otro apartado muy importante para la correcta toma de imágenes en el escaneo. Se debe asegurar que las trayectorias que sigue el brazo robot en ningún momento colisionen con el objeto a escanear.

Por ello es necesario que los puntos se recorran en un anillo concéntrico alrededor del objeto en cuestión. Para conseguir esto es necesario que no se cambien de orden las poses en ningún momento. Del fichero de posiciones de la superficie esférica, ya se encuentran las posiciones en el orden necesario para seguir trayectorias en anillo alrededor del objeto central.

Por otro lado, también se deberá mover el robot a velocidades lentas, para así evitar mayores daños de haber una colisión y además, se evita problemas de desplazamientos de la cámara.

3.2 PROGRAMACIÓN DEL MOVIMIENTO DEL UR3

En este apartado se ha desarrollado distintos programas para mover al robot a distintas posiciones. Para ello, se ha programado en el entorno URsim y las pruebas se han realizado en el simulador del que dispone URSim.

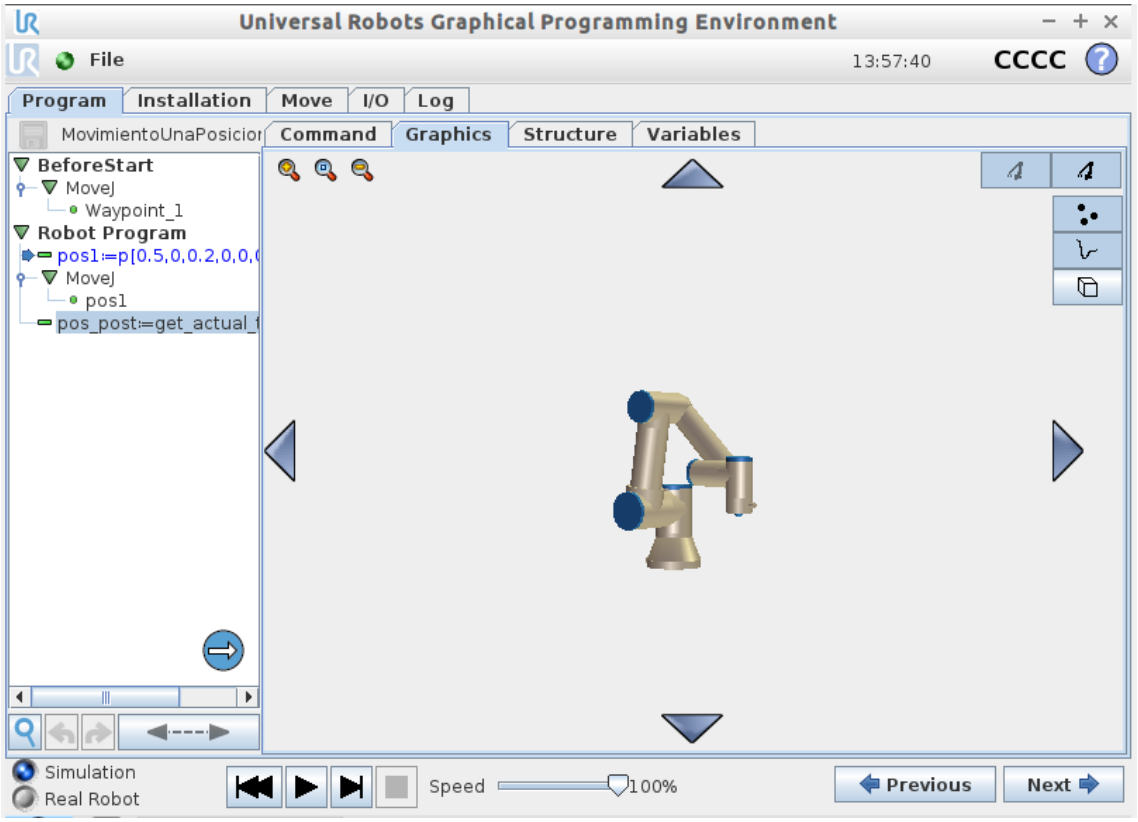


Imagen 23: Entorno URSim en el visulizador del simlador.

Este apartado es una base del proyecto, por lo que se debe entender completamente las funciones que se utilizan para después poder comunicarlo con los otros apartados. Para ello se han desarrollado los siguientes programas:

MovimientoUnaPosición

El objetivo de este programa es mover el efector final del robot a una pose indicada. Para ello, se le debe indicar una posición en X-Y-Z y la rotación en X-Y-Z. Después, se introduce esa pose a la función *MoveJ()* y se obtiene el movimiento deseado.

Para introducir variables de tipo pose, se debe indicar de la siguiente manera:

$$p[x, y, z, rx, ry, rz]$$

Por último, cabe destacar la parte más importante de este programa. Las rotaciones indicadas en el tipo de variable *pose* deben ser del tipo *axis-angle*, por lo que para introducir las orientaciones obtenidas en el apartado anterior se deberá hacer una transformación.

MovimientoPosFichero

El objetivo de este programa es hacer que el robot alcance todas las posiciones indicadas en un fichero. Este desarrollo es la base sobre la que se añadirán el resto de apartados. Se cambiará los datos del fichero por los que se reciba por socket y se obtendrá el programa final.

Por tanto, es un programa útil para observar algunos problemas que pueden aparecer con el movimiento del robot.

Para conseguir este objetivo, el programa tiene dos partes. Una preparación, llamada *BeforeStart,* y el bucle principal, llamado *RobotProgram.*

En el apartado *BeforeStart* se inicializan las variables y se mueve el robot a una posición inicial. Después, en el bucle principal, cada iteración se lee una pose de un fichero y se resuelve la cinemática inversa indicando qué solución debe tomar y finalmente, se mueve el robot a la posición indicada.

Para la resolución de la cinemática inversa se ha utilizado la función *get_inverse_kin.* Es necesario utilizar esta función ya que las soluciones de la cinemática inversa que toma la función *MoveJ* por defecto pueden hacer que el robot intente alcanzar posiciones imposibles, como que el codo esté por debajo del suelo (Imagen 24) o que partes del robot se atraviesen entre ellas.

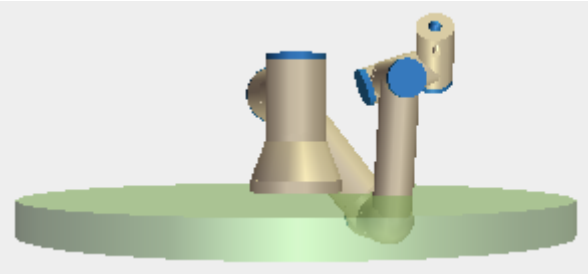


Imagen 24: Ejemplo de codo por debajo del suelo

3.3 PROGRAMACIÓN DE LA COMUNICACIÓN POR SOCKETS

En este apartado se ha desarrollado la base para la comunicación que se utilizará en el programa final. Esta comunicación es por vía de Sockets y se encargará de conectar el ordenador con el robot UR3. Debido a que el desarrollo del proyecto se ha tenido que realizar en simulador, se ha tenido que generar una comunicación vía Socket con el simulador, lo que ha añadido dificultades extra a este apartado.

Para la resolución del problema de la comunicación, se ha seguido los siguientes pasos:

Comprobación de la comunicación con *SocketTest*

Este es el primer paso que se ha realizado en el desarrollo de la comunicación. En él se ha utilizado un programa para comprobar si la comunicación vía socket con el simulador es posible. Este programa es el *SocketTest* y tiene la siguiente interfaz que se muestra en la imagen 25:

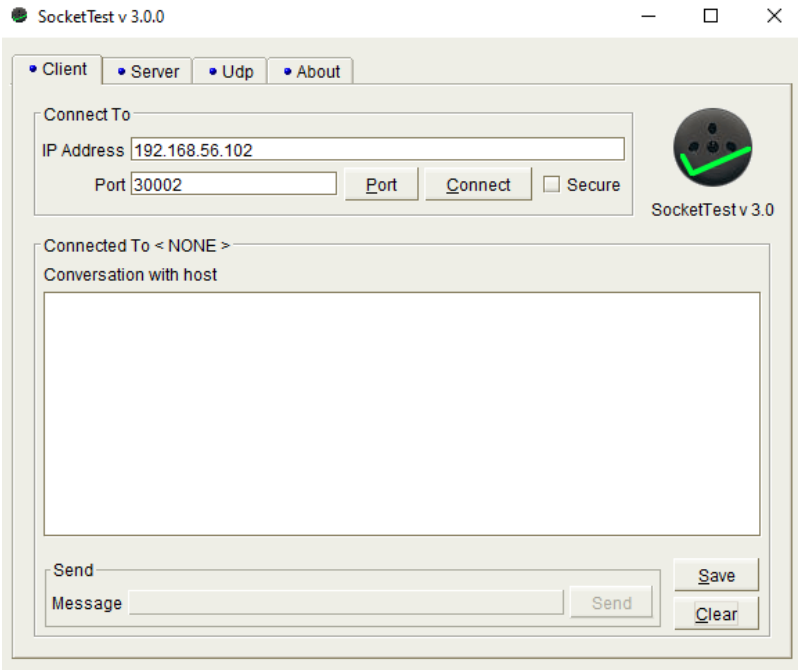


Imagen 25: Interfaz del programa SocketTest

Para realizar la comprobación se debe conectar a la IP de la máquina virtual y por un puerto dedicado a la comunicación socket, como por ejemplo el 30002. Después se puede realizar la conexión y mandar comandos desde el recuadro de Message. Se va a utilizar el comando *set_digital_out(0, True)* con el cual se activará una salida digital en el simulador del robot.

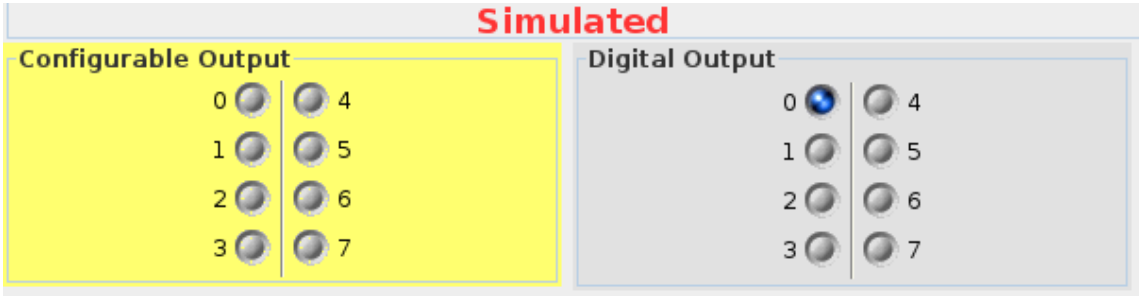


Imagen 26: Resultado de utilizar el comando set_digital_out(0,True), en el que se ha activado la salida digital 0 del simulador.

Con esta comprobación se demuestra que es posible realizar la comunicación vía socket con el simulador. Por lo que se puede pasar al siguiente paso, en el que se va a desarrollar la comunicación Matlab-URSim.

Comunicación por Sockets de Matlab con URSim

Aquí se va a desarrollar el Script de Matlab y de URscript necesario para poder mandar valores de pose desde Matlab al URSim y mover el robot a esa pose. Esta es una parte fundamental del proyecto, ya que es necesario tener un sistema de comunicación ordenador-robot para poder coordinar los movimientos del robot con la toma de imágenes en el escaneo.

En la solución de comunicación aportada el ordenador hará la función de servidor y el robot UR3 de cliente. Esto se ha implementado de la siguiente manera en Matlab y URSim:

Script de Matlab

```
%Puerto de comunicación
port = 30002;
%Configuración de servidor para cualuier ip que se intente conectar por
el
%puerto indicado.
s = tcpip("0.0.0.0", port,'NetworkRole', 'server');
s.terminator = '';
%Se abre la conexión y se espera a que se conecte el robot.
fopen(s);

disp('Conexión correcta');
```

Script de URSim

```
open:=socket_open("192.168.56.1",30002,"socket1")

     Loop open≟ False

       open:=socket_open("192.168.56.1",30002,"socket1")
```

El puerto utilizado para realizar la comunicación es el mismo que se ha utilizado con el SocketTest, ya que ha dado buenos resultados y corresponde a un puerto para la comunicación Socket.

En el script de Matlab, se ha indicado que la IP es "0.0.0.0" esto se debe a que si indicamos la IP de la máquina virtual la conexión daba error. Por tanto, se ha indicado la opción de conectar a cualquier IP que pida conectarse por el puerto 30002. De trabajar con el robot físico, convendría, por motivos de seguridad, introducir en ese apartado la IP del robot.

Con estos dos Scripts, ya se puede establecer una conexión entre el Matlab y el URSim, por lo que el siguiente paso consiste en mandar datos en ambas direcciones. Para ello, se va a utilizar las siguientes funciones:

- **fprintf() – socket_read_ascii_float():** La primera función de ellas es la que se encarga de mandar un string por el canal socket que se ha abierto anteriormente. Mientras que la segunda función se encarga de leer por el puerto socket un string y transformarlo en una variable de pose o una lista de valores tipo float. Con estas dos funciones se podrá mandar desde Matlab las poses y recibirlas en el robot UR3. A continuación se muestra un ejemplo de uso de estas dos funciones:

  Matlab:

  ```
  %Se manda una pose vía socket

  str=sprintf("p[%f,%f,%f,%f,%f,%f]",PosX,PosY,PosZ,RotX,RotY,Rot)
  fprintf(s,str);
  ```

  Para mandar los datos utilizando *fprintf,* antes se genera un string de los datos que se quiere mandar mediante el *sprintf.* Después, este string se incluye a la entrada de la función *fprinf,* siendo la otra entrada el canal por el que se va a mandar, en este caso llamado *s.*

  URscript:

  ```
  pose1:=socket_read_ascii_float(6,"socket1")

  Loop pose1[0] ≠6

    Wait: 0.3

    pose1:=socket_read_ascii_float(6,"socket1")
  ```

  Para utilizar esta función, se requiere indicar a sus entradas el tamaño de la lista de valores que le va a llegar y el canal por el cual van a llegar los datos. También, se ha añadido un bucle, del cual no saldrá hasta recibir un dato correcto por el canal *socket1.*

- **Socket_send_int() – fread():** La primera función es del srcipt de UR y se encarga de mandar por el canal socket un número entero. Mientras que la función *fread* se encarga de leer los valores que le lleguen por el canal que se le indique. Por lo tanto, con esta pareja de funciones se conseguirá tener una comunicación del UR3 al Matlab. Esta comunicación es de gran utilidad para indicar al script de Matlab cuándo tiene que tomar una imagen y mandar una pose nueva al robot. A continuación se muestra un ejemplo de uso de estas dos funciones:

Matlab:

```
%Se espera a que el robot llegue a la posición y orientación
indicada.
while foto ~= 2
    foto = fread(s,1);
end
foto = 0;
```

En este ejemplo se está leyendo continuamente por el canal Socket y comprobando si se ha mandado un 2. Este fragmento se utiliza para esperar mientras que el robot se está desplazando a la nueva pose. Cuando el robot alcance la posición indicada, mandará un 2 que será leído por este fragmento de código y permitirá continuar con la ejecución de forma correcta.

A la entrada de la función *fread* hay que introducirle en primer lugar el canal donde se va a leer y en segundo lugar, qué cantidad de datos se van a leer.

URscript:

```
Loop 10 times
      image:=socket_send_int(2,"socket1")
```

En este fragmento de código se está mandando diez veces el valor entero 2 por el canal Socket. Este bucle de diez iteraciones es necesario para asegurarse de que el script de Matlab va a leer el valor que se ha mandado.

A la entrada de la función *socket_send_int* hay que indicarle en primer lugar qué valor se va a mandar y en segundo lugar por qué canal socket se va a mandar.

Uniendo todas estas funciones en un Script de Matlab y uno de URscrip se ha obtenido un script en el que se manda una posición y orientación desde Matlab y el robot es capaz de recibirla, moverse a esa posición e indicarle mandarle un mensaje a Matlab de que se ha alcanzado la posición.

Con todo esto ya se tiene el funcionamiento general del sistema. A este funcionamiento hay que añadirle la toma de imágenes y que el sistema barra todas las posiciones y orientaciones obtenidas con la función *pose_puntos.*

3.4 IMPLEMENTACIÓN DE LA CAPTURA DE IMÁGENES CON LA CÁMARA

En este apartado se ha desarrollado el código necesario para la toma di imágenes con la cámara de profundidad de RealSense D415. Para ello, se ha utilizado un wrapper de Matlab descargado de la cuenta de GitHub de IntelRealSense (Referencia 5).

En ese wrapper vienen las funciones necesarias para facilitar el uso de la cámara en Matlab. Estas funciones son tanto para la configuración de la cámara, como para la toma de imágenes en sí. Como durante el desarrollo del proyecto se ha tenido que trabajar con simulaciones y sin poder disponer de la cámara por lo que, para comprobar el correcto funcionamiento se ha realizado las pruebas leyendo un fichero *.bag*, que es el formato de imágenes que toman las cámaras de profundidad. Cabe destacar que los cambios que habría que realizar para la toma de imágenes con una cámara son mínimos, lo que justifica esta decisión. Los cambios que hay que realizar son los relacionados con la configuración de captura, el tiempo de exposición, la resolución y los FPS; que son cambios de parámetros dentro de la inicialización de la cámara.

El desarrollo de este apartado se ha realizado en dos pasos, en primer lugar se ha comprobado el uso de las funciones del wrapper con un script de ejemplo y en segundo lugar, se ha desarrollado una función para configurar la cámara y otra función para la toma de imágenes. A continuación, se explica en profundidad estos pasos:

rosbag_example

Esta es la función de ejemplo de lectura de un fichero *.bag* que trae el wrapper de Matlab para la lectura de ficheros *.bag*.

Con esta función se ha podido tomar las imágenes en profundidad y color a partir de un fichero .bag dado por el tutor del proyecto. El resultado obtenido es el siguiente, mostrado en la Imagen 27:

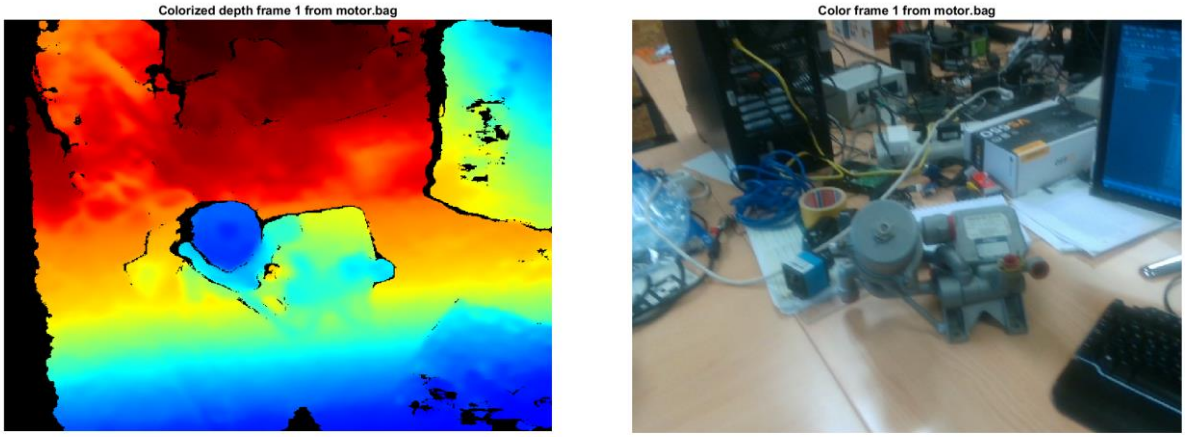


Imagen 27: Imagenes en profundidad y color obtenidas con la función rosbag_example

Esta es una versión modificada de la función desarrollada por el tutor del proyecto, por lo que se permite obtener imágenes en color y profundidad al mismo tiempo.

Gracias al uso de esta función se ha podido comprender mejor el uso del wrapper y de la toma de imágenes con cámaras de profundidad. Un punto importante es que los primeros frames capturados por la cámara deben ser desechados, ya que suelen ser tomas de imágenes

incorrectas. Por tanto, en el apartado de configuración de la cámara se deberá tomar cinco imágenes y desecharlas.

Después de analizar la función de ejemplo se ha desarrollado las funciones *preparación_camara y foto_camara, que serán las utilizadas en la solución final del proyecto.*

preparacion_camara

Esta función se encarga de la configuración inicial de la cámara y de la toma de los cinco frames que deben ser desechados. Esta función está completamente basada en la función *rosbag_example*, de donde se ha tomado el código relacionado a la configuración.

A continuación se muestra el código de la función *preparacion_camara:*

```
function [colorizer,pipe] = preparacion_camara(camera)
%Esta función prepara la configuración necesaria para tomar imágenes de
una
%cámara o de un archivo.

    % Make Config object to manage pipeline settings
    cfg = realsense.config();
    validateattributes(camera,     {'char','string'},     {'scalartext',
'nonempty'}, '', 'filename', 1);

    % Tell pipeline to stream from the given rosbag file
    cfg.enable_device_from_file(camera,0)

    % Make Pipeline object to manage streaming
    pipe = realsense.pipeline();
    % Make Colorizer object to prettify depth output
    colorizer = realsense.colorizer();

    % Start streaming from the rosbag with default settings
    profile = pipe.start(cfg);

    % Get streaming device's name
    dev = profile.get_device();
    name = dev.get_info(realsense.camera_info.name);

    % Get frames. We discard the first couple to allow
    % the camera time to settle
    for i = 1:5
        fs = pipe.wait_for_frames();
    end
end
```

Para poder separar la configuración de la toma de imágenes es necesario que la función devuelva la configuración del *colorizer* y del *pipe*. Estas variables deberán ser introducidas después a la entrada de la función *foto_camara*.

Esta función será llamada una sola vez en el programa principal, mientras que la función *foto_camara* se utilizará en cada toma de imágenes. Esta división permite liberar de muchos cálculos a Matlab, lo que aporta velocidad al proceso.

foto_camara

Esta función se encarga de tomar imágenes con la cámara o desde un fichero *.bag* y guardarlas en ficheros *.png.* Esta función también está basada en la función *rosbag_example*, pero en este caso se he realizado más modificaciones.

A continuación se muestra la estructura de esta función, donde se va a explicar las entradas y salidas que tiene:

```
function foto_camara(name,pipe,colorizer,cont)
```

Esta función no tiene ninguna salida, ya que al llamarla genera y guarda directamente las imágenes. Por otro lado, las cuatro entradas que necesita son las siguientes:

- **name**: esta variable se corresponde con un string para identificar de dónde se ha obtenido la imagen. Ese string se muestra en el título de las imágenes obtenidas.
- **pipe:** esta es una de las variables de la configuración de la cámara obtenido en la función *preparacion_camara*. Esta variable hace referencia al canal del que se obtiene las imágenes, por ejemplo: la cámara o el fichero *.bag*.
- **colorizer:** esta es la otra variable de la configuración de la cámara, que también se obtiene en la función *preparacion_camara*. Esta variable se utiliza a la hora de componer los colores de las imágenes de profundidad.
- **cont:** esta es una variable entera, que cuenta cuántas imágenes se han tomado. Se utiliza para nombrar los ficheros de cada imagen de forma distinta, por ejemplo: color0001, color0002, etc.

De introducir las entradas correctas a la función se obtendrá como resultado dos ficheros, uno de la imagen en profundidad y otro de la imagen en color. Estas imágenes se guardarán en una carpeta llamada *Fotos* y tendrán la siguiente estructura en el nombre: DephImageXXXX o ColorImageXXXX (el valor de las X es el de cont, rellenando por la izquierda con ceros). Se utiliza esta nomenclatura en las imágenes para facilitar su uso en programas de composición de imágenes 3D.

3.5 COORDINACIÓN DE TODOS LOS APARTADOS

En este apartado se ha obtenido la solución final de la programación. Se han juntado todos los apartados anteriores, realizando los cambios necesarios para asegurar el correcto funcionamiento.

Se ha obtenido dos scripts, uno de Matlab y otro de URscript, que permiten seleccionar una serie de puntos alrededor de un objeto y hacer que el robot pare en cada punto y tome una imagen del objeto. Estos scripts son *EscanerScript.m* y *Escaner.urp*, que a continuación van a ser explicados en profundidad:

EscanerScript.m

Este es el script de Matlab con el que se obtiene el funcionamiento completo del escáner 3D. Su funcionamiento es el siguiente. En primer lugar establecerá una conexión socket con el robot tal y como se ha mostrado en el apartado de *Programación de la comunicación Socket*. El programa se quedará en espera hasta que se conecte el robot. Después de realizar la conexión, el script continúa inicializando unas variables, obteniendo las poses mediante las funciones *posiciones_escaner y pose_puntos* y por último se inicializa la cámara con la función *preparacion_camara.* Esto se puede observar en el siguiente fragmento de código:

```
%Se inicializan las variables y se obtiene las poses según los parámetros
%indicados.
next = 0;
foto = 0;
[E] = posiciones_escaner([0.25,0,0.2],0.1,[0,90],[-125,125]);
[Cell] = pose_puntos(E,[0.25,0,0.2]);

%Se inicializa la cámara.
[colorizer,pipe] = preparacion_camara('motor.bag')
```

Cuando acabe la inicialización de la cámara comenzará el bucle principal del programa. Este se repetirá tantas veces como poses se hayan obtenido en *pose_puntos*.

El primer paso del bucle consiste en leer por el canal socket hasta recibir un 1, lo que significará que el robot está pidiendo una pose nueva. Esto se ha implementado como se muestra en el siguiente fragmento de código:

```
%Espera hasta que el robot necesite una pose nueva.
    while next ~= 1
        next = fread(s,1);
    end
    next = 0;
```

Cuando llegue el uno por el canal socket, el script continuará leyendo la primera matriz de transformación obtenida en *pose_puntos.* De esa matriz de transformación se obtendrá la posición y la matriz de rotación 3x3. Como los datos de rotación que acepta el robot son en formato *axis-angle*, se utiliza la función de Matlab *rotm2axang*, la cual transforma una matriz de rotación en las rotaciones en formato *axis-angle*. Con los valores de la posición y las rotaciones se forma el string que se mandará al robot mediante la función *fprintf*. Todo este funcionamiento se puede observar en el siguiente fragmento de código:

```
%Se obtiene las posiciones y orientaciones para mandarlas vía socket.
Mat = Cell{i};
RotMat = Mat(1:3,1:3);
%Rotaciones en formato axis-angle.
axang = rotm2axang(RotMat');

Rot(1,1) = axang(1)*axang(4);
Rot(1,2) = axang(2)*axang(4);
Rot(1,3) = axang(3)*axang(4);

PosX = Mat(1,4);
PosY = Mat(2,4);
PosZ = Mat(3,4);

%Se genera el script que será mandado por Socket.
str = sprintf ("[%f,%f,%f,%f,%f,%f]",PosX,PosY,PosZ,...
Rot(1),Rot(2),Rot(3));

if i == 1
       str = sprintf("[%f,%f,%f,%f,%f,%f]",PosX,PosY,PosZ,...
       Rot(1),Rot(2)+pi,Rot(3)-pi);
end
%Se mandan las posiciones y orientaciones.
fprintf(s,str);
```

Después de mandar la pose al robot el script de Matlab volverá a esperar hasta recibir una señal del robot que indique que ha llegado a la posición indicada. Esta espera se ha implementado de forma idéntica a la primera espera del bucle, por lo que no se va a mostrar el código.

Cuando el robot llegue a la pose y mande la señal, desde el script de Matlab se mandará tomar una imagen con la función *foto_camara*:

```
%Se toma la imagen.
foto_camara('motor.bag',pipe,colorizer,1)
```

Al terminar las fotos de profundidad y color habrá acabado una iteración del bucle principal y volverá al inicio del bucle, donde esperará de nuevo a que el robot pida una pose nueva.

Cuando el robot haya tomado imágenes en todas las posiciones y orientaciones indicadas, acabará el bucle principal, tras el cual se cerrará la comunicación Socket y la toma de imágenes de la cámara. Esto se ha implementado mediante estas dos funciones:

```
%Se cierra la comunicación y la toma de imágenes.
fclose(s);
pipe.stop();
```

Escaner.urp

Este es el script de URscript que se encarga de realizar la comunicación con el ordenador y de mover el robot a las distintas poses en el momento indicado.

Este script consta de dos partes, el *BeforeStart* y el *RobotProgram.* La primera parte se ejecutará una única vez y se utilizará para establecer la comunicación vía socket e inicializar las variables. El código correspondiente al apartado *BeforeStart* es el siguiente:

```
Program
   BeforeStart
     MoveJ
       Waypoint_1
     open:=socket_open("192.168.56.1",30002,"socket1")
     Cont:=0
     Loop open≟ False
       open:=socket_open("192.168.56.1",30002,"socket1")
     targetPos:=p[0,0,0,0,0,0]
     counter:=0
```

Este código presenta el fragmento para iniciar la comunicación, desarrollado en el apartado de *Programacion de la Comunicación por Sockets*. Además de inicializar la variable *targetPos*, a la que se irá cambiando los valores por los recibidos desde el script de Matlab.

La segunda parte del código es el bucle principal, que se repetirá tantas veces como datos vaya recibiendo. La primera acción que realiza el bucle es mandar la orden a Matlab para que mande una nueva pose y seguidamente se pondrá a leer el canal socket hasta conseguir un dato adecuado de la pose. Esto se puede observar en el siguiente extracto de código:

```
  next_point:=socket_send_int(1,"socket1")
  pose1:=socket_read_ascii_float(6,"socket1")
  Loop pose1[0] ≠6
    Wait: 0.3
    next_point:=socket_send_int(1,"socket1")
    pose1:=socket_read_ascii_float(6,"socket1")
```

Después de leer los datos recibidos del script de Matlab, se deben guardar en una variable tipo pose, ya que los datos que se han recibido son de tipo lista de floats. Para ello, se ha utilizado un bucle de seis iteraciones, que va colocando en la variable targetPos los valores de la lista de floats. Esto se puede observar en el siguiente extracto:

```
Loop 6 times
  targetPos[Loop_2] =pose1[Loop_2+1]
```

Con los datos guardados en una variable tipo pose ya se puede obtener la solución de la cinemática inversa correspondiente a tener el codo por arriba, tal y como se ha explicado en el apartado de *Programación del Movimiento del UR3*. Con esto se obtiene los ángulos de cada articulación para la posición y orientación requerida. Después, se puede introducir esos ángulos a la función *MoveJ* para mover el robot a la pose adecuada con la solución de la cinemática inversa adecuada. El código utilizado para esta funcionalidad es el siguiente

```
JointPos:=get_inverse_kin(targetPos,qnea=[0,-2.528,-2.224,0.072,1.5707,
1.95] ,maxPositionError=0.0001,maxOrientationError=0.0001)
MoveJ
   JointPos
```

Por último, cuando acabe el movimiento el programa mandará una señal por el canal socket para indicar que ya ha acabado el movimiento y se puede tomar las fotos. Tras esto, el bucle principal acaba y se vuelve a comenzar otra iteración, esperando a que se acabe de tomar las imágenes y que Matlab mande la nueva pose.

Conclusión

Con estos dos scripts acaba el desarrollo en forma de programación. En la simulación se consigue los resultados deseados, obteniendo las imágenes del fichero *.bag* al llegar a cada punto indicado. Cabe destacar que es necesario realizar pruebas en el entorno físico real, ya que ahí pueden aparecer nuevos problemas. Como por ejemplo no poder dar una vuelta 360º al objeto, ya que hay posiciones a las que el robot no pueda llegar tanto por cercanía al centro, como por estar demasiado lejos.

Estos problemas no están relacionados con la programación en sí, más bien, son problemas relacionados con la estrategia de la toma de imágenes. Por lo que, se explicarán en profundidad en el apartado de *Resultados* y *Problemas encontrados y soluciones adoptadas.*

3.6 CALIBRACIÓN DE LA CÁMARA

Este apartado es necesario para la implementación en el sistema real. Esto se debe a que para realizar una buena reconstrucción de un objeto 3D mediante fotogrametría es necesario tener la mayor cantidad de información disponible sobre la toma de imágenes.

Por lo tanto, antes de realizar la adquisición de imágenes se calibra la cámara con el fin de conocer su posición respecto a la base del brazo robot. Esta calibración se realiza mediante un patrón especializado. Para ello, es necesario conocer a qué distancia se ha colocado el patrón de calibración y de qué tamaño son los cuadrados. Después, dependiendo del tamaño que haya en los cuadrados fotografiados se puede obtener la posición de la cámara respecto al efector final en esa pose. Y puesto que la cámara está fija al efector final, esa distancia al efector final se mantendrá constante.

A continuación, se muestra la imagen tomada para realizar la calibración de la cámara (Imagen 28):

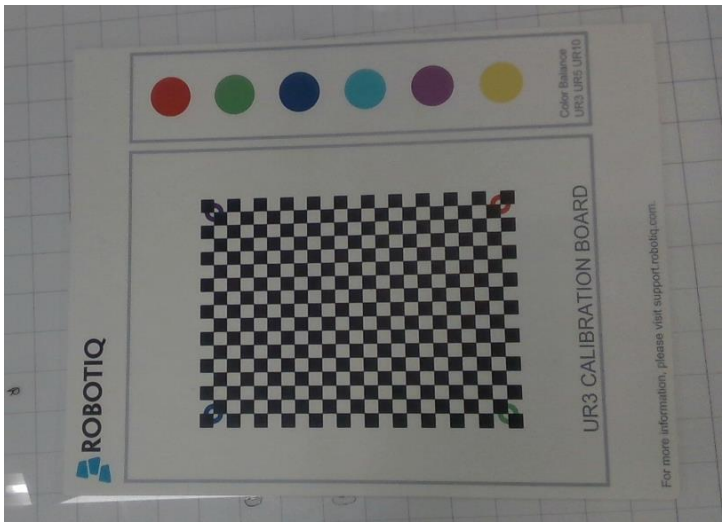


Imagen 28: Imagen tomada del patrón de calibración.

La calibración de la cámara se ha realizado en Matlab. Para ello se ha tomado y procesado una imagen para calcular los parámetros extrínsecos (Imagen 29). Y para los intrínsecos, se ha utilizado los siguientes parámetros del fabricante:

Calibration Values
$fovx$: 45.7790

$fovy$: 35.6023

$focalLength$: 4.7255

$principalPoint$: [2.2536 0.7304]

$aspectRatio$: 0.9637

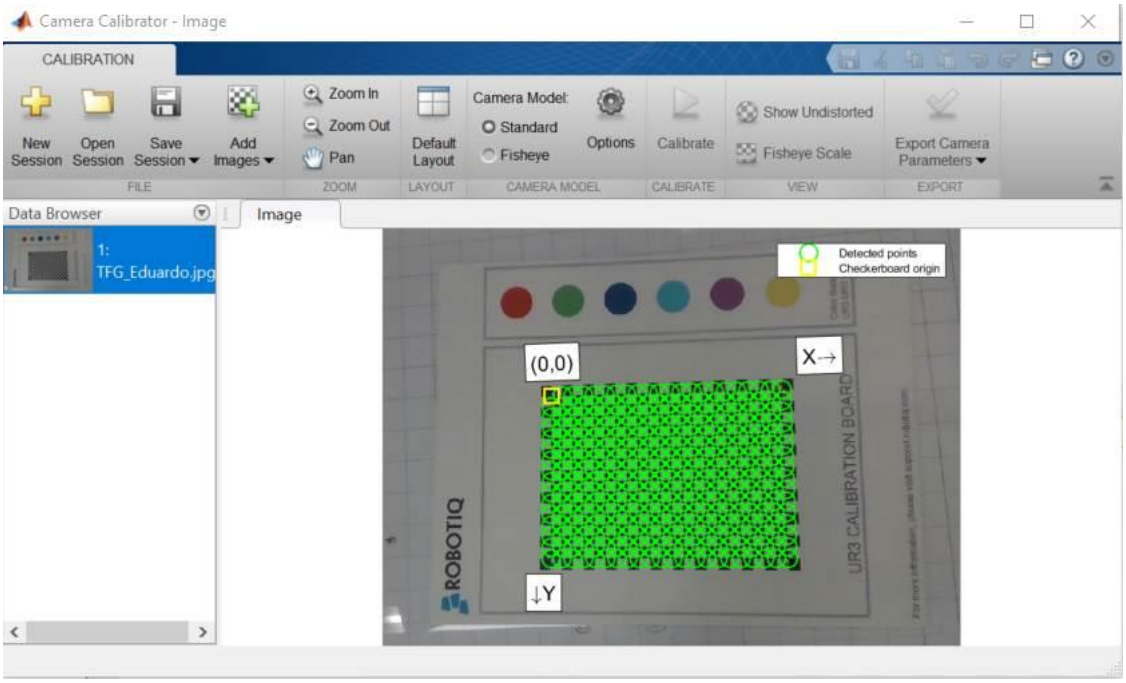


Imagen 29: Procesado del patrón de calibración para obtener los parámetros extrínsecos.

3.7 DISEÑO DEL ADAPTADOR DE LA CÁMARA

En este apartado se ha diseñado en SolidWorks un adaptador para poder acoplar cámaras al brazo robot UR3. Además, también se ha realizado todo el trabajo de impresión 3D para obtener un prototipo utilizable en el laboratorio. Por lo tanto, este apartado se va a dividir en el diseño de la pieza y la impresión:

Diseño del adaptador

Para este diseño se ha priorizado la posibilidad de adaptar distintos tipos de cámara al brazo, para ello, la sujeción principal de la cámara será a través de un tornillo de diámetro 1/4 de pulgada, de rosca tipo Whithworth. Se ha elegido este tipo de tornillo ya que es el estándar para trípodes y adaptadores de cámaras, por lo que encajará en la mayoría de cámaras.

En este diseño el acoplador de la cámara se engancha a la muñeca del robot, con una sujeción de tipo abrazadera. El resultado final de este diseño se puede observar en la Imagen 30 y en la Imagen 31 se puede observar el diseño acoplado al brazo robot:

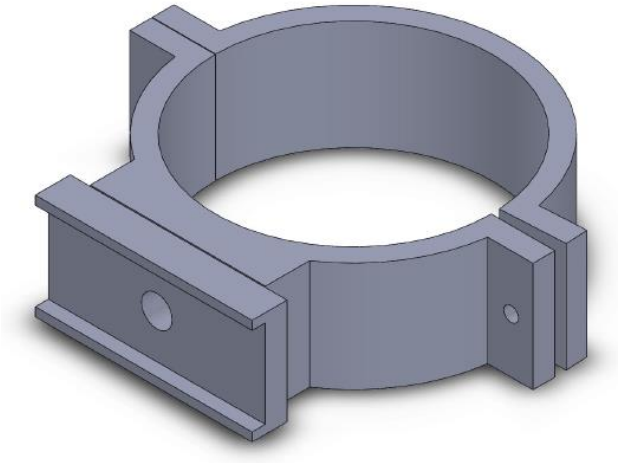


Imagen 30: Diseño realizado del adaptador de la cámara al brazo robot UR3.

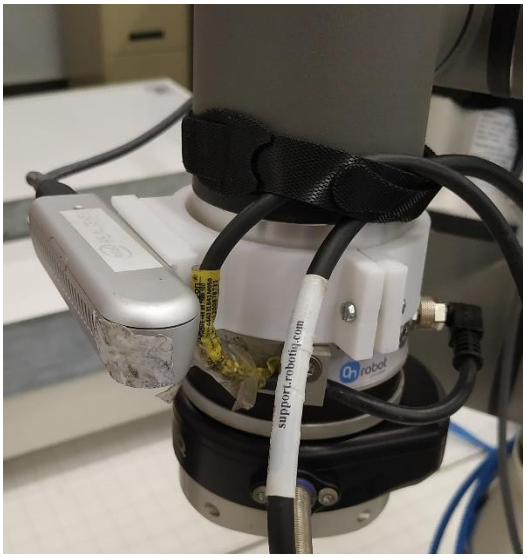


Imagen 31: Diseño final del adaptador en uso.

Este diseño se compone de tres piezas diferenciadas:

- **Soporte inferior:** es el semicírculo que quedaría en la parte contraria a la cámara. Como rasgo importante a destacar es que no es un semicírculo completo, si no que le falta a uno de los lados un poco para completarlo. Esta decisión se ha tomado ya que el mayor defecto de este diseño sería que el adaptador no sujetara con fuerza al brazo robot y se fuera deslizando poco a poco. Con esto, se hace que el cierre con los tornillos añada más fuerza a la sujeción del brazo.

  Como segundo elemento a destacar están los agujeros de diámetro $3.35\ mm$, que se utilizarán para pasar un tornillo de tipo M4, con un juego de $0.05\ mm$. Esto permitirá unir el soporte inferior y el superior mediante un tornillo y una tuerca de tipo M4.

- **Soporte superior:** este es el semicírculo que se encuentra más cerca de la cámara y donde se encuentra el hueco para el tornillo que sujetará la cámara. En él se encuentra la cama sobre los que se apoyarán los distintos tipos de soportes de cámara. Los soportes tienen poco juego respecto al enganche de la cama, por lo que tan solo por presión ya tiene suficiente sujeción. Aunque también se han añadido dos agujeros para tornillos de tipo M2.5, que sujeta el soporte de la cámara a la cama del soporte superior. Todo esto se puede observar en la Imagen 32 y en el apartado de Planos 3:

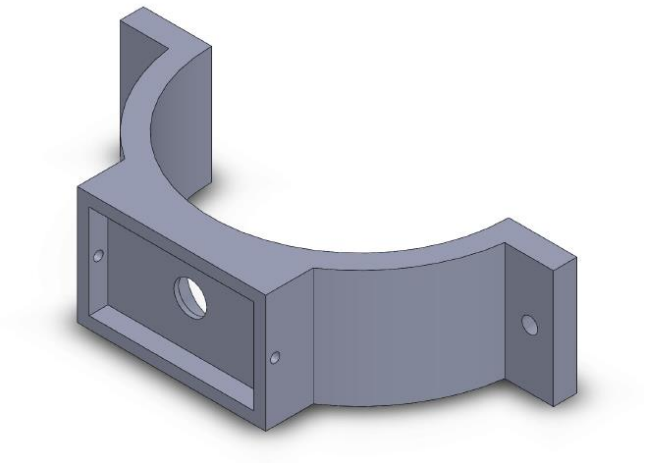


Imagen 32: Soporte superior del adaptador la cámara.

- **Soporte de la cámara:** esta es la pieza que facilitará la sujeción de la cámara. Es fácil de cambiar, por lo que se puede diseñar un soporte distinto para cada tipo de cámara. Esto permite tener mucha flexibilidad, ya que tan solo hay que diseñar una pequeña pieza para poder cambiar la cámara. Además, existe la posibilidad de hacer diseños de soportes genéricos que sean útiles para distintos tipos de cámaras.

  En este caso, se ha realizado un diseño para la cámara RealSense D415, siendo este el resultado obtenido (Imagen 33):

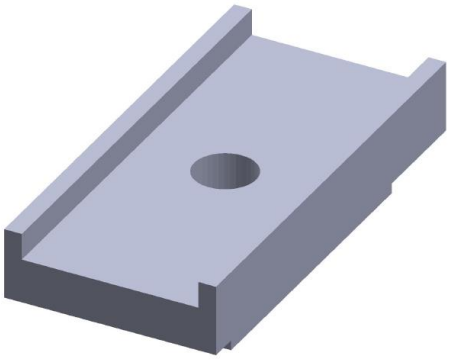


Imagen 33: Soporte para cámaras RealSense

Impresión 3D de las piezas

Para la impresión 3D se debe transformar el fichero *.sldprt* que genera el SolidWorks a uno *.gcode*, que es el que tiene las instrucciones necesarias para la impresión 3D. Para ello se ha utilizado el software *Ultimaker Cura*, con una configuración especializada para la impresora Ender 3.

Respecto a las propiedades de la impresión, se ha optado por una calidad media, con un relleno de pieza del 20 %. Este valor puede parecer pequeño, pero estas piezas no tienen que hacer frente a cargas muy grandes, por lo que ese valor es aceptable para un prototipo. Para un diseño final, se subiría el relleno a un 40 % o 50 %. También se ha optado por utilizar soportes para los agujeros de tornillo, ya que de no utilizarlos se pueden deformar y hacer que no entren los tornillos.

Con estas propiedades se tiene una duración de impresión total de 3 horas y 53 minutos, con un consumo total de PLA de 30 gramos. Esto se distribuye entre las piezas de la siguiente forma:

- **Soporte inferior:** 1 hora y 10 minutos, con un consumo de 10 gramos de PLA.
- **Soporte superior:** 1 hora y 46 minutos, con un consumo de 13 gramos de PLA.
- **Soporte de la cámara:** 57 minutos, con un consumo de 7 gramos de PLA.

Por otro lado, también es importante la posición de la pieza durante la impresión, ya que la orientación de las capas puede aportar mayor resistencia a la tracción o en cambio hacerlo muy poco resistente. Por lo tanto, las impresiones de los soportes superior e inferior, que son los que mayor fuerza deben aguantar, se han impreso tumbadas. Aumentando su resistencia y permitiendo aplicar más fuerza a la sujeción del brazo, como se ha indicado en el apartado de diseño *del soporte inferior*. En la Imagen 34, se muestra la posición en la que se ha impreso las piezas y cómo se disponen las capas de filamento:

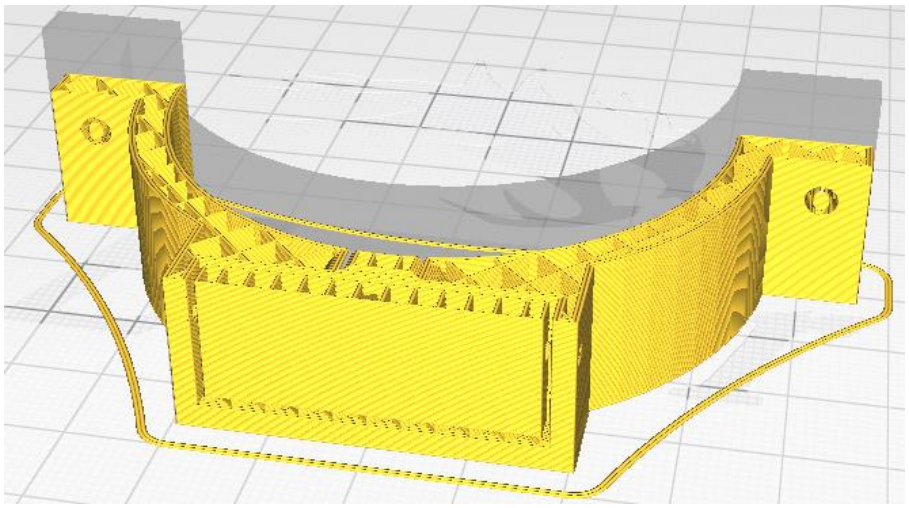


Imagen 34: Ejemplo de posición correcta de impresión para tener mayor resistencia a la tracción.

4 IMPLEMENTACIÓN

La implementación del escáner en 3D se ha realizado de dos formas distintas en este proyecto. La primera de ellas es mediante simuladores y la segunda de ellas es realizando las pruebas con el entorno físico del laboratorio.

La totalidad de proyecto se ha desarrollado en el simulador, ya que no era posible asistir al laboratorio antes debido a las normas de seguridad para la prevención del CoVid. Por lo tanto, no se ha podido ir haciendo comprobaciones del funcionamiento del proyecto durante el desarrollo y tan solo se ha podido comprobar el funcionamiento del programa final.

A continuación, se va a explicar de forma agregada la implementación de la solución de simulación y la solución utilizando el robot UR3, esto quiere decir que se va a explicar todos los apartados de cada solución de forma conjunta. Mientras que, en el apartado de *Desarrollo de la Solución*, se va a explicar en detalle cada uno de los apartados de la solución.

Solución de programación

Tanto para la implementación física, como para la simulada se ha desarrollado los mismos programas. Aquí se va a mostrar el funcionamiento de estos programas a alto nivel, ya que estos se explicarán detalladamente en el apartado de *Desarrollo de la Solución*

El funcionamiento de los programas se puede ver representado en los siguientes diagramas de flujo, Imagen 35. Donde a la izquierda se encuentra el diagrama del programa realizado en Matlab y a la derecha el programa realizado en URscript.

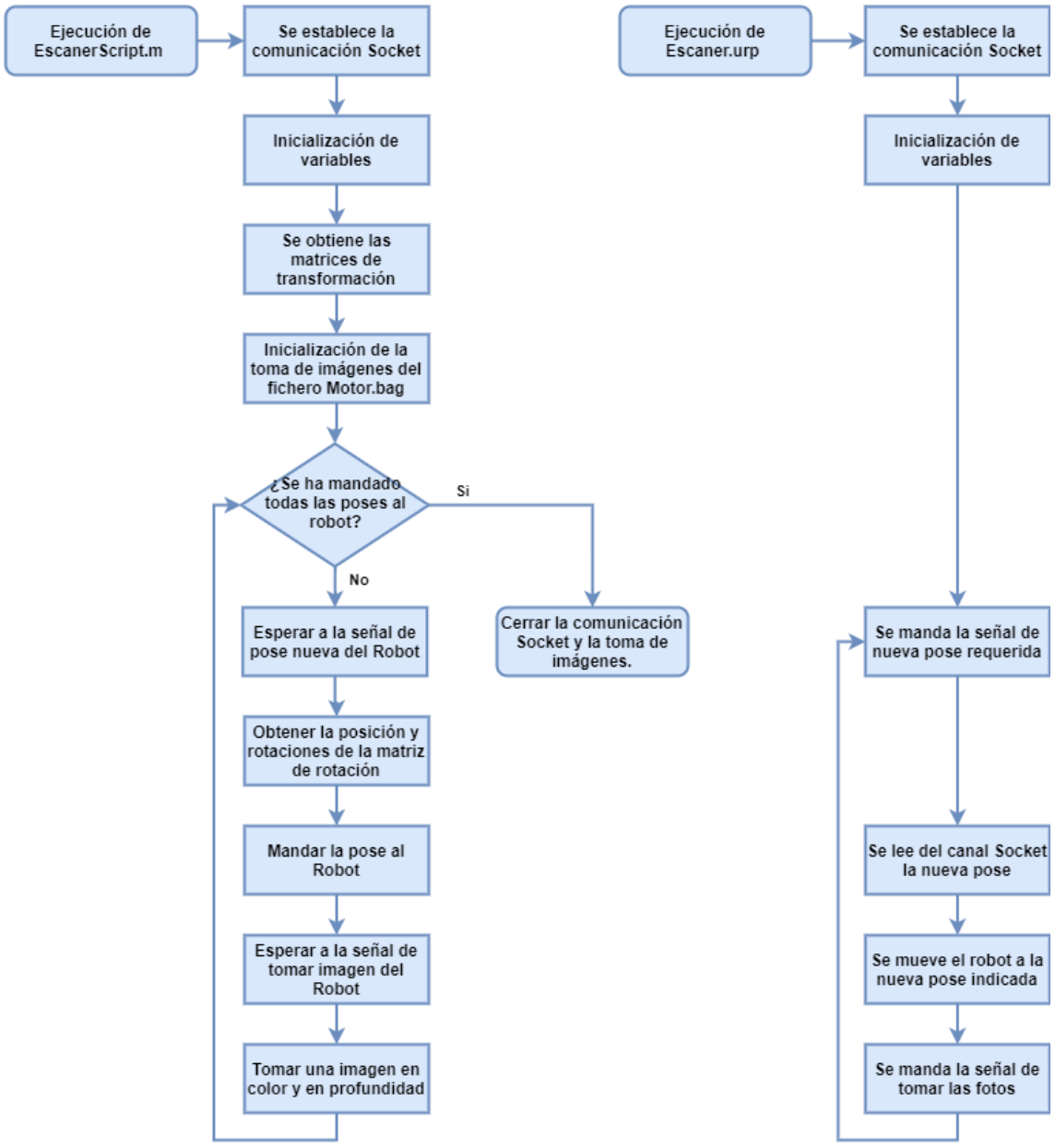


Imagen 35: Diagramas de flujo de los programas desarrollados. Izquierda: script de Matlab. Derecha: Script de URscript

Estos dos programas dependen el uno del otro y por ello es necesario utilizar la comunicación vía socket. Ambos programas se sincronizan con esperas hasta recibir mensajes de continuar con la ejecución. Por ejemplo, la espera que se ve en el diagrama de flujo de Matlab (Imagen 35), en el que se espera a recibir un mensaje del robot de tomar las fotos.

El funcionamiento en conjunto del programa sería el siguiente:

Primero se establece la comunicación socket entre el ordenador y el robot. Después, se inicializan las variables en ambos scripts y se entra en los bucles principales de funcionamiento del programa.

En este bucle primero se manda un mensaje del Robot al ordenador, indicando que es necesaria una nueva pose donde desplazarse. El script de Matlab manda la pose y el Robot se desplaza hasta la posición y orientación indicada. Cuando el robot llega al punto indicado, este manda un mensaje a Matlab de que ya se puede tomar las imágenes. Por último, tras la toma de imágenes, se vuelve al principio del bucle y se repite de nuevo toda la secuencia.

Tras pasar por todas las poses que ha elegido el usuario mediante las funciones *posiciones_escaner* y *pose_puntos,* acaba el bucle de Matlab y se cierra la comunicación con el robot y la cámara.

Implementación por Simulador

Debido a las circunstancias del desarrollo del proyecto este ha tenido que realizarse completamente en un simulador. Para ello, se ha utilizado el software URsim para simular el comportamiento y la programación del brazo robot UR3.

Por otro lado, para simular la toma de imágenes se está leyendo un fichero de imágenes de profundidad de tipo *.bag*. El proceso de lectura utiliza las mismas funciones de la librería de RealSense que las que utilizaría la cámara, por lo que es una decisión razonable para hacer comprobaciones de los programas.

Como resultado final de la implementación por simulador se ha conseguido visualizar el comportamiento del robot a la vez que de la toma de imágenes. Para activarlo, tan solo hay que introducir los valores para restringir la superficie esférica de puntos por donde pasará y ejecutar los dos programas. A continuación, se muestra varios ejemplos del simulador en funcionamiento (Imagen 36):

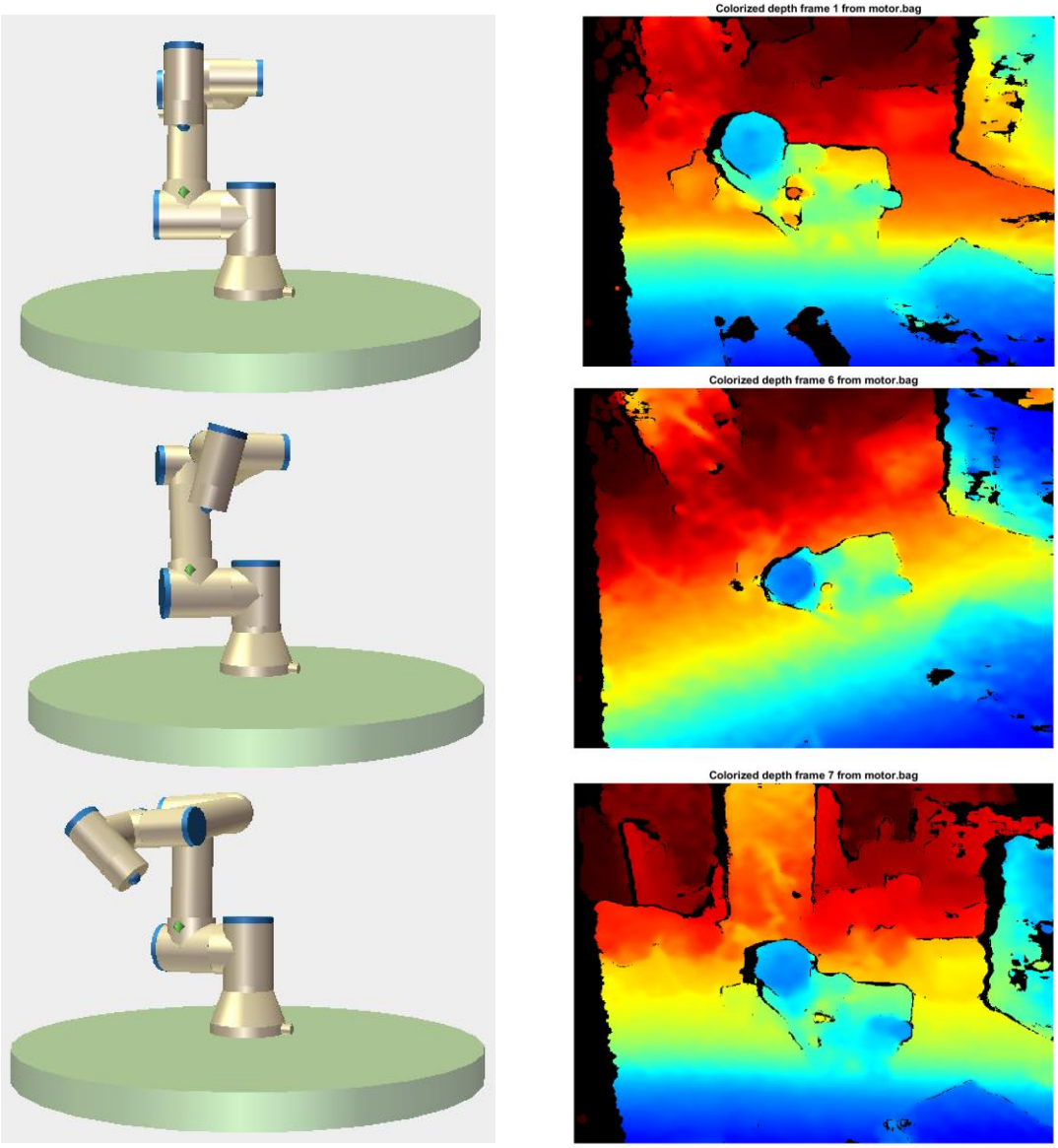


Imagen 36: Ejemplo de resultados obtenidos en el simulador URsim. Los imágenes han sido tomadas del fichero motor.bag

Implementación en el laboratorio

En este apartado se va a explicar cómo se ha implementado la solución adoptada al entorno físico real. Esta implementación se realizó como comprobación final de todo el proyecto en el sistema real.

Para realizar esta implementación en primer lugar se unió la cámara al brazo robot utilizando el adaptador impreso en 3D (Imagen 37). Después, se realizó la conexión de la cámara con el ordenador mediante un cable USB tipo c (Imagen 38).

Imagen 37: Unión de la cámara con el brazo robot.

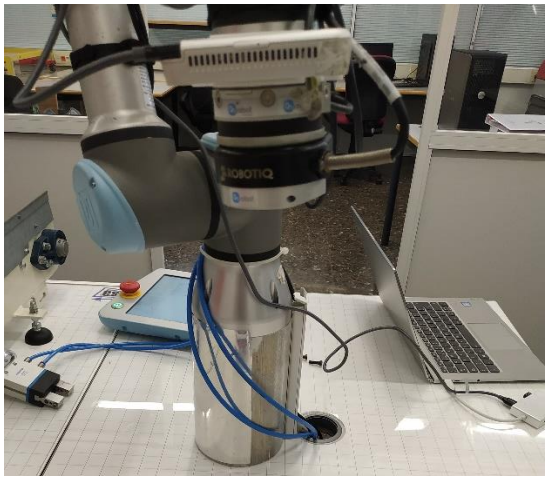


Imagen 38: Conexión de la cámara con el ordenador.

Con la cámara instalada, se pasó a realizar su calibración. Con ello se conocerá la posición de la cámara respecto a la base del brazo robot (Imagen 39).

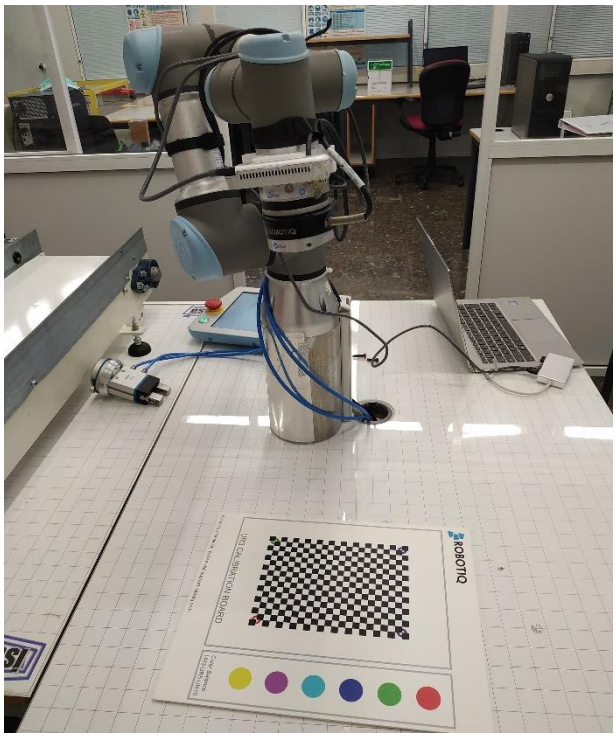


Imagen 39: Calibración de la cámara D415.

Por último, con la cámara ya calibrada, se puede tomar las imágenes del objeto a escanear (Imagen 40). En este caso se tomó como objeto una placa GPU por su gran cantidad de detalles, lo que facilita las reconstrucciones mediente fotogrametría.

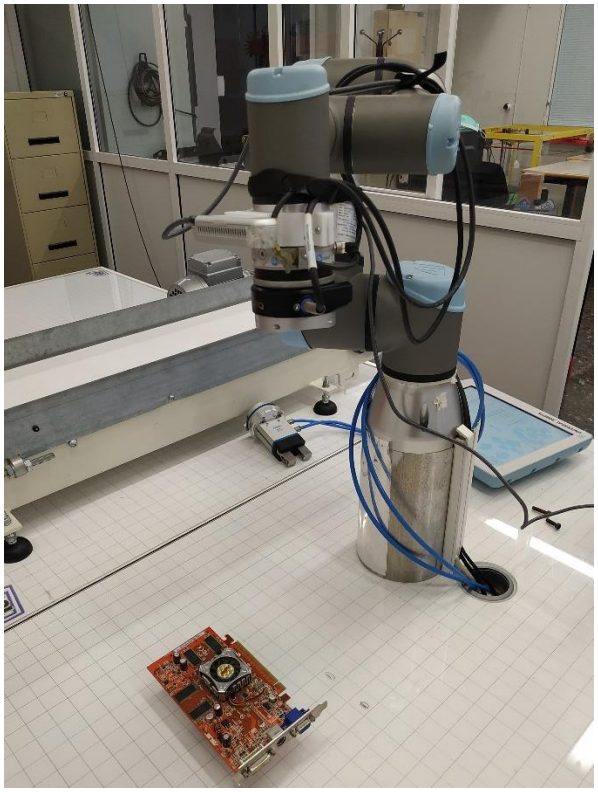


Imagen 40: Toma de imágenes de la placa GPU.

5 PROBLEMAS ENCONTRADOS Y SOLUCIONES ADOPTADAS

Durante el desarrollo de la solución ha aparecido una gran cantidad de problemas, los cuales se han tenido que ir solucionando. En este apartado se encuentran los problemas más destacados y cómo se han solucionado.

5.1 CONEXIÓN VÍA SOCKET CON EL SIMULADOR

Este fue uno de los problemas más relevantes de los que se han encontrado, ya que de no haber encontrado una solución, no se podría haber continuado con el proyecto.

Este error apareció después de probar la comunicación por Socket con el programa *SocketTest.* Como se ha explicado en el apartado 3.3, la conexión mediante este programa fue correcta y por tanto, era posible establecer comunicación Socket con el simulador.

Pero en el siguiente paso es donde surgió el problema, ya que la IP que se utilizó en el *SocketTest* no servía para establecer la conexión Matlab-URsim.

Después de esto, se intentó realizar la conexión con el Simulador como servidor y Matlab como cliente. Haciendo la conexión a la IP de la máquina virtual donde corre el simulador, obtenida con el comando *ifconfig*. Se probó también a cambiar los roles de servidor-cliente, utilizando la IP de address IPv4 del ordenador. Ambos intentos fallaron también.

Solución adoptada

Finalmente, a partir del último intento se llegó a la solución final. En esta solución el ordenador hace de servidor y la máquina virtual con el simulador hace de cliente. Para el servidor se utilizó una configuración por la que acepta conexión desde cualquier IP y para el cliente, se le indicó que estableciera conexión con la IP del ordenador indicada como *Adaptador de Ethernet VirtualBox Host-Only Network* dentro de ese apartado se ha tomado la de *Dirección IPv4,* en la Imagen 41 se muestra cómo aparece esta IP. A ella se puede llegar utilizando el comando *ipconfig* en el *Símbolo de Sistema.*

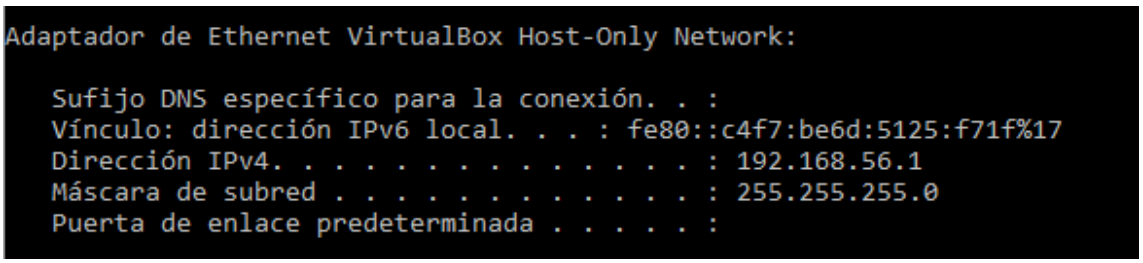


Imagen 41: Ejemplo de dónde se obtiene la dirección IP utilizada para la comunicación con el servidor.

Es importante indicar que esta IP es la que se debe introducir en la función *Socket_open* del script de UR.

Por otro lado, destacar que para llegar a esta solución se tuvo que realizar una gran cantidad de pruebas y se tuvo que investigar en el ámbito de las comunicaciones con máquinas virtuales. Por lo que no fue una solución directa ni fácil de obtener.

5.2 ALCANZAR PUNTOS SINGULARES Y EXTREMOS DEL UR3

Este problema apareció cuando se empezó a hacer pruebas con el programa final. Se observó en el simulador que a la hora de recorrer una superficie esférica de puntos con el brazo robot era muy fácil llegar a puntos singulares o extremos.

Esto es un problema, ya que para obtener un escaneado fiel mediante fotogrametría es importante imágenes desde todos los puntos de vista del objeto. Y de quitar puntos, se estaría quitando fiabilidad a la reconstrucción del objeto.

Solución adoptada

El primer paso para evitar estos puntos extremos fue conocer cuál es el límite de la escala de la esfera donde siempre se alcanza puntos extremos. Para ello, se realizó distintas pruebas y se estableció que el valor máximo para el parámetro de la escala sería de 0.1, que corresponde con una esfera de radio $10\ cm$. Para este valor aun así se alcanza valores extremos, pero son en mucha menor cantidad que para valores mayores de escala.

El siguiente paso fue desplazar el punto donde se colocará el objeto a escanear. Este se deberá colocar a una distancia de como mínimo en valor de la escala más 0.1. Para así dejar el espacio suficiente entre el punto más cercano al robot y el robot.

Por último, si continúan apareciendo puntos extremos, se deberá aplicar un cambio en la estrategia de escaneo. Para ello, se ha pensado tres soluciones distintas:

- **Restringir el azimut:** esta alternativa consiste en no tomar imágenes en los 360º del objeto. Se quitará los puntos singulares del robot, haciendo que ya no tenga que pasar por ahí. El punto fuerte de esta solución es que es muy fácil de implementar, tan solo hay que modificar los parámetros de la función *posiciones_escaner*, pero por otro lado, se están perdiendo imágenes para la composición de la imagen 3D.

- **Utilización de una cama rotatoria:** en esta alternativa se utiliza un elemento adicional, que es una cama rotatoria (Imagen 42). Esta hará una pequeña rotación cada vez que se tome una imagen. Para este caso, el robot debe pasar por los mismos puntos, cambiando la elevación y no el azimut, mientras que la cama va rotando el objeto. Esta es una solución muy precisa para la reconstrucción en 3D del objeto, pero es necesario añadir un elemento adicional al sistema.

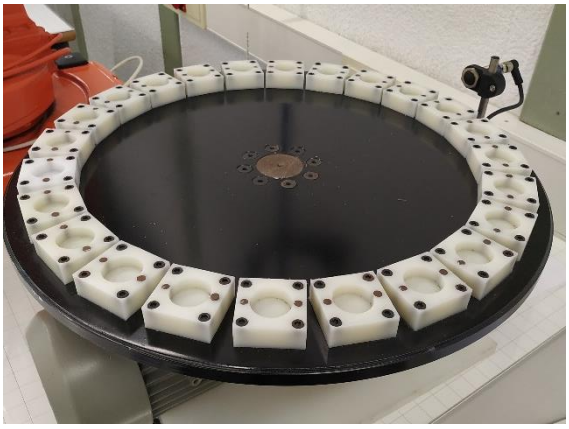


Imagen 42: Cama rotatoria instalada en el laboratorio.

- **Dos tomas de imágenes moviendo el objeto:** esta alternativa consiste en realizar dos tandas de fotos. En una de ellas el objeto estará más alejado y se tomará el intervalo de azimut de 180º más cercano al robot (Imagen 43). En la otra toma de imágenes el objeto estará más cerca y se tomará el intervalo de azimut de 180º más alejado del robot (Imagen 44). Esta estrategia hace que el robot tenga mucho más espacio para cada posición.
  Con esta solución se obtiene una reconstrucción mejor que eliminando los puntos que den problemas. Pero existe el riesgo de no realizar un desplazamiento muy exacto del objeto y por tanto perder calidad en la reconstrucción del objeto 3D.

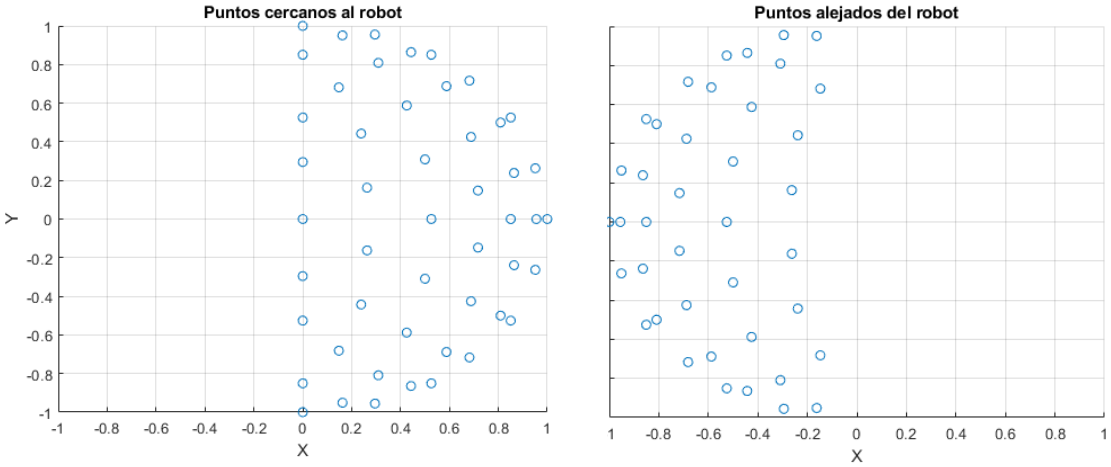


Imagen 43 (izquierda) e Imagen 44 (derecha): Ejemplos de la distribución de los puntos para la solución mediante dos tomas de imágenes.

Siguiendo los tres pasos mencionados en este apartado se apartado se puede realizar una toma de imágenes correcta evitando pasar por puntos problemáticos para el brazo robot. Se puede elegir para cada caso la solución que más convenga, dependiendo si se quiere más calidad en la composición del modelo 3D.

5.3 CORRECCIÓN DE LAS TRAYECTORIAS QUE SIGUE EL UR3

Este es un problema similar al mencionado anteriormente. En este caso el problema está en la trayectoria que toma el brazo robot, ya que hay momentos en los que chocaría contra el suelo o contra el objeto a escanear.

Con la solución aportada en el problema anterior se ve muy reducida la cantidad de trayectorias erróneas. Pero aun así siguen apareciendo una gran cantidad de ellas en las que el brazo robot chocaría contra el suelo.

Solución aportada

Para evitar la aparición de este problema se ha desarrollado las siguientes soluciones:

- **Seleccionar la solución de la cinemática inversa:** en las primeras versiones del programa se dejaba que el simulador tomara una solución de la cinemática inversa por defecto. Esto producía que en muchos puntos se tomara una solución en la que el codo del brazo robot esté hacia abajo, lo que producía que en ciertos tramos de la trayectoria se atravesara el suelo, tal y como se ha explicado en el apartado 3.2.
  Por ello, se ha implementado la selección de la solución de la cinemática inversa, tomando siempre la solución en la que el codo del robot esté hacia arriba.
- **Colocar el objeto a escanear en alto:** esta solución consiste en colocar el objeto a escanear en un soporte de $10\ cm$, lo que hace que el robot no tenga que ir a posiciones tan cercanas al suelo.

Si se utiliza estas dos propuestas, el robot no pasará en ningún momento por tramos que no permitan el correcto funcionamiento del sistema. Además, si se juntan las soluciones del apartado anterior con estas, se podrá realizar un escaneado completo y correcto.

5.4 VALORES DE ROTACIONES ACEPTADOS EN LA POSE EN EL UR3

Otro problema destacado apareció a la hora de mandar las rotaciones del script de Matlab al brazo robot. Este problema apareció ya que al principio se le mandaba al robot el valor de la rotación en X-Y-Z que se obtenía directamente de la función *pose_puntos*, se tomaba esas rotaciones pues eran con las que se formaba la matriz de rotación del resultado final.

Mandando esos valores de rotación se obtenía siempre orientaciones del efector final incorrectas.

Tras esto se desarrolló una función que a partir de cada matriz de transformación se obtuviera el valor de rotación en X-Y-Z. Con esta función se obtenían otros valores de rotaciones, pero al probarlo en el URSim la orientación del efector final era siempre la incorrecta.

Solución aportada

Después de probar varias modificaciones de la función desarrollada en el intento anterior se desechó esa idea. Con ello, se volvió a estudiar la composición de rotaciones y por último se volvió a repasar el manual de URscript.

Del manual de URscript se obtuvo que las rotaciones dentro de las variables de tipo *pose* deben estar en formato *axis-angle.*

Se estudió a fondo este tipo de formato de rotaciones. Encontrando una función de Matlab que realizaba la conversión de matriz de rotación a rotación en X-Y-Z en *axis-angle.* Esta función es *rotm2axang* y al probarla se obtuvo ya las orientaciones adecuadas en el efector final para todos los puntos de la superficie esférica.

6 RESULTADOS

En este apartado se van a analizar y valorar los resultados obtenidos tras realizar las comprobaciones finales de la solución adoptada.

Se separará en el resultado obtenido de las funciones de generación de posiciones, el resultado global obtenido en el simulador y el resultado obtenido en la implementación física.

6.1 RESULTADO DE LAS POSICIONES GENERADAS EN MATLAB

Respecto a las funciones de generación de poses cabe destacar que se ha conseguido satisfacer todas las especificaciones. Mediante la función *posiciones_escaner* se ha conseguido tomar la superficie de puntos necesaria en cada caso para realizar un escaneo correcto. Para después, con la función *pose_puntos* se ha conseguido obtener la orientación correcta para cada punto y guardar todos los datos en distintas matrices de transformación.

Por otro lado, la selección de los puntos aún es un poco rígida, ya que tan solo se puede tomar un intervalo en la elevación y azimut. Y de querer elegir dos subintervalos (por ejemplo, azimut de [0º, 90º] y de [180º, -90º]), se debería llamar a la función dos veces y juntar los puntos obtenidos para cada intervalo en una variable, por lo que es importante no tener los mismos puntos ya que el robot pasaría dos veces por el mismo punto. Esto se muestra en el siguiente extracto de código y se ve el resultado en la Imagen 46:

```
%Obtención de dos subintervalos de azimut.
E1 = posiciones_escaner([0,0,0],65,[0,90],[0,90]);
E2 = posiciones_escaner([0,0,0],65,[0,90],[180,-90]);
E = [E1;E2]
```

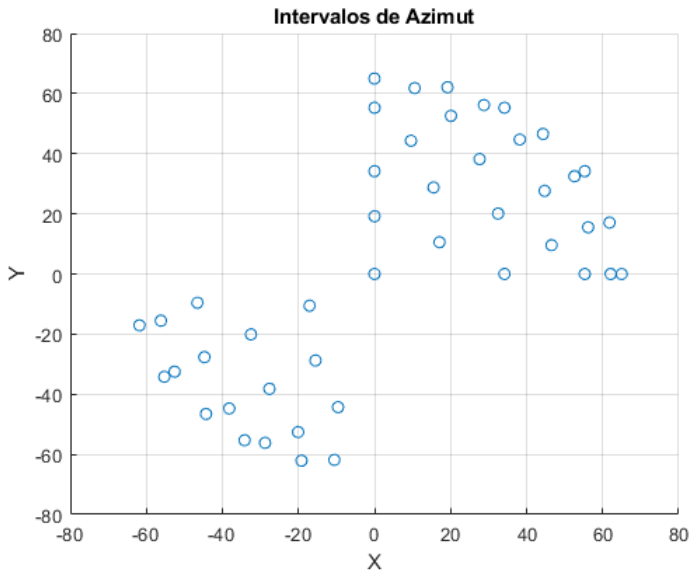


Imagen 46: Ejemplo del resultado obtenido tomando dos intervalos de azimut.

Este problema que se menciona es importante, ya que hay que evitar los puntos donde el robot no puede llegar y estos están siempre por la zona más alejada del robot y a más cercana, por lo que hay que hacer dos intervalos de azimut.

Por lo tanto, se ha conseguido cumplir las especificaciones para la resolución del problema, aunque en siguientes versiones del proyecto se hará menos rígida la función *posiciones_escaner*.

6.2 RESULTADOS OBTENIDOS EN EL SIMULADOR DEL UR3

Este apartado hace referencia a los resultados obtenidos en la implementación de toda la programación en el sistema del simulador.

Se ha conseguido cumplir el objetivo de funcionamiento y se obtiene una imagen para cada pose del robot.

Se ha realizado una gran cantidad de pruebas con distintas localizaciones del objeto a fotografiar. Con esto, se ha llegado a las siguientes conclusiones:

- Para superficies esféricas de 10 cm de radio no es posible realizar una toma de imágenes tomando los 360º del azimut. Esto se debe a que se llega a posiciones en las que el robot atravesaría el objeto o atravesaría su propio codo. Por lo tanto, es necesario restringir el azimut en las zonas más conflictivas tal y como se ha mencionado en el apartado anterior.
- Es recomendable colocar el objeto a fotografiar a una altura de entre $10\ cm$ y $20\ cm$ del suelo. Esto se hace para evitar que el robot pase por posiciones muy cercanas al suelo, lo que puede llevar a problemas.
- Necesidad de utilizar dos posiciones distintas del objeto para realizar un escaneo de 360º. U otra alternativa sería utilizar una cama rotatoria para el objeto. Tal y como se ha explicado en el apartado de *Problemas encontrados y soluciones adoptadas.*

Estas conclusiones han sido sacadas del simulador, el cual ha permitido hacer comprobaciones de posiciones más drásticas sin poner en riesgo el material real. Aunque, es necesario también, asegurarse de que el funcionamiento es el correcto en el entorno real, ya que pueden llegar a aparecer nuevos errores distintos a los de la simulación.

Por lo tanto, gracias al simulador, se ha podido comprobar que la funcionalidad del sistema es la correcta. Además de haber podido analizar los resultados y obtener la forma de funcionamiento más adecuada para el sistema.

6.3 RESULTADOS DE LA IMPLEMENTACIÓN FÍSICA

Después de la implementación del sistema en el laboratorio se tomaron una serie de imágenes a una placa GPU para posteriormente generar un modelo 3D de la placa. Estas son una muestra de las imágenes que se utilizaron para la composición (Imagen 47):

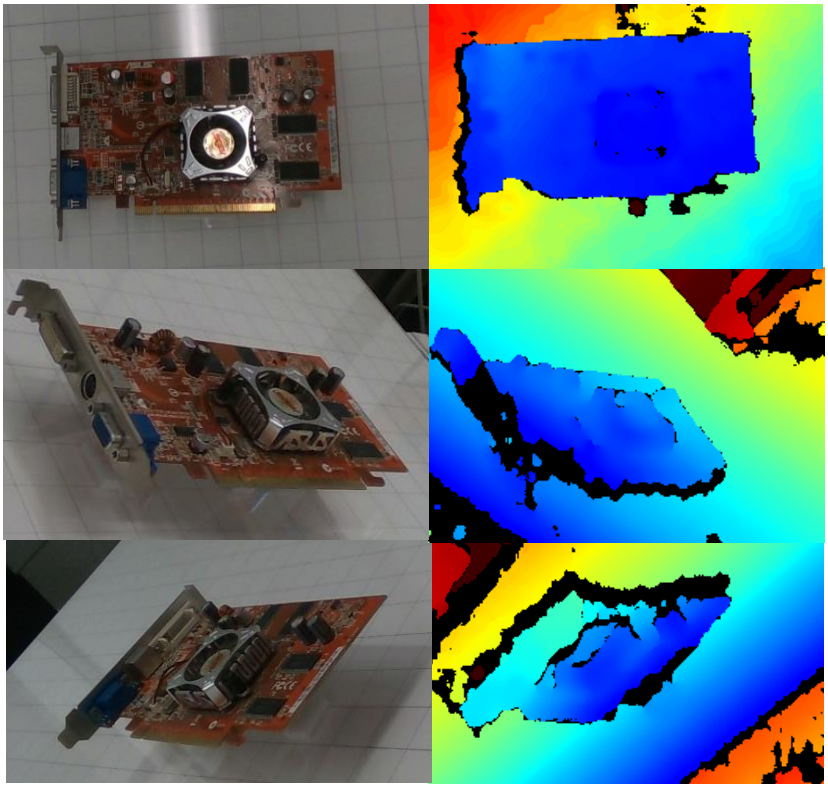


Imagen 47: Imágenes de la GPU tomadas desde el brazo robot UR3.

Tras la toma de imágenes, se utilizó el software *Agisoft* (Referencia 6) para generar un modelo 3D de la placa GPU. Este software tiene ya implementados algoritmos para el reconocimiento de la posición de la cámara a partir de las imágenes. Este algoritmo es útil para casos en los que haya referencias claras en las imágenes, siendo la placa GPU perfecta para facilitar el funcionamiento del algoritmo. El resultado obtenido de utilizar el software *Agisoft* con las imágenes tomadas es el siguiente (Imagen 48):

Imagen 48: Resultado del escaneado 3D de la placa GPU.

Como se puede observar en la Imagen 45 el resultado es muy fiel a la realidad, pudiendo representar pequeños detalles como son las resisencias smd. Por otro lado, en la reconstrucción también hay fallos e imprecisiones. Esto se puede deber a una falta de imágenes o a fallos relacionados con el algoritmo de reconocimiento de posición de la cámara.

Este problema se podría solucionar desarrollando un sistema de optimización de la reconstrucción del modelo. Para ello, se debería utilizar los datos de la posición y orientación exactos de la cámara en cada imagen, siendo datos conocidos a partir de las posiciones del efector final y las medidas del adaptador de la cámara.

7 CONCLUSIONES

A pesar de todas las dificultades se ha conseguido resolver el problema de forma eficiente. Se ha conseguido desarrollar un sistema de toma de imágenes que cumple con todas las especificaciones necesarias.

Realizar el proyecto basándose en el simulador ha sido todo un éxito. Este ha permitido continuar con el desarrollo de la solución a pesar de no poder utilizar el sistema real.

A partir de las observaciones tomadas de los resultados con el simulador, se ha llegado a las siguientes conclusiones:

- Antes de realiza un escaneado es necesario realizar pruebas con el simulador para comprobar que todos los puntos por los que pasará el robot son correctos y no hay ninguna trayectoria problemática.
- Para ciertos tamaños de objetos será necesario utilizar estrategias un poco más complejas para realizar un escaneado correcto. Estas estrategias se han explicado en el apartado 5.2.

Respecto a los costes de proyecto, cabe destacar que el mayor coste corresponde con el robot colaborativo UR3. Utilizar un robot colaborativo aporta muchas ventajas, sobretodo de seguridad material y humana. Pero de querer realizar una implementación similar de menor coste, se podría utilizar un brazo robot no colaborativo. Siempre y cuando se incrementen los sistemas de seguridad tanto para personas como para el entorno del robot.

Por último, comentar que los siguientes pasos a seguir después de este proyecto sería mejorar los resultados obtenidos con los programas de fotogrametría aplicando un sistema de optimización para la obtención de los modelos 3D. Esto se puede realizar gracias a que se puede saber con seguridad la posición desde la que se toma cada imagen.

8 REFERENCIAS

1. Entrada de Stack Overflow sobre las bases de la programación Socket: https://stackoverflow.com/questions/36520590/basic-flow-of-control-in-socket-programming

2. Manual de URscript: https://s3-eu-west-1.amazonaws.com/ur-support-site/18679/scriptmanual_en.pdf

3. Manual de usuario UR3: https://s3-eu-west-1.amazonaws.com/ur-support-site/69073/99436_UR3e_User_Manual_es_Global.pdf

4. Datasheet de las cámaras de la serie D400 de Intel: https://www.mouser.com/pdfdocs/Intel_D400_Series_Datasheet.pdf

5. Página de GitHub del wrapper de Matlab para las cámaras de Intel, Real Sense: https://github.com/IntelRealSense/librealsense/tree/master/wrappers/matlab

6. Página del software de fotogrametría: https://www.agisoft.com/

PRESUPUESTO

1 CONTENIDO

2 PRESUPUESTO

En este apartado se van a registrar todos los gastos asociados a este proyecto para después poder calcular el coste total del proyecto.

En primer lugar se mostrará una tabla de los costes de los materiales (Tabla 1). Después se mostrará una tabla con los costes de la mano de obra (Tabla 2). Y finalmente se mostrará los costes totales antes (Tabla 3) y después de impuestos (Tabla 4).

Costes de materiales para la instalación de un escáner 3D			
Cantidad	**Objeto**	**Precio unitario**	**Precio total**
1 ud	Brazo robot UR3	19.750,0 €	19.750,0 €
1 ud	Cámara D415	131,93 €	131,93 €
1 ud	Ordenador portátil ASUS Intel Core i5-6200U, 8 GB, 1TB, 920m, 15.6”	541,81 €	541,81 €
1 ud	Cable USB 3.0 tipo c, 2 metros	6,79 €	6,79 €
1 ud	Bobina 1 kg PLA para impresión 3D	19,99 €	19.99 €
2 ud	Tornillo M4	0,75 €	1,5 €
1 ud	Tornillo ¼ pulgada rosca Whithworth	0,75 €	0,75 €
-	TOTAL	-	19.904,14 €

Tabla 1: Gastos en materiales. Enlaces a las páginas de compra en Referencias..

Costes de personal			
Horas	**Personal**	**Precio a la hora**	**Precio total**
2 h	Técnico en diseño industrial	25,0 €	50,0 €
4 h	Técnico de impresión 3D	15,0 €	60,0 €
20 h	Ingeniero técnico industrial especializado en robótica	30,0 €	600,0 €
3 h	Técnico de instalaciones de brazos robot	20,0 €	60 €
-	TOTAL	-	770 €

Tabla 2: Gastos asociados al personal del proyecto.

Coste antes de impuestos		
Concepto	**Cantidad**	**Precio total**
Gastos en materiales	-	19.904,14 €
Gastos en personal	-	770 €
Beneficio industrial	6 %	1.240,14 €
TOTAL (antes de impuestos)	-	21.914,31 €

Tabla 3: Gastos antes de impuestos.

Coste después de impuestos		
Concepto	**Cantidad**	**Precio total**
Gastos antes de impuestos	-	21.914,31 €
IVA	21 %	4.602,01 €
TOTAL	-	26.516,31 €

Tabla 4: Gastos despúes de impuestos

El coste total del proyecto después de impuestos asciende a veintiséis mil quinientos dieciséis euros y treinta y un céntimos (26.516,31 €).

3 REFERENCIAS

- **UR3:**
  https://cobots.se/produkt/ur3-robot/

- **Cámara D415 de RealSense:**
  https://store.intelrealsense.com/buy-intel-realsense-depth-camera-d415.html

- **Bobina de PLA para impression 3D:**
  https://www.pccomponentes.com/sakata-3d-bobina-de-filamento-pla-3d850-175mm-negro-1kg

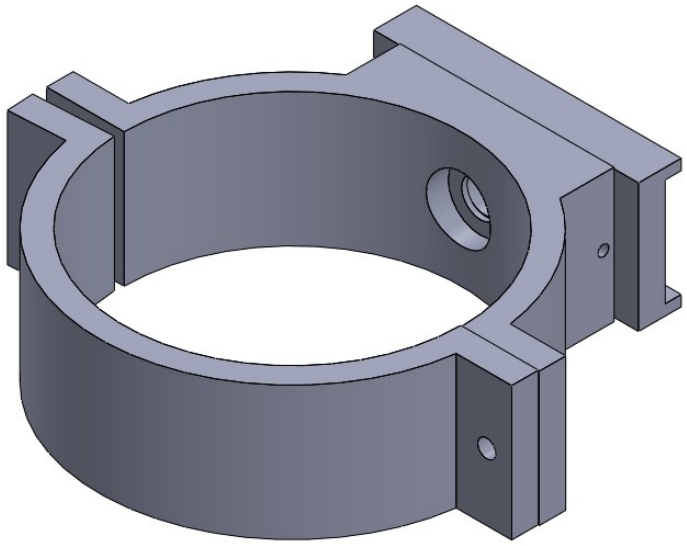
- **Ordenador portátil ASUS:**
  https://www.pccomponentes.com/asus-f556uj-xo009t-intel-core-i5-6200u-8gb-1tb-gt-920m-156

- **Cable USB tipo c, 2 metros:**
  https://www.amazon.es/AUKEY-Sincronizaci%C3%B3n-Samsung-Galaxy-MacBook/dp/B01ISLQZ2E

- **Tornillos:**
  Comprados en la Ferreteria-Cerrajeria Collado (Valencia).

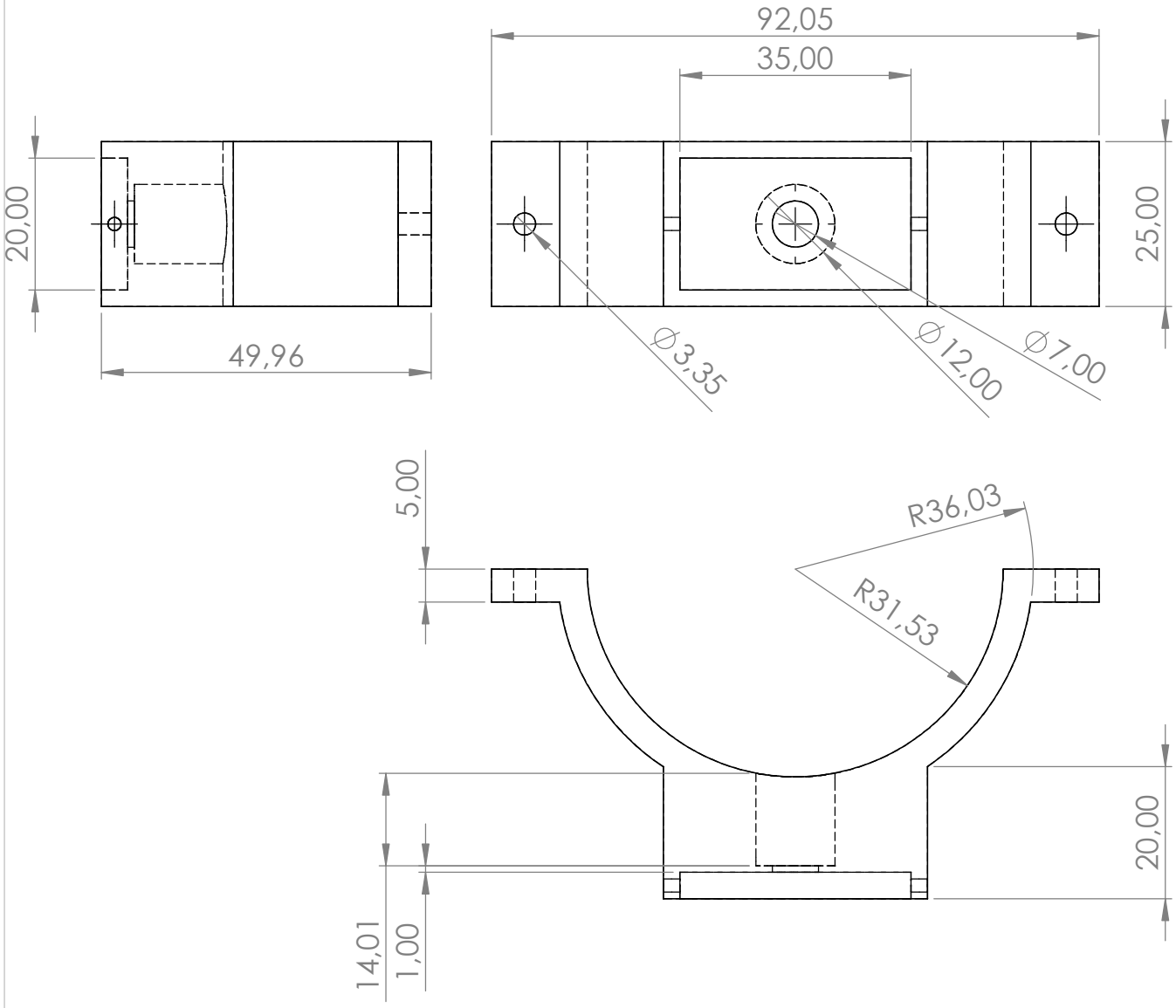
PLANOS

1 CONTENIDO



Proyecto:		**Fecha:** 28/06/20
Adaptador de la cámara a UR3		**Escala** 1:1
Autor: Eduardo Santos Navarro	**Plano:** Ensamlado del soporte de la cámara	**Plano Nº:** 001

Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.



Proyecto:

Adaptador de la cámara a UR3

Fecha: 28/06/20

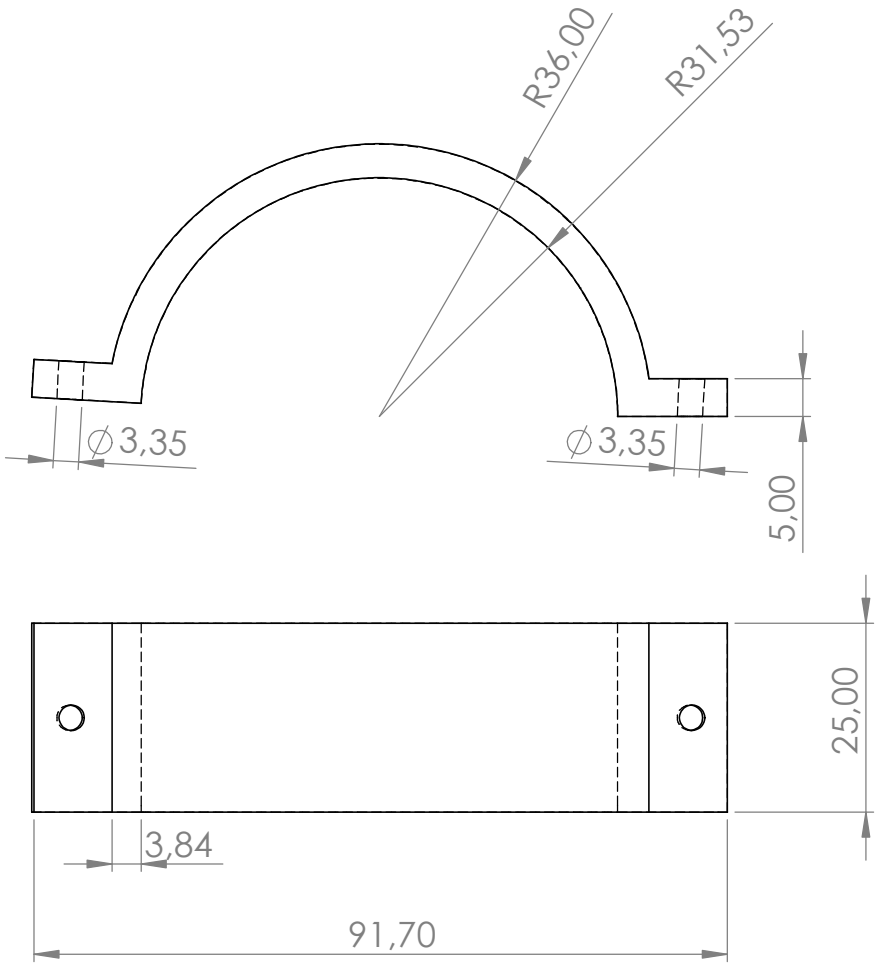
Escala
1:1

Autor:
Eduardo
Santos Navarro

Plano:
Vistas del soporte superior del adaptador de la cámara

Plano Nº:
002

Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.



Proyecto:

Adaptador de la cámara a UR3

Fecha: 28/06/20

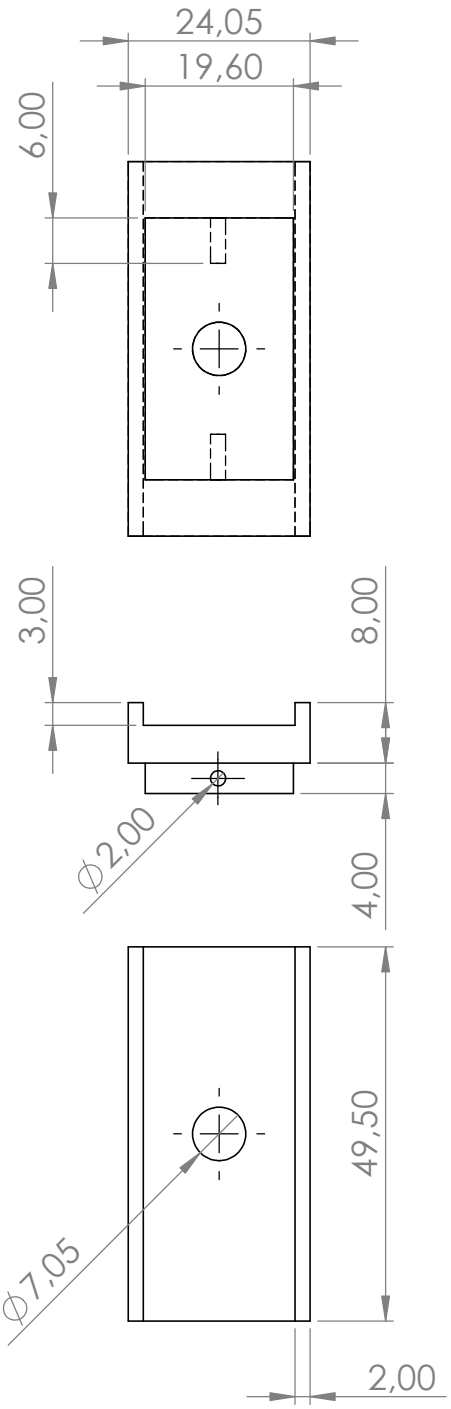
Escala
1:1

Autor:
Eduardo
Santos Navarro

Plano:
Vistas del soporte inferior del adaptador de la cámara

Plano Nº:
003

Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.



Proyecto:		**Fecha:** 28/06/20
Adaptador de la cámara a UR3		**Escala** 1:1
Autor: Eduardo Santos Navarro	**Plano:** Vistas del soporte para la cámara D415 de RealSense	**Plano Nº:** 004

Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.

ANEXOS

1 CONTENIDO

2 PROGRAMACIÓN EN MATLAB

2.1 ESCALA_ESFERA.M

```
function [E] = escala_esfera(escala,esfera)
%En esta función se cambia la escala de la esfera de puntos.
E = escala*esfera;
end
```

2.2 DESPLAZAMIENTO_ESFERA

```
function [E] = desplazamiento_esfera(desp,esfera)
%Esta función se encarga de desplazar el centro de la esfera.
E = esfera + desp;
end
```

2.3 ELEVACION_ESFERA.M

```
function [E] = elevacion_esfera(elevacion,escala,E)
%En esta función se eliminan los puntos de la esfera que no cumplan el
%criterio de la elevación indicada.
%Se ha de introducir primero el ángulo menor y después el mayor, con
%valores entre -90 y 90.
%elevacion = array(1x2).
%escala = valor de la escala.
%E = array de puntos.(:x3), siendo el orden X,Y,Z.



%Se calcula el valor mínimo y máximo de altura, para ello es necesario
el
%valor de la escala.
alt_max = sin((elevacion(1,2)*pi)/180)*escala;
alt_min = sin((elevacion(1,1)*pi)/180)*escala;

%Se eliminan los puntos de la esfera que estén fuera del rango entre
el
%mínimo y el máximo.
for i = length(E):-1:1

    if(E(i,3) < alt_min || E(i,3) > alt_max)
       E(i,:) = [];
    end
end

end
```

2.4 AZIMUT_ESFERA.M

```
function [E] = azimut_esfera(azimut,E)
%En esta función se eliminan los puntos de la esfera que no cumplan el
%criterio del azimut indicado.
%Se tomará el área recogida desde el primer ángulo que se introduzca
al
%segundo y los valores deben encontrarse entre -180 y 180.
%De escoger el mismo valor para las dos entradas se tomará el azimut
%completo.
%azimut = array(1x2).
%E = array de puntos.(:x3), siendo el orden X,Y,Z.

azimut1 = azimut(1,1)*pi/180;
azimut2 = azimut(1,2)*pi/180;

%Se comprueba en que caso del orden de azimuts estamos.
if(azimut1 < azimut2)
    caso = 'Menor_mayor';
elseif(azimut1 == azimut2)
    caso = 'Azimut completo';
else
    caso = 'Mayor_menor';
end

%Se eliminan los puntos de la esfera correspondientes.
if(strcmp(caso,'Azimut completo') == 0)
    for i = length(E):-1:1

        switch caso
            case 'Menor_mayor'
                if(atan2(E(i,2),E(i,1)) < azimut1 ||
atan2(E(i,2),E(i,1)) > azimut2)

                    E(i,:) = [];
                end
            case 'Mayor_menor'
                if(atan2(E(i,2),E(i,1)) > azimut1 ||
atan2(E(i,2),E(i,1)) < azimut2)

                else
                    E(i,:) = [];

                end
        end
    end
end

end
```

2.5 POSICIONES_ESCANER.M

```
function [E] = posiciones_escaner(desp,escala,elevacion,azimut)
%En esta función se obtiene los puntos de una superficie esférica
cumpliendo unos
%parámetros de desplazamiento escala, elevación y azimut.
%desp       = [X,Y,Z]
%escala     = valor de la escala.    (Valores > 0)
%elevacion  = array(1x2).            (Entre -90 y 90. 1º Menor 2º
Mayor)
%azimut     = array(1x2).            (Entre -180 y 180).

%Se leen los puntos de la esfera.
E = csvread('sphere2.csv');

%Se aplica la escala.
E = escala_esfera(escala,E);

%Se aplica el criterio de la elevación.
E = elevacion_esfera(elevacion,escala,E);

%Se aplica el criterio del azimut.
E = azimut_esfera(azimut,E);

%Se aplica el desplazamiento para centrarlo en el objeto.
E = desplazamiento_esfera(desp,E);
end
```

2.6 ANGULOSROT2.M

```
function [Angulos] = AngulosRot2(A)
        F = @sist;
        x0 = [0,0,0];
        Angulos = fsolve(F,x0);
   %Se utiliza la última fila, ya que es la que se corresponde con el
eje Z
   %del sistema móvil. Del que se puede obtener sus componentes
   %(A(1),A(2),A(3)) con la posición del origen de ambos sistemas de
   %coordenadas (ya que apunta del orgigen del sistema móvil al origen
del
   %sistema inicial).
    function F = sist(x)

    F(1) =(-sin(x(1)))*cos(x(2))-A(1);
    F(2) = (sin(x(1))*sin(x(2))*cos(x(3)))+(cos(x(1))*sin(x(3)))-A(2);
    F(3) = (cos(x(1))*cos(x(3)))-(sin(x(1))*sin(x(2))*sin(x(3)))-A(3);

    end
end
```

2.8 POSE_PUNTOS.M

```
function [ Cell,Posicion, Rotacion] = pose_puntos(E,desp)
%Esta función guarda en un cell array todas las matrices de
transformación
%a partir de una matriz de puntos.

%Se procesa cada punto en una iteración.
Cell(1) = { [-1  0  0 E(1,1);
             0 -1  0 E(1,2);
             0  0 -1 E(1,3);]};

for i = 2:length(E)
    Pos =[E(i,1);E(i,2);E(i,3)];
    modulo = sqrt((E(i,1)-desp(1))^2+(E(i,2)-desp(2))^2+(E(i,3)-
desp(3))^2);
    vector_uni =[ (desp(1)-E(i,1))/modulo; (desp(2)-E(i,2))/modulo;
(desp(3)-E(i,3))/modulo];

    %A partir del vector unitario del punto se calcula los ángulos de
las
    %rotaciones.
    [Ang] = AngulosRot2(vector_uni);
    Rot = [
           (cos(Ang(1))*cos(Ang(2))),...
           (sin(Ang(1))*sin(Ang(3)))-
(cos(Ang(1))*sin(Ang(2))*cos(Ang(3))),...

cos(Ang(1))*sin(Ang(2))*sin(Ang(3))+sin(Ang(1))*cos(Ang(3));

            sin(Ang(2)),...
            cos(Ang(2))*cos(Ang(3)),...
           (-cos(Ang(2)))*sin(Ang(3));

           (-sin(Ang(1)))*cos(Ang(2)),...

(sin(Ang(1))*sin(Ang(2))*cos(Ang(3)))+(cos(Ang(1))*sin(Ang(3))),...
           (cos(Ang(1))*cos(Ang(3)))-
(sin(Ang(1))*sin(Ang(2))*sin(Ang(3)))
            ];
    %Guarda la matriz de transformación, que incluye la matriz de
rotación
    %y la posición del punto.
    Cell(i) = {[   Rot Pos
                   0 0 0 1]};
    Rotacion(i,:) = Ang;
    Posicion(i,:) = Pos';


end
end
```

2.9 PREPARACION_CAMARA.M

```
function [colorizer,pipe,name] = preparacion_camara()
%Esta función prepara la configuración necesaria para tomar imágenes
de una
%cámara.
    % Make Pipeline object to manage streaming
    pipe = realsense.pipeline();
    % Make Colorizer object to prettify depth output
    colorizer = realsense.colorizer();

    % Start streaming from the rosbag with default settings
 %   profile = pipe.start(cfg);
     profile = pipe.start();

    % Get streaming device's name
    dev = profile.get_device();
    name = dev.get_info(realsense.camera_info.name);

    % Get frames. We discard the first couple to allow
    % the camera time to settle
    for i = 1:5
        fs = pipe.wait_for_frames();
    end
end
```

2.10 FOTO_CAMARA.M

```
function foto_camara(name,pipe,colorizer,cont)
%Esta función se encarga de tomar las imágenes de profundidad y color
%cuando es llamada.

    try
        fs = pipe.wait_for_frames();
    catch
        disp('Error taking a new frame');
    end
    try
        deph_img = figure;
        % Select depth frame
        depth = fs.get_depth_frame();
        % Colorize depth frame
        color = colorizer.colorize(depth);

        % Get actual data and convert into a format imshow can use
        % (Color data arrives as [R, G, B, R, G, B, ...] vector)
        data = color.get_data();
        img =
permute(reshape(data',[3,color.get_width(),color.get_height()]),[3 2
1]);

        % Display image
        imshow(img);
        title(sprintf("Colorized depth frame %i from %s",cont, name));
        if  i<10
                file_name = sprintf("Fotos/DephImage000%i.png",cont);
        elseif i<100
                file_name = sprintf("Fotos/DephImage00%i.png",cont);
        else
                file_name = sprintf("Fotos/DephImage0%i.png",cont);
        end
        saveas(deph_img,file_name);
        close(deph_img);
    catch
        disp('Depth not available');
   end

    try
        % Select color frame
        color_img = figure;
        colorRGB = fs.get_color_frame();
        data = colorRGB.get_data();
        imgRGB =
permute(reshape(data',[3,colorRGB.get_width(),colorRGB.get_height()]),
[3 2 1]);
        imshow(imgRGB);
        title(sprintf("Color frame %i from %s",cont,name));
        if  i<10
                file_name = sprintf("Fotos/ColorImage000%i.png",cont);
        elseif i<100
                file_name = sprintf("Fotos/ColorImage00%i.png",cont);
        else
                file_name = sprintf("Fotos/ColorImage0%i.png",cont);
        end

        saveas(color_img,file_name);
        close(color_img);
```

```
    catch
        disp('Color not available');
    end
end
```

2.11 ESCANERSCRIPT.M

```
%Se establece la conexión socket con el robot.
port = 30002;
s = tcpip("0.0.0.0", port,'NetworkRole', 'server');
s.terminator = '';
fopen(s);
%Espera hasta establecer la conexión.
disp('¡Conectado!');

%Se inicializan las variables y se obtiene las poses según los
parámetros
%indicados.
next = 0;
foto = 0;

posicion  = [0.25,0,0.2];
escala    = 0.1;
elevacion = [0,90];
azimut    = [-125,125];
[E] = esfera_completa(posicion,escala,elevacion,azimut);
[Cell] = pose_puntos(E,posicion);

%Se inicializa la cámara.
[colorizer,pipe] = preparacion_rosbag('motor.bag');

%Bucle principal, donde se mandan las poses y la orden de tomar
imágenes.
for i = 1:length(Cell)
    %Espera hasta que el robot necesite una pose nueva.
    while next ~= 1
        next = fread(s,1);
    end
    next = 0;
    %Se obtiene las posiciones y orientaciones para mandarlas vía
socket.
    Mat = Cell{i};
    RotMat = Mat(1:3,1:3);
    %Rotaciones en formato axis-angle.
    axang = rotm2axang(RotMat');

    Rot(1,1) = axang(1)*axang(4);
    Rot(1,2) = axang(2)*axang(4);
    Rot(1,3) = axang(3)*axang(4);

    PosX = Mat(1,4);
    PosY = Mat(2,4);
    PosZ = Mat(3,4);

    %Se genera el script que será mandado por Socket.
        str = sprintf ("[%f,%f,%f,%f,%f,%f]",PosX,PosY,PosZ,...
        Rot(1),Rot(2),Rot(3));
    if i == 1
        str = sprintf("[%f,%f,%f,%f,%f,%f]",PosX,PosY,PosZ,...
        Rot(1),Rot(2)+pi,Rot(3)-pi);
    end
    %Se mandan las posiciones y orientaciones.
    fprintf(s,str);
```

```
    %Se espera a que el robot llegue a la posición y orientación
indicada.
    while foto ~= 2
        foto = fread(s,1);
    end
    foto = 0;
    %Se toma la imagen.
    foto_camara('motor.bag',pipe,colorizer,i);

end
    %Se cierra la comunicación y la toma de imágenes.
    fclose(s);
    pipe.stop();
```

3 PROGRAMACIÓN EN UNIVERSAL ROBOTS

3.1 ESCANER.URP

```
Program
   BeforeStart
     open:=socket_open("192.168.56.1",30002,"socket1")
     Cont:=0
     Loop open≟ False
       open:=socket_open("192.168.56.1",30002,"socket1")
     targetPos:=p[0,0,0,0,0,0]
   Robot Program
     pose1:=socket_read_ascii_float(6,"socket1")
     Loop pose1[0] ≠6
       Wait: 0.3
       next_point:=socket_send_int(1,"socket1")
       pose1:=socket_read_ascii_float(6,"socket1")
     Loop 6 times
       targetPos[Loop_2] =pose1[Loop_2+1]

JointPos:=get_inverse_kin(targetPos,qnear=[0,2.528,2.224,0.072,1.5707,
1.95], maxPositionError = 0.0001 ,maxOrientationError = 0.0001)
     MoveJ
       JointPos
     Loop 10 times
       image:=socket_send_int(2,"socket1")
     Debug:=get_actual_tcp_pose()
     Cont:=Cont+1
```