



PROYECTO RLP - COMPUTER ENGINEERING

Matt-Omato

Trabajo realizado por

Alejandro Miranda Herrero

Martí Arnaus Comellas

Youssef Assbaghi Asbahi

Pablo Mora Claros

Índice general

1. Introducción	1
1.1. Descripción del proyecto	1
1.2. Componentes electrónicos	1
2. Esquemas	2
2.1. Introducción	2
2.2. Arquitectura Hardware	2
2.2.1. Cámara	2
2.2.2. Sensores	3
2.2.3. Motores	3
2.2.4. Actuadores	4
2.2.5. Controladores	4
2.2.6. Fuentes de Alimentación	5
2.3. Arquitectura Software	6
2.3.1. Visión por computador	7
2.3.2. Movimiento del robot	12
2.4. Flujo de trabajo de Matt-Omato	16
3. Contribuciones	17
4. Piezas 3D	18
5. Estrategia de simulación	19
5.1. Entorno	19
5.2. Matt-Omato en CoppeliaSim	21
5.3. Movimiento entre tomateras	21
5.4. Cosecha de tomates	22
6. Resultados obtenidos	24
6.1. Escenas sencillas	24
6.2. Escenas intermedias	25
6.3. Escenas difíciles	26
7. Riesgos previstos y planes de contingencia	27
8. Bibliografía	28

1. Introducción

1.1. Descripción del proyecto

Nuestro proyecto es sobre un robot autónomo cosechador de tomates. El objetivo del robot es ayudar a los agricultores para cosechar frutas automáticamente y sin esfuerzo físico.

Matt-Omato está formado por diferentes partes. La primera de ellas es el brazo robótico con una pinza en el extremo encargado de coger los tomates y depositarlos en la cesta incorporada a la estructura del robot. La base del robot, dotada de 4 ruedas que se mueve de manera autónoma (hacia delante y hacia atrás) en cuanto el usuario inicie el sistema. La otra parte más importante es el algoritmo de visión por computador basado en la detección de nube de puntos a través de una cámara RGB-D que es capaz de detectar color y profundidad.

Todo esto será posible mediante el uso de Raspberry Pi3 y el software Python 3.7.

Los proyectos en los cuales está basado nuestro robot son los siguientes [1, 2, 3, 4]

1.2. Componentes electrónicos

- PCB Board
- PLAYSTATION CAMERA (Cámara 3D estéreo)
- 4 ruedas
- RaspBerry Pi3
- Cableado
- 6 Servomotores
- 1 Motor NEMA
- Controlador I2C
- Controlador DC para las ruedas
- 2 motores DC
- Controlador L298
- Controlador L293
- Sensor de distancia por ultrasonido HC-SR04
- Batería 3.7V
- 2 Baterías 9V

2. Esquemas

2.1. Introducción

En este apartado se presentan los esquemas de Hardware y Software utilizados y la explicación detallada de cada uno de sus componentes.

2.2. Arquitectura Hardware

La parte Hardware de nuestro robot se basa en el esquema que se puede observar en la Figura 2.1. Lo más destacado son los 6 servomotores para el movimiento del brazo controlados por el “I2C Controller”, 2 motores DC para el movimiento de las ruedas controlado por el “DC controller”, un motor NEMA que es capaz de girar 360 grados para la base del brazo.

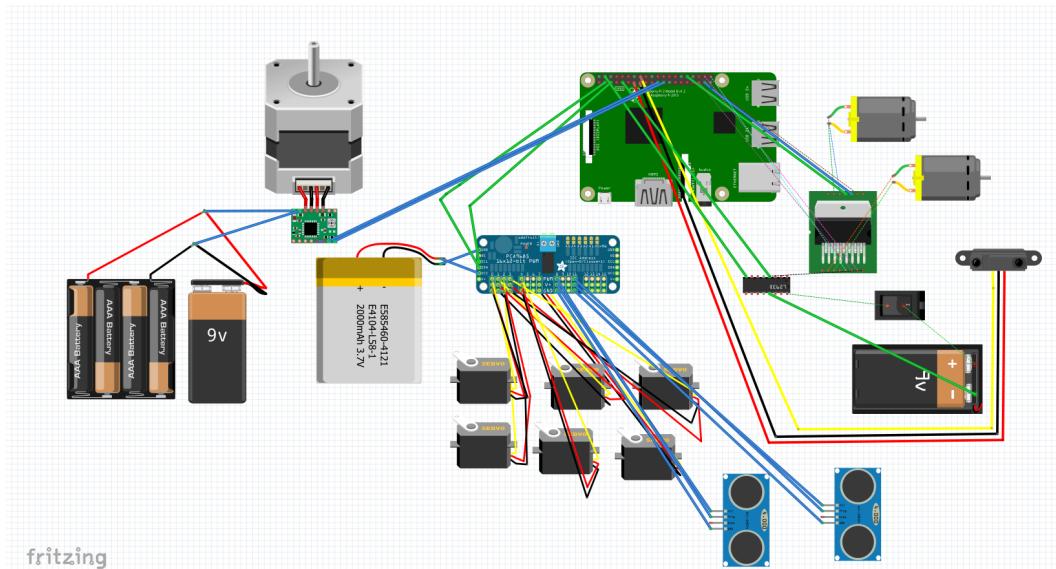


Figura 2.1: Diagrama de hardware

2.2.1. Cámara

Para la simulación de nuestro robot, utilizaremos una cámara con las mismas especificaciones que la “PlayStation Camera” la cual nos da información de profundidad e información de color de la escena capturada. Las especificaciones de la cámara utilizada son las siguientes:

PlayStation Camera	
Resolución	640px x 400px
Frame Rate	60 fps
Distancia focal	2.5 mm
Distancia Baseline	80 mm
Tamaño píxel	6 μ m
Campo de vista	83°
Distancia de uso	0.3 - ∞ m

Cuadro 2.1: Especificaciones PlayStation Camera.

2.2.2. Sensores

Es necesario saber cuando tiene que parar el robot cuando detecta la pared del invernadero. Así evitamos que se pueda chocar con cualquier objeto que detecte como también podría ser una persona. Por ello, utilizamos dos sensores Ultrasonido HC-SR04, uno en la parte delantera y otro en la parte trasera. Así cuando detecte un obstáculo de grandes dimensiones, Matt-Omato se para para evitar la colisión.

Sensor Ultrasonidos HC-SR04	
Rango Medición	2cm-400cm
Resolución	0.3cm
Frecuencia Ultrasonidos	40KHz
Alimentación	5V

Cuadro 2.2: Especificaciones Sensor HC-SR04.

2.2.3. Motores

Los servomotores son los encargados de realizar los movimientos del brazo del robot ya que tienen mas fuerza y precisión que un motor DC normal.

Servomotores	
Voltaje	5 V
Consumo	100 mA
Torque	12 kg/cm
Rotación	180°
Peso	55 g
Dimensiones	54 x 38 x 20 mm

Cuadro 2.3: Especificaciones servomotores.

Los motores DC son los encargados de mover las ruedas del robot.

Motor DC	
Voltaje	3 - 7.2 V
Potencia	24.6 W
Consumo	5.25 A
Torque	107.3 g/cm
Revoluciones	22356 rpm
Peso	69 g
Dimensiones	27 x 38 mm

Cuadro 2.4: Especificaciones motores DC.

A parte decidimos añadir un motor NEMA para la base del robot ya que podemos obtener más precisión y un giro de más de 360 grados para el momento de cambiar de línea de tomateras.

Motor NEMA	
Voltaje	12 V
Ángulo de paso	1.8 °
Fases	4
Torque	0.45 N/m
Dimensiones	42 x 40 x 72 mm

Cuadro 2.5: Especificaciones motor NEMA.

2.2.4. Actuadores

Interruptor encargado de iniciar la ejecución del robot.

Actuadores	
Valores nominales	16 A / 250 VAC
Dimensiones	25 x 30 x 38 mm
Temperatura	105°

Cuadro 2.6: Especificaciones interruptor.

2.2.5. Controladores

Ordenador principal encargado de ejecutar los algoritmos de visión por computador y de cinemática inversa para dar señal a los diferentes elementos del robot y realizar la acción final.

Raspberry pi3	
Procesador	Chipset Broadcom BCM2387 a 1,2 GHz de cuatro núcleos ARM Cortex-A53
GPU	Dual Core VideoCore IV Multimedia Co-procesador.
RAM	Proporciona Open GL ES 2.0 1GB LPDDR2.
Conectividad	Ethernet, 802.11 b/g/n, Bluetooth 4.1
GPIO	40-clavijas de 2,54 mm, Proporcionar 27 pines GPIO, así como 3,3 V, +5 V y GND líneas de suministro
Cámara	Conector de la cámara de 15 pines cámara MIPI interfaz en serie (CSI-2)

Cuadro 2.7: Especificaciones Raspberry pi3.

El controlador I2C se utiliza para controlar los 6 servomotores, pero solo se uno a la vez, por tema de la alimentación.

I2C	
Voltaje	5V
Máximo	16 servomotores

Cuadro 2.8: Especificaciones controlador I2C

El controlador L298 es el encargado de controlar los dos motores DC, los que mueven todo el robot hacia adelante o hacia atrás. Este controlador permite establecer a la velocidad que se mueven los motores así como la dirección (hacia adelante o atrás)

L298	
Voltaje	5V
Alimentación motores	12V máx

Cuadro 2.9: Especificaciones controlador L298

Este último controlador lo utilizamos para dar energía a través de una batería, a la Raspberry, así como a los motores DC y la cámara.

L293	
Voltaje	4.5V
Intensidad	600mA

Cuadro 2.10: Especificaciones controlador L293

2.2.6. Fuentes de Alimentación

Esta primera batería se conecta al controlador I2C para alimentar a ese mismo, así como a los servomotores, que como hemos comentado antes, se utilizan de uno en uno.

Batería 3.7V	
Capacidad	2000mA
Peso	80g

Cuadro 2.11: Especificaciones batería 3.7V

La segunda batería es la encargada de subministrar energía a la placa Raspberry y a los motores DC a través de la L293.

Batería 9V	
Capacidad	2500mA
Peso	80g

Cuadro 2.12: Especificaciones batería 9V

2.3. Arquitectura Software

En este apartado se definen los módulos necesarios para poder poner a nuestro robot en marcha. Tanto la parte de visión por computador implementada, capaz de detectar la posición del tomate mediante una nube de puntos 3D, como la parte de cinemática inversa capaz de mover el brazo robot a la posición indicada por la cámara.

En la Figura 2.2 se puede ver el esquema software que sigue nuestro robot Matt-Omato.

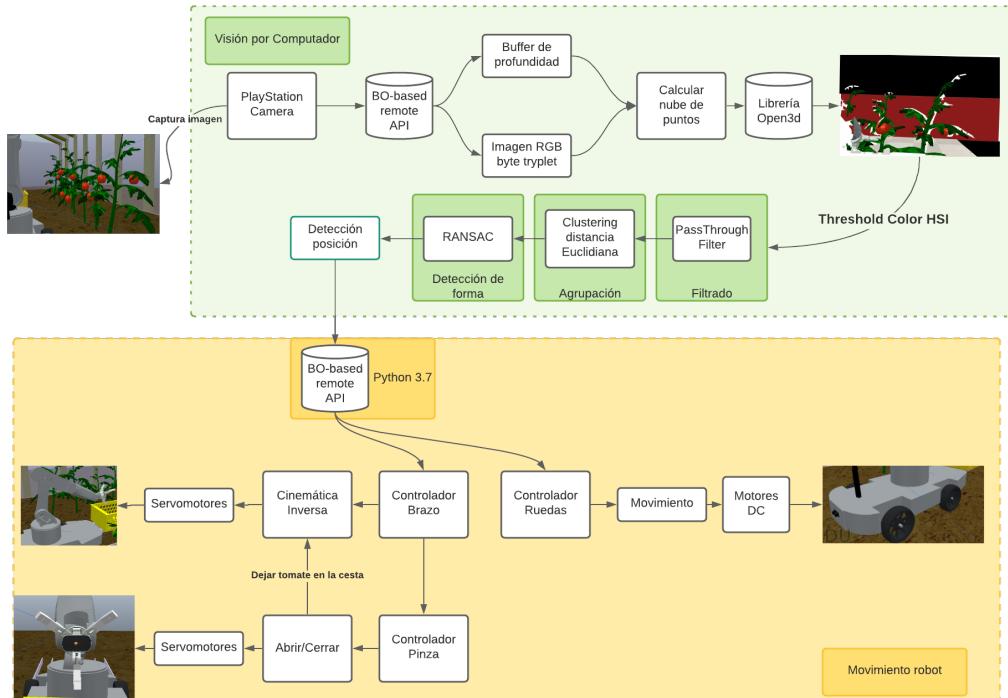


Figura 2.2: Diagrama de software

En él podemos ver diferenciadas las dos partes que conforman nuestro robot. En color verde tenemos la parte de visión por computador y en color amarillo la parte del movimiento del robot en sí.

2.3.1. Visión por computador

2.3.1.1. Nube de puntos

En cuanto a la parte de visión por computador, dada la escena simulada en Coppelia, tenemos la adquisición de la imagen mediante el objeto “Vision Sensor” con las propiedades de la cámara de PlayStation. Para controlarlo es necesario utilizar la BO-based remote API la cual nos permite conectar nuestro robot de manera remota gracias a código en Python 3.7.

Le pedimos a la API una imagen a color donde cada píxel es representado por un byte RGB triple, esta imagen la convertimos a una matriz 3D y después hacemos un flip vertical.

También obtenemos un buffer de profundidad donde los valores están en rango entre 0-1, lo convertimos a una matriz 2D y hacemos un flip vertical.

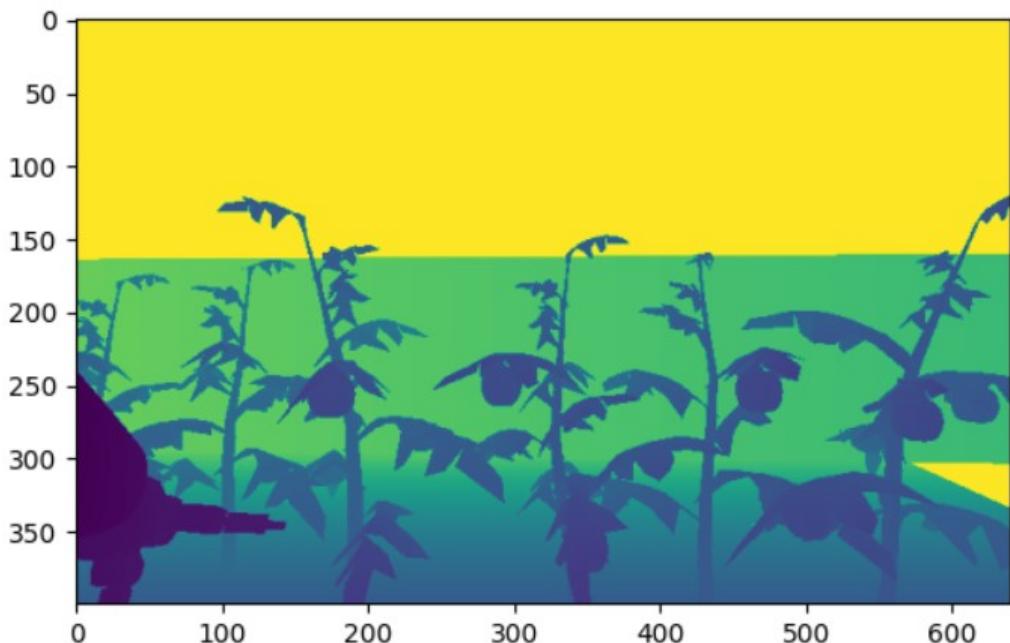


Figura 2.3: Imagen de profundidad

Una vez hecho esto se calcula la posición XYZ de los puntos para poder calcular

la nube de puntos. Las fórmulas utilizadas son las siguientes:

$$FocalLength = (v_res/2,0) / \tan((FOV/2,0) * \pi/180) \quad (2.1a)$$

$$Z = far_clip * bdeep[u][v] + near_clip \quad (2.1b)$$

$$X = (v - v_res/2,0) * z / FocalLength \quad (2.1c)$$

$$Y = (u - v_res/2,0) * z / FocalLength \quad (2.1d)$$

Donde v_res , en la ecuación 2.1a, hace referencia a la resolución vertical de la cámara, en este caso 640 y donde $FOV = Campo\ de\ Vista/2,0$. Para el cálculo de Z en la ecuación 2.1b, *far_clip* es lo máximo a lo que llega a enfocar la cámara en este caso 3.5 metros y *near_clip* es lo mínimo, en este caso 0.01 metros. Por último en las ecuaciones 2.1c y 2.1d, v y u hace referencia a las posiciones de los píxeles capturados.

Posteriormente, dado que la cámara está posicionada con cierto ángulo, en el momento de representar la nube de puntos correctamente, se le aplica una corrección a las coordenadas x e y previamente calculadas mediante la matriz de rotación 3D respecto el eje Z. En el caso de que el robot este circulando hacia delante se aplica una corrección del ángulo de la cámara respecto el plano paralelo a la linea de cultivo, en el caso que este circulando hacia atrás se hace una corrección con el ángulo más 90 grados. Los cálculos que se realizan en los puntos para hacer esta corrección son los siguientes, donde θ hace referencia al ángulo que tiene la cámara:

$$X = X * \cos \theta - Y * \sin \theta \quad (2.2a)$$

$$Y = X * \sin \theta + Y * \cos \theta \quad (2.2b)$$

$$Z = Z \quad (2.2c)$$

Para poder representar los puntos como una nube, usamos la librería Open3D [5] con la clase PointCloud, a estos puntos se le asigna un color HSI extraído de la imagen original.

2.3.1.2. Threshold HSI Color

En este punto realizamos un cambio de espacio de color en nuestra nube de puntos. De esta forma lo representaremos mediante el espacio HSI (o HSL) el cual nos permite definir el color mediante el tono (hue), saturación (saturation) e intensidad (intensity). Estos componentes nos permiten hacer una mejor representación ya que refleja la forma en que el sistema visual humano observa el color, la tonalidad indica la información intrínseca al color, la saturación es la diferencia en el color respecto el brillo y la luminancia es la percepción del brillo por los humanos.

Una vez tenemos este nuevo modelo procesado, el objetivo es realizar un threshold de manera que sólo nos quedemos con aquellos colores que nos interesa detectar (en este caso tomates de color rojo, tomates de un color verde simulando la no madurez

total y tomates de un color anaranjado que representa el estado intermedio hacia la madurez). En este caso nos quedamos con los umbrales siguientes:

$$(H < 0,20 \text{ or } H > 0,97), S > 0,6, I > 0,23$$

El valor H puede tomar valores que sean más pequeños que 0.20 y más grandes que 0.97, esto es debido a, como se puede ver en la Figura 2.4, el valor de H representa un ángulo (en nuestro caso normalizado, como el Software *CoppeliaSim*, para que de valores entre 0 y 1 al igual que la resta) y cuando llega al límite, empieza de nuevo al valor mínimo (movimiento circular).

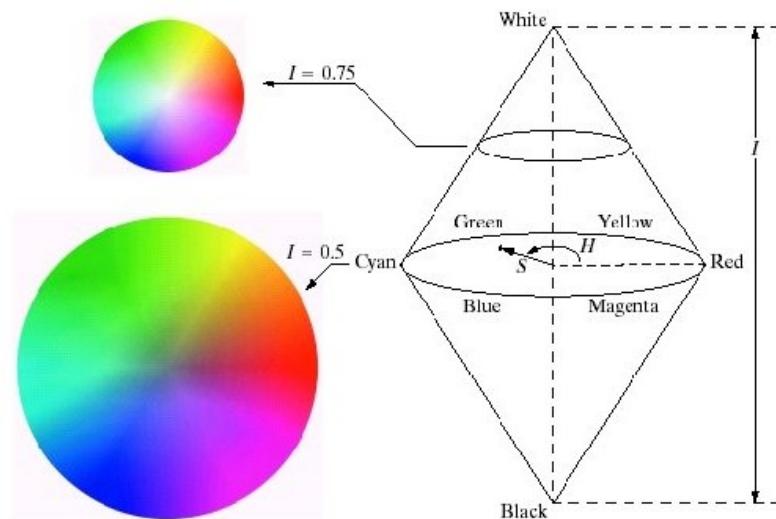


Figura 2.4: Espacio de color HSI

El resultado de la aplicación se puede ver en la Figura 2.5, donde por un lado se ve la escena original (2.5(a)) y por otro lado la nube de puntos con la aplicación del paso al espacio de color HSI con el threshold aplicado (2.5(b)) donde solo nos quedamos con el color rojo en este ejemplo.

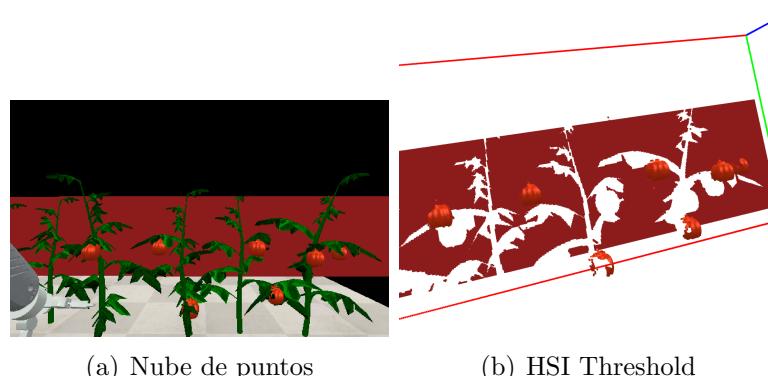


Figura 2.5: Diferencia de la aplicación de HSI Threshold

2.3.1.3. Filtrado

En este paso previo a la clusterización, realizamos un filtrado en la nube de puntos que nos permite reducir la cantidad de ellos dentro de un rango delimitado. Este filtro llamado “PassThrough Filter”, nos permite definir una dimensión (un rango) en la cual acotar la nube de puntos. Gracias a ello vamos a poder quitar aquellos puntos generados automáticamente por el *CoppeliaSim* que conforman el fondo de la escena.

Para ello se ha definido un rango fijo donde la cámara pueda detectar correctamente los tomates y quedarnos solo con los puntos cercanos referidos a ellos. En este caso la *Bounding Box* tiene los siguientes límites:

$$X : [-2, 2], Y : [-0,75, 0,75], Z : [0,5, 2]$$

En la Figura 2.6 podemos ver la diferencia entre la nube de puntos original y la nube de puntos después de la aplicación del filtro. Para hacerlo más visual hemos generado una *Bounding Box* donde los puntos que caen dentro de ella son los resultantes de este filtrado. De esta manera nos evitaremos crear puntos de fondo innecesarios y también nos ahorraremos tiempo.

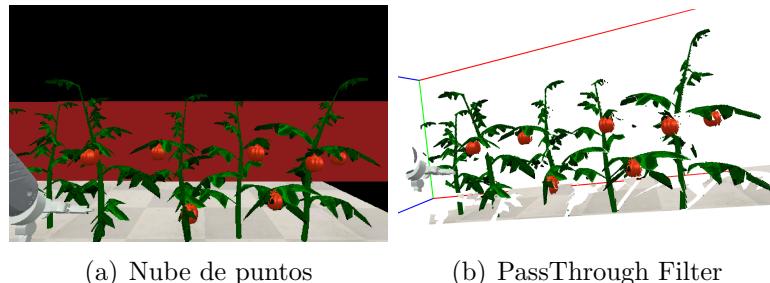


Figura 2.6: Diferencia de la aplicación de PassThrough Filter

2.3.1.4. Clustering

Gracias a las dos etapas anteriores hemos sido capaces de solo quedarnos con la nube de puntos de los propios tomates (Figura 2.7), y aunque se han tenido que hacer las comprobaciones necesarias en todo momento, el tiempo resultado es mucho menor ya que para la visualización hemos tenido que generar menos puntos totales.

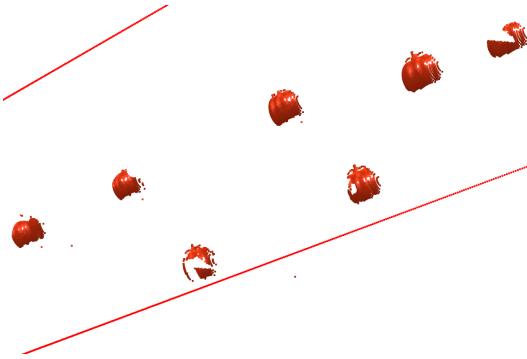


Figura 2.7: Nube de puntos resultante del HSI Threshold y PassThrough Filter

Ahora que solo tenemos aquella nube de puntos de colores que nos interesa, aplicamos un algoritmo de clustering. Para hacerlo posible, utilizamos la librería utilizada anteriormente para la creación y manipulación de nube de puntos 3D *Open3d* [5]. Esta librería nos permite hacer uso de una función llamada *cluster_dbscan*, inspirada en DBSCAN Cluster [6], la cual le pasamos por una banda el valor de ϵ como la densidad para encontrar los vecinos y *min_points* como el mínimo número de puntos para poder crear un cluster. Esta función devuelve las etiquetas de cada uno de los clusters encontrados. El resultado de dicha clusterización se puede ver en la Figura 2.8 donde se le ha dado un color diferente a cada uno de los cluster para poder diferenciarlo de manera correcta.

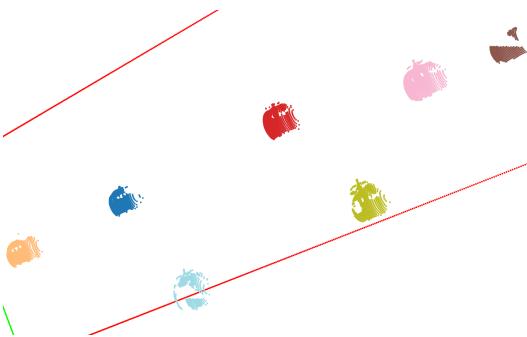


Figura 2.8: Clusterización

2.3.1.5. Extracción de forma

Con el clúster obtenido ya, aplicamos RANSAC para segmentar los clusters. De esta manera conseguimos una forma de esfera (adecuándonos a los diferentes tipos de tomates que puede haber). Para hacerlo posible, para cada uno de los clusters anteriormente conseguidos, utilizamos sus puntos para llamar a la función *Sphere.fit* de la libreria *pyRANSAC-3D* [7] la cual nos permite ajustarlos a una forma de esfera. El resultado de esta función nos devuelve el centro y el radio de la esfera creada. El resultado de la aplicación del RANSAC se puede ver en la Figura 2.9

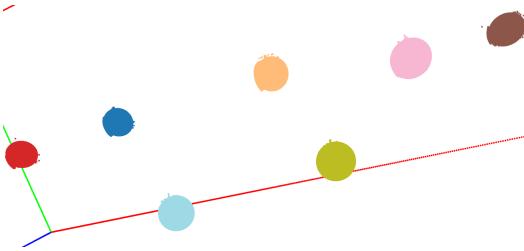


Figura 2.9: RANSAC

Gracias al cálculo de los centros de las esferas somos capaces de calcular las coordenadas de los tomates en referencia a CoppeliaSim.

2.3.2. Movimiento del robot

En cuanto a la parte de todo el movimiento del robot, como ya se ha comentado antes, se controla mediante la BO-based remote API conectada a Python. Ello está conectado a diferentes funciones que serán capaces de:

- Mover el brazo robótico a través de servomotores (revolute joints en CoppeliaSim) mediante cinemática inversa calculada para los $5 + 1$ grados de libertad de Matt-Omato.

Para esto, se ha creado una clase robot, donde se conecta con el Coppelia, obtenemos los “handles” de cada joint así como las medidas del brazo del robot. Con esta clase también obtenemos la posición de cualquier objeto de la escena. La clase movimiento es la encargada de la cinemática inversa, dándole unas coordenadas y las medidas del brazo del robot.

- Abrir o cerrar la pinza cuando el brazo se haya situado en las coordenadas de un tomate gracias al sevomotor que lo controla (revolute joints en CoppeliaSim).
 - Mover todo el robot mediante las ruedas conectadas a motores DC (revolute joints en CoppeliaSim) cuando no se detecte ningún tomate.
- Matt-Omato se mueve entre las tomateras del invernadero a través de raíles posicionados en el suelo de manera recta. Esto nos va a permitir no desviarnos de las tomateras y chocar contra ellas.

Para conseguir lo comentado anteriormente hemos implementado 4 clases:

- Robot
- Movimiento
- PointCloud
- Simulación

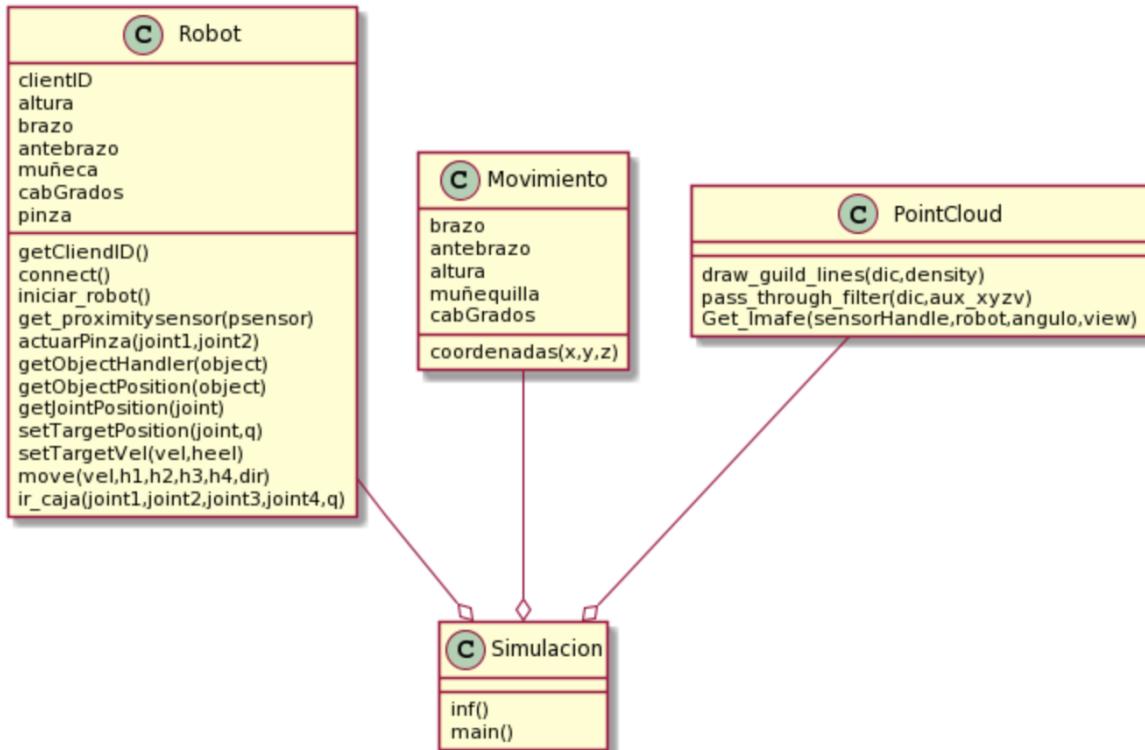


Figura 2.10: Diagrama Clases

2.3.2.1. Clase Robot

En esta clase tenemos definidas todas aquellas funciones encargadas de conectar el entorno creado de Python con el software de CoppeliaSim. Pasando los nombres definidos como variables constantes, somos capaces de obtener la conexión necesaria. Aparte de dichas funciones encontramos algunas más importantes como:

- **actuarPinza:** Encargada de abrir o cerrar la pinza según el punto de la simulación en que se encuentre.
- **move:** Encargada de mover los Joints relacionados con las ruedas. Según las ordenes que se envíen se moverán hacia un lado u hacia otro. Para el movimiento de las ruedas se ha definido un movimiento de un segundo, que relacionado con la respuesta de la parte de visión por computador hará más o menos veces. Es decir, si no detecta tomate, irá avanzando de segundo en segundo (con un intervalo en medio de otro segundo) hasta la próxima detección de tomate que es entonces cuando se para.

- **ir_caja:** Define el movimiento preestablecido del brazo hacia la caja situada en frente de Matt-Omato.

2.3.2.2. Clase Movimiento

Esta clase es la encargada de realizar la cinemática inversa del brazo robótico definiendo como atributos de la clase las medidas del brazo robótico que se moverán en el momento de coger un tomate. La cinemática inversa la hacemos de manera geométrica, donde es necesario saber la distancia entre cada joint así como la inclinación en grados del cabeceo del brazo. Luego con las siguientes fórmulas encontramos los grados que debe girar cada joint para llegar a la (x,y,z) destino. A continuación especificamos la función:

- **Coordenadas:** Aplica las fórmulas dadas para encontrar los ángulos para dar a los joints. El primer ángulo a calcular es el del joint de la base que se calcula de manera sencilla:

$$Angulo1 = atan2(y, x)$$

Para el resto de ángulos, el del codo, hombro y el de la muñeca se utilizan varias formulas relacionadas entre sí:

$$M = \sqrt{x^2 + y^2}$$

$$Afx = \cos(cabRAD) * m$$

$$B = M - Afx$$

$$Afy = \sin(cabRAD) * m$$

$$A = z + Afy - H$$

$$Hip = \sqrt{A^2 + B^2}$$

$$alfa = atan2(A, B)$$

$$\text{beta} = \arccos((b^2 - ab^2 + Hip^2) \div (2 * b * Hip))$$

$$Angulo2 = alfa + beta$$

$$\gamma = \arccos((b^2 - ab^2 + Hip^2) \div (2 * b * ab))$$

$$Angulo3 = \gamma$$

$$Angulo4 = 2\pi - cabRAD - Angulo2 - Angulo3$$

Por último, hay que hacer unas correcciones antes de que los ángulos sean los definitivos:

$$Angulo1 = Angulo1 + \pi \div 2$$

$$Angulo2 = \pi \div 2 - Angulo2$$

$$Angulo3 = \pi - Angulo3$$

$$Angulo4 = \pi - Angulo4$$

2.3.2.3. PointCloud

En esta sección se encuentran todas las funciones relacionadas con la parte de visión por computador. En ella encontramos el bucle principal el cual mediante el objeto de la cámara de CoppeliaSim se encarga de transformar los puntos obtenidos tal y como se explica detalladamente en toda la sección [2.3.1](#) para el posterior cálculo de las coordenadas de los tomates que aparecen en la escena.

2.3.2.4. Simulación

En la parte de simulación definimos el bucle principal de todo el robot. Definimos todas las partes necesarias del robot con la clase *Robot* [2.3.2.1](#) y definimos algunas variables constantes en el programa como lo son el número de tomates de la escena, el ángulo de la cámara de la escena y si queremos o no visualizar los resultados que la parte de visión por computador [2.3.2.3](#) nos va dando.

A partir de este punto es necesario, para la posterior simulación de la cosecha de manera correcta, la obtención de todas las coordenadas de las primitivas de esferas relacionadas con los tomates para poder después hacer correspondencia con las coordenadas resultantes de la visión por computador y hacer el bloqueo del objeto esfera como hijo de la pinza para poder moverlo mejor y para el posterior cambio de dinamismo para hacerlo de manera más realista.

Posteriormente realizamos el bucle principal en el cual en primera instancia comprobamos si el sensor de distancia detecta algún elemento, cosa que querrá decir que debemos de cambiar la dirección del robot y a su vez la orientación de la cámara, para poder ir a la otra fila de tomates.

En el caso de no detectar ningún elemento que impida el avance de Matt-Omato, se ejecuta la parte de visión por computador [2.3.2.3](#) para obtener los centros. En caso de no obtener ninguno avanzamos un segundo a una cierta velocidad. En caso de detectar algún tomate, realizamos la cinemática inversa no sin antes aplicar cierta corrección mínima a las coordenadas obtenidas por la visión. Cuando el brazo se sitúa en frente de un tomate, cierra la pinza y aplica la técnica de padre e hijo con la pinza y el tomate respectivamente, para poder coger de manera correcta el tomate y después poder dejarlo en la caja.

Este último bucle se realiza de manera infinita hasta que el usuario decida cortar la ejecución.

2.4. Flujo de trabajo de Matt-Omato

El flujo de trabajo de nuestro robot es mostrado en la Figura 2.11.

En él podemos ver que mediante la activación del robot por parte de un usuario, gracias a un interruptor instalado, Matt-Omato se pone en marcha. Lo primero que comprueba Matt-Omato a través de los sensores de profundidad es la posible detección de algún elemento situado delante de él. En el caso de detectar, se rota la cámara para poder apuntar a la otra línea de tomateras y se cambia la dirección de movimiento. En el caso de no detectar ningún elemento, se ejecuta el algoritmo de visión por computador. La cámara RGB-D está activa durante toda la simulación. Si detecta un tomate, el robot realiza la cinemática inversa para ir hasta las coordenadas del tomate que la parte de visión por computador nos devuelve. El brazo coge el tomate mediante el cierre de la pinza para posteriormente llevar dicho tomate a la cesta. Es entonces cuando vuelve al punto de la posible detección de elementos mediante el sensor de profundidad.

En el caso de no detectar ningún tomate, Matt-Omato se mueve hacia la dirección indicada durante un segundo, y vuelve a hacer la comprobación de la posible detección de elementos en la dirección a la que va.

La ejecución de Matt-Omato se hace de manera indeterminada hasta que el usuario decida poner fin mediante el interruptor instalado.

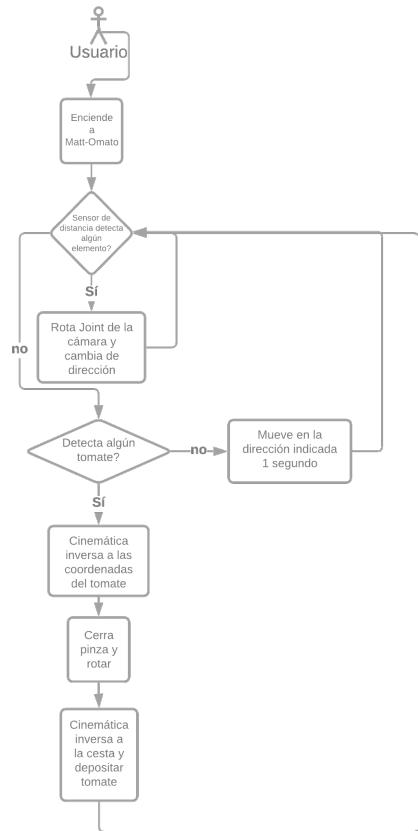


Figura 2.11: Flujo de trabajo Matt-Omato

3. Contribuciones

Este proyecto está diseñado para facilitar y agilizar la ardua tarea de la cosecha. En este país el sector de la agricultura sigue siendo una parte fundamental. En lugares como Andalucía o Extremadura, la mayor parte de la población se dedica a ello. En Almería, por ejemplo, una ciudad de Andalucía, es conocida por las grandes extensiones de invernaderos, donde aparte de plantar otras frutas y hortalizas, los tomates son los más frecuentes. Es por ello por lo que con este proyecto queremos ayudar a los agricultores para facilitar y evitar el trabajo físico que conlleva, ya que sabemos de primera mano que es un trabajo muy difícil.

Nuestro proyecto tiene un punto interesante desde la vista de la cosecha, ya que cogemos los tomates con una pinza de 3 para asegurar el agarre correcto. Con una rotación de la pinza, conseguimos que el tomate se arranque de manera suave y sin hacer gestos bruscos.

Matt-Omato solo necesita la mano humana para ponerse en marcha. Una vez está encendido, de manera autónoma hace todo el trabajo de cosecha de tomates en una línea de tomateras gracias a las raíles incorporados que hace que no se desvíe de su trayectoria recta.

Con la parte de visión por computador, queremos dotar a nuestro robot de una eficiencia buena, ya que el hecho de trabajar con nubes de puntos y cámara RGB-D puede hacer que la adquisición y detección de tomates se haga más fácil y de manera rápida y eficiente.

4. Piezas 3D

En esta parte se presentan las piezas 3D diseñadas en TinkerCad mediante producción propia y piezas del diseño de *misbah_46* [8]



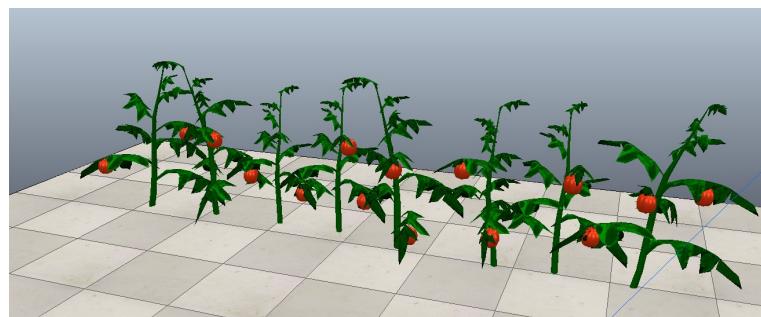
Figura 4.1: Piezas 3D Matt-Omato

5. Estrategia de simulación

El simulador utilizado es CoppeliSim. En él se simula una plantación de tomates mediante la importación de modelos 3D de AutoCad (tomates y plantas). Cabe destacar que se han creado un total de 10 escenas en las cuales iremos variando el nivel de dificultad de la detección y cosecha de los tomates variando tamaño, color y oclusión.

5.1. Entorno

Se puede decir que en esta simulación hay dos grandes partes. La primera consiste en el entorno de actuación del robot. Está compuesto por una o dos líneas (dependiendo de la dificultad de la escena) de tomateras a cada lado, estas plantas incluyen tomates entre las hierbas que tienen que ser accesibles por el robot, los tomates se ubican a múltiples alturas y profundidades para poder probar el robot de manera óptima. En la Figura 5.1(a) podemos ver un ejemplo de escenario montado para la simulación de Matt-Omato y en la Figura 5.1(b) podemos ver el diferente tipo de colores que podemos aceptar, los diferentes tamaños de tomates y también la aparición de inclusión en la escena.



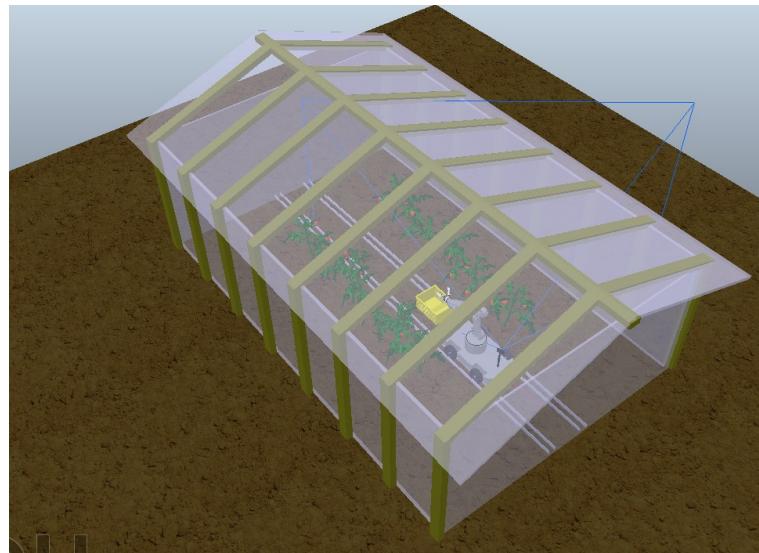
(a) Escena tomateras



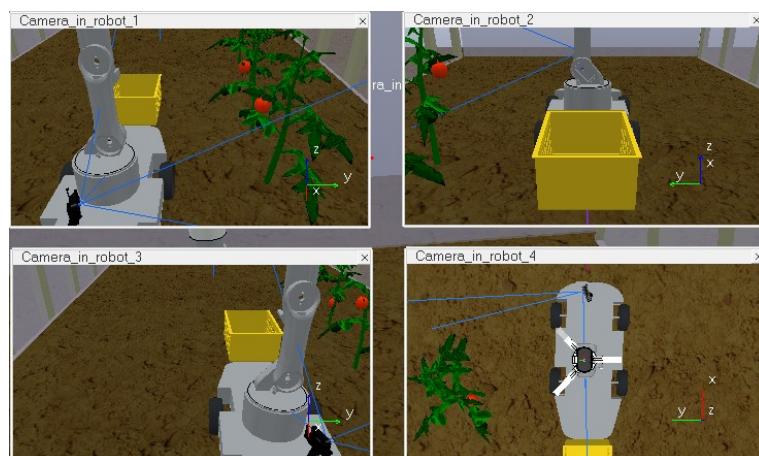
(b) Ejemplo tamaños, colores y oclusiones

Figura 5.1: Ejemplo simulación en *CoppeliaSim*

Para hacer una simulación lo más realista posible, extrapolándolo a la vida real, se ha realizado una estructura de invernadero para simular a Matt-Omato en un entorno real. Para poder ver correctamente el trabajo que está realizando Matt-Omato hemos instalado unas cámaras que lo siguen con el movimiento. En la Subfigura 5.2(a) se puede ver el entorno de invernadero creado y en la Subfigura 5.2(b) el conjunto de cámaras que siguen a Matt-Omato.



(a) Escena invernadero



(b) Cámaras

Figura 5.2: Escena CoppeliaSim

5.2. Matt-Omato en CoppeliaSim

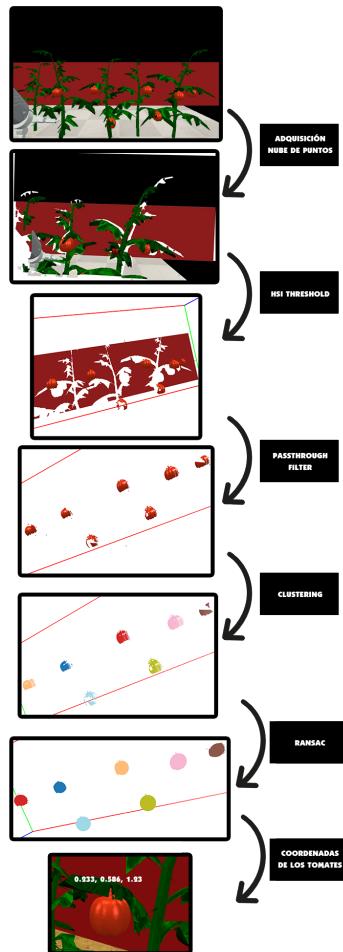


Figura 5.3: Flujo de Visión por Computador de Matt-Omato

medida y con fuerzas estándares adquiridas por referencias a otros robots similares.

5.3. Movimiento entre tomateras

Para poder simular correctamente el movimiento de Matt-Omato a través de las tomateras, se han creado raíles que van por debajo del mismo. De esta manera nos aseguramos que no atraviese ninguna planta ni se desvíe del camino recto establecido. Para poder aplicarlo a diferentes líneas de tomateras será preciso el uso de diferentes Matt-Omato's los cuales serán capaces de recoger tomateras primero en una dirección y luego en otra. En la Figura 5.4 se puede ver el uso de los raíles y del movimiento de Matt-Omato a través de dos líneas de tomateras.

El otro punto consta de todas las partes relacionadas con el robot, replicadas de la misma forma que en la vida real. En cuanto a la parte de visión por computador, la cámara RGB-D se simula mediante un objeto de CoppeliaSim llamado "Vision Sensor" al cual se le han modificado diferentes parámetros para que nos calcule la profundidad de la escena capturada y que tenga las mismas especificaciones que la "PlayStation Camera". Gracias a ello, somos capaces de obtener una nube de puntos de aquello que nos interesa. En la Figura 5.3 se puede ver la transición desde la imagen captada por la cámara hasta la obtención de la nube de puntos solo de la parte de los tomates que es la que realmente nos interesa, para el posterior cálculo de las coordenadas de los tomates.

A partir de aquí, Matt-Omato es capaz de realizar cinemática inversa para poder coger aquellos tomates los cuales la cámara ha captado. Para ello se ha creado a Matt-Omato en el entorno de CoppeliaSim a base de Joints que son los encargados de mover aquellas primitivas invisibles a las cuales va unido un disfraz que es lo que realmente se ve en la escena. De esta manera se visualiza correctamente todo el movimiento. Dichos Joints están hechos a

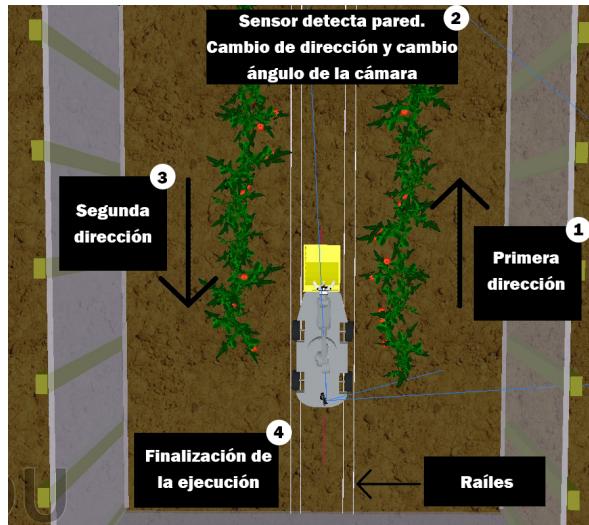


Figura 5.4: Movimiento de Matt-Omato entre tomateras

5.4. Cosecha de tomates

Para hacer una simulación de la cosecha de tomates de manera correcta ha sido necesario crear una primitiva de esfera por cada uno de los tomates (disfraz) que hay. Dichas esferas estarán conectadas a cada una de las plantas que hay en la escena. Al principio de la ejecución, no son dinámicas y solo son *respondable* con la caja donde lo guardaremos. En la Figura 5.5 se puede ver la jerarquía de las escenas creadas.

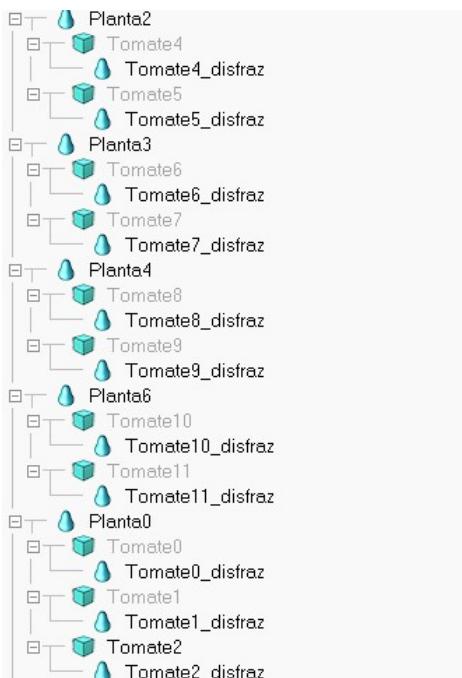


Figura 5.5: Jerarquía en CoppeliaSim

Cuando el brazo realiza la cinemática inversa para acercarse al tomate, realizamos

un cambio en las propiedades del tomate y de la pinza para ser capaces de cogerlo correctamente. Utilizamos la función `simxSetObjectParent` la cual realiza un bloqueo del objeto esfera con la base de la pinza como hijo. De esta manera podremos mover el brazo del robot de manera correcta y también mover correctamente el tomate. Una vez el movimiento del brazo llegue a la posición de la cesta, en el momento de la apertura de la pinza, cambiamos la propiedad del objeto esfera a objeto dinámico para hacer la simulación de la caída a la caja de forma realista. En la Figura 5.6 se puede ver de una manera más visual los pasos seguidos

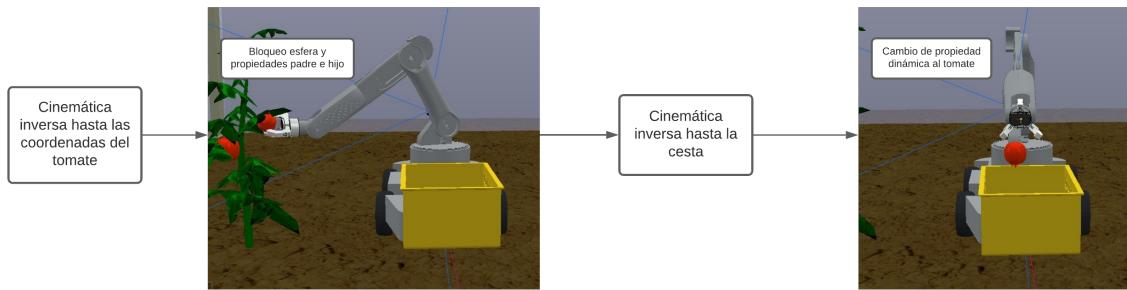


Figura 5.6: Cambio de propiedades en objetos de CoppeliaSim

6. Resultados obtenidos

En esta parte se observan los resultados obtenidos de la ejecución de Matt-Omato en relación a la exactitud calculada de la posición de los tomates, en un total de 10 escenas separadas según su nivel de dificultad. Tenemos un total de 3 dificultades empezando por la sencilla, la intermedia y por último la difícil.

6.1. Escenas sencillas

En esta sección se habla sobre las escenas más simples recreadas. En ella encontramos un total de 4 escenas con una sola fila, donde se va variando el tamaño, número, color y posición de los tomates. Las escenas se pueden ver en la Figura 6.1 donde se varía el tamaño y el color de los tomates.

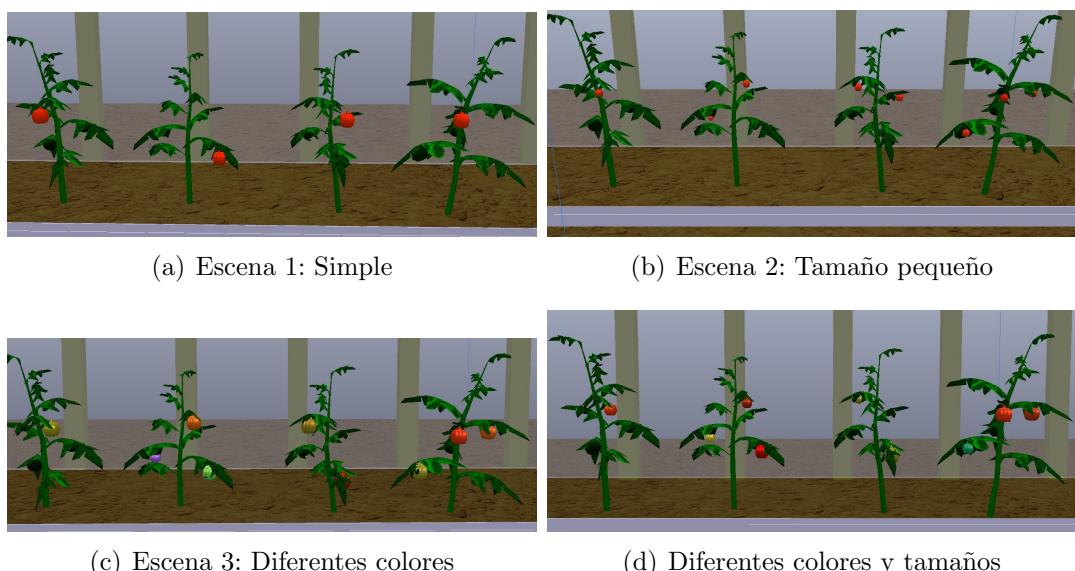


Figura 6.1: Escenas simples

En la Figura 6.4 podemos ver los resultados de los errores cometidos en cada una de las coordenadas de la diferencia entre los centros calculados por el robot y los centros reales de los tomates en *CoppeliaSim*. También podemos ver la distancia euclíadiana entre el supuesto centro calculado por el robot y el centro real del tomate así como también el error medio total.

Exactitud									
Escena	Tomates	Error Total X	Error medio X	Error Total Y	Error medio Y	Error Total Z	Error medio Z	Distancia Eucladiana Total	Error Medio Total
1	4	0,10431	0,0260775	-0,13623	-0,0340575	0,0633	0,015825	0,21308	0,05327
2	8	0,330662	0,04133275	-0,42686	-0,0533575	0,07016	0,00877	0,54757	0,06844625
3	7	0,2093	0,0299	-0,28257	-0,04036714	0,103086	0,014726571	0,3822	0,0546
4	8	0,3557	0,0419625	-0,4614	-0,057675	0,10121	0,01265125	1,16614	0,1457675

Figura 6.2: Resultado escenas sencillas

Se puede observar que para escenas simples el error medio cometido es significativamente pequeño. Se podría decir entonces que la solución es bastante exacta. Si nos fijamos en cada una de las coordenadas, podemos ver que nuestra solución es más exacta en la coordenada Z es decir, en la altura del tomate.

6.2. Escenas intermedias

En esta sección se habla sobre las escenas intermedias donde encontramos un total de 3 escenas en las cuales se han añadido más tomateras y tomates que en las escenas simples anteriores, en una única línea. En la Figura 6.3 podemos ver las diferentes escenas intermedias que se han creado

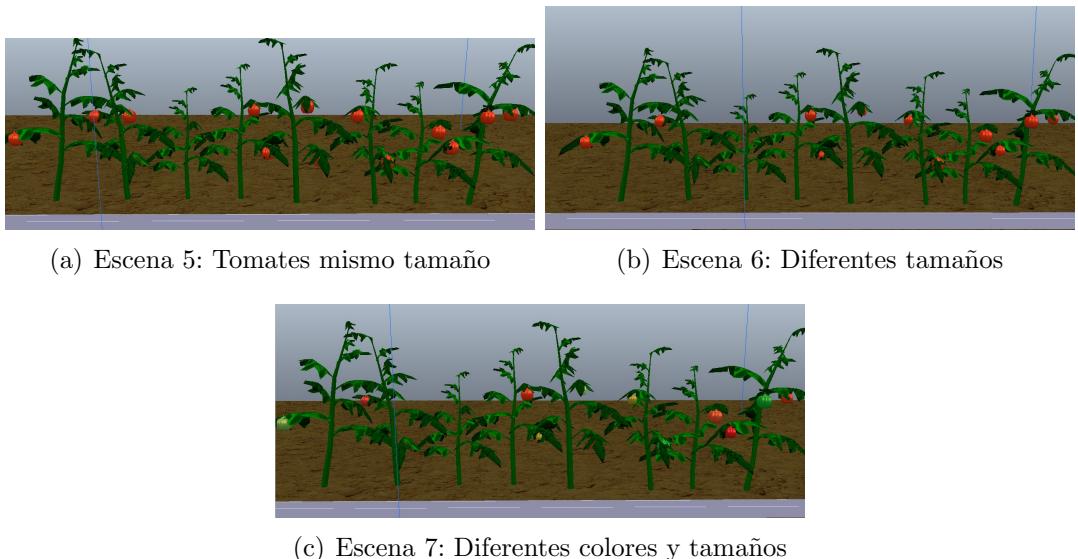


Figura 6.3: Escenas intermedias

En la Figura podemos ver los resultados de exactitud para este conjunto de escenas donde encontramos más tomates y por lo cual es más propenso a generar más errores.

Escena	Tomates	Exactitud								
		Error Total X	Error medio X	Error Total Y	Error medio Y	Error Total Z	Error medio Z	Distancia Euclíadiana Total	Error Medio Total	
5	13	0,52457715	0,040352088	-0,7759229	0,059686375	0,05861645	0,004508958	0,9649308	0,074225444	
6	12	0,40989956	0,034158297	-0,6767818	0,056398484	0,03389403	0,002824503	0,8138425	0,067820205	
7	8	0,30705157	0,038381446	-0,4682617	0,058532716	0,04832574	0,006040718	0,573148124	0,071643515	

Figura 6.4: Resultado escenas intermedias

Se puede observar que los errores al añadir más cantidad de tomates y por lo tanto tener que recrear más puntos en la nube de puntos es mayor. Aún así sigue siendo bastante preciso sin tener demasiado error significativo.

6.3. Escenas difíciles

En esta sección se habla sobre las escenas difíciles donde encontramos dos filas de tomateras y donde Matt-Omato puede ejecutarse en su totalidad. Hay un total de 3 escenas en las cuales varia el número de tomates, tamaño y colores. En la Figura 6.5 se puede ver las diferentes escenas.

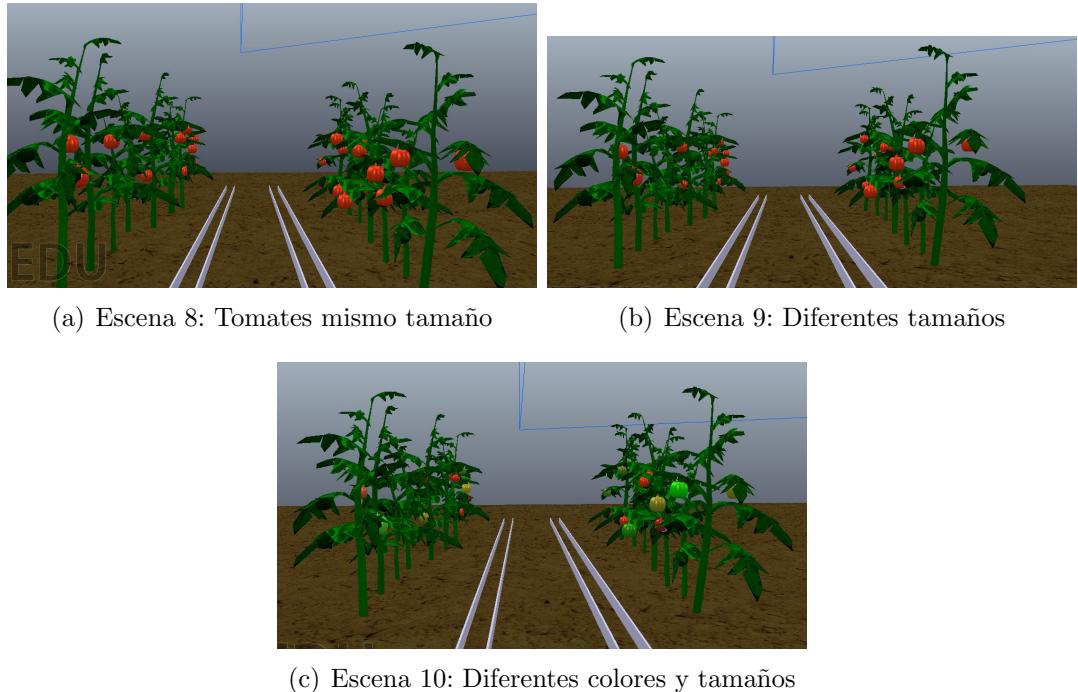


Figura 6.5: Escenas intermedias

En cuanto a los resultados de exactitud en torno a las escenas difíciles (6.6) se puede observar que pese aumentar el número de tomates significativamente, el error medio total se mantiene bastante estable en relación a todas las escenas creadas. El error en la coordenada Z sigue siendo el más bajo de todas las coordenadas.

Exactitud										
Escena	Tomates	Error Total X	Error medio X	Error Total Y	Error medio Y	Error Total Z	Error medio Z	Distancia Euclidian Total	Error Medio Total	
8	26	0,92196	0,03546	-1,291836	-0,049686	0,091208	0,003508	1,589707329	0,06114259	
9	36	1,32336	0,03676	-1,968696	-0,054686	0,0455445	0,001265125	2,372576644	0,065904907	
10	16	0,5264	0,0329	-0,67744	-0,04234	0,152	0,0095	0,871278322	0,054454895	

Figura 6.6: Resultado escenas difíciles

7. Riesgos previstos y planes de contingencia

Tabla de riesgos				
Riesgo	Descripción	Probabilidad	Impacto	Plan de contingencia
1	Coger el tomate con cinemática inversa y giro de más de 360 grados de la pinza	Media	Medio	Cosechar los tomates cortando por el tallo en vez de arrancarlos. Para ello sería necesario incorporar una tijera
2	Problemas con el procesamiento de la nube de puntos	Baja	Alto	Cambiar el algoritmo por una detección de contornos en vez de trabajar con nubes de puntos
3	Aplicar demasiada fuerza a la hora de coger el tomate y romperlo	Medio	Alto	Incorporar un sensor de fuerza para poder medirla
4	En relación con la simulación, no ser capaces de coger los tomates por debajo	Alta	Medio	Realizar la simulación de la manera más realista posible pudiendo traspasar objetos como son las plantas y coger los tomates de la mejor manera posible.
5	En relación con la simulación no ser capaces de coger los tomates con dinamismo.	Alta	Medio	Hacer que los tomates sean no dinámicos y aplicar la técnica de padre e hijo vista en clase de problemas.

Cuadro 7.1: Riesgos previstos

8. Bibliografía

- [1] Naurislv. Apple picking robot. https://github.com/Naurislv/apple_picking_robot, 2018.
- [2] Automato Robotics. Tomato harvester robot. <http://www.automatorobotics.com/autonomous-single-tomato-harvester/>, 2021. Acceso 28/03/2021.
- [3] Aleix Ripoll. Object recognition and grasping using bimanual robot. *Escola Tècnica Superior d'Enginyeria Industrial de Barcelona*, 2016.
- [4] T. HASEGAWA H. YAGUCHI, K. NAGAHAMA and M. INABA. Development of an autonomous tomato harvesting robot with rotational plucking gripper. *International Conference on Intelligent Robots and System*, 2016.
- [5] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.
- [6] Jiirg Sander Xiaowei Xu Martin Ester, Hans-Peter Kriegel. A density-based algorithm for discovering clusters, 1996.
- [7] L. MARIGA. pyransac-3d. <https://github.com/leomariga/pyRANSAC-3D>, 2020. Acceso 04/05/2021.
- [8] Misbah_46. Robotic arm. <https://www.tinkercad.com/users/ifysZbZAvdc-misbah46>, 2019. Acceso 14/04/2021.