UNIVERSIDAD
MODELO
INGENIERÍA

Examen Parcial 2

Visión por Computadora

Jorge Humberto Sosa García

Ingeniería Mecatrónica

Séptimo Semestre

(Agosto / diciembre 2025)

10 de noviembre de 2025

Universidad Modelo Campus Mérida

ÍNDICE

1 INTRODUCCIÓN 4
2 OBJETIVOS 5
2.1 Objetivo general 5
2.2 Objetivos específicos 5
3 MATERIALES Y HERRAMIENTAS 5
3.1 Software 5
3.2 Hardware 6
3.3 Archivos utilizados 6
4 METODOLOGÍA 6
4.1 Preparación del entorno de simulación 7
4.2 Diseño y carga del modelo del robot 8
4.3 Comunicación con Arduino y joystick 9
4.4 Configuración del entorno virtual 11
4.5 Implementación del sistema de visión y detección ArUco 13
4.6 Cámaras virtuales y visualización 15
1. Cámara montada en el robot (visión subjetiva) 15
2. Cámara externa tipo persecución 16
3. Cámara fija tipo vigilancia 17
5 CÓDIGO 18
5.1 Explicación código Arduino 18
5.2 Explicación código Python 20
6 RESULTADO 25
6.1 Conexión funcional Arduino–PyBullet 26
6.2 Textura ArUco como pared 26
6.3 Detección de marcador ArUco 27
6.4 Cálculo y visualización de distancia 27
6.5 Implementación de cámaras múltiples 27
6.6 Visualización en tiempo real 28

6.7 Pruebas realizadas 28
7 CONCLUSIONES 28
8 ANEXOS 29
8.1 Código Arduino 29
8.2 Código Python 30
9 REFERENCIAS 37

1 INTRODUCCIÓN

En la actualidad, los sistemas de visión por computadora se han convertido en herramientas clave para aplicaciones en robótica móvil, automatización industrial y navegación autónoma. Estos sistemas permiten a los robots obtener información del entorno a través de sensores visuales, procesarla en tiempo real y tomar decisiones informadas para ejecutar tareas específicas.

El presente proyecto tiene como objetivo integrar una simulación en el entorno físico virtual PyBullet con capacidades de visión por computadora, utilizando la biblioteca OpenCV y marcadores ArUco. Se desarrolla un sistema que permite a un robot móvil simulado detectar un marcador en una pared, calcular su distancia con respecto al mismo y visualizarla mediante líneas virtuales en el entorno.

El control del movimiento del robot se realiza de forma remota mediante un joystick físico conectado a un microcontrolador Arduino, el cual envía instrucciones al entorno simulado por medio de comunicación serial. Esta integración permite emular un entorno realista de navegación robótica, donde se combinan percepción visual, control remoto y simulación física de alta precisión.

Asimismo, se emplean múltiples cámaras dentro del entorno: una fija para supervisar el área de trabajo desde una perspectiva global y otra montada sobre el robot, que cumple la función de “visión en primera persona” para realizar la detección del marcador. A través de este enfoque se busca emular condiciones reales de operación, análisis espacial y localización visual.

Este trabajo constituye una base sólida para el desarrollo de futuras aplicaciones en robótica autónoma, al combinar herramientas de simulación, percepción visual y control físico a través de interfaces hombre-máquina.

2 OBJETIVOS

2.1 Objetivo general

Desarrollar un entorno de simulación en PyBullet que permita controlar un robot móvil a través de un joystick físico, integrando visión por computadora con detección de marcadores ArUco para estimar la distancia entre el robot y una pared con textura visual, empleando técnicas de percepción visual y comunicación serial mediante Arduino.

2.2 Objetivos específicos

- Implementar un entorno de simulación física utilizando PyBullet con un plano de suelo, una pared texturizada con un marcador ArUco y un robot móvil con ruedas.
- Integrar una cámara virtual fija y una cámara montada en el robot para capturar imágenes desde distintas perspectivas del entorno.
- Detectar marcadores ArUco en tiempo real utilizando OpenCV y calcular la distancia entre el robot y el marcador en coordenadas 2D y 3D.
- Controlar el desplazamiento del robot mediante un joystick físico conectado a un Arduino, utilizando comunicación serial como interfaz entre el entorno físico y el virtual.
- Visualizar las distancias calculadas dentro del entorno mediante líneas y anotaciones gráficas superpuestas en la imagen.
- Optimizar el rendimiento del sistema para asegurar una simulación fluida y en tiempo real, reduciendo la carga computacional sin comprometer la funcionalidad.

3 MATERIALES Y HERRAMIENTAS

3.1 Software

- Python 3.10+: Lenguaje principal utilizado para la programación del entorno de simulación y procesamiento de imagen.
- PyBullet: Motor de simulación física en tiempo real empleado para crear el entorno 3D, simular el movimiento del robot y generar las cámaras virtuales.
- OpenCV (cv2): Biblioteca utilizada para la captura y procesamiento de imágenes, así como para la detección de marcadores ArUco.

- Visual Studio Code: Entorno de desarrollo integrado (IDE) utilizado para la escritura, depuración y ejecución del código fuente.
- Microsoft Word: Procesador de texto utilizado para la elaboración del reporte técnico.
- PySerial: Librería de comunicación serial entre Arduino y Python.

3.2 Hardware

Arduino UNO: Microcontrolador encargado de leer el estado del joystick y enviar las instrucciones de movimiento al entorno simulado.

Joystick: Controlador analógico de dos ejes utilizado como interfaz física para el desplazamiento del robot.

Cable USB tipo A a B: Para la conexión entre el Arduino y el equipo de cómputo.

3.3 Archivos utilizados

powerchair.urdf: Archivo URDF que define la geometría, propiedades físicas y articulaciones del robot simulado.

aruco1.jpeg: Imagen con el marcador ArUco utilizado como referencia visual en la pared.

4 METODOLOGÍA

La presente sección describe detalladamente el proceso llevado a cabo para el desarrollo, implementación y evaluación del entorno de simulación utilizado en este proyecto. Se abordan las etapas técnicas relacionadas con la configuración del simulador físico, la integración del modelo del robot móvil, la implementación de la comunicación con hardware externo mediante Arduino, y la incorporación de visión artificial para la detección de marcadores visuales. Cada una de estas fases fue diseñada para replicar, de manera precisa, el comportamiento de un sistema mecatrónico real dentro de un entorno virtual controlado. El enfoque adoptado permite una evaluación sistemática del rendimiento del robot ante distintas condiciones de operación, así como el análisis visual de su movimiento y orientación con respecto a un entorno artificialmente estructurado.

4.1 Preparación del entorno de simulación

Para llevar a cabo la simulación del sistema mecatrónico, se utilizó el motor físico PyBullet, una herramienta en Python diseñada para entornos de robótica, dinámica y visualización 3D. La primera etapa consistió en la instalación de las librerías necesarias para ejecutar el entorno, entre ellas: pybullet, numpy, opencv-python y pyserial. Estas bibliotecas permitieron, respectivamente, la simulación física, el procesamiento de datos numéricos, la visión por computadora y la comunicación con dispositivos externos como Arduino.

Una vez cargadas las dependencias, se procedió a configurar el entorno de simulación. Se inicializó la física con gravedad terrestre y un paso de simulación de 1/240 segundos para lograr una respuesta fluida y realista. El entorno fue complementado con la carga de un plano base utilizando el archivo plane.urdf, que actúa como el suelo de la escena.

Adicionalmente, se incorporó una pared vertical que funciona como superficie de referencia visual. Esta pared fue generada también a partir de un plane.urdf, pero rotado 90 grados sobre su eje Y y 180 grados en Z, de modo que actúe como un muro perpendicular al suelo. Sobre esta superficie se aplicó una textura personalizada con un patrón de marcador ArUco, la cual fue cargada mediante la función p.loadTexture() y aplicada con p.changeVisualShape(). Esto permitió utilizar dicha pared como punto de referencia para cálculos de posición, distancia y reconocimiento visual durante el movimiento del robot.

```
26_código_optimizado.py
1   import pybullet as p
2   import pybullet_data
3   import serial
4   import time
5   import numpy as np
6   import cv2
7
8   # --- Configuración del puerto serial ---
9   arduino = serial.Serial('COM8', 9600, timeout=0.1)
10  time.sleep(2)
11
12  # --- Inicialización de PyBullet ---
13  p.connect(p.GUI)
14  p.setAdditionalSearchPath(pybullet_data.getDataPath())
15  p.resetSimulation()
16  p.setGravity(0, 0, -9.81)
17  p.setTimeStep(1 / 240)
18  p.loadURDF("plane.urdf")
```

Ilustración 1. Código del entorno

4.2 Diseño y carga del modelo del robot

Para la simulación del sistema móvil se utilizó un modelo de silla de ruedas motorizada, representando un robot diferencial con cuatro ruedas, dos de las cuales están directamente impulsadas. Este modelo se encuentra descrito mediante un archivo URDF (Unified Robot Description Format), el cual define su geometría, propiedades físicas y articulaciones. Dicho archivo fue cargado en el entorno de simulación PyBullet mediante la instrucción loadURDF, ubicándolo en una posición inicial cercana al origen del mundo virtual.

El robot cuenta con los siguientes elementos relevantes para la simulación:

- Cuerpo base: Representa la estructura principal del robot, con propiedades físicas como masa, fricción lateral y amortiguamiento.
- Ruedas motrices: Controladas mediante actuadores virtuales bajo el modo VELOCITY_CONTROL, lo que permite el movimiento autónomo en respuesta a comandos.
- Punto de montaje de cámara: Se utiliza un enlace específico (camera_link) para simular una cámara frontal montada en el robot, desde la cual se emula la visión del entorno.

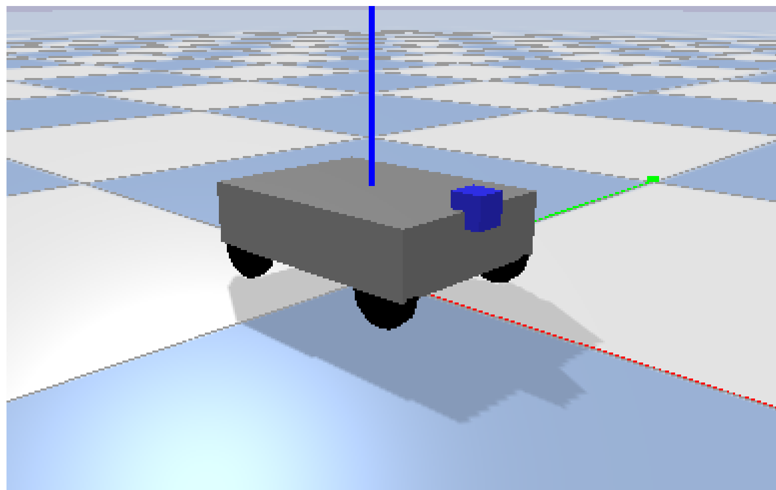

Ilustración 2. Visualización de "powerchair.urdf"

Durante la carga, se aplicaron parámetros de dinámica personalizados a cada componente del modelo. Esto incluyó la reducción de la inercia y el incremento del rozamiento para evitar comportamientos no deseados como el deslizamiento excesivo o movimientos inestables. Asimismo, se ajustaron las propiedades de las ruedas para mejorar el desempeño en giros y trayectorias rectas, asegurando una simulación más realista.

```
32    # --- Dinámica ---
33    p.changeDynamics(robot, -1,
34        mass=5,
35        localInertiaDiagonal=[0.05, 0.05, 0.02],
36        lateralFriction=1.4,
37        linearDamping=0.08,
38        angularDamping=0.12
39    )
40
41    for w in range(p.getNumJoints(robot)):
42        p.changeDynamics(robot, w,
43            lateralFriction=1.5,
44            rollingFriction=0.02,
45            spinningFriction=0.02,
46            linearDamping=0.04,
47            angularDamping=0.06,
48            restitution=0.0
49        )
50
```

Ilustración 3. Dinámica del robot

Este modelo sirve como base para todas las pruebas posteriores, permitiendo la interacción del robot con el entorno, así como su control a través de comandos externos y la observación de su comportamiento ante estímulos visuales simulados.

4.3 Comunicación con Arduino y joystick

La interacción física entre el usuario y el entorno simulado se logra mediante la integración de un joystick analógico conectado a una placa Arduino, la cual actúa como intermediario para interpretar las señales del controlador y enviarlas al entorno de simulación a través de una conexión serial.

Ilustración 4. Diagrama de conexión eléctrica

En el Arduino, se implementa un programa que lee las señales del joystick (valores analógicos de los ejes X y Y) y las convierte en comandos de texto simples (por ejemplo, "FORWARD", "BACKWARD", "LEFT", "RIGHT"). Estos comandos son enviados continuamente por el puerto serial a una tasa de 9600 baudios.

Ilustración 5. Puerto serial del Arduino IDE

En Python, se utiliza la biblioteca pyserial para establecer la comunicación entre la computadora y la placa Arduino. El programa permanece a la espera de datos entrantes desde el puerto designado (COM8 en este caso), y cuando se detecta un comando válido, se interpreta y se traduce en una instrucción de movimiento para el robot simulado.

La función encargada de interpretar los comandos (mover_carrito) aplica velocidades específicas a cada una de las ruedas del robot utilizando control de velocidad (VELOCITY_CONTROL) con fuerza máxima limitada. Esto permite movimientos diferenciales, permitiendo avanzar, retroceder o girar según la dirección del joystick.

```
54    # --- Movimiento y cámara del robot ---
55    v_forward = 85
56    v_turn = 85
57    v_stop = 0
58    force_value = 300
59    camera_link = 4
60    width, height = 320, 240
61    fov = 90
62    aspect = width / height
63    near, far = 0.01, 3.0
64    chase_distance = 1.5
65    chase_height = 0.5
```

Ilustración 6. Código de movimiento del robot

Este enfoque proporciona una forma intuitiva de controlar el robot dentro del entorno virtual y representa una aproximación realista a escenarios de teleoperación o conducción asistida mediante dispositivos físicos.

4.4 Configuración del entorno virtual

La simulación del entorno virtual se desarrolla utilizando la biblioteca PyBullet, que permite representar de manera física y visual un espacio tridimensional con elementos interactivos y dinámicos. Este entorno fue configurado cuidadosamente para replicar condiciones de navegación y percepción visual útiles en pruebas de robótica móvil.

Primero, se establece una superficie base utilizando el archivo plane.urdf, que actúa como el plano de rodadura sobre el cual se desplaza el robot. Posteriormente, se posiciona una pared perpendicular al plano, la cual utiliza como textura una imagen que contiene un marcador ArUco. Esta pared se representa también mediante plane.urdf, rotado 90° en el eje Y y 180° en el eje Z para ubicarlo de manera vertical frente al robot.

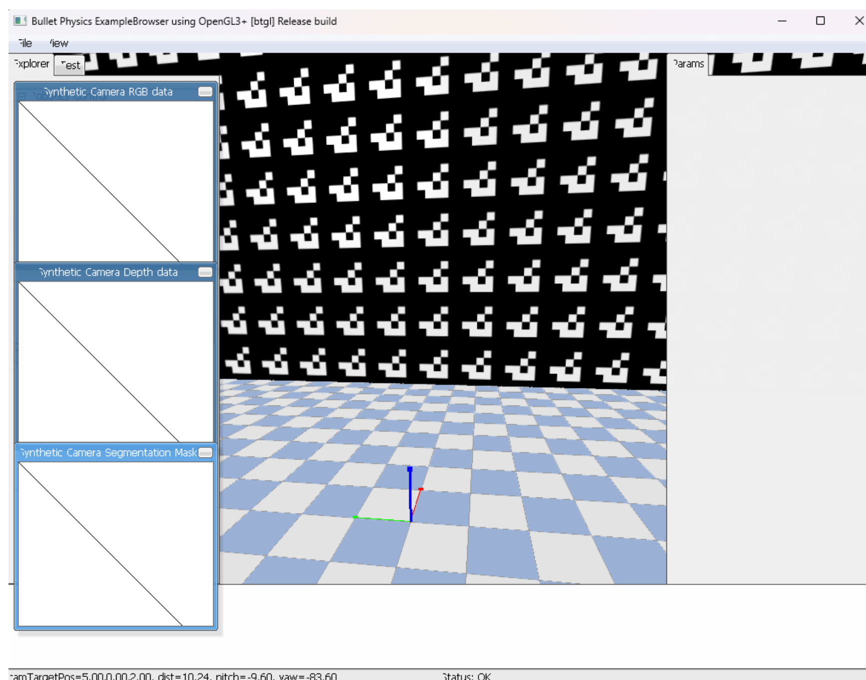

Ilustración 7. Pared de Arucos

El marcador ArUco simula un punto de referencia visual dentro del entorno. Se utiliza una textura en formato .jpeg que contiene dicho marcador, la cual se carga con la función p.loadTexture() y se aplica a la pared mediante p.changeVisualShape().

Además, se configuran cámaras virtuales dentro del entorno para capturar imágenes desde distintas perspectivas:

- Cámara montada en el robot: simula una cámara frontal que avanza junto al robot y permite realizar tareas de visión por computadora, como la detección del marcador ArUco.
- Cámara fija (de vigilancia): ubicada en un punto elevado y alejado del entorno, esta cámara permite observar desde una perspectiva global el comportamiento del robot en tiempo real.

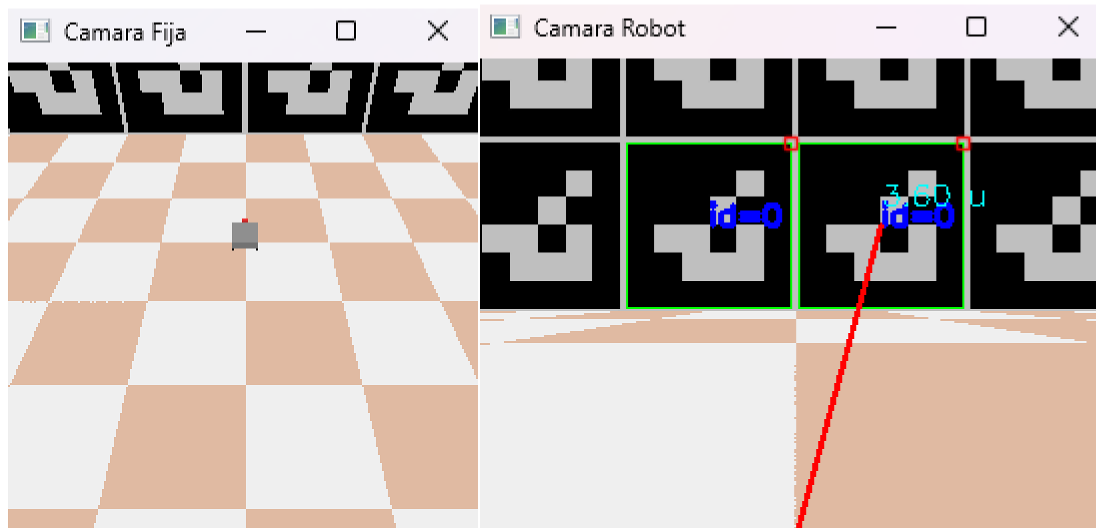

Ilustración 8. Cámaras virtuales configuradas

Finalmente, la gravedad, el paso del tiempo (1/240 s), y las propiedades físicas del entorno (como fricción y restitución) fueron ajustadas para mantener estabilidad en la simulación y realismo en la dinámica del movimiento.

4.5 Implementación del sistema de visión y detección ArUco

Una de las funcionalidades clave del sistema es la capacidad del robot para detectar marcadores visuales en su entorno mediante visión artificial. Para ello se utilizó la biblioteca OpenCV, que provee herramientas específicas para la detección de marcadores ArUco, ampliamente utilizados en aplicaciones de localización y navegación.

La cámara montada en el robot captura imágenes periódicamente desde el vínculo frontal (link) definido en el modelo URDF del robot. Esta cámara virtual se configura utilizando la

función p.getCameraImage(), en combinación con matrices de vista y proyección obtenidas mediante p.computeViewMatrix() y p.computeProjectionMatrixFOV() respectivamente. Estas matrices simulan el comportamiento de una cámara en perspectiva, considerando su posición, orientación, campo de visión y planos de recorte.

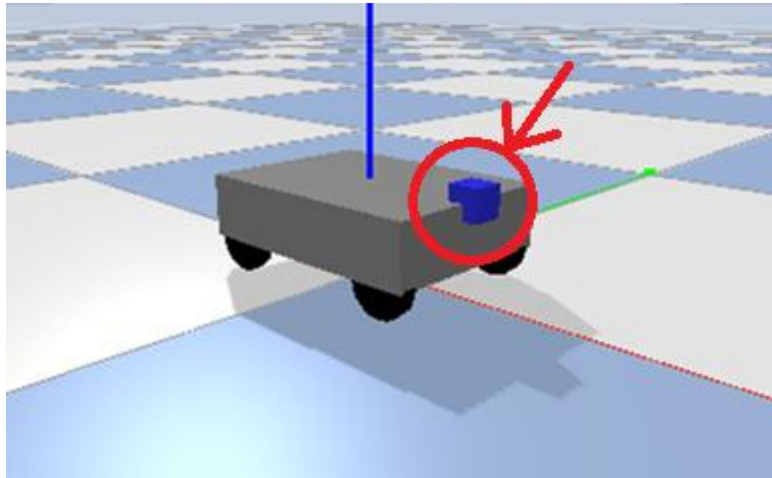

Ilustración 9. Cámara montada

Cada imagen capturada es convertida a escala de grises y posteriormente procesada por el detector ArUco. El flujo general es el siguiente:

- Captura de imagen desde PyBullet.
- Conversión a formato OpenCV (RGB y escala de grises).
- Detección de marcadores con cv2.aruco.ArucoDetector().
- Dibujo del marcador detectado en la imagen con cv2.aruco.drawDetectedMarkers().
- Cálculo de la posición del marcador en la imagen (coordenadas del centro).
- Estimación de la distancia entre el robot y el marcador en dos planos:
    - 2D: Distancia en píxeles entre el centro de la imagen y el marcador.
    - 3D: Distancia en el mundo simulado, usando la posición absoluta del robot y la pared.

Para mejorar la comprensión visual, se añade una línea sobre la imagen entre el robot y el marcador, y también una línea tridimensional en PyBullet que conecta ambos puntos con un trazo rojo, utilizando p.addUserDebugLine().

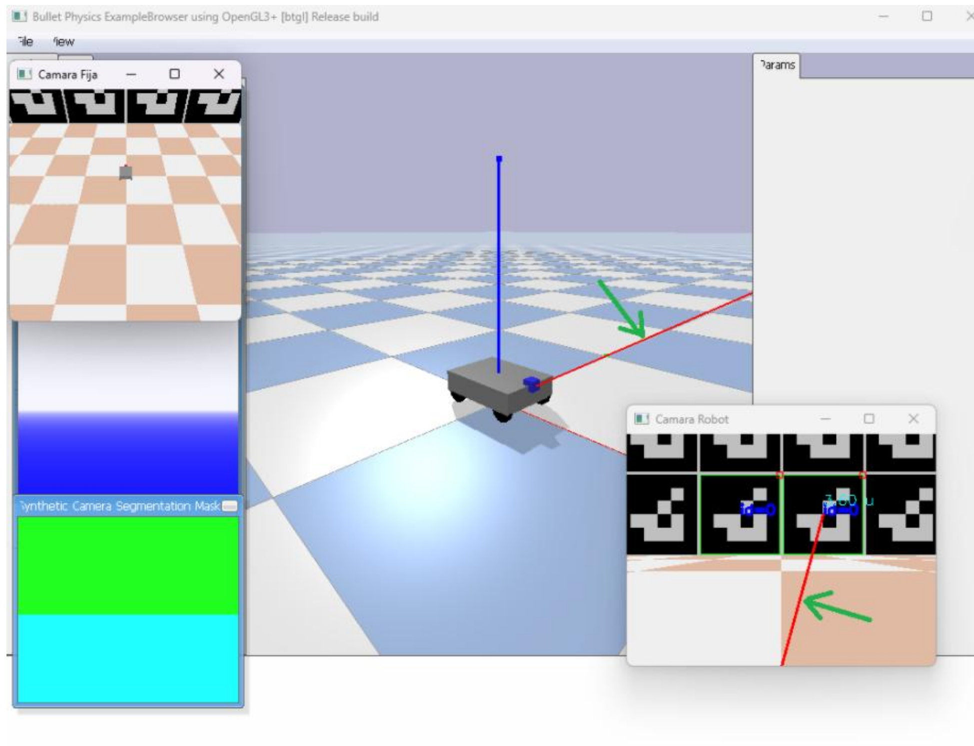

Ilustración 10. Líneas de visualización

Este sistema proporciona una retroalimentación clara y en tiempo real del entorno visual percibido por el robot, lo que sienta las bases para futuras implementaciones de navegación autónoma o localización basada en visión.

4.6 Cámaras virtuales y visualización

El sistema desarrollado hace uso de múltiples cámaras virtuales en el entorno simulado para facilitar tanto el monitoreo como la depuración del comportamiento del robot. Estas cámaras no solo permiten visualizar la escena desde diferentes perspectivas, sino que también cumplen funciones clave dentro del sistema de visión artificial.

Se implementaron tres tipos de cámaras:

1. Cámara montada en el robot (visión subjetiva)

Esta cámara está acoplada a un vínculo específico del robot, simulando una cámara física que permitiría la percepción directa del entorno. Se encuentra orientada hacia el frente del robot y proporciona una vista en primera persona.

- Esta cámara es utilizada para la detección del marcador ArUco, como se explicó en el apartado anterior.
- La imagen es capturada cada cierto número de frames para optimizar el rendimiento.

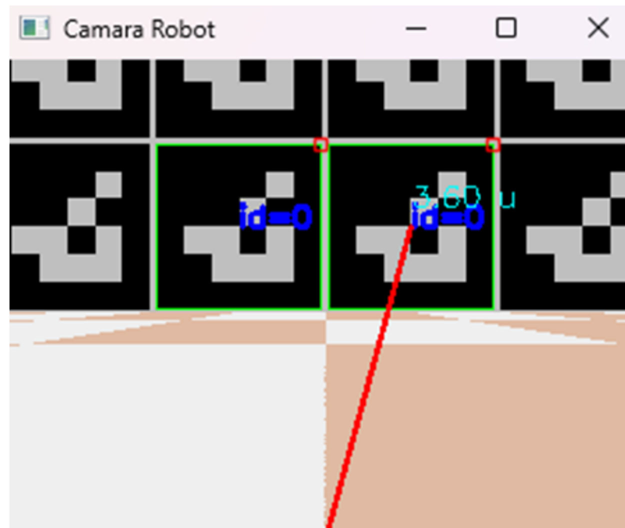

Ilustración 11. Cámara montada

2. Cámara externa tipo persecución

Además de la cámara montada, se emplea una cámara externa que sigue al robot desde atrás. Esta vista ofrece al usuario una perspectiva general del movimiento y orientación del robot en la escena.

- Se actualiza en tiempo real utilizando la función p.resetDebugVisualizerCamera(), ajustando su posición en función de la ubicación y orientación del robot.
- Esta cámara facilita el seguimiento visual del desplazamiento y posibles colisiones.

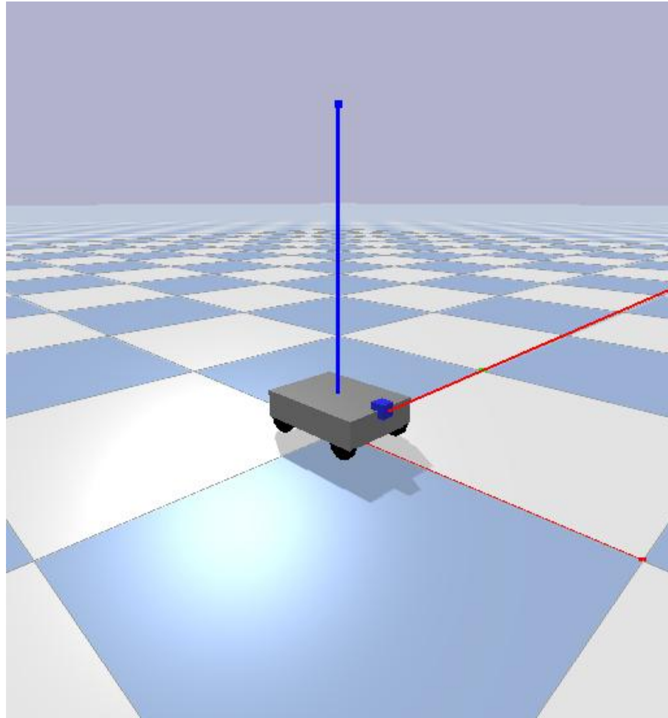

Ilustración 12. Cámara de seguimiento

3. Cámara fija tipo vigilancia

Con el fin de obtener una vista general del entorno, se agregó una cámara estática y elevada, ubicada en una posición estratégica detrás del robot, mirando hacia el centro del área de trabajo.

- Se configuró mediante las funciones p.computeViewMatrix() y p.computeProjectionMatrixFOV() para definir su orientación y parámetros ópticos.
- Su salida se muestra en una ventana separada mediante OpenCV, permitiendo observar desde una perspectiva de "seguridad" o supervisión.

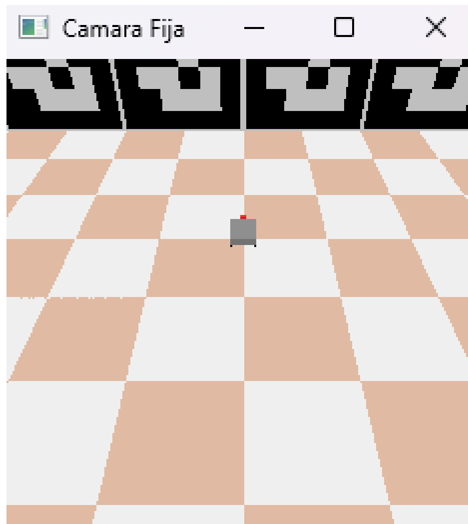

Ilustración 13. Cámara de vigilancia/fija

Cada cámara virtual contribuye al monitoreo de diferentes aspectos del sistema y complementa el análisis visual del desempeño del robot, ofreciendo al usuario una experiencia completa e inmersiva en el entorno simulado.

5 CÓDIGO

5.1 Explicación código Arduino

```
// --- Pines del joystick ---
int ejeX = A4;  // Pin conectado a VRX
int ejeY = A5;  // Pin conectado a VRY

int valorX = 0;
int valorY = 0;

void setup() {
  Serial.begin(9600);
}

void loop() {
  // Leer los valores analógicos
  valorX = analogRead(ejeX);
  valorY = analogRead(ejeY);
```

Ilustración 14. Fragmento 1 Arduino

```
  // --- Detección de dirección ---
  if (valorY > 600) {
    Serial.println("FORWARD");
  }
  else if (valorY < 400) {
    Serial.println("BACKWARD");
  }
  else if (valorX > 600) {
    Serial.println("LEFT");
  }
  else if (valorX < 400) {
    Serial.println("RIGHT");
  }
  else {
    Serial.println("STOP");
  }

  delay(200); // Pequeña pausa para evitar lecturas muy rápidas
}
```

Ilustración 15. Framento 2 Arduino

Este código de Arduino permite interpretar los movimientos de un joystick analógico para enviar comandos de dirección a través del puerto serial. Se definen los pines analógicos A4 y A5 para leer los ejes X e Y del joystick, y en el loop() se ejecuta la lectura continua de estos valores mediante analogRead(). Según los valores obtenidos (entre 0 y 1023), el programa determina si el joystick se está moviendo hacia adelante, atrás, izquierda o derecha comparando los valores con umbrales (mayores a 600 o menores a 400).

Por ejemplo, si el eje Y es mayor a 600, se interpreta como un movimiento hacia adelante ("FORWARD"), mientras que si es menor a 400, indica un retroceso ("BACKWARD"). Lo mismo aplica para el eje X con los comandos de giro. Si el joystick no se mueve lo suficiente en ninguna dirección (es decir, permanece en una posición central), se envía el comando "STOP". Además, se incluye un delay(200) para evitar lecturas excesivamente rápidas que podrían causar inestabilidad en la detección. Este script es útil para controlar un sistema robótico mediante señales simples y claras enviadas por comunicación serial.

5.2 Explicación código Python

```python
26_código_optimizado.py > ...
1   import pybullet as p
2   import pybullet_data
3   import serial
4   import time
5   import numpy as np
6   import cv2
7
8   # --- Configuración del puerto serial ---
9   arduino = serial.Serial('COM8', 9600, timeout=0.1)
10  time.sleep(2)
```

Ilustración 16. Fragmento 1 Python

Se importan las librerías necesarias para la simulación física (pybullet), procesamiento de imagen (cv2, numpy) y comunicación con el hardware (serial, time). Además, se configura el puerto serial para recibir comandos desde un joystick conectado a un Arduino.

```python
12  # --- Inicialización de PyBullet ---
13  p.connect(p.GUI)
14  p.setAdditionalSearchPath(pybullet_data.getDataPath())
15  p.resetSimulation()
16  p.setGravity(0, 0, -9.81)
17  p.setTimeStep(1 / 240)
18  p.loadURDF("plane.urdf")
```

Ilustración 17. Fragmento 2 Python

Este bloque establece la conexión con la GUI de PyBullet, configura la gravedad, el paso de simulación (para mayor precisión) y carga un plano como superficie base donde se moverá el robot.

```python
20  # --- Pared con textura ArUco ---
21  texture_path = "aruco1.jpeg"
22  texture_id = p.loadTexture(texture_path)
23  wall_id = p.loadURDF("plane.urdf",
24                       basePosition=[3, 0, 2],
25                       baseOrientation=p.getQuaternionFromEuler([0, 1.5708, 3.1416]))
26  p.changeVisualShape(wall_id, -1, textureUniqueId=texture_id)
```

Ilustración 18. Fragmento 3 Python

Se carga una imagen (aruco1.jpeg) como textura y se aplica a un plano vertical, simulando una pared con un marcador ArUco pegado. Este plano está rotado 90° en Y y 180° en Z para que actúe como una pared frente al robot.

```
# --- Dinámica ---
p.changeDynamics(robot, -1,
    mass=5,
    localInertiaDiagonal=[0.05, 0.05, 0.02],
    lateralFriction=1.4,
    linearDamping=0.08,
    angularDamping=0.12
)

for w in range(p.getNumJoints(robot)):
    p.changeDynamics(robot, w,
        lateralFriction=1.5,
        rollingFriction=0.02,
        spinningFriction=0.02,
        linearDamping=0.04,
        angularDamping=0.06,
        restitution=0.0
    )
```

Ilustración 19. Fragmento 4 Python

Se carga el robot desde un archivo URDF, se identifica qué uniones son ruedas y se ajustan los parámetros de física (como fricción, masa e inercia) tanto del cuerpo principal como de las ruedas, mejorando el realismo del movimiento y la estabilidad.

```
# --- Cámara externa PyBullet ---
p.resetDebugVisualizerCamera(2.5, 90, -30, [0.5, 0, 0.1])
```

Ilustración 20. Fragmento 5 Python

Se posiciona la cámara principal del entorno visual 3D para ver desde arriba al robot, útil durante el desarrollo para verificar el estado general de la simulación.

```python
54  # --- Movimiento y cámara del robot ---
55  v_forward = 85
56  v_turn = 85
57  v_stop = 0
58  force_value = 300
59  camera_link = 4
60  width, height = 320, 240
61  fov = 90
62  aspect = width / height
63  near, far = 0.01, 3.0
64  chase_distance = 1.5
65  chase_height = 0.5
```

Ilustración 21. Fragmento 6 Python

Se definen parámetros clave para el movimiento del robot (velocidades, fuerza, etc.), así como los parámetros ópticos de la cámara montada en el robot (ángulo de visión, resolución, plano de visión, etc.). También se ajusta la cámara tipo persecución (3ª persona).

```python
67  # --- Cámara fija ---
68  fixed_cam_pos = [-4, 0, 3]
69  look_at_pos = [0, 0, -1]
70  up_vector = [0, 0, 1]
71  fixed_view_matrix = p.computeViewMatrix(fixed_cam_pos, look_at_pos, up_vector)
72  fixed_projection_matrix = p.computeProjectionMatrixFOV(60, 1.0, 0.1, 10.0)
73
```

Ilustración 22. Fragmento 7 Python

Se configura una cámara externa estática con vista elevada desde atrás. Esta vista fija simula una cámara de seguridad que observa toda el área de trabajo.

```python
74  # --- Movimiento ---
75  def mover_carrito(direccion):
76      if direccion == "FORWARD": vel = [v_forward] * 4
77      elif direccion == "BACKWARD": vel = [-v_forward] * 4
78      elif direccion == "LEFT": vel = [-v_turn, v_turn, -v_turn, v_turn]
79      elif direccion == "RIGHT": vel = [v_turn, -v_turn, v_turn, -v_turn]
80      else: vel = [v_stop] * 4
81
82      for i, w in enumerate(wheels):
83          p.setJointMotorControl2(robot, w, p.VELOCITY_CONTROL,
84                                  targetVelocity=vel[i], force=force_value)
```

Ilustración 23. Fragmento 8 Python

Se define una función que traduce comandos como "FORWARD", "LEFT", etc., en velocidades individuales para las 4 ruedas. Esto permite que el robot avance, retroceda o gire de acuerdo con la entrada recibida desde el joystick vía Arduino.

```
86    # --- Detector ArUco ---
87    aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
88    parameters = cv2.aruco.DetectorParameters()
89    detector = cv2.aruco.ArucoDetector(aruco_dict, parameters)
90
```

Ilustración 24. Fragmento 9 Python

Se configura el diccionario de marcadores ArUco que se usarán para detección visual. Aquí se utiliza el tipo DICT_4X4_50, adecuado para entornos de prueba pequeños. Se crea el detector con parámetros estándar.

```
91    # --- Simulación ---
92    frame_counter = 0
93    last_command = ""
94
95    while True:
96        if arduino.in_waiting > 0:
97            comando = arduino.readline().decode().strip()
98            if comando and comando != last_command:
99                mover_carrito(comando)
100               last_command = comando
101
102       base_pos, base_ori = p.getBasePositionAndOrientation(robot)
103       rot_mat = p.getMatrixFromQuaternion(base_ori)
104       forward_vec = np.array([rot_mat[0], rot_mat[3], rot_mat[6]])
105       chase_pos = np.array(base_pos) - forward_vec * chase_distance
106       chase_pos[2] += chase_height
107       p.resetDebugVisualizerCamera(1.8, 45, -25, base_pos)
108
109       if frame_counter % 5 == 0:
110           try:
111               state = p.getLinkState(robot, camera_link)
112               cam_pos = state[0]
113               cam_ori = p.getMatrixFromQuaternion(state[1])
114               target_pos = [
115                   cam_pos[0] + 0.2 * cam_ori[0],
116                   cam_pos[1] + 0.2 * cam_ori[3],
117                   cam_pos[2] + 0.2 * cam_ori[6],
118               ]
119               view_matrix = p.computeViewMatrix(cam_pos, target_pos, [0, 0, 1])
120               projection_matrix = p.computeProjectionMatrixFOV(fov, aspect, near, far)
121               img = p.getCameraImage(width, height, view_matrix, projection_matrix)
122               rgb = np.reshape(img[2], (height, width, 4))[:, :, :3].astype(np.uint8)
123
```

Ilustración 25. Fragmento 10 Python

```python
            gray = cv2.cvtColor(rgb, cv2.COLOR_BGR2GRAY)
            corners, ids, _ = detector.detectMarkers(gray)
            if ids is not None:
                cv2.aruco.drawDetectedMarkers(rgb, corners, ids)
                aruco_pos = np.mean(corners[0][0], axis=0)
                robot_pos_2d = [width // 2, height]
                dist_2d = np.linalg.norm(robot_pos_2d - aruco_pos)
                aruco_world_pos = np.array([3, 0, 2])
                robot_world_pos = np.array(base_pos)
                dist_3d = np.linalg.norm(robot_world_pos - aruco_world_pos)
                cv2.line(rgb, tuple(robot_pos_2d), tuple(aruco_pos.astype(int)), (0, 0, 255), 2)
                cv2.putText(rgb, f"{dist_3d:.2f} u", (int(aruco_pos[0]), int(aruco_pos[1]) - 10),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 0), 1)
                p.addUserDebugLine(aruco_world_pos, robot_world_pos, [1, 0, 0], 2, lifeTime=0.1)

            cv2.imshow("Camara Robot", rgb)
        except Exception as e:
            print("Error camara robot:", e)

    if frame_counter % 10 == 0:
        img_fixed = p.getCameraImage(240, 240, fixed_view_matrix, fixed_projection_matrix)
        rgb_fixed = np.reshape(img_fixed[2], (240, 240, 4))[:, :, :3].astype(np.uint8)
        cv2.imshow("Camara Fija", rgb_fixed)

    if cv2.waitKey(1) & 0xFF == 27:
        break

    p.stepSimulation()
    frame_counter += 1
    time.sleep(1 / 240)
```

Ilustración 26. Fragmento 11 Python

Este bloque contiene todo el ciclo principal de simulación:

- Lectura de comandos del joystick: Se leen comandos enviados desde el Arduino y se ejecuta el movimiento correspondiente.
- Cámara de persecución del robot: Calcula la posición detrás del robot para una cámara virtual que lo sigue.
- Captura y análisis de la cámara del robot: Cada 5 ciclos, se toma una imagen desde la cámara montada en el robot. Luego, se procesa para detectar marcadores ArUco y calcular distancias. Se dibujan líneas y textos en la imagen para mostrar la detección.
- Captura desde cámara fija: Cada 10 ciclos se obtiene una imagen desde la cámara fija.
- Finalización de simulación y control de bucle: Verifica si se ha presionado la tecla ESC para cerrar las ventanas de OpenCV, avanza un paso de simulación y espera el tiempo definido.

```
155    cv2.destroyAllWindows()
156    p.disconnect()
```

Ilustración 27. Fragmento 12 Python

Una vez que se rompe el bucle (por ejemplo, al presionar ESC), se cierran todas las ventanas de OpenCV y se desconecta de PyBullet limpiamente.

6 RESULTADO

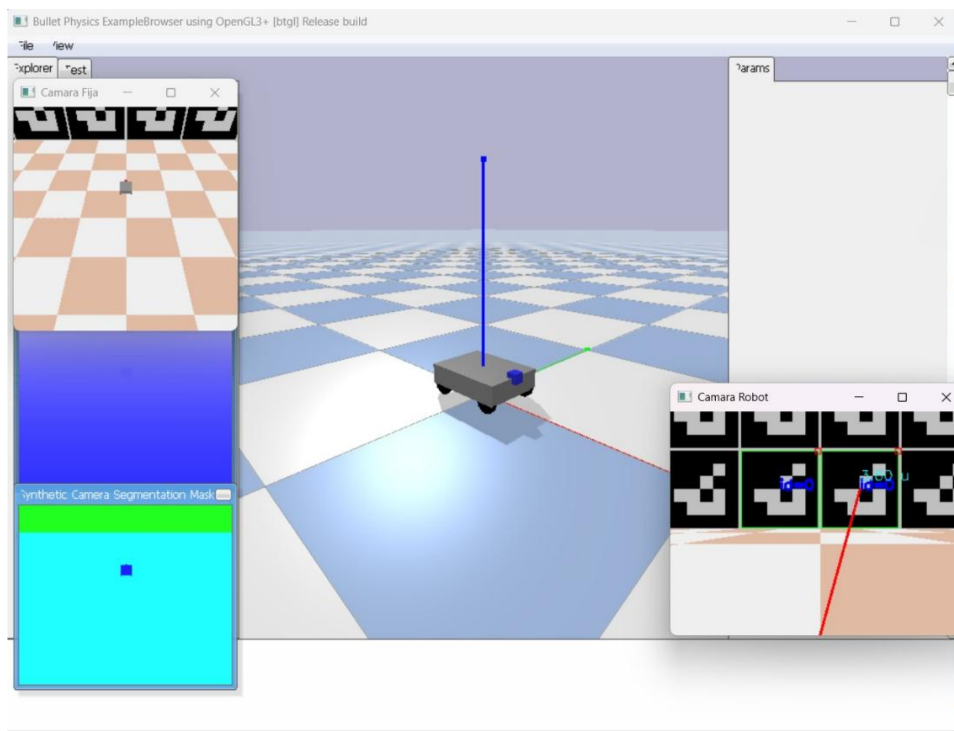

Ilustración 28. Código apenas inicia

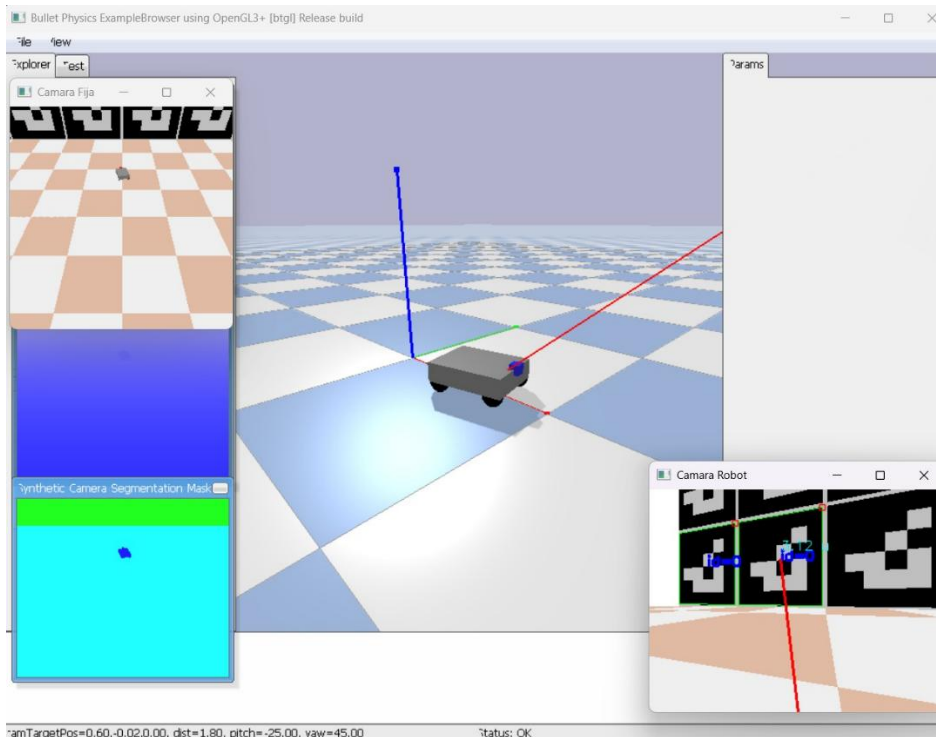

Ilustración 29. Código en uso

Durante el desarrollo del proyecto se logró implementar exitosamente un entorno de simulación utilizando PyBullet que integra hardware (joystick + Arduino) con visión computacional (detección de marcadores ArUco) en un robot móvil simulado. A continuación, se describen los principales resultados obtenidos:

6.1 Conexión funcional Arduino–PyBullet

Se estableció una comunicación serial entre el Arduino y el entorno de simulación en Python. El Arduino, conectado a un joystick físico, fue capaz de enviar cuatro comandos distintos que representan las direcciones básicas de movimiento: adelante, atrás, izquierda y derecha. Estos comandos fueron interpretados en tiempo real por el script en Python y traducidos en movimiento para el robot simulado en PyBullet.

6.2 Textura ArUco como pared

Inicialmente se intentó aplicar una imagen de ArUco como textura sobre una geometría 3D completa (por ejemplo, un cubo o una caja), pero esta envolvía toda la figura y distorsionaba

la imagen. La solución fue utilizar un objeto plano (plane.urdf) como "pared vertical" y rotarlo para que simule una superficie vertical frente al robot. Al aplicar la textura de los marcadores ArUco a este plano, se obtuvo una visualización clara y funcional para el reconocimiento.

6.3 Detección de marcador ArUco

Mediante la librería OpenCV y su módulo cv2.aruco, se implementó un sistema de visión artificial que detecta marcadores ArUco visibles por la cámara del robot. El sistema fue capaz de:

- Identificar correctamente la ID del marcador.
- Calcular la distancia en 2D (plano de la imagen) y en 3D (espacio real de simulación) entre el centro del robot y el centro del marcador.

6.4 Cálculo y visualización de distancia

Se implementó una visualización adicional que traza una línea roja desde el robot hasta el marcador detectado. Esta línea permite verificar visualmente la distancia calculada en 3D y se actualiza dinámicamente en cada ciclo de simulación. También se sobrepuso la distancia en unidades sobre la imagen RGB para retroalimentación visual.

6.5 Implementación de cámaras múltiples

Se incorporaron tres vistas diferentes para mejorar la supervisión del entorno:

- Cámara principal del simulador (debug visualizer): sigue automáticamente al robot desde un ángulo de 45° por detrás y ligeramente elevado.
- Cámara del robot (simulada): montada sobre un link específico del URDF, simula la visión que tendría una cámara montada físicamente.
- Cámara fija (vigilancia): colocada detrás del escenario y elevada, apunta hacia la zona de trabajo, permitiendo supervisar el comportamiento general del entorno y el movimiento del robot.

6.6 Visualización en tiempo real

Se desplegaron las imágenes obtenidas de la cámara fija y la del robot en ventanas separadas utilizando OpenCV. Esto permitió una mejor comparación entre el entorno general y lo que "ve" el robot, además de validar en tiempo real la detección y el cálculo de distancia a los marcadores.

6.7 Pruebas realizadas

Se realizaron pruebas desplazando el robot hacia diferentes posiciones respecto al plano con los marcadores ArUco. En todos los casos, el sistema respondió correctamente, detectando el marcador y mostrando la información correspondiente. Se comprobaron diferentes distancias y orientaciones, observando cómo se actualizaban la línea visual, los datos y las cámaras.

7 CONCLUSIONES

El presente proyecto permitió integrar conocimientos de simulación robótica, visión por computadora y sistemas embebidos, logrando la implementación exitosa de un entorno interactivo en PyBullet controlado mediante un joystick físico conectado a un Arduino.

Uno de los principales aprendizajes fue la correcta estructuración del entorno virtual, enfrentando desafíos como la aplicación de texturas (en este caso, patrones ArUco) sobre objetos tridimensionales. A pesar de las limitaciones iniciales para plasmar imágenes sobre un cubo o superficie envolvente, se encontró una solución funcional y eficiente al utilizar planos verticales como paredes texturizadas, permitiendo un reconocimiento preciso del marcador por parte del sistema de visión.

La incorporación del módulo ArUco de OpenCV demostró ser una herramienta confiable para la detección y localización de marcadores, lo cual permitió calcular distancias relativas en dos y tres dimensiones entre el robot y el objetivo. Adicionalmente, se integró una representación visual clara de dicha distancia mediante una línea trazada en el entorno de simulación.

La utilización de múltiples cámaras —una fija, una montada en el robot y otra externa de supervisión— enriqueció considerablemente la visualización del comportamiento del sistema, facilitando la validación del movimiento, la detección visual y el seguimiento del robot en todo momento.

El uso del joystick físico como interfaz de control añadió un componente de interacción realista y práctica, conectando el entorno físico con la simulación, lo cual refuerza la comprensión de los conceptos de control y percepción en robótica móvil.

En resumen, se alcanzaron los objetivos propuestos y se logró un entorno funcional, versátil y extensible, el cual puede servir como base para futuros trabajos de investigación o desarrollo más complejos en áreas como navegación autónoma, mapeo o visión avanzada.

8 ANEXOS

8.1 Código Arduino

```
// --- Pines del joystick ---

int ejeX = A4;  // Pin conectado a VRX

int ejeY = A5;  // Pin conectado a VRY


int valorX = 0;

int valorY = 0;


void setup() {

  Serial.begin(9600);

}


void loop() {
```

```
 // Leer los valores analógicos
 valorX = analogRead(ejeX);
 valorY = analogRead(ejeY);

 // --- Detección de dirección ---
 if (valorY > 600) {
  Serial.println("FORWARD");
 }
 else if (valorY < 400) {
  Serial.println("BACKWARD");
 }
 else if (valorX > 600) {
  Serial.println("LEFT");
 }
 else if (valorX < 400) {
  Serial.println("RIGHT");
 }
 else {
  Serial.println("STOP");
 }

 delay(200); // Pequeña pausa para evitar lecturas muy rápidas
}
```

8.2 Código Python

```
import pybullet as p
```

```
import pybullet_data

import serial

import time

import numpy as np

import cv2


# --- Configuración del puerto serial ---

arduino = serial.Serial('COM8', 9600, timeout=0.1)

time.sleep(2)


# --- Inicialización de PyBullet ---

p.connect(p.GUI)

p.setAdditionalSearchPath(pybullet_data.getDataPath())

p.resetSimulation()

p.setGravity(0, 0, -9.81)

p.setTimeStep(1 / 240)

p.loadURDF("plane.urdf")


# --- Pared con textura ArUco ---

texture_path = "aruco1.jpeg"

texture_id = p.loadTexture(texture_path)

wall_id = p.loadURDF("plane.urdf",

            basePosition=[3, 0, 2],

            baseOrientation=p.getQuaternionFromEuler([0, 1.5708, 3.1416]))

p.changeVisualShape(wall_id, -1, textureUniqueId=texture_id)
```

```
# --- Robot ---
robot = p.loadURDF("powerchair.urdf", [0, 0, 0.02], useFixedBase=False)
wheels = [0, 1, 2, 3]

# --- Dinámica ---
p.changeDynamics(robot, -1,
   mass=5,
   localInertiaDiagonal=[0.05, 0.05, 0.02],
   lateralFriction=1.4,
   linearDamping=0.08,
   angularDamping=0.12
)

for w in range(p.getNumJoints(robot)):
   p.changeDynamics(robot, w,
      lateralFriction=1.5,
      rollingFriction=0.02,
      spinningFriction=0.02,
      linearDamping=0.04,
      angularDamping=0.06,
      restitution=0.0
   )

# --- Cámara externa PyBullet ---
p.resetDebugVisualizerCamera(2.5, 90, -30, [0.5, 0, 0.1])
```

```
# --- Movimiento y cámara del robot ---
v_forward = 85
v_turn = 85
v_stop = 0
force_value = 300
camera_link = 4
width, height = 320, 240
fov = 90
aspect = width / height
near, far = 0.01, 3.0
chase_distance = 1.5
chase_height = 0.5

# --- Cámara fija ---
fixed_cam_pos = [-4, 0, 3]
look_at_pos = [0, 0, -1]
up_vector = [0, 0, 1]
fixed_view_matrix = p.computeViewMatrix(fixed_cam_pos, look_at_pos, up_vector)
fixed_projection_matrix = p.computeProjectionMatrixFOV(60, 1.0, 0.1, 10.0)

# --- Movimiento ---
def mover_carrito(direccion):
    if direccion == "FORWARD": vel = [v_forward] * 4
    elif direccion == "BACKWARD": vel = [-v_forward] * 4
    elif direccion == "LEFT": vel = [-v_turn, v_turn, -v_turn, v_turn]
    elif direccion == "RIGHT": vel = [v_turn, -v_turn, v_turn, -v_turn]
```

```python
    else: vel = [v_stop] * 4

    for i, w in enumerate(wheels):
        p.setJointMotorControl2(robot, w, p.VELOCITY_CONTROL,
                                targetVelocity=vel[i], force=force_value)

# --- Detector ArUco ---
aruco_dict = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_4X4_50)
parameters = cv2.aruco.DetectorParameters()
detector = cv2.aruco.ArucoDetector(aruco_dict, parameters)

# --- Simulación ---
frame_counter = 0
last_command = ""

while True:
    if arduino.in_waiting > 0:
        comando = arduino.readline().decode().strip()
        if comando and comando != last_command:
            mover_carrito(comando)
            last_command = comando

    base_pos, base_ori = p.getBasePositionAndOrientation(robot)
    rot_mat = p.getMatrixFromQuaternion(base_ori)
    forward_vec = np.array([rot_mat[0], rot_mat[3], rot_mat[6]])
    chase_pos = np.array(base_pos) - forward_vec * chase_distance
```

```
chase_pos[2] += chase_height
p.resetDebugVisualizerCamera(1.8, 45, -25, base_pos)

if frame_counter % 5 == 0:
  try:
    state = p.getLinkState(robot, camera_link)
    cam_pos = state[0]
    cam_ori = p.getMatrixFromQuaternion(state[1])
    target_pos = [
      cam_pos[0] + 0.2 * cam_ori[0],
      cam_pos[1] + 0.2 * cam_ori[3],
      cam_pos[2] + 0.2 * cam_ori[6],
    ]
    view_matrix = p.computeViewMatrix(cam_pos, target_pos, [0, 0, 1])
    projection_matrix = p.computeProjectionMatrixFOV(fov, aspect, near, far)
    img = p.getCameraImage(width, height, view_matrix, projection_matrix)
    rgb = np.reshape(img[2], (height, width, 4))[:, :, :3].astype(np.uint8)

    gray = cv2.cvtColor(rgb, cv2.COLOR_BGR2GRAY)
    corners, ids, _ = detector.detectMarkers(gray)
    if ids is not None:
      cv2.aruco.drawDetectedMarkers(rgb, corners, ids)
      aruco_pos = np.mean(corners[0][0], axis=0)
      robot_pos_2d = [width // 2, height]
      dist_2d = np.linalg.norm(robot_pos_2d - aruco_pos)
      aruco_world_pos = np.array([3, 0, 2])
```

```
            robot_world_pos = np.array(base_pos)
            dist_3d = np.linalg.norm(robot_world_pos - aruco_world_pos)
            cv2.line(rgb, tuple(robot_pos_2d), tuple(aruco_pos.astype(int)), (0, 0, 255), 2)
            cv2.putText(rgb, f"{dist_3d:.2f} u", (int(aruco_pos[0]), int(aruco_pos[1]) - 10),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 0), 1)
            p.addUserDebugLine(aruco_world_pos, robot_world_pos, [1, 0, 0], 2, lifeTime=0.1)

        cv2.imshow("Camara Robot", rgb)
      except Exception as e:
        print("Error camara robot:", e)

    if frame_counter % 10 == 0:
      img_fixed = p.getCameraImage(240, 240, fixed_view_matrix, fixed_projection_matrix)
      rgb_fixed = np.reshape(img_fixed[2], (240, 240, 4))[:, :, :3].astype(np.uint8)
      cv2.imshow("Camara Fija", rgb_fixed)

    if cv2.waitKey(1) & 0xFF == 27:
      break

    p.stepSimulation()
    frame_counter += 1
    time.sleep(1 / 240)

cv2.destroyAllWindows()
p.disconnect()
```

9 REFERENCIAS

- Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools. https://opencv.org/

- PyBullet. (2023). PyBullet Physics SDK Documentation. https://pybullet.org/

- Intel. (2023). ArUco Marker Detection with OpenCV. OpenCV Documentation. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

- Python Software Foundation. (2023). Python Language Reference, version 3.10. https://www.python.org/

- NumPy Developers. (2023). NumPy: Fundamental Package for Scientific Computing with Python. https://numpy.org/

- Arduino. (2023). Arduino Uno Technical Reference. https://www.arduino.cc/