Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Tecnológico de Monterrey

Implementación de robótica inteligente
TE3002B, Grupo 501

Visual control of a quasi-static robot

Elaborado Por:

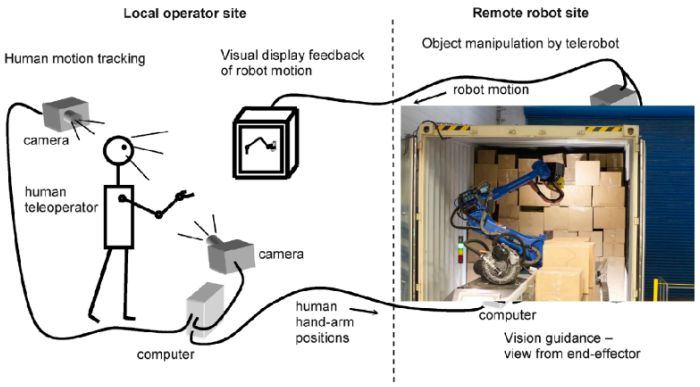
Emilio Rizo De la Mora	| A01721612
Jose Pablo Cedano Serna	| A00832019
Luis Antonio Zermeño De Gorordo	| A01781835
Omar Flores Sanchéz	| A01383530
Luis Mario Lozoya Chairez	| A00833364
Jorge Axel Castruita Bretado	| A00832843
Rodrigo Escandón López Guerrero	| A01704287

Dr. Luis Alberto Muñoz

1. Introducción

Este proyecto se centra en la simulación de un brazo robótico de 6 grados de libertad controlado mediante gestos de la mano, utilizando tecnologías de visión por computadora como OpenCV y MediaPipe en Python y renderizamos en Processsing el brazo robótico en un entorno 3D. A través de una cámara web, los movimientos y las interacciones del brazo robótico se determinan mediante posición y gestos de la mano detectados en tiempo real.

El brazo robótico puede moverse en respuesta a gestos, como "pulgar arriba" y "pulgar abajo", y es capaz de detectar y responder a la presencia de un cubo en la escena. La comunicación entre el cliente (Python) y el servidor (Processing) se realiza mediante TCP/IP.



2. Características

Renderización 3D: Utiliza Processing con OpenGL para renderizar formas en 3D, proporcionando una visualización realista del brazo robótico y su entorno.

Cinemática Inversa (IK): Calcula la posición y orientación del brazo robótico, permitiendo movimientos precisos y realistas en respuesta a los gestos de la mano.

Comunicación Cliente-Servidor: Recibe datos de gestos de la mano desde un cliente Python a través de TCP/IP, asegurando una transmisión eficiente y en tiempo real de las instrucciones de movimiento.

Detección de Gestos: Emplea OpenCV y MediaPipe para detectar gestos de la mano, como "pulgar arriba", "pulgar abajo" y “cerrar_puño” , y traducir estos gestos en movimientos del brazo robótico.

Interacción en Tiempo Real: El brazo robótico puede moverse y detectar si está tocando un cubo en la escena.

3. Instalación y uso

Para poner en marcha este proyecto y comenzar a experimentar con la simulación del brazo robótico controlado por gestos de la mano, siga los pasos detallados a continuación.

3.1. Prerrequisitos

Asegúrese de tener instalados los siguientes programas y bibliotecas en su sistema:

- Processing: Se puede descargar desde Processing.org.
- Python 3: Disponible en Python.org.
- Paquetes de Python:
    - opencv-python
    - mediapipe
    - numpy

3.2. Clonar el Repositorio

1. Abra una terminal o línea de comandos.
2. Clone el repositorio del proyecto desde GitHub ejecutando el siguiente comando:

```
git clone
https://github.com/jpcedano/Visuocontrol-of-a-quasi-static-robot.git
```

3. Cambie al directorio del proyecto clonado:

```
cd Visuocontrol-of-a-quasi-static-robot
```

3.3 Configuración de Python

1. Instale los paquetes de Python necesarios ejecutando el siguiente comando en la terminal:

```
pip install opencv-python mediapipe numpy
```

3.4 Ejecución del Cliente Python

1. Inicie el cliente Python, que se encargará de capturar los gestos de la mano. Desde la terminal, ejecute el siguiente script:

```
python hand_detector.py
```

3.5 Ejecución del Sketch de Processing

1. Abra el sketch de Processing ubicado en el directorio del proyecto.
2. Ejecute el sketch en el entorno de desarrollo de Processing.

3.6 Controles

- Mouse Drag: Arrastre el mouse en la ventana de Processing para rotar la vista de la cámara.
- Gestos de la mano: Use gestos como "pulgar arriba" y "pulgar abajo" con la mano derecha para controlar la posición en el eje Z del brazo robótico y usa la mano izquierda para controlar la posición del robot y cerrar la mano para agarrar el objeto.

4. Código Fuente

El código fuente del proyecto se divide en dos partes principales: el cliente Python (hand_detector.py) y el sketch de Processing (robotic_arm.pde). A continuación se detallan los códigos y su funcionamiento.

4.1. Cliente Python (hand_detector.py)

Este script se encarga de detectar gestos de la mano utilizando OpenCV y MediaPipe, y de enviar los datos al servidor (el sketch de Processing) a través de TCP/IP.

```
import cv2
import mediapipe as mp
import numpy as np
import socket

HOST = '127.0.0.1'
PORT = 65432

z=0.0

# Inicializar los módulos de MediaPipe para la detección de manos
mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

# Función para determinar el gesto del pulgar
def detect_thumb_gesture(hand_landmarks):
    thumb_tip = hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_TIP]
    thumb_ip = hand_landmarks.landmark[mp_hands.HandLandmark.THUMB_IP]
    wrist = hand_landmarks.landmark[mp_hands.HandLandmark.WRIST]

    # Calcular la diferencia vertical entre el pulgar y la muñeca
    thumb_up = thumb_tip.y < wrist.y and thumb_ip.y < wrist.y
    thumb_down = thumb_tip.y > wrist.y and thumb_ip.y > wrist.y

    if thumb_up:
```

```
        return "Thumb Up"
    elif thumb_down:
        return "Thumb Down"
    else:
        return "Neutral"

def is_hand_closed(hand_landmarks):
    """Determina si la mano está cerrada en un puño."""
    tips_ids = [mp_hands.HandLandmark.INDEX_FINGER_TIP,
                mp_hands.HandLandmark.MIDDLE_FINGER_TIP,
                mp_hands.HandLandmark.RING_FINGER_TIP,
                mp_hands.HandLandmark.PINKY_TIP]
    mcp_ids = [mp_hands.HandLandmark.INDEX_FINGER_MCP,
               mp_hands.HandLandmark.MIDDLE_FINGER_MCP,
               mp_hands.HandLandmark.RING_FINGER_MCP,
               mp_hands.HandLandmark.PINKY_MCP]

    closed = "True"
    for tip_id, mcp_id in zip(tips_ids, mcp_ids):
        if hand_landmarks.landmark[tip_id].y <
hand_landmarks.landmark[mcp_id].y:
            closed = "False"
            break

    return closed

# Función para detectar las manos en la imagen de la cámara
def detect_hands(webcam, conn):

    global z

    # Obtener las dimensiones de las ventanas
    window_width = 1200  # Ancho deseado de las ventanas
    window_height = 800  # Alto deseado de las ventanas

    # Crear una nueva ventana para mostrar la imagen de la cámara
    cv2.namedWindow('Detección de Manos', cv2.WINDOW_NORMAL)
    cv2.resizeWindow('Detección de Manos', window_width, window_height)

    # Obtener el tamaño de la ventana después de haberla creado
    actual_window_width = cv2.getWindowImageRect('Detección de Manos')[2]
    actual_window_height = cv2.getWindowImageRect('Detección de Manos')[3]
```

```
    # Establecer el tamaño de la cámara para que coincida con el tamaño de la
ventana
    webcam.set(cv2.CAP_PROP_FRAME_WIDTH, actual_window_width)
    webcam.set(cv2.CAP_PROP_FRAME_HEIGHT, actual_window_height)

    # Iniciar el bucle para capturar frames de la cámara
    with mp_hands.Hands(max_num_hands=2, min_detection_confidence=0.2,
min_tracking_confidence=0.3) as hands:
        while webcam.isOpened():
            success, image = webcam.read()
            if not success:
                print("No se pudo leer el frame de la cámara.")
                break

            image = cv2.flip(image, 1)

            # Convertir la imagen de BGR a RGB
            image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

            # Detectar las manos en la imagen
            results = hands.process(image_rgb)

            # Dibujar los landmarks de las manos
            if results.multi_hand_landmarks:
                for idx, hand_landmarks in
enumerate(results.multi_hand_landmarks):
                    # Determina si es la mano izquierda o derecha
                    hand_label =
results.multi_handedness[idx].classification[0].label

                    if hand_label == "Right":
                        mp_drawing.draw_landmarks(image, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

                        # Añade el texto de la clasificación en la imagen
                        coords = tuple(np.multiply(

[hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].x,

hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].y],
                                [image.shape[1], image.shape[0]]]).astype(int))
```

```
                        cv2.putText(image, hand_label, coords,
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2, cv2.LINE_AA)

                        # Detectar gesto del pulgar
                        gesture = detect_thumb_gesture(hand_landmarks)
                        cv2.putText(image, gesture, (coords[0], coords[1] -
30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2, cv2.LINE_AA)

                        if gesture == "Thumb Up":
                            z -= 1.0
                        elif gesture == "Thumb Down":
                            z += 1.0

                    elif hand_label == "Left":
                        mp_drawing.draw_landmarks(image, hand_landmarks,
mp_hands.HAND_CONNECTIONS)

                        # Obtener las coordenadas de la muñeca (wrist)
                        wrist_x =
hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].x
                        wrist_y =
hand_landmarks.landmark[mp_hands.HandLandmark.WRIST].y

                        # Ajustar las coordenadas al tamaño de la imagen
                        image_height, image_width, _ = image.shape
                        wrist_x = wrist_x * image_width
                        wrist_y = wrist_y * image_height

                        # Dividir la imagen en cuatro cuadrantes
                        half_width = image_width / 2
                        half_height = image_height / 2

                        if wrist_x < half_width:
                            wrist_x = wrist_x - half_width
                        else:
                            wrist_x = wrist_x - half_width

                        if wrist_y < half_height:
                            wrist_y = half_height - wrist_y
                        else:
                            wrist_y = half_height - wrist_y
```

```
                        is_closed = is_hand_closed(hand_landmarks)

                        # Enviar las coordenadas de la muñeca al servidor
                        Message = f'{(wrist_x / 2.5) * -1},{(wrist_y / 2.5) *
-1},{z},{is_closed}'
                        print(f'Coordenadas de la muñeca: ({wrist_x},
{wrist_y}, {z}, {is_closed})')
                        conn.sendall(Message.encode())

            # Mostrar la imagen con los landmarks de las manos
            cv2.imshow('Detección de Manos', image)

            # Salir del bucle si se presiona 'q'
            if cv2.waitKey(5) & 0xFF == ord('q'):
                break

    # Liberar la cámara y cerrar las ventanas
    webcam.release()
    cv2.destroyAllWindows()

def main():
    # Crear un socket TCP/IP
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as servidor:
        servidor.bind((HOST, PORT))  # Enlazar el socket al puerto
        servidor.listen()  # Escuchar las conexiones entrantes
        print(f"Servidor TCP iniciado. Esperando conexiones en {HOST}:{PORT}")
# Mostrar mensaje de inicio

        conn, addr = servidor.accept()  # Aceptar la conexión entrante
        with conn:
            print(f"Conexión establecida desde {addr}")
            # Inicializar la cámara
            webcam = cv2.VideoCapture(0)
            detect_hands(webcam, conn)

if __name__ == "__main__":
    main()
```

4.2. Sketch de Processing (robotic_arm.pde)

Este sketch se encarga de renderizar el brazo robótico en 3D y de recibir los datos de los gestos desde el cliente Python para actualizar la posición del brazo.

```
//------------------GLOBAL DEFINITIONS AND SETUP------------------//

import processing.net.*;

PShape base, shoulder, upArm, loArm, end, tableShape, cube;
float rotX, rotY;
float armPosX = -60, armPosY = -37, armPosZ = 0;
float alpha, beta, gamma;
float F = 50;
float T = 70;
Client client;
String serverIP = "127.0.0.1";
int serverPort = 65432;
float cubeCoordX, cubeCoordY, cubeCoordZ;
String handClosed = "False";
boolean isCubeGrabbed = false;

void setup() {
  size(800, 800, OPENGL);

  surface.setResizable(true);

  initializeShapes();

  client = new Client(this, serverIP, serverPort);

  // Open a duplicate window
  DuplicateWindow duplicate = new DuplicateWindow();
  PApplet.runSketch(new String[]{"DuplicateWindow"}, duplicate);
}

//------------------MAIN LIFECYCLE FUNCTIONS------------------//

void draw() {
  background(32);
  receiveClientData();

  if (isCubeGrabbed && isArmTouchingCube()) {
    armPosX = -cubeCoordY;
    armPosY = -cubeCoordZ;
```

```
    armPosZ = -cubeCoordX;
  }

  sendPositionData();
  renderScene();
}

void mouseDragged() {
  rotY -= (mouseX - pmouseX) * 0.01;
  rotX -= (mouseY - pmouseY) * 0.01;
}

//--------------------INITIALIZATION FUNCTIONS--------------------//

void initializeShapes() {
  base = loadShape("r5.obj");
  shoulder = loadShape("r1.obj");
  upArm = loadShape("r2.obj");
  loArm = loadShape("r3.obj");
  end = loadShape("r4.obj");
  tableShape = loadShape("table.obj");
  cube = loadShape("cube.obj");

  if (shoulder != null) shoulder.disableStyle();
  if (upArm != null) upArm.disableStyle();
  if (loArm != null) loArm.disableStyle();
}

//--------------------RENDERING FUNCTIONS--------------------//

void renderScene() {
  smooth();
  lights();
  directionalLight(51, 102, 126, -1, 0, 0);
  noStroke();

  translate(width / 2, height / 2);
  rotateX(rotX);
  rotateY(-rotY);
  scale(-4); // Added scaling here for consistency

  renderCube();
  renderTables();
  renderRoboticArm();
}
```

```
void renderTables() {
  pushMatrix();
  scale(0.25);
  shape(tableShape, -250, -200);
  popMatrix();

  pushMatrix();
  scale(0.25);
  shape(tableShape, 250, -200);
  popMatrix();
}

void renderCube() {
  pushMatrix();
  translate(armPosX, armPosY, armPosZ);
  scale(0.75);
  shape(cube);
  popMatrix();
}

void renderRoboticArm() {
  fill(#FFE308);
  translate(0, -40, 0);
  if (base != null) shape(base);

  translate(0, 4, 0);
  rotateY(gamma);
  if (shoulder != null) shape(shoulder);

  translate(0, 25, 0);
  rotateY(PI);
  rotateX(alpha);
  if (upArm != null) shape(upArm);

  translate(0, 0, 50);
  rotateY(PI);
  rotateX(beta);
  if (loArm != null) shape(loArm);

  translate(0, 0, -50);
  rotateY(PI);
  if (end != null) shape(end);
}

//------------------CLIENT COMMUNICATION FUNCTIONS------------------//
```

```
void receiveClientData() {
  String data = client.readString();
  if (data != null) {
    parseClientCoordinates(data);
  }
}

void parseClientCoordinates(String data) {
  // Parse coordinates received from the server
  if (data != null && data.length() > 0) {
    String[] parts = split(data, ',');
    if (parts.length == 4) {
      cubeCoordX = float(trim(parts[0]));
      cubeCoordY = float(trim(parts[1]));
      cubeCoordZ = float(trim(parts[2]));
      handClosed = trim(parts[3]);
      isCubeGrabbed = handClosed.equals("True");
    }
  }
}

void sendPositionData() {
  IK();
  String message = cubeCoordX + "," + cubeCoordY + "," + cubeCoordZ +
"," + handClosed;
  client.write(message);
}

//------------------KINEMATICS AND LOGIC FUNCTIONS------------------//

void IK() {
  float X = cubeCoordX;
  float Y = cubeCoordY;
  float Z = cubeCoordZ;

  float L = sqrt(Y * Y + X * X);
  float dia = sqrt(Z * Z + L * L);

  alpha = PI / 2 - (atan2(L, Z) + acos((T * T - F * F - dia * dia) / (-2
* F * dia)));
  beta = -PI + acos((dia * dia - T * T - F * F) / (-2 * F * T));
  gamma = atan2(Y, X);
}

boolean isArmTouchingCube() {
  float tolerance = 30.0; // Define the tolerance value
```

```
  // Calculate the difference between the coordinates of the arm and the
cube
  float diffX = abs(cubeCoordX + armPosZ);
  float diffY = abs(cubeCoordY + armPosX);
  float diffZ = abs(cubeCoordZ + armPosY);

  // Check if all differences are within the tolerance
  boolean isTouching = (diffX < tolerance) && (diffY < tolerance) &&
(diffZ < tolerance);

  return isTouching;
}

//-------------------DUPLICATE WINDOW CLASS-------------------//

public class DuplicateWindow extends PApplet {
  float rotX, rotY; // Mismo estado de la cámara que en la ventana
principal

  public void settings() {
    size(800, 800, OPENGL);
  }

  public void setup() {
    surface.setResizable(true); // Make the window resizable
    initializeShapes();
  }

  public void draw() {
    background(32);
    renderScene(rotX, rotY);
  }

  public void mouseDragged() {
    rotY -= (mouseX - pmouseX) * 0.01;
    rotX -= (mouseY - pmouseY) * 0.01;
  }

  void renderScene(float rotX, float rotY) {
    smooth();
    lights();
    directionalLight(51, 102, 126, -1, 0, 0);

    noStroke();
```

```
    translate(width / 2, height / 2);
    rotateX(rotX);
    rotateY(-rotY);
    scale(-4);

    pushMatrix();
    scale(0.25);
    shape(tableShape, -250, -200);
    popMatrix();
    pushMatrix();
    scale(0.25);
    shape(tableShape, 250, -200);
    popMatrix();

    pushMatrix();
    translate(armPosX, armPosY, armPosZ);
    scale(0.75);
    shape(cube);
    popMatrix();

    fill(#FFE308);
    translate(0, -40, 0);
    if (base != null) shape(base);

    translate(0, 4, 0);
    rotateY(gamma);
    if (shoulder != null) shape(shoulder);

    translate(0, 25, 0);
    rotateY(PI);
    rotateX(alpha);
    if (upArm != null) shape(upArm);

    translate(0, 0, 50);
    rotateY(PI);
    rotateX(beta);
    if (loArm != null) shape(loArm);

    translate(0, 0, -50);
    rotateY(PI);
    if (end != null) shape(end);
  }
}
```

5. Resultados

5.1. Funcionalidad General

El sistema desarrollado cumple con los objetivos planteados, permitiendo el control de un brazo robótico en un entorno virtual 3D mediante gestos de la mano capturados en tiempo real. Los principales resultados obtenidos son:

Detección de Gestos: El cliente Python utiliza OpenCV y MediaPipe para detectar y clasificar gestos de la mano.

Transmisión de Datos en Tiempo Real: Los datos de los gestos se envían mediante TCP/IP al servidor, donde el sketch de Processing recibe y procesa estos datos para actualizar la posición del brazo robótico.

Visualización en 3D: El sketch de Processing renderiza un brazo robótico en 3D que se mueve en respuesta a los gestos detectados. La interfaz gráfica permite observar los movimientos del brazo desde diferentes ángulos gracias a la capacidad de rotación de la vista.

5.2. Limitaciones

Aunque los resultados generales son positivos, se identificaron algunas limitaciones que podrían ser abordadas en futuras versiones del proyecto:

Físicas de los objetos: No se pudieron incluir las físicas de los objetos, como caídas, lo que limita la interacción realista entre el brazo robótico y el entorno virtual.

Gripper: No se implementó un gripper en el extremo del brazo robótico.

5.3 Video de funcionalidad:

Funcionalidad minireto4 - Creado con Clipchamp_1716779263085.…

5.4 Repositorio de Github:

https://github.com/jpcedano/Visuocontrol-of-a-quasi-static-robot/tree/main

7. Referencias

Guía de detección de puntos de referencia en la mano. (n.d.). Google for Developers. https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker?hl=es-419

OpenCV: OpenCV modules. (n.d.). https://docs.opencv.org/4.x/

socket — Low-level networking interface. (n.d.). Python Documentation. https://docs.python.org/3/library/socket.html