Facial Detection

Orin board users can directly open the terminal and input the tutorial commands to run directly. Jetson-Nano board users need to enter the docker container first, then input the tutorial commands in the docker to start the program.

1. Introduction

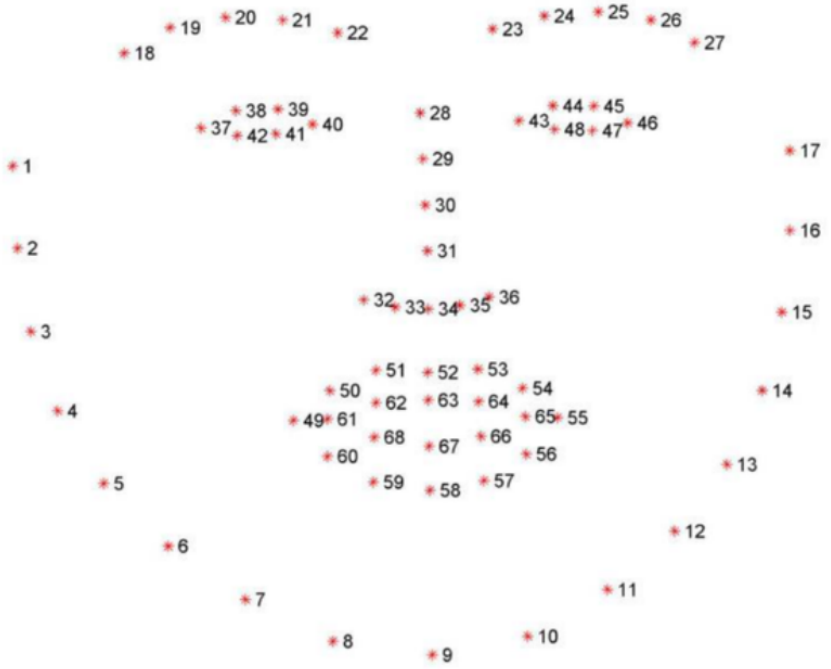
MediaPipe is a data stream processing machine learning application development framework developed and open-sourced by Google. It is a graph-based data processing pipeline used to build applications that use various forms of data sources such as video, audio, sensor data, and any time series data. MediaPipe is cross-platform and can run on embedded platforms (Jetson nano, etc.), mobile devices (iOS and Android), workstations and servers, and supports mobile GPU acceleration. MediaPipe provides cross-platform, customizable ML solutions for real-time and streaming media. The core framework of MediaPipe is implemented in C++ and provides support for languages such as Java and Objective C. The main concepts of MediaPipe include Packet, Stream, Calculator, Graph, and Subgraph.

Features of MediaPipe:

- End-to-end acceleration: Built-in fast ML inference and processing accelerates even on ordinary hardware.
- Build once, deploy anywhere: Unified solution for Android, iOS, desktop/cloud, web and IoT.
- Ready-to-use solutions: Cutting-edge ML solutions that showcase the full capabilities of the framework.
- Free and open source: Framework and solutions under Apache2.0, fully scalable and customizable.

2. Dlib

DLIB is a modern C++ toolkit containing machine learning algorithms and tools for creating complex software in C++ to solve real-world problems. It is widely used in industry and academia in fields such as robotics, embedded devices, mobile phones, and large high-performance computing environments. The dlib library uses 68-point landmarks to mark important facial parts, for example, points 18-22 mark the right eyebrow, and points 51-68 mark the mouth. The dlib library's get_frontal_face_detector module is used to detect faces, and the shape_predictor_68_face_landmarks.dat feature data is used to predict facial feature values.

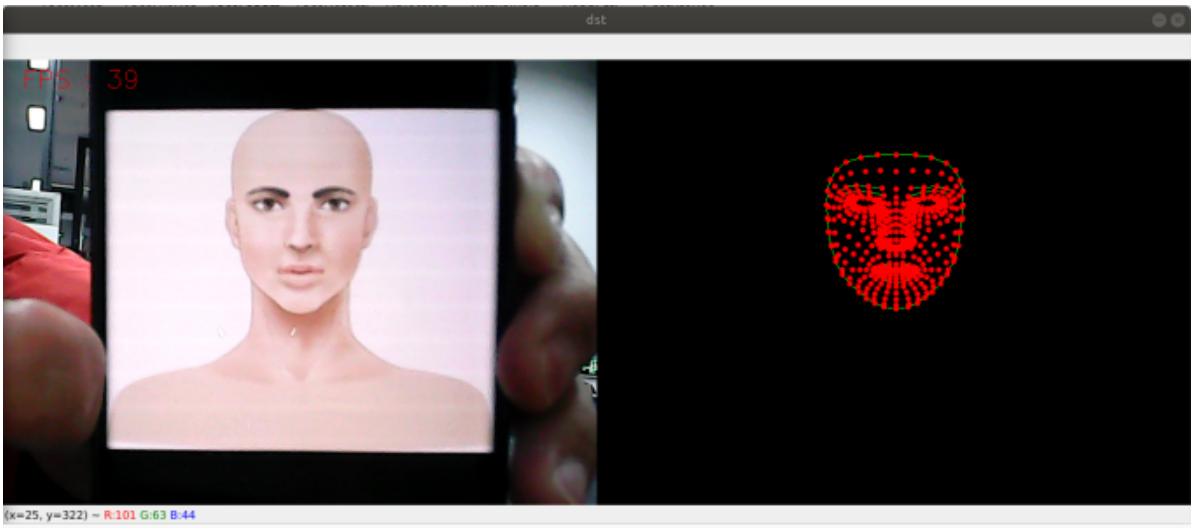


3. Facial Detection

3.1. Launch

- Enter the following command to start the program

```
ros2 run dofbot_pro_mediapipe 04_FaceMesh
```



3.2. Source Code

Source code location:

```
# Jetson-Nano users need to enter the docker container to view
~/dofbot_pro_ws/src/dofbot_pro_mediapipe/dofbot_pro_mediapipe/04_FaceMesh.py
```

```python
#!/usr/bin/env python3
# encoding: utf-8
import time
import rclpy
from rclpy.node import Node
import cv2 as cv
import numpy as np
import mediapipe as mp
from geometry_msgs.msg import Point
from dofbot_pro_msgs.msg import PointArray
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

class FaceMesh(Node):
    def __init__(self, staticMode=False, maxFaces=2, minDetectionCon=0.5,
minTrackingCon=0.5):
        super().__init__('face_mesh_detector')
        self.publisher_ = self.create_publisher(PointArray, '/mediapipe/points',
10)
        self.bridge = CvBridge()

        self.mpDraw = mp.solutions.drawing_utils
        self.mpFaceMesh = mp.solutions.face_mesh
        self.faceMesh = self.mpFaceMesh.FaceMesh(
            static_image_mode=staticMode,
            max_num_faces=maxFaces,
            min_detection_confidence=minDetectionCon,
            min_tracking_confidence=minTrackingCon )

        self.lmDrawSpec = mp.solutions.drawing_utils.DrawingSpec(color=(0, 0,
255), thickness=-1, circle_radius=3)
        self.drawSpec = self.mpDraw.DrawingSpec(color=(0, 255, 0), thickness=1,
circle_radius=1)

        self.capture = cv.VideoCapture(0, cv.CAP_V4L2)
        self.capture.set(6, cv.VideoWriter.fourcc('M', 'J', 'P', 'G'))
        self.capture.set(cv.CAP_PROP_FRAME_WIDTH, 640)
        self.capture.set(cv.CAP_PROP_FRAME_HEIGHT, 480)

        if not self.capture.isOpened():
            self.get_logger().error("Failed to open the camera")
            return

        self.get_logger().info(f"Camera FPS:
{self.capture.get(cv.CAP_PROP_FPS)}")
        self.pTime = time.time()

        self.timer = self.create_timer(0.03, self.process_frame)

    def process_frame(self):
        ret, frame = self.capture.read()
        if not ret:
            self.get_logger().error("Failed to read frame")
            return

        frame, img = self.pubFaceMeshPoint(frame, draw=True)

        cTime = time.time()
```

```
        fps = 1 / (cTime - self.pTime)
        self.pTime = cTime
        text = "FPS : " + str(int(fps))
        cv.putText(frame, text, (20, 30), cv.FONT_HERSHEY_SIMPLEX, 0.9, (0, 0,
255), 1)

        combined_frame = self.frame_combine(frame, img)
        cv.imshow('FaceMeshDetector', combined_frame)

        if cv.waitKey(1) & 0xFF == ord('q'):
            self.get_logger().info("Exiting program")
            self.capture.release()
            cv.destroyAllWindows()
            self.destroy_node()
            rclpy.shutdown()
            exit(0)

    def pubFaceMeshPoint(self, frame, draw=True):
        pointArray = PointArray()
        img = np.zeros(frame.shape, np.uint8)
        imgRGB = cv.cvtColor(frame, cv.COLOR_BGR2RGB)
        self.results = self.faceMesh.process(imgRGB)
        if self.results.multi_face_landmarks:
            for i in range(len(self.results.multi_face_landmarks)):
                try:
                    if draw: self.mpDraw.draw_landmarks(frame,
self.results.multi_face_landmarks[i], self.mpFaceMesh.FACE_CONNECTIONS,
self.lmDrawSpec, self.drawSpec)
                    self.mpDraw.draw_landmarks(img,
self.results.multi_face_landmarks[i], self.mpFaceMesh.FACE_CONNECTIONS,
self.lmDrawSpec, self.drawSpec)
                except:
                    if draw: self.mpDraw.draw_landmarks(frame,
self.results.multi_face_landmarks[i], self.mpFaceMesh.FACEMESH_CONTOURS,
self.lmDrawSpec, self.drawSpec)
                    self.mpDraw.draw_landmarks(img,
self.results.multi_face_landmarks[i], self.mpFaceMesh.FACEMESH_CONTOURS,
self.lmDrawSpec, self.drawSpec)

                for id, lm in
enumerate(self.results.multi_face_landmarks[i].landmark):
                    point = Point()
                    point.x, point.y, point.z = lm.x, lm.y, lm.z
                    pointArray.points.append(point)
        self.publisher_.publish(pointArray)
        return frame, img

    def frame_combine(self, frame, src):
        if len(frame.shape) == 3:
            frameH, frameW = frame.shape[:2]
            srcH, srcW = src.shape[:2]
            dst = np.zeros((max(frameH, srcH), frameW + srcW, 3), np.uint8)
            dst[:, :frameW] = frame[:, :]
            dst[:, frameW:] = src[:, :]
        else:
            src = cv.cvtColor(src, cv.COLOR_BGR2GRAY)
            frameH, frameW = frame.shape[:2]
            imgH, imgW = src.shape[:2]
```

```
            dst = np.zeros((frameH, frameW + imgW), np.uint8)
            dst[:, :frameW] = frame[:, :]
            dst[:, frameW:] = src[:, :]
        return dst


def main(args=None):
    rclpy.init(args=args)
    node = FaceMesh()
    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        node.capture.release()
        cv.destroyAllWindows()
        node.destroy_node()
        rclpy.shutdown()

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