Human portrait drawing using robot arm

Wang Yexiang Contribution: Control part

Yue Yiyao Contribution: Image processing, path planning

Shi Yongqi Contribution: Image processing, path planning

Fan Hangrui Contribution: Image processing

***Abstract*—In this experiment, we use Intel's RealSense camera to capture face photos, and make a real-time image processing algorithm system. The face photos captured by the camera will be transmitted to the image processing system and converted into stick figure images, and then use the depth-first search algorithm to process the data. The data was transmitted to the PD control algorithm program that controlled the robotic arm and the spraying, so that the robotic arm could draw a stick figure of the face consistent with the taken picture in a few minutes with the best path.**

Keywords—Canny edge detection operator, Image processing, Real-time image capture, Depth-first search algorithm, PD control, Mechanical arm.

I. INTRODUCTION

Based on the study of manipulator control, our group designed a very creative small experiment, which is very interesting and covers a wide range of knowledge. Not only can we learn a lot of useful knowledge from it, but also can use the results of this experiment for the development of robot automation, making a certain contribution to the robot and automation industry.

In this experiment, we will use the real-time image capture method to collect the face image, and then process the collected image to generate a completely consistent stick figure image, and then control the mechanical arm to complete the fast and accurate painting work.

The technical points of the experiment include real-time image capture, image transmission, image processing algorithm system, stroke path planning system, real-time data transmission system, robot control system, process automation and robot work efficiency improvement algorithm. The results of the experiment have certain help and development prospects in the fields of robot control algorithm, image processing algorithm, automation and so on.

II. METHODOLOGY

A. Camara capture

We use a camera to capture photos of faces and transmit them in real-time to the system for facial image processing algorithms. The camara that we use is the Intel RealSense. This depth camera is a camera device that can obtain real-time 3D depth information. This camera uses Structured light, time of flight, ToF or parallax and other technologies to obtain the distance information between the subject and the camera in real time. Compared to traditional color cameras, depth cameras can simultaneously obtain the color and depth of pixels with spatial position information, thereby generating corresponding 3D point cloud data. However, in this project, we only used the camera's photography function and did not explore depth. In short, using a camera for real-time capture can provide better interaction with users and make the project more complete.

B. Image processing

After a photo is captured by the camera, it needs to be transformed to a gray scale image. And then use interpolation to change the size of the image, if it is too big or too small. After this, a Gaussian blurring filter is applied in order to reduce the noise. Finally the blurred image is processed by a Canny edge detection operator with appropriate parameters to generate a figure of edges. Canny operator is a classic edge detect algorithm. It computes the gradient of the image, and suppresses the non-maximum values along the direction of the gradient to generate thin edges. And then it uses double thresholds to detect and connect the edges. In the result of the edge detection, the pixels on the edges has an intensity of 255, corresponding to white color.

The following 5 figures are the original image, the gray scale image, the interpolated image, the Gaussian blurred image, and the edge image.

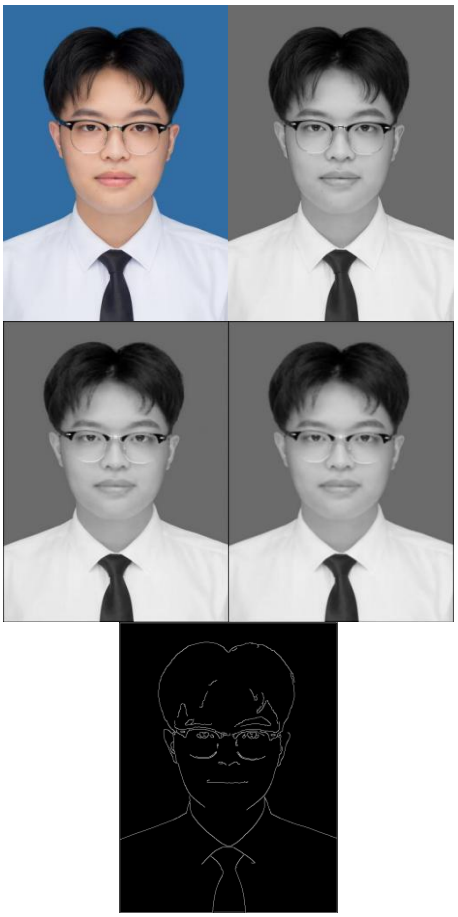


Fig. 1. Original image, gray scale image, interpolated image, Gaussian blurred image, edge image (From up tp down)

C. Pixel classification

In the process of the path planning, it is required to find the start points of all edges, so that the robot can start form the start points to draw continuous lines. The start points are defined as pixels whose 8 neighbors contain 1 or 2 continuous edge pixels as shown below.

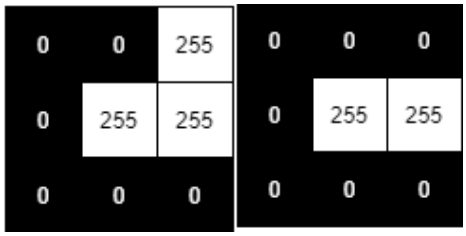


Fig. 2. Start point

To classify a pixel on the edge into isolated point, the start point, and the normal point, the concept of finite state machine (FSM) is used in the algorithm. The 8 neighbors around the center edge pixel will be sent into the FSM in order, and the machine will operate according to the current state and input pixel intensity. The FSM is shown in the following figure.

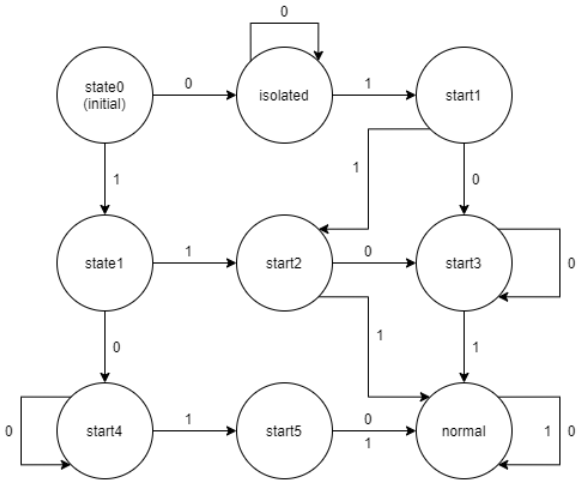


Fig. 3. FSM chart

The circular neighbors are expressed by an array which is not circular. So problems may exist when 2 adjacent neighbor edge pixels are placed at the start and the end of the array respectively. To solve this problem, whether the first pixel is on edge leads to 2 different cases. If the first pixel is not on edge, the problem mentioned above will not exist. In this case, if all the neighbors are not on edge, the center pixel is isolated. If the FSM detects 1 or 2 continuous edge pixels, the center pixel should be a start point, as long as there is no other edge pixels after this. Otherwise the center pixel will be a normal one. In the case where the first pixel is an edge pixel, the above problem should be considered. If the second one is also an edge pixel, the center pixel is classified into a start point as long as none of the remained pixels is an edge pixel. If the second pixel is not an edge pixel, the center pixel will be regarded as a start point if there is no remained edge pixel or there is only one remained one which is the last pixel in the array. Otherwise it will be considered as a normal point.
When the 8 neighbors are all searched, the finite state machine gives the result of the classification according to the current state. The point will be isolated point, which will be neglected, if the state is “isolated”, normal point if the state is “normal”, and start point if the state is one of the “start” state. An test input and output of classification is shown in the figure below. Here the start points are represented as gray pixels, and normal points are white pixels. And the algorithm is obviously effective.

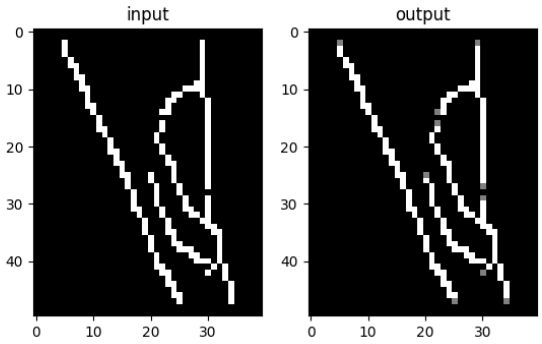


Fig. 4. Test figure

D. Path palnning using first depth search

After the processing and classification of the facial images captured by the camera are completed, we need to generate the strokes of the image, which tells the robot the coordinate values of each stroke of the image. If path planning is not carried out, robot painting involves traversing pixel points line by line starting from the top left corner of the image. Although painting can still be completed, this goes against the original intention of robot painting and is very slow. This project aims to make robots paint as much as possible like humans, and humans generate a drawing path in their minds during the process of painting. Humans will efficiently find a complete stroke that allows the pen to move continuously on paper without lifting it up. So we also need to plan the path of the processed facial image and generate the path for robot painting. Here, we use the Depth-first search algorithm.

The Depth-first search algorithm is a common graph traversal algorithm, which accesses every possible branch until it finds the target node or the maximum depth. Different from the Breadth-first search algorithm, the Depth-first search algorithm will search each branch as deeply as possible, which is usually more suitable for solving problems with far goals. The main idea of this algorithm is to start from the starting node and continue to access the remaining child nodes along a path until no further progress can be made. Then, it returns to the previous node and continues to access the remaining child nodes.

In concrete implementation, the Depth-first search algorithm is usually implemented by recursion or stack. Through recursive calls or stack operations, the algorithm always keeps track of the current node while recording nodes that have been visited to avoid duplicate searches.

Steps for searching and traversing the graph of the algorithm

(1) First, locate the initial node A,

(2) Starting from the adjacent points that have not been accessed by A, perform depth first traversal of the graph

(3) If a node has not been accessed, trace back to that node and continue with depth first traversal

(4) Until all nodes that connect with the vertex A path have been accessed once

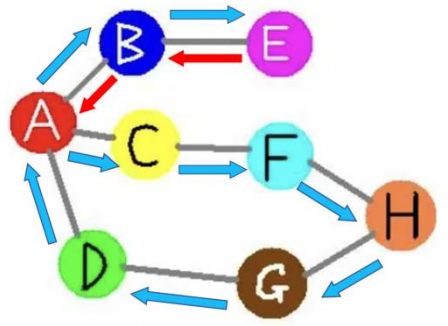


Fig. 5. Depth first search example

As shown in the figure, the depth first search will generate the path: ABE, ACFHDG. This algorithm helps generate painting paths in the image. The image to be processed by Depth-first search is shown in the following figure.

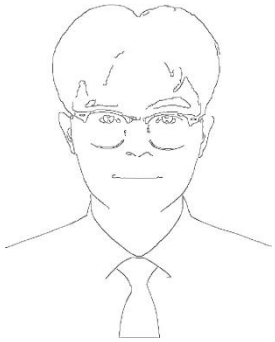


Fig. 6. After depth first search

Due to the precision of each stroke in the image, we sample the coordinate points in each stroke every five times during actual processing. While ensuring that facial features are not changed, we try to reduce the number of drawing points as much as possible to improve the speed of the robot. The graph after sampling is as follows.

Fig. 7. After sampling the points

Afterwards, we will proceed with the painting control part of the robot.

E. Control part

PD control, also known as Proportional-Derivative control, is a feedback control strategy commonly used in control systems to regulate the behavior of a system. It is particularly effective in controlling dynamic systems where stability and responsiveness are important. The PD control algorithm utilizes two components: proportional control and derivative control.

The proportional control component generates an output signal that is proportional to the error between the desired setpoint and the current state of the system. The error is calculated by subtracting the setpoint from the measured value. The proportional gain, denoted as Kp, determines the magnitude of the control signal generated. The proportional control component helps in reducing the steady-state error by directly influencing the control signal based on the current error. However, it may not be sufficient for addressing the transient response and stability issues.

The derivative control component takes into account the rate of change of the error. It calculates the derivative of the error signal with respect to time. The derivative gain, denoted as Kd, is multiplied by the derivative of the error to produce the derivative control signal. The derivative control component provides an anticipatory action by considering the trend of the error. It helps in improving the system's response time and damping oscillations. By increasing the derivative gain, the control system can react more strongly to changes in the error rate.

Overall, the PD control combines the proportional and derivative control signals to generate the final control signal applied to the system. The PD control signal is the sum of the proportional and derivative control signals:

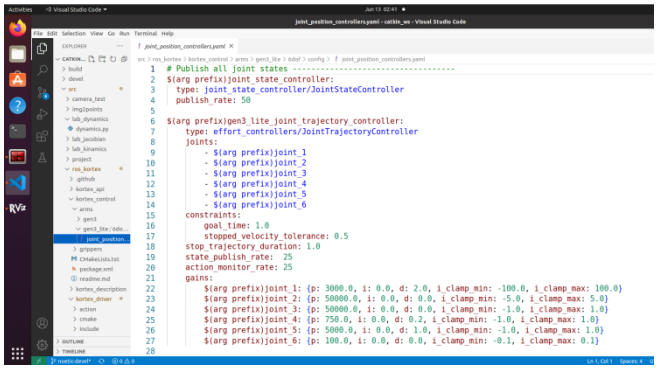
PD Control Signal = Proportional Control Signal + Derivative Control Signal

$$u(t) = K_p\, e(t) + T_d\, de(t)/dt$$

By adjusting the gains (Kp and Kd), the controller's performance can be tuned to achieve the desired response. The proportional gain determines the steady-state error, while the derivative gain influences the system's transient response and stability. It is important to strike a balance between these gains to ensure stable and optimal control.

PD control is widely used in various applications, such as robotics, process control, autonomous systems, and motion control, where precise and responsive control is required.

In this project, we can check the Kp and Kd in the yaml file as shown below:



```
# Publish all joint states -----------------------------------
${arg prefix}joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50

${arg prefix}gen3_lite_joint_trajectory_controller:
    type: effort_controllers/JointTrajectoryController
    joints:
      - ${arg prefix}joint_1
      - ${arg prefix}joint_2
      - ${arg prefix}joint_3
      - ${arg prefix}joint_4
      - ${arg prefix}joint_5
      - ${arg prefix}joint_6
    constraints:
      goal_time: 1.0
      stopped_velocity_tolerance: 0.5
    stop_trajectory_duration: 1.0
    state_publish_rate:  25
    action_monitor_rate: 25
    gains:
      ${arg prefix}joint_1: {p: 3000.0, i: 0.0, d: 2.0, i_clamp_min: -100.0, i_clamp_max: 100.0}
      ${arg prefix}joint_2: {p: 50000.0, i: 0.0, d: 0.0, i_clamp_min: -5.0, i_clamp_max: 5.0}
      ${arg prefix}joint_3: {p: 50000.0, i: 0.0, d: 0.0, i_clamp_min: -1.0, i_clamp_max: 1.0}
      ${arg prefix}joint_4: {p: 750.0, i: 0.0, d: 0.2, i_clamp_min: -1.0, i_clamp_max: 1.0}
      ${arg prefix}joint_5: {p: 5000.0, i: 0.0, d: 1.0, i_clamp_min: -1.0, i_clamp_max: 1.0}
      ${arg prefix}joint_6: {p: 100.0, i: 0.0, d: 0.0, i_clamp_min: -0.1, i_clamp_max: 0.1}
```

Fig. 8. Yaml file

PD control have many advantages, including improved stability, faster response, reduced steady-state error, flexibility and adaptability.

```
if config.output.model == ModelId.MODEL_ID_L31:
    h = 0.0665

    for i in range(len(picture)):
        goal.trajectory.append(self.FillCartesianWaypoint(picture[i][0][0],  picture[i][0][1],  h+0.05,
                                                          math.radians(90), 0, math.radians(90), 0))
        for j in range(len(picture[i])):
            goal.trajectory.append(self.FillCartesianWaypoint(picture[i][j][0],  picture[i][j][1],  h,
                                                              math.radians(90), 0, math.radians(90), 0))
        goal.trajectory.append(self.FillCartesianWaypoint(picture[i][-1][0],  picture[i][-1][1],  h+0.05,
                                                          math.radians(90), 0, math.radians(90), 0))
```

Fig. 9. Core code

This code snippet is a loop that generates a trajectory by appending waypoints to a goal.trajectory list.

Firstly, The code begins by initializing a variable h and assigning it a value of 0.0665. This variable represents a height value or another parameter related to the trajectory.

Next, a for loop iterates over the range of the picture list. This loop iterates through each element of the list.

Inside the loop, the self.FillCartesianWaypoint function is called to generate a waypoint. This function takes multiple parameters, including x, y, z, roll, pitch, and yaw, which define the robot's position and orientation in 3D space.In the first iteration, the function is called with picture[i][0][0], picture[i][0][1], h+0.05, and other parameters unchanged. This increases the value of z by 0.05 compared to the previous waypoint.

Then, within a nested for loop, the code iterates through each element of picture[i]. For each element, the function is called with picture[i][j][0], picture[i][j][1], h, and other parameters unchanged. This keeps the value of z the same as the previous waypoint.At the end of the inner for loop, the function is called again with picture[i][-1][0], picture[i][-1][1], h+0.05, and other parameters unchanged. This raises the height slightly at the end of the trajectory.Finally, the generated waypoints are appended to the goal.trajectory list.

In summary, this code generates a trajectory based on the data points in the picture list and appends the waypoints to the goal.trajectory list. The self.FillCartesianWaypoint function is used to define the robot's position and orientation, with the z coordinate potentially varying throughout the loop.

III. Result

The project ultimately generated a portrait of a character drawn by a robot, which shows that the drawing results are quite realistic. Within 5 minutes, the robot completed the painting, and the painting speed is guaranteed. The painting results can clearly distinguish the identity and details of the characters.

Fig. 10. Drawing result

IV. Conclusion

This project has completed the portrait drawing based on a robotic arm. We start from collecting face images, first conduct image processing, then classify pixels, and then complete path planning through Depth-first search. Finally, a control algorithm was applied to the robotic arm to complete this painting. This project is an interesting attempt at robot painting, and more functions will be improved in the future.

References

[1] S. Calinon, J. Epiney and A. Billard, "A humanoid robot d rawing human portraits," 5th IEEERAS International Confe rence on Humanoid Robots, 2005., Tsukuba, Japan, 2005, p p. 161-166, doi: 10.1109/ICHR.2005.1573562.

[2] Chyi-Yeu Lin, and Chien-Wei Li, " Human Portrait Generation System for Robot Arm Drawing, " The 13th Conference on Artificial Intelligence and Applications. Taipei, Taiwan, 2008.

[3] Xiaoyang Mao, Yoshiyasu Nagasaka and Atsumi Imamiya, "Automatic Generation of Pencil Drawing From 2D Images Using Line Integral Convolution," CAD/Graphics , 2001.

[4] Shigefumi Yamamoto, Xiaoyang Mao and Atsumi Imamiya, " Enhanced LIC Pencil Filter, " IEEE Computer Graphics, Imaging and Visualization Proceedings, 2004.

[5] M. Pichkalev, R. Lavrenov, R. Safin and K. - H. Hsia, "Face Drawing by KUKA 6 Axis Robot Manipulat or," 2019 12th International Conference on Developments i n eSystems Engineering (DeSE), Kazan, Russia, 2019, pp. 7 09-714, doi: 10.1109/DeSE.2019.00132